

Overview of the Course

Final Exam

The exam will consist in

1. a **project** developed partly during the hours of thursdays
2. a **seminar** on a chosen topic between a list
3. an **oral** discussion mainly on the project

Please, actively partecipate to lectures

Compilers

- What is a compiler?
 - A program that takes other programs and prepare them for execution

In particular, a program that takes a program and translate it in program written in a target language

- The target language is in general the instruction set of an architecture
- The target language can be a human-oriented programming language (Source-to-source translators)

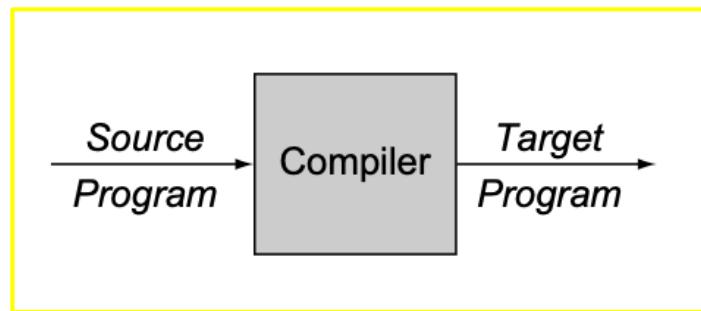
AOT vs JIT

- Most compiler are designed to run in a separate step before execution (ahead-of-time AOT)
- Some new compilers translate the code to executable form at runtime (just-in-time JIT)
the cost of the translation has to be summed to the one of the execution

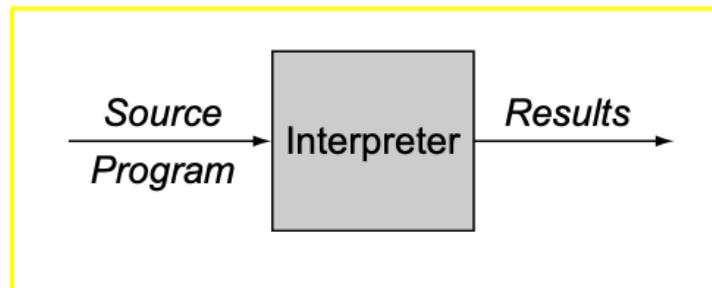
COMPILED / INTERPRETED IS A PROPERTY OF THE IMPLEMENTATION OF A LANGUAGE

Compilers vs Interpreters

- A **compiler** is a program that takes a **program** and translates it in a **program** written in an another language



- What is an **interpreter**?
 - A program that reads a **program** and an **input** and produces the **results** of executing that **program** on the **input**



Compilers vs Interpreters

Compiler scans the whole program in one go.	Translates program one statement at a time.
It converts the source code into object code.	It does not convert source code into object code instead it scans it line by line
The translation is performed before executing	The translation and execution is performed at the same time
Good execution time.	Slow in executing the object code.
It does not require source code for later execution.	It requires source code for later execution.
The errors are shown at the end together.	Errors are shown line by line.

Performance: reducing the price of language abstraction

Computer Science is the art of creating virtual objects and making them useful.

- We invent abstractions and uses for them
- Programming is the way we realize these inventions

Well written compilers make such abstraction affordable

- Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
- Change in expression should bring small performance change
- Cannot expect compiler to devise better algorithms
 - Don't expect bubblesort to become quicksort

Simple Examples

Which is faster?

```
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        A[i][j] = 0;
```

All three loops have distinct performance.

0.51 sec on 10,000 × 10,000 array

```
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        A[j][i] = 0;
```

1.65 sec on 10,000 × 10,000 array

```
p = &A[0][0];  
t = n * n;  
for (i=0; i<t; i++)  
    *p++ = 0;
```

0.11 sec on 10,000 × 10,000 array

A good compiler should know these tradeoffs,
on each target, and generate the best code.

Few real compilers do.

Conventional wisdom suggests using

```
bzero((void*) &A[0][0],(size_t) n*n*sizeof(int))
```

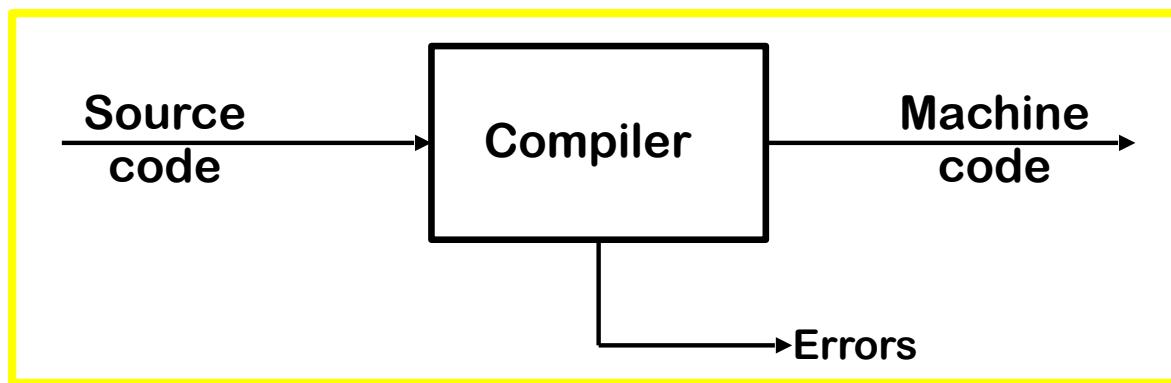
0.52 sec on 10,000 × 10,000 array

Fundamental Principles of Compilation

- The compiler must preserve the meaning of the program being compiled
- The compiler must improve the input program

The View from 35,000 Feet

High-level View of a Compiler

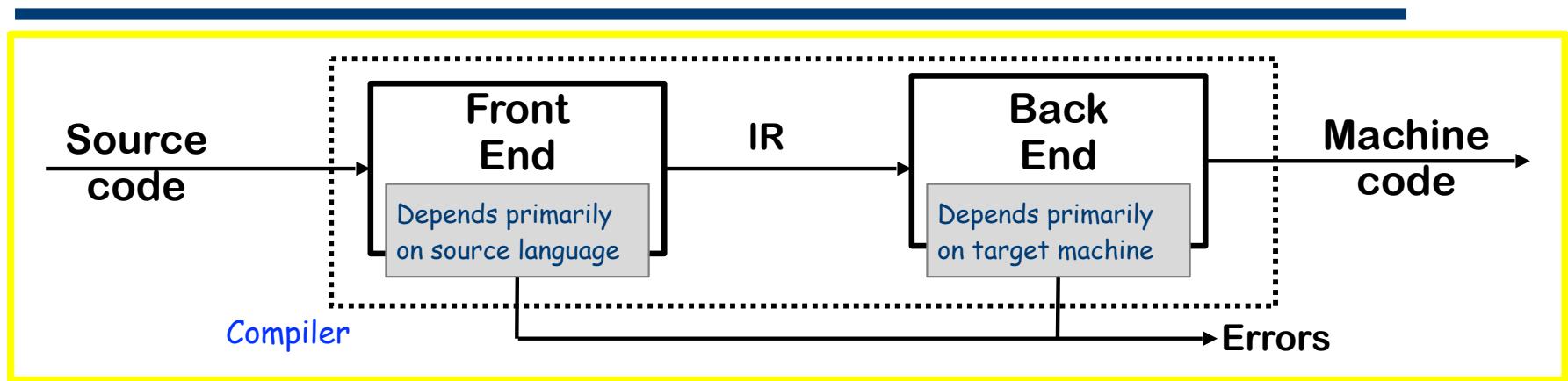


Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language

Traditional Two-pass Compiler

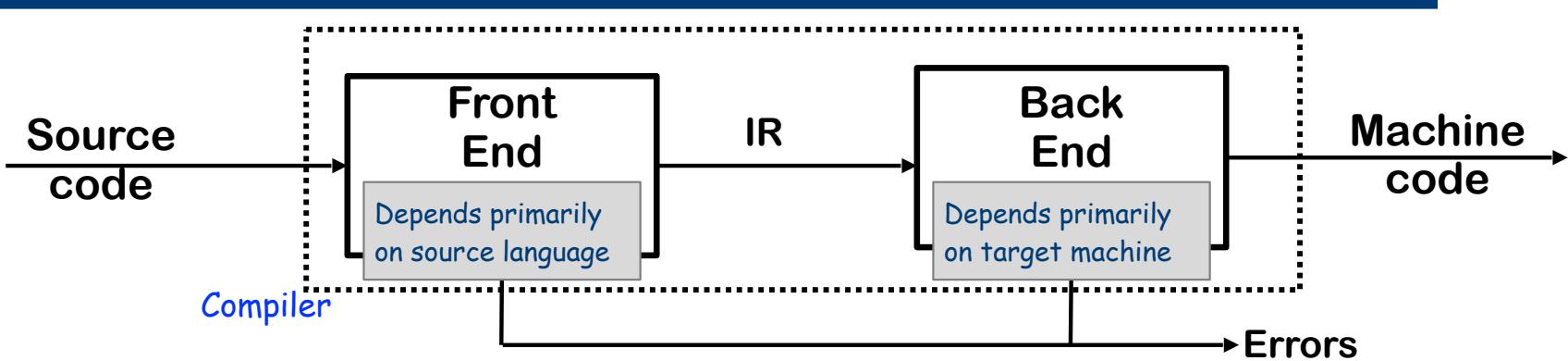


Implications of the division:

- Use an intermediate representation (IR)
- Front end maps **legal source code** into IR
- Back end maps IR into **target machine** code
- Admits multiple passes (better code)

Front end is $O(n)$ or $O(n \log n)$ Back end is NP-Complete

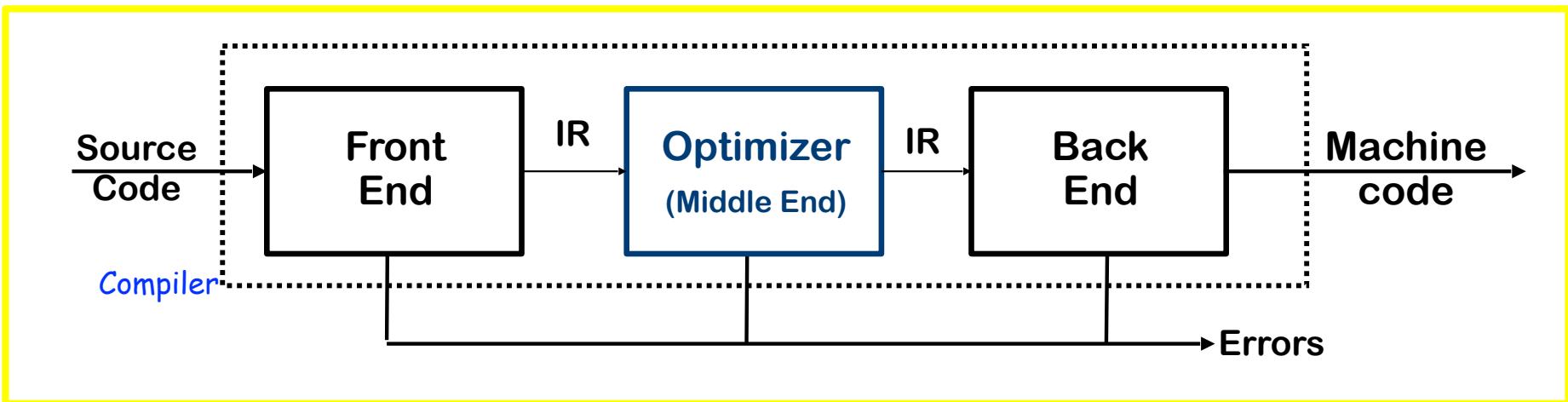
Advantages of two-pass compiler



Classic principle from
software engineering:
Separation of concerns

- The same architecture can target a different machine code
SWAP THE BACKEND, THE IR IS STILL GOOD
- The same architecture can target a source code
SWAP THE FRONTEND, //
- The IR has to encode all the knowledge that the compiler has on the program

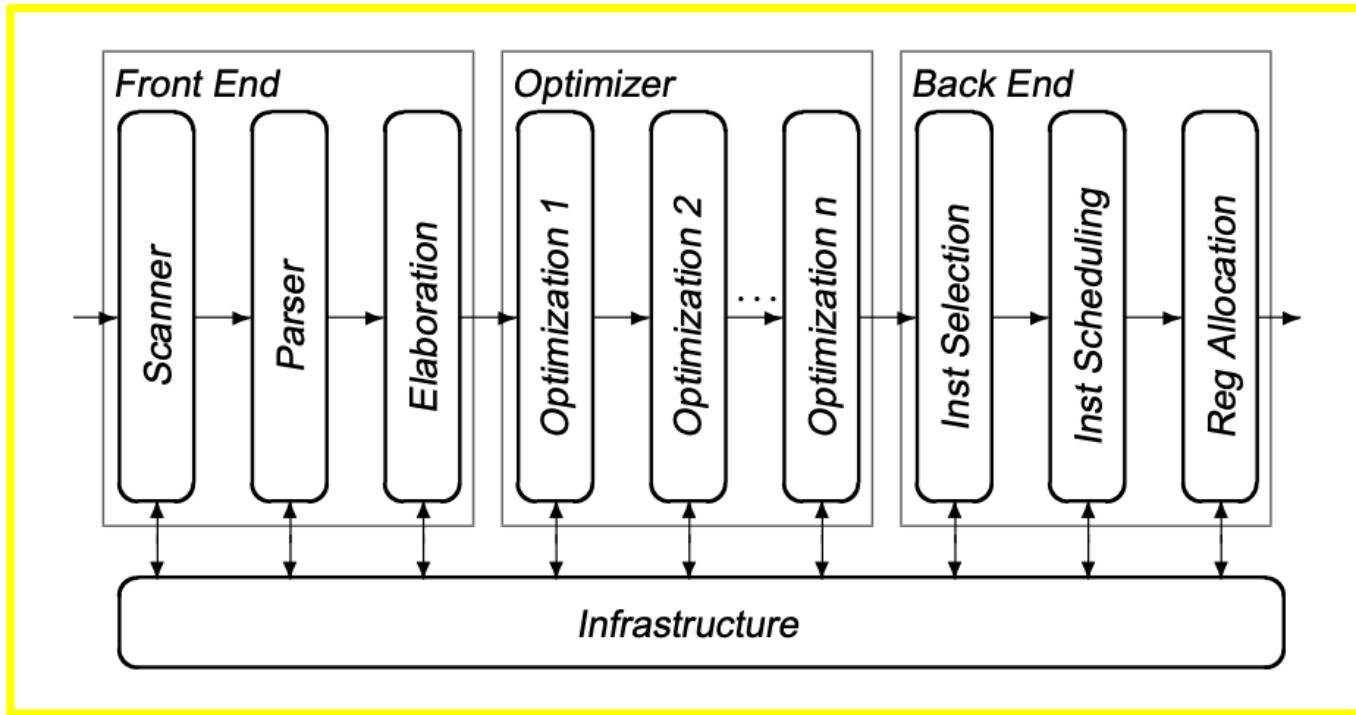
Traditional three-part Compiler



Code Improvement (or Optimization)

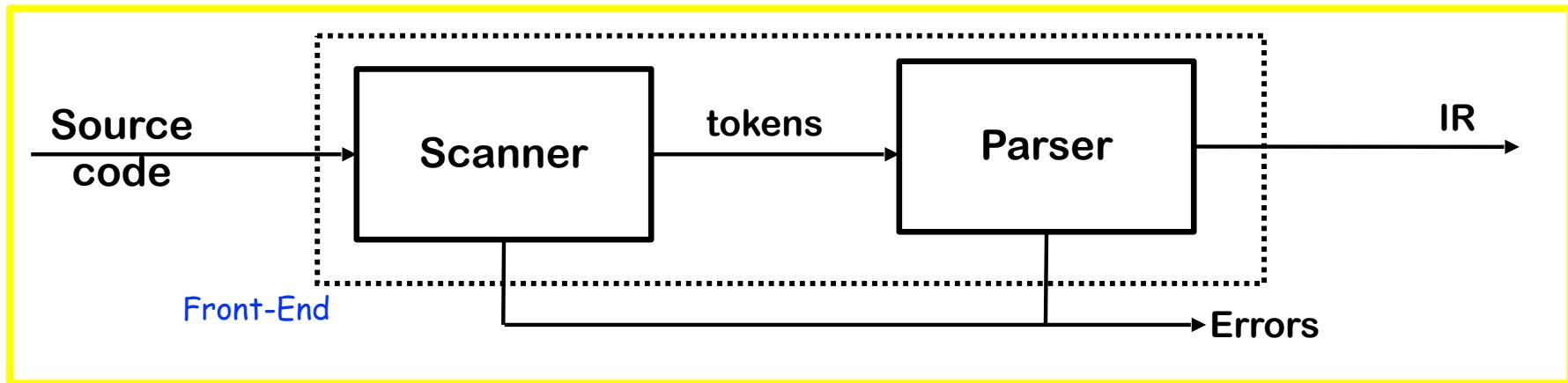
- analyzes IR and transforms IR
- primary goal is to reduce running time of the compiled code
 - and/or reduce code space, power consumption, page faults.....
- Must preserve “meaning” of the code

The phases of a Compiler



- In the actual architecture each phase is divided into a series of passes
- The optimiser contains passes that use distinct analyses and transformation to improve the code

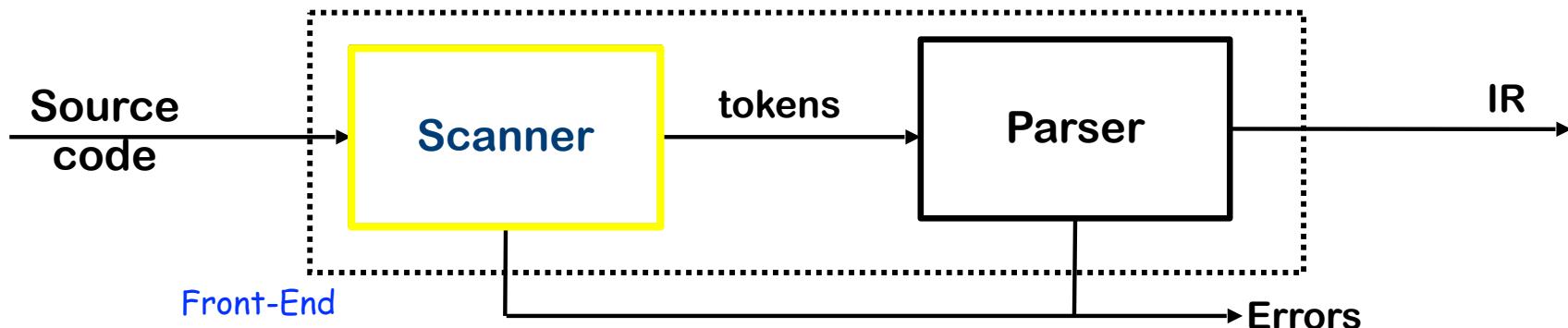
The Front End



Responsibilities

- Recognize legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the rest of the compiler
- Much of Front-End construction can be automated

The Front End



Scanner

- Maps stream characters into stream of words (**Lexical analysis**)
- Produces pairs – a word & its part of speech
 $x = x + y ;$ becomes $\langle id, x \rangle = \langle id, x \rangle + \langle id, y \rangle ;$
- Typical words include numbers, identifier, +, -, new, while, if
- Speed is important

Textbooks advocate automatic scanner generation

Commercial practice appears to be hand-coded scanners

Lexical analysis

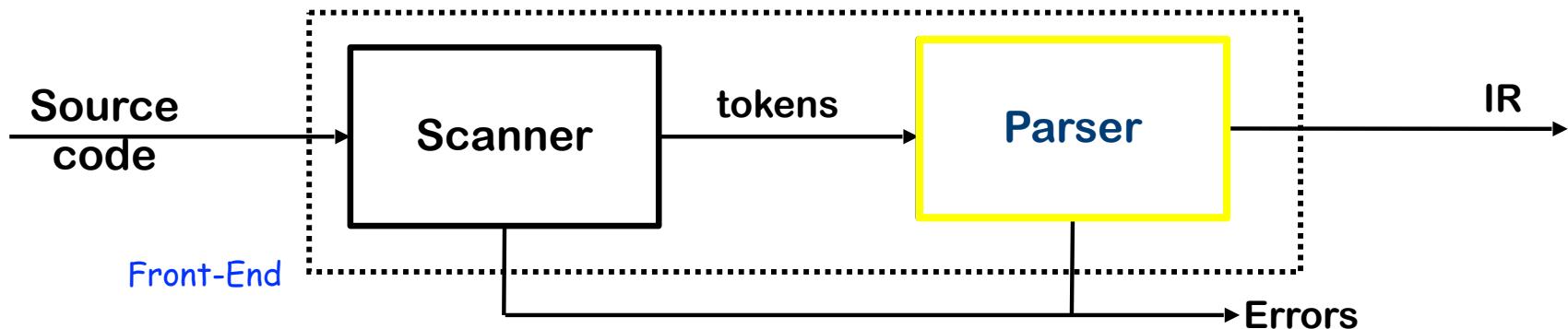
Split program into individual words that makes sense:

1g2h3i is neither a valid identifier nor a valid number

```
while (y < z) {  
    int x = a + b;  
    y += x; }
```

T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace

The Front End



Parser

- Check the syntax & reports errors (**Syntax Analysis**)
- It determines if the stream of words is a sentence in the source language
- Builds IR for source program

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

Grammars for the Front-End

To recognise words and sentences of the source language, the Front-End uses grammars like

$$\begin{aligned}\text{SheepNoise} \rightarrow & \text{ SheepNoise } \underline{\text{baa}} \\ & | \quad \underline{\text{baa}}\end{aligned}$$

It defines the set of noises that a sheep makes under normal circumstances

It is written in a variant of Backus-Naur Form (BNF)

A grammar $G = (S, N, T, P)$

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of productions or rewriting rules $(P : N \rightarrow N \cup T)$

Grammars for simple expressions

$S = Goal$

$T = \{ \underline{\text{number}}, \underline{\text{id}}, +, - \}$

$N = \{ Goal, Expr, Term, Op \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr \; Op \; Term$
3. / $Term$
4. $Term \rightarrow \underline{\text{number}}$
5. / $\underline{\text{id}}$
6. $Op \rightarrow +$
7. / $-$

- It defines simple expressions with $+$ & $-$ over number and id
- This grammar falls in a class called “context-free grammars”, abbreviated CFG

The Front End

Given a CFG, we can derive sentences by repeated substitution

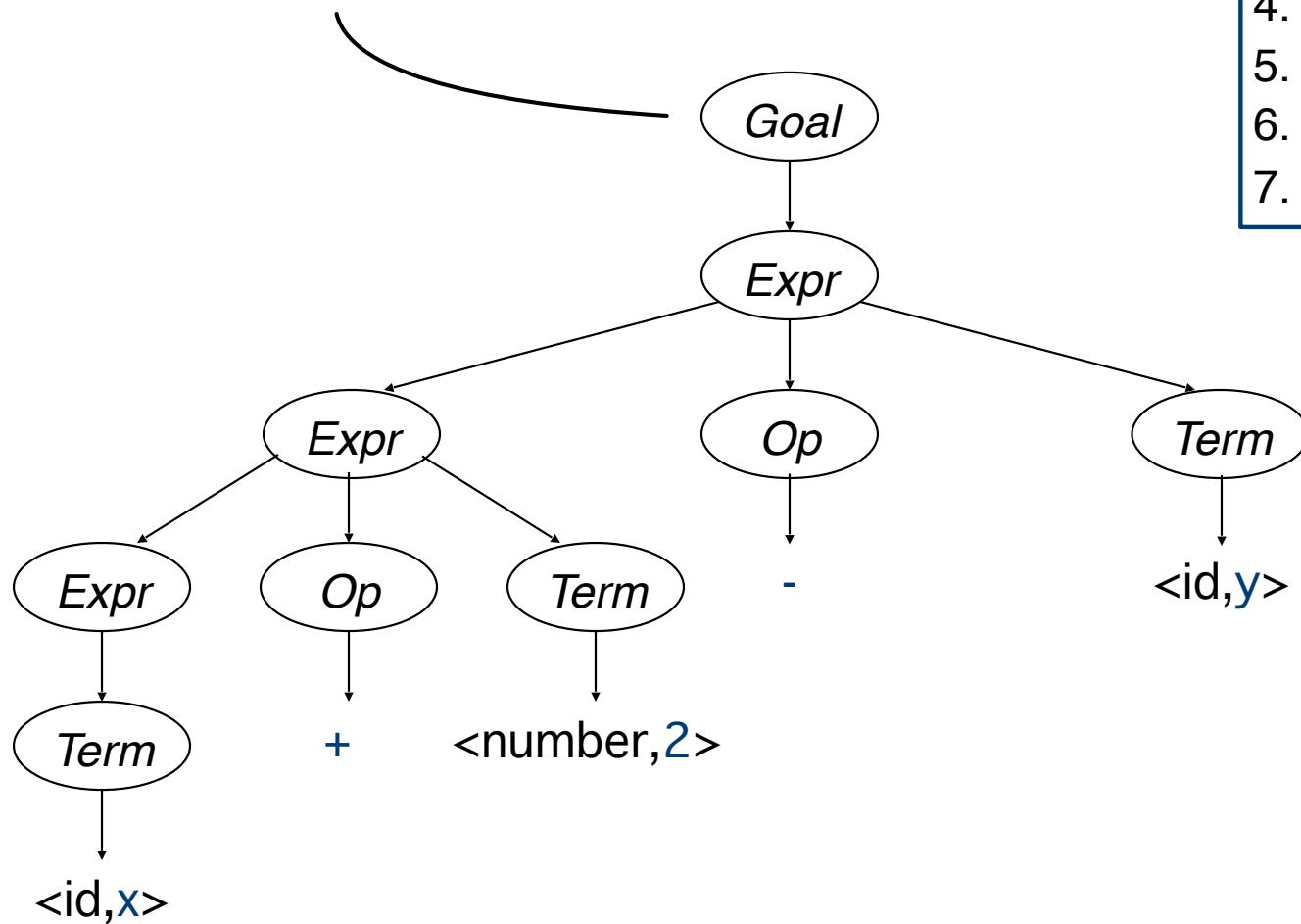
<u>Production</u>	<u>Result</u>
	<i>Goal</i>
1	<i>Expr</i>
2	<i>Expr Op Term</i>
5	<i>Expr Op y</i>
7	<i>Expr - y</i>
2	<i>Expr Op term - y</i>
4	<i>Expr Op 2 - y</i>
6	<i>Expr + 2 - y</i>
3	<i>Term + 2 - y</i>
5	<i>x + 2 - y</i>

A derivation

To recognize a valid sentence, we reverse this process and start from $x+2-y$

The Front End

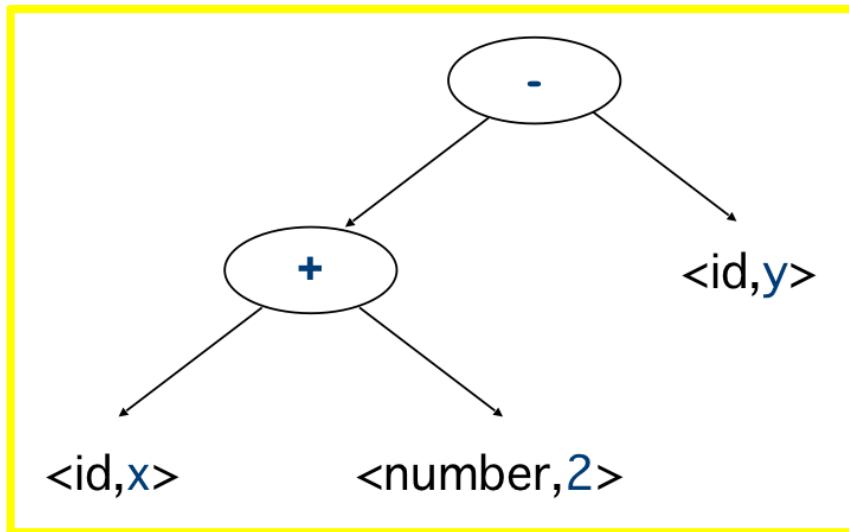
To recognise if $x + 2 - y$ belongs to the language we construct the parsing tree (or syntax tree)



1. $\text{Goal} \rightarrow \text{Expr}$
2. $\text{Expr} \rightarrow \text{Expr } \text{Op } \text{Term}$
3. | Term
4. $\text{Term} \rightarrow \text{number}$
5. | id
6. $\text{Op} \rightarrow +$
7. | $-$

The Front End

Compilers often use an abstract syntax tree instead of a parse tree



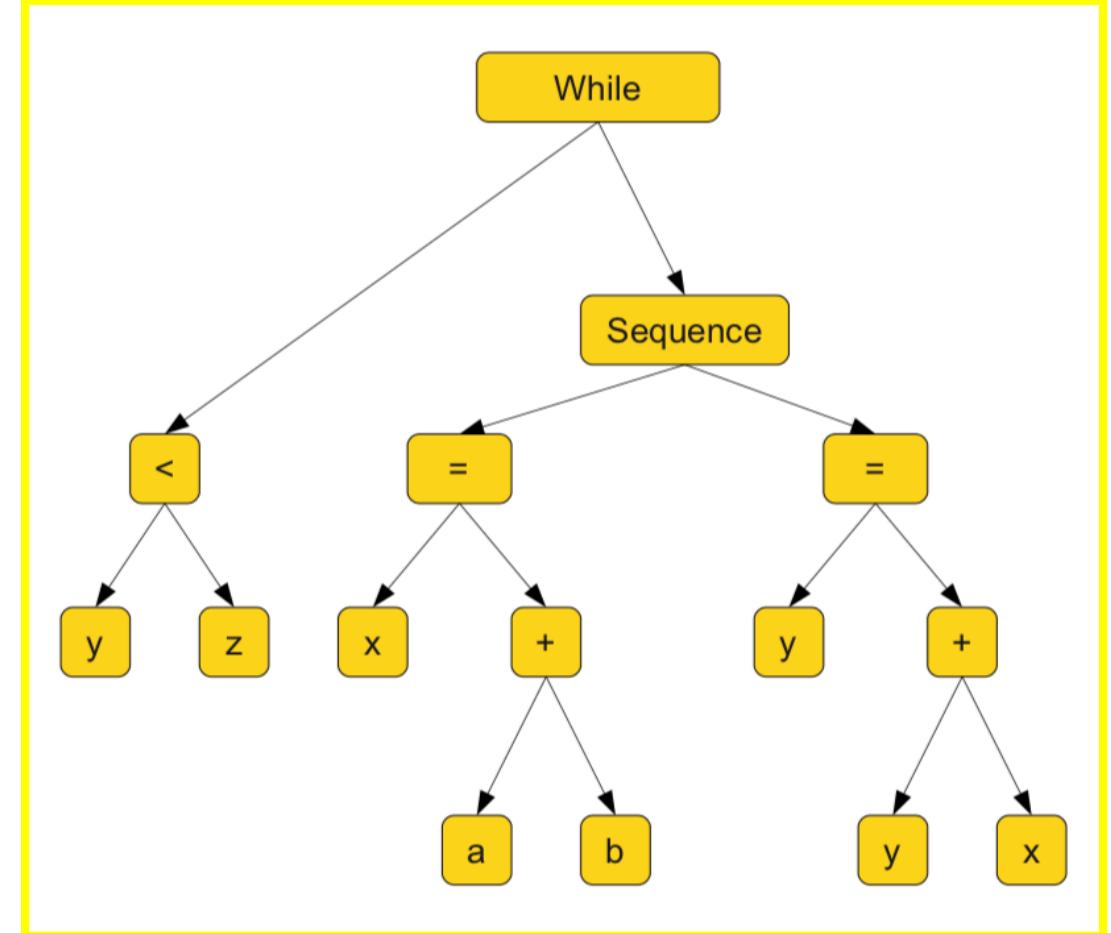
The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

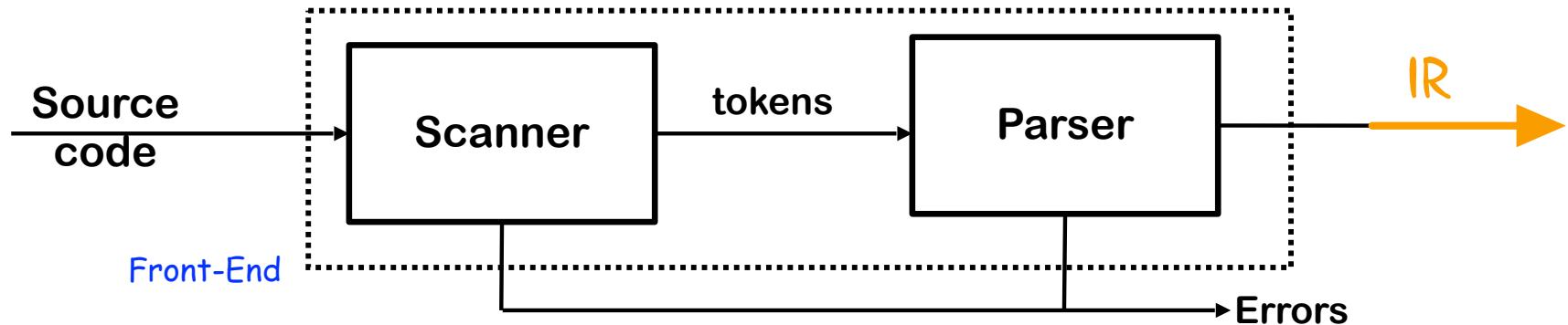
ASTs can be used as intermediate representation

Syntax analysis } PARSER'S TASKS

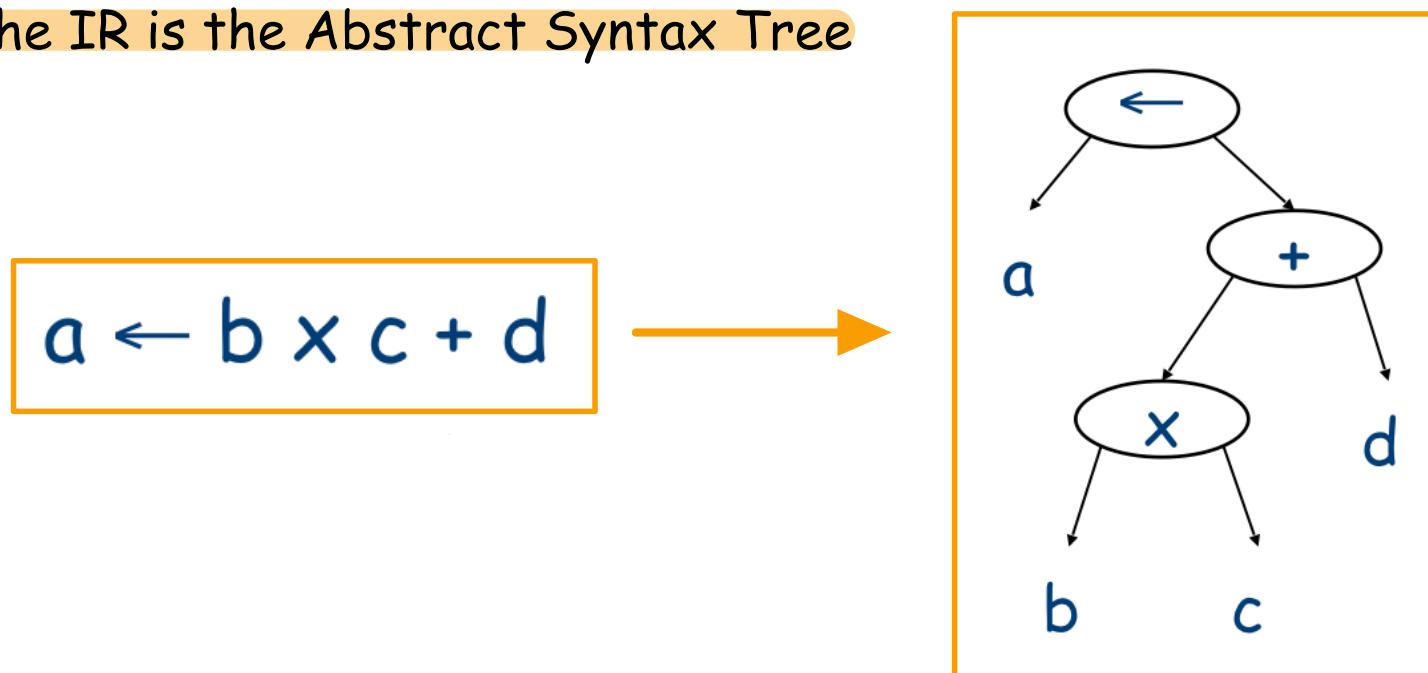
```
while (y < z) {  
    int x = a + b;  
    y += x; }
```



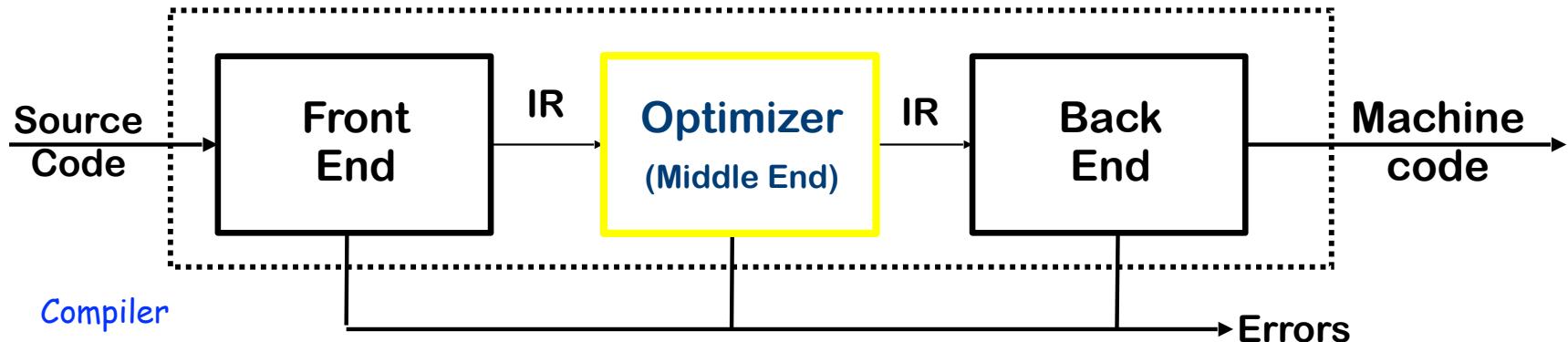
Front-End produces the IR



If the IR is the Abstract Syntax Tree



The Optimizer



The IR emitted by the Front-End is generated by looking to each statement at the time

The IR program contains code that will work for **any** surrounding context

The optimizer can discover something on the context from the entire IR code and use this knowledge to improve the code

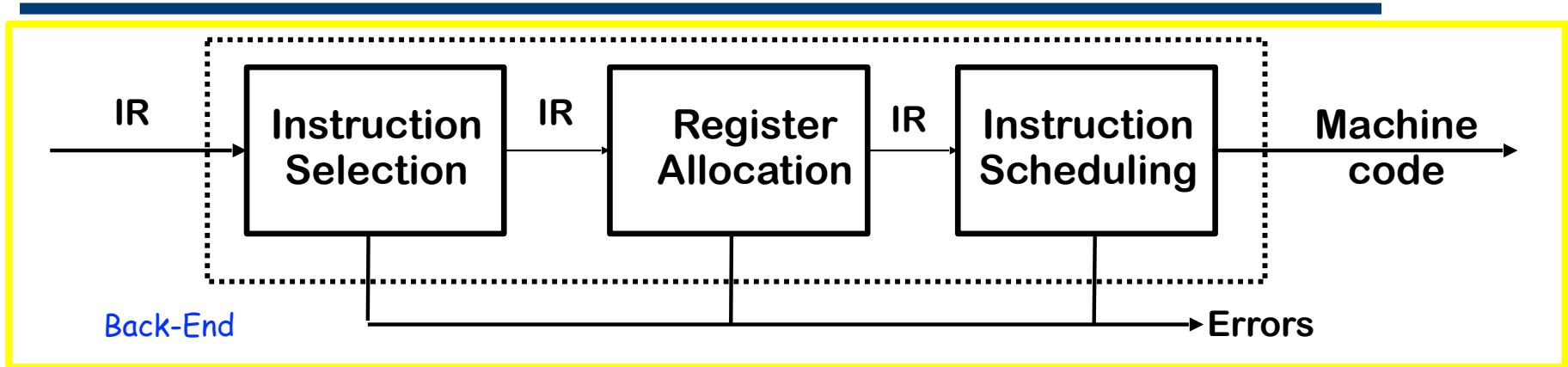
Example of optimizations: loop invariant

```
b ← ...
c ← ...
a ← 1
for i = 1 to n
    read d
    a ← a × 2 × b × c × d
end
```

```
b ← ...
c ← ...
a ← 1
t ← 2 × b × c
for i = 1 to n
    read d
    a ← a × d × t
end
```



The Back End



Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Reorder the instructions so that efficiency is gained

Automation has been less successful in the back end

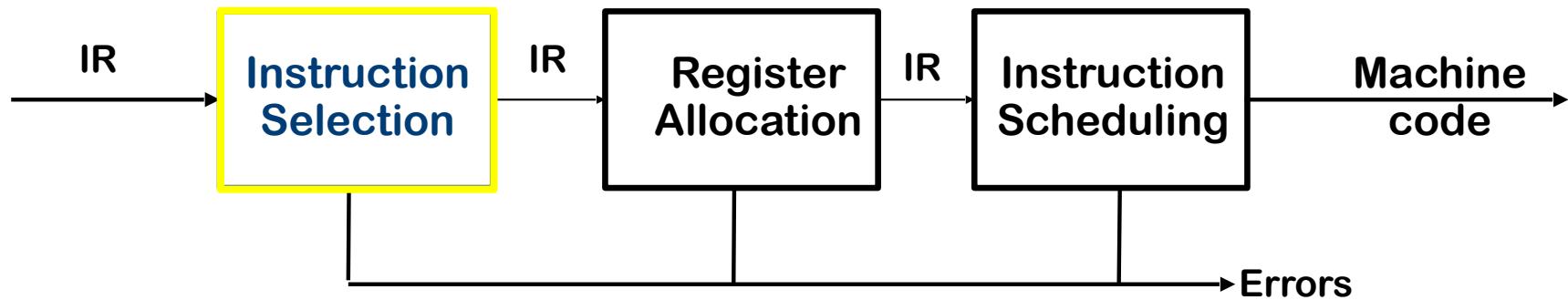
About ILOC

- ILOC (Intermediate Language for Optimizing Compiler) is an assembly language for a simple RISC machine.

/
"ASSEMBLY EXECUTING" MACHINE

ILOC Operation	Meaning
loadAI $r_1, c_2 \Rightarrow r_3$	$\text{Memory}(r_1 + c_2) \rightarrow r_3$
loadI $c_1 \Rightarrow r_2$	$c_1 \rightarrow r_2$
mult $r_1, r_2 \Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
storeAI $r_1 \Rightarrow r_2, c_3$	$r_1 \rightarrow \text{Memory}(r_2 + c_3)$

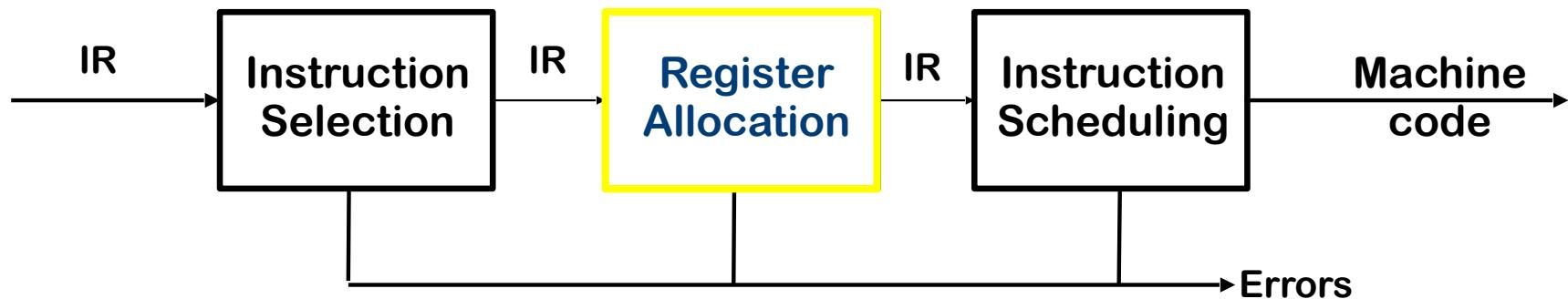
The Back End



Instruction Selection

- It has to translate the IR code into sequence of ISA instructions
- Take advantage of features of the target machine
- Assume an infinite number of (virtual) registers
- Usually viewed as a pattern matching problem
 - ad hoc methods, pattern matching
 - Form of the IR influences choice of technique
 - RISC architecture simplified this problem

The Back End

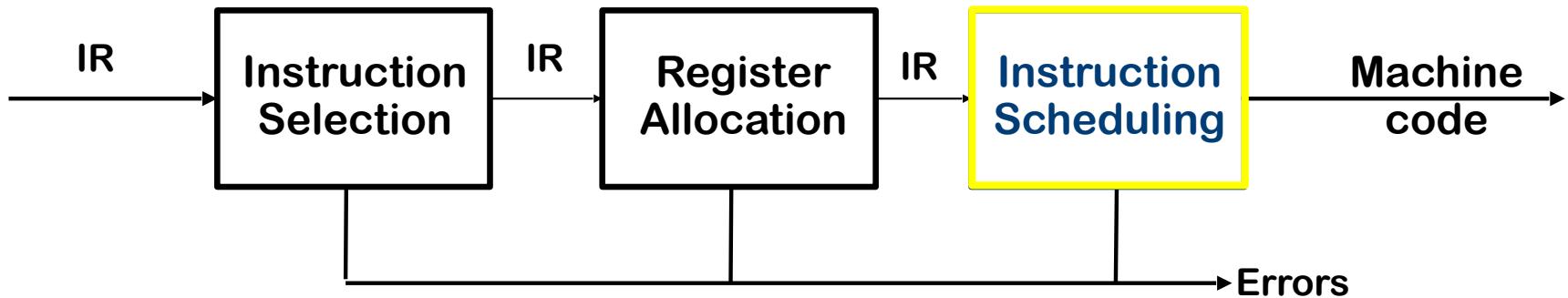


Register Allocation

- It has to map virtual to physics registers
- Manage a limited set of resources
- Can change instruction choices & insert LOADS & STORES
- Optimal allocation is NP-Complete in most settings

Compilers approximate solutions to NP-Complete problems

The Back End



Instruction Scheduling

- It reorders the sequence of instructions to avoid stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed



FORMAL LANGUAGES

Strings

- An **alphabet** is a finite set of symbols

A **string** over alphabet Σ is a finite sequence of symbols in Σ .

The **empty string** is a string having no symbol, denoted by ϵ .

Operations on strings: lenght

The **length** of a string x , denoted by $|x|$, is the number of symbols which compose x .

Operations on strings: concatenation and substrings

- The **concatenation** of two strings x and y is a string xy ,
It is an associative operation that admits the neutral element ϵ
- s is a **substring** of x if there exist two strings y and z such that $x = ysz$.
- In particular,
when $x = sz$ (substring with $y=\epsilon$), s is called a **prefix** of x ;
when $x = ys$ (substring with $z=\epsilon$), s is called a **suffix** of x ;
 ϵ is a prefix and a suffix of any string (including ϵ itself)

Power of an alphabet

- We define the set of all strings over Σ of a given length.
 Σ^n denotes the strings of length n whose symbols are in Σ

If $\Sigma = \{0,1\}$

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4 \cup \dots = \bigcup_{i>0} \Sigma^i \quad \left. \right\} \text{SET OF ALL NON-EMPTY STRINGS}$
- $\Sigma^* = \{\epsilon\} \cup \Sigma^+ \quad \left. \right\} \backslash \text{SET OF ALL POSSIBLE STRINGS}$

Languages

A **language** is a set of strings over an alphabet:

$L \subseteq \Sigma^*$ is a language over Σ

Operations on Languages

Union: $A \cup B$

Intersection: $A \cap B$

Difference: $A \setminus B$ (when $B \subseteq A$)

Complement: $\bar{A} = \Sigma^* - A$ where Σ^* is the set of all strings on Σ

Concatenation: $AB = \{ab \mid a \in A \text{ and } b \in B\}$

Kleene Clousure

● Kleene closure:

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

$$A^+ = \bigcup_{i=1}^{\infty} A^i$$

Problems

• Does the string w belong to the language L ?

We want to define a procedure to decide it!

- A) We can try to generate all words....
- B) We can try to recognise when a word belongs to L

The generative approach: Grammars

A)

Starting from a particular initial symbol, using the rewriting rules of the productions,

we generate the set of all the strings belonging to the language

Definition of Grammars

We define a Grammar $G=(\Sigma, N, S, P)$ where :

- Σ is the alphabet, a set of symbols (called terminals)
- N is the set of nonterminals
- $S \in N$ is the starting symbol
- P is the set of productions, each of the form

$$U \rightarrow V$$

where $U \in (\Sigma \cup N)^+$ and $V \in (\Sigma \cup N)^*$.

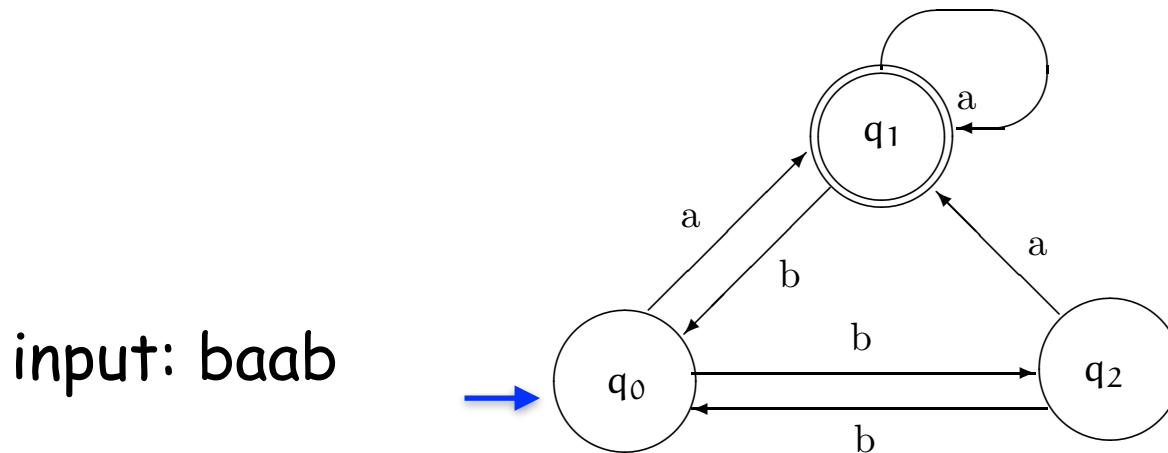
Derivations of $G = (\Sigma, N, S, P)$

A string $w \in \Sigma^*$ is generated by G if there exists a derivation starting from S and resulting in w obtained by rewriting the string using the productions in P .

A language generated by grammar G is denoted $L(G)$ and it is the set of strings derived using G .

Recognising a language: Automata B)

- A finite state automaton is finite state machine with an input of discrete values.
- The state machine consumes the input and possibly moves to a different state.
- The system may be in a state among a finite set of possible states. Being in a state allows to keep track of previous history.



Back to our Problems

- Does the string w belong to the language L ?

We want to define a procedure to decide it!

- Which is the computational complexity necessary to answer to the previous question ?



It depends on the complexity of the language!!

Regular languages

All the following ways to represent regular languages are equivalent:

- Regular grammars (RG, type 3)
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Non-deterministic finite automata with ϵ transitions (ϵ -NFA)
- Regular expressions (RE)

Regular Grammars

A Right (or, analogously, Left) Regular Grammar is a grammar, where

- every production has the form $A \rightarrow aB \mid a$
- only for the starting symbol S we can have $S \rightarrow \epsilon$

Deterministic Finite Automata

A deterministic finite automaton (DFA) $(Q, \Sigma, \delta, q_0, F)$

Q a finite set of states

Σ a finite set Σ of symbols

$\delta : Q \times \Sigma \rightarrow Q$ the transition function takes as argument a state and a symbol
and returns one state

q_0 the starting state

$F \subseteq Q$ the set of final or accepting states

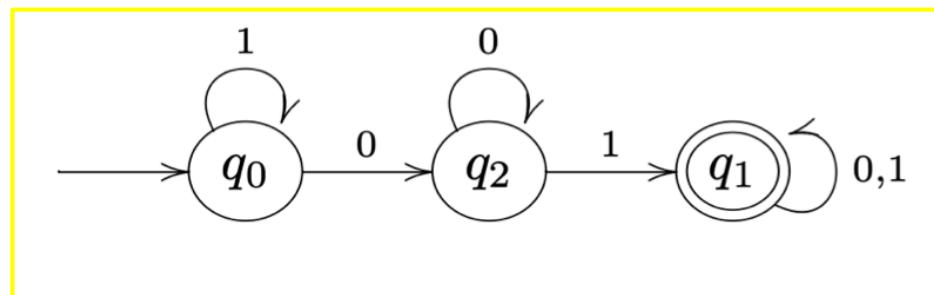
Deterministic Finite Automata

How to represent a DFA? With a **transition table**

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

-> indicates the starting state
* indicates the final states

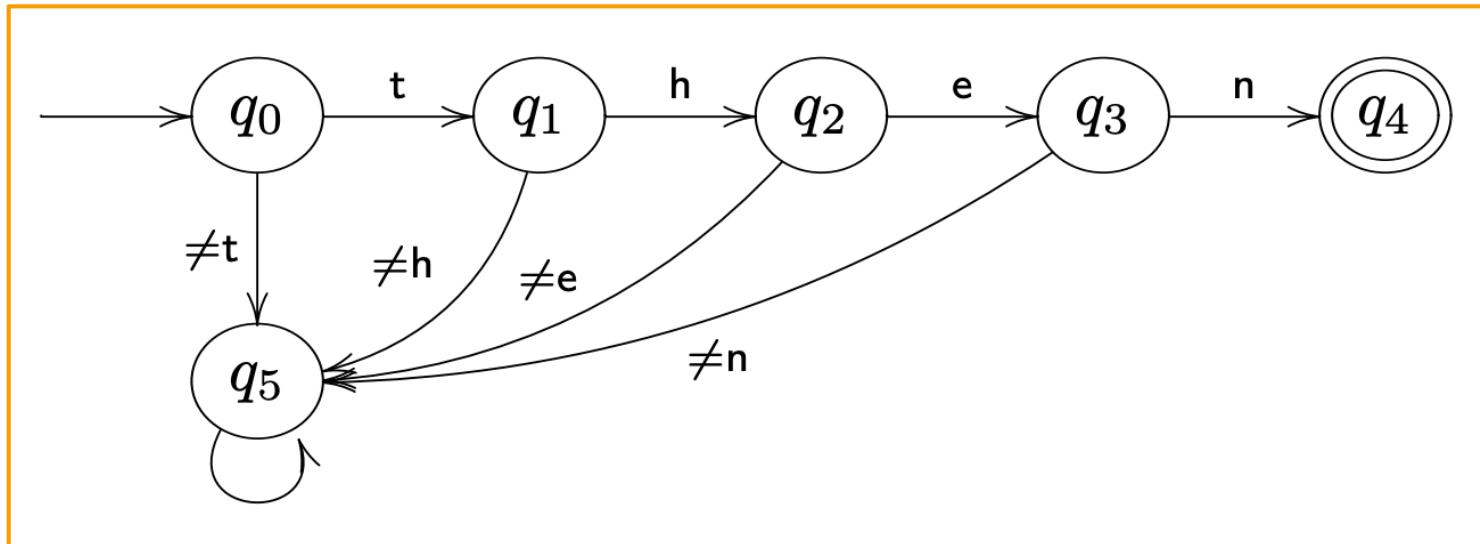
This defines the following transition system



Deterministic Finite Automata

When does an automaton accept a word?

It reads a word and accept it if it stops in an accepting state



here $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$ $F = \{q_4\}$

Only the word **then** is accepted

Extending the transition function to strings

We define the transitive closure of δ

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow Q$$
$$\left\{ \begin{array}{lcl} \hat{\delta}(q, \varepsilon) & = & q \\ \hat{\delta}(q, wa) & = & \delta(\hat{\delta}(q, w), a) \end{array} \right.$$

A string x is accepted by $M=(Q, \Sigma, \delta, q_0, F)$ iff $\hat{\delta}(q_0, x) \in F$

$$L(M) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\}$$

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) allows more than one transition on the same input symbol.

Formally, a NFA is defined as $(Q, \Sigma, \delta, q_0, F)$ where the only difference is the transition function

$\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ a transition function that takes as argument a state and a symbol and returns a set of states

Extending the transition function to strings

We define the transitive closure of δ

$$\left\{ \begin{array}{l} \hat{\delta}(q, \varepsilon) = \{q\} \\ \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q, w)} \delta(p, a) \end{array} \right.$$

A string x is accepted by $M=(Q, \Sigma, \delta, q_0, F)$ iff $\hat{\delta}(q_0, x) \cap F \neq \emptyset$

$$L(M) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \cap F \neq \emptyset\}$$

- NFAs do not expand the class of language that can be accepted.

Different characterisation of Regular Languages

There are different ways to characterise a regular language

- Regular grammars
- Deterministic Finite Automata
- Non deterministic Finite Automata
- Epsilon non deterministic Finite Automata
- Regular expression

Roadmap: equivalence between NFA and RG

DFA

NFA

RG

RE

ϵ -NFA

From Regular Grammars to NFA

Theorem 1.

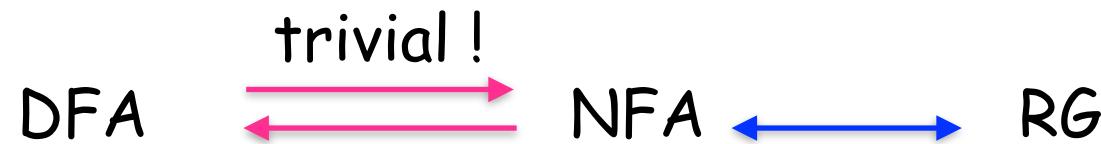
For each right grammar RG there is a non deterministic finite automaton NFA such that $L(RG)=L(NFA)$.

From NFA to Regular Grammars

Theorem 2

For each nondeterministic automaton NFA, there is one right grammar RG such that $L(RG)=L(NFA)$.

Roadmap: equivalence between DFA and NFA



From a NFA to a DFA

The NFA are usually easier to "program".

For each NFA N there is a DFA D , such that $L(D) = L(N)$.

This involves a subset construction.

Given an

NFA $N =$

we will build a $(Q_N, \Sigma, \delta_N, q_0, F_N)$

DFA $D =$

such that $(Q_D, \Sigma, \delta_D, q_0, F_D)$

$L(D) = L(N)$



From NFA to a DFA

$$Q_D = \wp(Q_N),$$

Note that not all these states are necessary, most of them will be unreachable.

$$\forall P \in \mathcal{P}(Q_N) : \quad \delta_D(P, a) = \bigcup_{p \in P} \delta_N(p, a)$$

$$F_D = \{P \in \mathcal{P}(Q_N) \mid P \cap F \neq \emptyset\}$$

The ϵ -NFA: NFA with epsilon transitions

- Extension of finite automaton.
- The new feature: we allow transition on ϵ , the empty string.
- An NFA that is allowed to make transition spontaneously, without receiving any input symbol.
- As in the case of NFA w.r.t. DFA this new feature does not expand the class of languages that can be accepted.

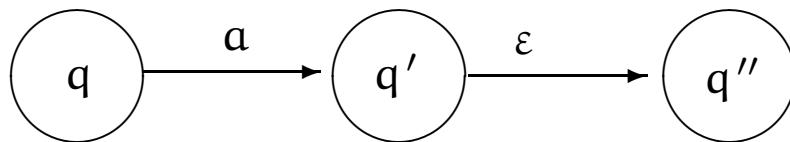
Definition of ϵ -NFA

A NFA whose transition function can always choose epsilon as input symbol

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \wp(Q)$$

Definition of ϵ -closure for extending δ to Strings

We need to define the ϵ -closure that applied to a state gives all the states reachable with ϵ -transitions



$$\epsilon\text{-closure}(q) = \{q\} \quad \epsilon\text{-closure}(q') = \{q', q''\}$$

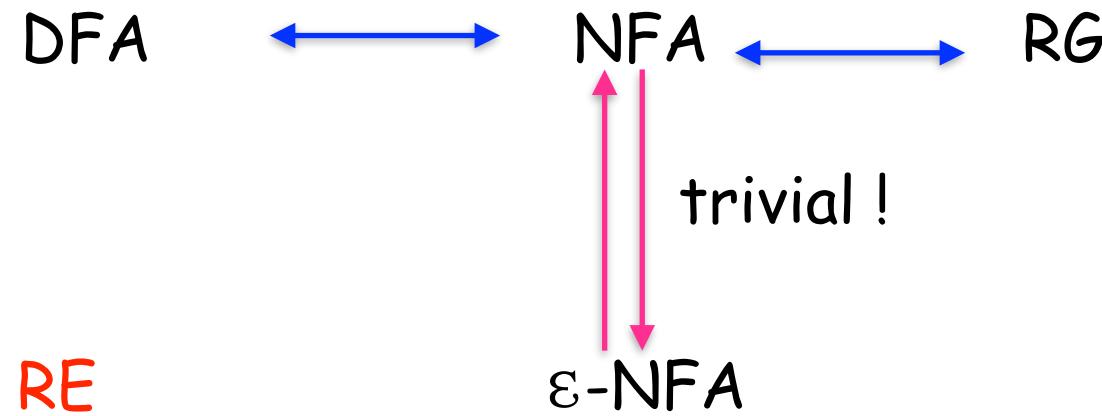
$$\epsilon\text{-closure}(P) = \bigcup_{p \in P} \epsilon\text{-closure}(p)$$

The extension of δ to strings

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow \wp(Q)$$

$$\begin{cases} \hat{\delta}(q, \varepsilon) = \varepsilon\text{-closure}(q) \\ \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q, w)} \varepsilon\text{-closure}(\delta(p, a)) \end{cases}$$

Roadmap: equivalence between NFA and ϵ -NFA



From ϵ -NFA to NFA

For each ϵ -NFA E there is a NFA N , such that $L(E) = L(N)$, and vice versa.

Given an

$$\epsilon\text{-NFA } E = (Q, \Sigma, \delta_E, q_0, F_E)$$

we build a

$$\text{NFA } N = (Q, \Sigma, \delta_N, q_0, F_N)$$

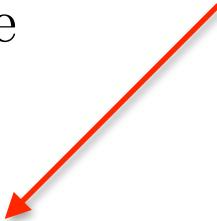
such that

$$L(E) = L(N)$$

Equivalence between ϵ -NFA and NFA

$$\delta_N(q, a) = \widehat{\delta}_E(q, a)$$

$$F_N = \begin{cases} F_E \cup \{q_0\} & \text{if } \epsilon\text{-closure}(q_0) \cap F_E \neq \emptyset \\ F_E & \text{otherwise} \end{cases}$$



if a final state can be reached with an epsilon transition from the initial state

Regular Expressions

A regular expression denotes a set of strings (a language).

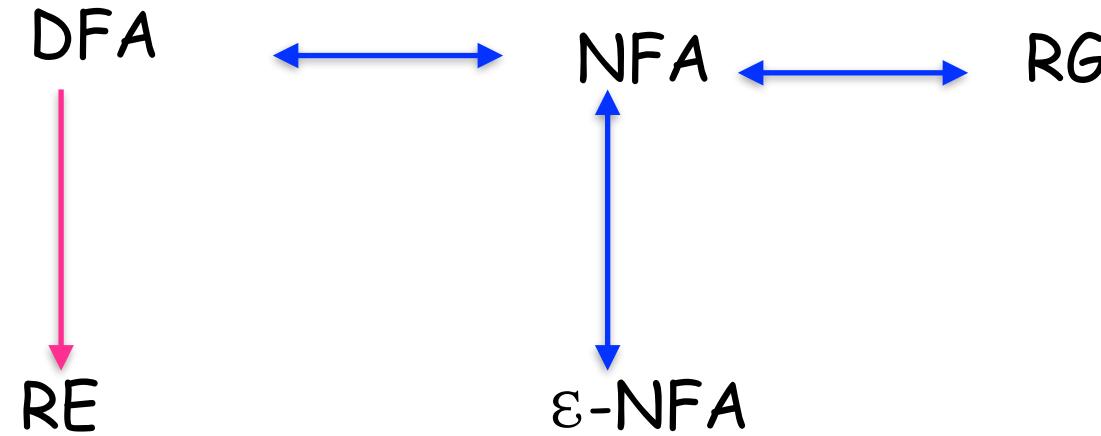
Given a finite alphabet Σ , the following constants are defined as regular expressions:

- \emptyset denoting the empty set,
- ϵ denoting the set $\{\epsilon\}$,
- a in Σ denoting the set containing only the character $\{a\}$

If r and s are regular expression (denoting the sets R and S , respectively) then $(r+s)$, (rs) and r^* denotes the set $R \cup S$, RS and R^* , respectively.

$L(r)$ indicates the language denoted by r

Roadmap

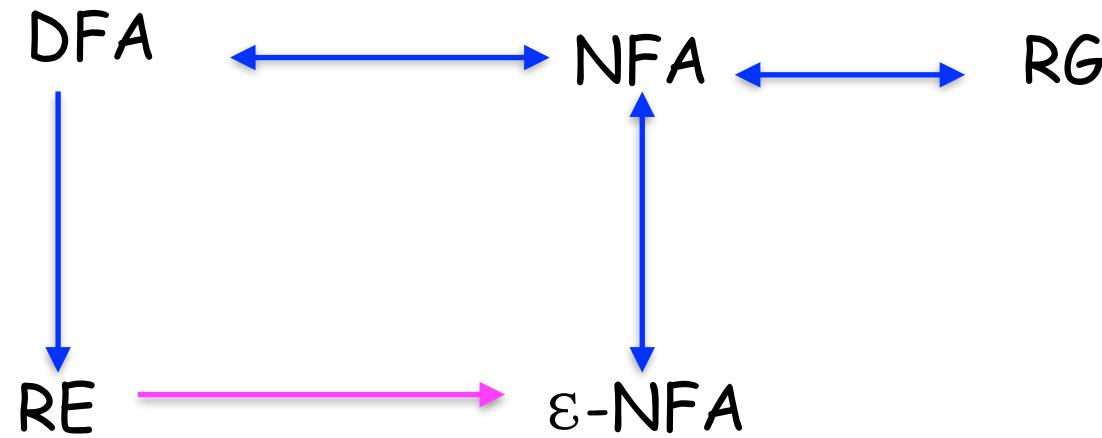


Encoding the language of a DFA into a RE

Theorem 3

For each DFA D , there is a regular expression r such that $L(D)=L(r)$.

Roadmap



Converting a RE to an Automata

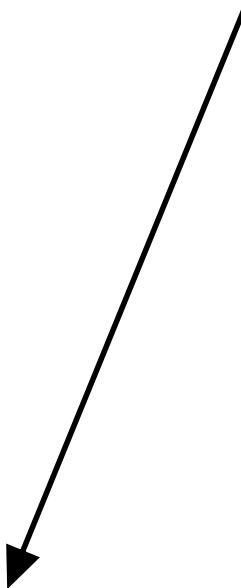
- We can convert a RE to an ϵ -NFA
 - Inductive construction
 - Start with a simple basis, use that to build more complex parts of the NFA

What have we shown?

- Regular expressions, finite state automata and regular grammars are different ways of expressing the same languages
- In some cases you may find it easier to start with one and move to the other
 - e.g., the language of an even number of 1's is typically easier to design as a NFA or DFA and then convert it to a RE

Not all languages are regular!

- $L = \{ a^n b^n \mid n \in \text{Nat} \}$



Pumping Lemma

Given L an infinite regular language then there exists an integer k such that for any string $z \in L, |z| \geq k$ it is possible to split z into 3 substrings

$z = uvw$ with $|uv| \leq k, |v| > 0$ such that $\forall i \in \mathbb{N}, uv^i w \in L$



Negating the PL

The PL gives a necessary condition, that can be used to prove that a language **is not a regular language!**

If $\forall k \in \mathbf{N} \exists z \in L. |z| \geq k$ for all possible splitting

$z = uvw$ with $|uv| \leq k, |v| > 0 \exists i \in \mathbf{N}$ such that $uv^i w \notin L$

then L is not a regular language!

Property of Regular languages

The regular languages are closed with respect to the union, concatenation and Kleene closure.

The complement of a regular language is always regular.

The regular language are closed under intersection

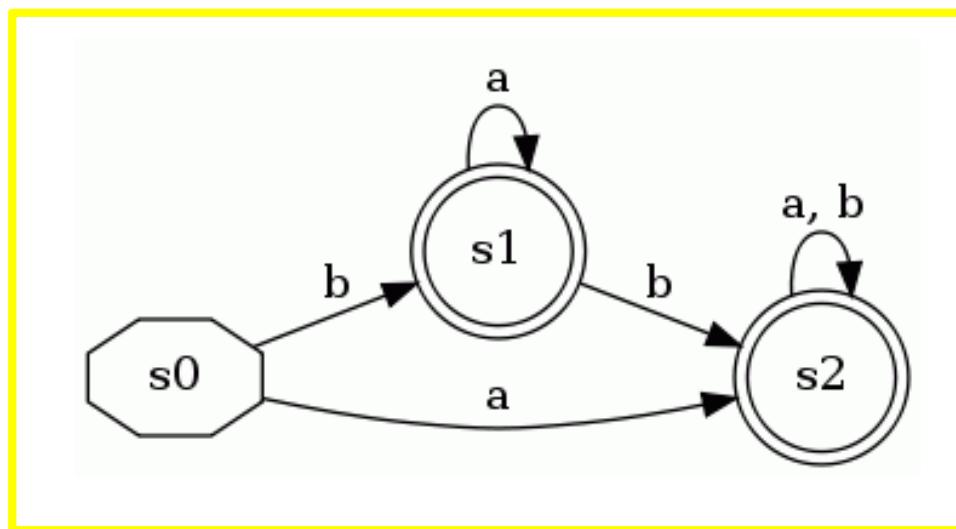
Decision Properties:

Approximately all the properties are **decidable** in case of finite automaton.

- (i) Emptiness
- (ii) Non-emptiness
- (iii) Finiteness
- (iv) Infiniteness
- (v) Membership

DFA Minimization

- Some states can be redundant:
 - The following DFA accepts $(a|b)^*$
 - State $s1$ is not necessary



DFA Minimization

- The task of the DFA minimization is to automatically transform a given DFA into a state-minimized DFA
 - Several algorithms and variants are known

A DFA Minimization Algorithm

- Recall that a DFA $M=(Q, \Sigma, \delta, q_0, F)$
- Two states p and q are distinct if
 - $p \in F$ and $q \notin F$ or vice versa, or
 - $\delta(p, a)$ and $\delta(q, a)$, for some a in Σ , are distinct
- Using this inductive definition, we can calculate which states are distinct

DFA Minimization Algorithm

- Create lower-triangular table DISTINCT , initially blank
- For every pair of states (p,q) :
 - If p is final and q is not, or vice versa
 - $\text{DISTINCT}(p,q) = \epsilon$
- Loop until no change for an iteration:
 - For every pair of states (p,q) and each symbol α
 - If $\text{DISTINCT}(p,q)$ is blank and $\text{DISTINCT}(\delta(p,\alpha), \delta(q,\alpha))$ is not blank
 - $\text{DISTINCT}(p,q) = \alpha$
- Combine all states that are not distinct

CONTENT FREE LANGUAGES

Context free Grammars

A **Context free Grammar** (Σ, N, S, P) is a grammar, where

- every production has the form $U \rightarrow V$

$$U \in N \text{ and } V \in (\Sigma \cup N)^+$$

- only for the starting symbol S , we can have $S \rightarrow \epsilon$

Parse tree

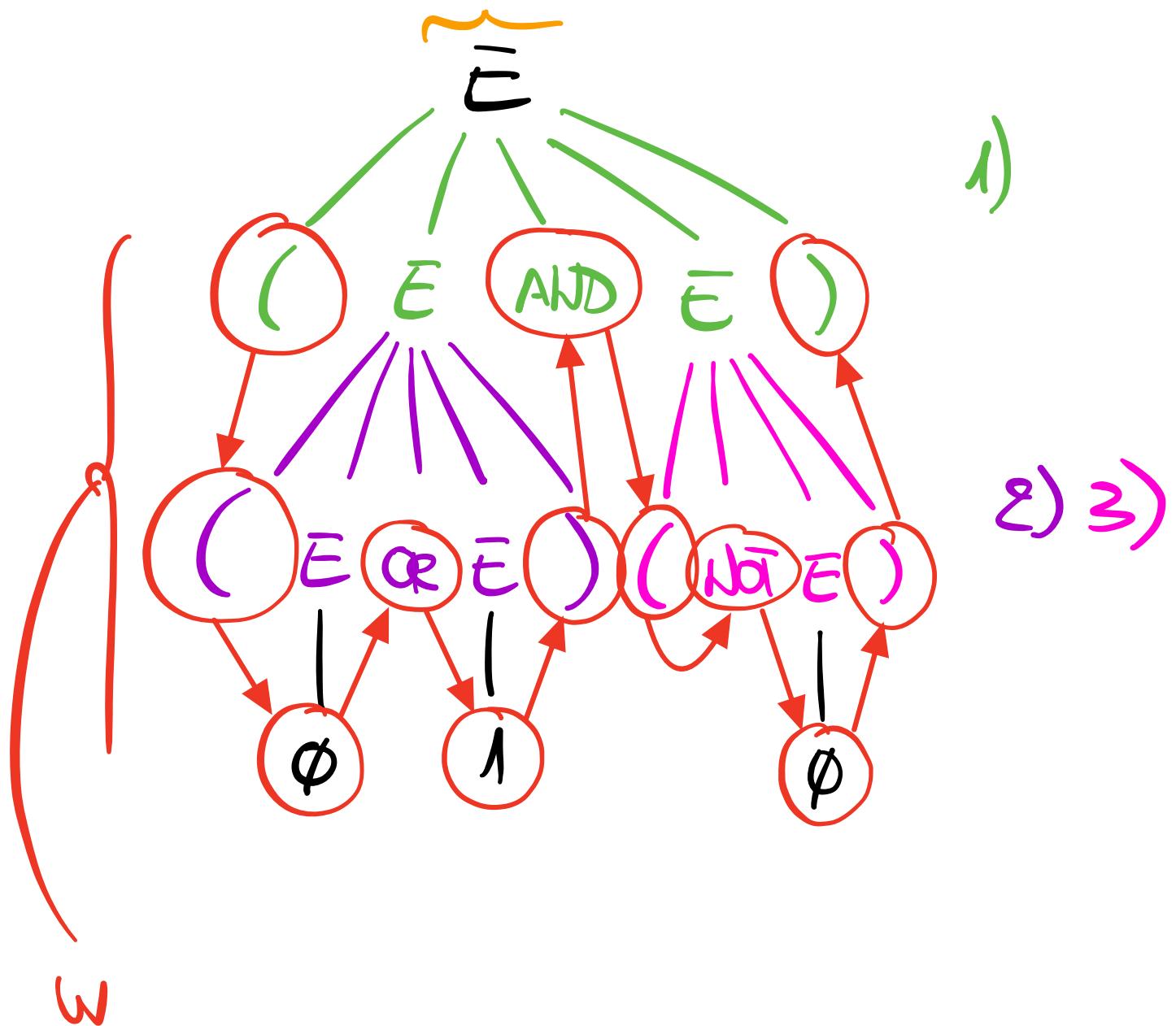
Given a grammar (Σ, N, S, P) .

The parse tree is the graph representation of a derivation, which can be defined in the following way:

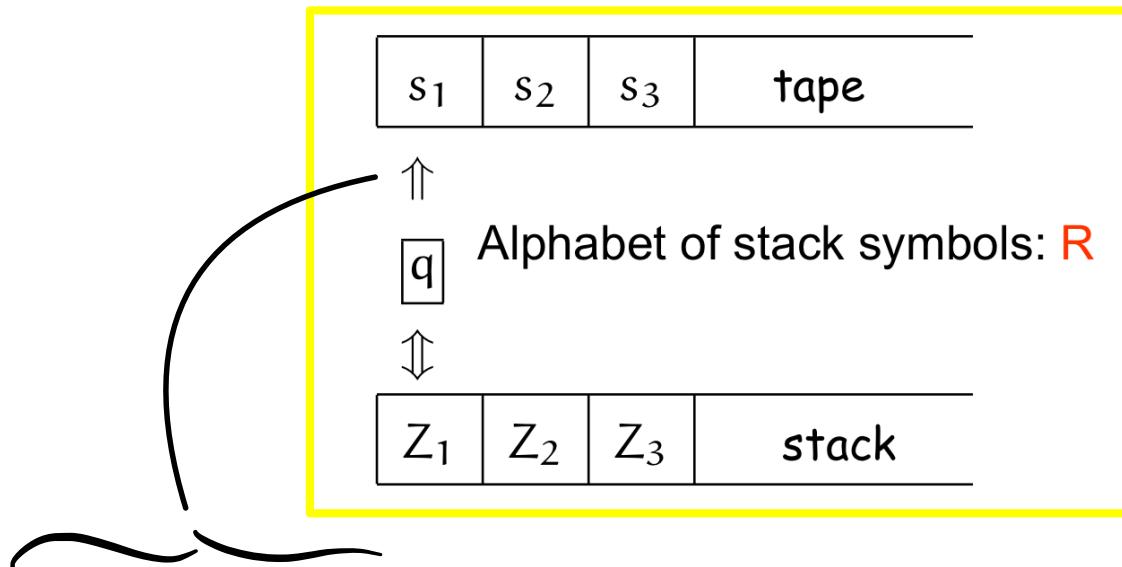
- every vertex has a label in $\Sigma \cup N \cup \{\epsilon\}$,
- the label of the root and of every internal vertex belongs to N ,
- if a vertex is labeled with A and has m children labeled with X_1, \dots, X_k
- then the production $A \rightarrow X_1 \dots X_k$ belongs to P ,
- if a vertex is labeled with ϵ then it is a leaf and is an only child.

E: $\emptyset | 1 | (\bar{E} \text{ or } \bar{\bar{E}}) | (\bar{E} \text{ AND } \bar{E}) | (\text{NOT } \bar{E})$

w = $((\underbrace{\emptyset \text{ or } 1}_{\text{AND}}) \text{ AND } (\text{NOT } \emptyset))$



Pushdown automata



The stack head always scans the top symbol

It performs three basic operations:

- Push: add a new symbol at the top of the stack
- Pop: read and remove the top symbol
- Empty: verify if the stack is empty

Pushdown automata

A push down automaton is $M = (Q, \Sigma, R, \delta, q_0, z_0, F)$ where

- R is the alphabet of stack symbols,

STATE SYMBOL STACK SYMBOL



- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times R \rightarrow \underbrace{\mathcal{P}(Q \times R^*)}$ is the transition function
SUBSET OF STATES AND
STACK SYMBOLS

- z_0 belonging to R is the starting symbol on the stack

Instantaneous Description

The evolution of the PDA is described by triples (q, w, γ) where;

- q is the current state
- w is the unread part of the input string or the remaining input
- γ is the current contents of the stack

A move from one instantaneous description to another

will be denoted by

$$(q_0, aw, Z_r) \mapsto (q_1, w, \gamma_r) \text{ iff } (q_1, \gamma) \text{ belongs to } \delta(q_0, a, Z)$$

The language accepted by a pushdown automaton

Two ways to define the accepted language:

- with empty stack (in this case F is the empty set)

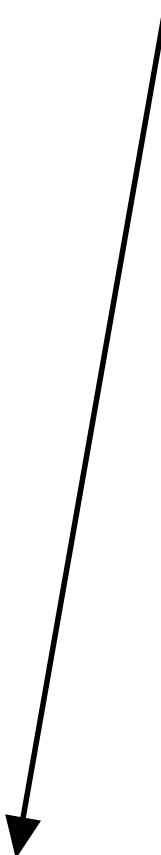
$$L_p(M) = \{x \in \Sigma^* : (q_0, x, Z_0) \xrightarrow{M}^* (q, \varepsilon, \varepsilon), q \in Q\}$$

- with explicit final states F

$$L_F(M) = \{x \in \Sigma^* : (q_0, x, Z_0) \xrightarrow{M}^* (q, \varepsilon, \gamma), \gamma \in R^*, q \in F\}$$

Unfortunately...

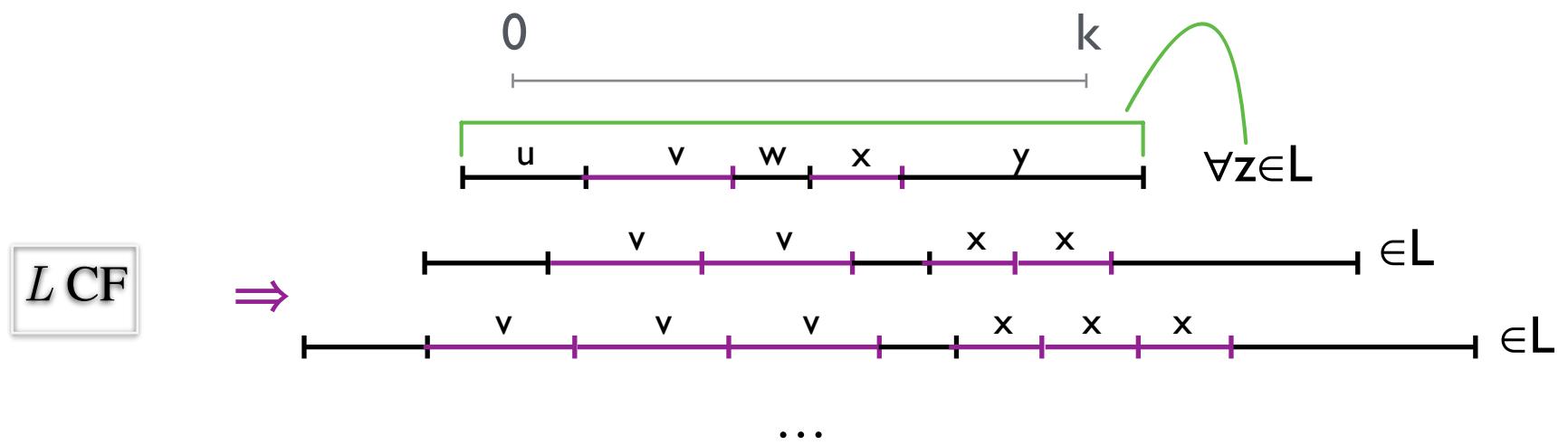
not all languages are Context Free !



Pumping Lemma for CF

Given a context free language L there exists an integer k such that for any string $z \in L. |z| \geq k$ it is possible to split z into 5 substrings

$z = uvwxy$ with $|vwx| \leq k, |vx| > 0$ such that $\forall i \in \mathbf{N}, uv^iwx^i y \in L$



Negating the PL for CF

The PL for CF gives a necessary condition, that can be used to prove that a language **is not context free!**

If $\forall k \in \mathbf{N} \exists z \in L. |z| \geq k$ for all possible splitting of the form

$z = uvwxy$ with $|vwx| \leq k, |vx| > 0 \exists i \in \mathbf{N}$ such that $uv^iwx^i y \notin L$

then **L is not context free!**

Context Sensitive Grammar

Productions of the form $U \rightarrow V$ such that $|U| \leq |V|$

Example

$$\begin{array}{ll} S \rightarrow aSBC \mid aBC & bC \rightarrow bc \\ CB \rightarrow BC & cC \rightarrow cc \\ bB \rightarrow bb & aB \rightarrow ab \\ & \{a^i b^i c^i : i \geq 1\}. \end{array}$$

Examples of Language Hierarchy

The expressing expressive power:

regular C context-free C context-sensitive C phrase-structure

Relationships between Languages and Automata

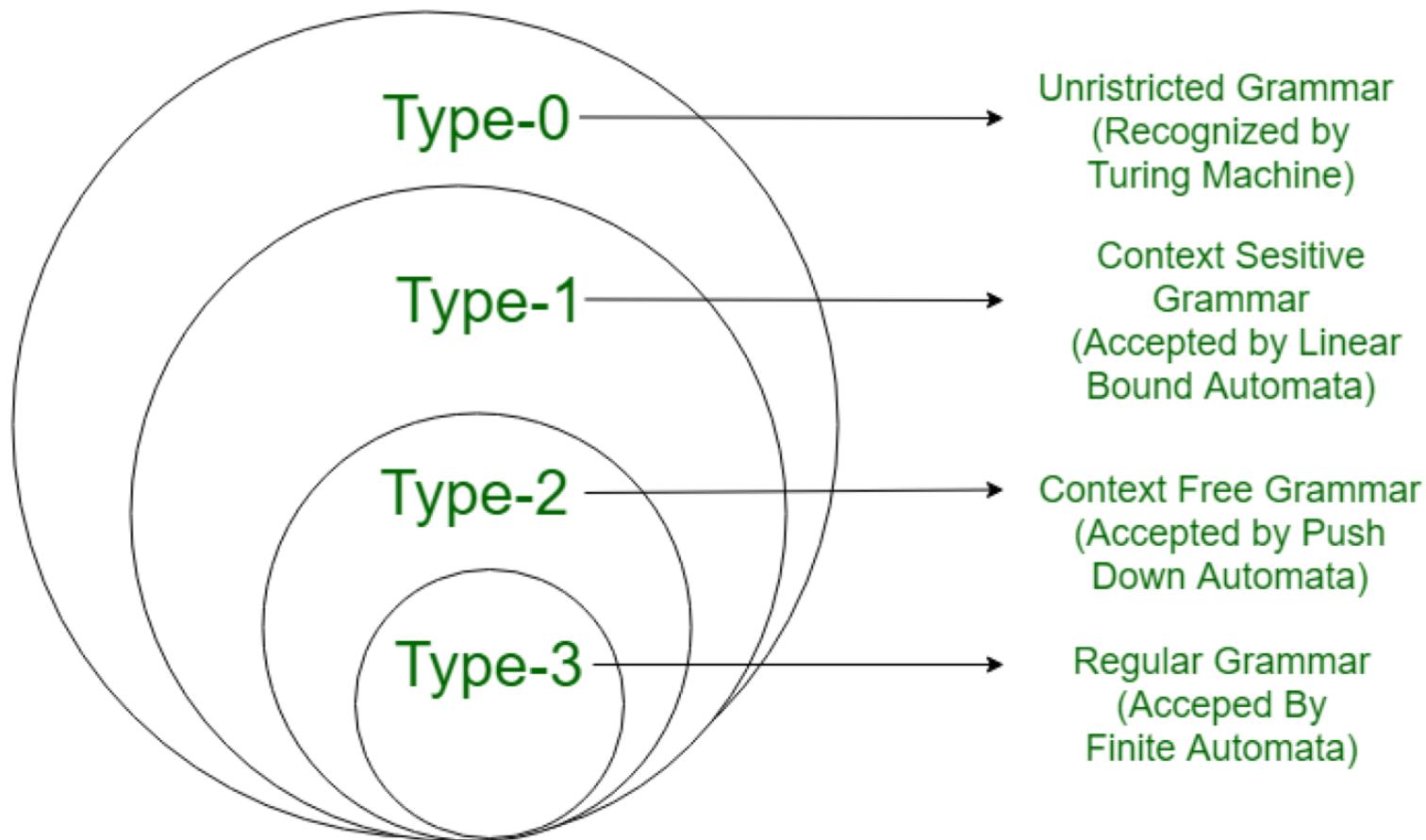
A language is :

regular
context-free
context-sensitive
phrase-structure

iff accepted by

finite-state automata
pushdown automata
linear-bounded automata
Turing machine

Chomsky's Hierarchy





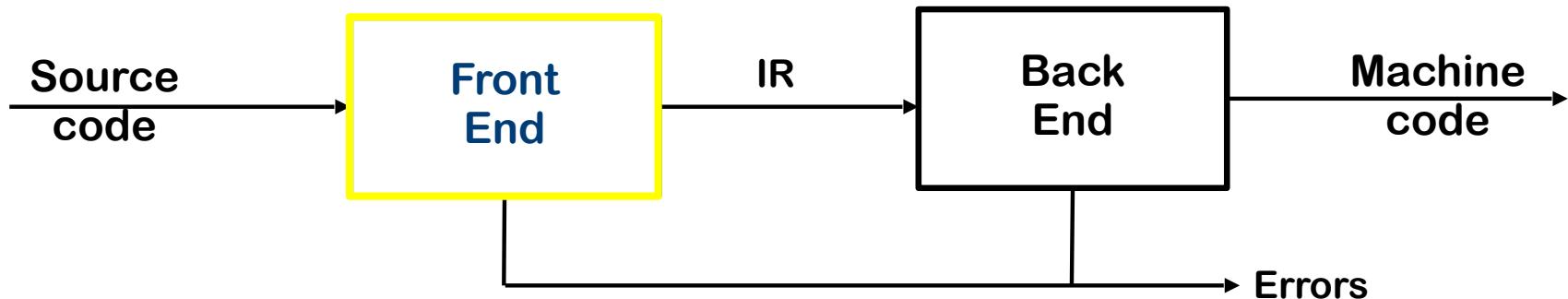
Lexical Analysis

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

The Front End



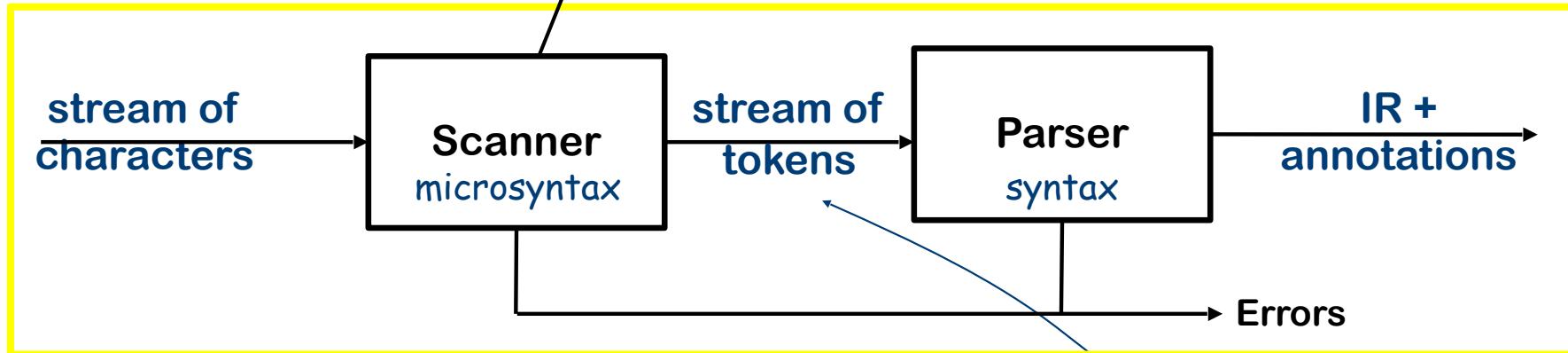
The purpose of the front end is to deal with the input language

- Perform a membership test: $\text{code} \in \text{source language}$?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end deals with form (syntax) & meaning (semantics)

The Front End

Scanner is only pass
that touches every
character of the input



Why separate the scanner and the parser?

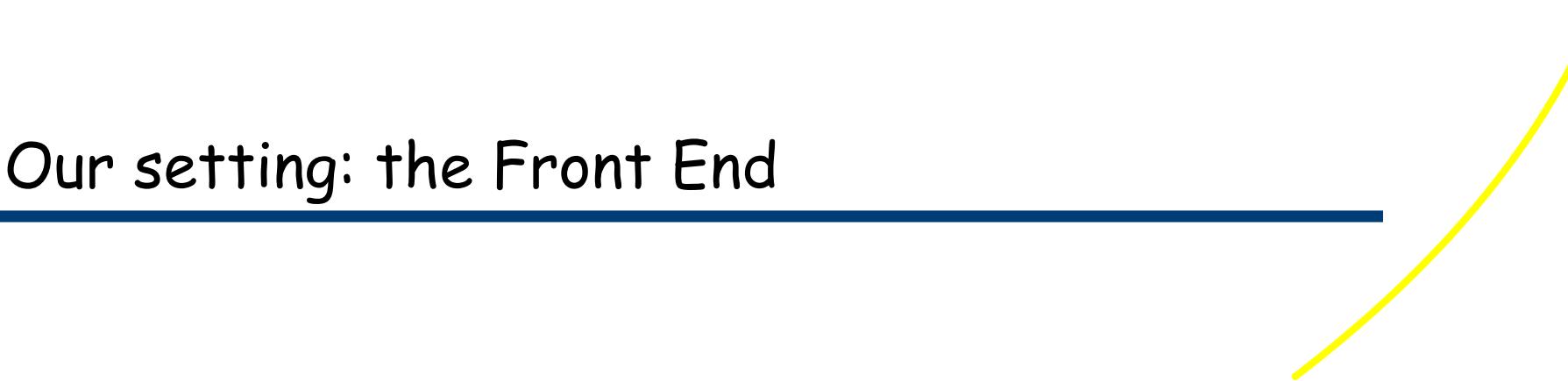
- Scanner classifies words
- Parser constructs grammatical derivations
- Parsing is harder and slower

Separation simplifies the implementation

- Scanners are simple
- Scanner leads to a faster, smaller parser

token is a pair
<part of speech, lexeme>

Our setting: the Front End



	Scanning	Parsing
Specify Syntax	regular expressions	context-free grammars
Implement Recognizer	deterministic finite automaton	push-down automaton
Perform Work	Actions on transitions in automaton	

THE SCANNER'S TASK

Lexical Analysis

Relates to the words of the vocabulary of a language, (as opposed to grammar, i.e., correct construction of sentences).

Lexical Analyzer, a.k.a. lexer, scanner or tokenizer, splits the input program, seen as a stream of characters, into a sequence of tokens.

Tokens are the words of the (programming) language, e.g., keywords, numbers, comments, parenthesis, semicolon.

Tokens are classes of concrete input (called lexeme).

Example

```
#include <stdio.h>
int maximum(int x, int y) {
    // This will compare 2 numbers
    if (x > y)
        return x;
    else {
        return y;
    }
}
```

TOKENS

int Keyword

maximum Identifier

(Operator

int Keyword

x Identifier

, Operator

int Keyword

y Identifier

(Operator

{ Operator

if Keyword

Non-Token

Comment

// This will compare 2
numbers

Pre-processor directive

#include <stdio.h>

Lexical analysis

Lexical analysis is the very first phase in the compiler designing, the only one that analyses the entire code

A **lexeme** is a sequence of characters that are included in the source program according to the matching pattern of a **token**

Lexical analyzer helps to identify token into the symbol table

A character sequence which is not possible to scan into any valid token is a **lexical error**

Constructing a Lexical Analyser

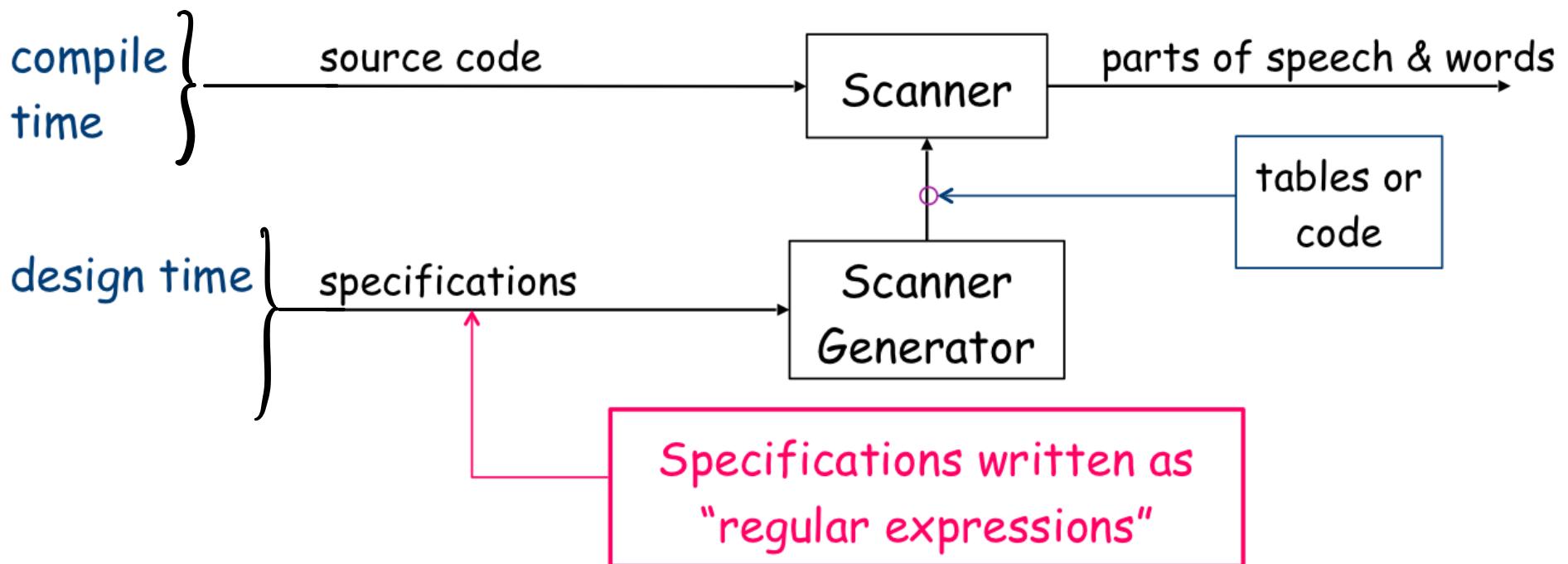
- By hand - Identify lexemes in input and return tokens
- Automatically - **Lexical-Analyser generator**: it compiles the patterns that specify the lexemes into a code (the lexical analyser).

Lexical analysis decides whether the individual tokens are well formed, this can be expressed by a **regular** language.

Automatic Scanner Construction

- Avoid writing scanners by hand

In practice, many scanners are hand coded. Even if you build a hand-coded scanner, it is useful to write down the specification and the automaton.



The syntax can be expressed by a regular grammar

The syntax of a programming language can be expressed by a regular grammar

Example

The following grammar generates all the legal identifier

$$\begin{aligned} S &\rightarrow aT|...|zT|AT|...|ZT \\ T &\rightarrow \epsilon|0T|...|9T|S \end{aligned}$$

that can be more neatly be expressed using a regular expressions !

$$(a|...|z|A|...|Z) (a|...|z|A|...|Z|0|....|9)^*$$

EXAMPLE

Consider the problem of recognizing ILOC register names

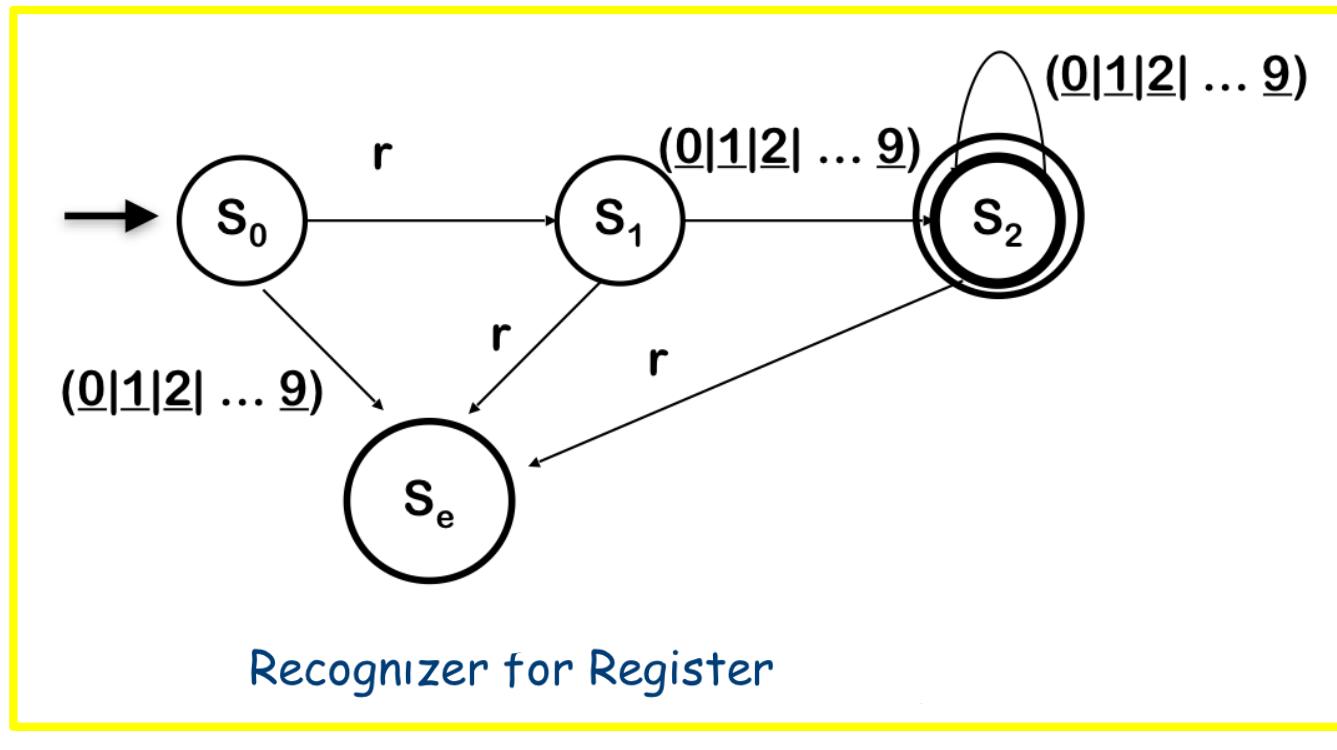
Register $\rightarrow r \ (\underline{0}|\underline{1}|\underline{2}| \dots | \underline{9}) \ (\underline{0}|\underline{1}|\underline{2}| \dots | \underline{9})^*$

- Allows registers of arbitrary number
- Requires at least one digit



DFA operation

- Start in state S_0 & make transitions on each input character



So,

- r17 takes it through s_0, s_1, s_2 and accepts
- r takes it through s_0, s_1 and fails
- Q takes it straight to s_e

To be useful, the DFA must be converted into code

```
Char ← next character  
State ←  $s_0$   
  
while (Char ≠ EOF)  
    State ←  $\delta(\text{State}, \text{Char})$   
    Char ← next character  
  
if (State is a final state)  
    then report success  
else report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	All others
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding the RE
 $O(1)$ cost per character (or per transition)

We can add "actions" to each transition

Table encoding RE

```
Char ← next character
State ←  $s_0$ 
while (Char ≠ EOF)
    Next ←  $\delta$ (State,Char)
    Act ←  $\alpha$ (State,Char)
    perform action Act
    State ← Next
    Char ← next character
if (State is a final state )
    then report success
    else report failure
```

Skeleton recognizer

δ	α	0,1,2,3,4,5,6 ,7,8,9	All others
s_0	s_1 start	s_e error	s_e error
s_1	s_e error	s_2 add	s_e error
s_2	s_e error	s_2 add	s_e error
s_e	s_e error	s_e error	s_e error

Typical action is to capture the lexeme

What if we need a tighter specification?

- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

— Register $\rightarrow r ((\underline{0}|\underline{1}|\underline{2}) (\text{Digit} \mid \varepsilon) \mid (\underline{4}|\underline{5}|\underline{6}|\underline{7}|\underline{8}|\underline{9}) \mid (\underline{3}|\underline{30}|\underline{31}))$

Produces a more complex DFA

- DFA has more states
- DFA has same cost per transition
- DFA has same basic implementation

Automating Scanner Construction

- 1 Write down the RE for the input language
- 1 Build a ϵ -NFA collecting all the NFA for the RE
- 2 Build a NFA corresponding to the ϵ -NFA
- 3 Build the DFA that simulates the NFA
- 4 Systematically minimise the DFA
- 5 Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser
- You could build one in a weekend!

Implementing Scanners

The overall construction: RE \rightarrow ϵ -NFA \rightarrow NFA \rightarrow DFA \rightarrow minimized DFA

How we transform a DFA into code?

- Table driven scanners
- Direct code scanners
- Hand-coded scanners

all will simulate the DFA!

The actions that the different implementations have in common

- They repeatedly read the next character in the input and simulate the corresponding DFA transition
- This process stops when there are not outgoing transition from the state s with the input character
 - If s is an accepting state the scanner recognises the word and its syntactic category
 - If s is a nonaccepting state the scanner must determine whether or not it passes a final state at some point,
 - If yes it should **roll back** its internal state and its input stream and report **success**
 - If not it should report the **failure**

The differences between the different approaches

- Table driven scanners
- Direct code scanners
- Hand-coded scanners

All constant cost per character (with different constants) plus the cost of rollback

Differs from the way they implement the transition table and simulate the operations of the DFA

Table-Driven Scanners

Table(s) + Skeleton Scanner

- So far, we have used a simplified skeleton

```
state ←  $s_0$ ;
```

```
while ( $state \neq s_{\text{error}}$ ) do
```

```
    char ← NextChar()           // read next character
```

```
    state ←  $\delta(state, char)$ ; // take the transition
```

In practice, the skeleton is more complex

- Character classification for table compression

- Building the lexeme

- Recognizing subexpressions

- Practice is to combine all the REs into one DFA

- Must recognize individual words without hitting EOF

Table-Driven Scanners

Character Classification

- Group together characters by their actions in the DFA
 - Combine identical columns in the transition table, δ
 - Indexing δ by class shrinks the table

```
state ←  $s_0$ ;
while (state ≠  $s_{\text{error}}$ ) do
    char ← NextChar()           // read next character
    cat ← CharCat(char)         // classify character
    state ←  $\delta(\text{state}, \text{cat})$  // take the transition
```

r	0, 1, 2, ..., 9	EOF	Other
Register	Digit	Other	Other

The Classifier Table, CharCat

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

Table-Driven Scanners

Building the Lexeme

- Scanner produces syntactic category (part of speech)
 - Most applications want the lexeme (word), too

```
state ←  $s_0$ 
```

```
lexeme ← empty string
```

```
while (state ≠  $s_{\text{error}}$ ) do
```

```
    char ← NextChar() // read next character
```

```
    lexeme ← lexeme + char // concatenate onto lexeme
```

```
    cat ← CharCat(char) // classify character
```

```
    state ←  $\delta(\text{state}, \text{cat})$  // take the transition
```

- This problem is trivial
 - Save the characters

Recognising subexpressions : RollBack

- A stack is used to track all the traversed states

```
lexeme ← empty string
while (state ≠ serror) do
    char ← NextChar()           // read next character
    lexeme ← lexeme + char     // concatenate onto lexeme
    push (state);              //remember all traversed states
    cat ← CharCat(char)        // classify character
    state ← δ(state,cat)
    while (state ≠ sA) do      //RollBack
        state ← pop();
        truncate lexeme;        //sA final state
        Rollback();
    end
```

Table-Driven Scanners

Choosing a Category from an Ambiguous RE

- We want a DFA, so we combine all the REs into one
 - Some strings may fit RE for more than 1 syntactic category
→ Keywords versus general identifiers
 - Scanner must choose a category for ambiguous final states
→ Classic answer: specify priority by order of REs (return 1st)

A TABLE DRIVEN SCANNER FOR REGISTERS NAMES

initialization

```
NextWord()
state ←  $s_0$ ;
lexeme ← “ ”;
clear stack;
push(bad);
```

scanning loop

```
while (state ≠  $s_e$ ) do
    NextChar(char);
    lexeme ← lexeme + char;
    if state ∈  $S_A$ 
        then clear stack;
    push(state);
    cat ← CharCat[char];
    state ←  $\delta$ [state, cat];
end;
```

roll-back

```
while(state ∈  $S_A$  and
      state ≠ bad) do
    state ← pop();
    truncate lexeme;
    RollBack();
end;
```

final-section

```
if state ∈  $S_A$ 
    then return Type[state];
else return invalid;
```

r	0, 1, 2, ..., 9	EOF	Other
Register	Digit	Other	Other

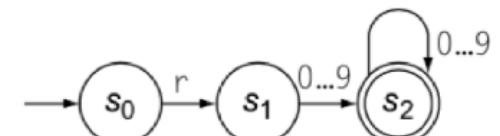
The Classifier Table, $CharCat$

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

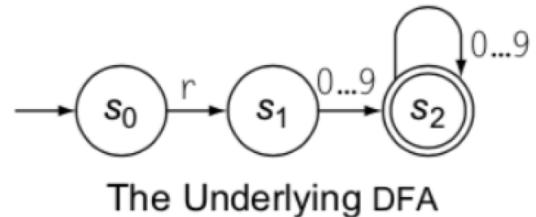
s_0	s_1	s_2	s_e
invalid	invalid	register	invalid

The Token Type Table, $Type$



The Underlying DFA

Direct-Coded scanners



For each character, the table driven scanner performs two table lookups, one in *CharCat* and the other in δ : to improve efficiency

```
sinit : lexeme ← “ ”;  
        clear stack;  
        push(bad);  
        goto s0;  
  
s0 : NextChar(char);  
       lexeme ← lexeme + char;  
       if state ∈ SA  
           then clear stack;  
       push(state);  
       if (char = ‘r’)  
           then goto s1;  
       else goto sout;  
  
s1 : NextChar(char);  
       lexeme ← lexeme + char;  
       if state ∈ SA  
           then clear stack;  
       push(state);  
       if (‘0’ ≤ char ≤ ‘9’)  
           then goto s2;  
       else goto sout;  
  
s2 : NextChar(char);  
       lexeme ← lexeme + char;  
       if state ∈ SA  
           then clear stack;  
       push(state);  
       if ‘0’ ≤ char ≤ ‘9’  
           then goto s2;  
       else goto sout  
  
sout : while (state ≠ SA and  
            state ≠ bad) do  
         state ← pop();  
         truncate lexeme;  
         RollBack();  
     end;
```

If the state test is complex (e.g., many cases),
scanner generator should consider other schemes
• Binary search

Building Scanners

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting

Of course, not everything fits into a regular language ...

What About Hand-Coded Scanners?

Many (most?) modern compilers use hand-coded scanners

- Starting from a DFA simplifies design & understanding
 - Can use old assembly tricks
 - Combine similar states
- Scanners are fun to write
 - Compact, comprehensible, easy to debug



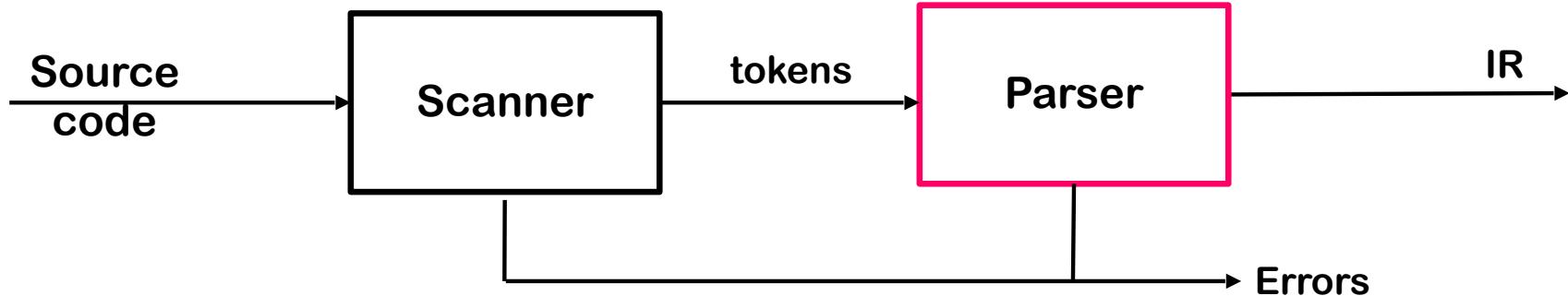
Introduction to Parsing

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

The Study of Parsing

The process of discovering a derivation for some sentence

- Need a mathematical model of syntax – a grammar G
- Need an algorithm for testing membership in $L(G)$

Roadmap for our study of parsing

- 1 Context-free grammars and derivations
- 2 Top-down parsing
 - Generated LL(1) parsers & hand-coded recursive descent parsers
- 3 Bottom-up parsing
 - Generated LR(1) parsers

Why Not Use Regular Languages & DFAs?

Not all languages are regular (RL's ⊂ CFL's ⊂ CSL's)

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ (correspondence between declarations and variables)
- $L = \{ w c w^r \mid w \in \Sigma^* \}$ (parenthesis languages)

Neither of these is a regular language

To recognize these features requires an arbitrary amount of context (left or right ...)

But, this issue is somewhat subtle. You can construct DFA's for

- Strings with alternating 0's and 1's
 $(\epsilon \mid 1)(01)^*(\epsilon \mid 0)$
- Strings with an even number of 0's and 1's

RE's can count bounded sets and bounded differences

⇒ Cannot add parenthesis, brackets, begin-end pairs, ...

A More Useful Grammar Than Sheep Noise

To explore the uses of CFGs, we need a more complex grammar

0	Expr	\rightarrow	Expr Op Expr
1			<u>num</u>
2			<u>id</u>
3	Op	\rightarrow	+
4			-
5			*
6			/



Rule	Sentential Form
-	Expr
0	Expr Op Expr
2	<id, <u>x</u> > Op Expr
4	<id, <u>x</u> > - Expr
0	<id, <u>x</u> > - Expr Op Expr
1	<id, <u>x</u> > - <num, <u>2</u> > Op Expr
5	<id, <u>x</u> > - <num, <u>2</u> > * Expr
2	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >

- Such a sequence of rewrites is called a derivation
- Process of discovering a derivation is called parsing
for the sequence of tokens

Derivations

The goal of parsing is to construct a derivation

- At each step, we choose a nonterminal to replace
- Different choices can lead to different derivations

Two kind of derivations are of interest

- Leftmost derivation – replace leftmost NT at each step
- Rightmost derivation – replace rightmost NT at each step

These are the two systematic derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a leftmost derivation

- Of course, there is also a rightmost derivation
- Interestingly, it turns out to be different

Derivations

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$$

- Each γ_i is a sentential form
 - If γ contains only terminal symbols, γ is a sentence in $L(G)$
 - If γ contains 1 or more non-terminals, γ is a sentential form
- To get γ_i from γ_{i-1} , expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
 - Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
 - In a leftmost derivation, it would be the first NT $A \in \gamma_{i-1}$

A left-sentential form occurs in a leftmost derivation

A right-sentential form occurs in a rightmost derivation

The Two Derivations for $x - 2 * y$

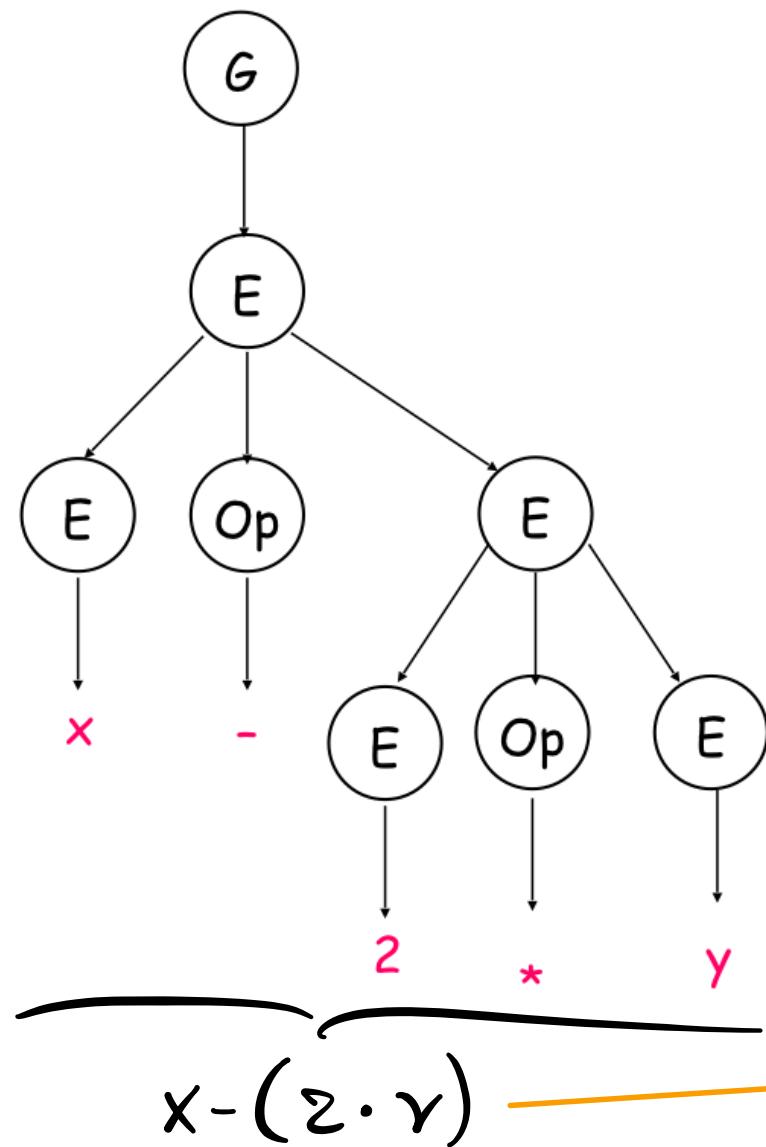
Rule	Sentential Form	Leftmost derivation
-	Expr	
0	Expr Op Expr	
2	<id, <u>x</u> > Op Expr	
4	<id, <u>x</u> > - Expr	
0	<id, <u>x</u> > - Expr Op Expr	
1	<id, <u>x</u> > - <num, <u>2</u> > Op Expr	
5	<id, <u>x</u> > - <num, <u>2</u> > * Expr	
2	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >	

Rule	Sentential Form	Rightmost derivation
-	Expr	
0	Expr Op Expr	
2	Expr Op <id, <u>y</u> >	
5	Expr * <id, <u>y</u> >	
0	Expr Op Expr * <id, <u>y</u> >	
1	Expr Op <num, <u>2</u> > * <id, <u>y</u> >	
4	Expr - <num, <u>2</u> > * <id, <u>y</u> >	
2	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >	

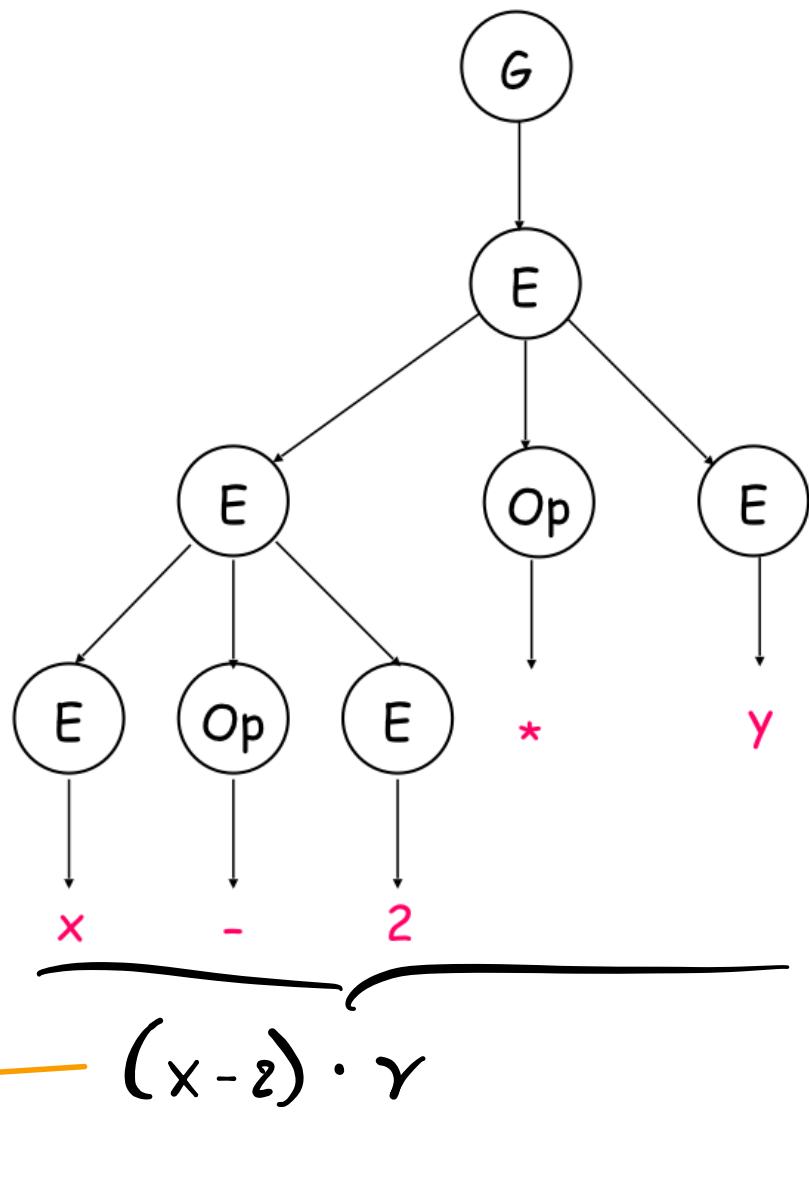
In both cases, $\text{Expr} \Rightarrow \underline{\text{id}} - \underline{\text{num}} * \underline{\text{id}}$

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

LEFT MOST PARSE TREE



RIGHT MOST PARSE TREE



THE AMBIGUITY IS NOT GOOD!

Derivations and Precedence

These two derivations point out a problem with the grammar:

It has no notion of precedence, or implied order of evaluation

To add precedence

- Create a nonterminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions

- Parentheses first (level 1)
- Multiplication and division, next (level 2)
- Subtraction and addition, last (level 3)

Derivations and Precedence

Adding the standard algebraic precedence produces:

level 3	0	Goal	\rightarrow	Expr
	1	Expr	\rightarrow	Expr + Term
	2			Expr - Term
	3			Term
level 2	4	Term	\rightarrow	Term * Factor
	5			Term / Factor
	6			Factor
level 1	7	Factor	\rightarrow	(Expr)
	8			<u>number</u>
	9			<u>id</u>

This grammar is slightly larger

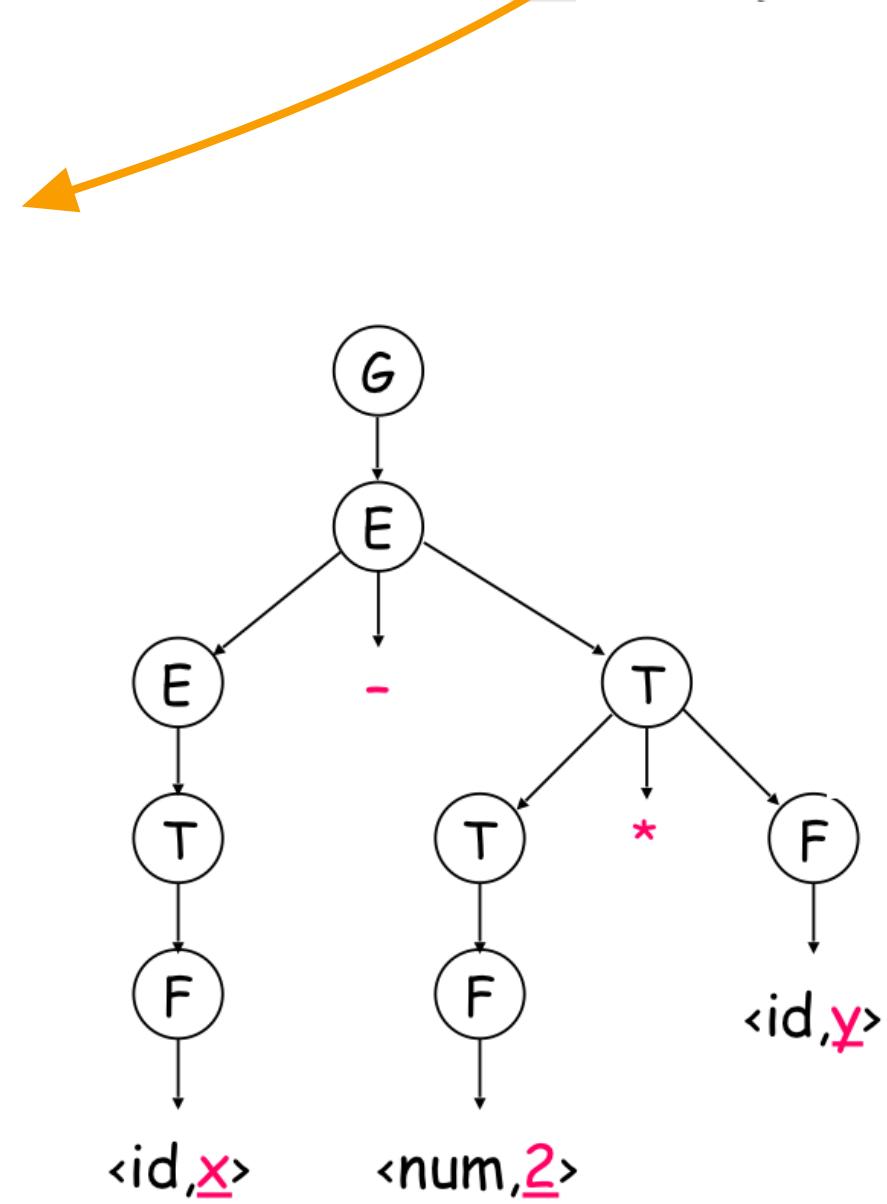
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations
- **Correctness trumps the speed of the parser**

Let's see how it parses $x - 2 * y$

Introduced parentheses, too
(beyond power of an RE)

Derivations and Precedence

Rule	Sentential Form
-	Goal
0	Expr
2	Expr - Term
4	Expr - Term * Factor
9	Expr - Term * <id, <u>y</u> >
6	Expr - Factor * <id, <u>y</u> >
8	Expr - <num, <u>2</u> > * <id, <u>y</u> >
3	Term - <num, <u>2</u> > * <id, <u>y</u> >
6	Factor - <num, <u>2</u> > * <id, <u>y</u> >
9	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >



Both the leftmost and rightmost derivations give the same parse tree, because the grammar explicitly encodes the desired precedence.

Ambiguous Grammars

Definitions

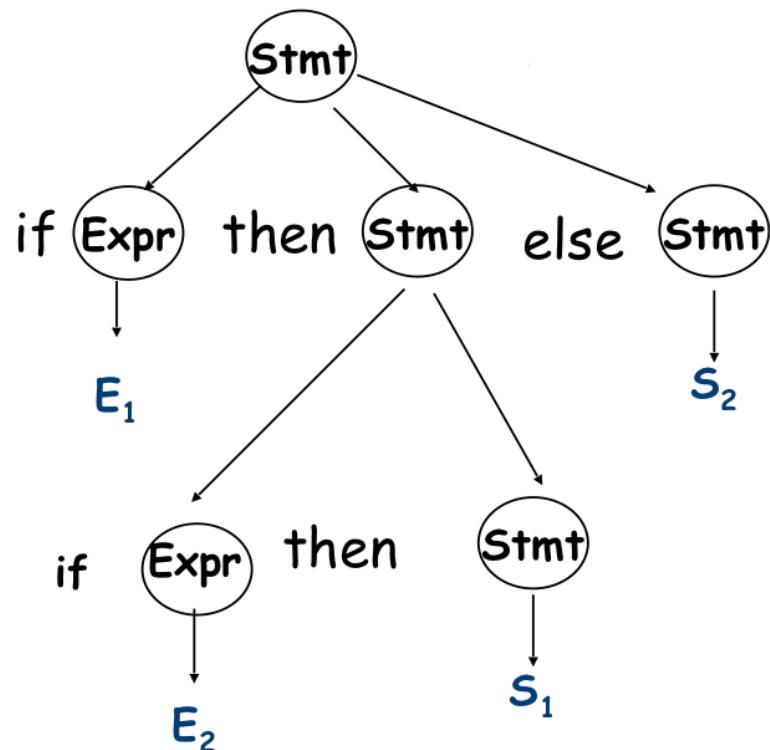
- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar
 - However, they must have the same parse tree!

Classic example – the if-then-else problem

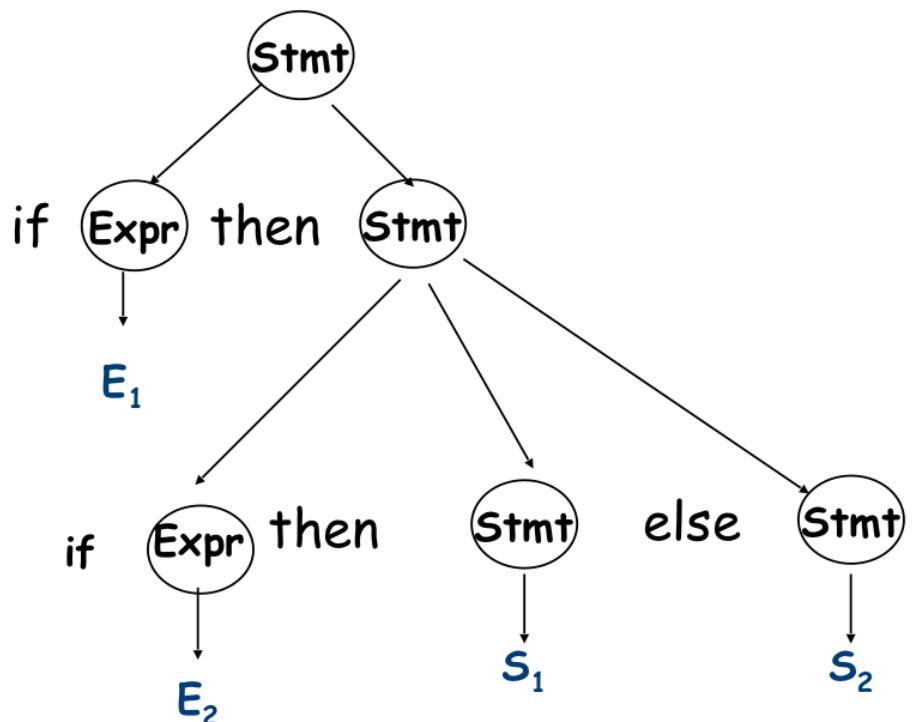
$$\begin{aligned} \text{Stmt} \rightarrow & \text{ if Expr } \underline{\text{then}} \text{ Stmt} \\ | & \text{ if Expr } \underline{\text{then}} \text{ Stmt } \underline{\text{else}} \text{ Stmt} \\ | & \dots \text{ other stmts } \dots \end{aligned}$$

This ambiguity is inherent in the grammar

PRODUCTION 2, THEN 1



PRODUCTION 1, THEN 2



THE PROBLEM IS THAT THE STRUCTURE BUILT BY THE PARSER WILL DETERMINE THE INTERPRETATION OF THE CODE, AND THESE TWO FORMS HAVE DIFFERENT MEANING!

Ambiguity

The grammar forces the structure
to match the desired meaning.

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if (common sense rule)

0	Stmt	\rightarrow	if Expr <u>then</u> Stmt
1			if Expr <u>then</u> WithElse <u>else</u> Stmt
2			Other Statements
3	WithElse	\rightarrow	if Expr <u>then</u> WithElse <u>else</u> WithElse
4			Other Statements

With this grammar, the example has only one rightmost derivation

Intuition: once into WithElse, we cannot generate an unmatched else
... an if without an else can only come through rule 0...

Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

$a = f(17)$

In many Algol-like languages, f could be either a function or a
subscripted variable

USED TO STORE VALUES OF THE
SAME TYPE IN AN ARRAY

Disambiguating this one requires context

- Need values of declarations
- Really an issue of type, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
 - Step outside grammar rather than use a more complex grammar

Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
 - Knowledge of declarations, types, ...
 - Accept a superset of $L(G)$ & check it by other means (Context Sensitive analysis)
 - This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- Parsing techniques that "do the right thing"
- i.e., always select the same derivation

Parsing Techniques

Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” \Rightarrow may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

A top-down parser starts with the root of the parse tree

The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until lower fringe of the parse tree matches the input string

- 1 At a node labeled A , select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
- 2 When a terminal symbol is added to the border and it doesn't match the border, backtrack
- 3 Find the next node to be expanded

(label \in NT)

The key is picking the right production in step 1

- That choice should be guided by the input string

Remember the expression grammar?

We will call this version “the classic expression grammar”

0	Goal	\rightarrow	Expr
1	Expr	\rightarrow	Expr + Term
2			Expr - Term
3			Term
4	Term	\rightarrow	Term * Factor
5			Term / Factor
6			Factor
7	Factor	\rightarrow	(Expr)
8			<u>number</u>
9			<u>id</u>

And the input x - 2 * y

Example

W.i.P DERIVATION

Let's try $\underline{x} - \underline{z} * y$:

Rule	Sentential Form	Input
-	Goal	$\uparrow \underline{x} - \underline{z} * y$

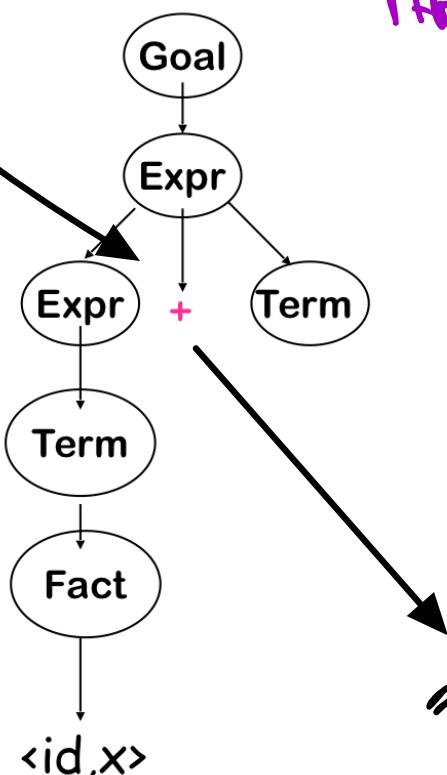
\uparrow is the position in the input buffer



Rule	Sentential Form	Input
-	Goal	$\uparrow x - 2 * y$
0	Expr	$\uparrow x - 2 * y$
1	Expr + Term	$\uparrow x - 2 * y$
3	Term + Term	$\uparrow x - 2 * y$
6	Factor + Term	$\uparrow x - 2 * y$
9	$\langle id, x \rangle + Term$	$\uparrow x - 2 * y$
\rightarrow	$\langle id, x \rangle + Term$	$x \uparrow - 2 * y$

0	Goal	$\rightarrow Expr$
1	Expr	$\rightarrow Expr + Term$
2		$ Expr - Term$
3		$ Term$
4	Term	$\rightarrow Term * Factor$
5		$ Term / Factor$
6		$ Factor$
7	Factor	$\rightarrow (Expr)$
8		$ number$
9		$ id$

TOP-DOWN, FIRST WE TRY
THE PLUS!



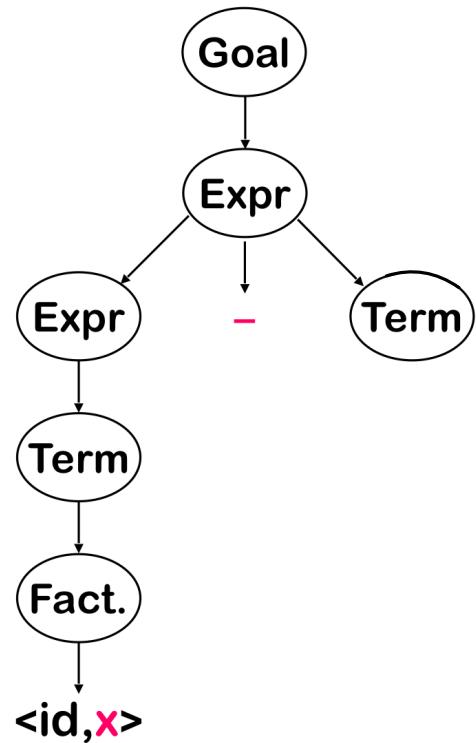
"+" DON'T MATCH "-"
WE HAVE TO BACKTRACK

TO

Rule	Sentential Form	Input
-	Goal	$\uparrow x - \underline{2} * y$
0	Expr	$\uparrow x - \underline{2} * y$
2	Expr - Term	$\uparrow x - \underline{2} * y$
3	Term - Term	$\uparrow x - \underline{2} * y$
6	Factor - Term	$\uparrow x - \underline{2} * y$
9	$\langle id, x \rangle - \text{Term}$	$\uparrow x - \underline{2} * y$
\rightarrow	$\langle id, x \rangle \text{-Term}$	$x \uparrow \underline{2} * y$
\rightarrow	$\langle id, x \rangle \text{-Term}$	$x - \uparrow \underline{2} * y$

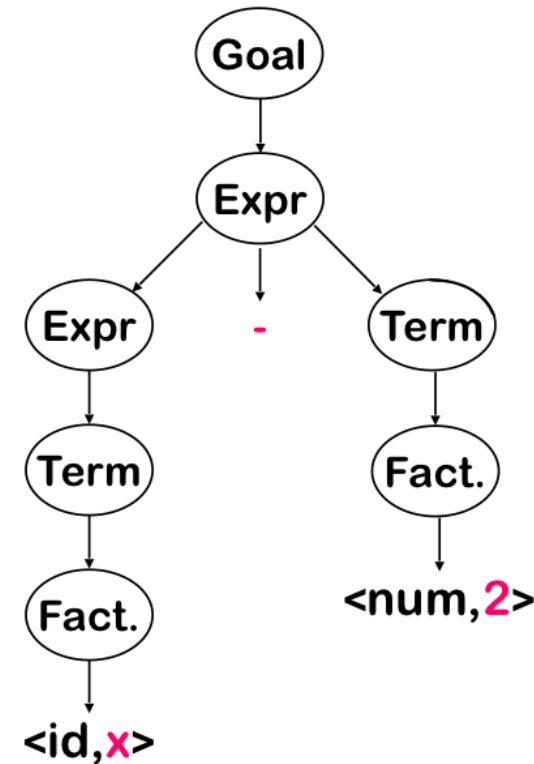
Now, "-" and "-" match

Now we can expand Term to match "2"



Trying to match the "2" in $x - 2 * y$:

Rule	Sentential Form	Input
\rightarrow	$\langle id, \underline{x} \rangle - \text{Term}$	$x - \underline{1} \underline{2} * y$
6	$\langle id, \underline{x} \rangle - \text{Factor}$	$x - \underline{1} \underline{2} * y$
8	$\langle id, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle$	$x - \underline{1} \underline{2} * y$
\rightarrow	$\langle id, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle$	$x - \underline{2} \underline{1} * y$



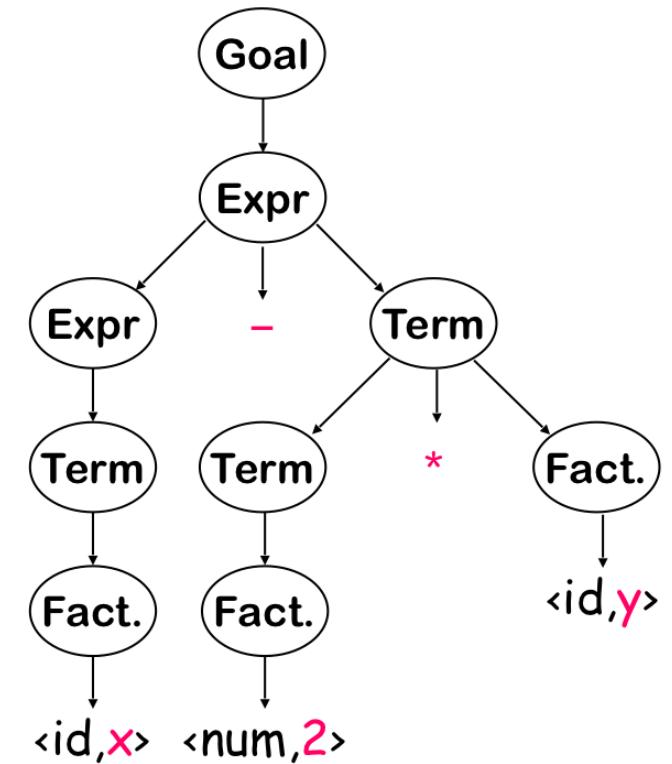
Where are we?

- "2" matches "2"
 - We have more input, but no NTs left to expand
 - The expansion terminated too soon
- ⇒ Need to backtrack

The Point: The parser must make the right choice when it expands a NT. Wrong choices lead to wasted effort.

Trying again with "2" in $x - \underline{2} * y$:

Rule	Sentential Form	Input
\rightarrow	$\langle id, \underline{x} \rangle - \text{Term}$	$x - \underline{1} \underline{2} * y$
4	$\langle id, \underline{x} \rangle - \text{Term} * \text{Factor}$	$x - \underline{1} \underline{2} * y$
6	$\langle id, \underline{x} \rangle - \text{Factor} * \text{Factor}$	$x - \underline{1} \underline{2} * y$
8	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \text{Factor}$	$x - \underline{1} \underline{2} * y$
\rightarrow	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \text{Factor}$	$x - \underline{2} \uparrow * y$
\rightarrow	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \text{Factor}$	$x - \underline{2} * \uparrow y$
9	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \langle id, \underline{y} \rangle$	$x - \underline{2} * \uparrow y$
\rightarrow	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \langle id, \underline{y} \rangle$	$x - \underline{2} * y \uparrow$



This time, we matched & consumed all the input
 \Rightarrow Success!

0	Goal	\rightarrow Expr
1	Expr	\rightarrow Expr + Term
2		Expr - Term
3		Term
4	Term	\rightarrow Term * Factor
5		Term / Factor
6		Factor
7	Factor	\rightarrow (Expr)
8		number
9		id

Another possible parse

Other choices for expansion are possible

Rule	Sentential Form	Input
-	Goal	$\uparrow \underline{x} - 2 * \underline{y}$
0	Expr	$\uparrow \underline{x} - 2 * \underline{y}$
1	Expr + Term	$\uparrow \underline{x} - 2 * \underline{y}$
1	Expr + Term + Term	$\uparrow \underline{x} - 2 * \underline{y}$
1	Expr + Term + Term + Term	$\uparrow \underline{x} - 2 * \underline{y}$
1	And so on	$\uparrow \underline{x} - 2 * \underline{y}$

Consumes no input!

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

The property that we just saw: Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left recursive if $\exists A \in NT$ such that
 \exists a derivation $A \xrightarrow{*} A\alpha$, for some string $\alpha \in (NT \cup T)^*$

Our classic expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- In a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is always a bad property in a compiler

0	Goal	\rightarrow Expr
1	Expr	\rightarrow Expr + Term
2		Expr - Term
3		Term
4	Term	\rightarrow Term * Factor
5		Term / Factor
6		Factor
7	Factor	\rightarrow (Expr)
8		<u>number</u>
9		<u>id</u>

PHYSICALLY ON THE LEFT!

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$\text{Fee} \rightarrow \text{Fee } \alpha \\ | \quad \beta$$

where β does not start with Fee

We can rewrite this fragment as

$$\text{Fee} \rightarrow \beta \text{ Fie} \\ \text{Fie} \rightarrow \alpha \text{ Fie} \\ | \quad \epsilon$$

where Fie is a new non-terminal

"A" IS NEVER ON THE
"LEFT PART" OF THE RIGHT SIDE
OF A PRODUCTION

The new grammar defines
the same language as the
old grammar, using only
right recursion.

Added a reference
to the empty string

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

$\text{Expr} \rightarrow \text{Expr} + \text{Term}$	$\text{Term} \rightarrow \text{Term} * \text{Factor}$
$\text{Expr} - \text{Term}$	$\text{Term} * \text{Factor}$
Term	Factor

Applying the transformation yields

$\text{Expr} \rightarrow \text{Term Expr}'$	$\text{Term} \rightarrow \text{Factor Term}'$
$\text{Expr}' \rightarrow + \text{Term Expr}'$	$\text{Term}' \rightarrow * \text{Factor Term}'$
$- \text{Term Expr}'$	$/ \text{Factor Term}'$
ϵ	ϵ

These fragments use only right recursion

Right recursion often means right associativity. In this case, the grammar does not display any particular associative bias.

Eliminating Left Recursion

Substituting them back into the grammar yields

0	Goal	\rightarrow	Expr
1	Expr	\rightarrow	Term Expr'
2	Expr'	\rightarrow	+ Term Expr'
3			- Term Expr'
4			ϵ
5	Term	\rightarrow	Factor Term'
6	Term'	\rightarrow	* Factor Term'
7			/ Factor Term'
8			ϵ
9	Factor	\rightarrow	(Expr)
10			<u>number</u>
11			<u>id</u>

- This grammar is correct, but somewhat non-intuitive.
- It is left associative, as was the original
 - ⇒ The naive transformation yields a right recursive grammar, which changes the implicit associativity
- A top-down parser will terminate using it.
- even if it may still need to backtrack with it.

Eliminating Left Recursion

The transformation eliminates **immediate** left recursion

What about more general, indirect left recursion ?

The general algorithm:

arrange the NTs into some order A_1, A_2, \dots, A_n

for $i \leftarrow 1$ to n

 for $s \leftarrow 1$ to $i - 1$

 replace each production $A_i \rightarrow A_s \gamma$ with $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$,

 where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current productions for A_s

 eliminate any immediate left recursion on A_i using the direct transformation

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$),
and no epsilon productions

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding A_i has no non-terminal A_s in its rhs, for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion

At the start of the i^{th} outer loop iteration

For all $k < i$, no production that expands A_k contains a non-terminal A_s in its rhs, for $s < k$

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are LL(1) and LR(1) grammars

We start with LL(1) grammars & predictive parsing

LL(k) grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose (between α & β) the right production to expand A in the parser tree at each step

How can it do it?

Guided by the input string!



LL(k) grammars

- An LL(k) grammar is a context-free grammar that can be parsed by predictive parser (no backtracking) which reads the input Left to right and construct a Leftmost derivation looking to k symbols in the input string
- A language that has a LL(k) grammar is said an LL(k) language
- LL(k) is a grammar that can predict the right production to apply with lookahead of most k symbols

$$LL(0) \subset LL(1) \subset LL(2) \subset \dots \subset LL(*)$$

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

The parser will decide what to choose on the base of the input and of the following sets:

- The **FIRST** set: $\text{FIRST}(\alpha)$ with $\alpha \in (T \cup NT)^*$
- The **FOLLOW** set: $\text{FOLLOW}(A)$ with $A \in NT$

The FIRST set

FIRST sets

For some rhs $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \underline{x} \gamma$, for some γ

We will learn how to compute it!

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset \quad \left. \right\} \text{KINDA, SEE NEXT SLIDE}$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

Example

0	Goal	\rightarrow	Expr
1	Expr	\rightarrow	Term Expr'
2	Expr'	\rightarrow	+ Term Expr'
3			- Term Expr'
4			ϵ
5	Term	\rightarrow	Factor Term'
6	Term'	\rightarrow	* Factor Term'
7			/ Factor Term'
8			ϵ
9	Factor	\rightarrow	(Expr)
10			<u>number</u>
11			id

$\text{first}(\text{Expr}') = \{ +, -, \epsilon \}$

But what else I need to consider?

{ eof,) }

Predictive Parsing

What about ϵ -productions?

⇒ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\epsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(A)$, too, where

$\text{FOLLOW}(A)$ = the set of terminal symbols that can immediately follow A in a sentential form

Define $\text{FIRST}^+(A \rightarrow \alpha)$ as

Later we will learn
how to compute them!

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, if $\epsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$, otherwise

A GRAMMAR IS LL(1)



$(A \rightarrow \alpha \wedge A \rightarrow \beta \Rightarrow \text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset)$

FIRST and FOLLOW Sets — REMINDER

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of symbols that appear as the first one in some string that derives from α

That is, $x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$, for some γ

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentential form

$\text{FOLLOW}(S) = \{\text{EOF}\}$, where S is the starting symbol

Recursive Descent Parsing

Recall the expression grammar, after transformation

0	Goal	\rightarrow	Expr
1	Expr	\rightarrow	Term Expr'
2	Expr'	\rightarrow	+ Term Expr'
3			- Term Expr'
4			ϵ
5	Term	\rightarrow	Factor Term'
6	Term'	\rightarrow	* Factor Term'
7			/ Factor Term'
8			ϵ
9	Factor	\rightarrow	(Expr)
10			<u>number</u>
11			<u>id</u>

This produces a parser with six mutually recursive routines:

- Goal
- Expr
- EPrime
- Term
- TPrime
- Factor

Each recognizes one NT or T

The term descent refers to the direction in which the parse tree is built.

Recursive Descent Parsing

(Procedural)

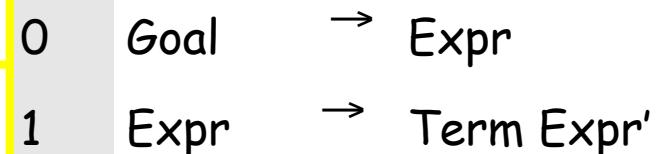
A couple of routines from the expression parser, they return a boolean

Goal()

```
token ← next_token();
if (Expr( ) = true & token = EOF)
    then next compilation step;
else
    report syntax error;
    return false;
```

Expr()

```
if (Term( ) = false)
    then return false;
else return Eprime( );
```



Recursive Descent Parsing II

Eprime()

```
if (token = '+' OR token = '-')
then
    token ← next_token();
    if Term() then return Eprime ();
    else report syntax error;
    end;
else if (token = ')' OR token = EOF )
    then return true;
else return false;
```

2	Expr' → + Term Expr'
3	- Term Expr'
4	ε

$\text{FIRST}^+(\text{Expr}' \rightarrow + \text{Term Expr}') = \{+\}$

$\text{FIRST}^+(\text{Expr}' \rightarrow - \text{Term Expr}') = \{-\}$

$\text{FIRST}^+(\text{Expr}' \rightarrow \varepsilon) = \{\text{EOF}, \text{,}\}$

Term, & *Tprime* follow the same basic lines

Recursive Descent Parsing III

Factor()

```
if (token = Number) then
    token ← next_token();
    return true;
else if (token = Identifier) then
    token ← next_token();
    return true;
else if (token = Lparen)
    token ← next_token();
    if (Expr() = true & token = Rparen) then
        token ← next_token();
        return true;
// fall out of if statement
report syntax error;
return false;
```

9 Factor \rightarrow (Expr)
10 | number
11 | id

FIRST⁺(Factor-> (Expr))={{}
FIRST⁺(Factor-> number)= number}
FIRST⁺(Factor-> id)={id}

looking for Number, Identifier,
or "(", found token instead, or
failed to find Expr or ")" after "("

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
 - Context-free grammars ✓
- Top-down parsers
 - Algorithm & its problem with left recursion ✓
 - Ambiguity ✓
 - Left-recursion removal ✓
- Predictive top-down parsing
 - The LL(1) condition ✓
 - Simple recursive descent parsers ✓
 - Transforming a grammar to be LL(1)
 - First and Follow sets
 - Table-driven LL(1) parsers

What If My Grammar Is Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) grammar?

- In general, the answer is no, however, sometime it is yes

Assume a grammar G with productions $A \rightarrow \alpha \beta_1$ and $A \rightarrow \alpha \beta_2$

- If α derives anything other than ϵ , then

$$\text{FIRST}^+(A \rightarrow \alpha \beta_1) \cap \text{FIRST}^+(A \rightarrow \alpha \beta_2) \neq \emptyset$$

- And the grammar is not LL(1)
- If we pull the common prefix, α , into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha A', A' \rightarrow \beta_1 \text{ and } A' \rightarrow \beta_2$$

Now, if $\text{FIRST}^+(A' \rightarrow \beta_1) \cap \text{FIRST}^+(A' \rightarrow \beta_2) = \emptyset$, G may be LL(1)

For each nonterminal A
find the longest prefix α common to 2 or more alternatives
for A

if $\alpha \neq \epsilon$ then

replace all of the productions

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \alpha \beta_3 \mid \dots \mid \alpha \beta_n \mid \gamma$$

with

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$$

Repeat until no nonterminal has alternative rhs' with a common
prefix

"LEFT FACTORING"

This transformation makes some grammars into LL(1) grammars

There are languages for which no LL(1) grammar exists

FIRST

- We are concerned with $\text{FIRST}(X)$ only for the nonterminals of the grammar
- $\text{FIRST}(X)$ for terminals is trivial
- According to the definition, to determine $\text{FIRST}(A)$, we must inspect all productions that have A on the left

Computing FIRST Sets

For a grammar symbol X , $\text{FIRST}(X)$ is defined as follows.

- For every terminal X , $\text{FIRST}(X) = \{X\}$.
- For every nonterminal X , if $X \rightarrow Y_1Y_2\dots Y_n$ is a production, then
 - $\text{FIRST}(Y_1) \subseteq \text{FIRST}(X)$.
 - Furthermore, if Y_1, Y_2, \dots, Y_k are nullable ($Y_i \xrightarrow{*} \epsilon$) then
$$\text{FIRST}(Y_{k+1}) \subseteq \text{FIRST}(X).$$

Computing FOLLOW Sets

- For a grammar symbol X , $\text{FOLLOW}(X)$ is defined as follows
 - If S is the start symbol, then $\text{EOF} \in \text{FOLLOW}(S)$
 - If $A \rightarrow aB\beta$ is a production, then $\text{FIRST}(\beta) \subseteq \text{FOLLOW}(B)$
 - If $A \rightarrow aB$ is a production, or $A \rightarrow aB\beta$ is a production and β is nullable, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

FOLLOW

- We are concerned about $\text{FOLLOW}(X)$ only for the nonterminals of the grammar.
- According to the definition, to determine $\text{FOLLOW}(A)$, we must inspect all productions that have A on the right.

Building Top-down Parsers for LL(1) Grammars

Given an LL(1) grammar, and its FIRST & FOLLOW sets ...

- Emit a routine for each non-terminal
 - Nest of if-then-else statements to check alternate rhs's
 - Each returns true on success and throws an error on false
 - Simple, working (perhaps ugly) code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
 - Good case statement implementation would be better
- What about a table to encode the options?
 - Interpret the table with a skeleton, as we did in scanning

Building Top-down Parsers

Strategy

- Encode knowledge in a table
- Use a standard “skeleton” parser to interpret the table

Example

- The non-terminal Factor has 3 expansions
 - (Expr) or Identifier or Number
- Table might look like:

		Terminal Symbols								
		+	-	*	/	Id.	Num.	()	EOF
Non-terminal Symbols	Factor	–	–	–	–	10	9	11	–	–

Expand Factor by rule 9 with input “number”

0	Goal	→ Expr
1	Expr	→ Term Expr'
2	Expr'	→ + Term Expr'
3		- Term Expr'
4		ε
5	Term	→ Factor Term'
6	Term'	→ * Factor Term'
7		/ Factor Term'
8		ε
9	Factor	→ (Expr)
10		number
11		id

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need a table-driven interpreter for the table
- Need an algorithm to build the table

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T

	+	-	*	/	Id	Num	()	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

LL(1) Skeleton Parser

```
word ← NextWord()          // Initial conditions, including
push $ onto Stack          // a stack to track the border of the parse tree
push the start symbol, S, onto Stack
TOS ← top of Stack         // we read the first symbol on the top of the stack
loop forever
  if TOS = $ and word = EOF then
    break & report success // exit on success
  else if TOS is a terminal then
    if TOS matches word then
      pop Stack           // recognized TOS
      word ← NextWord()
    else report error looking for TOS // error exit
  else                      // TOS is a non-terminal
    if TABLE[TOS,word] is  $A \rightarrow B_1 B_2 \dots B_k$  then
      pop Stack           // get rid of A
      push  $B_k, B_{k-1}, \dots, B_1$  // in that order
    else break & report error expanding TOS
  TOS ← top of Stack
```

Filling the table

	+	-	*	/	Id	Num	()	EOF
Goal	-	-	-	-	0	0	0	-	-
Expr	-	-	-	-	1	1	1	-	-
Expr'	2	3	-	-	-	-	-	4	4
Term	-	-	-	-	5	5	5	-	-
Term'	8	8	6	7	-	-	-	8	8
Factor	-	-	-	-	10	9	11	-	-

Prod'n	FIRST ⁺	
0	(,id, num)	Goal \rightarrow Expr
1	(,id, num)	Expr \rightarrow Term Expr'
2	+	Expr' \rightarrow + Term Expr'
3	-	Expr' \rightarrow - Term Expr'
4),EOF	Expr' \rightarrow ε
5	(,id, num)	Term \rightarrow Factor Term'
6	*	Term' \rightarrow * Factor Term'
7	/	Term' \rightarrow / Factor Term'
8	+,-,),EOF	Term' \rightarrow ε
9	number	Factor \rightarrow number
10	id	Factor \rightarrow id
11	(Factor \rightarrow (Expr)

Filling in TABLE[X,y], X \in NT, y \in T

1. write the rule $X \rightarrow \beta$, if $y \in \text{FIRST}^+(X \rightarrow \beta)$
2. write error if rule 1 does not apply

If any entry has more than one rule, G is not LL(1)

We call this algorithm the LL(1) table construction algorithm

Bottom-up Parsing

6 7

BOTTOM-UP & LR ARE SKIPPED



Summary

	Advantages	Disadvantages
Top-down Recursive descent, $LL(1)$	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
$LR(1)$	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

Context-sensitive Analysis or Semantic Elaboration

8

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.
Faculty from other educational institutions may use these materials for nonprofit
educational purposes, provided this copyright notice is preserved.

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(int a, int b,int c,int d) {  
    ...  
}  
fee() {  
    int f[3],g[0], h, i, j, k;  
    char *p;  
    fie(h,i,"ab",j, k);  
    k = f * i + j;  
    h = g[17];  
    printf("<%s,%s>.\n",p,q);  
    p = 10;  
}
```

What is wrong with this program?
(let me count the ways ...)

- number of args to fie()
- declared g[0], used g[17]
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are

"deeper than syntax"

To generate code, we need to understand its meaning

Beyond Syntax

These are beyond the expressive power of a CFG

To generate code, the compiler needs to answer many questions

- Is "x" a scalar, an array, or a function? Is "x" declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of "x" does a given use reference?
- Is the expression "x * y + z" type-consistent?
- In "a[i,j,k]", does a have three dimensions?
- Where can "z" be stored? (register, local, global, heap, static)
- In "f ← 15", how should 15 be represented?
- How many arguments does "fie()" take? What about "printf ()" ?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to the same memory location?
- Is "x" defined before it is used?

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
 - Context-sensitive grammars?
 - Attribute grammars
- Use ad-hoc techniques
 - Symbol tables
 - Ad-hoc code (action routines)

In context-sensitive analysis, ad-hoc techniques dominate in practice.

Beyond Syntax

Telling the story

- We will study the formalism – an attribute grammar
 - Clarify many issues in a succinct and immediate way
 - Separate analysis problems from their implementations
- We will see that the problems with attribute grammars motivate actual, ad-hoc practice
 - Non-local computation
 - Need for centralised information

We will cover attribute grammars, then move on to ad-hoc ideas

When?

- These kind of analyses are either performed together with parsing or in a post-pass that traverses the IR produced by the parser

Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of rules computing values
- Each symbol in the derivation (or parse tree) has a set of named values, or attributes
- The rules specify how to compute a value for each attribute
 - Attribution rules are functional; they uniquely define the value
 - Each attribute is defined by rules that can refer to the values of all the other attributes in the production (**local information**)

Example

1	<i>Number</i>	\rightarrow	<i>Sign List</i>
2	<i>Sign</i>	\rightarrow	+
3			-
4	<i>List</i>	\rightarrow	<i>List Bit</i>
5			<i>Bit</i>
6	<i>Bit</i>	\rightarrow	0
7			1

This grammar defines
signed binary numbers
e.g., -10010 or +00101

Examples

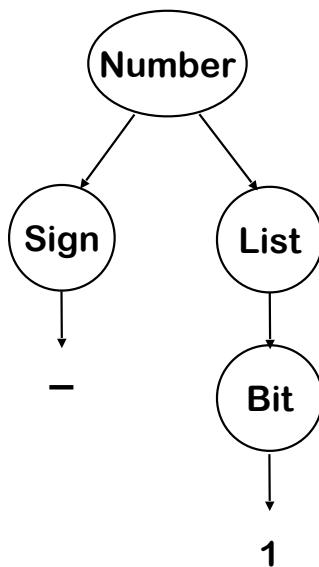
For “-1”

Number → Sign List

→ Sign Bit

→ Sign 1

→ - 1



For “-101”

Number → Sign List

→ Sign List Bit

→ Sign List 1

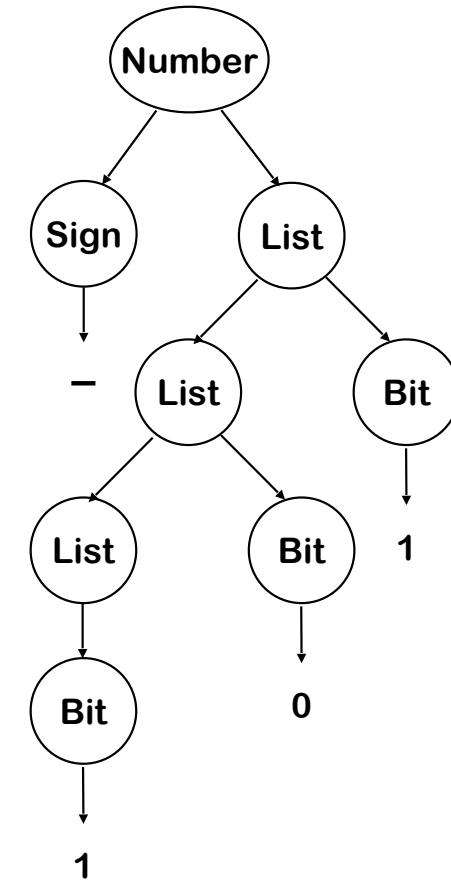
→ Sign List Bit 1

→ Sign List 0 1

→ Sign Bit 0 1

→ Sign 1 0 1

→ - 101



We will use these two examples throughout the lecture

Attribute Grammars

1	Number	\rightarrow	Sign List
2	Sign	\rightarrow	+
3			-
4	List	\rightarrow	List Bit
5			Bit
6	Bit	\rightarrow	0
7			1

- We would like to augment it with rules that defines an attribute containing the decimal value of each valid input string:

-10010 \rightarrow -18 +00101 \rightarrow +5

- For this we consider the following attributes

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Attribute Grammars

Add rules to compute the decimal value of a signed binary number

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Productions	Attribution Rules
$\text{Number} \rightarrow \text{Sign } \text{List}$	$\text{List}.pos \leftarrow 0$ if $\text{Sign}.neg$ then $\text{Number}.val \leftarrow -\text{List}.val$ else $\text{Number}.val \leftarrow \text{List}.val$
$\text{Sign} \rightarrow +$ -	$\text{Sign}.neg \leftarrow \text{false}$ $\text{Sign}.neg \leftarrow \text{true}$
$\text{List}_0 \rightarrow \text{List}_1 \text{ Bit}$ Bit	$\text{List}_1.pos \leftarrow \text{List}_0.pos + 1$ $\text{Bit}.pos \leftarrow \text{List}_0.pos$ $\text{List}_0.val \leftarrow \text{List}_1.val + \text{Bit}.val$ $\text{Bit}.pos \leftarrow \text{List}.pos$ $\text{List}.val \leftarrow \text{Bit}.val$
$\text{Bit} \rightarrow 0$ 1	$\text{Bit}.val \leftarrow 0$ $\text{Bit}.val \leftarrow 2^{\text{Bit}.pos}$

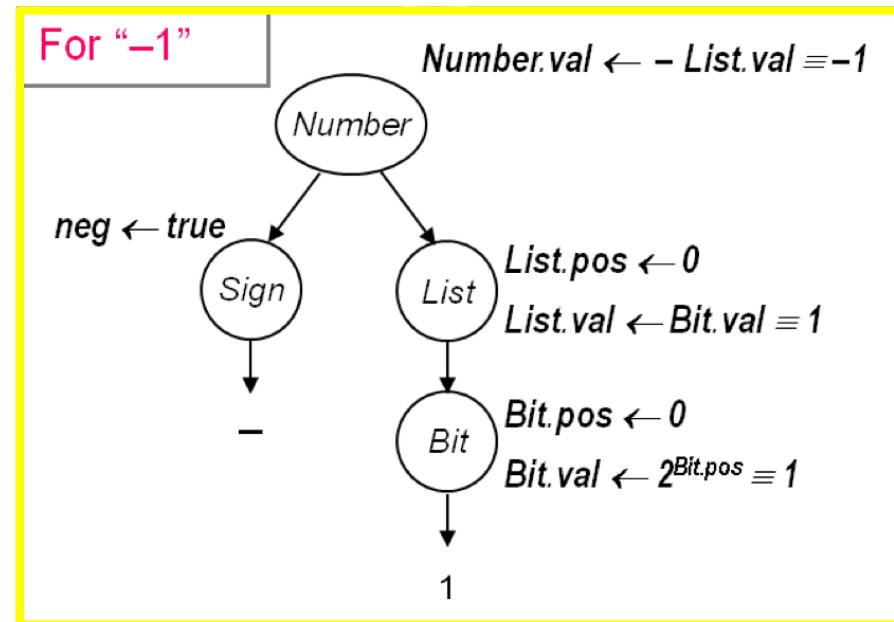
Note: for some rules the information flows from left to right

for some rules the information flows from right to left

Back to the Examples

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

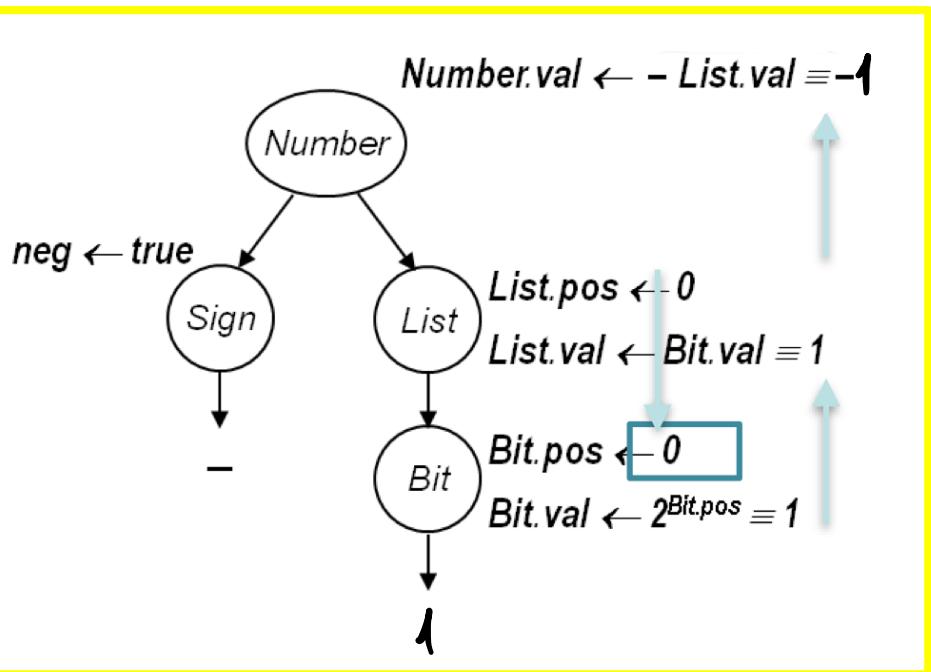
Productions		Attribution Rules
Number	\rightarrow Sign List	$List.pos \leftarrow 0$ if Sign.neg then Number.val $\leftarrow -List.val$ else Number.val $\leftarrow List.val$
Sign	\rightarrow + -	Sign.neg $\leftarrow \text{false}$ Sign.neg $\leftarrow \text{true}$
List ₀	\rightarrow List ₁ Bit	List ₁ .pos $\leftarrow List_0.pos + 1$ Bit.pos $\leftarrow List_0.pos$ List ₀ .val $\leftarrow List_1.val + Bit.val$
	Bit	Bit.pos $\leftarrow List.pos$ List.val $\leftarrow Bit.val$
Bit	\rightarrow 0 1	Bit.val $\leftarrow 0$ Bit.val $\leftarrow 2^{Bit.pos}$



Evaluation order



RULES + PARSE TREE IMPLY AN ATTRIBUTE DEPENDENCE GRAPH



One possible evaluation order:

- 1 List.pos
- 2 Sign.neg
- 3 Bit.pos
- 4 Bit.val
- 5 List.val
- 6 Number.val

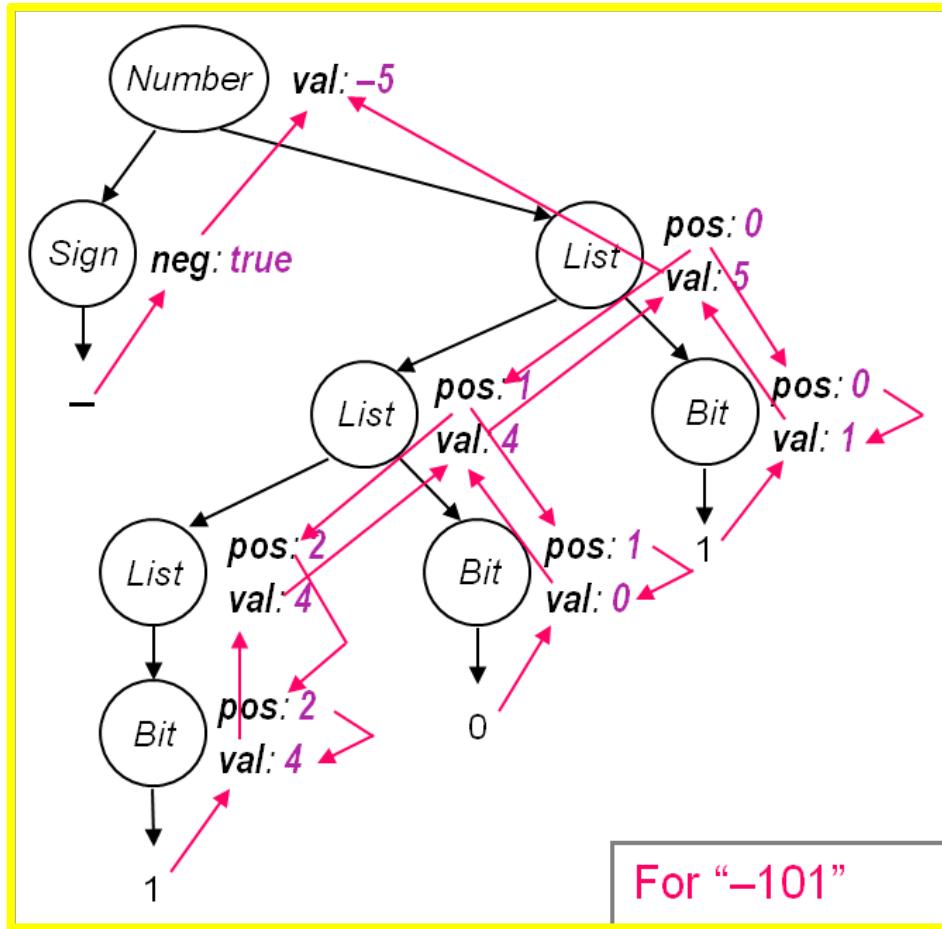
Other orders are possible

THE EVALUATION ORDER MUST BE CONSISTENT WITH THE ATTRIBUTE DEPENDENCE GRAPH

Knuth suggested a data-flow model for evaluation

- Independent attributes first
- Others in order as input values become available

Back to the Examples



This is the complete attribute dependence graph for “-101”.

It shows the flow of all attribute values in the example.

Some flow downward

→ inherited attributes

Some flow upward

→ synthesized attributes

A rule may use attributes in the parent, children, or siblings of a node

The Rules of the Game

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Attribute is defined once, using local information
- Rules & parse tree define an attribute dependence graph
 - Graph must be non-circular

This produces a high-level, functional specification

We need an attributed grammar evaluator

N.B.: AG is a specification
for the computation, not an
algorithm

Using Attribute Grammars

Attribute grammars can specify context-sensitive actions

- Take values from syntax
- Perform computations with values
- Insert tests, logic, ...

Synthesized Attributes

- Use values from children & from constants
- S-attributed grammars
- Evaluate in a single bottom-up pass

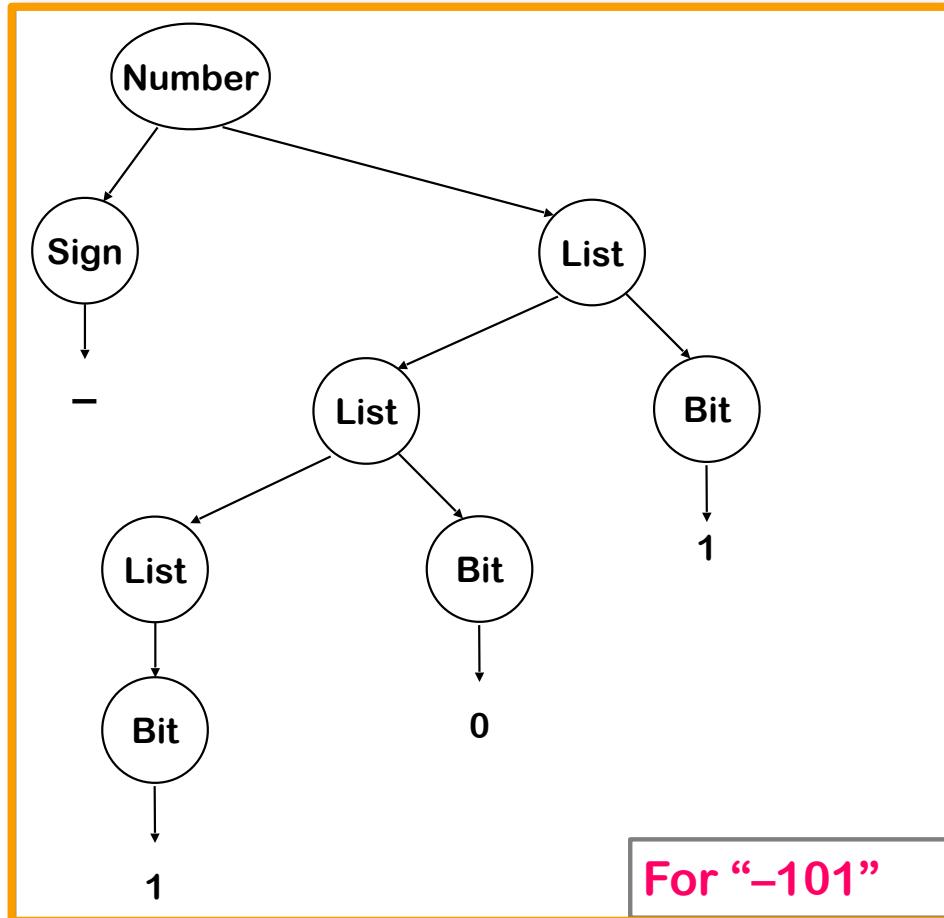
Good match to LR parsing

Inherited Attributes

- Use values from parent, constants, & siblings
- Thought to be more natural
Not easily done at parse time

We want to use both kinds of attributes

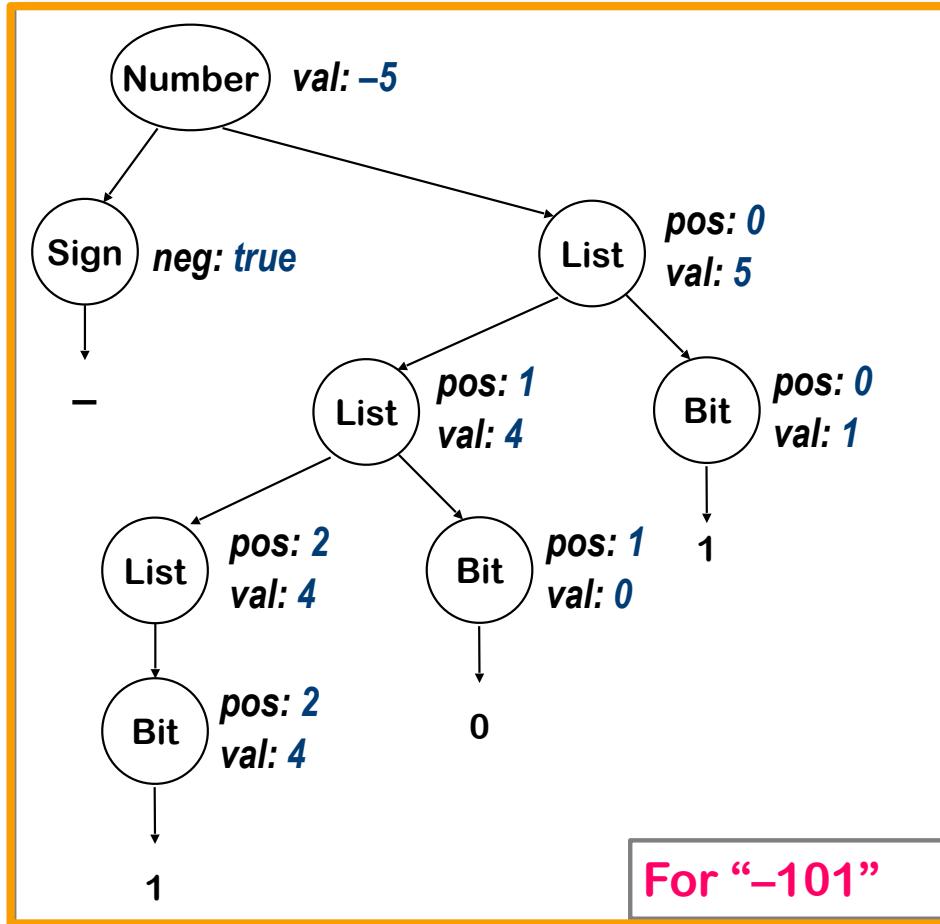
Back to the Example



Syntax Tree

For “ -101 ”

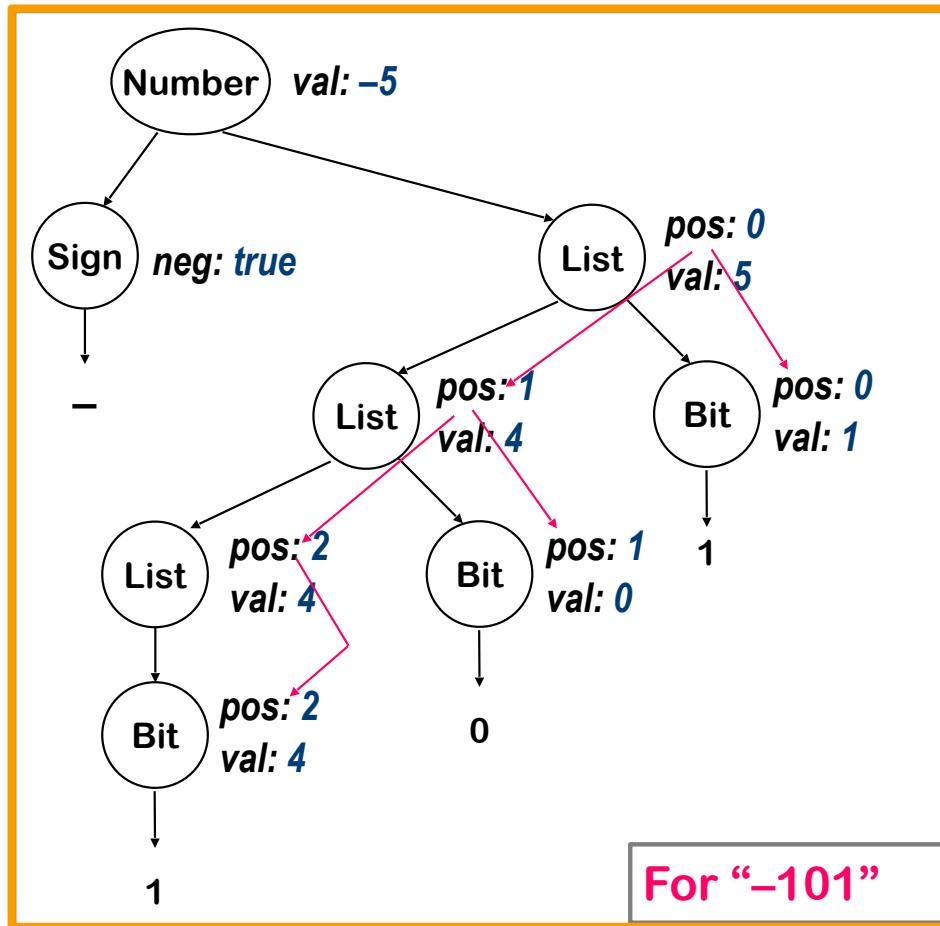
Back to the Example



Attributed Syntax Tree

For “-101”

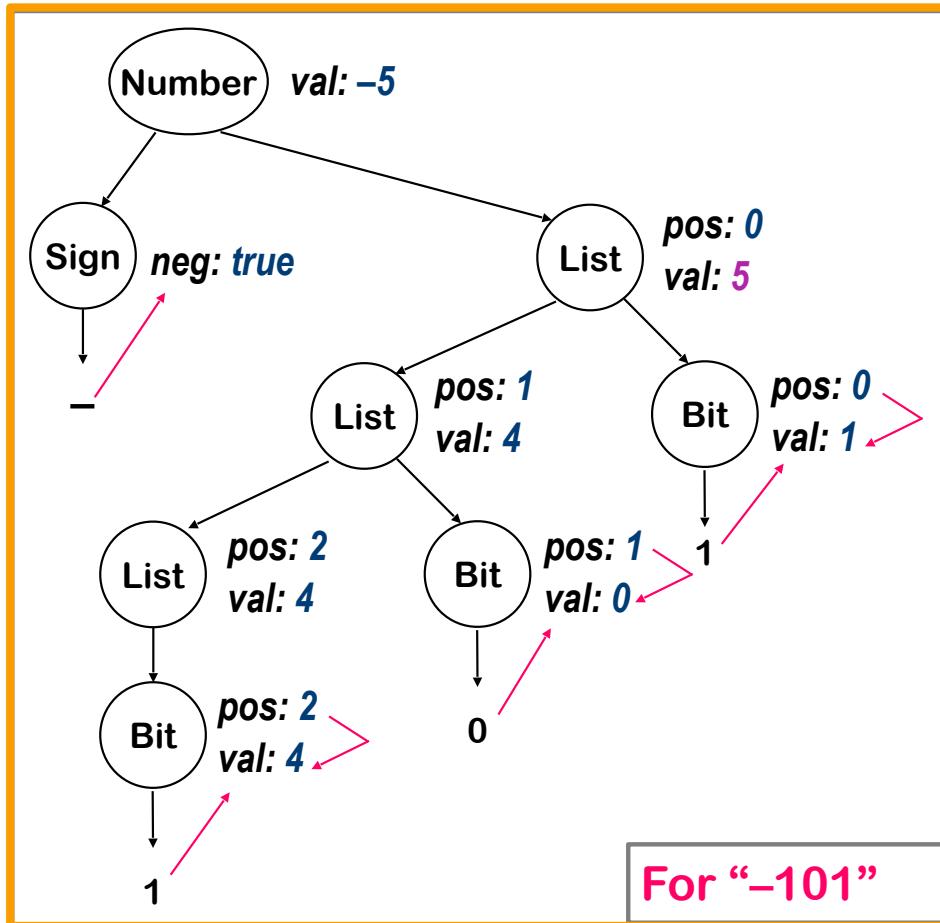
Back to the Example



Inherited Attributes

For -101

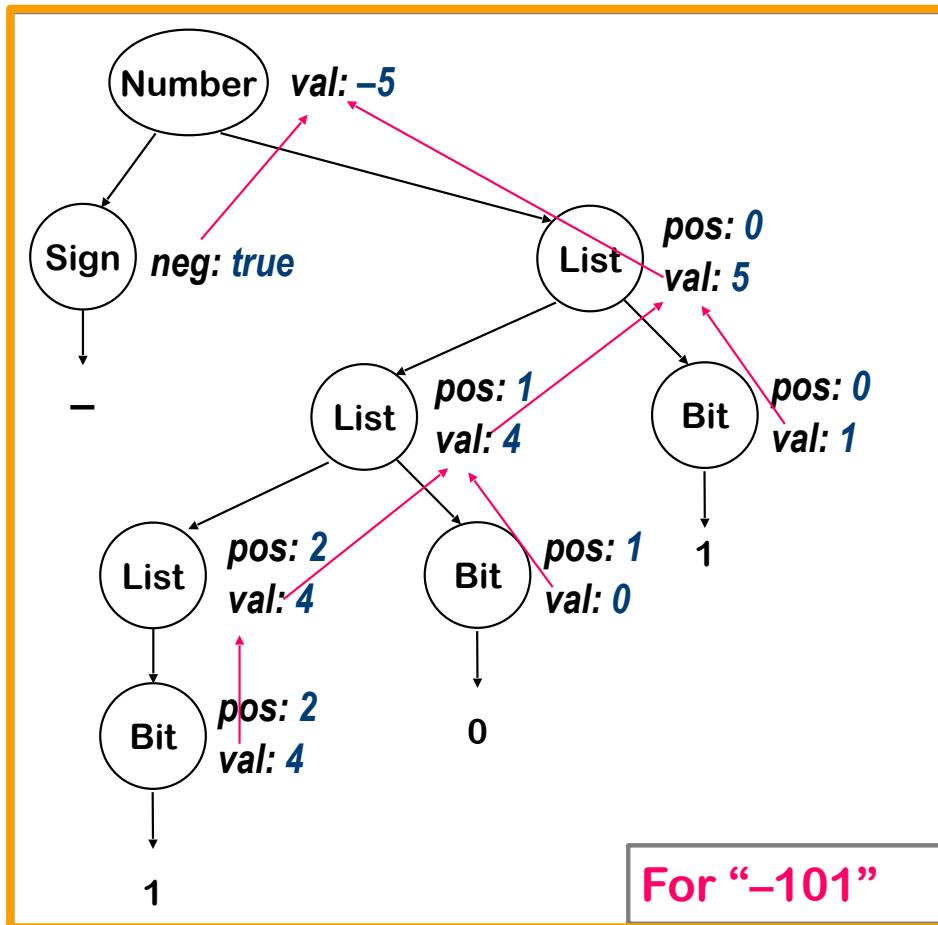
Back to the Example



Synthesized attributes

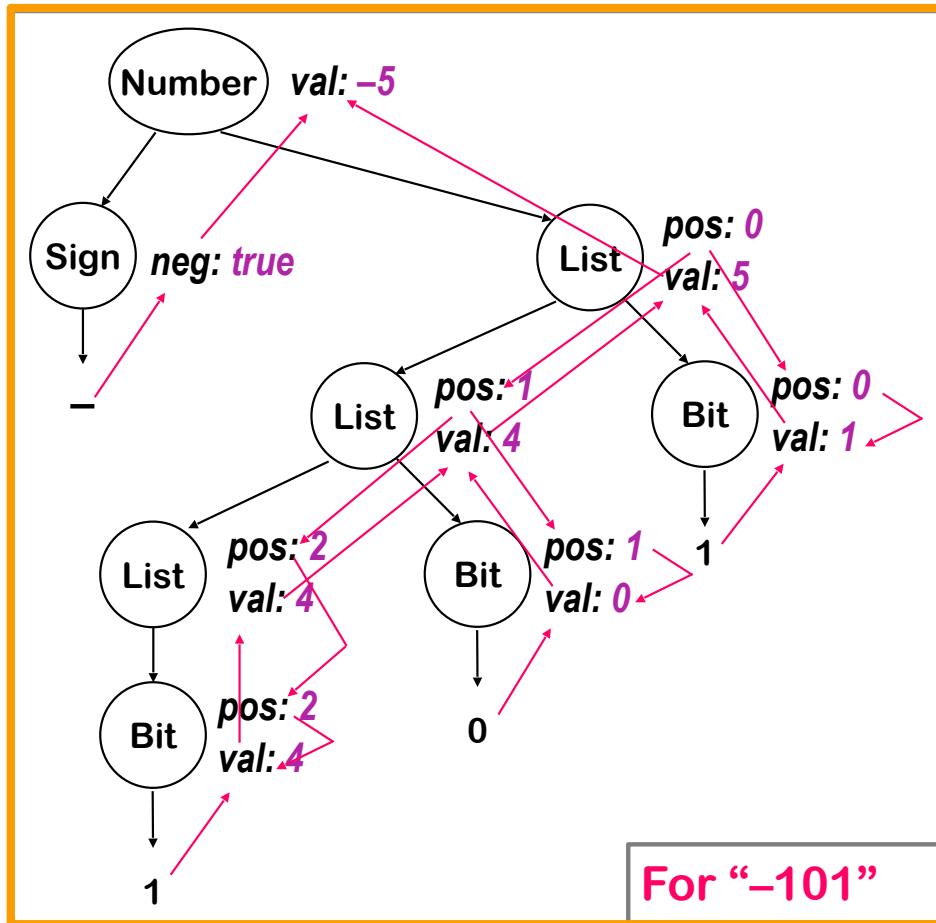
Val draws from children & the same node.

Back to the Example



More Synthesized attributes

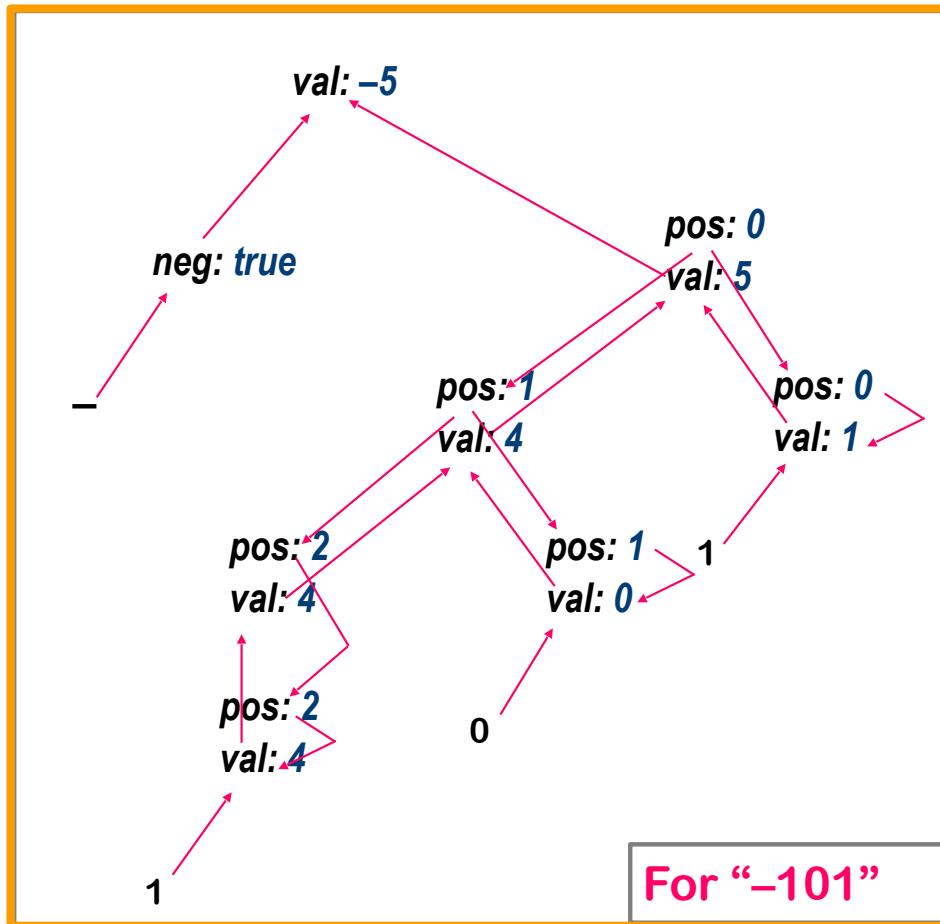
Back to the Example



If we show the computation ...

& then peel away the parse tree ...

Back to the Example



All that is left is the **attribute dependence graph**.

This succinctly represents the flow of values in the problem instance.

The dynamic methods sort this graph to find independent values, then work along graph edges.

The rule-based methods try to discover “good” orders by analyzing the rules.

The oblivious methods ignore the structure of this graph.

The dependence graph **must** be acyclic

Circularity

We can only evaluate acyclic instances

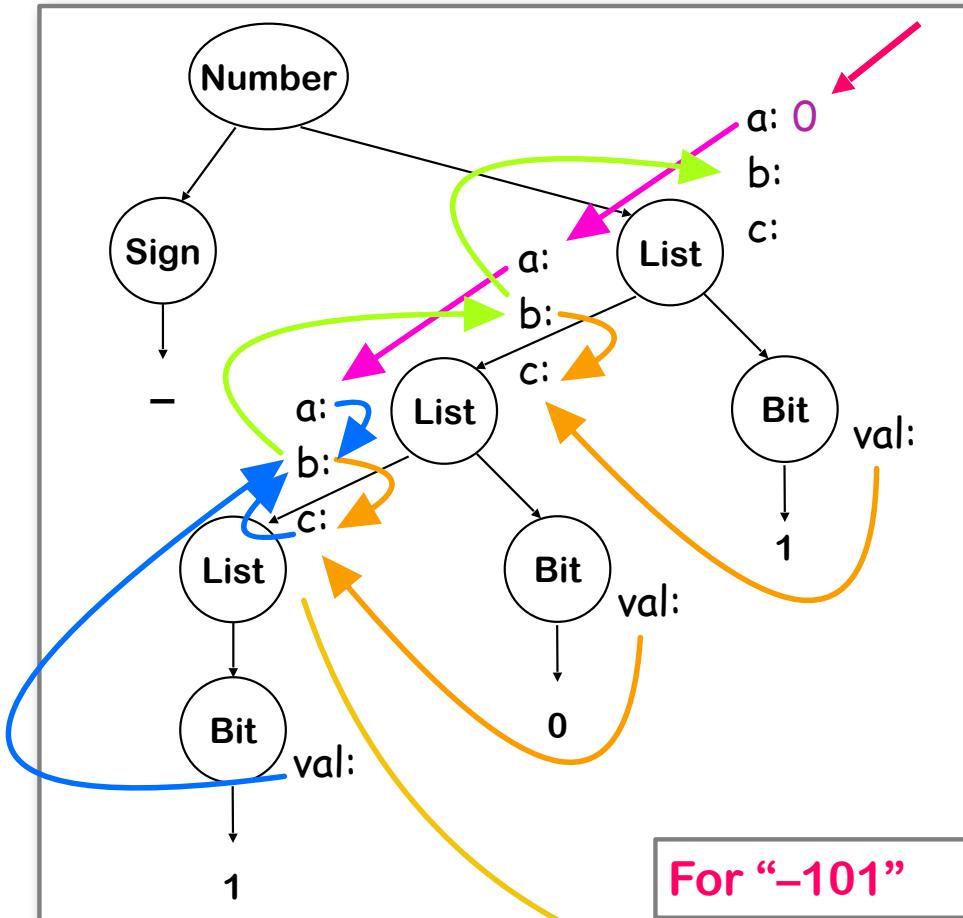
- General circularity testing problem is inherently exponential!
 - We can prove that some grammars can only generate instances with acyclic dependence graphs
 - Largest such class is “strongly non-circular” grammars (SNC)
 - SNC grammars can be tested in polynomial time
 - Failing the SNC test is not conclusive (sufficient conditions)
 - Many evaluation methods discover circularity dynamically
- ⇒ Bad property for a compiler to have

A Circular Attribute Grammar

Productions	Attribution Rules	
Number \rightarrow List	List.a $\leftarrow 0$	
List ₀ \rightarrow List ₁ Bit	List ₁ .a \leftarrow List ₀ .a + 1	
	List ₀ .b \leftarrow List ₁ .b	
	List ₁ .c \leftarrow List ₁ .b + Bit.val	
Bit	List ₀ .b \leftarrow List ₀ .a + List ₀ .c + Bit.val	
Bit \rightarrow 0	Bit.val $\leftarrow 0$	
1	Bit.val $\leftarrow 1$	

Remember, the circularity is in the attribution rules, not the underlying CFG

Circular Grammar Example



Productions	Attribution Rules
$\text{Number} \rightarrow \text{List}$	$\text{List}.a \leftarrow 0$ 1
$\text{List}_0 \rightarrow \text{List}_1$	$\text{List}_1.a \leftarrow \text{List}_0.a + 1$
Bit	$\text{List}_0.b \leftarrow \text{List}_1.b$ 3
	$\text{List}_1.c \leftarrow \text{List}_1.b + \text{Bit.val}$ 4
	$\text{List}_0.b \leftarrow \text{List}_0.a + \text{List}_0.c + \text{Bit.val}$ 5
$\text{Bit} \rightarrow 0$	$\text{Bit.val} \leftarrow 0$
$\text{Bit} \rightarrow 1$	$\text{Bit.val} \leftarrow 1$

HERE'S THE CYCLE

Circularity – The Point

- Circular grammars have indeterminate values
 - Algorithmic evaluators will fail
- Noncircular grammars evaluate to a unique set of values

⇒ Should (undoubtedly) use provably noncircular grammars

Remember, we are studying AGs to gain insight

- We should avoid circular, indeterminate computations
- If we stick to provably noncircular schemes, evaluation should be easier

Another Example on Attribute Grammar

Grammar for a basic block

```
1   $Block_0 \rightarrow Block_1 \text{ Assign}$ 
2          |  $\text{Assign}$ 
3   $Assign_0 \rightarrow Ident = Expr;$ 
4   $Expr_0 \rightarrow Expr_1 + Term$ 
5          |  $Expr_1 - Term$ 
6          |  $Term$ 
7   $Term_0 \rightarrow Term_1 * Factor$ 
8          |  $Term_1 / Factor$ 
9          |  $Factor$ 
10  $Factor \rightarrow ( Expr )$ 
11          |  $Number$ 
12          |  $Ident$ 
```

Let's estimate cycle counts

- Each **operation** has a **COST**
- Assume a **load** per value that has a **COST**
- **Add them, bottom up**
- Assume no reuse

Simple problem for an AG

An Extended Example

(continued)

1	$Block_0 \rightarrow Block_1 \text{ Assign}$	$Block_0.\text{cost} \leftarrow Block_1.\text{cost} + \text{Assign}.\text{cost}$
2	Assign	$Block_0.\text{cost} \leftarrow \text{Assign}.\text{cost}$
3	$Assign_0 \rightarrow \text{Ident} = \text{Expr};$	$\text{Assign}.\text{cost} \leftarrow \text{COST(store)} + \text{Expr}.\text{cost}$
4	$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.\text{cost} \leftarrow Expr_1.\text{cost} + \text{COST(add)} + \text{Term}.\text{cost}$
5	$Expr_1 - Term$	$Expr_0.\text{cost} \leftarrow Expr_1.\text{cost} + \text{COST(sub)} + \text{Term}.\text{cost}$
6	$Term$	$Expr_0.\text{cost} \leftarrow \text{Term}.\text{cost}$
7	$Term_0 \rightarrow Term_1 * Factor$	$Term_0.\text{cost} \leftarrow Term_1.\text{cost} + \text{COST(mult)} + \text{Factor}.\text{cost}$
8	$Term_1 / Factor$	$Term_0.\text{cost} \leftarrow Term_1.\text{cost} + \text{COST(div)} + \text{Factor}.\text{cost}$
9	$Factor$	$Term_0.\text{cost} \leftarrow \text{Factor}.\text{cost}$
10	$Factor \rightarrow (\text{Expr})$	$Factor.\text{cost} \leftarrow \text{Expr}.\text{cost}$
11	Number	$Factor.\text{cost} \leftarrow \text{COST(loadI)}$
12	Ident	$Factor.\text{cost} \leftarrow \text{COST(load)}$

These are all synthesized attributes!

Values flow from rhs to lhs in prod'ns

An Extended Example

(continued)

Properties of the example grammar

- All attributes are synthesized \Rightarrow **S-attributed grammar**
- Rules can be evaluated bottom-up in a single pass
 - Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement? $x=y+y$

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded

An Extended Example

- We would like something like

```
if ( name has not been loaded )  
    then Factor.cost ← Cost(load);  
else Factor.cost ← 0;
```

Non local information!

- to realize it we consider two attributes **before** and **after** that contains set of **names**
 - **before** contains the set of all **names** that occur earlier in the **block**
 - **after** contain all names in **before** plus any **name** that was loaded in the subtree rooted at that node

A Better Execution Model

Adding load tracking

- Need sets Before and After for each production
- Must be initialized, updated, and passed around the tree

```
10 Factor → ( Expr )      Factor.cost ← Expr.cost  
                           Expr.before ← Factor.before  
                           Factor.after ← Expr.after  
11          | Number       Factor.cost ← COST(loadI)  
                           Factor.after ← Factor.before  
12          | Ident        If (Ident.name ∉ Factor.before)  
                           then  
                               Factor.cost ← COST(load)  
                               Factor.after ← Factor.before  
                                   ∪ {Ident.name}  
                           else  
                               Factor.cost ← 0  
                               Factor.after ← Factor.before
```

This version is much more complex

A Better Execution Model

- Load tracking adds complexity
- But, most of it is in the “copy rules”
- Every production needs rules to copy Before & After

A sample production

4	$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$
		$Expr_1.before \leftarrow Expr_0.before$
		$Term.before \leftarrow Expr_1.before$
		$Expr_0.after \leftarrow Term.after$

These copy rules multiply rapidly

Each creates an instance of the set

Lots of work, lots of space, lots of rules to write

A second example: inferring expression types

- Any compiler that tries to generate efficient code for a typed language must confront the problem of inferring types for every expression in the program
- This relies on context-sensitive information: the type of `name` or of a `num` depends on its identity rather than its syntactic category

Type inference for expressions

Assume

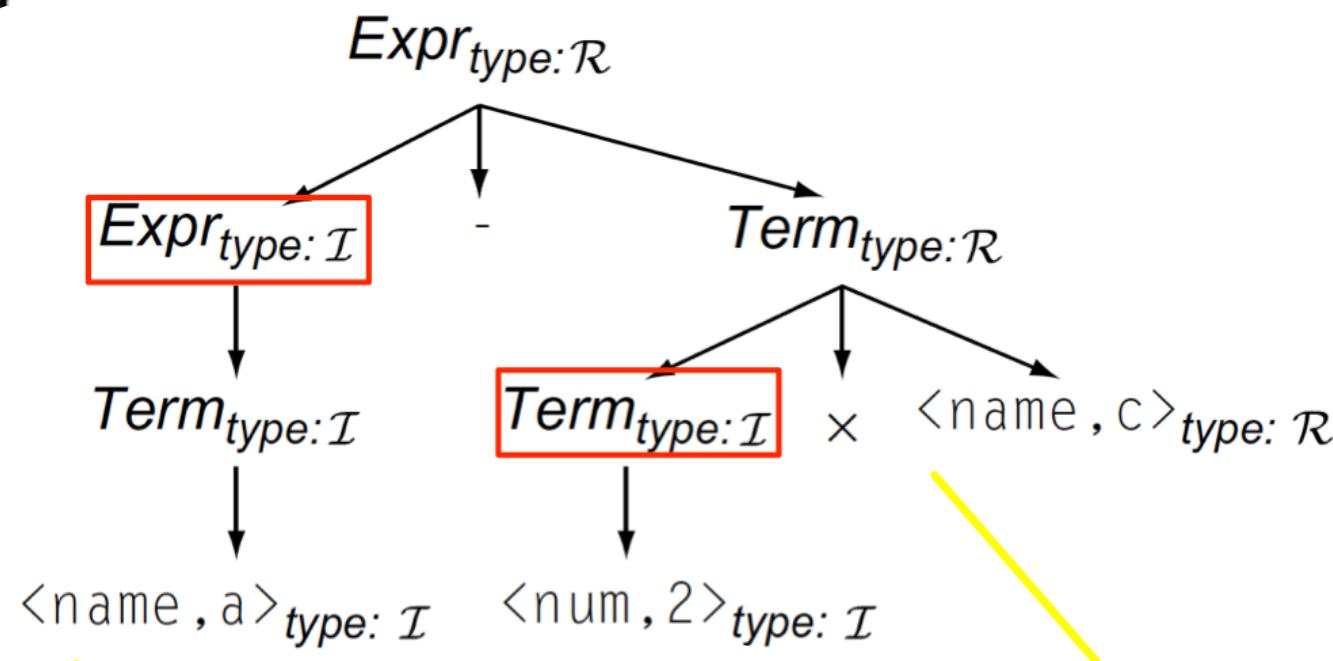
- name and num that appear in the parse tree has already an attribute type
- $\mathcal{F}_+ \mathcal{F}_- \mathcal{F}_x \mathcal{F}_\div$ encode information as the one for + in this table

+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	<i>illegal</i>
complex	complex	complex	<i>illegal</i>	complex

The attribute Grammar

Production	Attribution Rules
$Expr_0 \rightarrow Expr_1 + Term$ $Expr_1 - Term$ $Term$	$Expr_0.type \leftarrow \mathcal{F}_+(Expr_1.type, Term.type)$ $Expr_0.type \leftarrow \mathcal{F}_-(Expr_1.type, Term.type)$ $Expr_0.type \leftarrow Term.type$
$Term_0 \rightarrow Term_1 Factor$ $Term_1 Factor$ $Factor$	$Term_0.type \leftarrow \mathcal{F}_x(Term_1.type, Factor.type)$ $Term_0.type \leftarrow \mathcal{F}_\div(Term_1.type, Factor.type)$ $Term_0.type \leftarrow Factor.type$
$Factor \rightarrow (Expr)$ num $name$	$Factor.type \leftarrow Expr.type$ $num.type$ is already defined $name.type$ is already defined

$a - 2 \times c$



+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	illegal
complex	complex	complex	illegal	complex

Production	Attribution Rules
$Expr_0 \rightarrow Expr_1 + Term$ $ Expr_1 - Term$ $ Term$	$Expr_0.type \leftarrow \mathcal{F}_+(Expr_1.type, Term.type)$ $Expr_0.type \leftarrow \mathcal{F}_-(Expr_1.type, Term.type)$ $Expr_0.type \leftarrow Term.type$
$Term_0 \rightarrow Term_1 Factor$ $ Term_1 Factor$ $ Factor$	$Term_0.type \leftarrow \mathcal{F}_x(Term_1.type, Factor.type)$ $Term_0.type \leftarrow \mathcal{F}_÷(Term_1.type, Factor.type)$ $Term_0.type \leftarrow Factor.type$
$Factor \rightarrow (Expr)$ $ num$ $ name$	$Factor.type \leftarrow Expr.type$ $num.type \text{ is already defined}$ $name.type \text{ is already defined}$

For each case the operand will have a different type from the type of the other operand the compiler need to add a conversion

Type inference for expressions

- We have assumed that `name.type` and `num.type` were already defined
- but to fill those values using an attribute grammar the compiler writer would need to develop a set of rules for the portion of the grammar that handle declarations, to collect this information and to add attributes for propagate that information on all variables: **many copy rules!**
- at the leaf node the rules need to extract the appropriate facts
The result set of rules would be similar the one of the previous example

Problems with Attribute-Grammar Approach

- Attribute grammars handle well problems where all information flows in the same direction and is local
- There is a problem in handling non local information
- Non-local computation need a lots of supporting rules
 - Copy rules increase cognitive overhead
 - Copy rules increase space requirements
 - Need copies of attributes
- Result is an attributed tree
 - Must build the parse tree
 - All the answer are in the values of the attributed tree. To find them later phases has either visit the tree for answers or copy relevant information in the root (more copy rules)

To solve the Problems

- Drop the functional approach of the rules
- Add a central repository for attributes
- An attribute rule can write or read from a global table: it can access to non-local information

The Realist's Alternative

Ad-hoc syntax-directed translation

- Build on the grammar as attribute grammar
- Associate a snippet (action) of code with each production
- If you have a descendent parser call a procedure at each parsing routine
- In the bottom up parser, for each reduction, the corresponding snippet runs (in the next slides assume a bottom up parser!)

Reworking the Example

The variable **cost** is **global!**

1	Block_0	$\rightarrow \text{Block}_1 \text{ Assign}$	
2		Assign	
3	Assign_0	$\rightarrow \text{Ident} = \text{Expr};$	$\text{cost} \leftarrow \text{cost} + \text{COST(store)}$
4	Expr_0	$\rightarrow \text{Expr}_1 + \text{Term}$	$\text{cost} \leftarrow \text{cost} + \text{COST(add)}$
5		$\text{Expr}_1 - \text{Term}$	$\text{cost} \leftarrow \text{cost} + \text{COST(sub)}$
6		Term	
7	Term_0	$\rightarrow \text{Term}_1 * \text{Factor}$	$\text{cost} \leftarrow \text{cost} + \text{COST(mult)}$
8		$\text{Term}_1 / \text{Factor}$	$\text{cost} \leftarrow \text{cost} + \text{COST(div)}$
9		Factor	
10	Factor	$\rightarrow (\text{Expr})$	
11		Number	$\text{cost} \leftarrow \text{cost} + \text{COST(loadI)}$
12		Ident	$i \leftarrow \text{hash}(\text{Ident});$ $\text{if } (\text{Table}[i].\text{loaded} = \text{false})$ $\text{then } \{$ $\text{cost} \leftarrow \text{cost} + \text{COST(load)}$ $\text{Table}[i].\text{loaded} \leftarrow \text{true}$ $\}$

This looks cleaner
& simpler than the
AG!

One missing detail:
initializing cost

Reworking the Example (with load tracking)

0	<i>Start</i>	<i>Init Block</i>
.5	<i>Init</i>	ε
1	$Block_0$	$\rightarrow Block_1 \text{ Assign}$
2		<i>Assign</i>
3	$Assign_0$	$\rightarrow Ident = Expr; \quad cost \leftarrow cost + COST(store)$

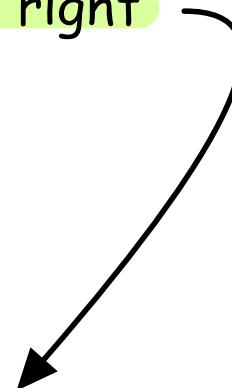
and so on as shown on previous slide...

- Before parser can reach *Block*, it must reduce *Init*
- Reduction by *Init* sets *cost* to zero

We split the production to create a reduction in the middle – for the sole purpose of hanging an action there. This trick is often used.

To make this work

- Need names for attributes of each symbol on lhs & rhs
 - Yacc introduced \$\$, \$1, \$2, ... \$n, left to right
- Need an evaluation scheme
 - Fits nicely into LR(1) parsing algorithm



- \$\$ IS THE "RESULT"
- $\frac{\text{EXPR} \times \text{TERM}}{\$1 \ \$2 \ \$3 \dots}$

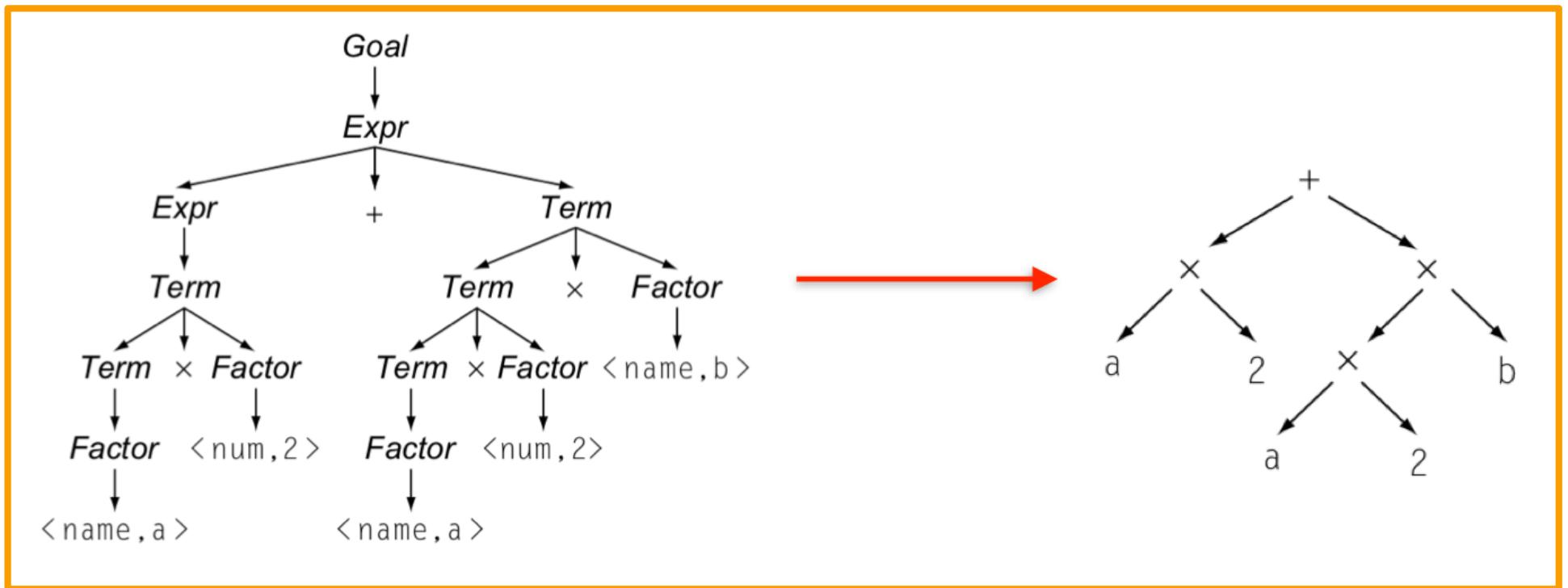
Different kinds of Intermediate Representations

Three major categories

- Structural
 - Graphically oriented
 - Heavily used in source-to-source translators
 - Tend to be large
- Linear
 - Pseudo-code for an abstract machine
 - Level of abstraction varies
 - Simple, compact data structures
 - Easier to rearrange
- Hybrid
 - Combination of graphs and linear code

Intermediate representations: Abstract syntax tree

- Abstract syntax tree: retains the essential structure of the parse tree but eliminates the non-terminal nodes



Intermediate representations: Linear IR

- Linear code: sequence of instructions that execute in their order of appearance

```
push 2  
push b  
multiply  
push a  
subtract
```

Stack-Machine Code

```
t1 ← 2  
t2 ← b  
t3 ← t1 × t2  
t4 ← a  
t5 ← t4 - t3
```

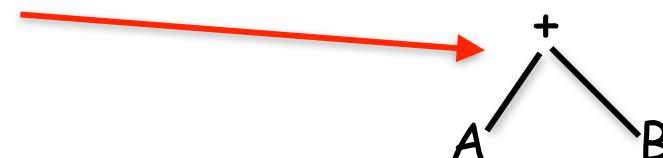
Three-Address Code

- In your book ILOC is an example of three-address code

Building an Abstract Syntax Tree

Assume the following 4 routines :

- MakeAddNode (A, B)
- MakeSubNode (A, B)
- MakeDivNode (A, B)
- MakeMulNode (A, B)



and

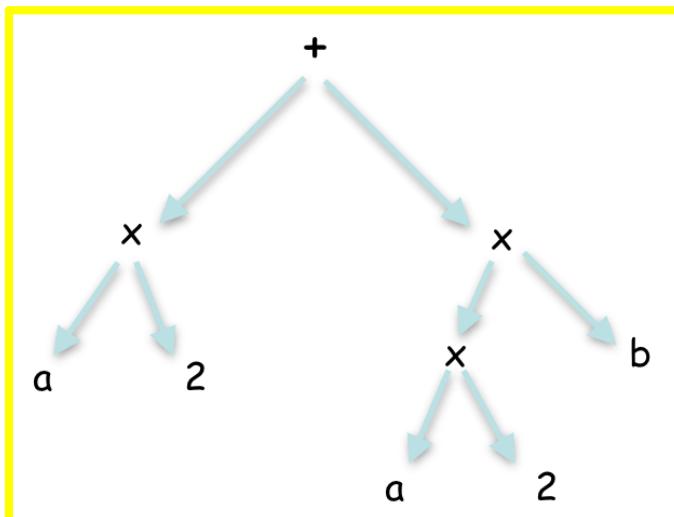
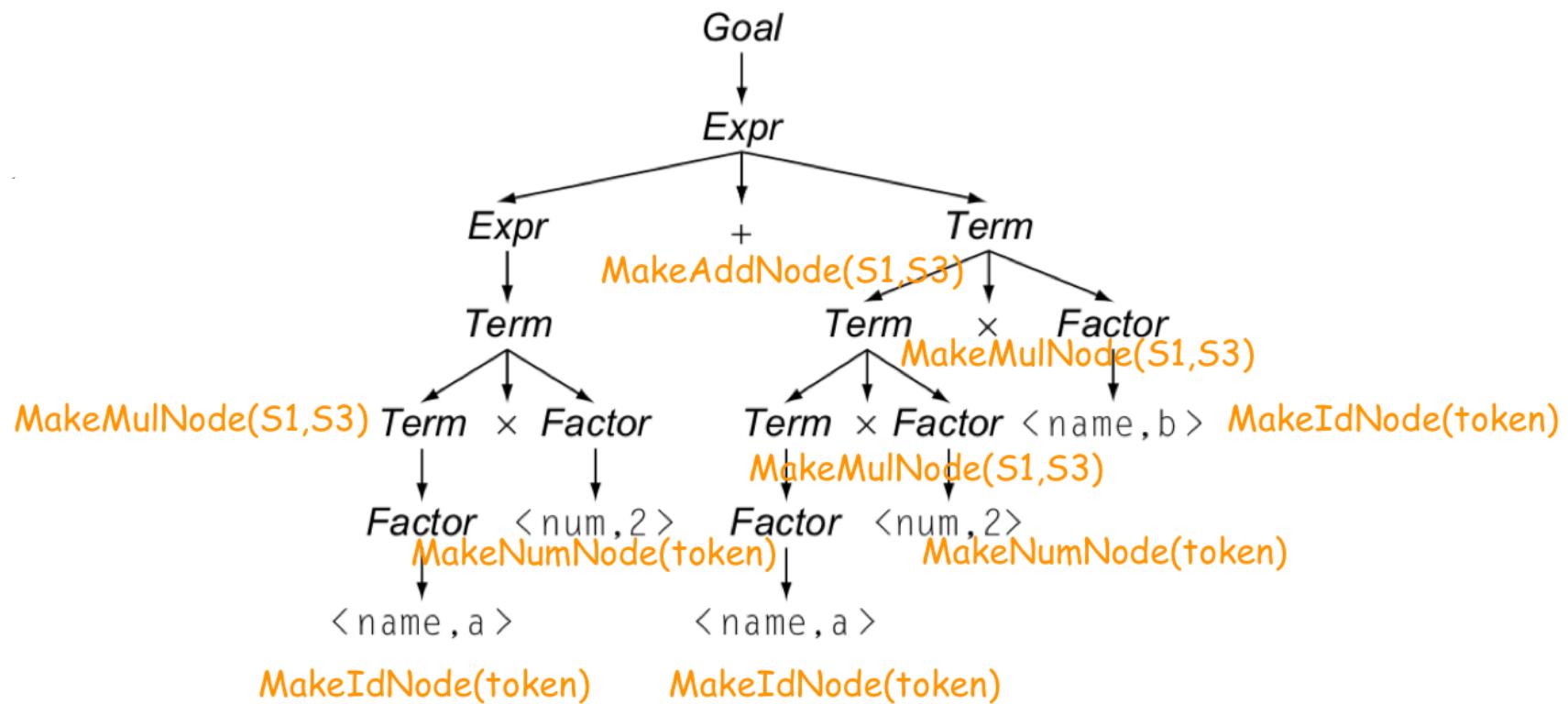
- MakeNumNode(<num,val>) → val
- MakeIdNode(<name,x>) → x

Example – Building an Abstract Syntax Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume yacc syntax

1	<i>Goal</i>	$\rightarrow Expr$	$$$ = \$1;$
2	<i>Expr</i>	$\rightarrow Expr + Term$	$$$ = \text{MakeAddNode}(\$1,\$3);$
3		$ Expr - Term$	$$$ = \text{MakeSubNode}(\$1,\$3);$
4		$ Term$	$$$ = \$1;$
5	<i>Term</i>	$\rightarrow Term^* Factor$	$$$ = \text{MakeMulNode}(\$1,\$3);$
6		$ Term / Factor$	$$$ = \text{MakeDivNode}(\$1,\$3);$
7		$ Factor$	$$$ = \$1;$
8	<i>Factor</i>	$\rightarrow (Expr)$	$$$ = \$2;$
9		$ \underline{\text{number}}$	$$$ = \text{MakeNumNode(token);}$
10		$ \underline{\text{ident}}$	$$$ = \text{MakeIdNode(token);}$

$$(a \times z) + (a \times z) \times b$$



Emitting ILOC

Assume

- NextRegister() returns a new register name
- 4 routines

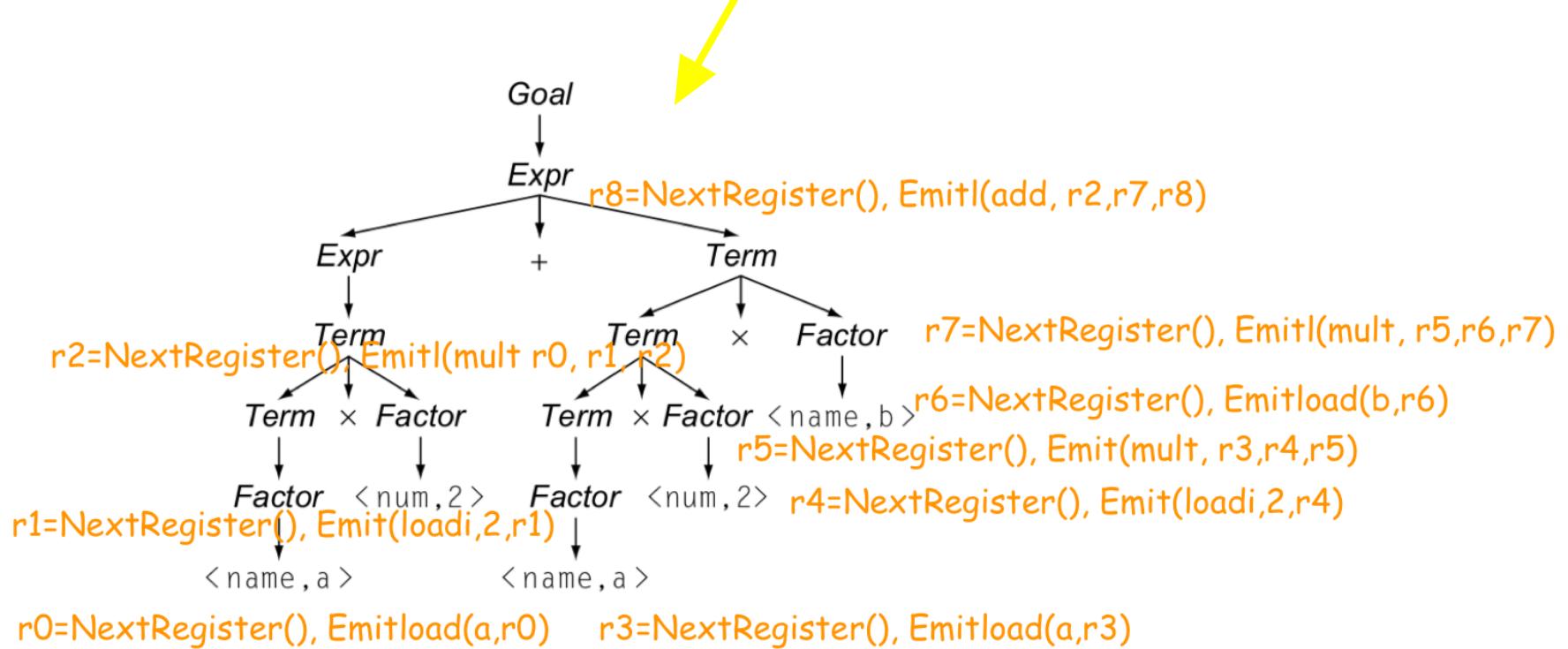
- Emit(sub, r1,r2,r3)	→	sub r1, r2, r3 (r1-r2->r3)
- Emit(mult, r1,r2,r3)	→	mult r1, r2, r3 (r1xr2->r3)
- Emit(add, r1,r2,r3)	→	add r1, r2, r3 (r1+r2->r3)
- Emit(div, r1,r2,r3)	→	div r1, r2, r3 (r1/r2->r3)
		activationrecordpointer
• EmitLoad(iden, r)	→	loadAI(rarp,@iden,r) Memory(rarp + c)->r
• Emit(loadi,n,r)	→	loadI(n,r) n->r

Example – Emitting ILOC



INTERMEDIATE LANGUAGE FOR
OPTIMIZING COMPILER (AN IR)

1	<i>Goal</i>	$\rightarrow Expr$	
2	<i>Expr</i>	$\rightarrow Expr + Term$	$$$ = \text{NextRegister}();$ $\text{Emit}(\text{add}, \$1, \$3, $$);$
3		$ Expr - Term$	$$$ = \text{NextRegister}();$ $\text{Emit}(\text{sub}, \$1, \$3, $$);$
4		$ Term$	$$$ = \$1;$
5	<i>Term</i>	$\rightarrow Term^* Factor$	$$$ = \text{NextRegister}();$ $\text{Emit}(\text{mult}, \$1, \$3, $$)$
6		$ Term / Factor$	$$$ = \text{NextRegister}();$ $\text{Emit}(\text{div}, \$1, \$3, $$);$
7		$ Factor$	$$$ = \$1;$
8	<i>Factor</i>	$\rightarrow (Expr)$	$$$ = \$2;$
9		$ \underline{\text{number}}$	$$$ = \text{NextRegister}();$ $\text{Emit}(\text{loadi}, \text{Value}(\text{lexeme}), $$);$
10		$ \underline{\text{ident}}$	$$$ = \text{NextRegister}();$ $\text{EmitLoad}(\text{ident}, $$);$



LoadAI rarp, @a, r0

LoadI 2 r1

Mult r0, r1, r2

LoadAI rarp, @a, r3

LoadI 2 r4

Mult r3, r4, r5

LoadAI rarp, @b, r6

Mult r5, r6, r7

Add r2, r7, r8

Reality

Most parsers are based on this ad-hoc style of context-sensitive analysis

Advantages

- Addresses the shortcomings of the AG paradigm
- Efficient, flexible

Disadvantages

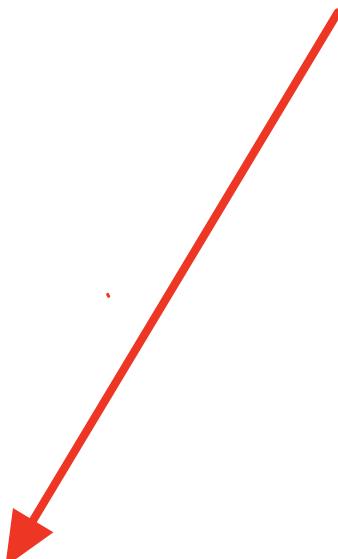
- Must write the code with little assistance
- Programmer deals directly with the details

Making Ad-hoc SDT Work

How do we fit this into an LR(1) parser?

```
stack.push(INVALID);
stack.push( $s_0$ );           // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce  $A \rightarrow \beta$ " ) then {
        stack.popnum(2*| $\beta$ |);    // pop 2*| $\beta$ | symbols
        s = stack.top();
        stack.push(A);           // push A
        stack.push(GOTO[s,A]);   // push next state
    }
    else if ( ACTION[s,token] == "shift  $s_i$ " ) then {
        stack.push(token); stack.push( $s_i$ );
        token  $\leftarrow$  scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
              & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

From previous lectures



```

stack.push(INVALID);
stack.push(NULL);
stack.push( $s_0$ );                                // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(3*|β|);      // pop 3*|β| symbols
    /* insert case statement here computing $$ */
        s = stack.top();
        stack.push(A);           // push A
        stack.push($$);          // push $$ 
        stack.push(GOTO[s,A]);   // push next state}
    else if ( ACTION[s,token] == "shift  $s_i$ " ) then {
        stack.push(token); stack.push( $s_i$ );
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
              & token == EOF )
        then break;
    else throw a syntax error;
} report success;

```

To add yacc-like actions

- Stack 3 items per symbol rather than 2 (2nd is \$\$)
- Add case statement to the reduction processing section
 - Case switches on production number
 - Each case clause holds the code snippet for that production
 - Substitute appropriate names for \$\$, \$1, \$2, ...
- Slight increase in parse time
- increase in stack space

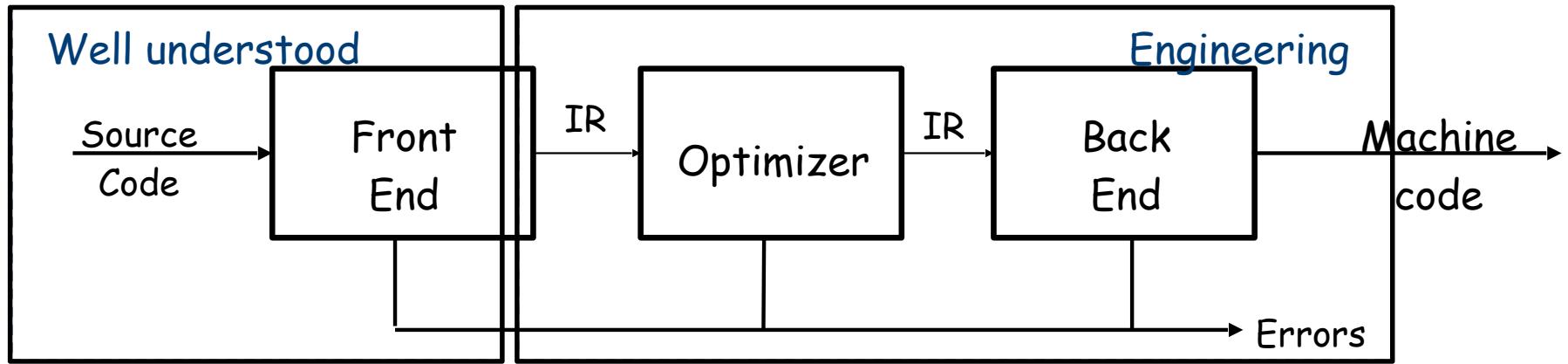
How do we fit this into an LR(1) parser?

- Need a place to store the attributes
 - Stash them in the stack, along with state and symbol
 - Push three items each time, pop $3 \times |\beta|$ symbols
- Need a naming scheme to access them
 - $\$n$ translates into stack location ($\text{top} - 3(n-1)-1$)
- Need to sequence rule applications
 - On every reduce action, perform the action rule
 - Add a giant case statement to the parser



The Procedure Abstraction

Where are we?



The latter half of a compiler contains more open problems, more challenges, and more gray areas than the front half

- This is “compilation,” as opposed to “parsing” or “translation”

Conceptual Overview

The compiler must provide, for each programming language construct, an implementation (or at least a strategy).

Those constructs fall into two major categories

- Individual statements (code shape)
- Procedures

We will look at procedures first, since they provide the surrounding context needed to implement statements

Object-oriented languages add some peculiar twists

Conceptual Overview

Procedures provide the central abstractions that make programming practical & large software systems possible

- Information hiding
- Distinct and separable name spaces
- Uniform interfaces

Hardware does little to support these abstractions

- Part of the compiler's job is to implement them
 - Compiler makes good on lies that we tell programmers
- Part of the compiler's job is to make it efficient
 - Role of code optimization

Practical Overview

The compiler must decide almost everything

- Location for each value (named and unnamed)
- Method for computing each result
 - For example, how should be translated a case statement?
- Compile-time versus runtime behaviour

```
input(x);  
if x>3  
then foo(x);  
else fee(x);
```

All of these issues come to the forefront when we consider the implementation of procedures

The Procedure Abstraction

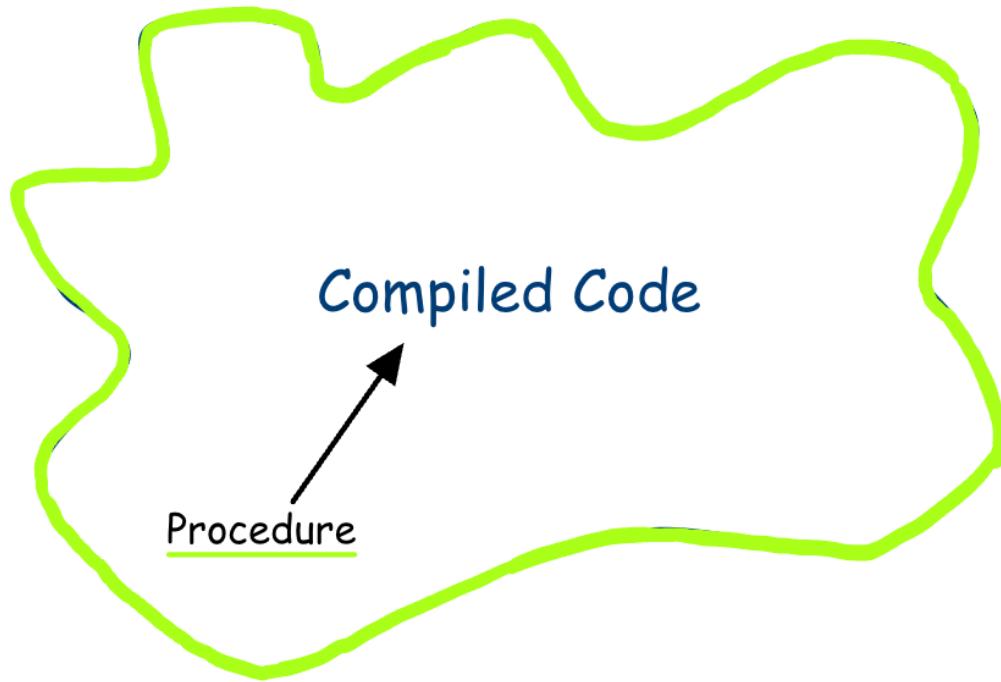
Most of the tricky issues arise in implementing "procedures"

Issues

- Compile-time versus run-time behavior
- Assign storage for everything & map names to addresses
- Generate code to address any value
 - Compiler knows where some of them are
 - Compiler cannot know where others are
- Interfaces with other programs, other languages, & the OS
- Efficiency of implementation

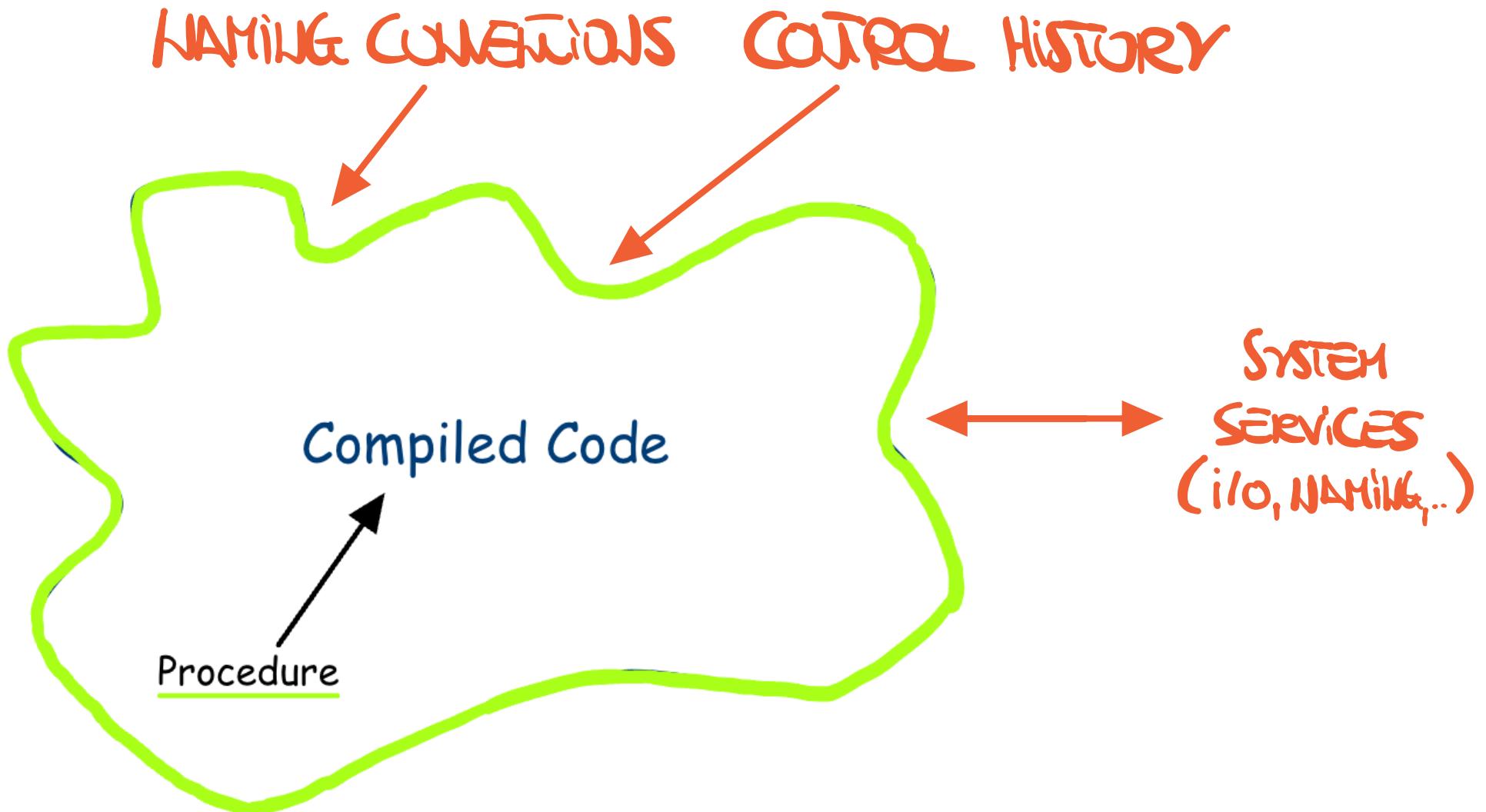
The Procedure & Its Three Abstractions

The compiler produces code for each procedure



The individual code bodies must fit together to form a working program

The Procedure & Its Three Abstractions



The Procedure & Its Three Abstractions

NAMING CONVENTIONS

Each procedure inherits a set of names

- ⇒ Variables, values, procedures, objects, locations, ...
- ⇒ Clean slate for new names, "scoping" can hide other names

"Naming" includes
the ability to find
and access objects in
memory

CONTROL HISTORY

Each procedure inherits a control history

- ⇒ Chain of calls that led to its invocation.
- ⇒ Mechanism to return control to caller

SYSTEM SERVICES

Each procedure has access to external interfaces

- ⇒ Access by name, with parameters
- ⇒ Protection for both sides of the interface

The Procedure: Three Abstractions

- Control Abstraction
 - Well defined entries & exits
 - Mechanism to return control to caller
 - Some notion of parameterization (formal and actual parameters)
- Clean Name Space
 - Clean slate for writing locally visible names
 - Local names may obscure identical, non-local names
 - Local names cannot be seen outside
- External Interface
 - Access is by procedure name & parameters
 - Clear protection for both caller & callee
 - Invoked procedure can ignore calling context

Procedures permit a critical separation of concerns

The Procedure

Procedures allow us to use separate compilation

- Separate compilation allows us to build non-trivial programs
- Keeps compile times reasonable
- Lets multiple programmers collaborate
- Requires independent procedures

Without separate compilation, we would not build large systems

The procedure linkage convention (agreement that defines the actions to take to call a procedure)

- Ensures that each procedure inherits a valid run-time environment and that the callers environment is restored on return
 - The compiler must generate code to ensure this happens according to conventions established by the system

The Procedure

(More Abstract View)

A procedure is an abstract structure constructed via software

Underlying hardware directly supports little of the abstraction—it understands bits, bytes, integers, reals, & addresses, but not:

- Entries and exits
- Interfaces
- Call and return mechanisms
 - Typical machine supports the transfer of control (call and return) but not the rest of the calling sequence (e.g., preserving context)
- Name space
- Nested scopes

All these are established by carefully-crafted mechanisms provided by compiler, run-time system, linker, loader, and OS;

Run Time versus Compile Time

These concepts are often confusing

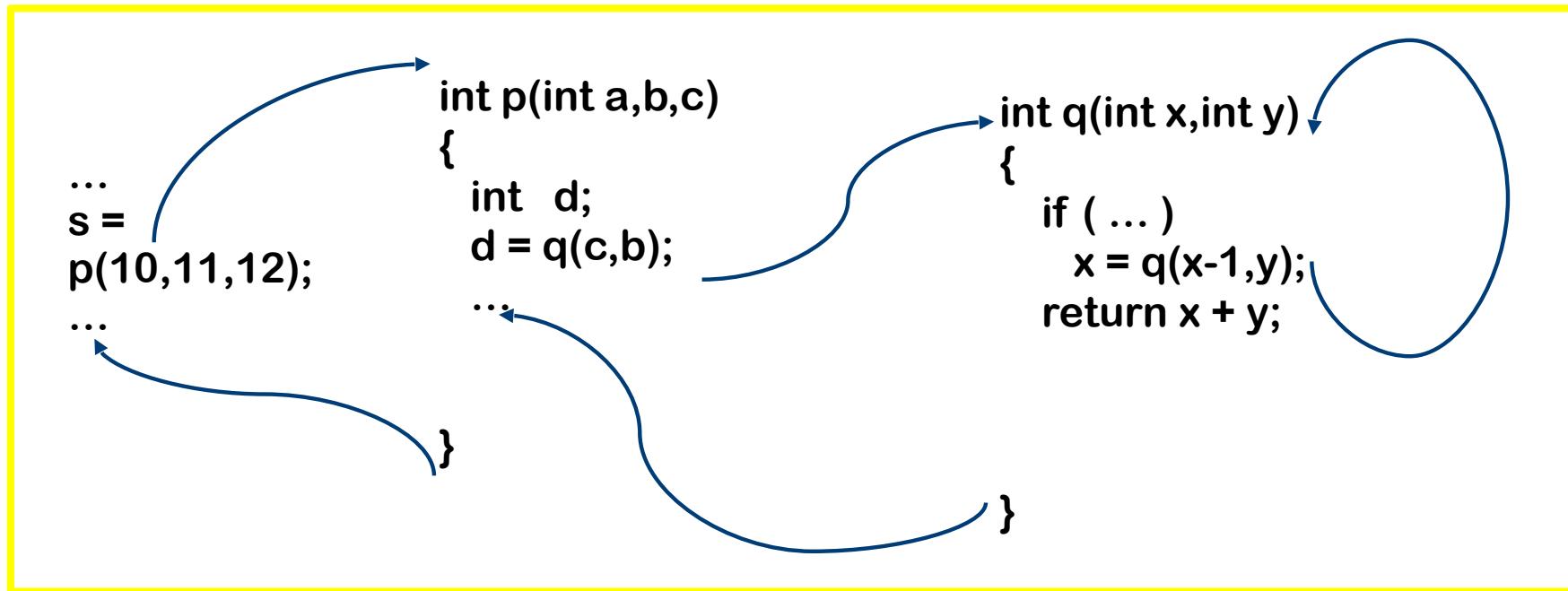
- Linkages (and code for procedure body) execute at run time
- Code for the linkage is emitted at compile time
- The linkage is designed long before either of these

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

Invoked at a call site, with some set of actual parameters

- Control returns to call site, immediately after invocation



- Most languages allow recursion

The Procedure as a Control Abstraction



Implementing procedures with this behavior

- Requires code to **save** and **restore** a "return address"
- Must map **actual parameters** to **formal parameters** ($c \rightarrow x, b \rightarrow y$)
- Must create storage for **local variables**
- p needs space for d
- where does this space go in recursive invocations?

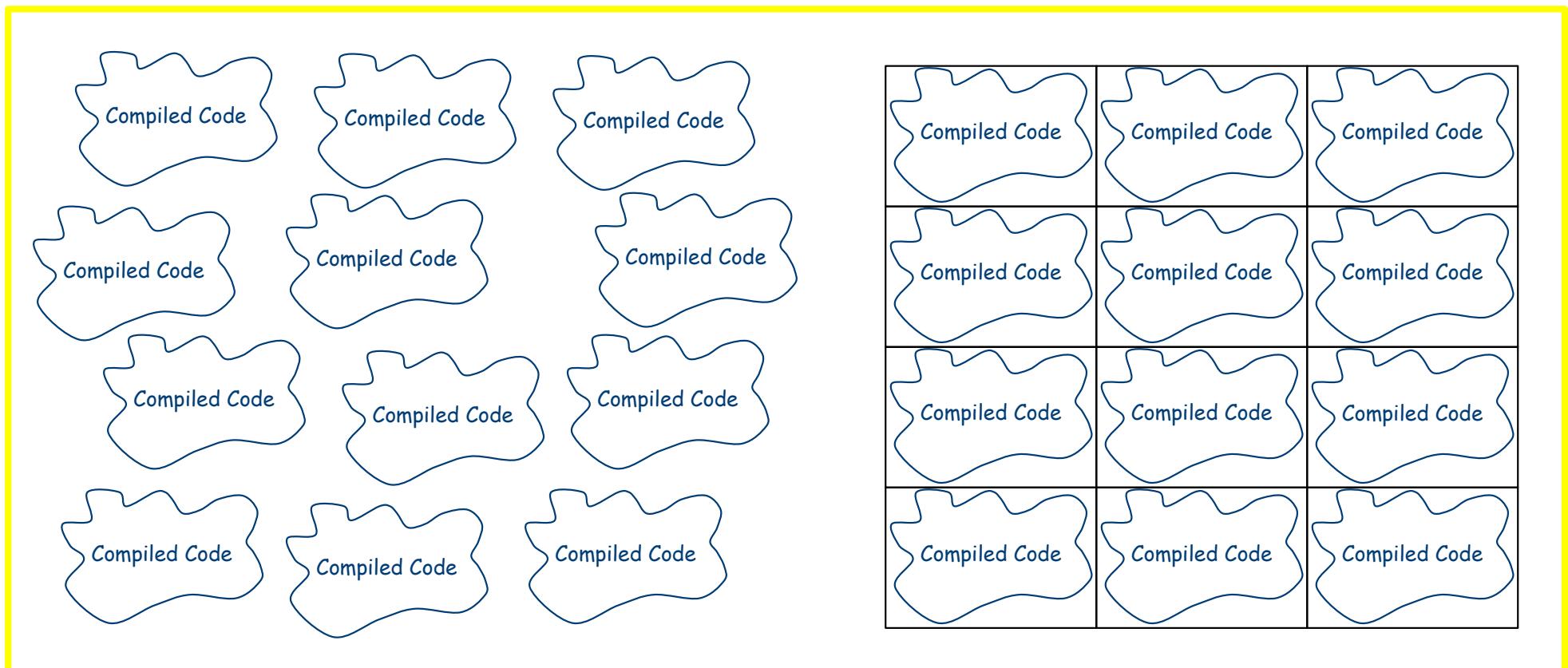
Compiler emits code that causes all this to happen at run time

The Procedure as a Control Abstraction

- Must preserve p's **state** while q executes
 - recursion causes the real problem here
- Strategy: Create unique location for each procedure **activation**
 - In simple situations, can use a "stack" of memory blocks to hold local storage and return addresses **closures (procedure+runtime context) ⇒ heap allocate**

The Procedure as a Control Abstraction

In essence, the procedure linkage wraps around the unique code of each procedure to give it a uniform interface



Similar to building a brick wall rather than a rock wall

The Procedure as a Name Space

Each procedure creates its own name space

- Any name can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
 - Nested procedures are "inside" by definition
- We call this set of rules & conventions "lexical scoping"

Examples

- C has global, static, local, and block scopes (Fortran-like)
 - Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes (let)

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding "free" variables
- Simplifies rules for naming & resolves conflicts
- Lets the programmer introduce "local" names with impunity

How can the compiler keep track of all those names?

The Problem

- At point p, which declaration of x is current?
- At run-time, where is the value of x that can be used?
- As parser goes in & out of scopes, how does it delete x?

The Answer

- The compiler must model the name space
- Lexically scoped symbol tables

Lexical vs Dynamical scoping

- **Lexical scoping:** each free variable is bound to the declaration for its name that is lexically closest to the use
 - The declaration always come from a scope that encloses the reference.
- **Dynamic scoping:** a free variable is bound to the variable by that name that was most recently created at runtime. Example LISP or as possibility Common LISP

Where Do All These Variables Go?

Automatic & Local

- Keep them in the procedure activation record or in a register
- Automatic \Rightarrow lifetime matches procedure's lifetime

Static

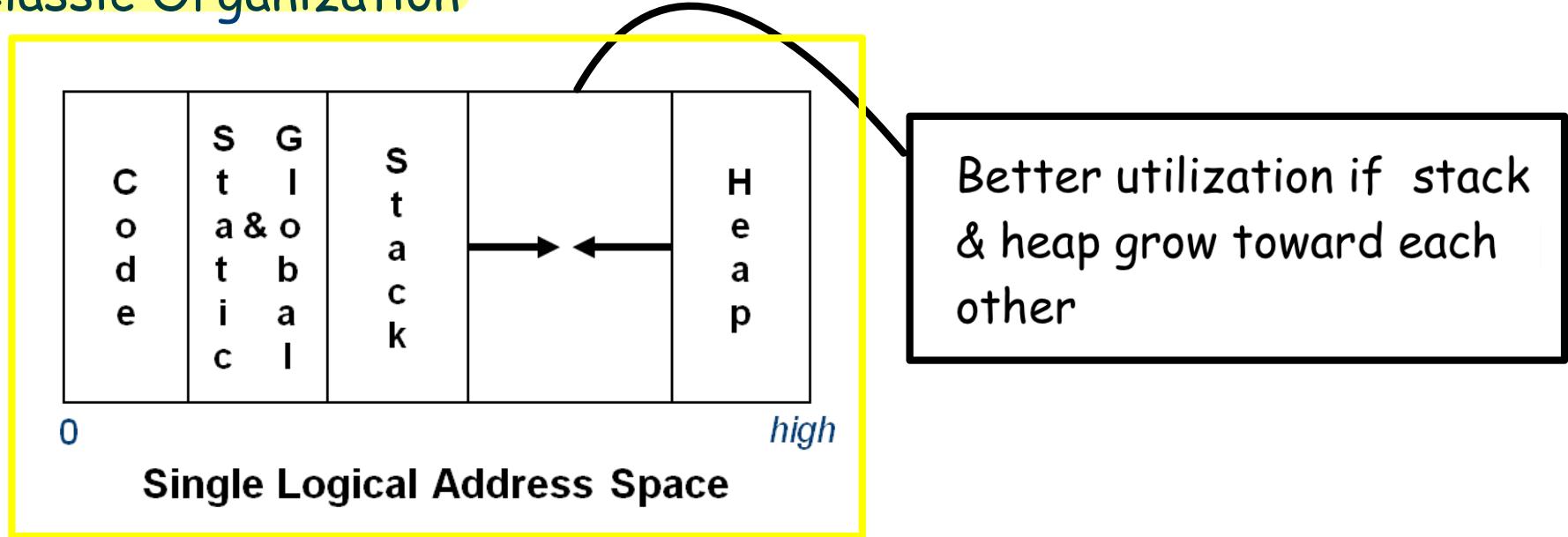
- Procedure scope \Rightarrow storage area affixed with procedure name
- File scope \Rightarrow storage area affixed with file name
- Lifetime is entire execution

Global

- One or more named global data areas
- Lifetime is entire execution

Placing Run-time Data Structures

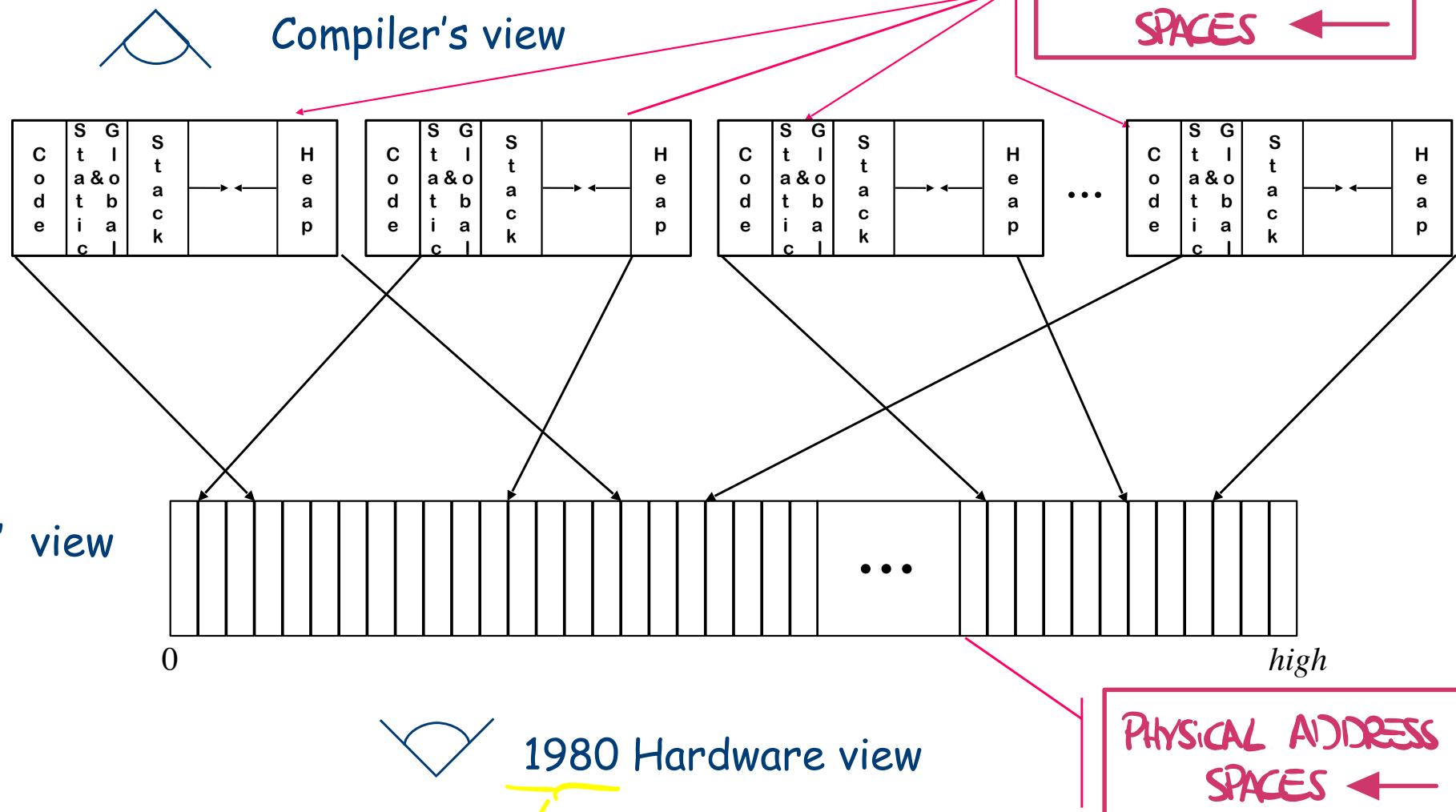
Classic Organization



- Code, static, & global data have known size
 - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a virtual address space

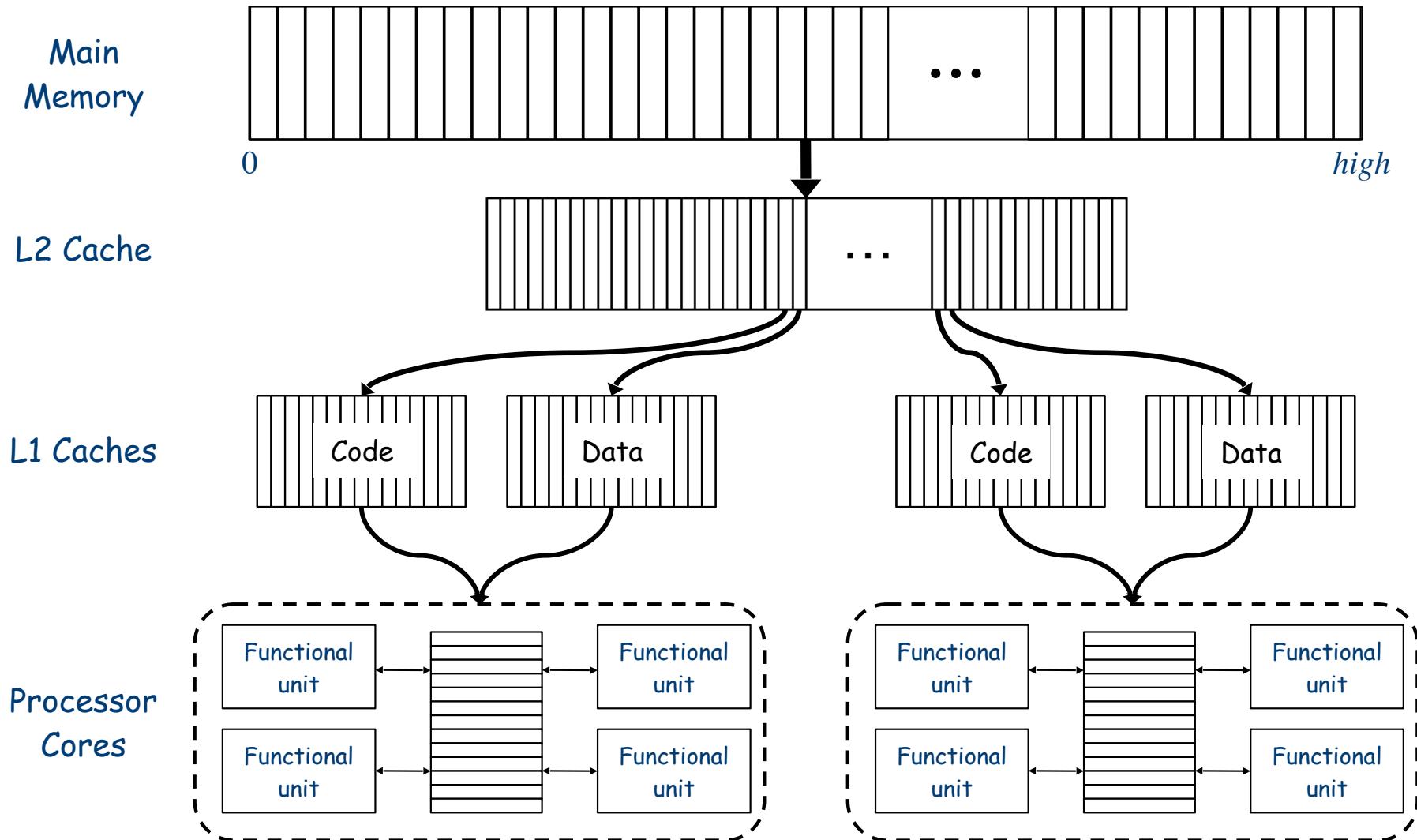
How Does This Really Work?

The Big Picture



Cache structure matters for performance, not correctness

How Does This Really Work?



Where Do Local Variables Live?

A Simplistic model

- Allocate a data area for each distinct scope

What about recursion?

- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there!

More complex scheme

- One activation record (AR) per procedure instance
- All the procedure's scopes share a single AR (may share space)
- Static relationship between scopes in single procedure

USED THIS WAY, "STATIC" MEANS KNOWNABLE AT COMPILE TIME, AKA FIXED

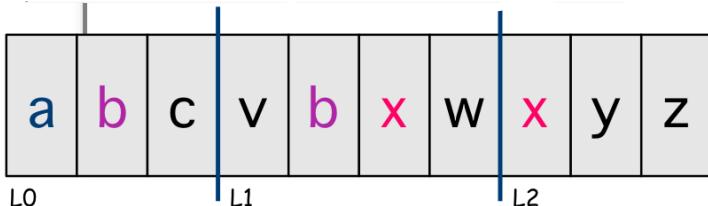
Translating Local Names

How does the compiler represent a specific instance of x ?

- Name is translated into a static coordinate
 - $\langle \text{level}, \text{offset} \rangle$ pair
 - "level" is lexical nesting level of the procedure
 - "offset" is unique within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "level" is a function of the table in which x is found
 - Stored in the entry for each x
- "offset" must be assigned and stored in the symbol table
 - Assigned at compile time
 - Known at compile time
 - Used to generate code that executes at run-time

Storage for Blocks within a Single Procedure

```
B0: {  
    int a, b, c  
B1: {  
    int v, b, x, w  
B2: {  
    int x, y, z  
    ...  
}  
B3: {  
    int x, a, v  
    ...  
}  
    ...  
}
```

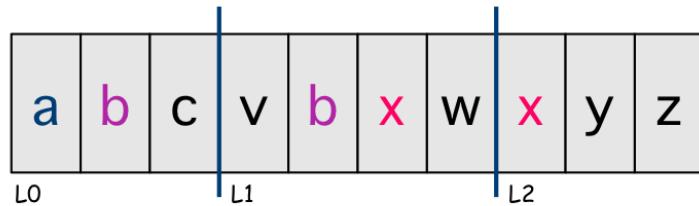


Storage in block B2

Fixed length data can always be at a constant offset from the beginning of a procedure

- In our example, the **a** declared at **level 0** will always be the first data element, stored at **byte 0** in the **fixed-length data area**
- The **x** declared at **level 1** will always be the **sixth data item**, stored at **byte 20** in the **fixed data area**
- The **x** declared at **level 2** will always be the **eighth data item**, stored at **byte 28** in the **fixed data area**
- But what about the **a** declared in the second block at **level 2**?

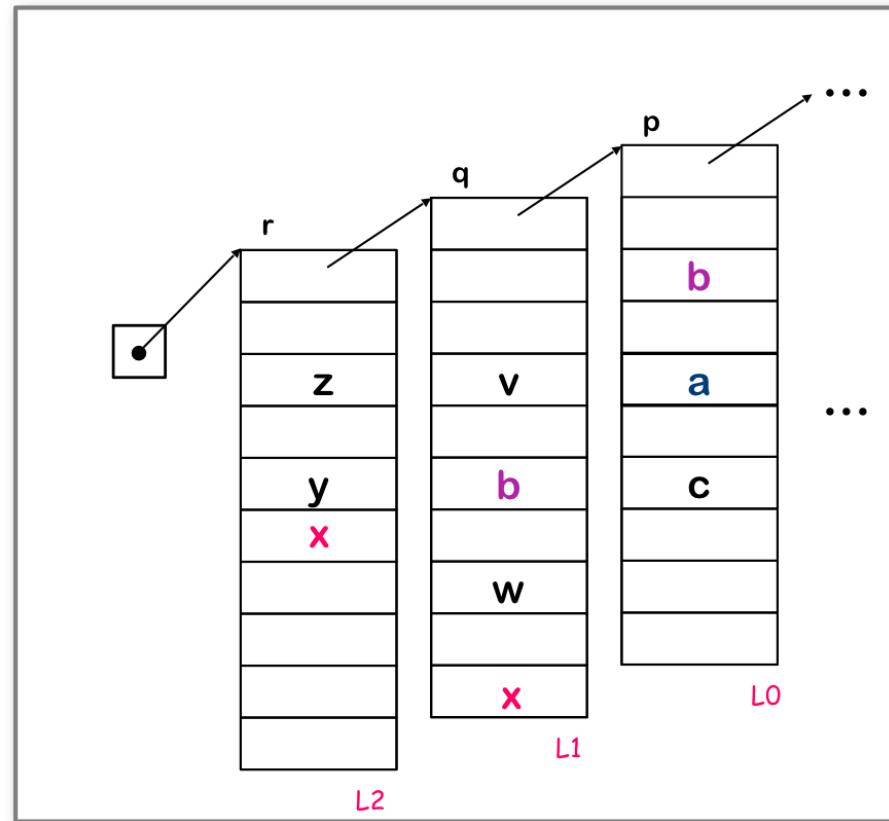
Lexically-scoped Symbol Tables



Storage in block B2

High-level idea

- Create a new table for each scope
- Chain them together for lookup

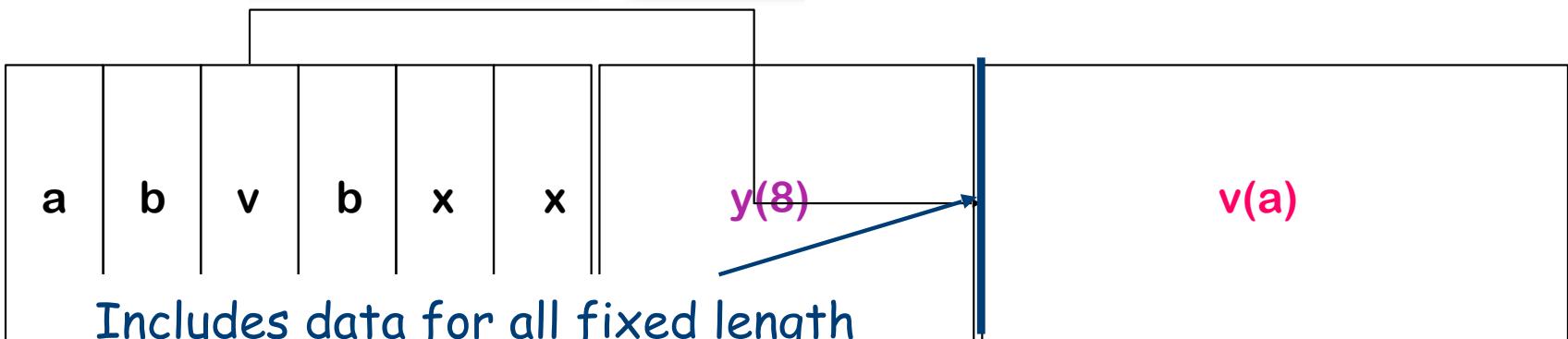


Variable-length Data

```
B0: { int a, b  
      ...  
      assign value to a  
      ...  
B1: { int v(a), b, x  
      ...  
B2: { int x, y(8)  
      ...  
    }  
}  
}
```

Arrays

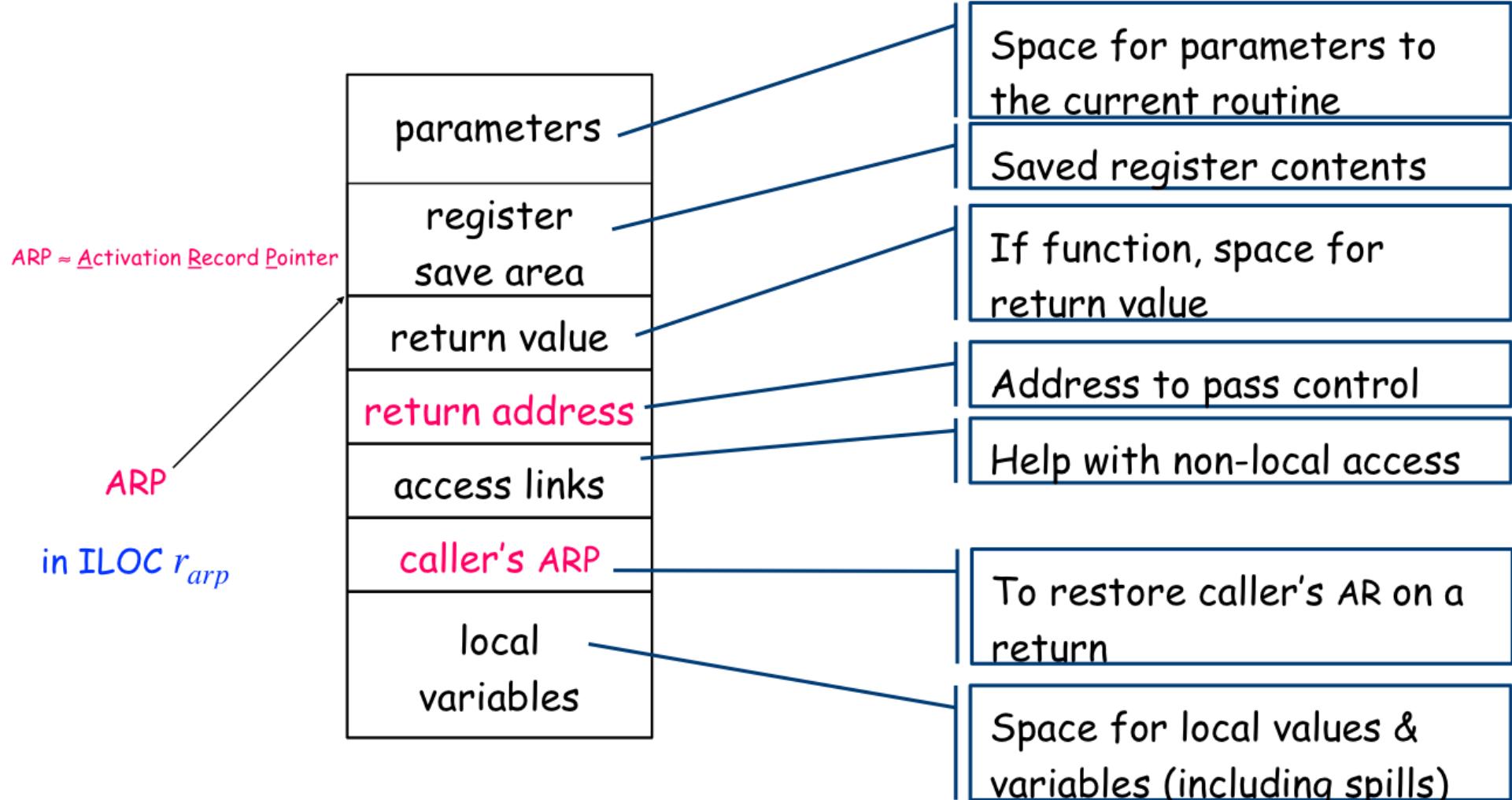
- If size is fixed at compile time, store in fixed-length data area
- If size is variable, store descriptor in fixed length area, with pointer to variable length area
- Variable-length data area is assigned at the end of the fixed length area for the block in which it is allocated (including all contained blocks)



Includes data for all fixed length objects in all blocks

Variable-length data

Activation Record Basics



One AR for each invocation of a procedure

Activation Record Details

How does the compiler find the variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a "loadAI" operation
 - Level specifies an ARP, offset is the constant

Variable-length data

- If AR can be extended, put it above local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

- Must generate explicit code to store the values
- Among the procedure's first actions

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, AND
- If code normally executes a "return"

⇒ Keep ARs on a stack

C,Pascal

- If a procedure can outlive its caller, OR
- If it can return an object that can reference its execution state

ML

⇒ ARs must be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Automatic	\Rightarrow	Lifetime matches procedure activation
Static	\Rightarrow	Lifetime may be as long as entire execution
Dynamic	\Rightarrow	Lifetime is under program control & not known at compile time

Recap

Where do variables live?

- Local & automatic \Rightarrow in procedure's activation record (AR)
- Static (@ any scope) \Rightarrow in a named static data area
- Dynamic (@ any scope) \Rightarrow on the heap

Variable length items?

- Put a descriptor in the "natural" location
- Allocate item at end of AR or in the heap

Represent variables by their static coordinates, <level,offset>

- Must map, at runtime, level into a data-area base address
- Must emit, at compile time, code to perform that mapping

Establishing Addressability

Must compute base addresses for each kind of data area

- Local variables
 - Convert to static data coordinate and use ARP + offset
- Local variables of other procedures
 - Convert to static coordinates
 - Find appropriate ARP
 - Use that ARP + offset
- Global & static variables

Use static
coordinates

<1.0>

Must find the right AR
Need links to nameable ARs

Establishing Addressability

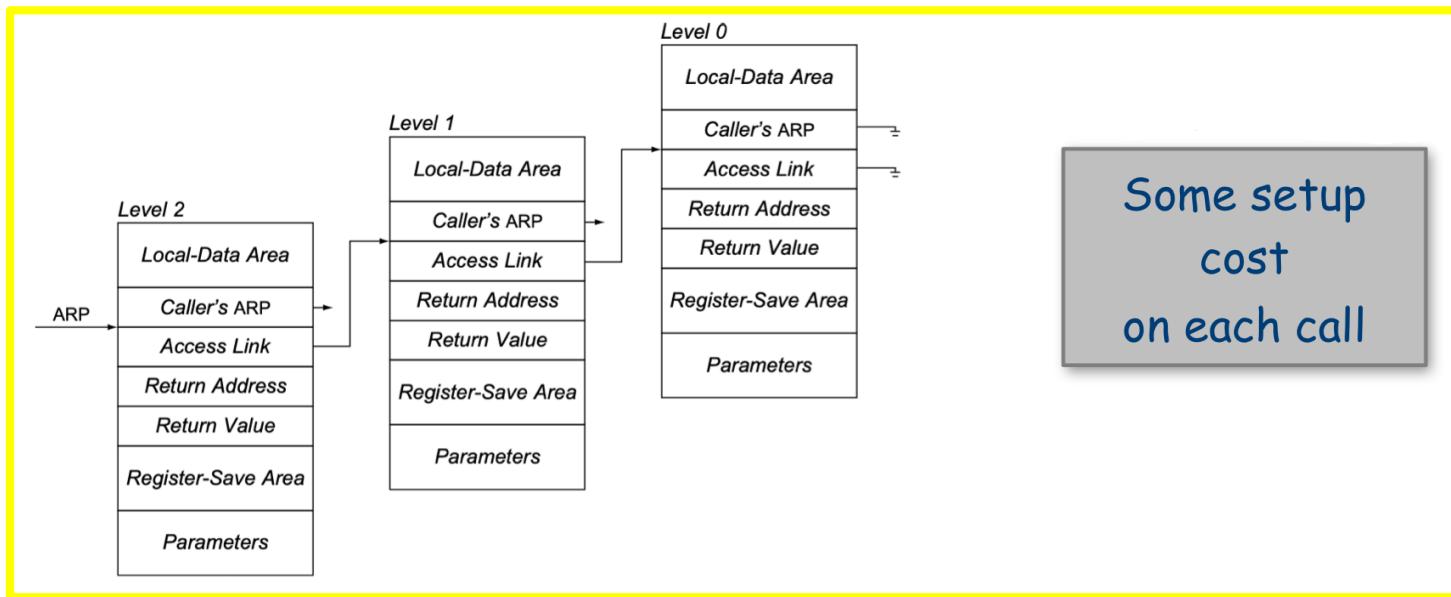
Two different ways:

- Using a Address link technique
- Using a Display technique

Establishing Addressability

Using Access Links to Find an ARP for a Non-Local Variable

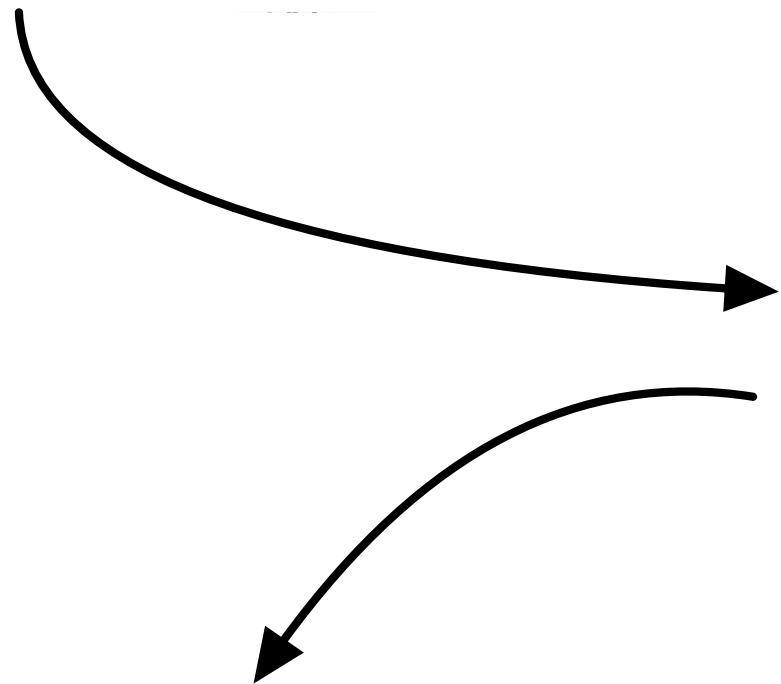
- Each AR has a pointer to AR of **lexical** ancestor
- Lexical ancestor need not be the caller



- Reference to `<p,16>` runs up access link chain to `p`
- Cost of access is proportional to lexical distance

Assume

- Current lexical level is 2
- Access link is at ARP - 4
- ARP is in r_0



Using Access Links

SC	Generated Code
<2,8>	loadAI $r_0,8 \Rightarrow r_{10}$
<1,12>	loadAI $r_0,-4 \Rightarrow r_1$
	loadAI $r_1,12 \Rightarrow r_{10}$
<0,16>	loadAI $r_0,-4 \Rightarrow r_1$
	loadAI $r_1,-4 \Rightarrow r_1$
	loadAI $r_1,16 \Rightarrow r_{10}$

Access cost varies with level

All accesses are relative to ARP (r_0))

Maintain access links

The compiler must add code to each procedure call that finds the appropriate ARP and stores in the activation record of the callee (at the position reserved for the access link)

Maintaining access link

For a caller at level p and a callee is defined at level q

- $q=p+1$ the callee is nested inside the caller

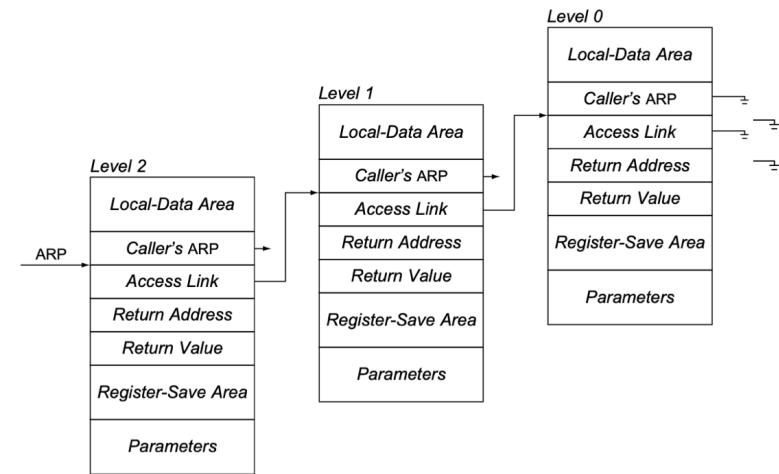
- $q=p$

→ Callee copy the access link of the caller

- $q < p$

→ Find ARP for level $q - 1$

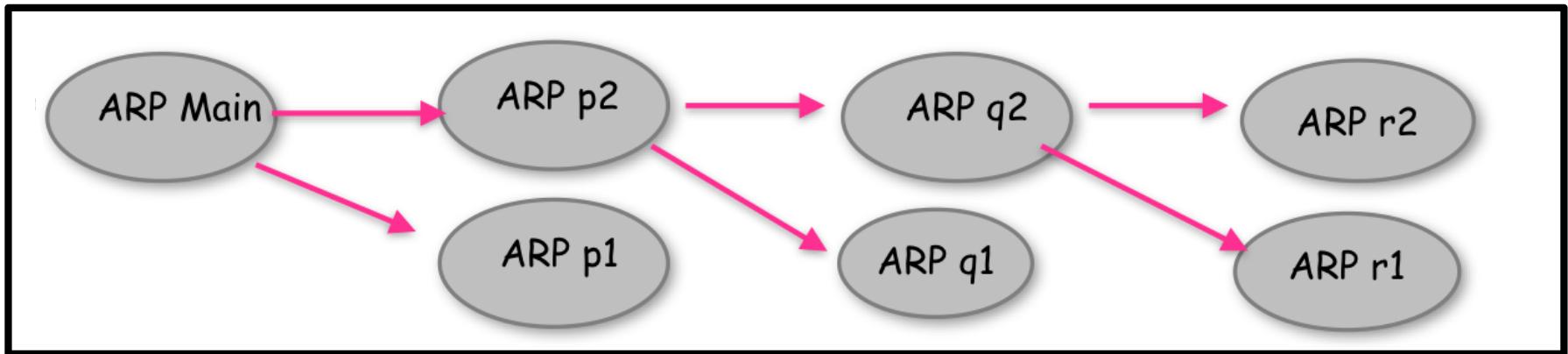
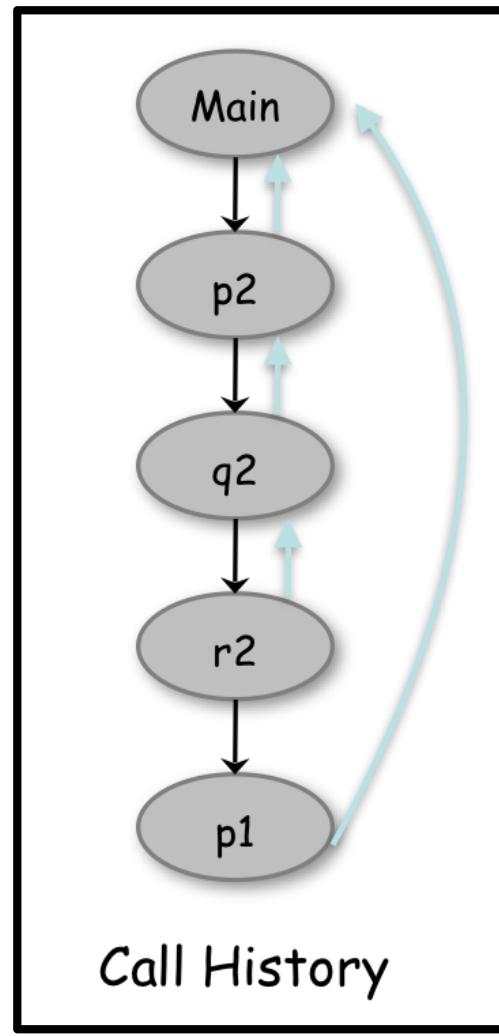
→ Use that ARP as link



Cost of maintenance is proportional to lexical distance

THE STATIC AND THE CALL CHAIN DO NOT COINCIDE

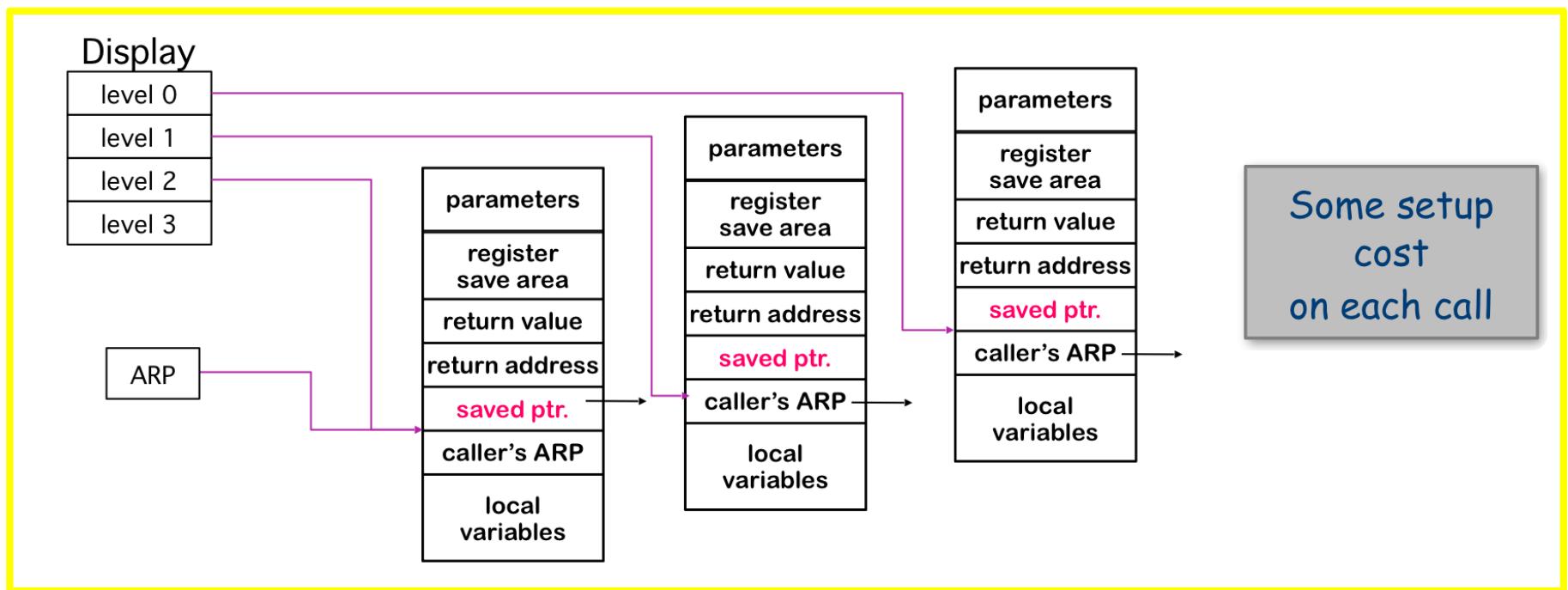
```
procedure main {  
    procedure p1 { ... }  
    procedure p2 {  
        procedure q1 { ... }  
        procedure q2 {  
            procedure r1 { ... }  
            procedure r2 {  
                call p1; ... // call up from level 3 to level 1  
            } // end of r2  
                call r2;    // call down from level 2 to level 3  
            } //end of q2  
                call q2;      // call down from level 1 to level 2  
        } //end of p2  
        call p2;          // call down from level 0 to level 1  
    } // end of main
```



Establishing Addressability

Using a Display to Find an ARP for a Non-Local Variable

- Global array of pointer to nameable ARPs
- Needed ARP is an array access away



- Reference to `<p,16>` looks up p's ARP in display & adds 16
- Cost of access is constant $(\text{ARP} + \text{offset})$

Establishing Addressability

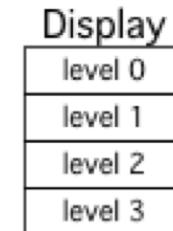
Using a Display

SC	Generated Code	
<2,8>	loadAI r ₀ ,8	$\Rightarrow r_{10}$
<1,12>	loadl _disp	$\Rightarrow r_1$
	loadAI r ₁ ,4	$\Rightarrow r_1$
	loadAI r ₁ ,12	$\Rightarrow r_{10}$
<0,16>	loadl _disp	$\Rightarrow r_1$
	loadAI r ₁ ,0	$\Rightarrow r_1$
	loadAI r ₁ ,16	$\Rightarrow r_{10}$

Assume

- Current lexical level is 2
- Display is at label _disp

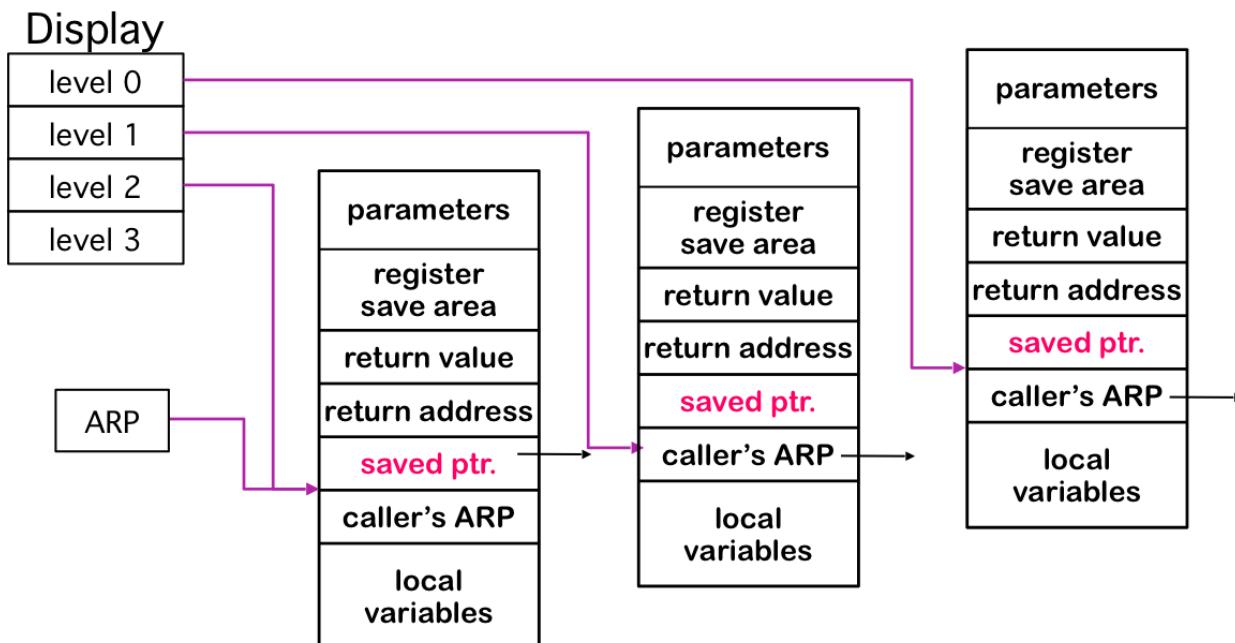
Desired AR is at _disp + 4 × level



Access costs are fixed

Address of display may consume a register

Maintaining Display



Maintaining access links

- On entry to level j
 - Save level j entry into AR (**saved ptr field**)
 - Store ARP in level j slot
- On exit from level j
 - Restore old level j entry

Establishing Addressability

Access Links Versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
 - Overhead only incurred on references & calls
 - If ARs outlive the procedure, access links still work
- Display costs are fixed for all references
 - References & calls must load display address
 - Typically, this requires a register
 - Depends on ratio of non-local accesses to calls

For either scheme to work, the compiler must insert code into each procedure call & return

Creating and Destroying Activation Records

All three parts of the procedure abstraction leave state in the activation record

- How are ARs created and destroyed
 - Procedure call must allocate & initialize (preserve caller's world)
 - Return must dismantle environment (and restore caller's world)
- Caller & callee must collaborate on the problem
 - Caller alone knows some of the necessary state
 - Return address, parameter values, access to other scopes
 - Callee alone knows the rest
 - Size of local data area, registers it will use

Their collaboration takes the form of a linkage convention

Procedure Linkages

How do procedure calls actually work?

At compile time, callee may not be available for inspection

- Different calls may be in different compilation units
- All calls must use the same protocol

Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement, to allow interoperability

Saving Registers

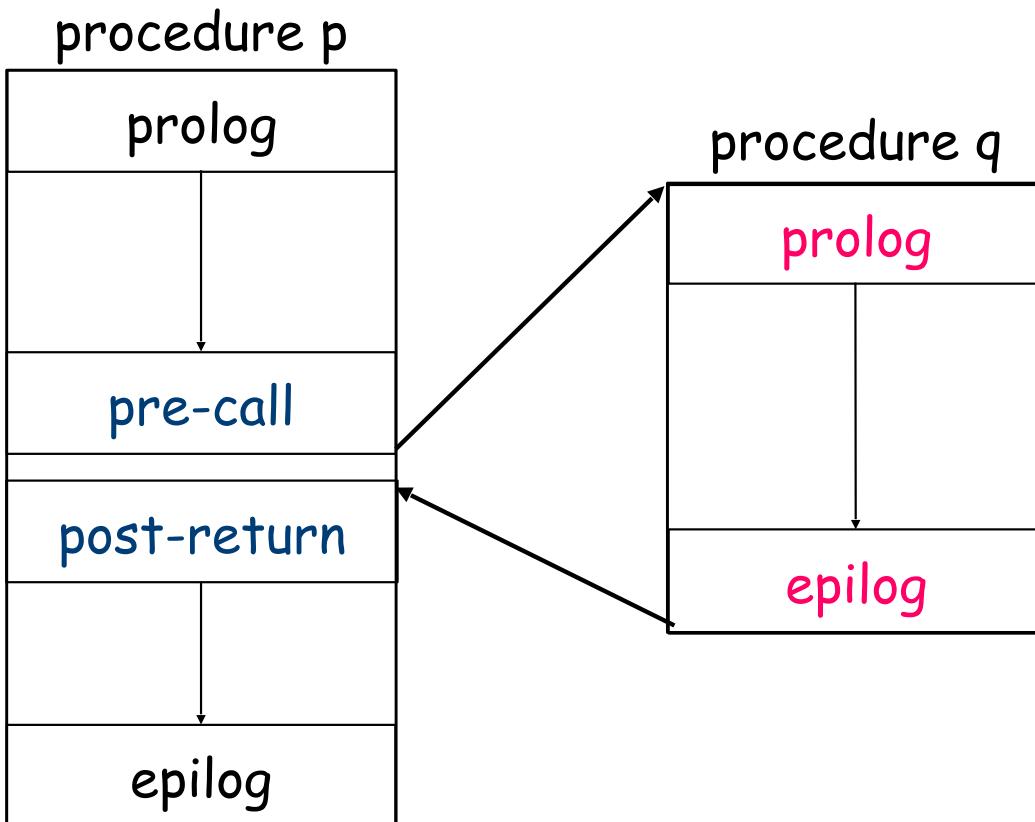
Who saves the registers? Caller or callee?

- Arguments for saving on each side of the call
 - Caller knows which values are LIVE across the call
 - Callee knows which registers it will use
- Conventional wisdom: divide registers into three sets
 - Caller saves registers
 - Caller targets values that short-LIVED value across the call
 - Callee saves registers
 - Callee only uses these AFTER filling caller saved registers
 - Registers reserved for the linkage convention
 - ARP, return address (if in a register), ...

Where are they stored? In one of the ARs ...

Procedure Linkages

Standard Procedure Linkage



Procedure has

- standard **prolog**
- standard **epilog**

Each call involves a

- pre-call sequence
- post-return sequence

These are completely predictable from the call site \Rightarrow depend on the number & type of the actual parameters

Procedure Linkages

Pre-call Sequence

- Starts setting up callee's basic environment
- Evaluates formal parameters

The Details

- Allocate space for the callee's AR
- Evaluates each parameter & stores value or address
- Saves return address: caller's ARP into callee's AR
- If access links are used
 - Find appropriate lexical ancestor & copy into callee's AR
- Save any caller-save registers
 - Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Undo the actions of the precall sequence
- Place any value back where it belongs

The Details

- Free the callee's AR
- Restore any caller-saved registers
- Restore any call-by-reference parameters to registers, if needed
 - Also copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code

- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

The Details

- Preserve any callee-saved registers
- If display is being used
 - Save display entry for current lexical level
 - Store current ARP into display for current lexical level
- Allocate space for local data
- Handle any local variable initializations

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value
- Restore callee-saved registers
- Free space for local data, if necessary
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

How is it realised? It depends on where the AR are...

If activation records are stored on the stack

Algol-60 rules

- Easy to extend – simply bump top of stack pointer
- Caller & callee share responsibility
 - Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
 - Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

ML rules

- Hard to extend
- Several options
 - Caller passes everything in registers; callee allocates & fills AR
 - Store parameters, return address, etc., in caller's AR !
 - Store callee's AR size in a defined static constant

Without recursion, activation records can be static

Fortran 66 & 77

Communicating Between Procedures

Most languages provide a parameter passing mechanism

⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
 - Requires slot in the AR (for address of parameter)
 - Multiple names with the same address
- Call-by-value passes a copy of its value at time of call
 - Requires slot in the AR
 - Each name gets a unique location (may have same value)
 - Arrays are mostly passed by reference, not value

call
fee(x,x,x);

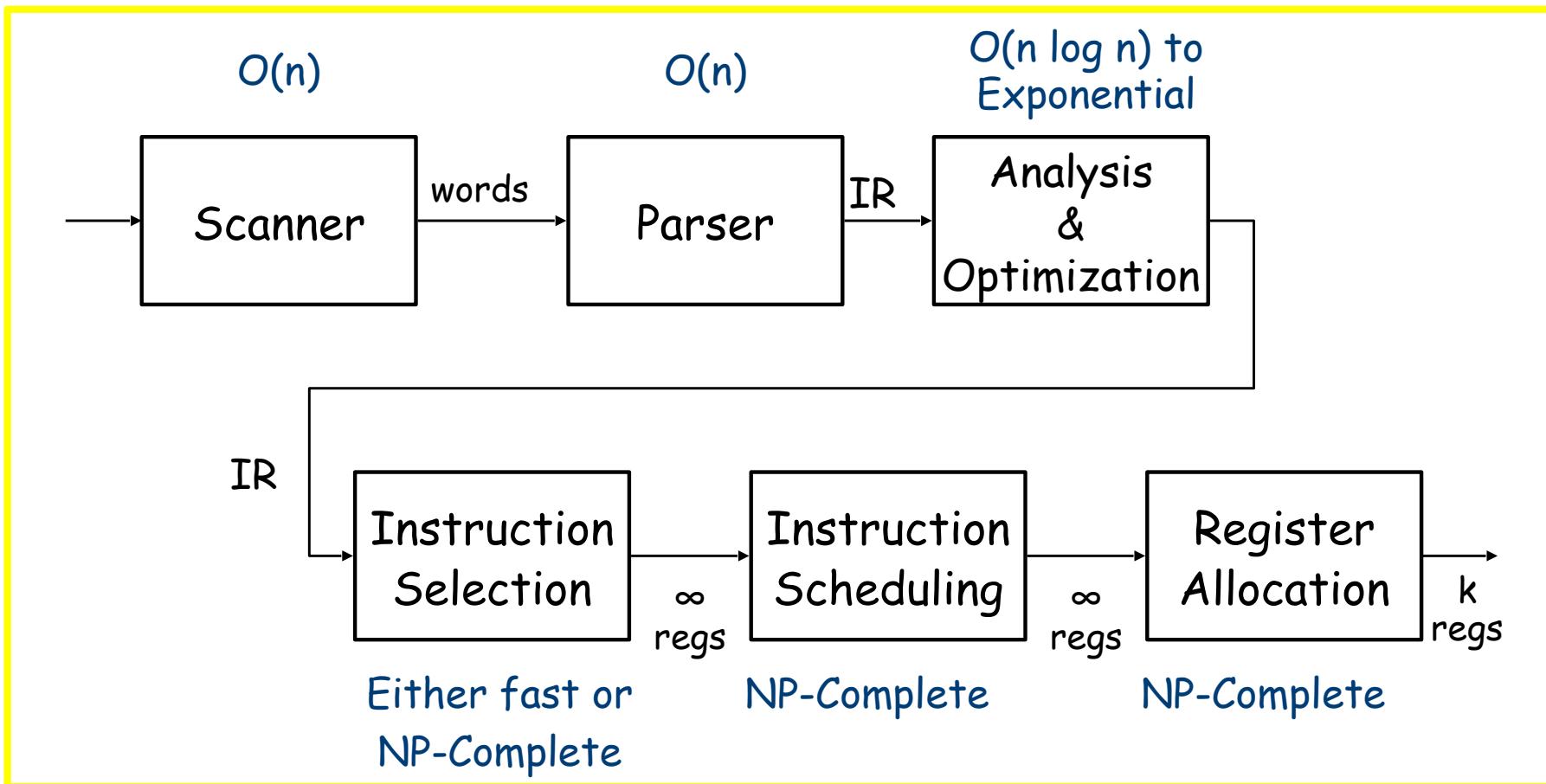


Introduction to Code Generation

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Structure of a Compiler

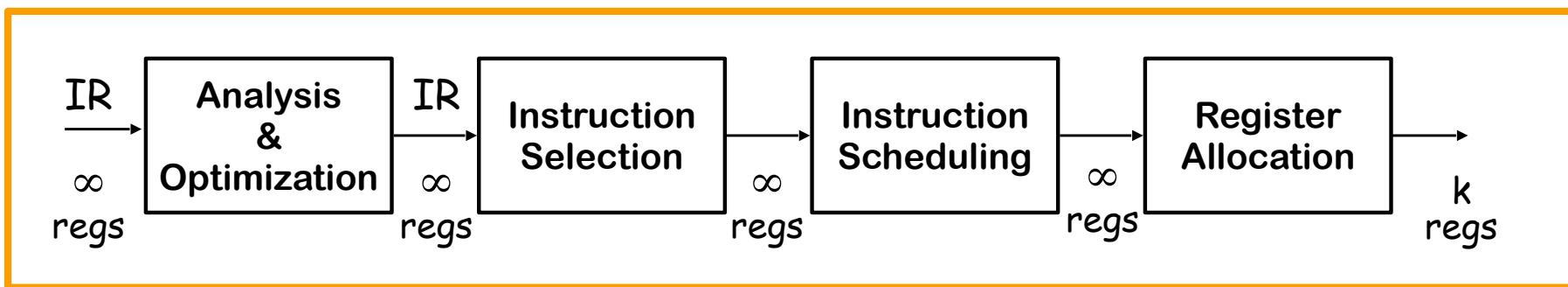


A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in **code generation** and **optimization**
- For multicores, we need to manage parallelism & sharing
- For unicore performance, allocation & scheduling are critical

Structure of a Compiler

We assume the following model



- Selection can be fairly simple (problem of the 1980s)
- Allocation & scheduling are complex
- Operation placement is not yet critical
we assumed a unified register set

What about the IR ?

- Low-level, RISC-like IR such as ILOC
- Has "enough" registers
- ILOC was designed for this stuff with:
 - Branches, compares, & labels
 - Memory tags
 - Hierarchy of loads & stores
 - Provision for multiple ops/cycle

Analysis & Optimization

- The translation of the front end was obtained by considering the statements one at a time as they were encountered
- This initial IR contains general implementation strategies that will work in any surrounding context
- At run time the code will be executed in a more constrained and predictable context
- The optimizer analyses the IR form of the code to discover facts about the context and use them to rewrite (transform) the code so that it will compute the same answer in a more efficient way

The Back End

The compiler back end traverses the IR form and emits the code for the target machine

- It selects target-machine operations to implement each IR operation (**Instruction selection**)
- It chooses an order in which the operations will execute efficiently (**Instruction scheduling**)
- It will decide which values will reside in registers and which in memory (**Register allocation**)

Memory Models

Two major models

- Register-to-register model
 - Keep all values that can legally be stored in a register in registers
 - Ignore machine limitations on number of registers
 - Compiler back-end must insert loads and stores
- Memory-to-memory model
 - Keep all values in memory
 - Only promote values to registers directly before they are used
 - Compiler back-end can remove loads and stores
- Compilers for RISC machines usually use register-to-register
 - Easier to determine when registers are used

use virtual
registers!

Definitions

Instruction selection

- Mapping IR into assembly code
- Assumes a fixed memory model & code shape
- Combining operations, using address modes (instr. reg+offset or reg to reg mode)

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program (set of operations)
- Changes demand for registers

These 3 problems
are tightly coupled
and need static
analysis

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

Definition

- The compiler must choose among many alternative ways to implement each construct on a given processor
- Those choices have a strong and direct impact on the quality of the final produced code
- Code shape is the end product of many decisions (big & small)

Impact

- Code shape has a strong impact on the behaviour of the compiled code and on the ability of the optimizer and back end to improve it
- Code shape can encode important facts, or hide them

Code Shape

Example -- the case statement on a character value

- Implement it as cascaded if-then-else statements
 - Cost depends on where your case actually occurs
 - $O(256)$
- Implement it as a binary search
 - Need a dense set of conditions to search
 - Uniform ($\log 256$) cost
- Implement it as a jump table
 - Lookup address in a table & jump to it
 - We trade data space for speed
 - Uniform (constant) cost

All these are legal (and reasonable) implementations of the switch statement

Which implementation for switch?

The one that is the best for a particular switch statement depends on many factors such as:

- The number of cases and their relative executions frequencies
- The knowledge of the cost structure for branching on the processor

Even when the compiler does not have enough information to choose it must choose an implementation strategy

No amount of massaging or transforming will convert one into another

Code Shape: the ternary operation $x+y+z$

Several ways to implement $x+y+z$

$$x + y + z$$

$$x + y \rightarrow t1$$

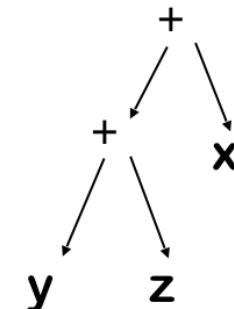
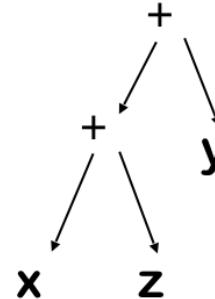
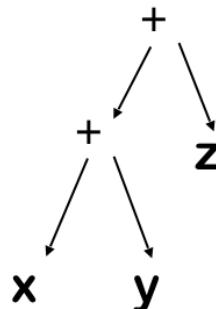
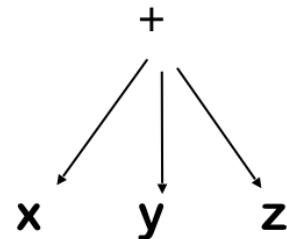
$$x + z \rightarrow t1$$

$$y + z \rightarrow t1$$

$$t1 + z \rightarrow t2$$

$$t1 + y \rightarrow t2$$

$$t1 + x \rightarrow t2$$



- What if the compiler knows that x is constant 2 and z is 3?
The compiler should detect $2+3$ evaluates and fold it into the code
- What if $y+z$ is evaluated earlier?
The “best” shape for $x+y+z$ depends on contextual knowledge
 - There may be several conflicting options

Code Shape

Why worry about code shape? Can't we just trust the optimizer and the back end?

- Optimizer and back end approximate the answers to many hard problems
- The compiler's individual passes must run quickly
- It often pays to encode useful information into the IR
 - Shape of an expression or a control structure
 - A value kept in a register rather than in memory
- Deriving such information may be expensive, when possible
- Recording it explicitly in the IR is often easier and cheaper

How to generate ILOC code

- The three-address form lets the compiler name the result of any operation and preserve it for later reuse
- It uses always new register and leave to the allocator the duty of reduce them
- To generate code for a trivial expression $a+b$ the compiler emits code to ensure that the values of a and b are in registers
- If a is stored in memory at offset $@a$ in the current Activation Record (AR), the code is

$loadI$	$@a$	$\Rightarrow r_1$
$loadA0$	r_{arp}, r_1	$\Rightarrow r_a$

Generating Code for Expressions

THE NODE OF THE AST

The idea

- Assume an **AST** as input and **ILOC** as output
- Use a **postorder treewalk evaluator**
 - > Visits & evaluates children
 - > Emits code for the op itself
 - > Returns register with result
- Bury complexity of addressing names in routines that it calls
 - > **base()**, **offset()** and **val()**
- Works for simple expressions
- Easily extended to other operators

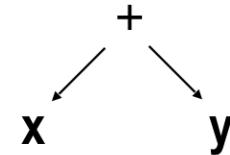
```
expr(node) {  
    register result, t1, t2;  
    switch (type(node)) {  
        case ×,÷,+,- :  
            t1← expr(left child(node));  
            t2← expr(right child(node));  
            result ← NextRegister();  
            emit (op(node), t1, t2, result);  
            break;  
        case IDENTIFIER:  
            t1← base(node);  
            t2 ← NextRegister();  
            emit (loadl, offset(node), none, t2);  
            result ← NextRegister();  
            emit (loadAO, t1, t2, result);  
            break;  
        case NUMBER:  
            result ← NextRegister();  
            emit (loadl, val(node), none, result);  
            break;  
    }  
    return result;  
}
```

Generating Code for Expressions (a naive translation)

```
expr(node) {  
    register result, t1, t2;  
    switch (type(node)) {  
        case ×,÷,+,- :  
            t1← expr(left child(node));  
            t2← expr(right child(node));  
            result ← NextRegister();  
            emit (op(node), t1, t2, result);  
            break;  
        case IDENTIFIER:  
            t1← base(node);  
            t2 ← NextRegister();  
            emit (loadI, offset(node), none, t2);  
            result ← NextRegister();  
            emit (loadAO, t1, t2, result);  
            break;  
        case NUMBER:  
            result ← NextRegister();  
            emit (loadI, val(node), none, result);  
            break;  
    }  
    return result;  
}
```

base(id) loads the right pointer to the AR where id is defined in register r_{arp}

Example:



Produces for register counter 0 :

expr("x"):

NextRegister(): r1 loadI @x → r1

NextRegister(): r2 loadAO rarp, r1 → r2

expr("y"):

NextRegister(): r3 loadI @y → r3

NextRegister(): r4 loadAO rarp, r3 → r4

NextRegister(): r5

Emit(add, r2,r4,r5) :

add r2, r4 → r5

Effects of code shape on the demand of registers

- Code shape decisions encoded into the tree walk code generator have an effect on the demand of registers
- The previous naive code uses 8 registers + r_{arp}
- The register allocator (later in compilation) can reduce the demand for register to $3 + r_{arp}$

Some observations

What if our IDENTIFIER is

- already in a register?
- in a global data area?
- a parameter value?
 - * call by value
 - * call by reference

Extending the Simple Treewalk Algorithm

It assumes a single case for id, more cases for IDENTIFIER

- What about values that reside in registers?
 - Modify the IDENTIFIER case
 - Already in a register \Rightarrow return the register name
 - Not in a register \Rightarrow load it as before, but record the fact
 - Choose names to avoid creating false dependences
- What about parameter values ?
 - Call-by-value \Rightarrow it can be handled as it was a local variable as before
 - Call-by-reference \Rightarrow extra indirection 3 instructions. The value may not be kept in a register across an assignment (see next slide)
- What about function calls in expressions?
 - Generate the calling sequence & load the return value
 - Severely limits compiler's ability to reorder operations

Keeping values in registers

- In a register-to register memory model, the compiler tries to assigns many values as possible to virtual registers
- Then the register allocator will map the set of virtual to physical registers inserting the spills
- However, the compiler can keep values in a register only for unambiguous value:
a value that can be accessed with just one name is unambiguous

The problem with ambiguous values

Consider a and b ambiguous and the following code

```
a := m+n;  
b := 13;  
c:= a+b;
```

If a and b refers to the same location c gets value 26,
otherwise c gets value $m+n+13$;

The compiler cannot keep a in a register during the assignment of b unless it proves that the set of location that the two name refer to are disjoint. This analysis can be expensive!

Where do ambiguous values arise?

Ambiguous values may arise in several ways :

- values stored in a pointer based variable
- call by reference formal parameter
- many compilers treat array element values as ambiguous because they can not tell if two references $A[i,j]$ e $A[n,m]$ refer to the same location

Extending the Simple Treewalk Algorithm

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls (exp. and trig fun.)

Mixed-type expressions

- Insert conversion code as needed from conversion table
- Most languages have symmetric & rational conversion tables

Typical
Table for
Addition

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

If the type cannot be inferred at compile time, the compiler must insert code for run-time checks that test for illegal cases!

Extending the Simple Treewalk Algorithm

What about evaluation order?

Can use commutativity & associativity to improve code for integers

- For recognising that already computed that value

$$a+b = b+a$$

- For recognising that it can compute subexpressions

$$a+b+d \text{ and } c+a+b$$

(it does not if it evaluates the expressions in strict left right order!)

It should not reorder floating point expressions!

- The subset of reals represented on a computer does not preserve associativity

$a-b-c$ the results may depend on the evaluation order!

Handling Assignment

(just another operator)

lhs \leftarrow *rhs*

Strategy

- Evaluate rhs to a **value** (an rvalue)
- Evaluate lhs to a **location** (an lvalue)
 - lvalue is a register \Rightarrow move rhs
 - lvalue is an address \Rightarrow store rhs
- If rvalue & lvalue have different types
 - Evaluate rvalue to its “natural” type
 - Convert that value to the type of *lvalue

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Handling Assignment

What if the compiler cannot determine the type of the rhs?

- It is a property of the language & the specific program
- For type-safety, compiler must insert a run-time check
 - Some languages & implementations ignore safety (bad idea)
- Add a tag field to the data items to hold type information
 - Explicitly check tags at runtime

Code for assignment becomes more complex

```
evaluate rhs  
if type(lhs) ≠ rhs.tag  
then  
    convert rhs to type(lhs) or  
    signal a run-time error  
lhs ← rhs
```

Choice between conversion & a runtime exception depends on details of language & type system
Much more complex than static checking, plus costs occur at runtime rather than compile time

Handling Assignment

Compile-time type-checking

- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation

Can design the language so all checks are static

Summary

Code Generation for Expressions

- Simple treewalk produces reasonable code
 - Execute most demanding subtree first
 - Can implement treewalk explicitly, with an Attributed grammar or ad hoc Syntax directed translation ...
- Handle assignment as an operator
 - Insert conversions according to language-specific rules
 - If compile-time checking is impossible, check tags at runtime

Next computing Array access!

Computing an Array Address of an array $A[\text{low}:\text{high}]$

$A[i]$

- $\text{@}A + (i - \text{low}) \times \text{sizeof}(A[i])$
- In general: $\text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[i])$

Color Code:

Invariant
Varying

Depending on how A is declared, $\text{@}A$ may be

- an offset from the ARP,
- an offset from some global label, or
- an arbitrary address.

The first two are compile time constants.

Computing an Array Address $A[\text{low}:\text{high}]$

$A[i]$

- $\text{@}A + (\text{i} - \text{low}) \times w$
- In general: $\text{base}(A) + (\text{i} - \text{low}) \times w$

where $w = \text{sizeof}(A[i])$

Almost always a power of 2, known at compile-time
⇒ use a shift for speed

If the compiler knows low it can fold the subtraction
into $\text{@}A$

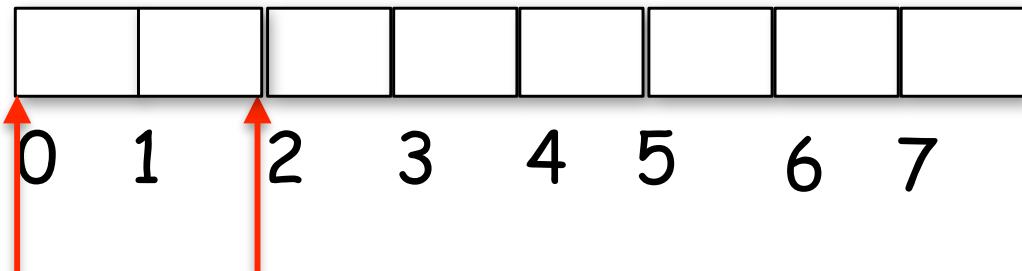
$$A_0 = \text{@}A - (\text{low} * w)$$

The false zero of A

The False Zero

$A[2..7]$

$$A_0 = @A - (low * w)$$



computing $A[i]$ with A

$$\begin{array}{lll} loadI & @A & \Rightarrow r_{@A} \\ subI & r_i, 2 & \Rightarrow r_1 \\ lshiftI & r_1, 2 & \Rightarrow r_2 \\ loadA0 & r_{@A}, r_2 & \Rightarrow r_v \end{array}$$

computing $A[i]$ with A_0

$$\begin{array}{lll} loadI & @A_0 & \Rightarrow r_{@A_0} \\ lshiftI & r_i, 2 & \Rightarrow r_1 \\ loadA0 & r_{@A_0}, r_1 & \Rightarrow r_v \end{array}$$

How does the compiler handle $A[i,j]$?

First, must agree on a storage scheme

Row-major order (most languages)

Lay out as a sequence of consecutive rows

Rightmost subscript varies fastest

$A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$

Column-major order (Fortran)

Lay out as a sequence of columns

Leftmost subscript varies fastest

$A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]$

Indirection vectors (Java)

Vector of pointers to pointers to ... to values

Takes much more space, trades indirection for arithmetic

Not amenable to analysis

Laying Out Arrays

The Concept

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

Row-major order

A

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4
-----	-----	-----	-----	-----	-----	-----	-----

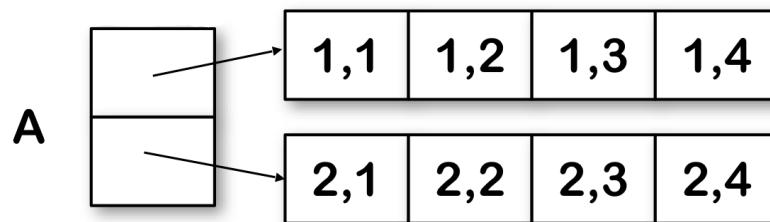
Column-major order

A

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4
-----	-----	-----	-----	-----	-----	-----	-----

These can have distinct & different cache behavior

Indirection vectors



Computing an Array Address

$A[i]$

- $\text{@}A + (i - \text{low}) \times w$
- In general: $\text{base}(A) + (i - \text{low}) \times w$

where $w = \text{sizeof}(A[1,1])$

$\text{low}_1 \quad \text{low}_2 \quad \text{high}_2$

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

What about $A[i_1, i_2]$?

Row-major order, two dimensions

$$\text{@}A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times w$$
$$A[2,3] \quad \text{@}A + (2-1) \times 4 + (3-1)$$

This stuff looks expensive!
Lots of implicit +, -, \times ops

Column-major order, two dimensions

$$\text{@}A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times w$$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

e.g., $\text{@}A + (i_1 - \text{low}) \times w$

Computing an Array Address

In row-major order

$$@A + (i - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w$$

Which can be factored into

$$@A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w$$

$$- (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) - (\text{low}_2 \times w)$$

If low_i , high_i , and w are known, the last term is a constant

Define $@A_0$ as

$$@A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) - \text{low}_2 \times w$$

And len_2 as $(\text{high}_2 - \text{low}_2 + 1)$

Then, the address expression becomes

$$@A_0 + (i \times \text{len}_2 + j) \times w$$

	low_1	low_2	high_2
high_1	1,1	1,2	1,3
A	2,1	2,2	2,3
			1,4

If $@A$ is known, $@A_0$ is a known constant.

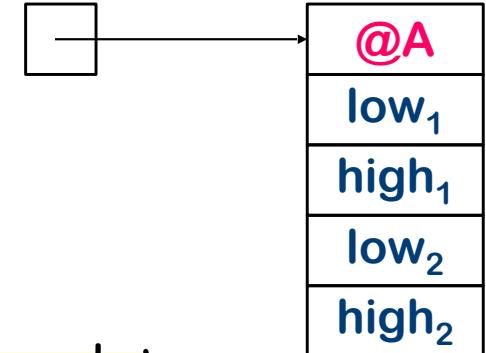
Compile-time constants

Array References

What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Need dimension information \Rightarrow build a **dope vector**
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference



Some improvement is possible

- Choose the address polynomial based on the false zero
- Pre-compute the fixed terms in prologue sequence

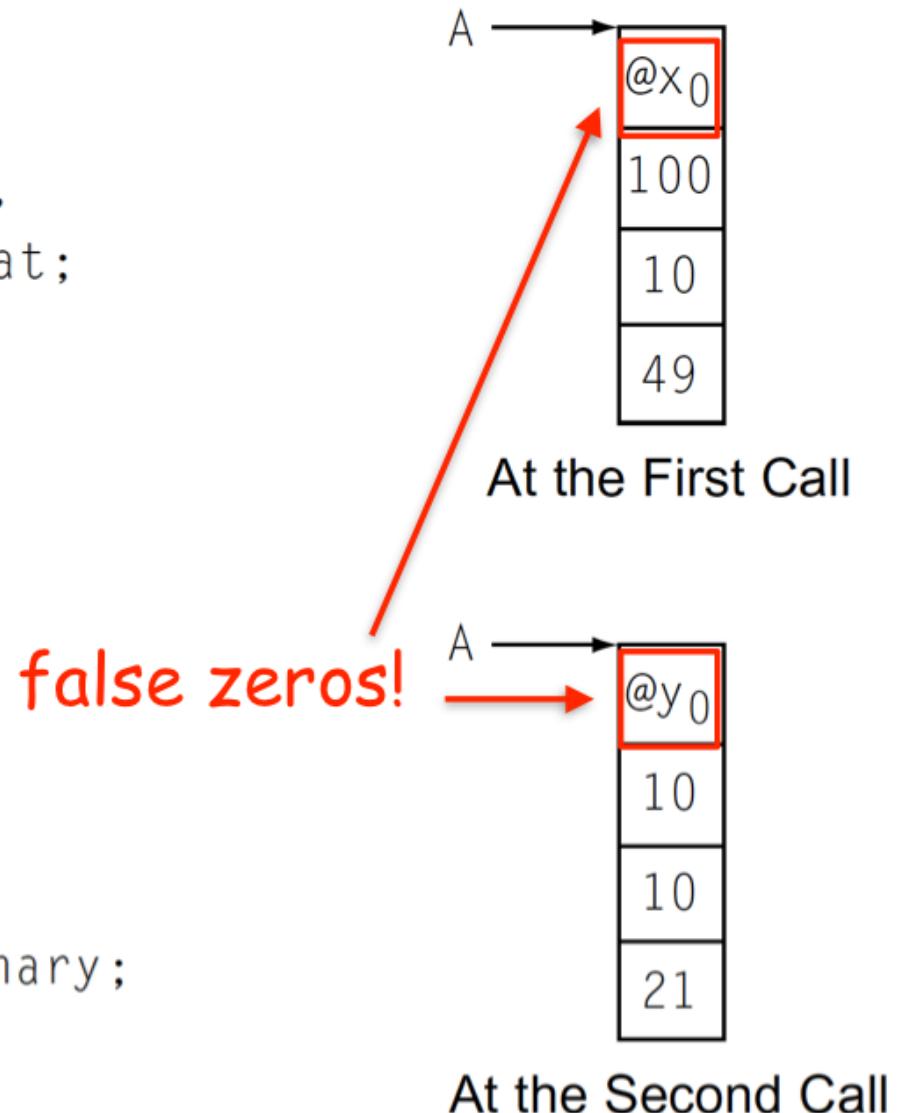
What about call-by-value?

- Most languages pass arrays by reference
- This is a language design issue

The Dope vector

```
program main;
begin;
    declare x(1:100,1:10,2:50),
            y(1:10,1:10,15:35) float;
    ...
    call fee(x)
    call fee(y);
end main;

procedure fee(A)
declare A(*,*,*) float;
begin;
    declare x float;
    declare i, j, k fixed binary;
    ...
    x = A(i,j,k);
    ...
end fee;
```



Range checking

A program that refers out-of-the-bound array elements is not well formed.

Some languages like Java require out-of-the-bound accesses be detected and reported.

In other languages compilers have included mechanisms to detect and report out-of-the-bound accesses.

The easy way is to introduce a runtime check that verifies that the index value falls in the array range



Expensive!!

the compiler has to prove
that a given reference cannot
generate an out-of-bounds reference

Information on the bounds in the dope vector

Array Address Calculations

Array address calculations are a major source of overhead

- Scientific applications make extensive use of arrays and array-like structures
 - Computational linear algebra, both dense & sparse
- Non-scientific applications use arrays, too
 - Representations of other data structures
 - Hash tables, adjacency matrices, tables, structures, ...

Array calculations tend iterate over arrays

- Loops execute more often than code outside loops
- Array address calculations inside loops make a huge difference in efficiency of many compiled applications

Reducing array address overhead has been a major focus of optimization since the 1950s.

Example: Array Address Calculations in a Loop

```
DO J = 1, N
```

$$A[I, J] = A[I, J] + B[I, J]$$

```
END DO
```

A, B are declared as conformable
floating-point arrays

In column-major order

$$@A_0 + (j \times \text{len}_1 + i) \times w$$

number of rows!

Naïve: Perform the address calculation twice

```
DO J = 1, N
```

$$R1 = @A_0 + (J \times \text{len}_1 + I) \times w$$

$$R2 = @B_0 + (J \times \text{len}_1 + I) \times w$$

$$\text{MEM}(R1) = \text{MEM}(R1) + \text{MEM}(R2)$$

```
END DO
```

Example: Array Address Calculations in a Loop

More sophisticated: Move common calculations out of loop

$$R1 = I \times w$$

$$c = \text{len}_1 \times w \quad ! \text{ Compile-time constant}$$

$$R2 = @A_0 + R1$$

$$R3 = @B_0 + R1$$

DO J = 1, N

$$a = J \times c$$

$$R4 = R2 + a$$

$$R5 = R3 + a$$

$$\text{MEM}(R4) = \text{MEM}(R4) + \text{MEM}(R5)$$

END DO

Loop-invariant code motion

Example: Array Address Calculations in a Loop

Very sophisticated: Convert multiply to add

$$R1 = I \times w$$

$$c = \text{len}_1 \times w \quad ! \text{ Compile-time constant}$$

J is now bookkeeping

$$R2 = @A_0 + R1 ;$$

$$R3 = @B_0 + R1;$$

DO J = 1, N

$$R2 = R2 + c$$

$$R3 = R3 + c$$

$$\text{MEM}(R2) = \text{MEM}(R2) + \text{MEM}(R3)$$

END DO

Operator Strength Reduction (§ 10.4.2 in EaC)

Representing and Manipulating Strings

Character strings differ from scalars, arrays, & structures

- Languages support can be different:
 - In C most manipulations takes the form of calls to library routines
 - Other languages provide first-class mechanism to specify substrings or concatenate them
- Fundamental unit is a character
 - Typical sizes are one or two bytes
 - Target ISA may (or may not) support character-size operations

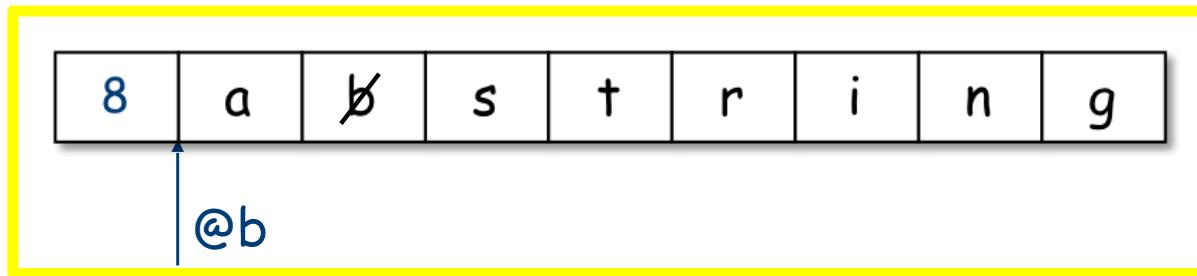
String operation can be costly

- Older CISC architectures provide extensive support for string manipulation
- Modern RISC architectures rely on compiler to code this complex operations using a set a of simpler operations

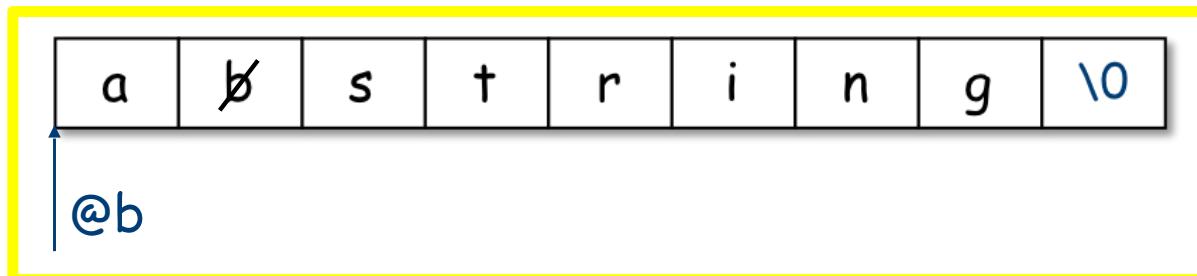
Representing and Manipulating Strings

Two common representations of string "a string"

- Explicit length field



- Null termination



- Language design issue

Length field may take more space than terminator

Representing and Manipulating Strings

Each representation has advantages and disadvantages

Operation	Explicit Length	Null Termination
Assignment	Straightforward	Straightforward
Checked Assignment	Checking is easy	Must count length
Length	$O(1)$	$O(n)$
Concatenation	Must copy data	Length + copy data

Unfortunately, null termination is almost considered normal

- Hangover from design of C
- Embedded in OS and API designs

Manipulating Strings

Single character assignment $a[1]=b[2]$

- With character operations
 - Compute address of rhs, load character
 - Compute address of lhs, store character
- With only word operations (>1 char per word)
 - Compute address of word containing rhs & load it
 - Move character to destination position within word
 - Compute address of word containing lhs & load it
 - Mask out current character & mask in new character
 - Store lhs word back into place

Manipulating Strings

Multiple character assignment

Two strategies

1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

With character operations

With only word operations

Manipulating Strings

Concatenation

- String concatenation is a length computation followed by a pair of whole-string assignments
 - Touches every character
 - There can be length problems!

Manipulating Strings

Length Computation

- Representation determines cost
- Length computation arises in other contexts
 - Whole-string or substring assignment
 - Checked assignment (buffer overflow)
 - Concatenation
 - Evaluating call-by-value actual parameter

Boolean & Relational Values

How should the compiler represent them?

- Answer depends on the target machine

Implementation of booleans, relational
expressions & control flow constructs
varies widely with the ISA

Two classic approaches

- Numerical (explicit) representation
- Positional (implicit) representation

Best choice depends on both context and ISA

Some cases works better with the first
representation other ones with the second!

Boolean & Relational Expressions

First, we need to recognize boolean & relational expressions

Expr	\rightarrow Expr \vee AndTerm	NumExpr	\rightarrow NumExpr + Term
	AndTerm		NumExpr - Term
AndTerm	\rightarrow AndTerm \wedge RelExpr		Term
	RelExpr	Term	\rightarrow Term \times Value
RelExpr	\rightarrow RelExpr $<$ NumExpr		Term \div Value
	RelExpr \leq NumExpr		Value
	RelExpr = NumExpr	Value	\rightarrow \neg Factor
	RelExpr \neq NumExpr		Factor
	RelExpr \geq NumExpr	Factor	(Expr)
	RelExpr $>$ NumExpr		number

Boolean & Relational Values

Next, we need to represent the values

Numerical representation

- Assign numerical values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations
- Use comparison to get a boolean from a relational

If the target machine supports boolean operations that compute the boolean result $\text{cmp_LT } rx, ry \rightarrow r_1 \quad r_1 = \text{True if } rx < ry, r_1 = \text{False otherwise}$

$x < y$ becomes $\text{cmp_LT } r_x, r_y \Rightarrow r_1$

$\text{if } (x < y)$
 then stmt_1 becomes $\text{cmp_LT } r_x, r_y \Rightarrow r_1$
 else stmt_2 $\text{cbr} \quad r_1 \rightarrow _\text{stmt}_1, _\text{stmt}_2$

Boolean & Relational Values

What if the target machine uses a **condition code** instead than **boolean operations** as `cmp_LT`?

`cmp r1,r2 -> cc` sets cc with code for LT,LE,EQ,GE,GT,NE

- Must use a conditional branch to interpret result of compare

If the target machine computes a code result of the comparison and we need to store the result of the boolean operation

$x < y$	becomes	<code>cmp</code>	r_x, r_y	\Rightarrow	CC_1
cbr_LT CC 2, 3 sets PC= 2 if CC=LT PC= 3 otherwise		<code>cbr_LT</code>	CC_1	\rightarrow	L_T, L_F
$L_T:$		<code>loadl</code>	1	\Rightarrow	r_2
$L_F:$		<code>br</code>		\rightarrow	L_E
$L_E:$		<code>loadl</code>	0	\Rightarrow	r_2
		<i>... other statements ...</i>			

Boolean & Relational Values

The last example actually encoded result in r2

If result is used to control an operation we may not need to write explicitly the result! **Positional encoding!**

Example

```
if (x < y)
    then a ← c + d
    else a ← e + f
```

Straight Condition Codes			Boolean Comparisons		
comp	r_x, r_y	$\Rightarrow CC_1$	cmp_LT	r_x, r_y	$\Rightarrow r_1$
cbr_LT	CC_1	$\rightarrow L_1, L_2$	cbr	r_1	$\rightarrow L_1, L_2$
L_1 : add	r_c, r_d	$\Rightarrow r_a$	L_1 : add	r_c, r_d	$\Rightarrow r_a$
br		$\rightarrow L_{OUT}$	br		$\rightarrow L_{OUT}$
L_2 : add	r_e, r_f	$\Rightarrow r_a$	L_2 : add	r_e, r_f	$\Rightarrow r_a$
L_{OUT} : nop			L_{OUT} : nop		

Short-circuit Evaluation

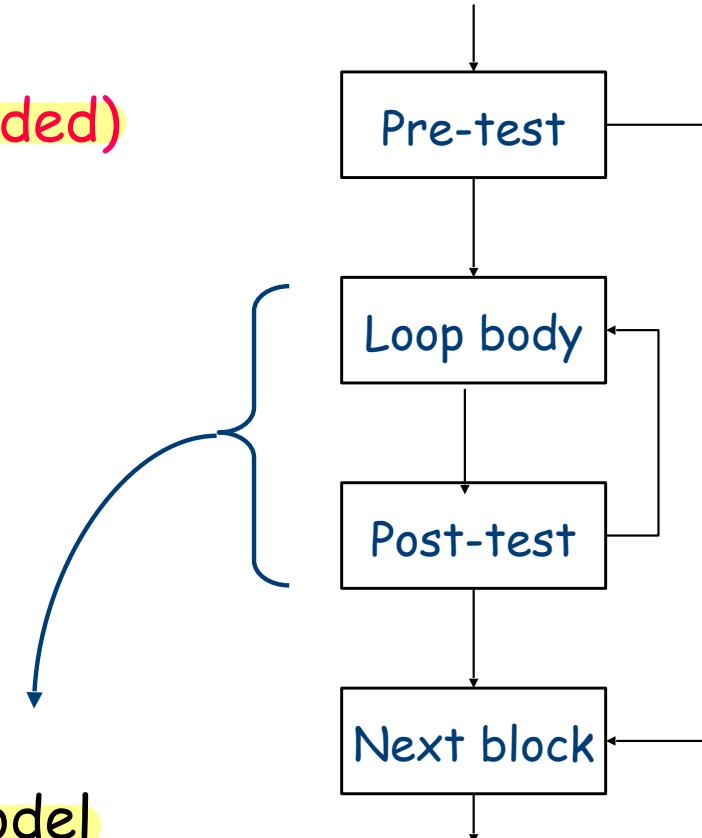
Optimize boolean expression evaluation (**lazy evaluation**)

- Once value is determined, skip rest of the evaluation
 - if (x or y and z) then ...
 - If x is true, need not evaluate y or z
 - Branch directly to the "then" clause
 - On a PDP-11 or a VAX, short circuiting saved time
- Modern architectures may favor evaluating full expression
 - Rising branch latencies make the short-circuit path expensive
 - Conditional move and predication may make full path cheaper
- Past: compilers analyzed code to insert short circuits
- Future: compilers analyze code to prove legality of full path evaluation where language specifies short circuits

Control Flow

Loops

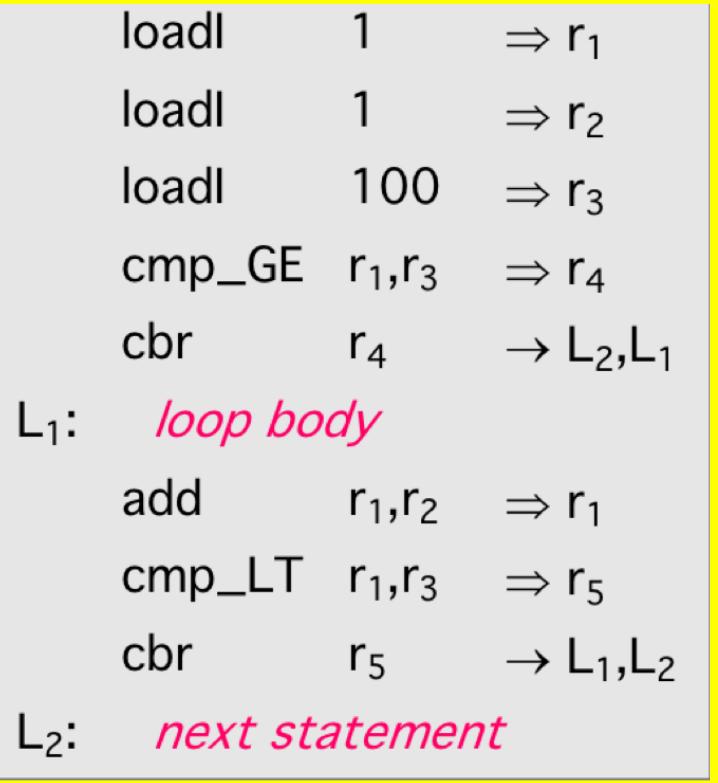
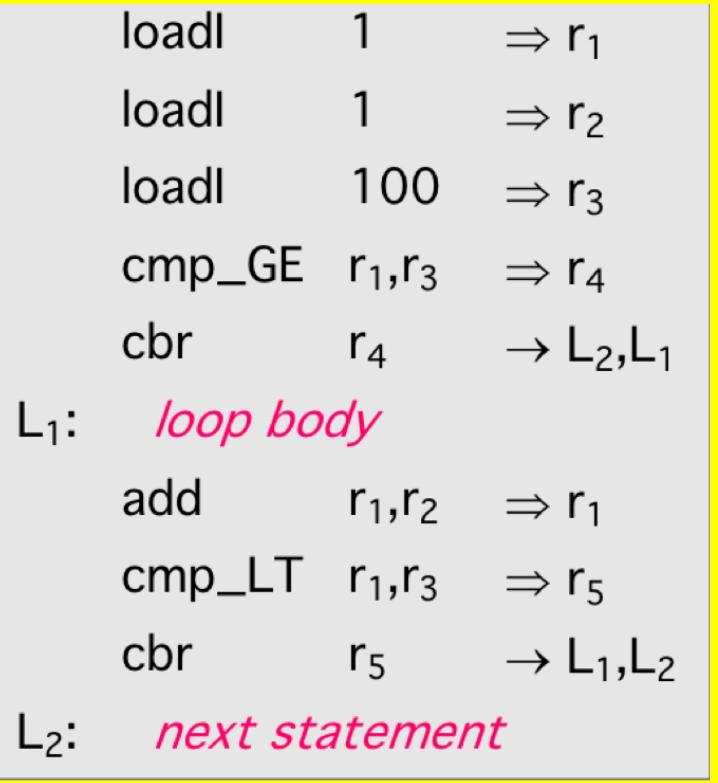
- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)



while, for, do, & until all fit this basic model

Implementing Loops

```
for (i = 1; i < 100; 1) { loop body }  
next statement
```

loadl 1 ⇒ r ₁		}	Initialization
loadl 1 ⇒ r ₂			
loadl 100 ⇒ r ₃			
cmp_GE r ₁ ,r ₃ ⇒ r ₄			
cbr r ₄ → L ₂ ,L ₁			
L ₁ : loop body		}	Pre-test
add r ₁ ,r ₂ ⇒ r ₁			
cmp_LT r ₁ ,r ₃ ⇒ r ₅			
cbr r ₅ → L ₁ ,L ₂			
L ₂ : next statement			

Case (switch) Statements

- 1 Evaluate the controlling expression
- 2 Branch to the selected case
- 3 Execute the code for that case
- 4 Branch to the statement after the case (use break)

Parts 1, 3, & 4 are well understood,

part 2 is the key:

need an efficient method to locate the designated code

many compilers provide several different search schemas each one
can be better in some cases.

Case Statements

Strategies

- Linear search (nested if-then-else constructs)
- Build a table of case expressions & binary search it
- Directly compute address (requires dense case set)



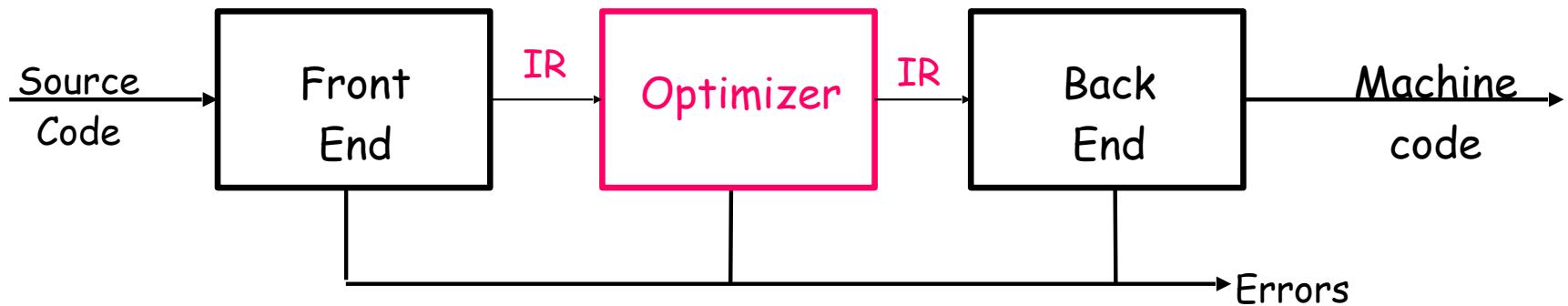
This lecture begins
the material from
Chapter 8 of EaC

Introduction to Code Optimization

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

Traditional Three-Phase Compiler



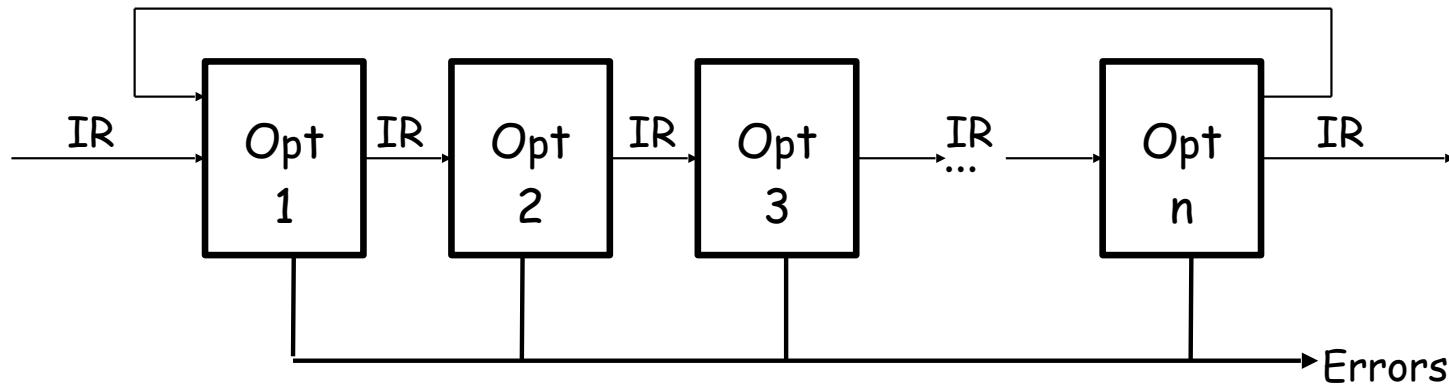
Optimization (or Code Improvement)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
 - May also improve space, power consumption, ...

Transformations have to be:

- Safely applied and (it does not change the result of the running program)
- Applied when profit has expected

The Optimizer



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code

The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is “better”
 - Speed, code size, data space, ...

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
 - Data-flow analysis, pointer disambiguation, ...
 - General term is “static analysis”
- Uses that knowledge in an attempt to improve the code
 - Literally hundreds of transformations have been proposed
 - Large amount of overlap between them

Nothing “optimal” about optimization

- Proofs of optimality assume restrictive & unrealistic conditions

Scope of Optimization

In scanning and parsing, "scope" refers to a region of the code that corresponds to a distinct name space.

In optimization "scope" refers to a region of the code that is subject to analysis and transformation.

- Notions are somewhat related
- Connection is not necessarily intuitive

Different scopes introduces different challenges & different opportunities

Historically, optimization has been performed at several distinct scopes.

Scope of Optimization

Local optimization

- Operates entirely within a single basic block
- Properties of block lead to strong optimizations

Regional optimization

- Operate on a region in the CFG that contains multiple blocks
- Loops, trees, paths, extended basic blocks

Whole procedure optimization (intraprocedural)

- Operate on entire CFG for a procedure

Whole program optimization (interprocedural)

- Operate on some or all of the call graph (multiple procedures)
- Must contend with call/return & parameter binding

Redundancy Elimination as an Example

An expression $x+y$ is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that $x+y$ is redundant, or available
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

Rewriting to avoid Redundancy

$$a \leftarrow b + c$$
$$b \leftarrow a - d$$
$$c \leftarrow b + c$$
$$d \leftarrow a - d$$

Original Block

$$a \leftarrow b + c$$
$$b \leftarrow a - d$$
$$c \leftarrow b + c$$
$$d \leftarrow b$$

Rewritten Block

The resulting code runs more quickly but extend the lifetime of b
This could cause the allocator to spill the value of b

Since the optimiser cannot predict the behaviour of the register allocator, it assumes that rewriting to avoid redundancy is profitable!

Local Value Numbering

The key notion

- Assign an identifying number, $V(e)$, to each identifier, constant or expression in general with the following property:
 - $V(e1) = V(e2)$ iff $e1$ and $e2$ always have the same value for all possible operand
 - Use hashing over the value numbers to make it efficient
- Use these numbers to improve the code

Improving the code

- Replace redundant expressions
 - Same $V(e) \Rightarrow$ refer rather than recompute

Local Value Numbering

The Algorithm

For each operation $o = \langle \text{operator}, o_1, o_2 \rangle$ in the block, in order

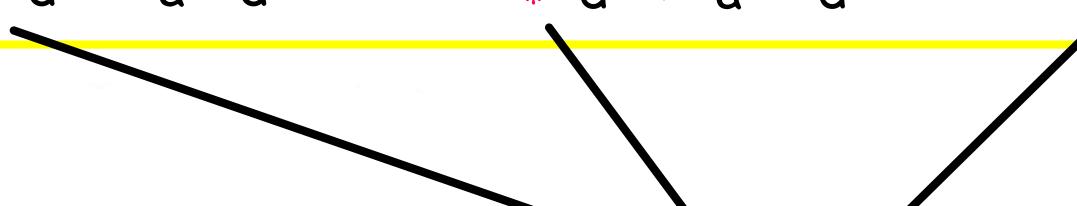
1. Get value numbers $\text{VN}(o_1)$ and $\text{VN}(o_2)$ for operands from hash lookup
2. Hash $\langle \text{operator}, \text{VN}(o_1), \text{VN}(o_2) \rangle$ to get a value number for o
3. If o already had a value number, replace o with a reference $\langle \text{operator}, \text{VN}(o_1), \text{VN}(o_2) \rangle$

If hashing behaves, the algorithm runs in linear time

Local Value Numbering

An example

<u>Original Code</u>	<u>With VNs</u>	<u>Rewritten</u>
$a \leftarrow b + c$	$a^3 \leftarrow b^1 + c^2$	$a \leftarrow b + c$
$b \leftarrow a - d$	$b^5 \leftarrow a^3 - d^4$	$b \leftarrow a - d$
$c \leftarrow b + c$	$c^6 \leftarrow b^5 + c^2$	$c \leftarrow b + c$
* $d \leftarrow a - d$	* $d^5 \leftarrow a^3 - d^4$	* $d \leftarrow b$

Three black arrows originate from the bottom of each column in the table and point towards a light gray rectangular box at the bottom center. The box contains the text "One redundancy" and a bulleted list.

One redundancy

- Eliminate stmt with *

Local Value Numbering: the role of naming

Original Code

$a \leftarrow x + y$

$b \leftarrow x + y$

$a \leftarrow 17$

$c \leftarrow x + y$

With VNs

$a^3 \leftarrow x^1 + y^2$

$b^3 \leftarrow x^1 + y^2$

$a^4 \leftarrow 17$

$c^3 \leftarrow x^1 + y^2$

Rewritten

$a^3 \leftarrow x^1 + y^2$

* $b^3 \leftarrow a^3$

$a^4 \leftarrow 17$

* $c^3 \leftarrow a^3$ (oops!)

Two redundancy

- Eliminate stmt with *

Options

- Use $c^3 \leftarrow b^3$

with a mapping from values
to names

- Save a^3 in t^3
- Rename around it

Local Value Numbering: renaming

Example (continued):

<u>Original Code</u>	<u>With VNs</u>	<u>Rewritten</u>
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:

- Give each value a unique name
- Makes it clear

Notation:

- While complex, the meaning is clear

Result:

- a_0^3 is available
- Rewriting now works

How to reconcile this new subscripted names with the original ones? A clever implementation would map

$$a_1 \rightarrow a$$

$$b_0 \rightarrow b$$

$$c_0 \rightarrow c$$

$$a_0 \rightarrow t$$

The impact of indirect assignments on SSA form

- To manage the subscripted naming the compiler maintain a map from names to the current subscript.
- With a direct assignment $a \leftarrow b + c$, the changes are clear
- With an indirect assignment $*p \leftarrow 0$?
- The compiler can perform static analysis to disambiguate pointer references (to restrict the set of variables to whom p can refer to).

Ambiguous reference

the compiler cannot isolate a single memory location

Simple Extensions to Value Numbering

Commutative operations

- commutative operations that differ only for the order of their operands should receive the same value numbers $a \times b$ and $b \times a$
Impose an order !!

Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace an operation with load of the immediate value

Algebraic identities

- Must check (many) special cases
(organize them into operator-specific decision tree)
- Replace result with input VN

Identities (on VNs)

$x \leftarrow y, x + 0, x - 0, x * 1, x / 1, x - x, x * 0,$
 $x / x, x \vee 0, x \wedge x, \dots$
 $\max(x, \text{MAXINT}), \min(x, \text{MININT}),$
 $\max(x, x), \min(y, y), \text{and so on} \dots$

Local Value Numbering (Recap)

The LVN Algorithm, with bells & whistle

for $i \leftarrow 0$ to $n-1$

1. get the value numbers V_1 and V_2 for L_i and R_i
2. if L_i and R_i are both constant then Constant folding
evaluate $L_i \text{ Op}_i R_i$, assign it to T_i and mark T_i as a constant
3. if $L_i \text{ Op}_i R_i$ matches an identity then Algebraic identities
replace it with a copy operation or an assignment
4. if Op_i commutes and $V_1 > V_2$ then Commutativity
swap V_1 and V_2
5. construct a hash key $\langle V_1, \text{Op}_i, V_2 \rangle$
 - if the hash key is already present in the table then
replace operation I with a copy into T_i and mark T_i with the VN
 - else
insert a new VN into table for hash key & mark T_i with the VN

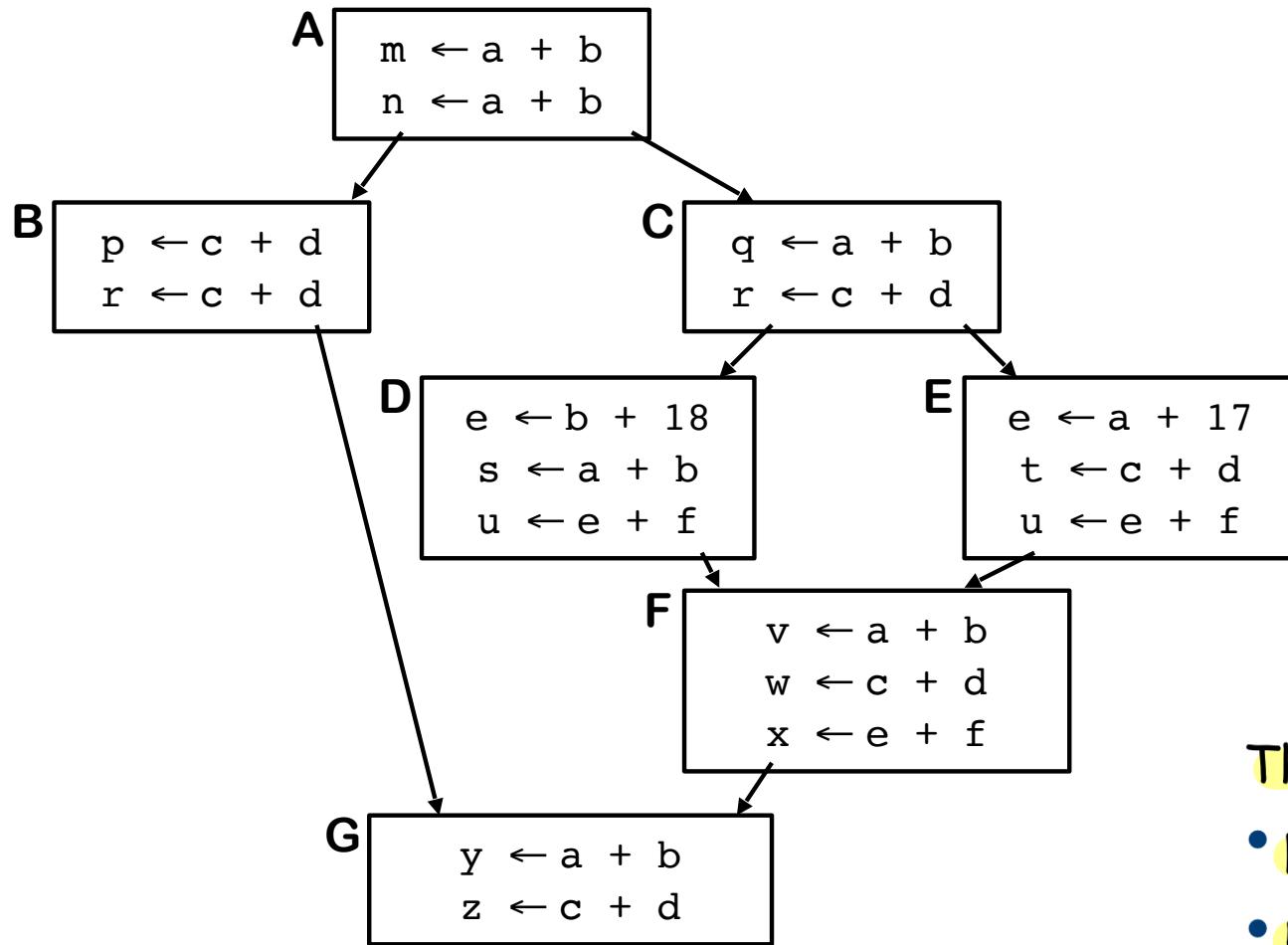
Block is a sequence of n operations of the form
 $T_i \leftarrow L_i \text{ Op}_i R_i$

Local Value Numbering

Complexity & Speed Issues

- "Get value numbers" – linear search versus hash
- " $\text{Hash } \langle \text{op}, \text{VN}(o_1), \text{VN}(o_2) \rangle$ " – linear search versus hash
- Copy folding – set value number of result
- Commutative ops – double hash versus sorting the operands

Terminology Control-flow graph (CFG)

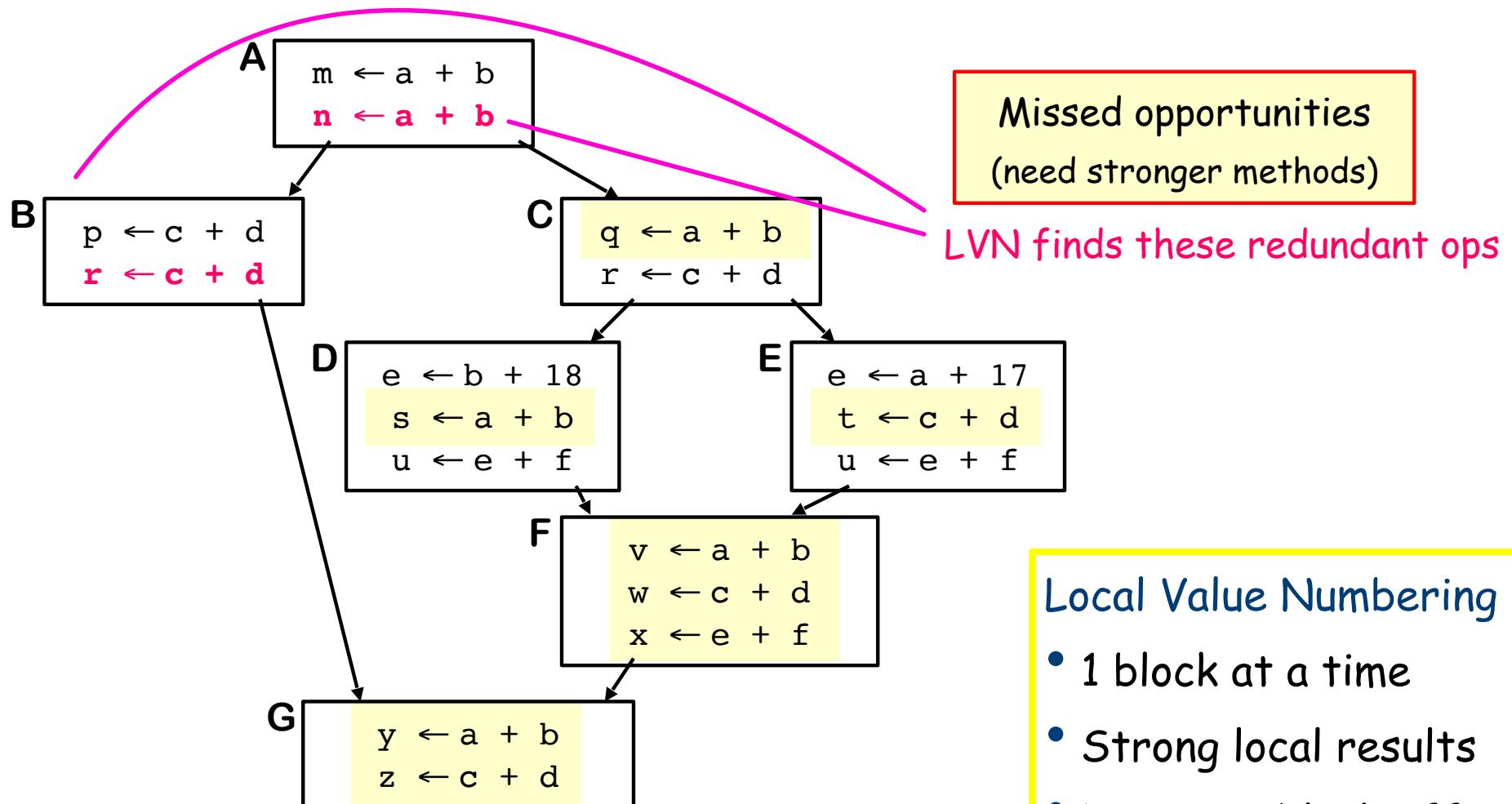


- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

This CFG, $G = (N, E)$

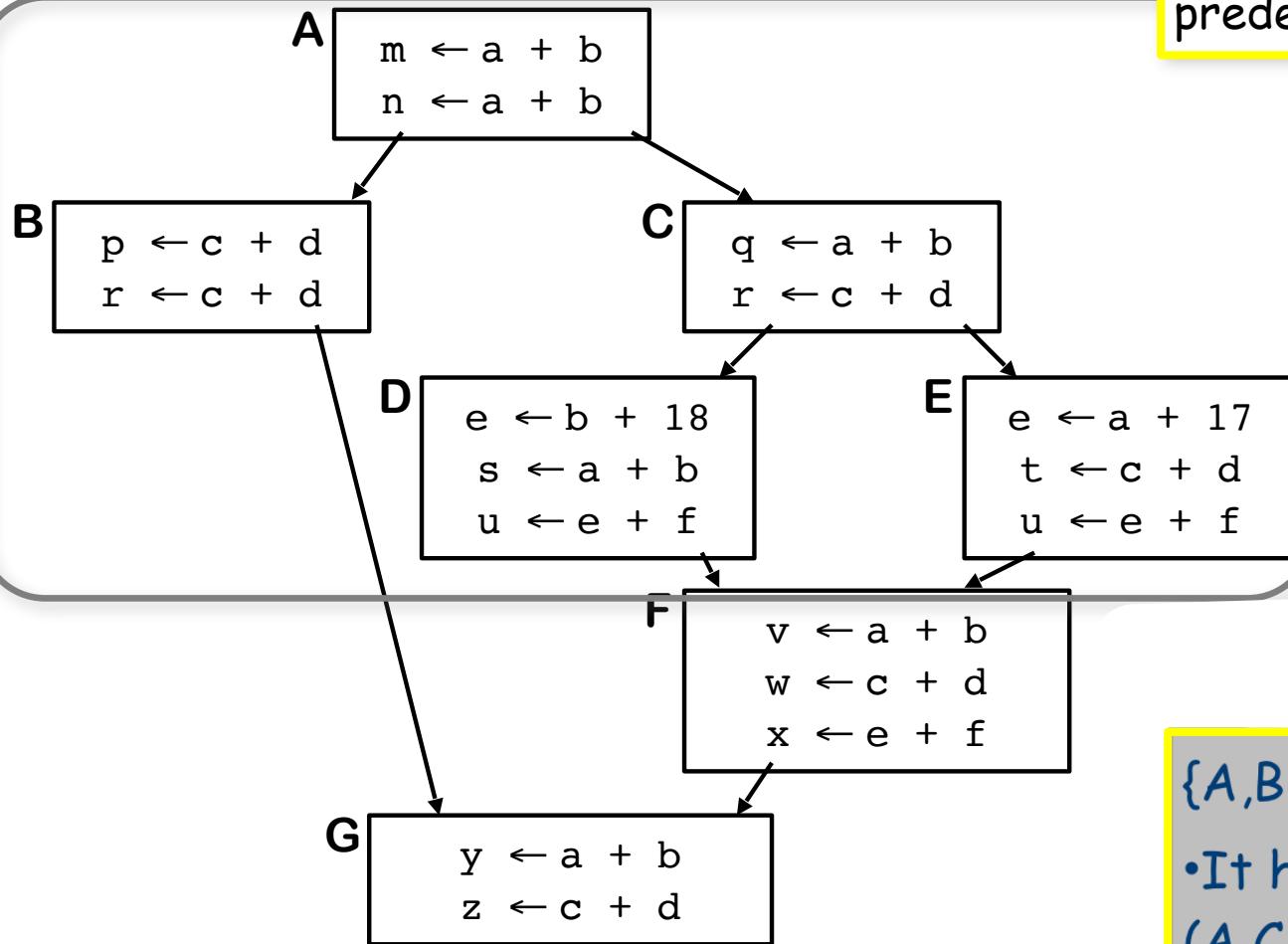
- $N = \{A, B, C, D, E, F, G\}$
- $E = \{(A, B), (A, C), (B, G), (C, D), (C, E), (D, F), (E, F), (F, E)\}$
- $|N| = 7, |E| = 8$

Local Value Numbering



A Regional Technique

Superlocal Value Numbering

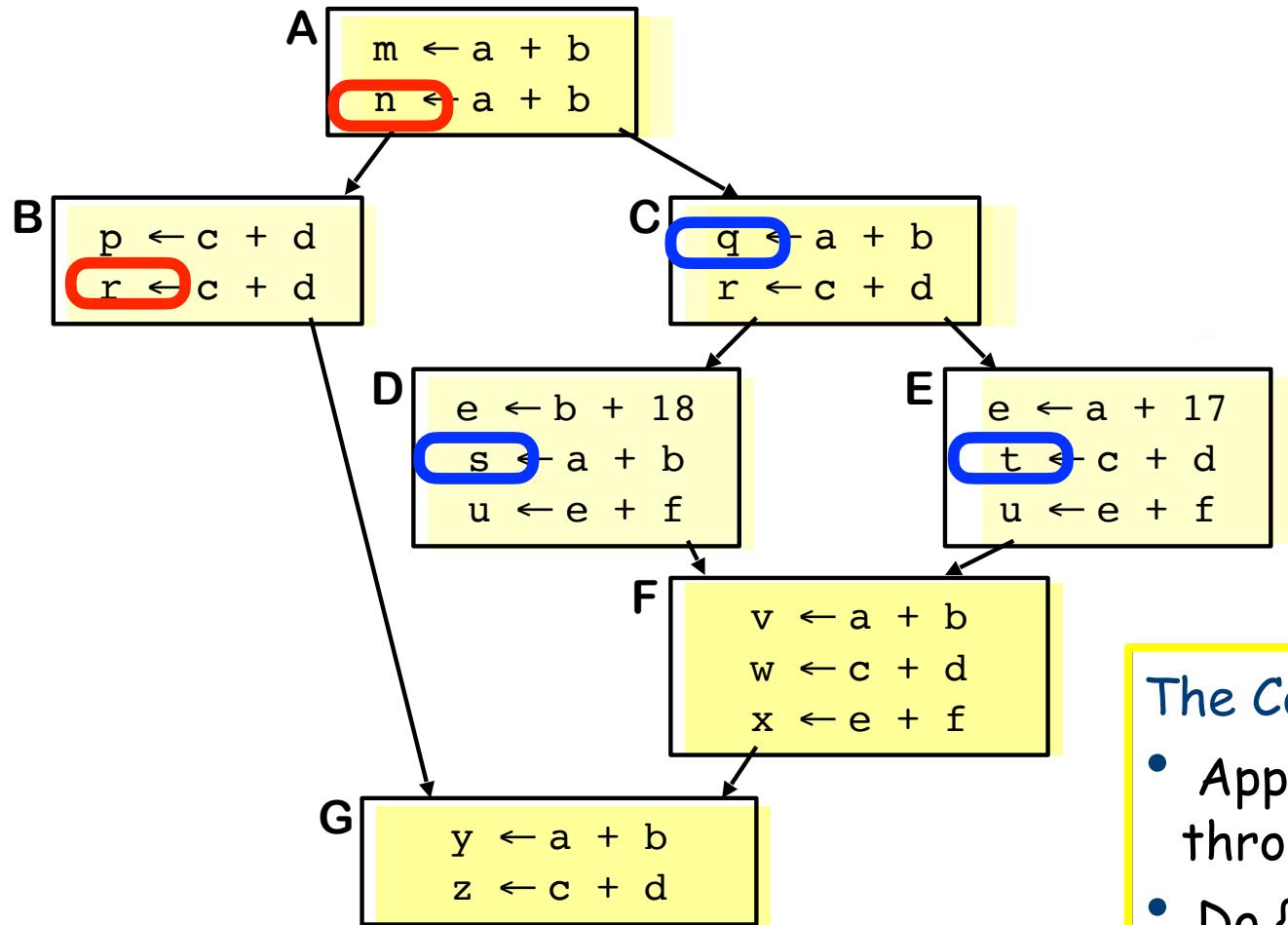


Extended Basic Block: maximal set of blocks B_1, B_2, \dots, B_n where each B_i , except B_1 , has exactly one predecessor in the EBB itself.

{A,B,C,D,E} is an EBB

- It has 3 paths: (A,B), (A,C,D), & (A,C,E)
- Can sometimes treat each path as if it were a block
- {F} & {G} are degenerate EBBs

Superlocal Value Numbering



The Concept

- Apply local method to paths through the EBBs
- Do {A,B}, {A,C,D}, & {A,C,E}
- Obtain reuse from ancestors
- Avoid re-analyzing A & C
- Does not help with F or G

Superlocal Value Numbering

Efficiency

- Use A's table to initialize tables for B & C
- To avoid duplication, use a scoped hash table
 - A, AB, A, AC, ACD, AC, ACE, F, G
- Need a $VN \rightarrow \text{name}$ mapping to handle kills
 - Must restore map with scope
 - Adds complication, not cost

"kill" is a re-definition
of some name

To simplify THE PROBLEM

- Need unique name for each definition
- Use the SSA name space

SSA Name Space

(locally)

Example (from earlier):

<u>Original Code</u>	<u>With VNs</u>	<u>Rewritten</u>
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:

- Give each value a unique name
- Makes it clear

Notation:

- While complex, the meaning is clear

Result:

- a_0^3 is available
- Rewriting just works

SSA Name Space

(in general)

Two principles

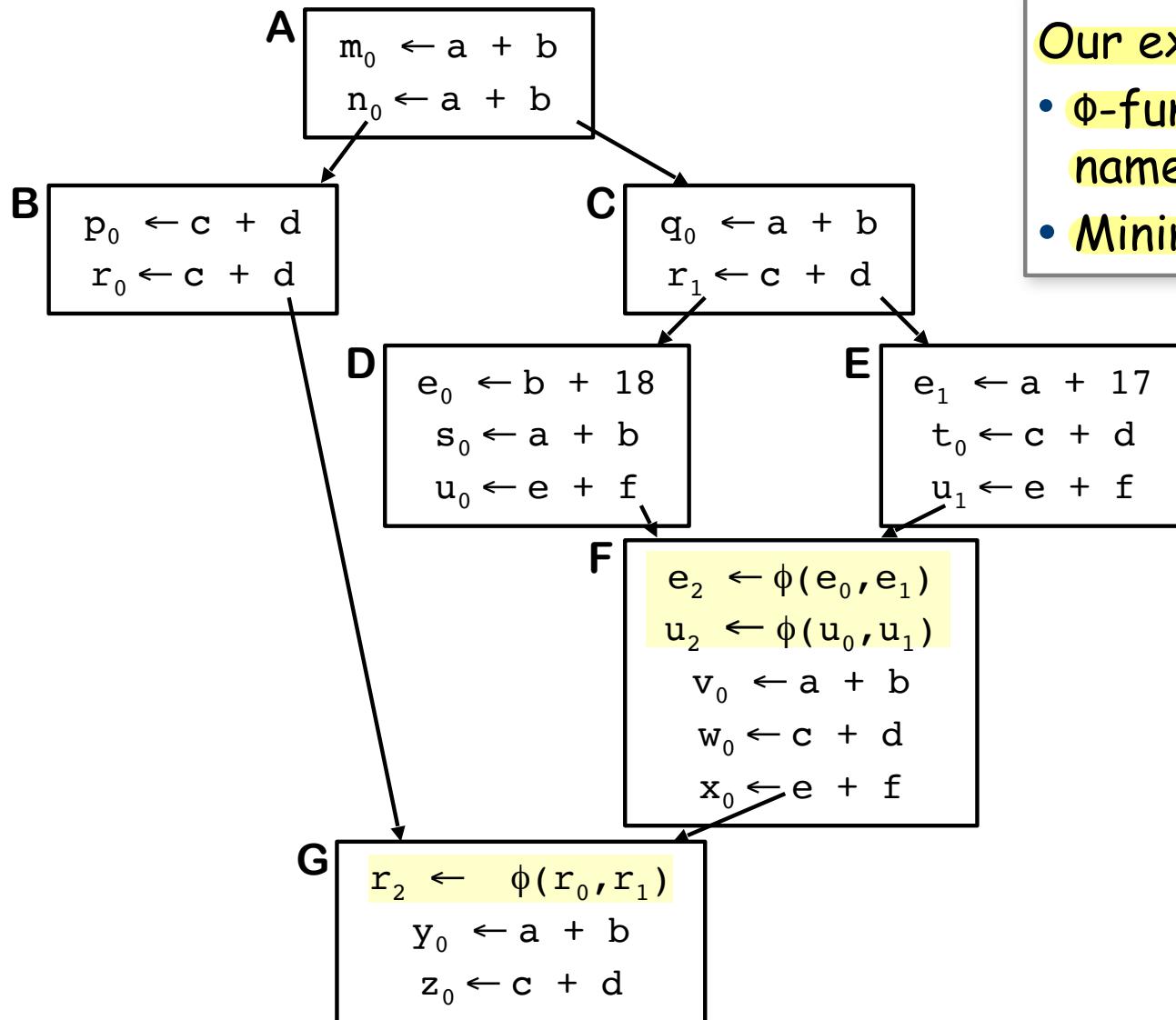
- Each name is defined by exactly one operation
- Each operand refers to exactly one definition

To reconcile these principles with real code

- Insert ϕ -functions at merge points to reconcile name space
- Add subscripts to variable names for uniqueness



Superlocal Value Numbering



Our example in SSA form

- ϕ -functions at join points for names that need them
- Minimal set of ϕ -functions

Superlocal Value Numbering

The SVN Algorithm

```
WorkList  $\leftarrow \{ \text{entry block} \}$ 
Empty  $\leftarrow \text{new table}
\text{while (WorkList is not empty)} \\
\quad \text{remove a block } b \text{ from WorkList} \\
\quad \text{SVN}(b, \text{Empty})$ 
```

```
SVN( Block, Table)
```

t \leftarrow new table for Block, with Table linked as surrounding scope

```
LVN( Block, t)
```

```
for each successor s of Block
```

```
if s has just 1 predecessor
```

```
then SVN( s, t )
```

```
else if s has not been processed
```

```
then add s to WorkList
```

```
deallocate t
```

Blocks to process

Table for base case

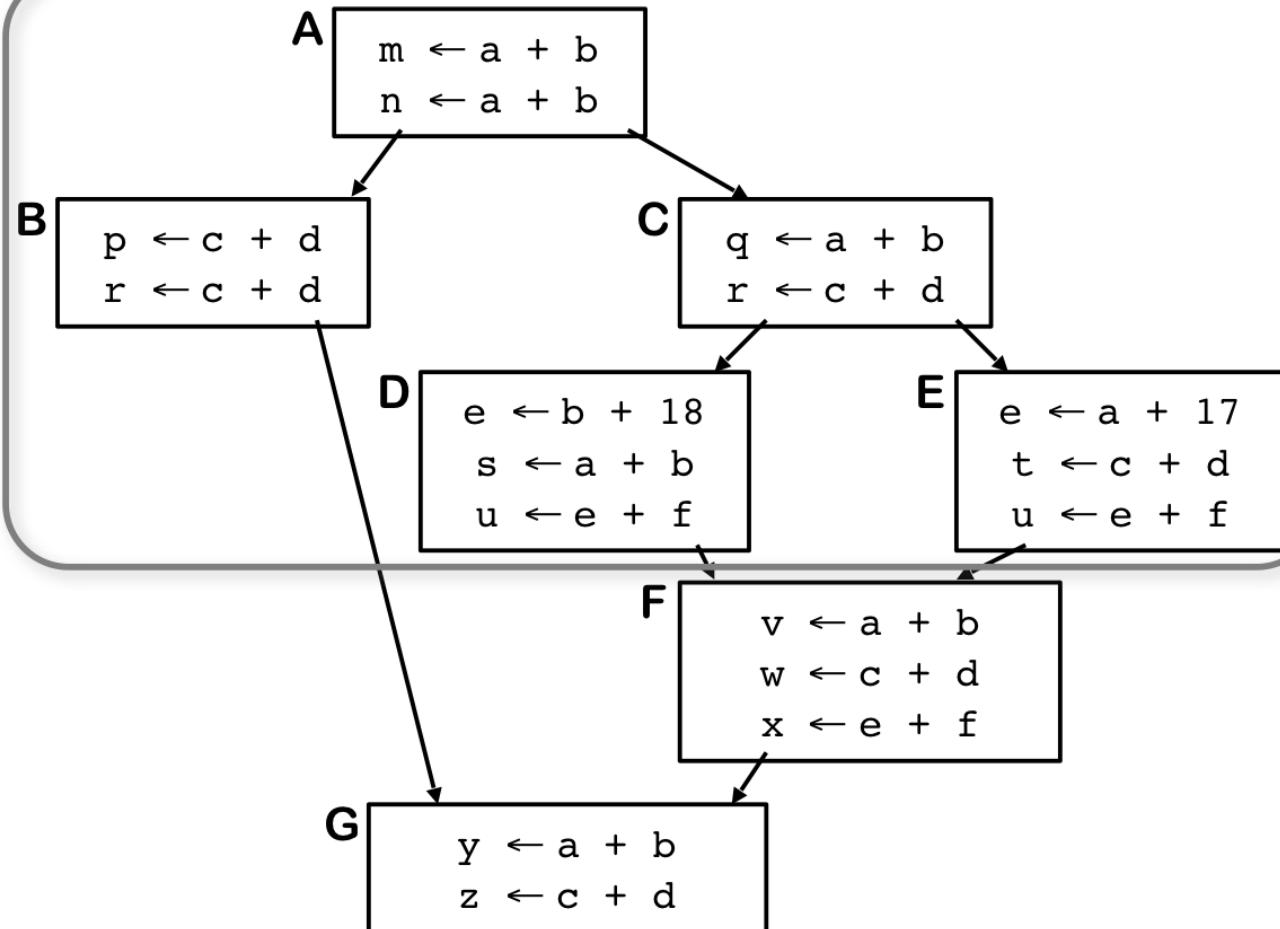
Assumes LVN has been parameterized
around block and table

Use LVN for the work

In the same EBB

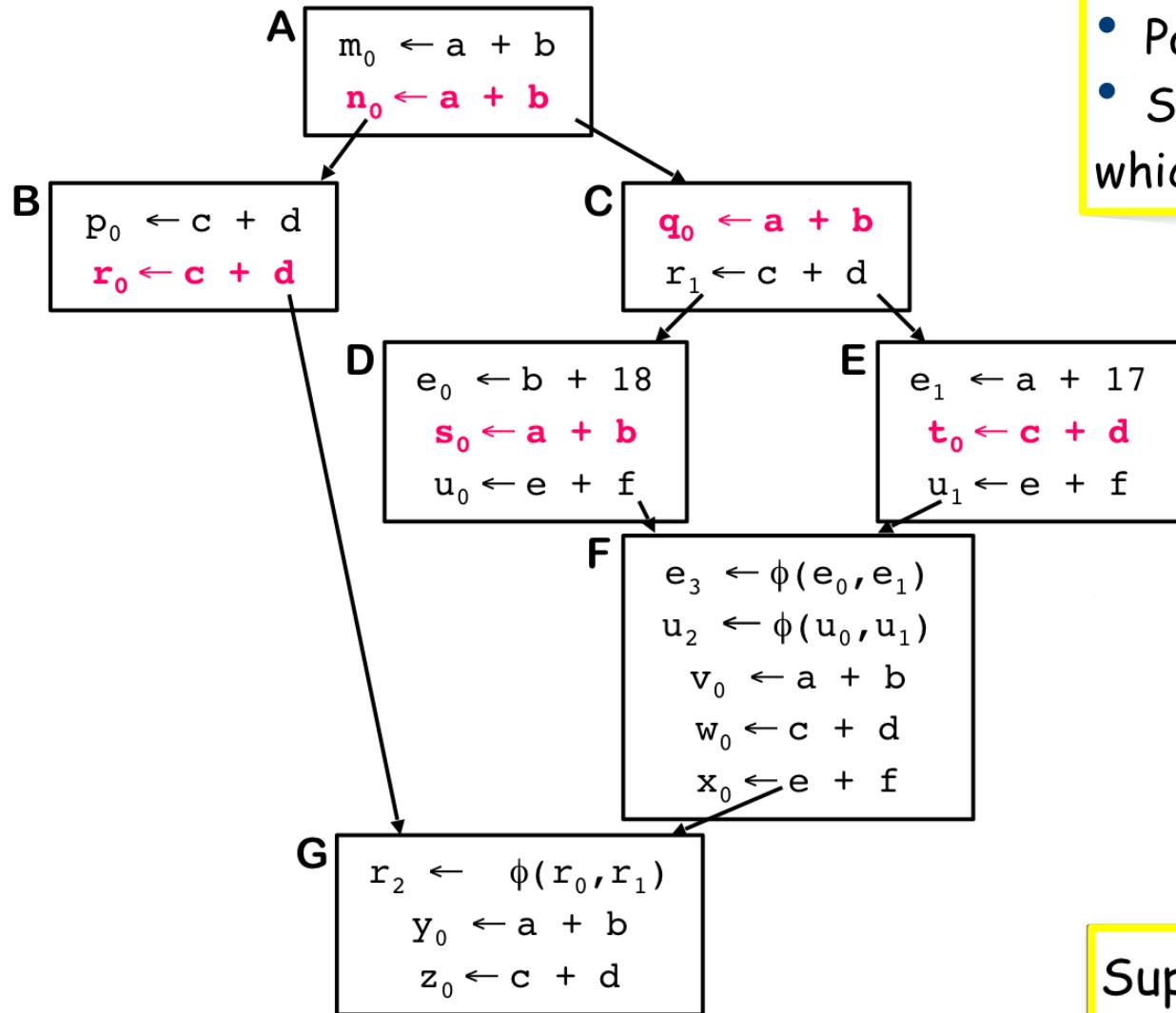
Starts a new EBB

Superlocal Value Numbering



1. Create scope for B_0
2. Apply LVN to B_0
3. Create scope for B_1
4. Apply LVN to B_1
5. Add B_6 to WorkList
6. Delete B_1 's scope
7. Create scope for B_2
8. Apply LVN to B_2
9. Create scope for B_3
10. Apply LVN to B_3
11. Add B_5 to WorkList
12. Delete B_3 's scope
13. Create scope for B_4
14. Apply LVN to B_4
15. Delete B_4 's scope
16. Delete B_2 's scope
17. Delete B_0 's scope
18. Create scope for B_5
19. Apply LVN to B_5
20. Delete B_5 's scope
21. Create scope for B_6
22. Apply LVN to B_6
23. Delete B_6 's scope

Superlocal Value Numbering



With all we saw SVN

- Find more **redundancy**
- Pay minimal extra cost
- Still does nothing for F & G which have some opportunities....

Superlocal techniques

- Some local methods extend cleanly to superlocal scopes

Loop Unrolling

A Regional Technique

Applications spend a lot of time in loops

- We can reduce loop overhead by unrolling the loop

```
do i = 1 to 100 by 1  
  a(i) ← b(i) * c(i)  
end
```



```
a(1) ← b(1) * c(1)  
a(2) ← b(2) * c(2)  
a(3) ← b(3) * c(3)  
...  
a(100) ← b(100) * c(100)
```

- Eliminated additions, tests and branches: reduce the number of operations Can subject resulting code to strong local optimization!
- Only works with fixed loop bounds & few iterations
- The principle, however, is sound
- Unrolling is always safe, as long as we get the bounds right

Loop Unrolling

Unrolling by smaller factors can achieve much of the benefit

Example: unroll by 4 (8, 16, 32? depends on # of registers)

```
do i = 1 to 100 by 1  
    a(i) ← b(i) * c(i)  
end
```



Unroll by 4

```
do i = 1 to 100 by 4  
    a(i) ← b(i) * c(i)  
    a(i+1) ← b(i+1) * c(i+1)  
    a(i+2) ← b(i+2) * c(i+2)  
    a(i+3) ← b(i+3) * c(i+3)  
end
```

Achieves much of the savings with lower code growth

- Reduces tests & branches by 25%
- LVN will eliminate duplicate adds and redundant expressions
- Less overhead per useful operation

But, it relied on knowledge of the loop bounds...

Loop Unrolling

Unrolling with unknown bounds

Need to generate guard loops

```
do i = 1 to n by 1  
    a(i) ← b(i) * c(i)  
end
```



```
i ← 1  
do while (i+3 < n )  
    a(i) ← b(i) * c(i)  
    a(i+1) ← b(i+1) * c(i+1)  
    a(i+2) ← b(i+2) * c(i+2)  
    a(i+3) ← b(i+3) * c(i+3)  
    i ← i + 4  
end  
  
do while (i < n)  
    a(i) ← b(i) * c(i)  
    i ← i + 1  
end
```

Achieves most of the savings

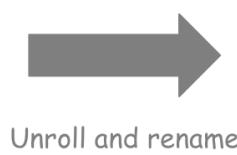
- Reduces tests & branches by 25%
- LVN still works on loop body
- Guard loop takes some space

Can generalize to arbitrary upper & lower bounds, unroll factors

One other unrolling trick

Eliminate copies at the end of a loop

```
t1 ← b(0)  
do i = 1 to 100 by 1  
    t2 ← b(i)  
    a(i) ← a(i) + t1 + t2  
    t1 ← t2  
end
```



$i=1, \dots, 100 : a(i)=a(i)+b(i)+b(i-1)$

```
t1 ← b(0)  
do i = 1 to 100 by 2  
    t2 ← b(i)  
    a(i) ← a(i) + t1 + t2  
    t1 ← b(i+1)  
    a(i+1) ← a(i+1) + t2 + t1  
end
```



- Eliminates the copies, which were a naming artifact
- Achieves some of the benefits of unrolling
 - Lower overhead, longer blocks for local optimization
- Situation occurs in more cases than you might suspect

Sources of Degradation

- It increases the size of the code
- The unrolled loop may have more demand for registers
- If the demand for registers forces additional register spills (store and reloads) then the resulting memory traffic may overwhelm the potential benefits of unrolling

12

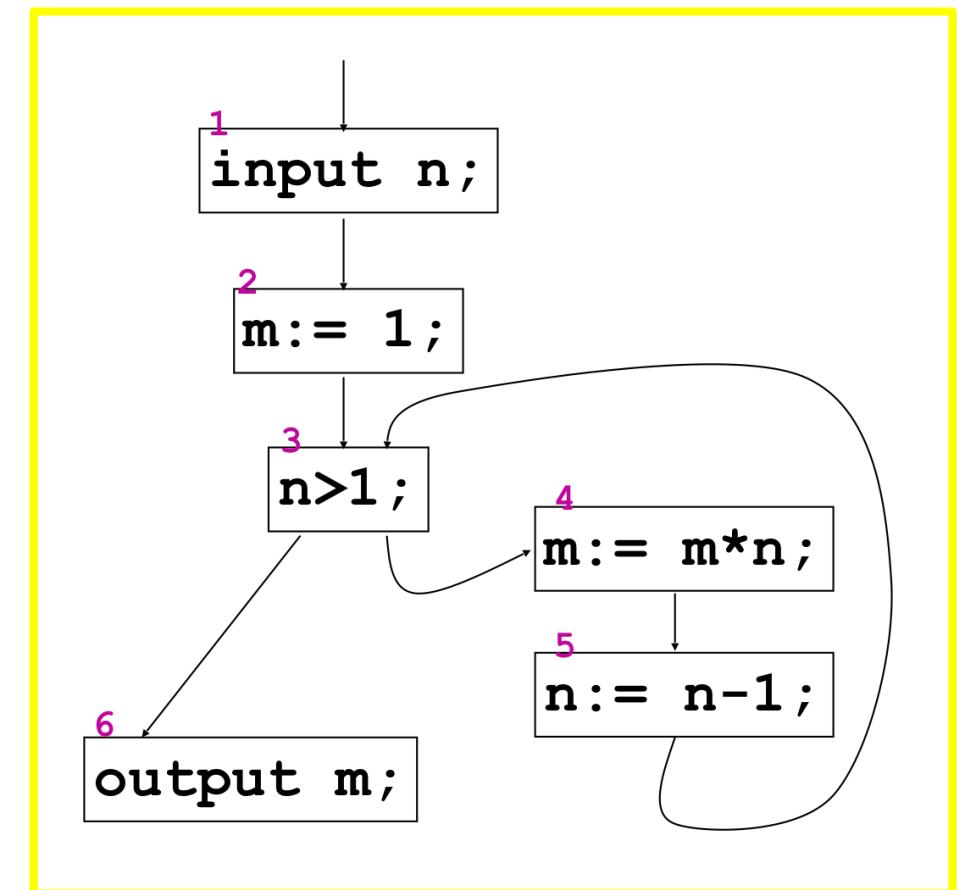
Dataflow Analyses

Control flow graph

- Program commands are encoded by nodes in a control flow graph
- If a command S may be directly followed by a command T then the control flow graph must include a direct arc from the node encoding S to the node encoding T

Example

```
[ input n; ]1
[ m:= 1; ]2
[ while n>1 do ]3
[ m:= m * n; ]4
    [ n:= n - 1; ]5
[ output m; ]6
```



Data-Flow analyses

We will see data-flow analyses:

- Liveness analysis
- Reaching definitions analysis
- Available Expressions analysis

Liveness or Live Variables Analysis

- We need to translate the source program in the intermediate representation IR that can use a large (potentially unbounded) number of registers.
- but the program will be executed by a processor with a (finite and) small number of registers
- Two variables a and b can be stored in the same register when it turns out that a and b are never simultaneously “used”

IR: Three Address Code

Three-address instruction has at most three operands and is typically a combination of an assignment and a binary operator.

For example: $t1 := t2 + t3$.

The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

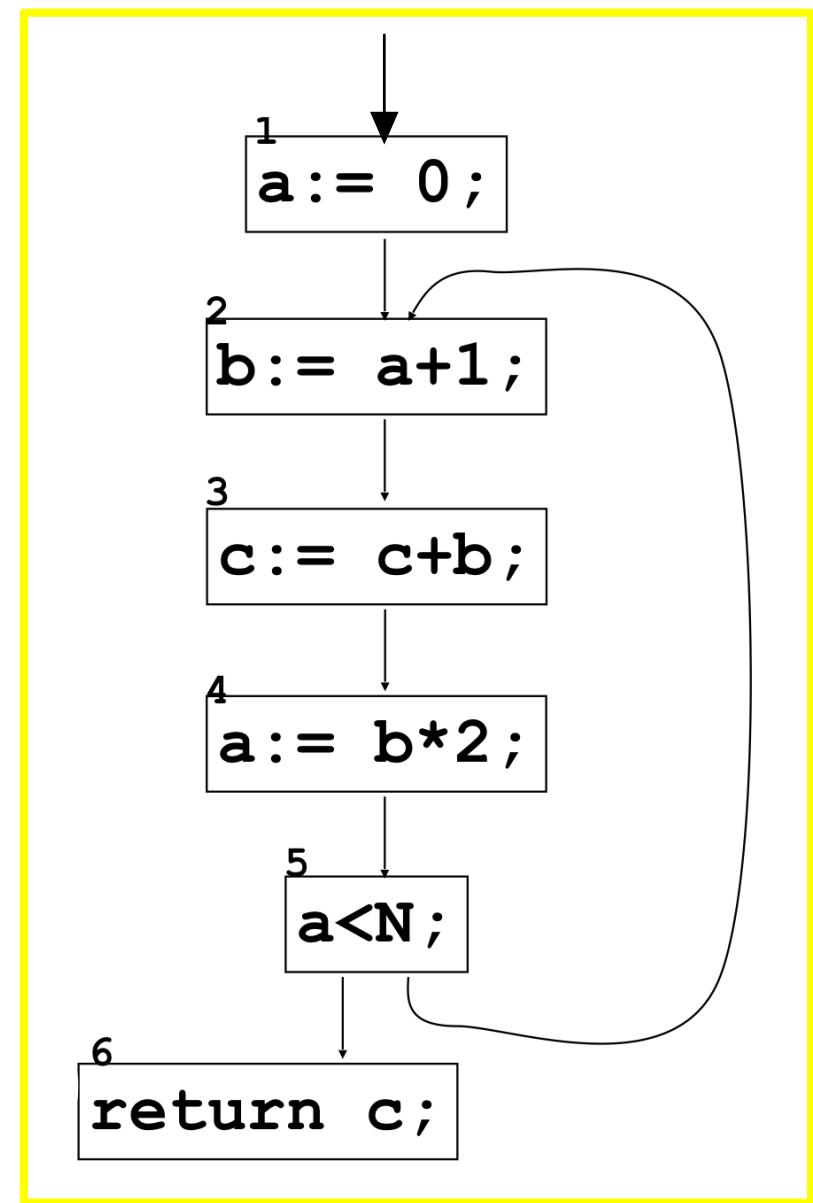
Live Variables Analysis

- A compiler needs to analyze programs in IR in order to find out which variables are simultaneously used
- A variable X is **live** at the exit of a command C if X stores a value which will be **actually used** in the future, that is, X will be used as R-value with no previous use as L-value
- A variable X which is not live at the exit of C is also called **dead** (this information can be used for dead code elimination)
- This is an undecidable property

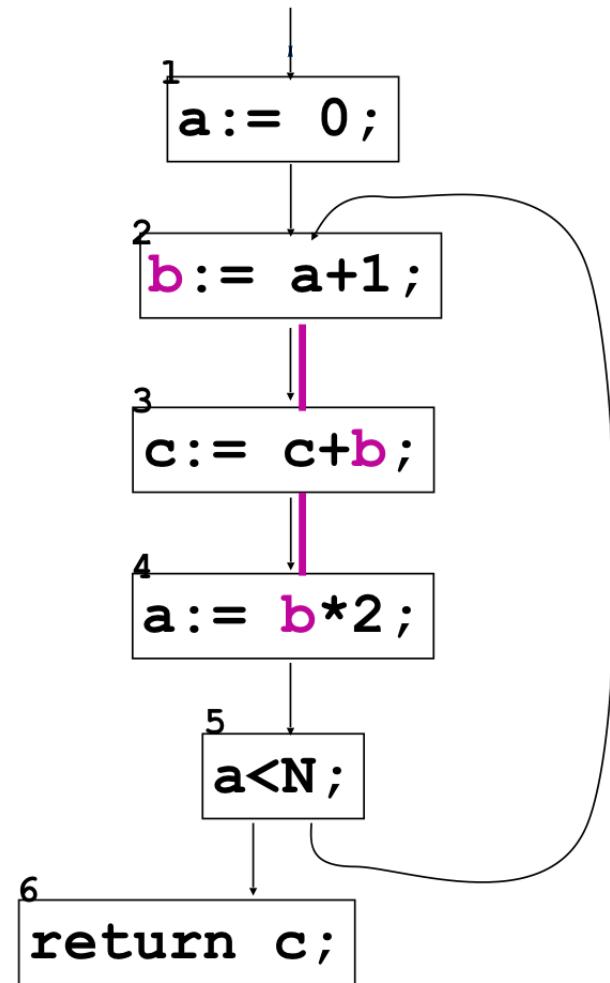
Example

```
a = 0;  
do {  
    b = a+1;  
    c += b;  
    a = b*2;  
}  
while (a<N);  
return c;
```

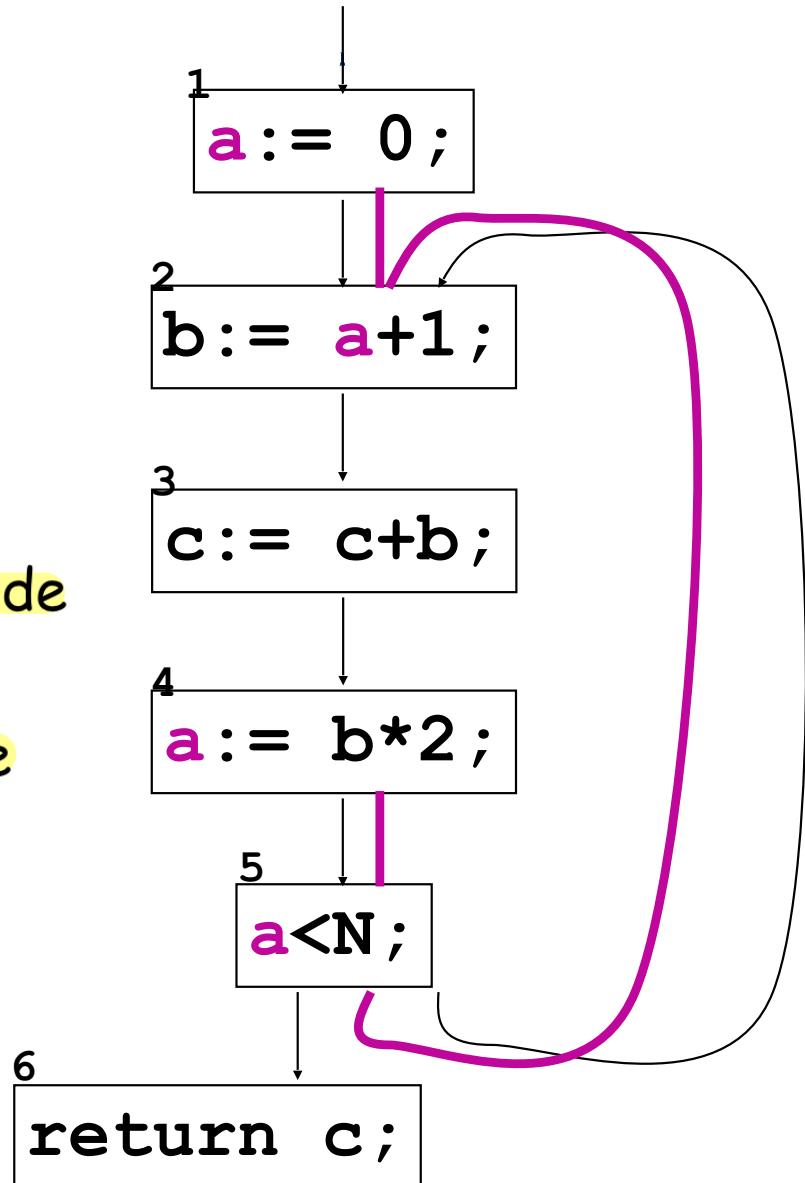
We want to know if **a** and **b** are simultaneously used.



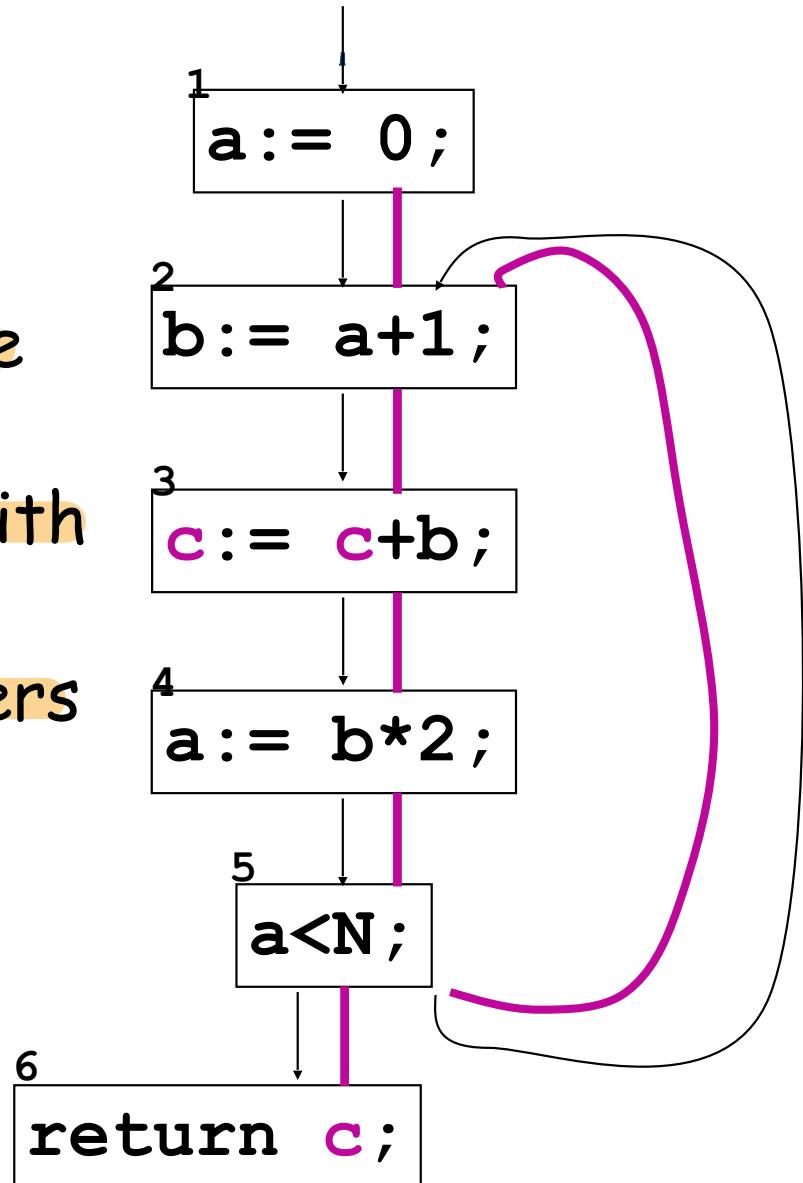
- A variable X is live when it stores a value which will be later used with no prior assignment to X
- The "last" use of the variable b as r-value is in command 4
- The variable b is used in command 4: it is therefore live along the arc $3 \rightarrow 4$
- Command 3 does not assign b, hence b is live along $2 \rightarrow 3$
- Command 2 assigns b. This means that the value of b along $1 \rightarrow 2$ will not be used later
- Thus, the "live range" of b turns out to be: { $2 \rightarrow 3, 3 \rightarrow 4$ }

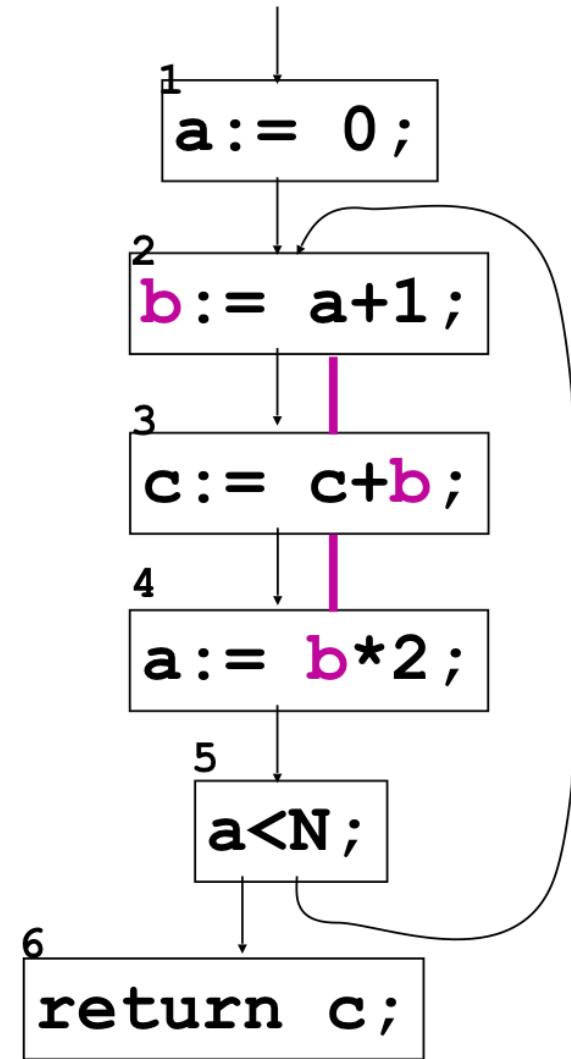
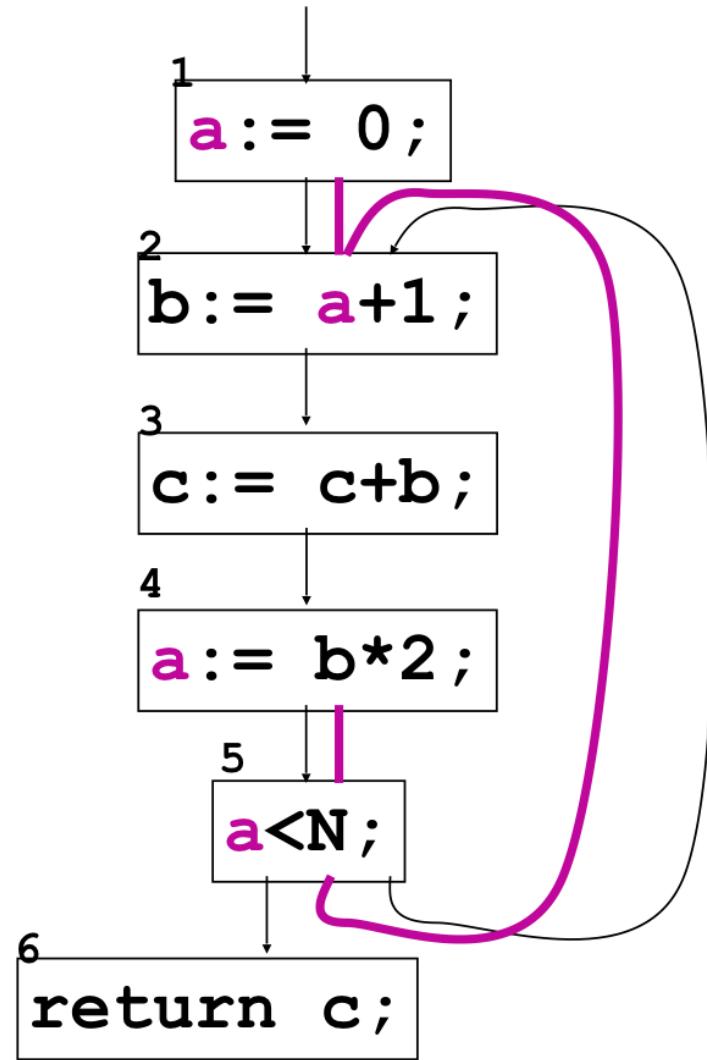


- **a** is live along $4 \rightarrow 5$ and $5 \rightarrow 2$
- **a** is live along $1 \rightarrow 2$
- **a** is not live along $2 \rightarrow 3$ and $3 \rightarrow 4$
- Even if the variable **a** stores a value in node 3, this value will not be later used, since node 4 assigns a new value to the variable **a**.



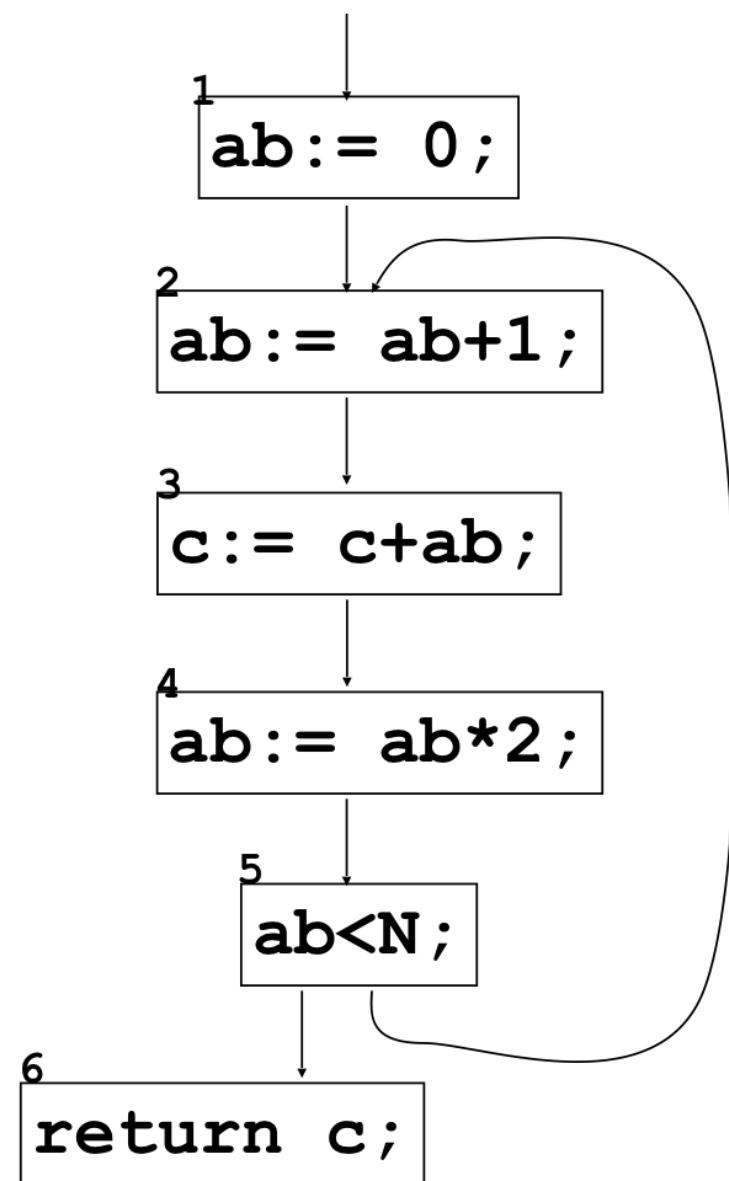
- **c** is live along all the arcs
 - By the way: liveness analysis can be exploited to deduce that if **c** is a local variable then **c** will be used with no prior initialization (this information can be used by compilers to raise a warning message)





→ Two registers are enough: variables **a** and **b** will be never simultaneously live along the same arc

Variables **a** and **b** will be never simultaneously live along the same arc. Hence, instead of using two distinct variables **a** and **b** we can correctly employ a single variable **ab**



We need a way to compute live variables

- A CFG has outgoing edges (**out-edges**) that lead to successor nodes, and incoming edges (**in-edges**) that originate from predecessor nodes.
- $\text{pre}[n]$ and $\text{post}[n]$ denote, respectively, the predecessor and successor nodes of some node n .

Notation

- An assignment to some variable (a use of the variable as L-value) is called a **definition** of the variable
- A use of some variable as R-value in a command is called a **use** of this variable
- **def[n]** denotes the set of variables that are defined in the node n
- **use[n]** denotes the set of variables that are used in the node n

Formalization of the property:

- A variable x is **live** along an arc $e \rightarrow f$ if there exists a real execution path P from the node e to some node n such that:
 - $e \rightarrow f$ is the first arc of such path P
 - $x \in \text{use}[n]$
 - for any node $n' \neq e$ and $n' \neq n$ in the path P ,
 $x \notin \text{def}[n']$
- A variable x is **live-out** in some node n if x is live along **some** (i.e., at least one) out-edge of n
- A variable x is **live-in** in some node n if x is live along **any** in-edge of n

Example

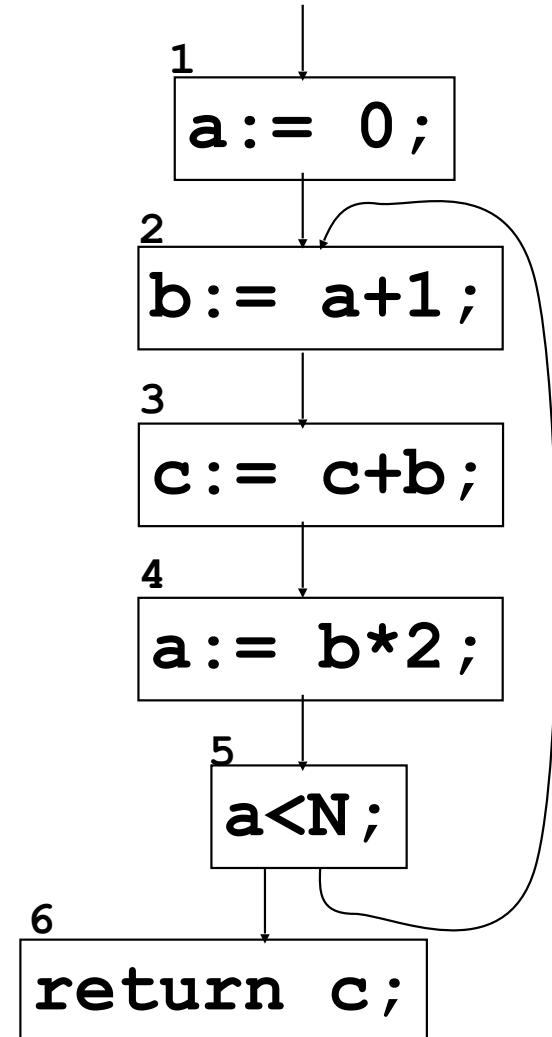
a is live along $1 \rightarrow 2, 4 \rightarrow 5$ and $5 \rightarrow 2$

b is live along $2 \rightarrow 3, 3 \rightarrow 4$

c is live along any arc

a is live-in in node 2, while it is not live-out in node 2

a is live-out in node 5



Computing an approximation of Liveness property

Let us define the following notation:

$\text{in}[n]$ is the set of variables that the static analysis determines to be live-in at node n

$\text{out}[n]$ is the set of variables that the static analysis determines to be live-out at node n

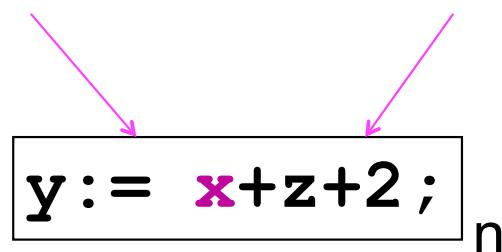
Computing an approximation of Liveness property

Liveness information: the sets $\text{in}[n]$ and $\text{out}[n]$ is computed as an over-approximation in the following way

n node of the CFG

2. If a variable $x \in \text{use}[n]$ then x is live-in in node n .

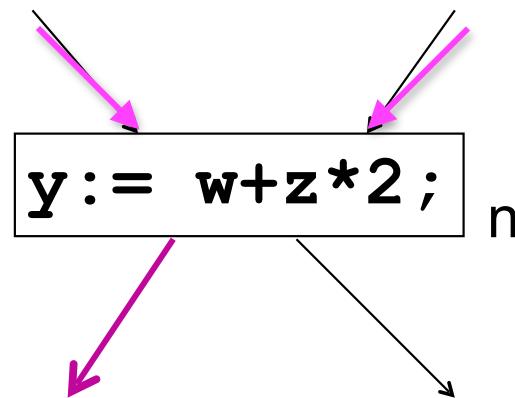
In other terms, if a node n uses a variable x as R-value then this variable x is live along each arc that enters into n .



$$\text{in}[n] \supseteq \text{use}[n]$$

Computing Liveness

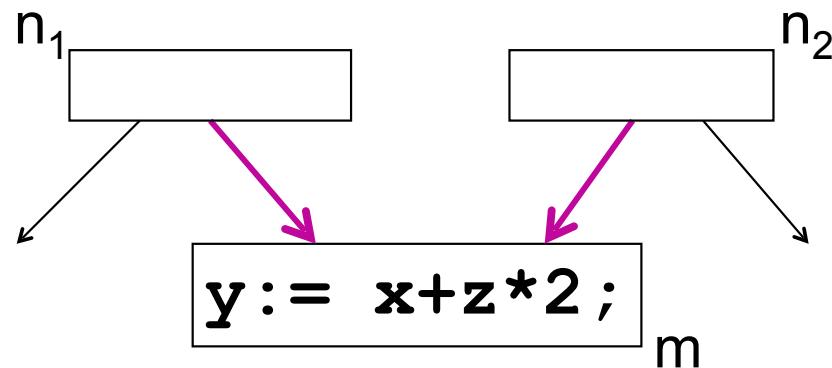
2. If a variable x is live-out in a node n and $x \notin \text{def}[n]$ then the variable x is also live-in in this node n .
If a variable x is live for some arc that leaves a node n and x is not assigned in n then x is live for all the arcs that enter in n



$$\text{in}[n] \supseteq \text{out}[n] - \text{def}[n]$$

Computing Liveness

3. If a variable x is live-in in a node m then x is live-out for all the nodes n such that $m \in \text{post}[n]$.
This is clearly correct by definition.



$$\begin{aligned} \text{out}[n_1] &\supseteq \cup\{\text{in}[m] \mid m \in \text{post}[n_1]\} \\ \text{out}[n_2] &\supseteq \cup\{\text{in}[m] \mid m \in \text{post}[n_2]\} \end{aligned}$$

Dataflow Equations

The previous three rules of liveness analysis can be thus formalized by two equations for each node n :

$$1. \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \quad (\text{rules 1 and 2})$$

$$2. \text{out}[n] = \bigcup \{\text{in}[m] \mid m \in \text{post}[n]\} \quad (\text{rule 3})$$

Correctness of the analysis of Liveness

This definition of liveness analysis $\text{in}[n]$ and $\text{out}[n]$ is **correct**:

If x is concretely live-in (live-out) in some node n then the static analysis will detect that $x \in \text{in}[n]$ ($x \in \text{out}[n]$):

$$\begin{aligned}\text{in}[n] &\supseteq \text{live-in}[n] \\ \text{out}[n] &\supseteq \text{live-out}[n]\end{aligned}$$

In other terms, no actually live variable is neglected by liveness analysis.

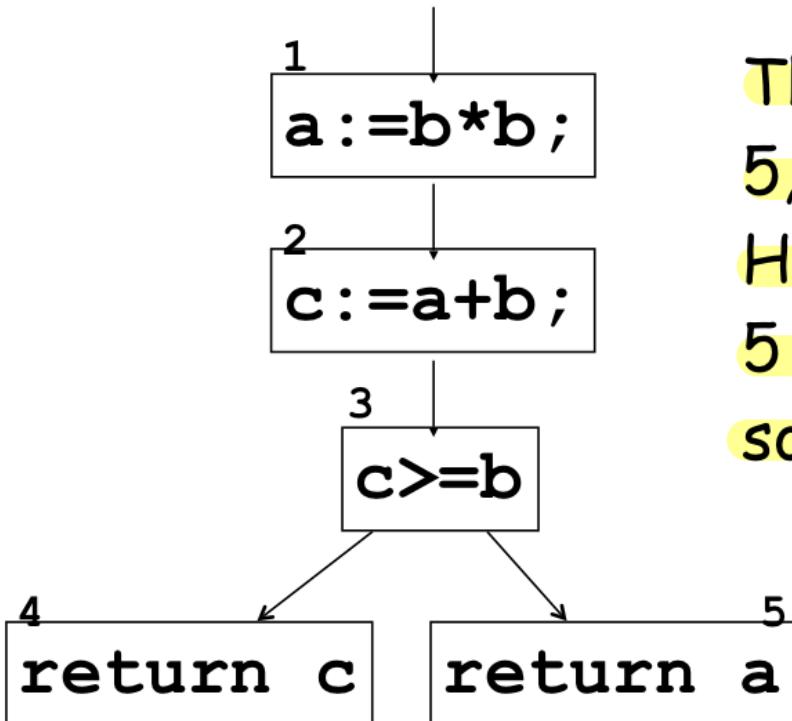
Computing Liveness

Liveness analysis is approximate:

it assumes that each path of the CFG is a feasible path
while this hypothesis is obviously not true



Computing Liveness



The analysis determines that `a` is live-in in 5, and therefore `a` is live-out in 3.
However, no real execution path from 3 to 5 exists (because $b+b^*b < b$ is always false)
so that `a` is not really live when exiting 3!

How can we compute a solution to 1 and 2?

$$1. \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

$$2. \text{out}[n] = \bigcup \{\text{in}[m] \mid m \in \text{post}[n]\}$$

Correctness tells us that $\text{in}[n] \supseteq \text{live-in}[n]$ and $\text{out}[n] \supseteq \text{live-out}[n]$

But we need a way to compute Live variable analysis



- We need to compute a fix point
- but how can we be sure that such fix-points exist?
It depends on the domain and on the function!





Questions

- Does a solution of the semantic equation always exist?
- If it exists, is it unique?
- How to compute it?

Fixpoint?

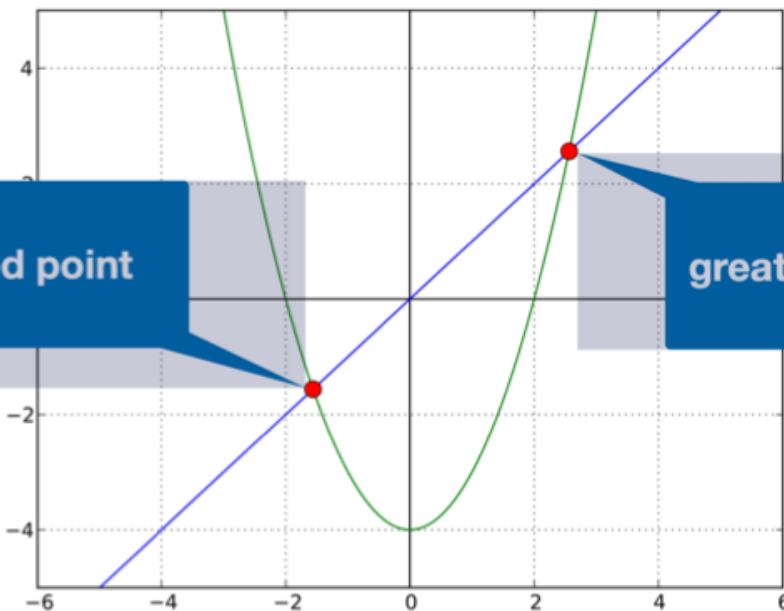
$\text{fix } F = X \text{ such that } F(X) = X$

$$(x) = x^2 - 4$$

lfp

least fixed point

gfp



The fix point theory

Definition (Partial Order). A binary relation \sqsubseteq is a partial order on a set D if it holds:

1. reflexivity: $a \sqsubseteq a$ for all $a \in D$
2. Antisymmetry: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$
3. Transitivity: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$

A set D with a partial order \sqsubseteq is called a partially ordered set (D, \sqsubseteq) , or simply poset.

Least Upper Bound

Definition (Least Upper Bound).

For a partial ordered set (D, \sqsubseteq) and $X \subseteq D$,

$d \in D$ is an **upper bound** of X iff $\forall x \in X. x \sqsubseteq d$

An upper bound d is the **least upper bound** of X iff for all upper bounds y of X

$$d \sqsubseteq y$$

The **least upper bound** of X is denoted by $\sqcup X$

Chain

Definition (Chain). Let (D, \sqsubseteq) be a partial ordered set. A subset $X \subseteq D$, is called a **chain** if X is totally ordered:

$$\forall x_1, x_2 \in X. x_1 \sqsubseteq x_2 \text{ or } x_2 \sqsubseteq x_1$$

CPO

Definition (CPO). A poset (D, \sqsubseteq) is a **CPO** (complete partial order) if every chain X of D has a LUB

$$\bigcup X \in D$$

We consider CPOs (D, \sqsubseteq) with a least element \perp

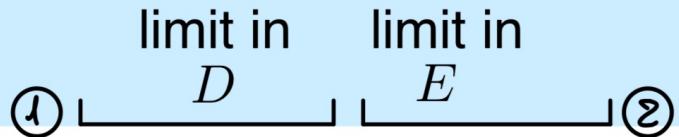
Monotone and Continuous Functions

Definition (Monotone Function). Given two partially ordered sets D and E a function $f : D \rightarrow E$ is **monotone** if it preserves orders between any two elements in D 'es orders

$$\forall d_1, d_2 \in D. d_1 \sqsubseteq d_2 \implies f(d_1) \sqsubseteq f(d_2)$$

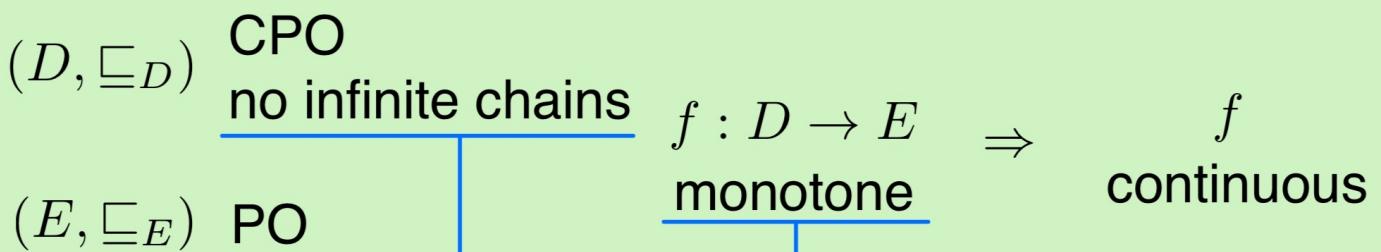
(D, \sqsubseteq_D) CPO (E, \sqsubseteq_E) CPO $f : D \rightarrow E$ monotone

f is continuous if $\forall \{d_i\}_{i \in \mathbb{N}}$ chain $f\left(\bigsqcup_{i \in \mathbb{N}} d_i\right) = \bigsqcup_{i \in \mathbb{N}} f(d_i)$



Continuous = limit preserving

- ① FIND THE LIMIT OF THE CHAIN $\{d_i\}_{i \in \mathbb{N}}$ WHICH IS $\bigsqcup_{i \in \mathbb{N}} d_i$ AND APPLY f TO IT
- ② f IS MONOTONE, SO IF $d_i \sqsubseteq d_{i+1}$ THEN $f(d_i) \sqsubseteq f(d_{i+1})$:
BUILD THE CHAIN $\{f(d_i)\}_{i \in \mathbb{N}}$ AND FIND ITS LIMIT $\bigsqcup_{i \in \mathbb{N}} f(d_i)$
- IF ① == ② THEN f IS CONTINUOUS



PROOF

TAKE A CHAIN $\{d_i\}_{i \in \mathbb{N}}$

f CAN ONLY BUILD
FINITE CHAINS

- THE CHAIN IS FINITE ((D, \sqsubseteq) IS A CPO): $\exists k \in \mathbb{N}. \bigsqcup_{i \in \mathbb{N}} d_i = d_k$
- f IS MONOTONE AND $\{d_i\}_{i \in \mathbb{N}}$ IS FINITE: $\{f(d_i)\}_{i \in \mathbb{N}}$ IS FINITE

f MONOTONE (AKA ORDER PRESERVING): d_k (LUB OF $\{d_i\}$) IS SENT TO THE LUB OF $\{f(d_i)\}$

d_k LUB OF $\{d_i\} \Rightarrow f(d_k)$ IS THE LUB OF $\{f(d_i)\}_{i \in \mathbb{N}}$ (f IS MONOTONE)

$$\text{LUB of } \{f(d_i)\}_{i \in \mathbb{N}} = \boxed{\bigsqcup_{i \in \mathbb{N}} f(d_i)} \stackrel{\substack{\text{DEF OF LUB} \\ \text{DEF OF } d_k}}{=} f(d_k) = \boxed{f(\bigsqcup_{i \in \mathbb{N}} d_i)}$$

DEF OF CONTINUITY ■

Repeated application

$$f : D \rightarrow D$$

$$f^0(d) \triangleq d$$

$$f^n(d) = \overbrace{f(\cdots(f(d))\cdots)}^{n \text{ times}}$$

$$f^{n+1}(d) \triangleq f(f^n(d))$$

$$f^n : D \rightarrow D$$

Lemma (D, \sqsubseteq) PO $_{\perp}$ $f : D \rightarrow D$ monotone



$\{f^n(\perp)\}_{n \in \mathbb{N}}$ is a chain

Towards Kleene's

$\{f^n(d)\}_{n \in \mathbb{N}}$
not necessarily
a chain!

when (D, \sqsubseteq) is a CPO $_{\perp}$ then $\underbrace{\{f^n(\perp)\}_{n \in \mathbb{N}}}$ is a chain it must have a limit



Kleene's fix point theorem states that if f is continuous, then the limit of the above chain is the least fixpoint of f

Pre-Fixpoints

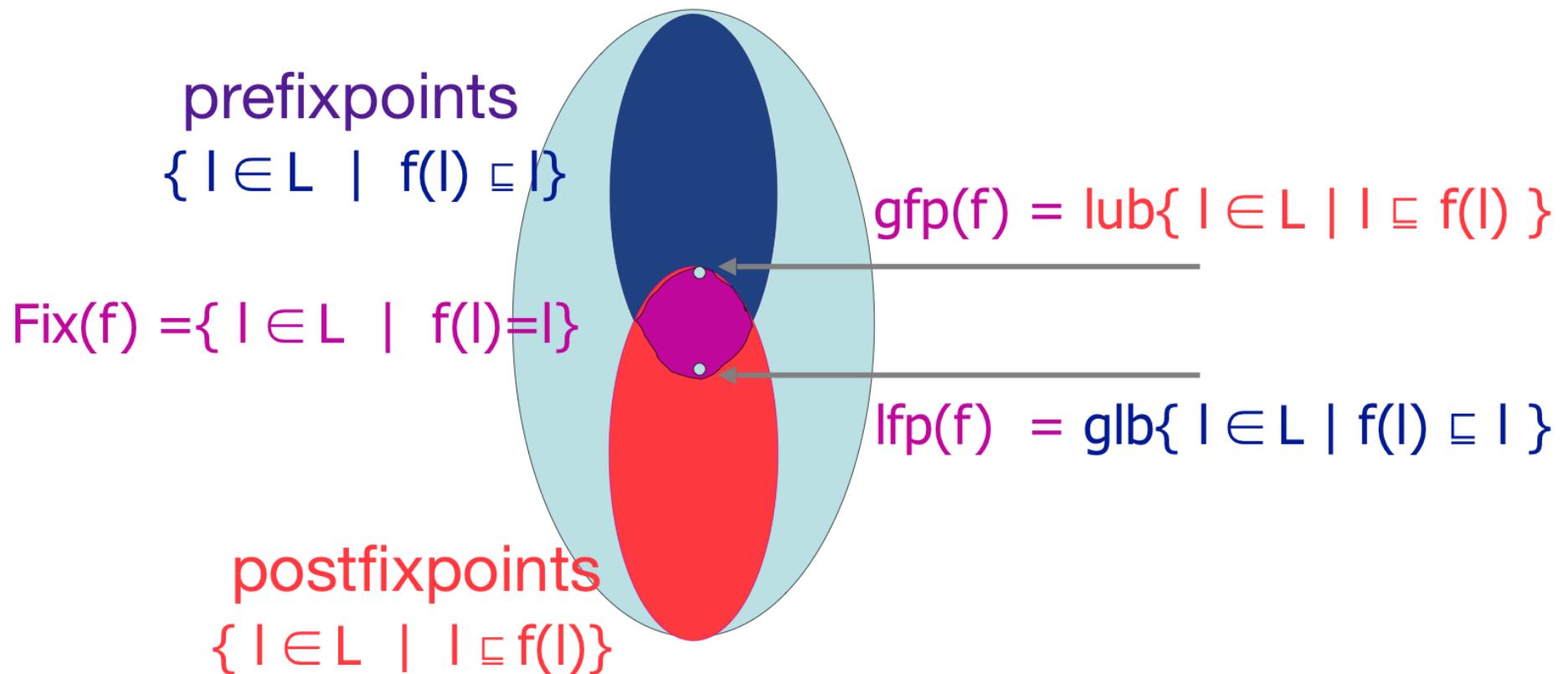
(D, \sqsubseteq) PO $f : D \rightarrow D$ monotone

fixpoint $p \in D$ $f(p) = p$

pre-fixpoint $p \in D$ $f(p) \sqsubseteq p$

Clearly any fixpoint is also a pre-fixpoint

Tarsky 's Theorem



Kleene's Theorem

(D, \sqsubseteq) CPO $_{\perp}$ $f : D \rightarrow D$ continuous

let $\text{fix}(f) \triangleq \bigsqcup_{n \in \mathbb{N}} f^n(\perp)$

1. $\text{fix}(f)$ is a fix point of f

$$f(\text{fix}(f)) = \text{fix}(f)$$

2. $\text{fix}(f)$ is the least pre-fixpoint of f

$$\forall d \in D. f(d) \sqsubseteq d \Rightarrow \text{fix}(f) \sqsubseteq d$$

if d is a pre-fixpoint then $\text{fix}(f)$ is smaller than d

1. $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$
2. $\text{out}[n] = \bigcup \{\text{in}[m] \mid m \in \text{post}[n]\}$

Which is our domain?

Our objects:

Given a node we need to compute the set **in** and the set **out** (sets of variables)

- Let **Vars** be the finite set of variables that occur in the program P to analyze. We consider all possible subsets: $\mathcal{P}(\text{Vars})$

Given a node we will need a set for in and a set for out: $\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars})$

But we have **N** nodes, one for each node of the **CFG** so our domain will be

$(\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^N$: N-tuples of pairs of subsets of **Vars**

The order : \subseteq^{2N}

$$\langle \text{in}_1^1, \text{out}_1^1, \dots, \text{in}_N^1, \text{out}_N^1 \rangle \subseteq^{2N} \langle \text{in}_1^1, \text{out}_1^1, \dots, \text{in}_N^1, \text{out}_N^1 \rangle$$

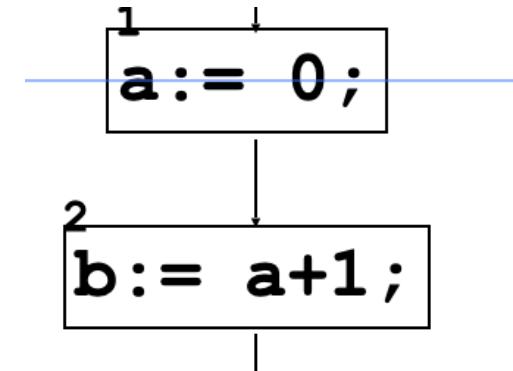
iff

$$\text{in}_i^1 \subseteq \text{in}_i^2 \text{ and } \text{out}_i^1 \subseteq \text{out}_i^2$$

Example

Vars ={a,b} N=2.

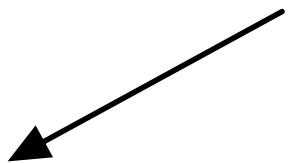
$\langle (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^2, \subseteq^4 \rangle$ is a finite domain.



Our domain

$\langle (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^{\mathbb{N}}, \subseteq^{2^N} \rangle$ CPO with bottom?

- It is a CPO because it is finite
- bottom?



Which is our function?

1. $in[n] = use[n] \cup (out[n] - def[n])$
2. $out[n] = \bigcup \{in[m] \mid m \in post[n]\}$

The map Live:

$$((\text{Vars}) \times \mathcal{P}(\text{Vars}))^N \rightarrow (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^N$$

$\text{Live}(<in_1, out_1, \dots, in_N, out_N>) =$

$$<use[1] \cup (out_1 - def[1]), \bigcup_{m \in post[1]} in_m, \dots, use[N] \cup (out_N - def[N]), \bigcup_{m \in post[N]} in_m>$$

is continuous?

Yes! because it is monotone on a finite domain



In conclusion

The map Live :

$(\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^{\mathbb{N}} \rightarrow (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^{\mathbb{N}}$ defined by

is a monotonic (and therefore continuous) function on the finite CPO

$\langle (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^{\mathbb{N}}, \subseteq^{2N} \rangle$

and therefore Live has a least fixpoint

Why a least fixpoint

Live is a possible analysis,

$$\text{in}[n] \supseteq \text{live-in}[n] \text{ and } \text{out}[n] \supseteq \text{live-out}[n]$$

i.e., if a variable x will be really live in a node n during some program execution then x belongs to $\text{in}[n]$ of all the fixpoints of the function Live

All fixpoints of the equation system is an over-approximation of really live variables.

We want the least fixpoint (more precise over approximations)

Conservative Approximation

- How to interpret the output of this static analysis?
- Correctness tells us that:

$$\text{in}[n] \supseteq \text{live-in}[n] \text{ and } \text{out}[n] \supseteq \text{live-out}[n]$$

If the variable x will be really live in some node n during some program execution then x belongs to $\text{in}[n]$ of all the fixpoints of the function Live (least fixpoint)

- The converse does not hold: the analysis can tell us that x is in the computed set $\text{out}[n]$, but this does not imply that x will be necessarily live in n during some program execution
- In liveness analysis “conservative approximation” means that the analysis may erroneously derive that a variable is live, while the analysis is not allowed to erroneously derive that a variable is “dead” (i.e., not live).

★ if $x \in \text{in}[n]$ then x could be live at program point n .

★ if $x \notin \text{in}[n]$ then x is definitely dead at program point n .

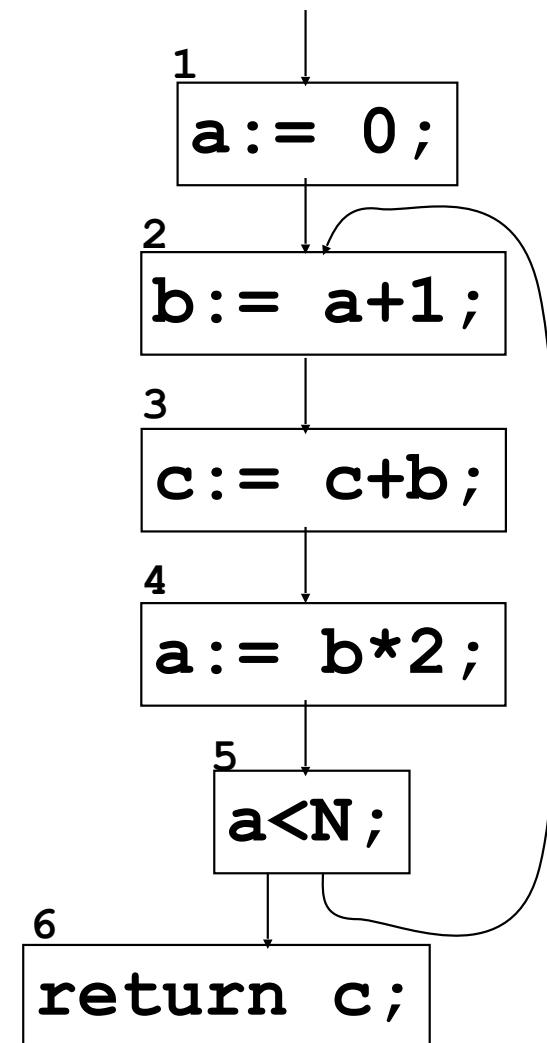
```

for all n
    in[n] := {} out[n] := {};
repeat
    for all n (1 to 6)
        in'[n] := in[n]; out'[n] := out[n];
        in[n] := use[n] ∪ (out[n] - def[n]);
        out[n] := ∪ { in[m] | m ∈ post[n] };
until (for all n: in'[n]=in[n] && out'[n]=out[n])

```

TODO

		use	def	in	out	in	out	in	out
		Live ¹		Live ²		Live ³			
1	a					a		a	
2	a b	a		a	b c	a c	b c	a c	b c
3	b c c	b c		b c	b	b c	b	b c	b
4	b a	b		b	a	b	a	b a	
5	a	a a		a	a c	a c	a c	a c	a c
6	c	c		c		c		c	

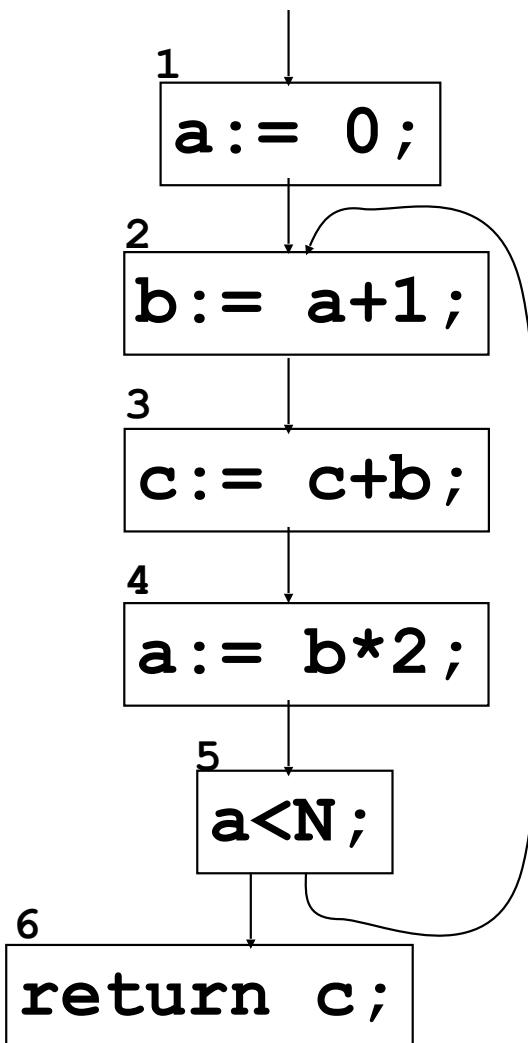


```

for all n
    in[n] :=?; out[n] :=?;
repeat
    for all n (1 to 6)
        in'[n] := in[n]; out'[n] := out[n];
        in[n] := use[n] ∪ (out[n] - def[n]);
        out[n] := ∪ { in[m] | m ∈ post[n] };
until (for all n: in'[n]=in[n] && out'[n]=out[n])

```

TODO



	use	def	in	out	in	out	in	out
1	a		a		a	c	c	a c
2	a b		a c	b c	a c	b c	a c	b c
3	b c	c	b c	b	b c	b	b c	b
4	b a		b a		b	a c	b c	a c
5	a		a c	a c	a c	a c	a c	a c
6	c		c		c		c	

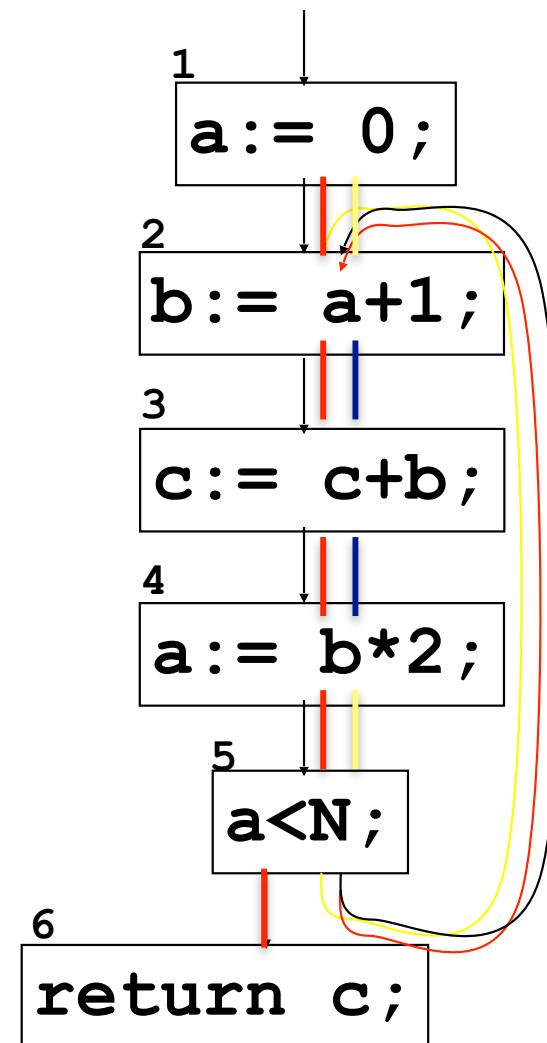
TODO

		Live ⁵		Live ⁶		Live ⁷	
	use def	in	out	in	out	in	out
1	a	c	ac	c	ac	c	ac
2	a b	ac	b c	ac	b c	ac	b c
3	b c c	b c	b	b c	b c c	b c	b c
4	b a	b c	ac	b c	ac	b c	ac
5	a	a c	ac	a c	ac	a c	ac
6	c	c		c		c	

The algorithm thus gives the following output:

out[1]={a,c}, out[2]={b,c}, out[3]={b,c}, out[4]={a,c},
 out[5]={a,c}

In this case, the output of the analysis is precise



Improvement

In this iterative computation, observe that we have to wait for the next iteration in order to exploit the new information computed for in and out on the nodes.

By a suitable reordering of the nodes and by first computing $\text{out}[n]$ and then $\text{in}[n]$, we are able to converge to the fixpoint in just 3 iteration steps.

```
for all n
    in[n]:=?; out[n]:=?;
repeat
    for all n (6 to 1)
        in'[n]:=in[n]; out'[n]:=out[n];
        out[n]:= U { in[m] | m ∈ post[n] };
        in[n]:= use[n] U (out[n] - def[n]);
until (for all n: in'[n]=in[n] && out'[n]=out[n])
```

```

for all n
    in[n] := ?; out[n] := ?;
repeat
    for all n (6 to 1)
        in'[n] := in[n]; out'[n] := out[n];
        out[n] := U { in[m] | m ∈ post[n] };
        in[n] := use[n] U (out[n] - def[n]);
until (for all n: in'[n]=in[n] && out'[n]=out[n])

```

TODO

	Live ¹		Live ²		Live ³	
	use	def	out	in	out	in
6	c		c		c	
5	a		c	a c	a c	a c
4	b	a	a c	b c	a c	b c
3	b c	c	b c	b c	b c	b c
2	a	b	b c	a c	b c	a c
1		a	ac	c	ac	c

Backward Analysis

As shown by the previous example, Live Variable Analysis is a “backward” analysis. This means that information propagates “backward” from terminal nodes to initial nodes:

1. $\text{in}[n]$ can be computed from $\text{out}[n]$;
2. $\text{out}[n]$ can be computed from $\text{in}[m]$ for all the nodes m that are successors of n .

Application: Dead Code Elimination

```
i := 0;
t3 := 0;
while i <= n do    dead variable
    j := 0;
    t2 := t3;
    while j <= m do
        t1 := t3 + j;
        temp := Base(A) + t1;
        Cont(temp) := Cont(Base(B) + t1)
                      + Cont(Base(C) + t1);
        j := j+1
    od;
    i := i+1;
    t3 := t3 + (m+1)
od
```

```
i := 0;
t3 := 0;
while i <= n do
    j := 0;
    while j <= m do
        t1 := t3 + j;
        temp := Base(A) + t1;
        Cont(temp) := Cont(Base(B) + t1)
                      + Cont(Base(C) + t1);
        j := j+1
    od;
    i := i+1;
    t3 := t3 + (m+1)
od
```

Reaching Definitions (Reaching Assignment) Analysis

One of the more useful data-flow analysis

```
d1 : y := 3  
d2 : x := y
```

d1 is a reaching definition for d2

```
d1 : y := 3  
d2 : y := 4  
d3 : x := y
```

d1 is no longer a reaching definition for d3, because d2 kills its reach:
the value defined in d1 is no longer available and cannot reach d3

A definition d at point i reaches a point p if there is a path from the point i to p such that d is not killed (redefined) along that path

Reaching definitions

This information is very useful

- The compiler can know whether x is a constant at point p
- The debugger can tell whether it is possible that x is an undefined variable at point p

Reaching definitions

Given a program point n , which definitions are actual - not successively overwritten by a different assignment - when the execution reaches n ?

And when the execution leaves n ?

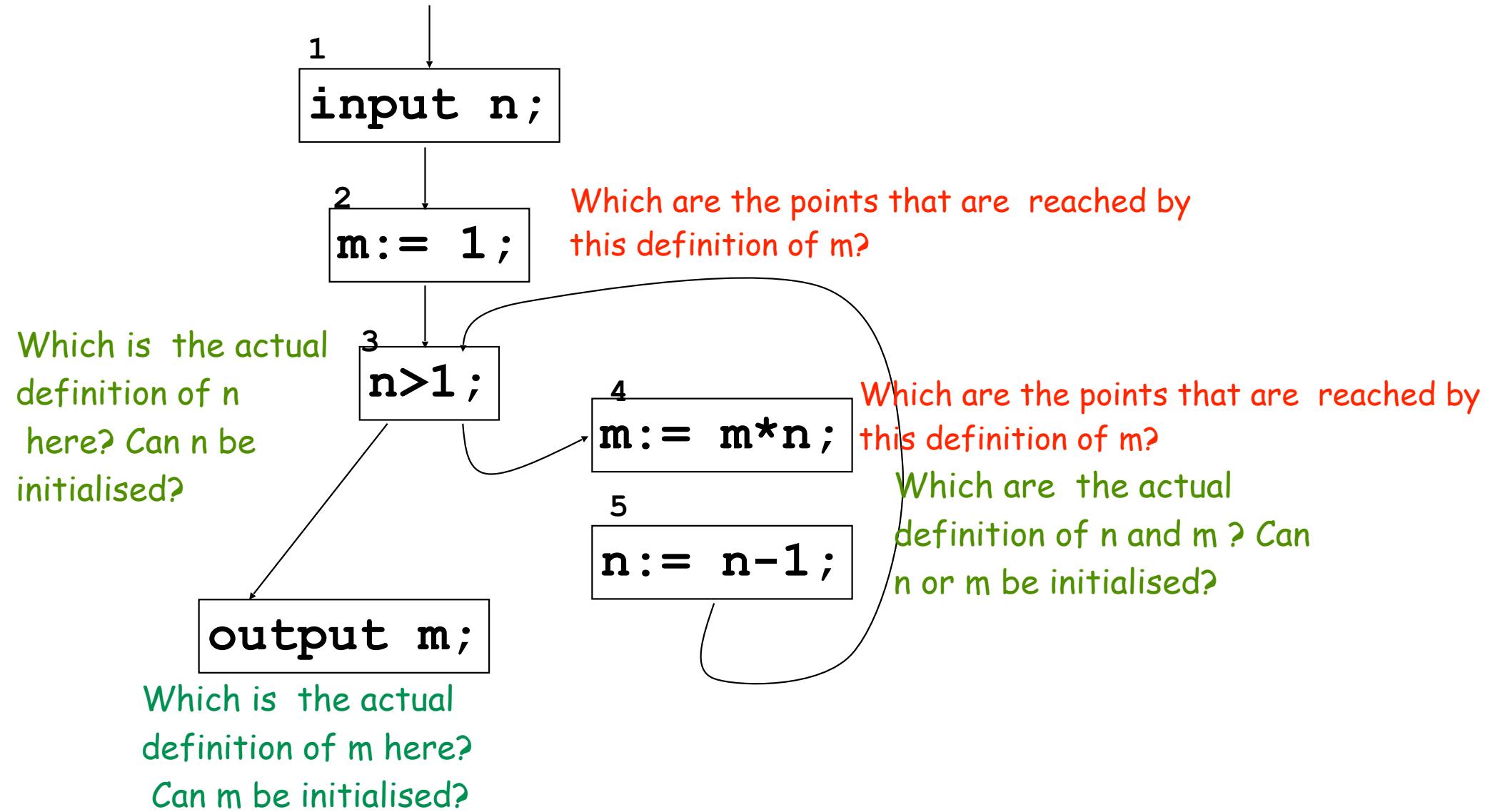
A program point may clearly "generate" new definitions

A program point n may "kill" a definition:

if n is an assignment $x := \text{exp}$ then n kills all the assignments to the variable x which are actual in input to n

We are thus interested in computing input and output reaching definitions for any program point

The intuition: the factorial of n



Formalization of the reaching definition property

- The property can be represented by sets of pairs:
 $\{(x,p) \mid x \in \text{Vars}, p \text{ is a program point}\} \in \mathcal{P}(\text{Vars} \times \text{Points})$
where (x,p) means that the variable x is assigned at
program point p
- For each program point, this dataflow analysis computes a
set of such pairs
- The meaning of a pair (x,p) in the set for a program point q
is that the assignment of x at point p is actual at point q
- $?$ is a special symbol that we add to **Points** and we use to
represent the fact that a variable x is not initialized.
- The set $I = \{(x,?) \mid x \in \text{Vars}\}$ therefore denotes that all the
program variables are not initialized.

The domain for Reaching Definitions Analysis

Vars is the (finite) set of variables occurring in the program P.

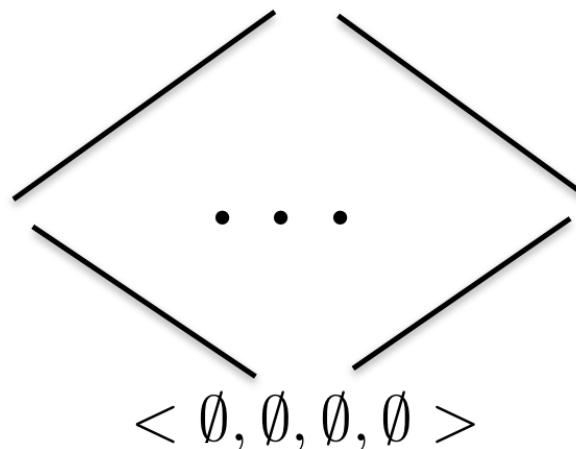
Let N be the number of nodes of the CFG of P.

Let Points={?,1,...N}.

$$\langle (\mathcal{P}(\text{Vars} \times \text{Points}) \times \mathcal{P}(\text{Vars} \times \text{Points}))^N, \subseteq^{2N} \rangle$$

Example Vars={a,b} e N=2

$$\langle S = \{(a, ?), (a, 1), (b, ?), (b, 1)\}, S, S, S \rangle$$

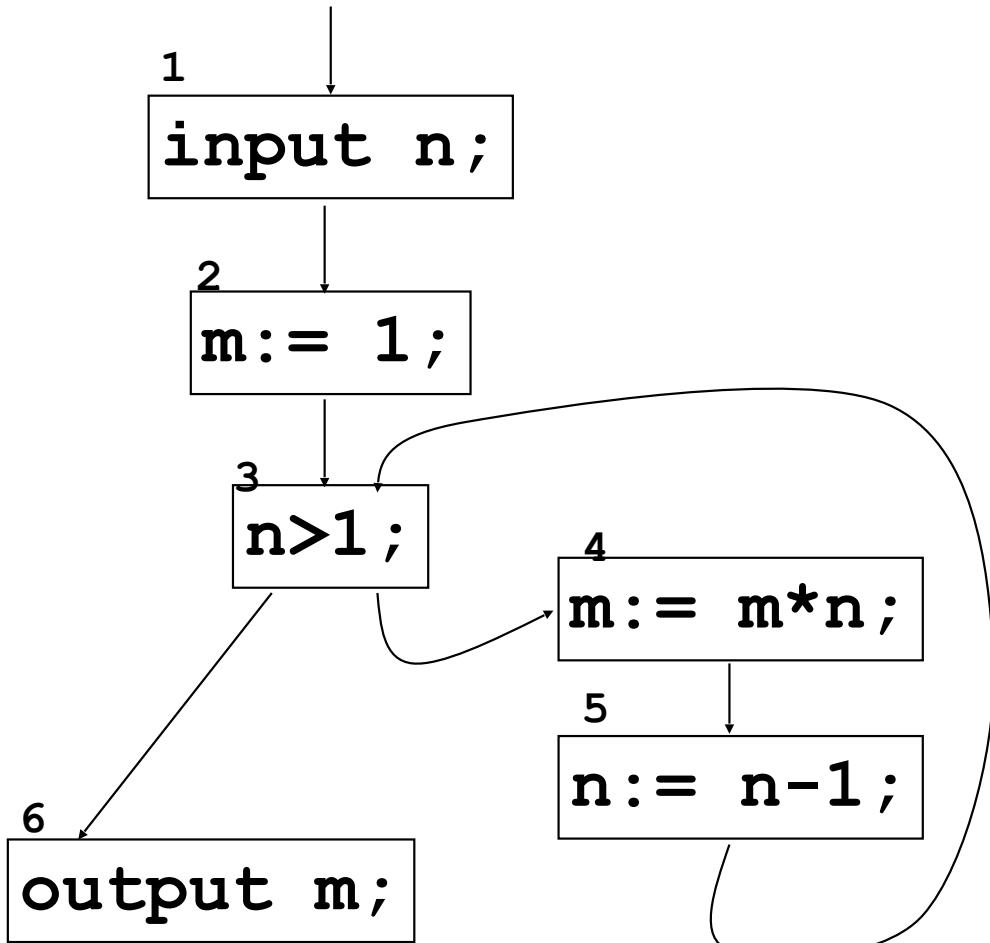


Specification

- $\text{kill}_{\text{RD}}[p] = \begin{cases} \{(x,q) \mid q \in \text{Points} \text{ and } \{x\} = \text{def}[q]\} & \text{if } \{x\} = \text{def}[p] \\ \emptyset & \text{if } \emptyset = \text{def}[p] \end{cases}$
- $\text{gen}_{\text{RD}}[p] = \begin{cases} \{(x,p)\} & \text{if } \{x\} = \text{def}[p] \\ \emptyset & \text{if } \emptyset = \text{def}[p] \end{cases}$

As usual, $\text{def}[p] = \{x\}$ when the command in the point p is an assignment
 $x := \text{exp}$

Kill and Gen



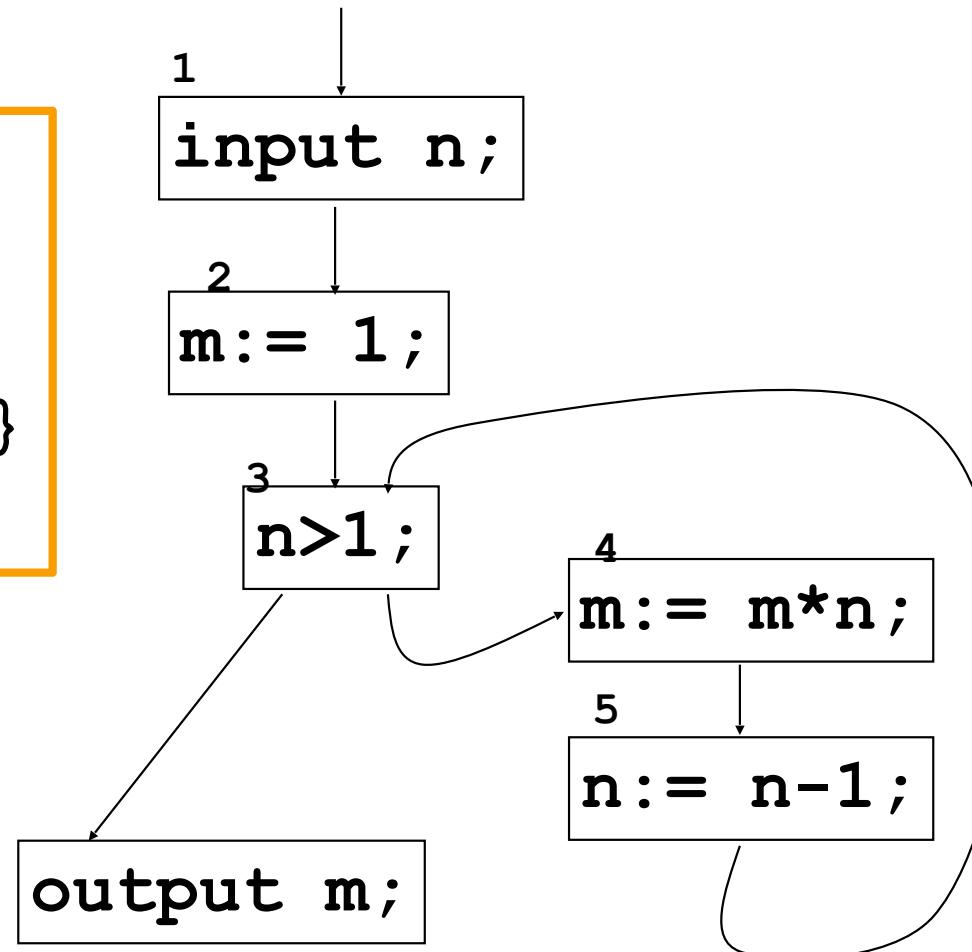
	kill _{RD}	gen _{RD}
1		
2	(m,?)(m,2) (m,4)	(m,2)
3		
4	(m,?)(m,2) (m,4)	(m,4)
5	(n,?) (n,5)	(n,5)
6		

Specification

- Reaching definitions analysis is specified by equations:

$$RD_{\text{entry}}(p) = \begin{cases} \{(x,?) \mid x \in \text{VARS}\} & \text{if } p \text{ is initial} \\ \bigcup \{RD_{\text{exit}}(q) \mid q \in \text{pre}[p]\} & \text{if } p \text{ is not initial} \end{cases}$$

$$RD_{\text{exit}}(p) = (RD_{\text{entry}}(p) \setminus \text{kill}_{RD}(p)) \cup \text{gen}_{RD}(p)$$



The solution of the previous system

Once again the solution for the equations in the previous system requires the existence of a fix point

We can apply the Kleene theorem if we have

- a) a continuous function on
- b) a CPO with bottom

Point b

$\langle (\mathcal{P}(\text{Vars} \times \text{Points}) \times \mathcal{P}(\text{Vars} \times \text{Points}))^{\mathbb{N}}, \subseteq^{2N} \rangle$

is a CPO with bottom?

- It is a CPO because it is finite
- Bottom?

TODO

Point a: the function

The map Reach:

$$\langle \mathcal{P}_{(\text{Vars} \times \text{Points}) \times \mathcal{P}_{(\text{Vars} \times \text{Points})}} \rangle^N \rightarrow \langle \mathcal{P}_{(\text{Vars} \times \text{Points}) \times \mathcal{P}_{(\text{Vars} \times \text{Points})}} \rangle^N$$

defined by

(assuming 1 is the only initial node)

Reach($\langle RD_{\text{entry}_1}, RD_{\text{exit}_1}, \dots, RD_{\text{entry}_N}, RD_{\text{exit}_N} \rangle$) =

$$\langle \{(x,?) \mid x \in \text{VARS}\}, (RD_{\text{entry}_1} \setminus kill_{RD}[1]) \cup gen_{RD}[1],$$

$$\cup \{RD_{\text{exit}_2} \mid m \in \text{pre}[2]\}, (RD_{\text{entry}_2} \setminus kill_{RD}[2]) \cup gen_{RD}[2],$$

....,

$$\cup \{RD_{\text{exit}_m} \mid m \in \text{pre}[N]\}, (RD_{\text{entry}_N} \setminus kill_{RD}[N]) \cup gen_{RD}[N] \rangle$$

TODO

Point a

Reach(<RDentry₁, RDexit₁, ..., RDentry_N, RDexit_N>) =

< {(x,?) | x in VARS}, (RD_{entry1}\kill_{RD}[1]) U gen_{RD}[1],

U{RD_{exit2} | m in pre[2]}, (RD_{entry2}\kill_{RD}[2]) U gen_{RD}[2]

....,

U{RD_{exitm} | m in pre[N]}, (RD_{entryN}\kill_{RD}[N]) U gen_{RD}[N] >

- Example

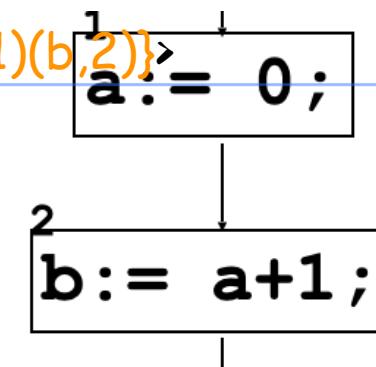
Reach(<{(a,?)}, {}, {}, {}>) = <{(a,?)(b,?)}, {(a,1)(b,?)}, {(a,1)(b,?)}, {(a,1)(b,2)}>

Reach(<{(a,?)(a,2)}, {(a,2)}, {}, {(b,1)}>) =

<{(a,?)(b,?)}, {(a,1)(b,?)}, {(a,1)(b,?)}, {(a,1),(b,2)}>

Note that Reach is monotone!

$$\begin{aligned} \text{kill}_{RD}(1) &= \{(a,?)\}, \text{gen}_{RD}(1) = \{(a,1)\} \\ \text{kill}_{RD}(2) &= \{(b,?)\}, \text{gen}_{RD}(2) = \{(b,2)\} \end{aligned}$$



Since it is monotone on a finite domain then it is continuous

Why a least fix point

RD analysis is possible,

if an assignment $x:=a$ in some point q is really actual in entry to some point p then

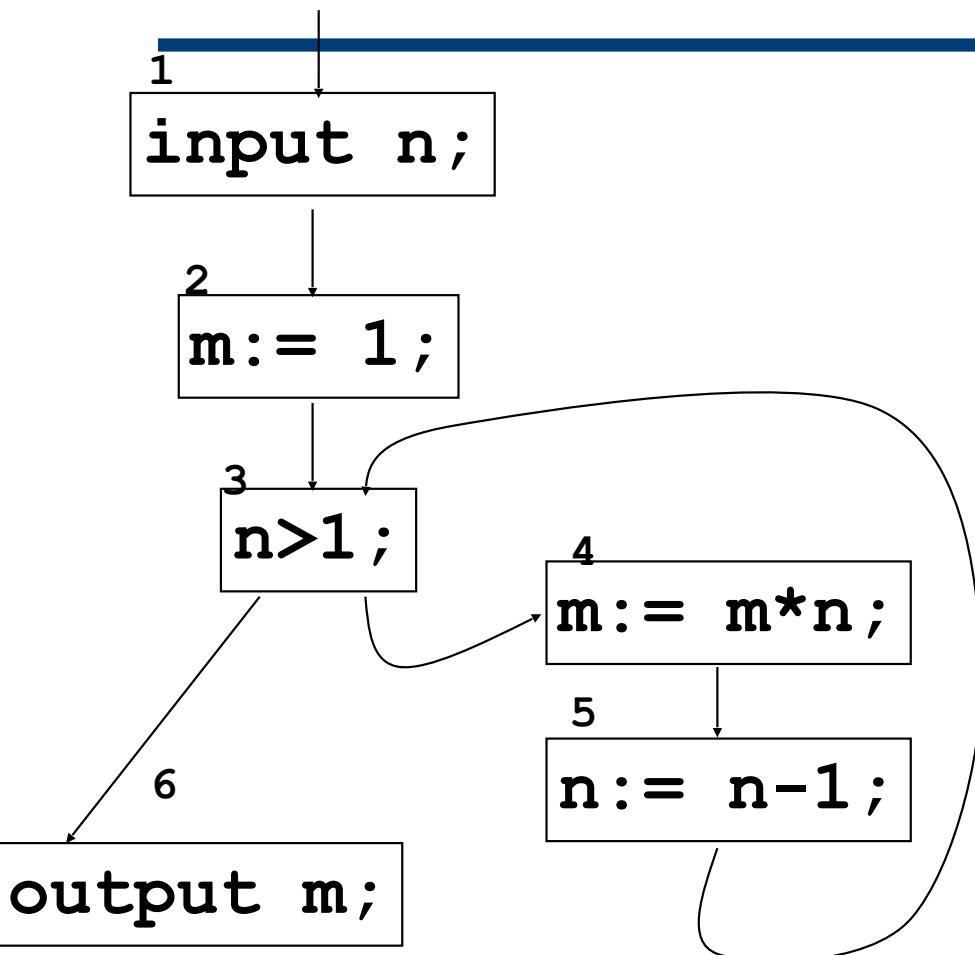
$$(x, q) \in RD_{\text{entry}}(p)$$

The vice versa does not hold

All fixpoints of the above equation system is an over-approximation of really reaching definitions.

Computing the least fixpoint gives a more precise over approximation

First iteration:



$RD_{entry}(p) = \{(x,?) \mid x \text{ in } Vars\}$, if p is initial

$RD_{entry}(p) = \bigcup \{RD_{exit}(q) \mid q \text{ in } pre[p]\}$, otherwise

$RD_{exit}(p) = (RD_{entry}(p) \setminus kill_{RD}[p]) \cup gen_{RD}[p]$

	kill	gen
2	(m,?)(m,2) (m,4)	(m,2)
4	(m,?)(m,2) (m,4)	(m,4)
5	(n,?) (n,5)	(n,5)

$$RD_{entry}(1) = \{(n,?), (m,?)\}$$

$$RD_{exit}(1) = \{(n,?), (m,?)\}$$

$$RD_{entry}(2) = \{(n,?), (m,?)\}$$

$$RD_{exit}(2) = \{(n,?), (m,2)\}$$

$$RD_{entry}(3) = \{(n,?), (m,2)\}$$

$$RD_{exit}(3) = \{(n,?), (m,2)\}$$

$$RD_{entry}(4) = \{(n,?), (m,2)\}$$

$$RD_{exit}(4) = \{(n,?), (m,4)\}$$

$$RD_{entry}(5) = \{(n,?), (m,4)\}$$

$$RD_{exit}(5) = \{(n,5), (m,4)\}$$

$$RD_{entry}(6) = \{(n,?), (m,2)\}$$

$$RD_{exit}(6) = \{(n,?), (m,2)\}$$

TODO

Second iteration:

1
input n;

2
 $m := 1;$

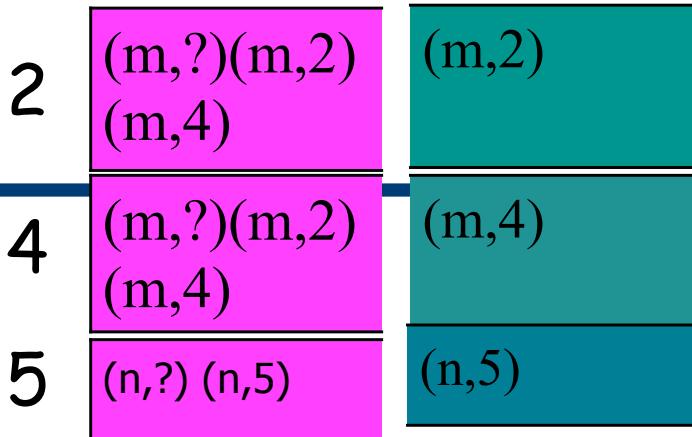
3
 $n > 1;$

4
 $m := m * n;$

5
 $n := n - 1;$

6
output m;

TODO



$RD_{entry}(1) = \{(n,?), (m,?)\}$	$RD_{exit}(1) = \{(n,?), (m,?)\}$
$RD_{exit}(1) = \{(n,?), (m,?)\}$	$RD_{exit}(1) = \{(n,?), (m,?)\}$
$RD_{entry}(2) = \{(n,?), (m,?)\}$	$RD_{entry}(2) = \{(n,?), (m,?)\}$
$RD_{exit}(2) = \{(n,?), (m,2)\}$	$RD_{exit}(2) = \{(n,?), (m,2)\}$
$RD_{entry}(3) = \{(n,?), (m,2)\}$	$RD_{entry}(3) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{exit}(3) = \{(n,?), (m,2)\}$	$RD_{exit}(3) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{entry}(4) = \{(n,?), (m,2)\}$	$RD_{entry}(4) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{exit}(4) = \{(n,?), (m,4)\}$	$RD_{exit}(4) = \{(n,?), (n,5)(m,4)\}$
$RD_{entry}(5) = \{(n,?), (m,4)\}$	$RD_{entry}(5) = \{(n,?), (n,5)(m,4)\}$
$RD_{exit}(5) = \{(n,5), (m,4)\}$	$RD_{exit}(5) = \{(n,5), (m,4)\}$
$RD_{entry}(6) = \{(n,?), (m,2)\}$	$RD_{entry}(6) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{exit}(6) = \{(n,?), (m,2)\}$	$RD_{exit}(6) = \{(n,?), (m,2), (n,5)(m,4)\}$

fix point!

$RD_{entry}(p) = \{(x,?) \mid x \in Vars\}$, if p is initial

$RD_{entry}(p) = \bigcup \{RD_{exit}(q) \mid q \in pre[p]\}$, otherwise

$RD_{exit}(p) = (RD_{entry}(p) \setminus kill_{RD}[p]) \cup gen_{RD}[p]$

RD analysis

- RD analysis is forward and **possible**,
i.e., if an assignment $x:=a$ in some point q is really actual in entry
to some point p then
 $(x,q) \in RD_{\text{entry}}(p)$ (while the vice versa does not hold).

How can we use this?

- If the analysis tells us that a variable is undefined (that is we just have the pair $(x,?)$) then it is
- Loop invariant code motions

TODO

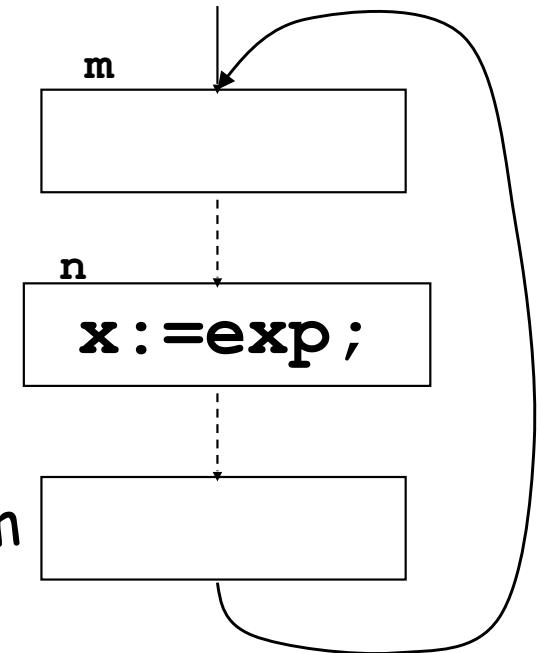
Application: Loop invariant code motion

Consider a loop where:

1. m is the entry point
2. an inner point n contains an assignment $x := \text{exp}$
3. if for any variable y occurring in exp (i.e. $y \in \text{vars}(\text{exp})$) and for any program point p, we have that

$$(y, p) \in \text{RD}_{\text{entry}}(m) \iff (y, p) \in \text{RD}_{\text{entry}}(n)$$

then, the assignment $x := \text{exp}$ can be correctly moved out as preceding the entry point of the loop



Application: Loop invariant code motion

Loop-invariant code motion

```
y:=3; z:=5;  
for(int i=0; i<9; i++) {  
    x = y + z;  
    a[i] = 2*i + x;  
}
```

```
y:=3; z:=5;  
x = y + z;  
for(int i=0; i<9; i++) {  
    a[i] = 2*i + x;  
}
```

Available Expressions Analysis

Let p be a program point. For each execution path ending in p , we want track the expressions that have already been evaluated and then not modified.

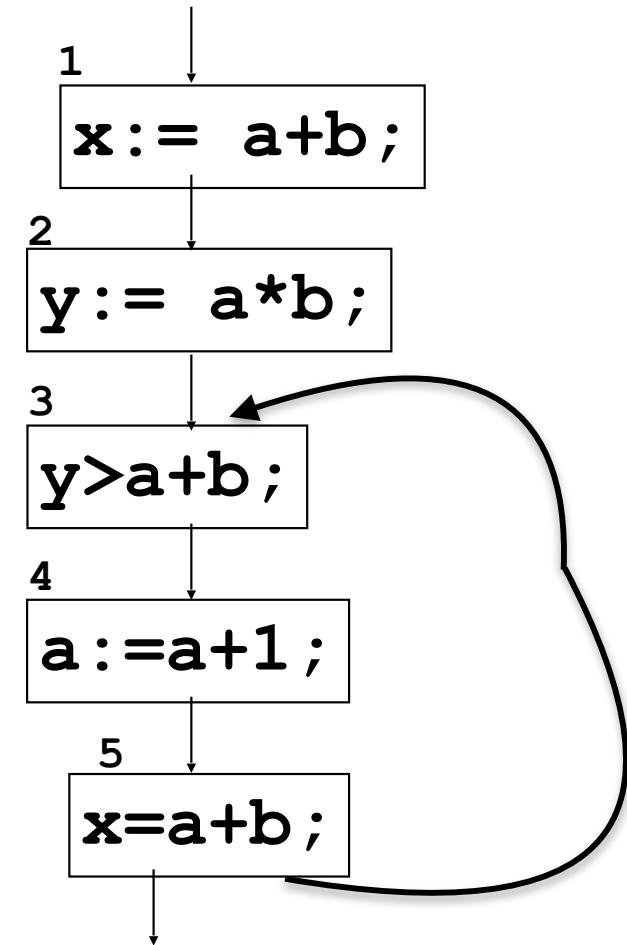
These are called **available expressions**

Example

```
x:=a+b;  
y:=a*b;  
while y>a+b  
do (a:=a+1;  
    x:=a+b;)
```

when the execution reaches 3, the expression $a+b$ is available, since it has been previously evaluated (in point 1 for the first iteration of the while-loop and in point 5 for the next iterations) and does not need to be evaluated again in 3

- This analysis can be therefore used to avoid re-evaluations of available expressions



The domain

Let $\mathbf{E} = \{ e \mid e \text{ is a sub-expressions/expression appearing in } P\}$

Let \mathbf{N} be the number of nodes of the CFG of P

$\langle (\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbf{N}}, \subseteq^{2^N} \rangle$ is a finite domain

Kill_{AE} and Gen_{AE}

- An expression e in E is killed in a program point p (e is in $\text{kill}_{AE}(p)$) if a variable occurring in e is modified (i.e., it is defined by some assignment) by the command in p .

$$\text{kill}_{AE}([x:=e']^p) = \{e \in E \mid x \in \text{vars}(e)\}$$

- An expression e is generated in a program point p (e is in $\text{gen}_{AE}(p)$) if e is evaluated in p and no variable occurring in e is modified in p .

$$\text{gen}_{AE}([x:=e]^p) = \{e\} \quad \text{if } x \notin \text{vars}(e),$$

$$\text{gen}_{AE}([x:=e]^p) = \emptyset \quad \text{if } x \in \text{vars}(e);$$

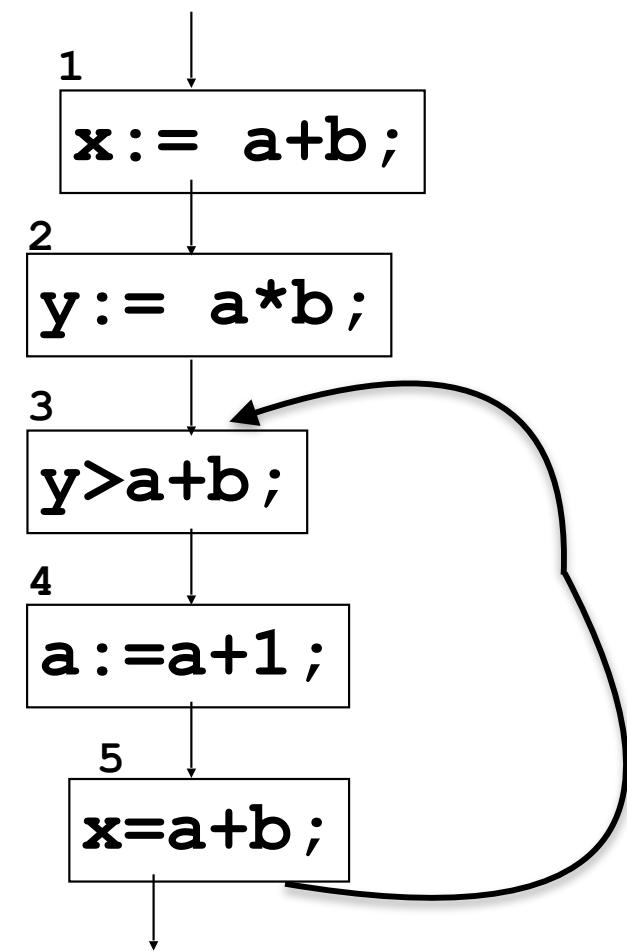
$\text{gen}_{AE}([e_1 > e_2]^p) = \text{expr}(\{e_1, e_2\})$ where $\text{expr}(S)$ returns
the subset of S that are expressions

Example

$x := a + b; y := a^*b; \text{while } y > a + b \text{ do } (a := a + 1; x := a + b)$

$$E = \{a + b, a^*b, a + 1\}$$

n	$\text{kill}_{AE}(n)$	$\text{gen}_{AE}(n)$
1	\emptyset	{a+b}
2	\emptyset	{a*b}
3	\emptyset	{a+b}
4	{a+b, a*b, a+1}	\emptyset
5	\emptyset	{a+b}



Specification

- Available expressions analysis is specified by the following equations, for any program point p:

$$AE_{\text{entry}}(p) = \begin{cases} \emptyset & \text{if } p \text{ is initial} \\ \cap \{AE_{\text{exit}}(q) \mid q \in \text{pre}[p]\} & \text{otherwise} \end{cases}$$

$$AE_{\text{exit}}(p) = (AE_{\text{entry}}(p) \setminus \text{kill}_{AE}(p)) \cup \text{gen}_{AE}(p)$$

Point a and b to apply Kleene Theorem

To find a solution to the previous equation system we need to apply Kleene Theorem

b) $(\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbb{N}}$, \subseteq^{2^N} is a finite domain therefore is a CPO, moreover, it has a bottom element

a) The map $(\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbb{N}} \rightarrow (\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbb{N}}$ defined by
(assuming 1 is the only initial node)

$$\begin{aligned} AE(<AE_{entry1}, AE_{exit1}, \dots, AE_{entryN}, AE_{exitN}>) &= \\ &< \emptyset, (AE_{entry1} \setminus kill_{AE}(1)) \cup gen_{AE}(1), \\ &\cap \{AE_{exitq} \mid q \text{ in } pre[2]\}, (AE_{entry2} \setminus kill_{AE}(2)) \cup gen_{AE}(2), \\ &\dots \\ &\cap \{AE_{exitq} \mid q \text{ in } pre[N]\}, (AE_{entryN} \setminus kill_{AE}(N)) \cup gen_{AE}(N)> \end{aligned}$$

Point a

a) The map

$$AE(<AE_{entry_1}, AE_{exit_1}, \dots, AE_{entry_N}, AE_{exit_N}>) =$$

$$<\emptyset, (AE_{entry_1} \setminus kill_{AE}(1)) \cup gen_{AE}(1),$$

$$\cap \{AE_{exit_q} \mid q \text{ in } pre[2]\}, (AE_{entry_2} \setminus kill_{AE}(2)) \cup gen_{AE}(2),$$

.....

$$\cap \{AE_{exit_q} \mid q \text{ in } pre[N]\}, (AE_{entry_N} \setminus kill_{AE}(N)) \cup gen_{AE}(N)>$$

is monotone on the finite domain

$$(\mathcal{P}(E) \times \mathcal{P}(E))^N, \subseteq^{2^N} >$$

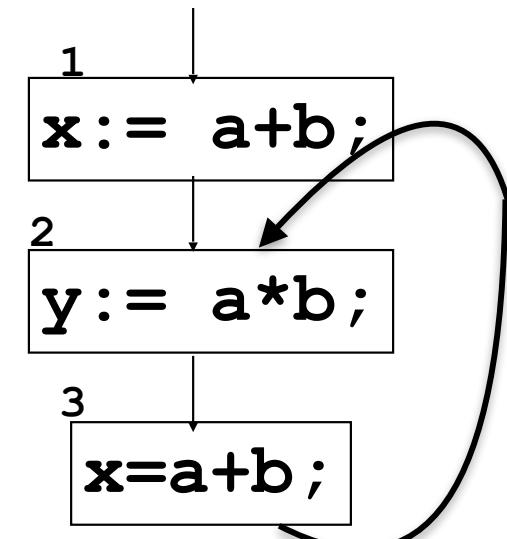
• Example

$$AE(<\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset>) =$$

$$<\emptyset, \{a+b\}, \{\}, \{a^*b\}, \{a^*b\}, \{a+b, a^*b\}>$$

$$AE(<\emptyset, \{a+b\}, \{\}, \{a^*b\}, \{a^*b\}, \{a+b, a^*b\}>) =$$

$$<\emptyset, \{a+b\}, \{a+b\}, \{a+b, a^*b\}, \{a+b, a^*b\}, \{a+b, a^*b\}>$$



Which fix point?

AE is a definite analysis:

if $e \in AE_{\text{entry}}(p)$ then e is really available in entry to p

the converse does not hold

- Any fixpoint of the above equation system is an under-approximation of really available expressions.

Between all fix points, we are thus interested in computing the greatest fixpoint (the more precise approximation)

Also, observe that this is a forward analysis.

The starting point, for all n
 $AE_{entry}(n) = AE_{exit}(n) = \{a+b, a^*b, a+1\}$

Computing the greatest fix point

$x := a+b; y := a^*b; \text{while } y > a+b \text{ do } (a := a+1; x := a+b)$

$$E = \{a+b, a^*b, a+1\}$$

n	kill _{AE} (n)	gen _{AE} (n)
1	\emptyset	{a+b}
2	\emptyset	{a^*b}
3	\emptyset	{a+b}
4	{a+b, a^*b, a+1}	\emptyset
5	\emptyset	{a+b}

$$AE_{entry}(1) = \emptyset$$

$$AE_{entry}(2) = \{a+b\}$$

$$AE_{entry}(3) = \{a+b, a^*b\}$$

$$AE_{entry}(4) = \{a+b, a^*b\}$$

$$AE_{entry}(5) = \{\}$$

$$AE_{exit}(1) = \{a+b\}$$

$$AE_{exit}(2) = \{a+b, a^*b\}$$

$$AE_{exit}(3) = \{a+b, a^*b\}$$

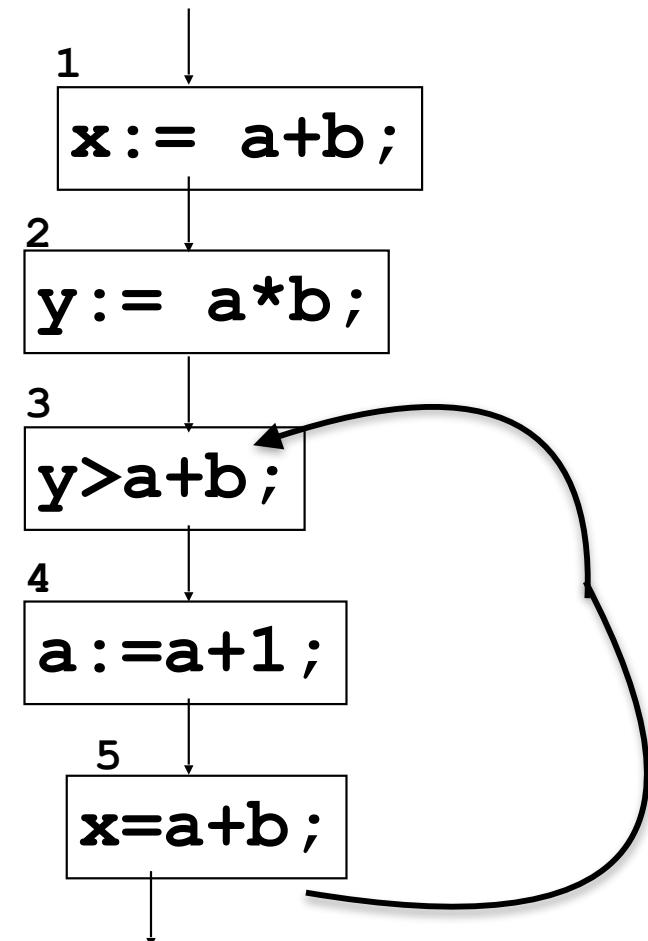
$$AE_{exit}(4) = \{\}$$

$$AE_{exit}(5) = \{a+b\}$$

$$AE_{entry}(p) = \emptyset \text{ if } p \text{ is initial}$$

$$AE_{entry}(p) = \bigcap \{AE_{exit}(q) \mid q \text{ in pre}[p]\}$$

$$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$$



Second iteration

$AE_{entry}(p) = \emptyset$ if p is initial

$AE_{entry}(p) = \bigcap \{AE_{exit}(q) \mid q \text{ in } pre[p]\}$

$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$

Previous iteration

n	$AE_{entry}(n)$	$AE_{exit}(n)$
1	\emptyset	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b, a*b}	{a+b, a*b}
4	{a+b, a*b}	\emptyset
5	\emptyset	{a+b}

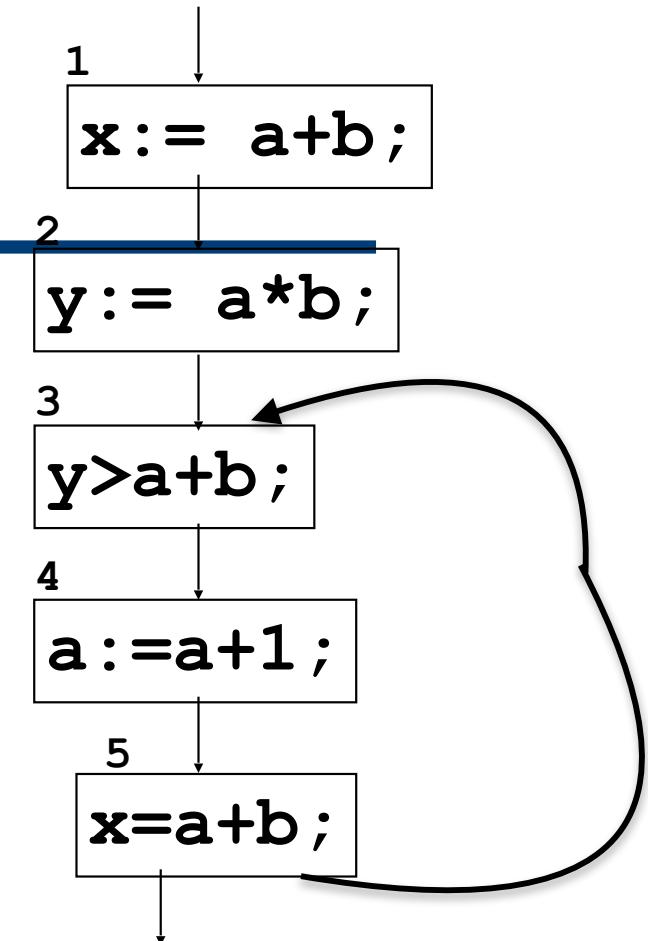
$$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$$

$$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$$

$$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$$

$$AE_{exit}(4) = AE_{entry}(4) - \{a+b, a*b, a+1\}$$

$$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$$



n	$AE_{entry}(n)$	$AE_{exit}(n)$
1	\emptyset	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	\emptyset
5	\emptyset	{a+b}

Third iteration and Greatest Fixpoint

$AE_{entry}(p) = \emptyset$ if p is initial

$AE_{entry}(p) = \bigcap \{AE_{exit}(q) \mid q \text{ in } pre[p]\}$

$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$

Previous iteration

n	$AE_{entry}(n)$	$AE_{exit}(n)$
1	\emptyset	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	\emptyset
5	\emptyset	{a+b}

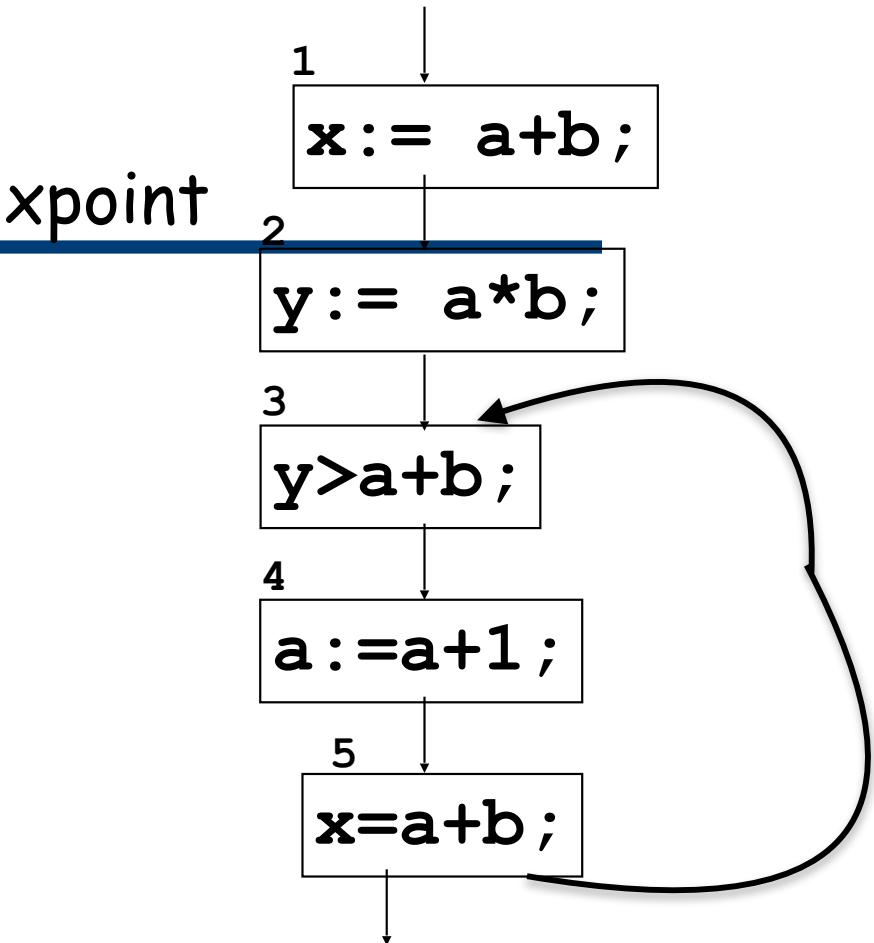
$$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$$

$$AE_{exit}(2) = AE_{entry}(2) \cup \{a^*b\}$$

$$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$$

$$AE_{exit}(4) = AE_{entry}(4) - \{a+b, a^*b, a+1\}$$

$$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$$

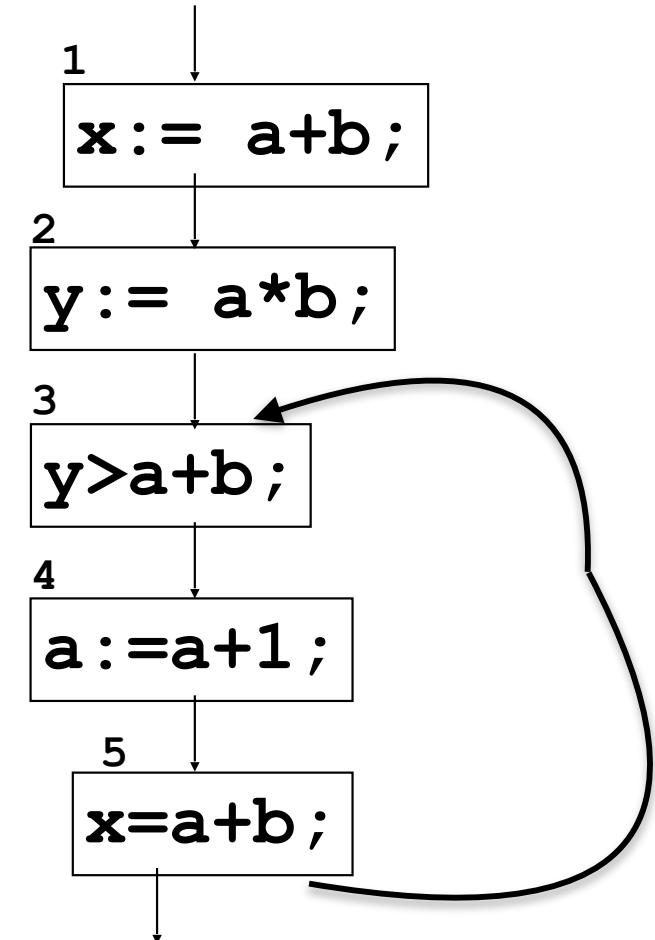


n	$AE_{entry}(n)$	$AE_{exit}(n)$
1	\emptyset	{a+b}
2	{a+b}	{a+b, a^*b}
3	{a+b}	{a+b}
4	{a+b}	\emptyset
5	\emptyset	{a+b}

Result

$x := a+b; y := a^*b; \text{while } y > a+b \text{ do } (a := a+1; x := a+b)$

n	$AE_{\text{entry}}(n)$	$AE_{\text{exit}}(n)$
1	\emptyset	{ $a+b$ }
2	{ $a+b$ }	{ $a+b, a^*b$ }
3	{$a+b$}	{ $a+b$ }
4	{ $a+b$ }	\emptyset
5	\emptyset	{ $a+b$ }

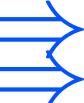


Dataflow Analyses

A Dataflow Analysis Framework

- The above dataflow analyses (Reaching Definitions, Available Expressions, Live Variables) reveal many similarities.
- One major advantage of a unifying framework of dataflow analysis lies in the design of a generic analysis algorithm that can be instantiated in order to compute different dataflow analyses.

Catalogue of Dataflow Analyses

	<i>Possible Analysis</i> Semantics \subseteq Analysis	<i>Definite Analysis</i> Analysis \subseteq Semantics
<i>Forward</i> in[n]  out[n] pre  post	Reaching definitions	Available expressions
<i>Backward</i> out[n]  in[n] post  pre	Live variables	Very busy expressions

TODO ↑

Principles of Abstract Interpretation



Program Analysis

A technique to check if a program satisfies a semantic property

Useful for optimisation and verification

What to Analyse:

Target Programs

- Domain-specific vs Non-domain-specific analyses
- Program-level vs Model-level analyses

Target Properties

- Safety properties: some behavior observable in finite time will never occur.
- Liveness properties: some behavior observable after infinite time will never occur.
- Information flow properties

When to Analyse:

Dynamic vs Static techniques

What to Analyse: Safety Properties

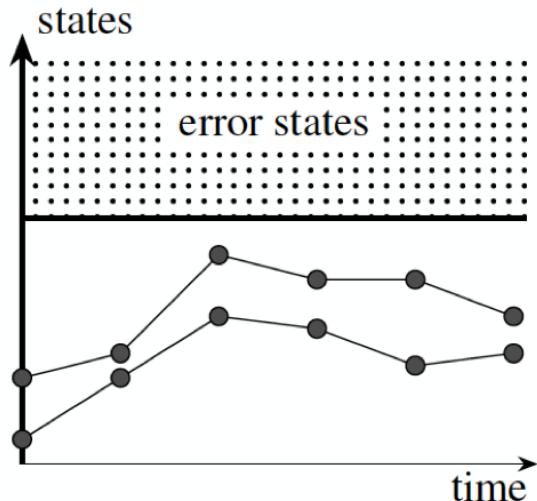
Some behaviors observable in finite time will never occur.

Examples:

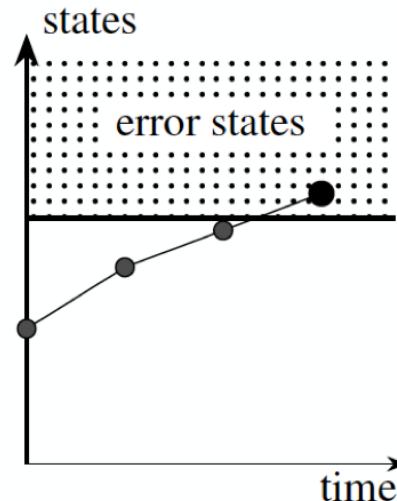
- No crashing error — e.g., no divide by zero, no uncaught exceptions, etc
- No invariant violation
 - Loop invariant: assertion that holds at the beginning of every loop iteration

What to Analyse: Safety Properties

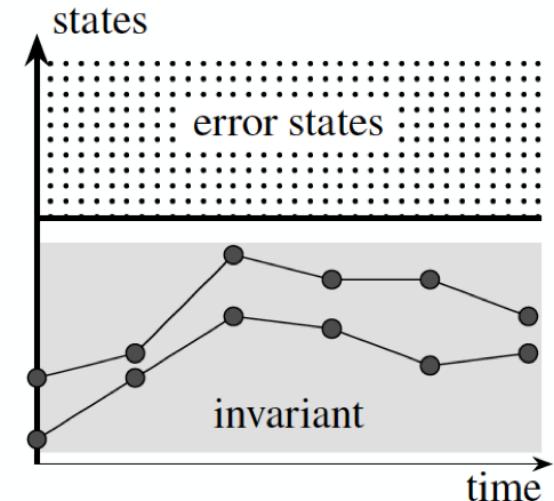
```
x = 0;  
while (x < 10) “x is an integer”  
{ x = x + 1;} “0 <= x < 10”
```



(a) Correct executions



(b) An incorrect execution



(c) Proof by invariance

What to Analyse: Liveness Properties

Some behaviors observable after infinite time will never occur

Examples:

- No unbounded repetition of a given behavior
- No non-termination

What to Analyse: Liveness Properties

```
x = read_int ();
while ( x > 0 )
{ x = x - 1; }
```

- If x is initially a negative integer \Rightarrow the program terminates
- If x is initially a positive integer $\Rightarrow x$ strictly decreases every iteration
 \Rightarrow the program terminates

Undecidability in the way

Rice theorem.

Let L be a Turing-complete language, and let P be a nontrivial semantic property of programs of L .

There exists no algorithm such that, for every program $p \in L$, it returns true if and only if p satisfies the semantic property P

```
while (x>0)
    x=x+1;
print("27");
```

Limitations of the analysis

We need to give something up:

automation: machine-assisted techniques

the universality "for all programs": targeting only a restricted class of programs

claim to find exact answers: introduce approximations

Approximation: Soundness and Completeness

Given a semantic property P and a program $p \in L$. An analysis is perfectly accurate iff

for all program p , $\text{analysis}(p) = \text{true} \Leftrightarrow p \text{ satisfies the property } P$

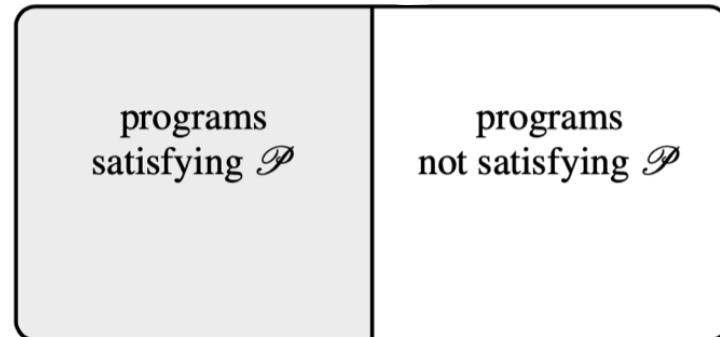
which consists of

1) for all program $p \in L$, $\text{analysis}(p) = \text{true} \Rightarrow p \text{ satisfies } P$ (soundness)

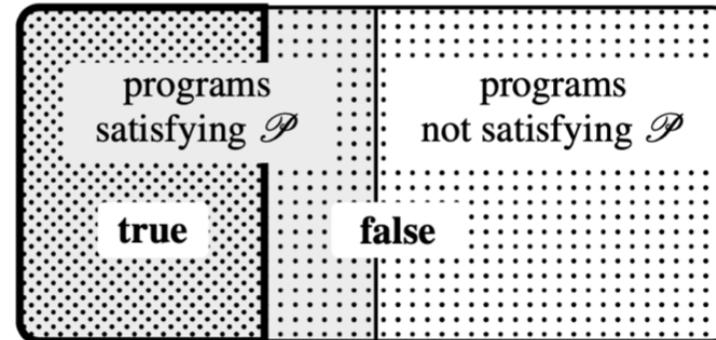
2) for all program $p \in L$, $\text{analysis}(p) = \text{true} \Leftarrow p \text{ satisfies } P$ (completeness)



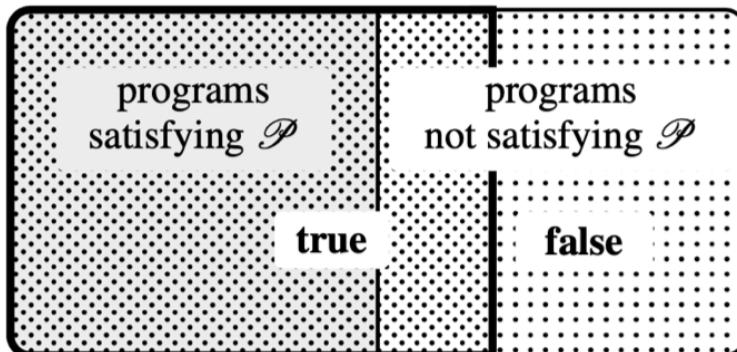
Approximation: Soundness and Completeness



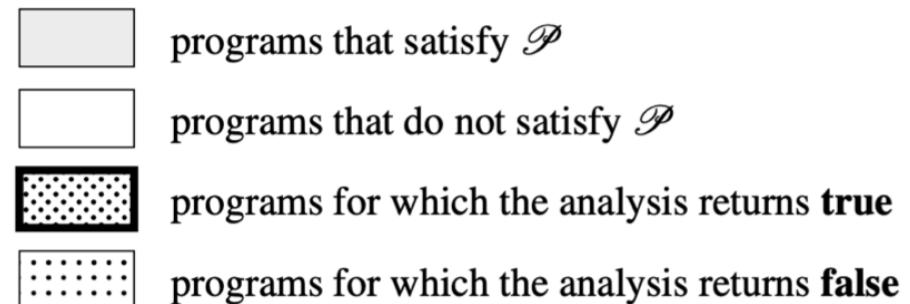
(a) Programs



(b) Sound, incomplete analysis



(c) Unsound, complete analysis



(d) Legend

Spectrum of Program Analysis Techniques

Testing

Machine-assisted proving

Finite-state model checking

Conservative static analysis

Bug-finding

Comparison

	automatic	sound	complete
testing	yes	no	yes
machine-assisted proving	no	yes	yes/no
finite-state model checking	yes	yes/no	yes/no
conservative static analysis	yes	yes	no
bug-finding	yes	no	no

Abstract Interpretation

A general technique, for any programming language L and safety property S , that checks, for input program P in L , if $[|P|]$ is contained in S

automatic (software)

finite (terminating)

sound (guarantee)

malleable for arbitrary precision

Denotational Semantics

Semantics

What is the meaning of a program "1 + 2" ?

Meaning = what it "denotes":

"3" (Denotational semantics)

Meaning = how to compute the result:

"add 1 into 2 and get 3" (Operational semantics)

Different approaches for different purposes and languages

Denotational Semantics

Mathematical meaning of a program (no machine states or transitions)

Program semantics is a function from input states to output states

The semantics of a program is determined by that of each component (compositionality principle)

Semantics of a Simple Language (WHILE)

$$C \rightarrow \begin{array}{l} \text{skip} \\ | \\ x := E \\ | \\ \text{if } E C C \\ | \\ C; C \\ | \\ \text{while } E C \end{array}$$
$$E \rightarrow \begin{array}{ll} n & (n \in \mathbb{Z}) \\ | \\ x \\ | \\ E + E \\ | \\ - E \end{array}$$

The semantics of C is a function from memories to memories

Memory = Function from memory locations to values

Semantic Domain

A set of objects used to define program semantics (i.e., semantic objects)

$x \in \mathbb{X} = \text{ProgramVariables}$

$\mathbb{V} = \mathbb{Z}$

$m \in \mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$

Memories

Meaning of commands

$\llbracket C \rrbracket : \mathbb{M} \rightarrow (\mathbb{M} \cup \perp)$

Meaning of expressions

$\llbracket E \rrbracket : \mathbb{M} \rightarrow \mathbb{V}$

may diverge

Denotational Semantics of Expressions

$$\begin{aligned} \llbracket x \rrbracket \ m &= m(x) \\ \llbracket n \rrbracket \ m &= n \\ \llbracket E_1 + E_2 \rrbracket \ m &= (\llbracket E_1 \rrbracket \ m) + (\llbracket E_2 \rrbracket \ m) \\ \llbracket -E \rrbracket \ m &= -(\llbracket E \rrbracket \ m) \end{aligned}$$

$$\begin{aligned} \llbracket 3 + x \rrbracket \{x \mapsto 2, y \mapsto 1\} &= \llbracket 3 \rrbracket \{x \mapsto 2, y \mapsto 1\} + \llbracket x \rrbracket \{x \mapsto 2, y \mapsto 1\} \\ &= 3 + 2 = 5 \end{aligned}$$

Denotational Semantics of Commands

$$[\![\text{skip}]\!] \ m = m$$

$$[\![x := E]\!] \ m = m \ \{x \mapsto [\![E]\!]m\}$$

$$[\![\text{if } E \ C_1 \ C_2]\!] \ m = \text{ if } [\![E]\!]m \neq 0 \text{ then } [\![C_1]\!]m \text{ else } [\![C_2]\!]m$$

$$[\![C_1 ; C_2]\!] \ m = [\![C_2]\!]([\![C_1]\!]m)$$

memory update

may diverge

$$[\![C]\!] \perp = \perp$$

E.g., $[\![x:=7; y:=3]\!]\{\} = \{x \mapsto 7, y \mapsto 3\}$

Compositional!

(i.e., the semantics of a program is determined by that of its sub-components)

Semantics of Loops

The semantics of `while E C`

$\llbracket \text{while } E \text{ } C \rrbracket m$

$= \text{if } \llbracket E \rrbracket m \neq 0 \text{ then } \llbracket \text{while } E \text{ } C \rrbracket (\llbracket C \rrbracket M) \text{ else } m$

is not compositional!

Not a definition, but a recursive equation!

Semantics of Loops

$\llbracket \text{while } E \text{ } C \rrbracket m$

$= \text{if } \llbracket E \rrbracket m \neq 0 \text{ then } \llbracket \text{while } E \text{ } C \rrbracket (\llbracket C \rrbracket m) \text{ else } m$

PSC

$\llbracket \text{while } E \text{ } C \rrbracket =$

$\lambda m. \text{if } \llbracket E \rrbracket m \neq 0 \text{ then } \llbracket \text{while } E \text{ } C \rrbracket (\llbracket C \rrbracket m) \text{ else } m$

how to denote functions:

$\lambda x. \text{function body}$

where x is the parameter

e.g. $\text{inc}(x) = x + 1$ vs

$\text{inc} = \lambda x. x + 1$



$$F_{E,C}(X) = \lambda m. \begin{cases} X(\llbracket C \rrbracket m) & \text{if } \llbracket E \rrbracket m \neq 0 \\ m & \text{otherwise} \end{cases}$$

$\llbracket \text{while } E \text{ } C \rrbracket = F_{E,C}(\llbracket \text{while } E \text{ } C \rrbracket)$

Semantics of Loops

Semantics of a loop: a solution of this equation

$$[\![\text{while } E \text{ } C]\!] = F_{E,C}([\![\text{while } E \text{ } C]\!])$$

Solution: a fixed point of $F_{E,C}$

$$F_{E,C}(X) = \lambda m. \begin{cases} X([\![C]\!] m) & \text{if } [\![E]\!] m \neq 0 \\ m & \text{otherwise} \end{cases}$$

Domain for Commands

$\llbracket C \rrbracket : \mathbb{M} \rightarrow \mathbb{M}_\perp$ where $\forall m \in \mathbb{M}, \perp \sqsubseteq m$

$\llbracket \text{while } E \text{ } C \rrbracket = F_{E,C}(\llbracket \text{while } E \text{ } C \rrbracket)$

$(\mathbb{M} \rightarrow \mathbb{M}_\perp)$

A partial function,
Sets of pairs (m, m')

$$F_{E,C}(X) = \lambda m. \begin{cases} X(\llbracket C \rrbracket m) & \text{if } \llbracket E \rrbracket m \neq 0 \\ m & \text{otherwise} \end{cases}$$

$$F_{E,C} : (\mathbb{M} \rightarrow \mathbb{M}_\perp) \rightarrow (\mathbb{M} \rightarrow \mathbb{M}_\perp)$$

It is monotone and continuous on the domain of partial functions



Semantics of while

By applying Kleene's theorem

$$[\![\text{while } E \text{ } C]\!] = \text{fix } F_{E,C} = \bigsqcup_n F_{E,C}^n(\frac{\perp_{\mathbb{M} \rightarrow \mathbb{M}}}{\lambda \sigma. \perp})$$

$\text{while } \underbrace{x > 1}_E \text{ do } \underbrace{x := x - 1}_C$

$$F_{E,C}(X) = \lambda m. \begin{cases} (m, X(m[m(x) - 1/x])) & m(x) > 1 \\ (m, m) & m(x) \leq 1 \end{cases}$$

$$F_{E,C}^0(\emptyset) = \emptyset$$

$$F_{E,C}(X) = \lambda m. \begin{cases} (m, m') & m(x) > 1, (m[m(x) - 1/x], m') \in X \\ (m, m) & m(x) \leq 1 \end{cases}$$

$$F_{E,C}^1(\emptyset) = \{(m, m) \mid m(x) \leq 1\} \supseteq \{(m[1/x], m[1/x])\}$$

$$F_{E,C}^2(\emptyset) = F_{E,C}^1(\emptyset) \cup \{(m, m[1/x]) \mid m(x) = 2\}$$

$$F_{E,C}^3(\emptyset) = F_{E,C}^2(\emptyset) \cup \{(m, m[1/x]) \mid m(x) = 3\}$$

...

$$F_{E,C}^n(\emptyset) = \{(m, m) \mid m(x) \leq 1\} \cup \{(m, m[1/x]) \mid 1 < m(x) \leq n\}$$

...

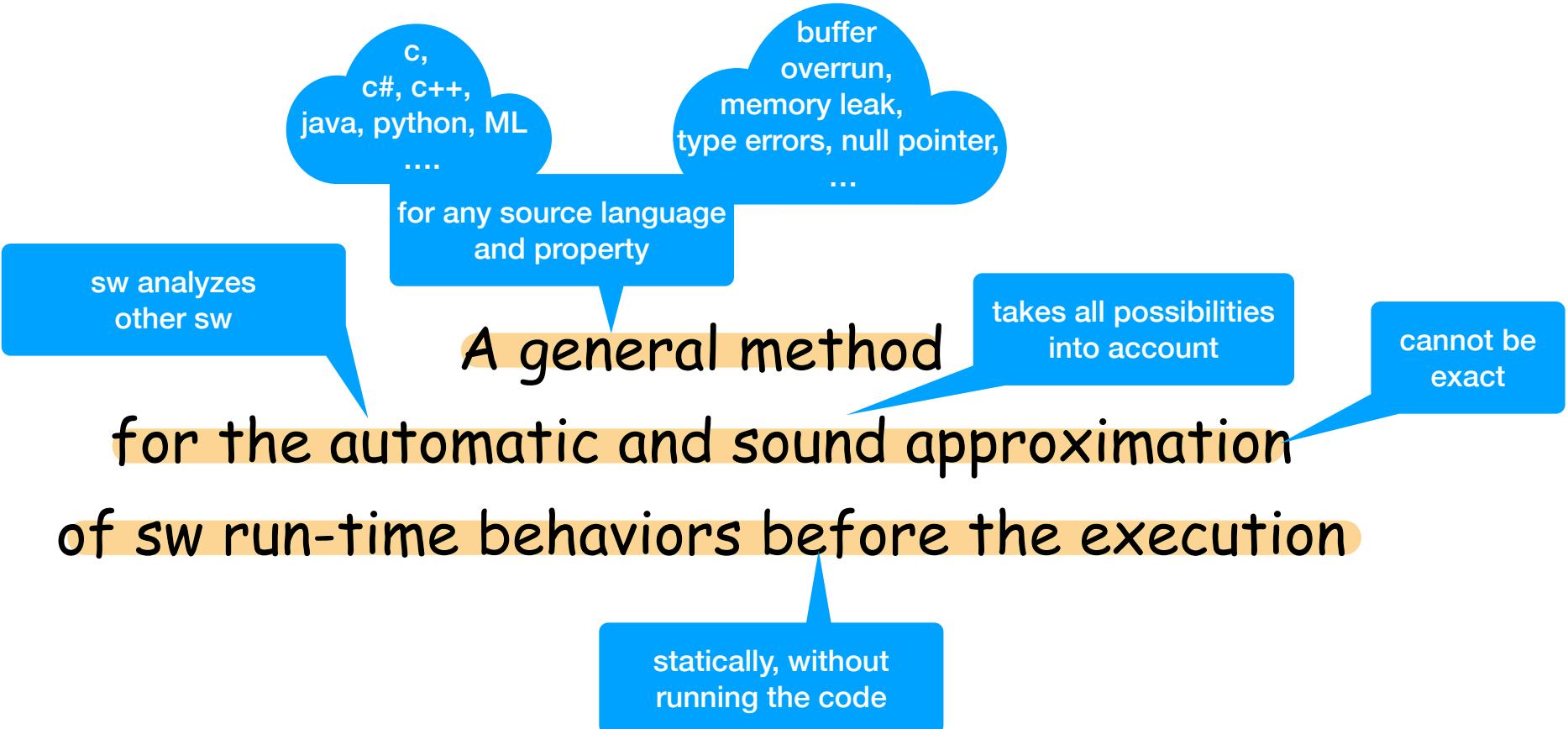
$$\text{fix } F_{E,C} = \{(m, m) \mid m(x) \leq 1\} \cup \{(m, m[1/x]) \mid 1 < m(x)\}$$

Principles of Abstract Interpretation

14

A gentle introduction to static analysis

Abstract Interpretation



Abstract interpretation

can explain all static analyses

guiding principles,
reusability, sound by design

A powerful framework
for designing correct static analyses:
simple and eye-opening

apply some recipes

any static analysis is
an abstract interpretation

Why abstraction?

it is not possible to terminate and
capture all possible executions!

wait, there is also 0!

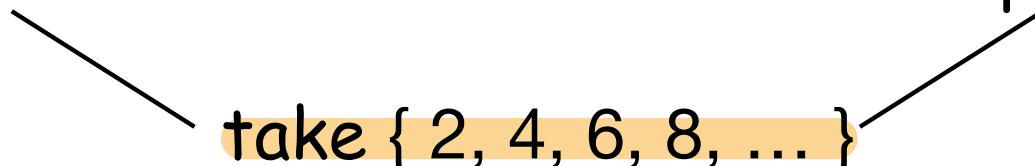
abstraction ≠ omission

hey, what about 2, 6, ...!

take even numbers

take multiples of 4

take { 2, 4, 6, 8, ... }



Which values can be printed?

repeat a random
number of times,
possibly none

unknown guard

```
x:=3 ; while (?) { x:=x+2 } ; x:=x-1; print(x) ;
```

concrete executions: infinitely many values (2, 4, 6, 8, ...)

abstract interpretation 1: integers (too coarse)

abstract interpretation 2: positive integers (precise)

abstract interpretation 3: even integers (precise)

abstract interpretation 4: positive even integers (more precise)

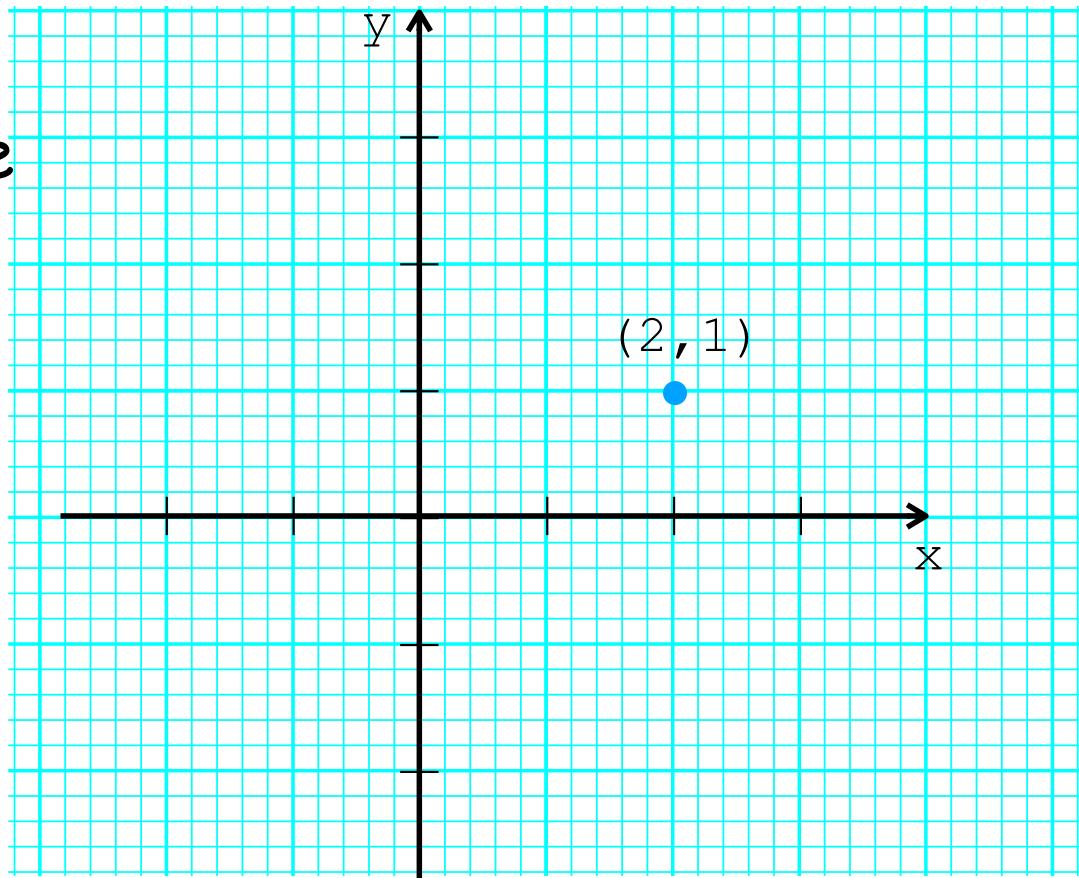
Compositional approach

A sample language

state: (x, y)

a point in the Cartesian plane

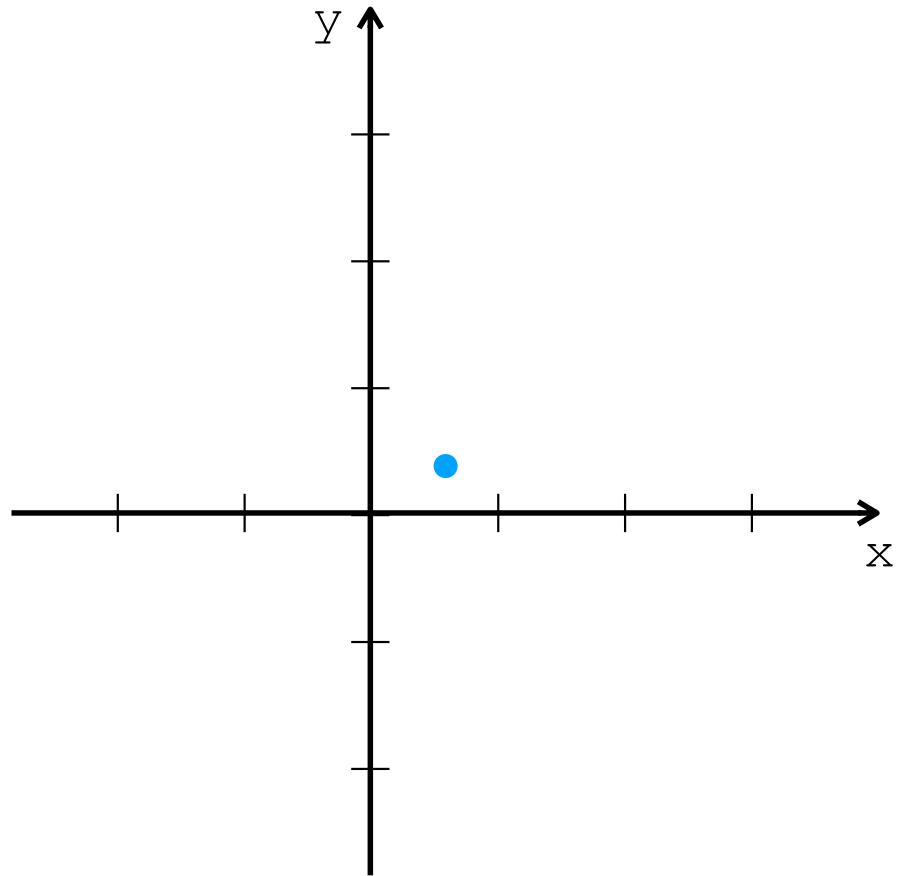
```
p ::= init(Region)
  | translate( $u, v$ )
  | rotate( $u, v, \theta$ )
  | p ; p
  | {p} or {p}
  | iter{p}
```



Concrete semantics

init($[0, 1] \times [0, 1]$)

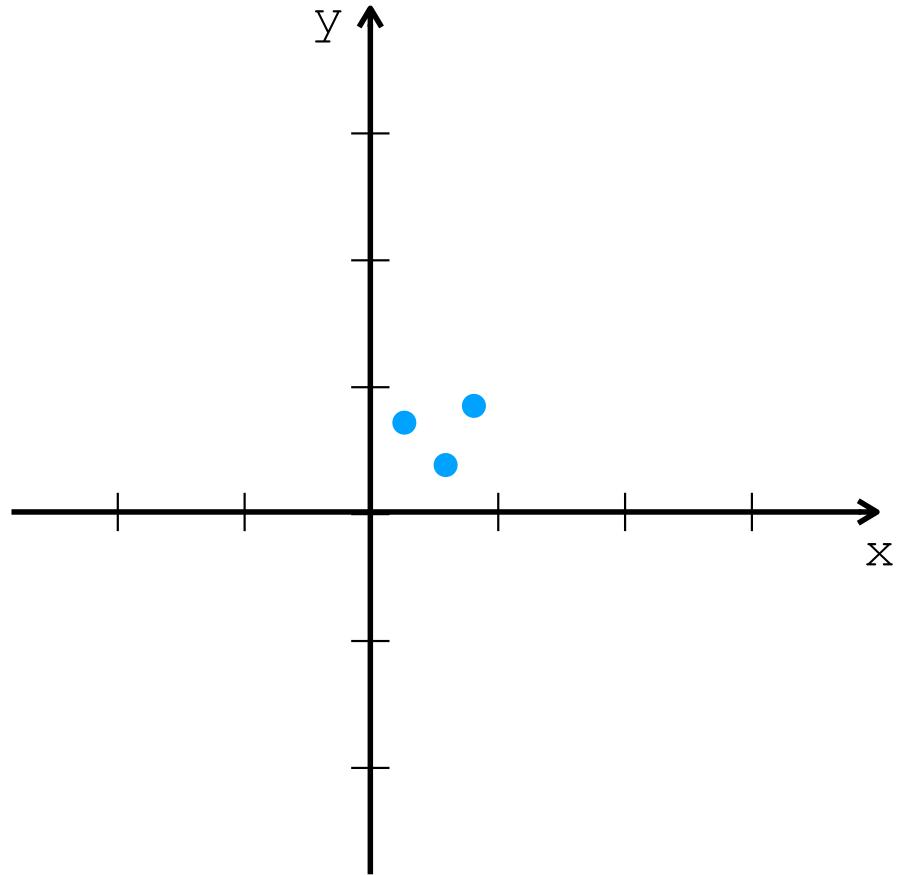
pick a state in the
region $[0,1] \times [0,1]$



Nondeterminism

init($[0, 1] \times [0, 1]$)

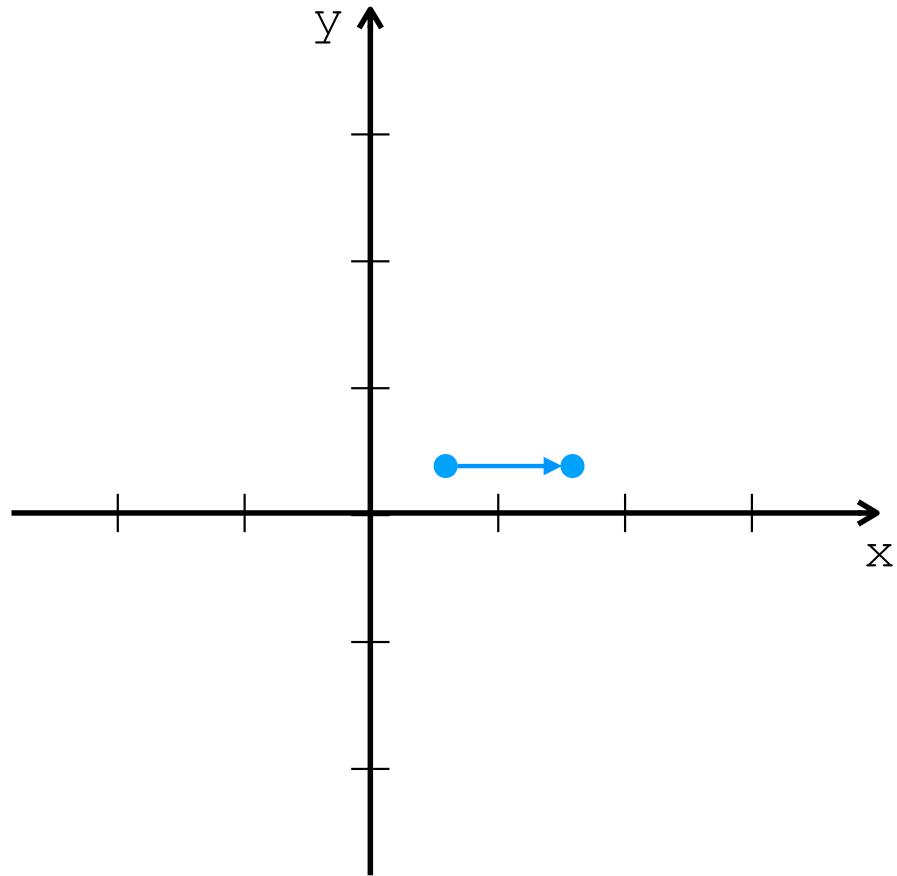
pick a state in the
region $[0,1] \times [0,1]$



Concrete semantics

translate(1, 0)

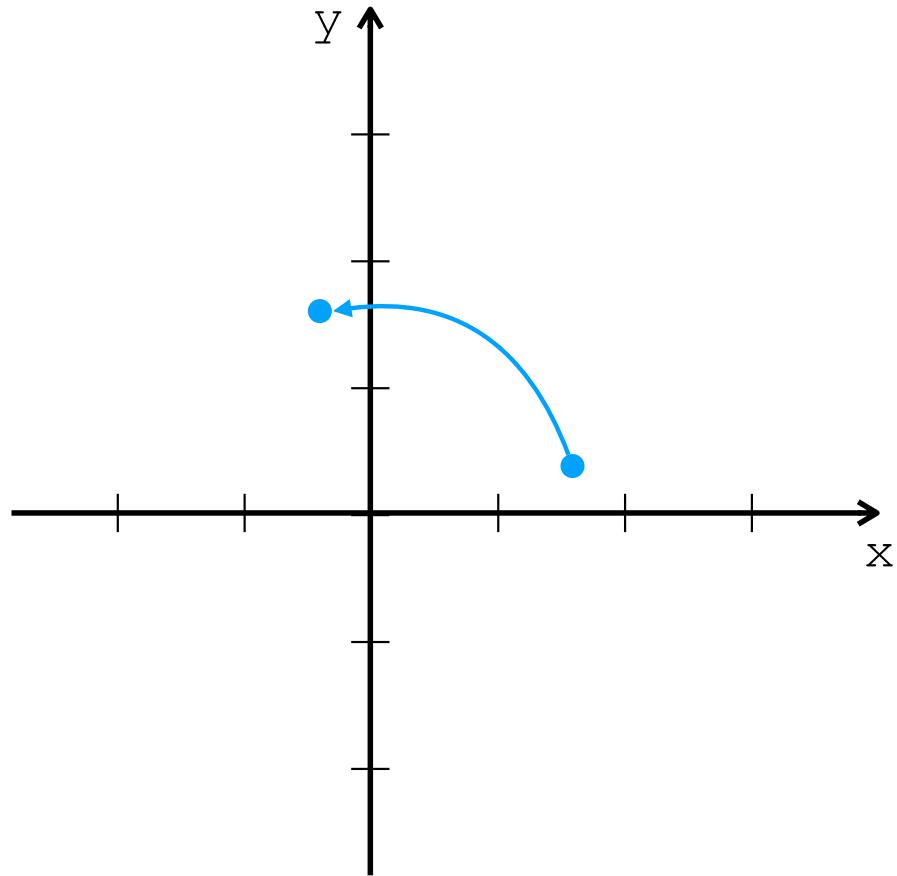
translate the current
state by vector (1,0)



Concrete semantics

rotate($0, 0, 90^\circ$)

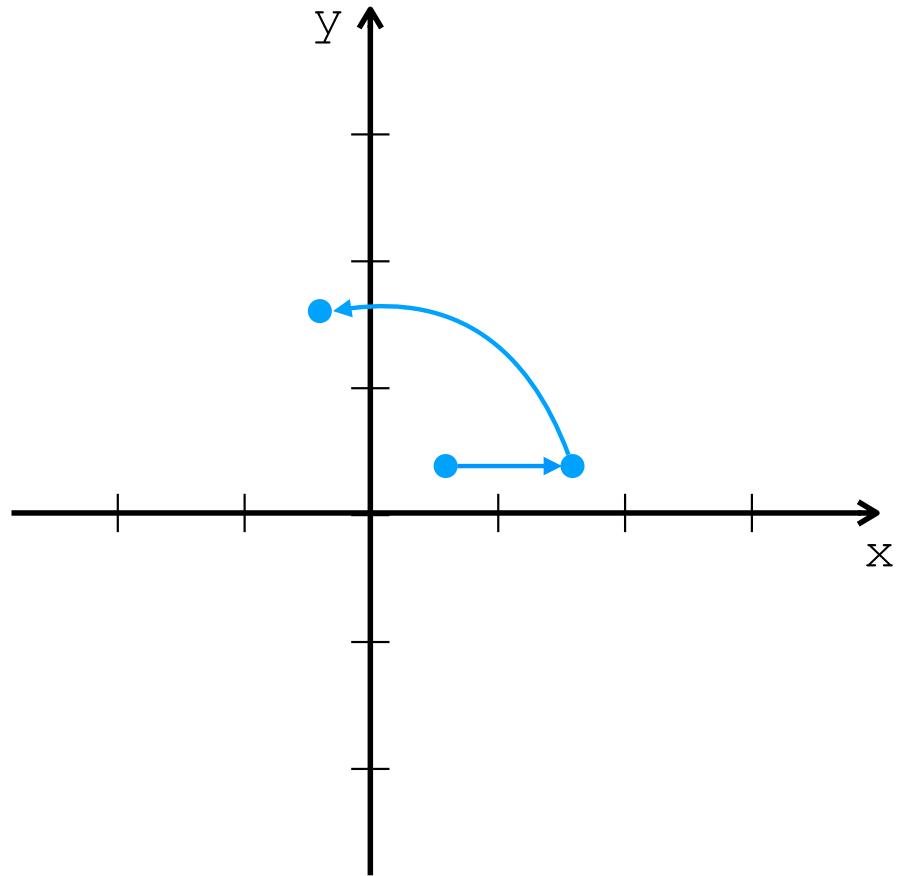
rotate by center in (0,0)
and angle 90°



Concrete semantics

sequential composition

```
translate(1, 0);  
rotate(0, 0, 90°)
```



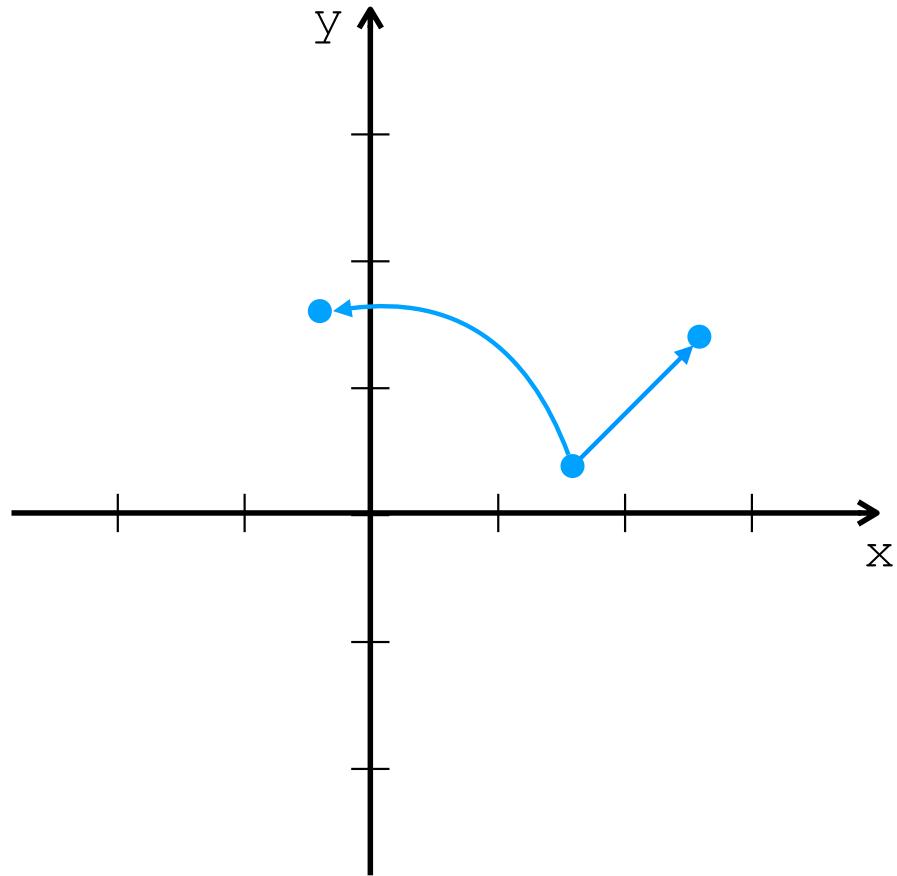
Concrete semantics

nondeterministic
choice

{ translate(1, 1) }

or

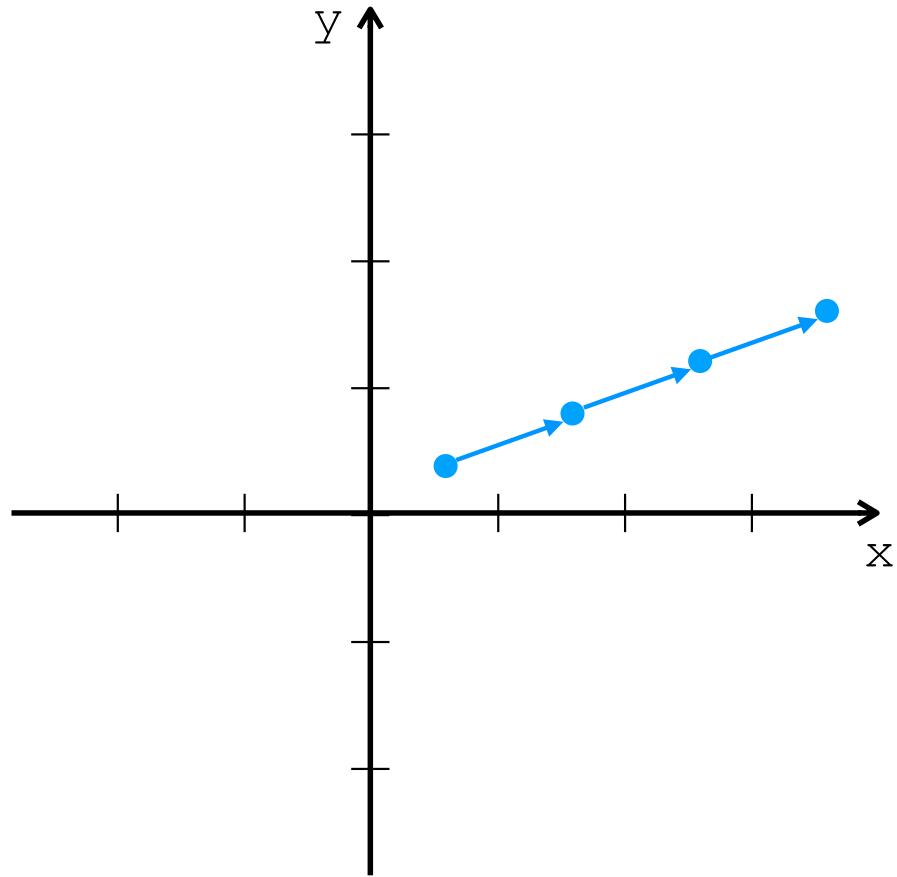
{ rotate(0, 0, 90°) }



Concrete semantics

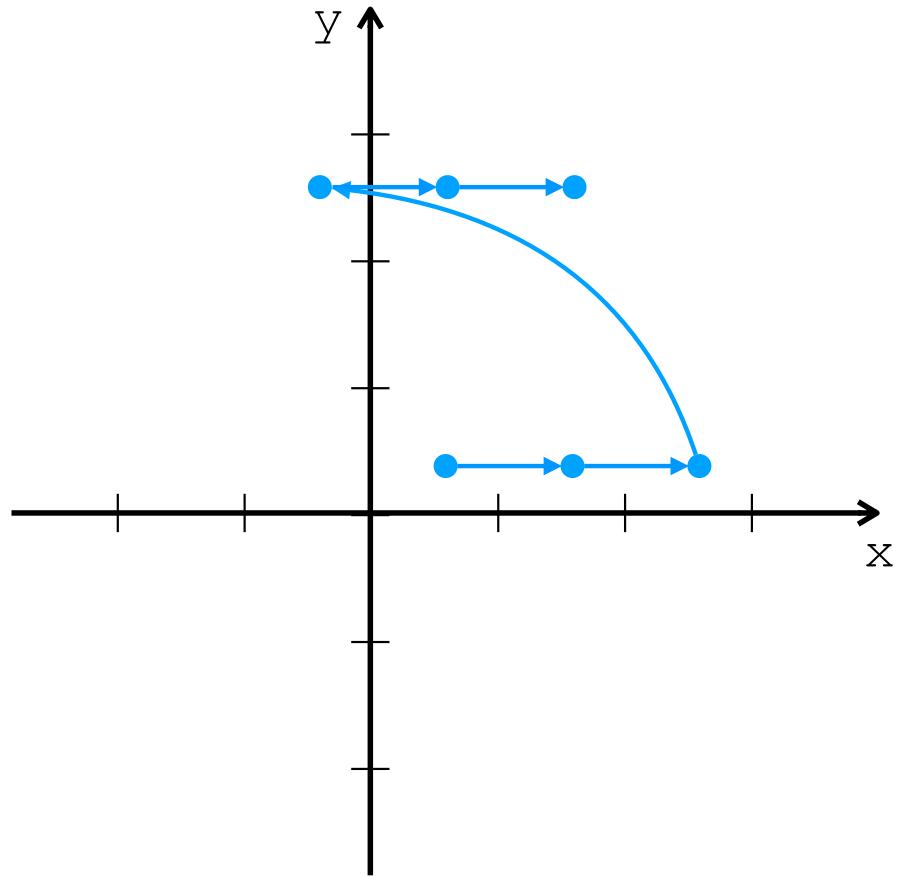
```
iter {  
    translate(1, 0.4)  
}
```

nondeterministic
iteration



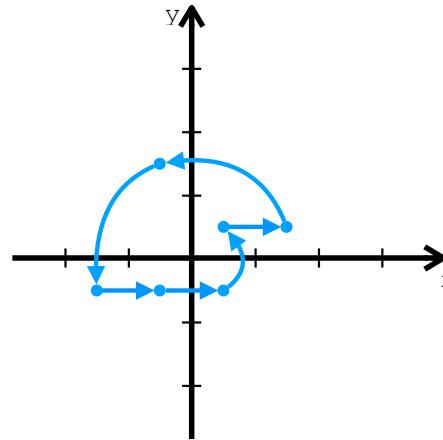
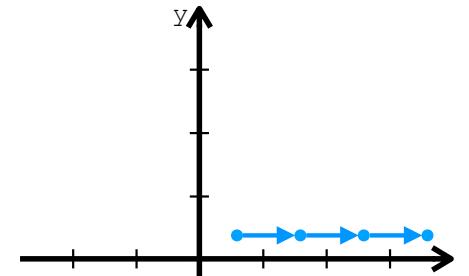
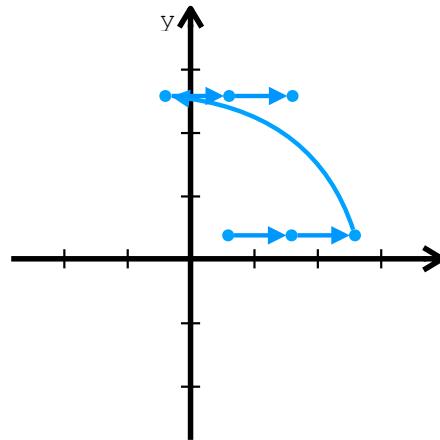
Program executions

```
init([0,1]x[0,1]); ←  
translate(1,0);  
iter {  
}  
    translate(1,0)  
} or {  
    rotate(0,0,90°)  
}  
}
```



Program executions

```
init([0,1]x[0,1]);  
translate(1,0);  
iter {  
}  
    translate(1,0)  
} or {  
    rotate(0,0,90°)  
}  
}
```



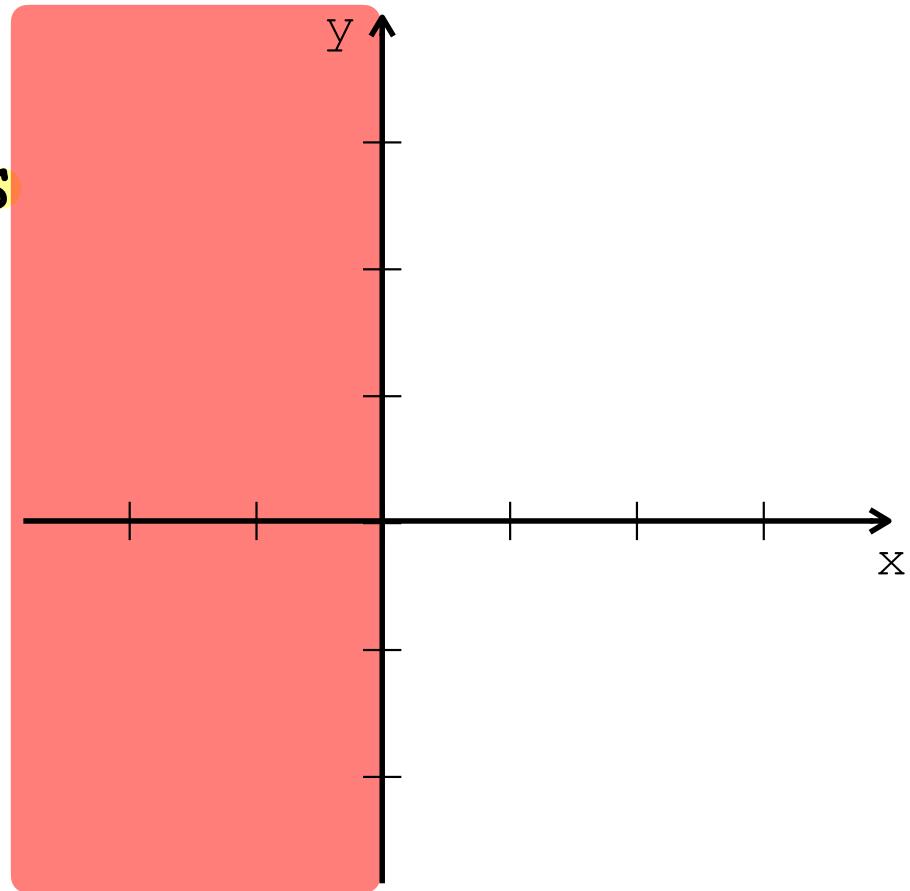
Analysis goal: safety

given a hypothetical error zone E

analyze the set of reachable points

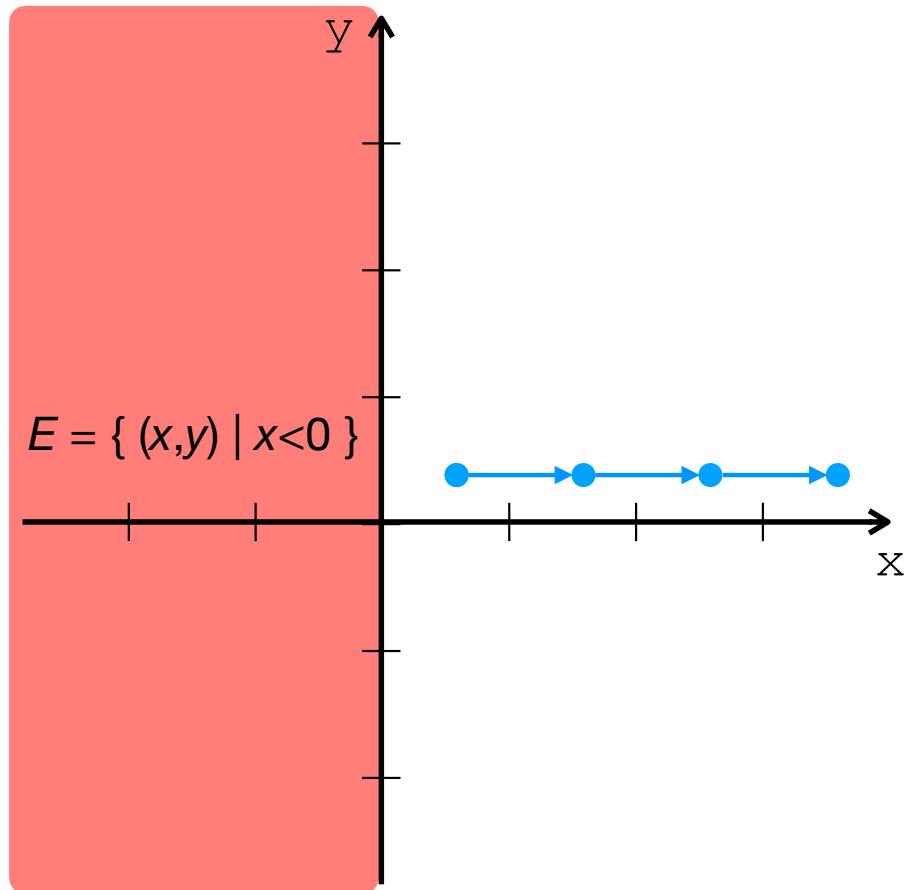
to check if they can intersect E

$$E = \{ (x,y) \mid x < 0 \}$$



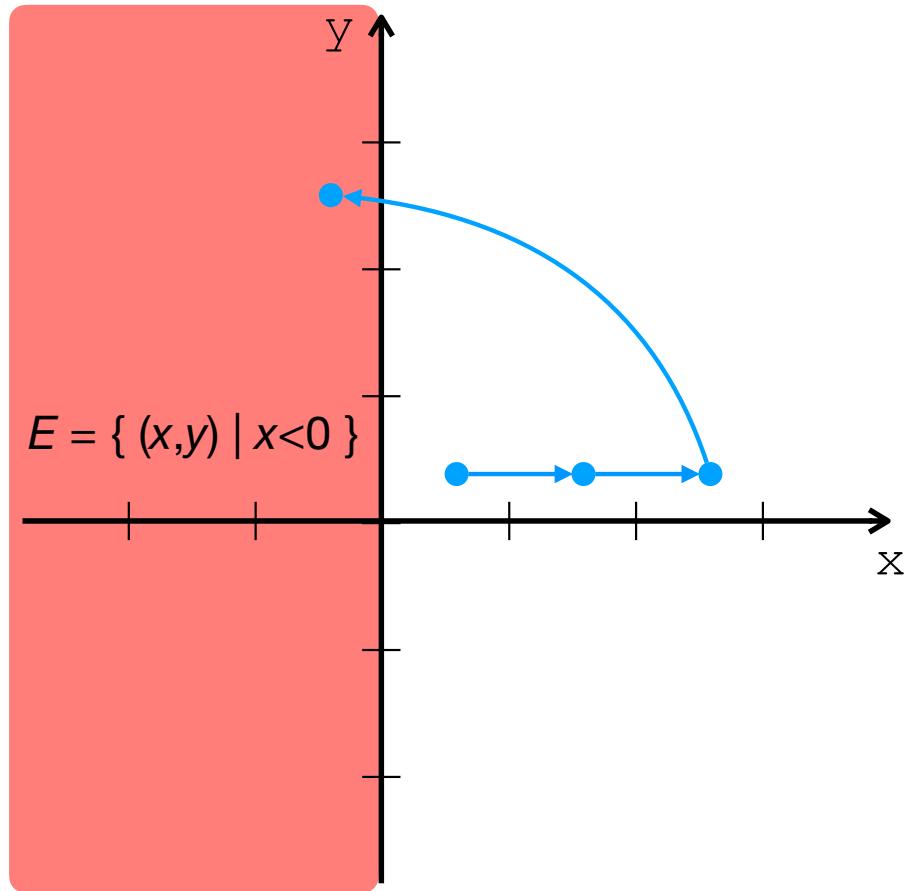
A correct execution

```
init([0,1]x[0,1]);  
translate(1,0);  
iter {  
    {  
        translate(1,0)  
    } or {  
        rotate(0,0,90°)  
    }  
}
```



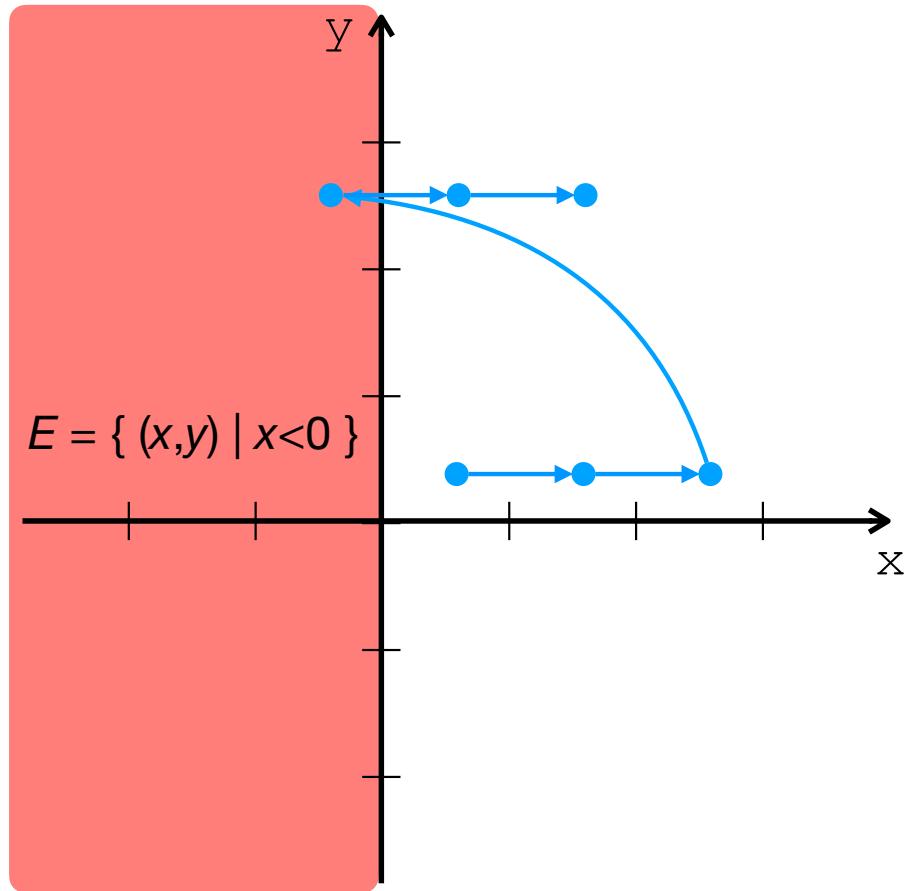
An incorrect execution

```
init([0,1]x[0,1]);  
translate(1,0);  
iter {  
    {  
        translate(1,0)  
    } or {  
        rotate(0,0,90°)  
    }  
}
```



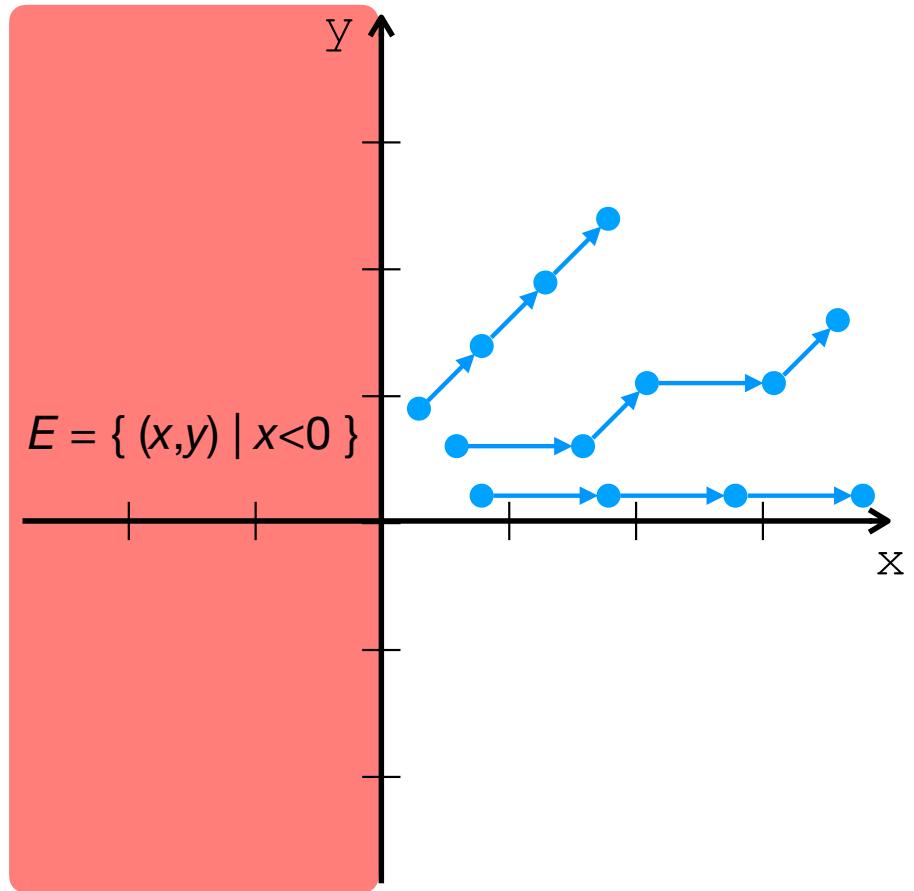
A correct execution?

```
init([0,1]x[0,1]);  
translate(1,0);  
iter {  
    translate(1,0)  
} or {  
    rotate(0,0,90°)  
}  
}
```



A safe program

```
init([0,1]x[0,1]);  
iter {  
    translate(1,0)  
} or {  
    translate(0.5,0.5)  
}  
}
```



Can static analysis prove $\neg E$?

How can we check safety for every program?

the set of possible initial states is infinite!

the length of executions is unbounded!

possibly infinitely many different choices!



Enumeration of all executions does not work!

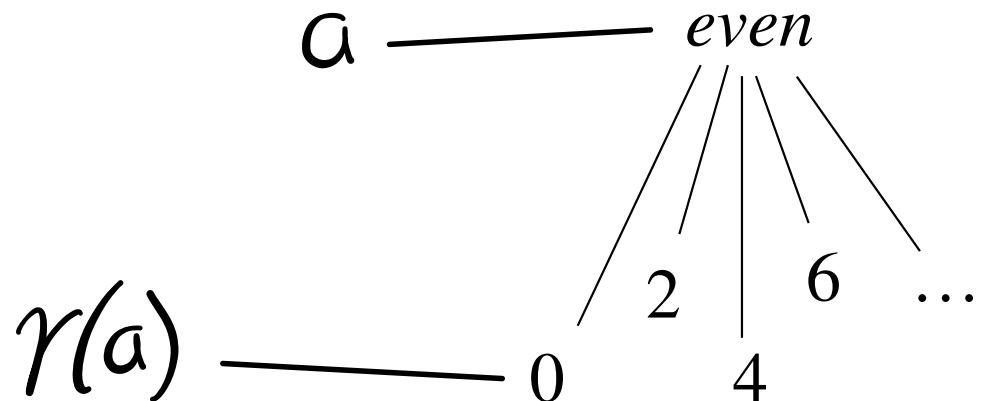
Abstraction

A set of logical properties of program states
called abstract elements

A set of abstract elements is called
an abstract domain

Concretization

The concretization $\gamma(a)$ of an abstract element a
is the set of program states that satisfy it

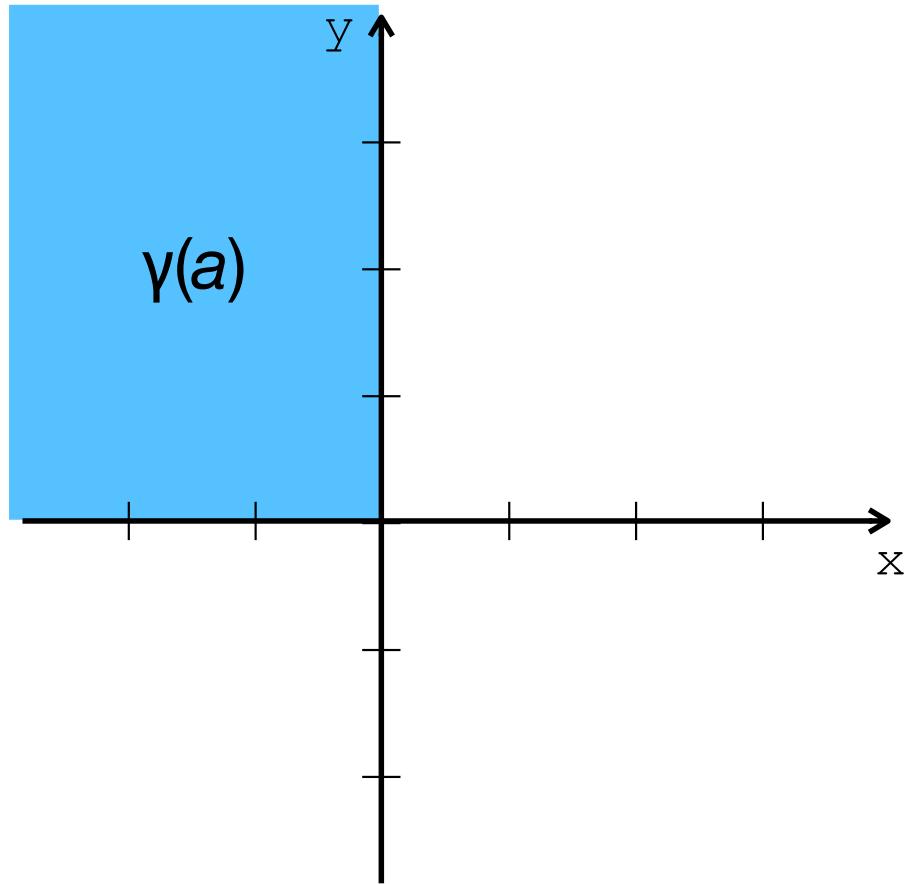


Signs abstraction

Describe a set of states by two pieces of information:
the possible values of the sign of x
and the possible values of the sign of y
(positive, negative, non-negative, etc.)

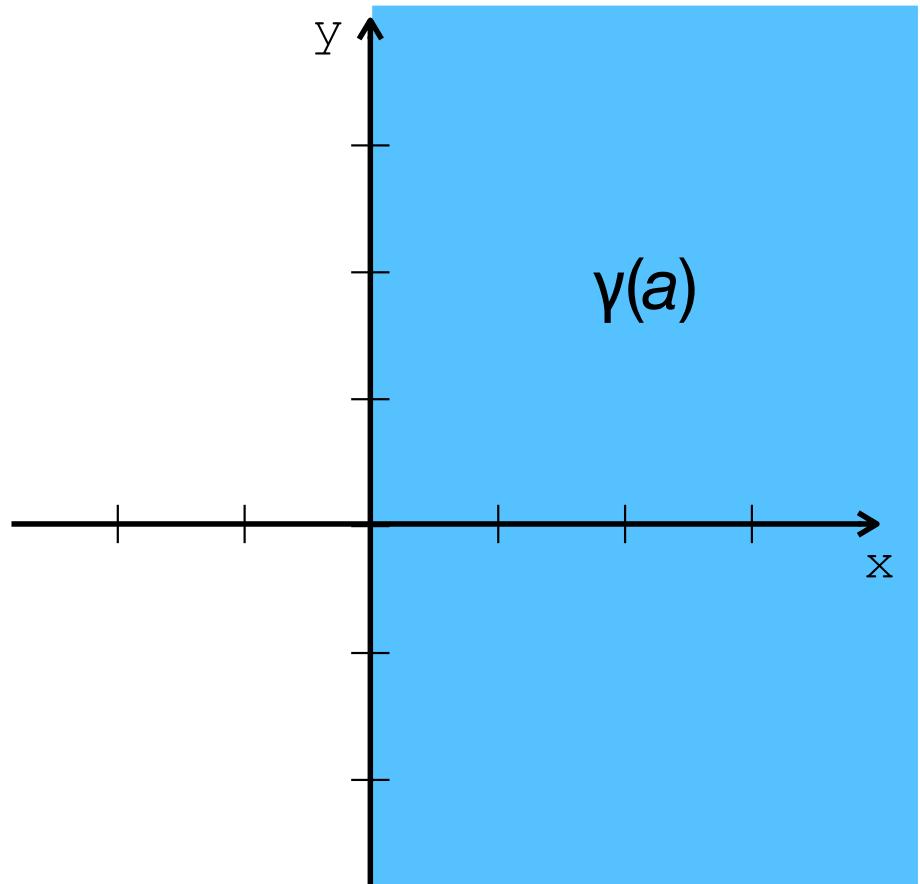
Signs abstraction

$$a = [x \leq 0, y \geq 0]$$



Signs abstraction

$$a = [x \geq 0]$$



Intervals abstraction

Conjunctions of different pieces of information:

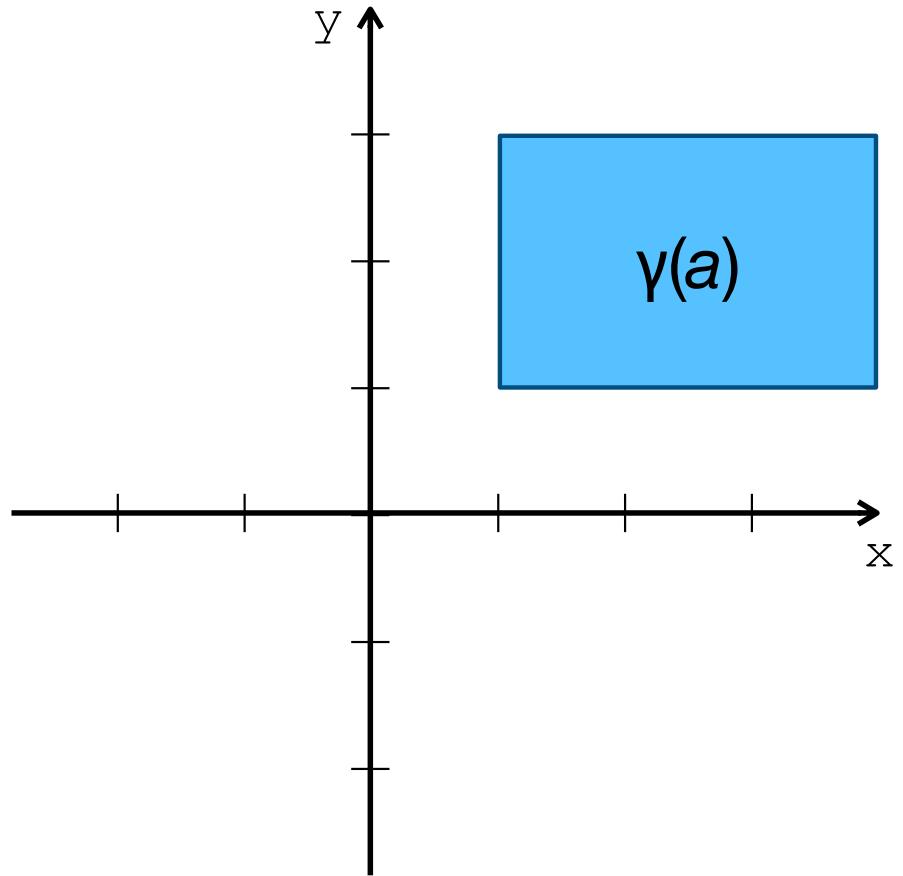
the range constraints of x
and the range constraints of y
(expressed as inequalities:

$$u \leq x \leq v, \quad n \leq y \leq m$$

we just omit infinite bound constraints)

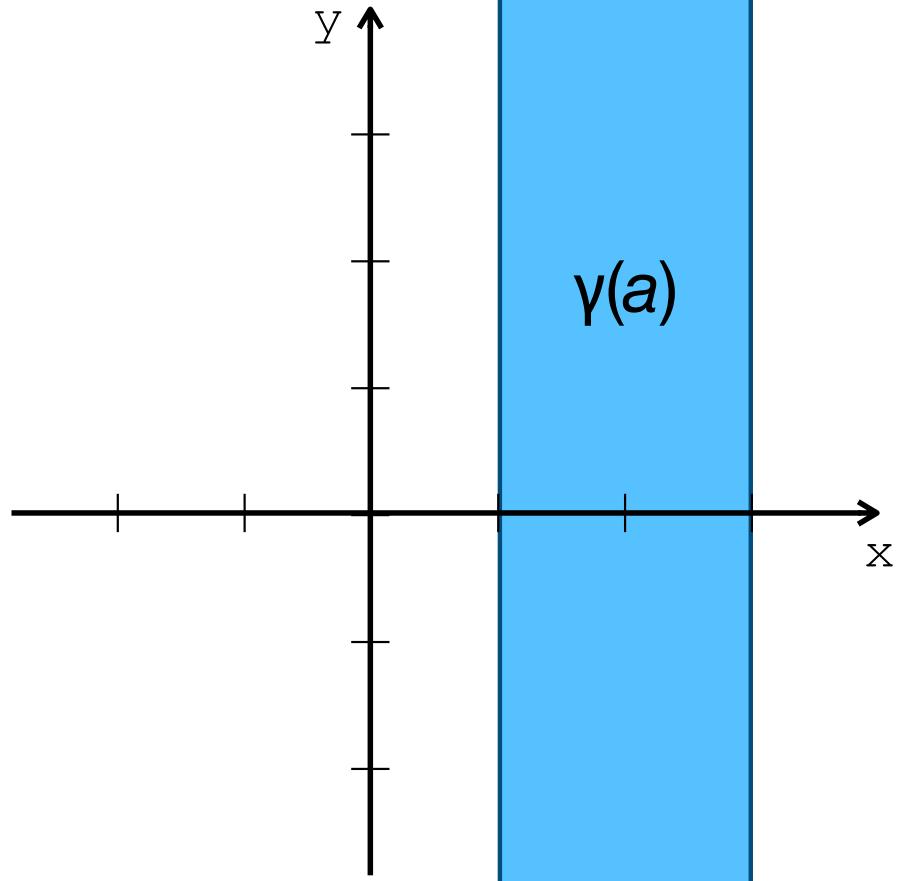
Intervals abstraction

$$a = [1 \leq x \leq 4, 1 \leq y \leq 3]$$



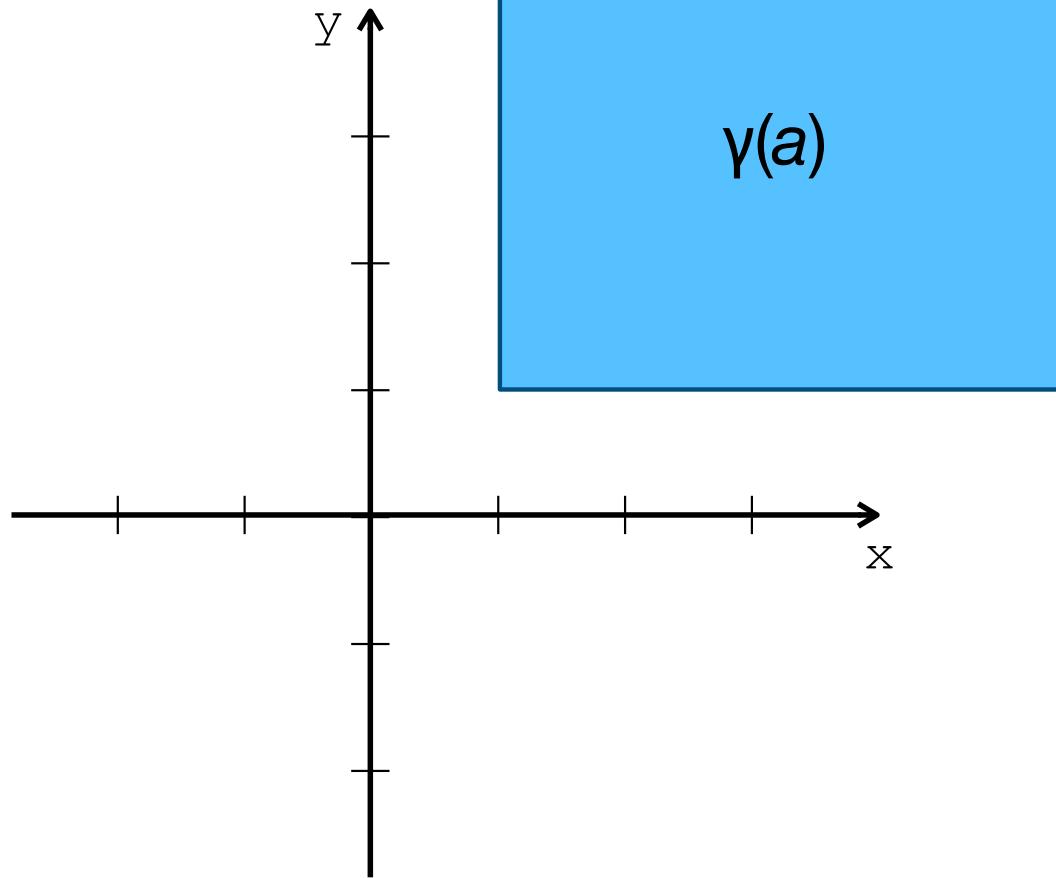
Intervals abstraction

$$a = [1 \leq x \leq 3]$$



Intervals abstraction

$$a = [1 \leq x , 1 \leq y]$$



Intervals abstraction

intervals are more expressive than signs:

any abstract element of Signs abstract domain

corresponds to an element of Intervals abstract domain

abstract elements can be represented by

at most two numerical constants per variable

[min,max] with $\text{min}, \text{max} \in \mathbb{Z} \cup \{-\infty, +\infty\}$

Best abstraction

Given a set of points (program states) S ,
can we find an abstract element a that
over-approximate S more closely than any other?

Best abstraction

An abstract element a is the best abstraction

of a set of points S iff

$$1) S \subseteq \gamma(a)$$

is an over-
approximation

$$2) S \subseteq \gamma(b) \text{ implies } b \text{ coarser than } a$$

$$\gamma(a) \subseteq \gamma(b)$$

any other over-
approximation of S
is coarser

Relational abstractions

Signs and Intervals are non-relational abstractions:
the range of x is not influenced by the range of y

Relational abstractions can capture
numerical constraints over both x and y
(e.g., $x \leq y$)

Convex Polyhedra abstraction

Abstract elements are conjunctions
of linear inequality constraints of the form

$$c_1x + c_2y \leq c$$

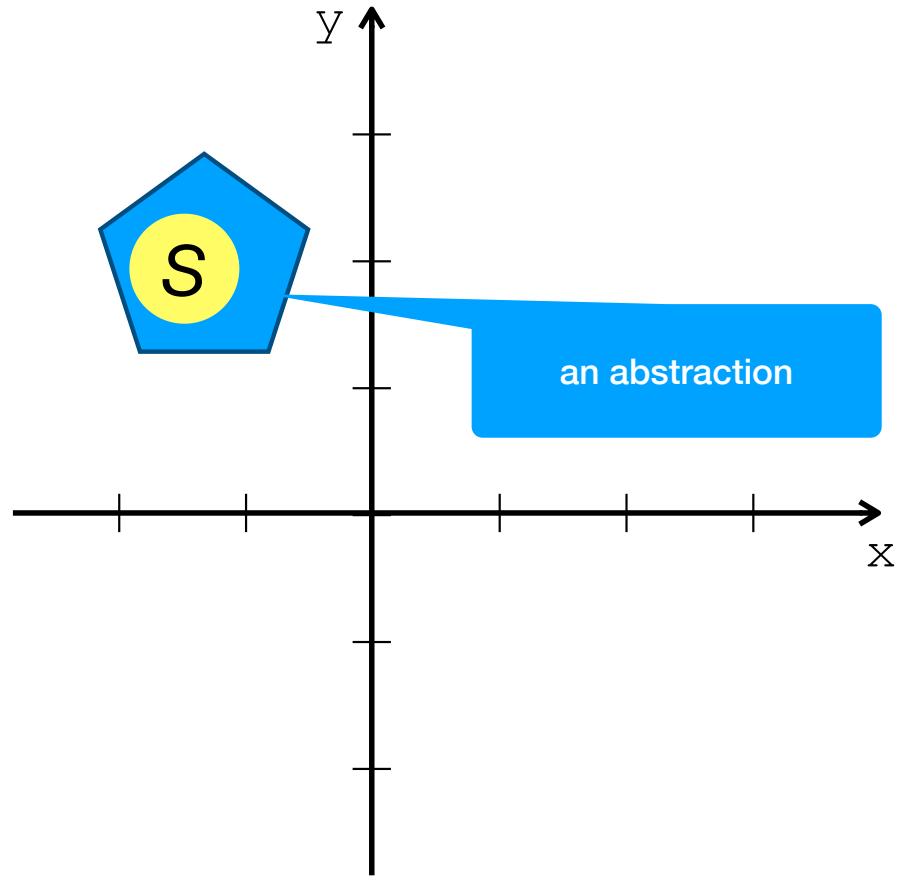
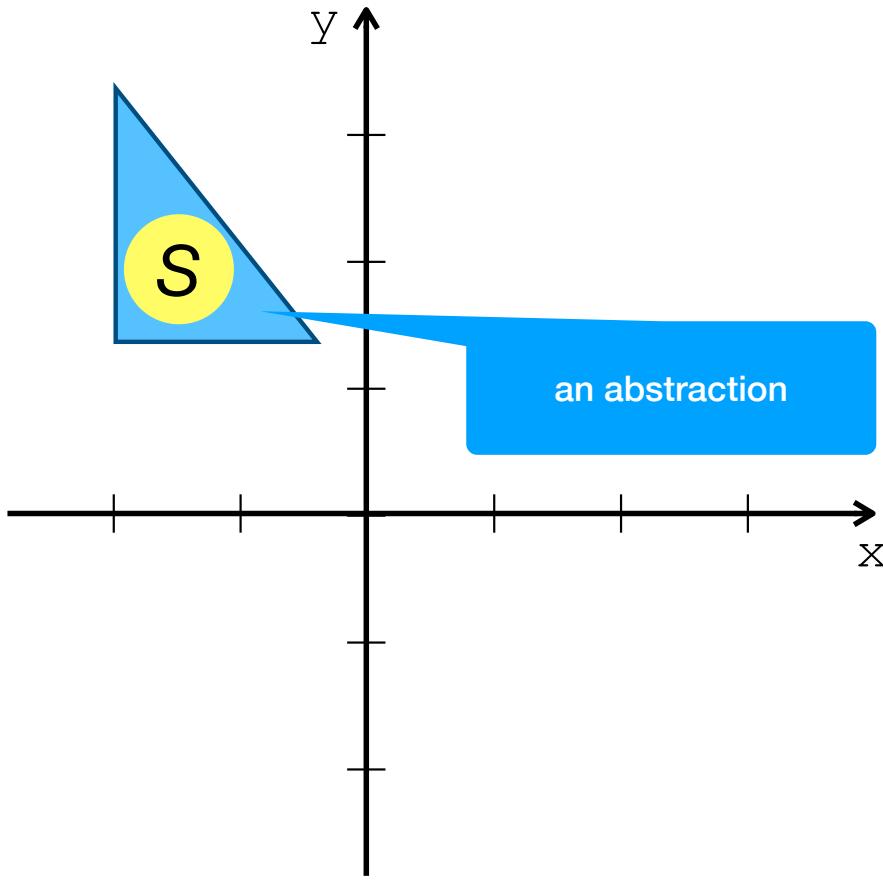
(but other representations are possible,
e.g., vertices and edges)

Convex Polyhedra abstraction

convex polyhedra are more expressive than intervals:
any abstract element of Intervals abstract domain
corresponds to an element of Convex Poly abstract domain

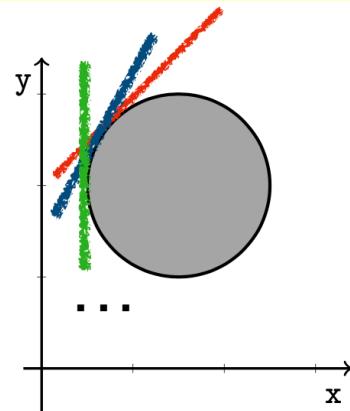
abstract elements can be represented by
(an unbounded number of) numerical constraints

Best abstraction?



Best: not always possible

Computing the best abstraction can be expensive
or sometimes not even possible



In practice we can use abstractions
as precise as possible but maybe not the best
(still best abstractions can exist for some sets of states)

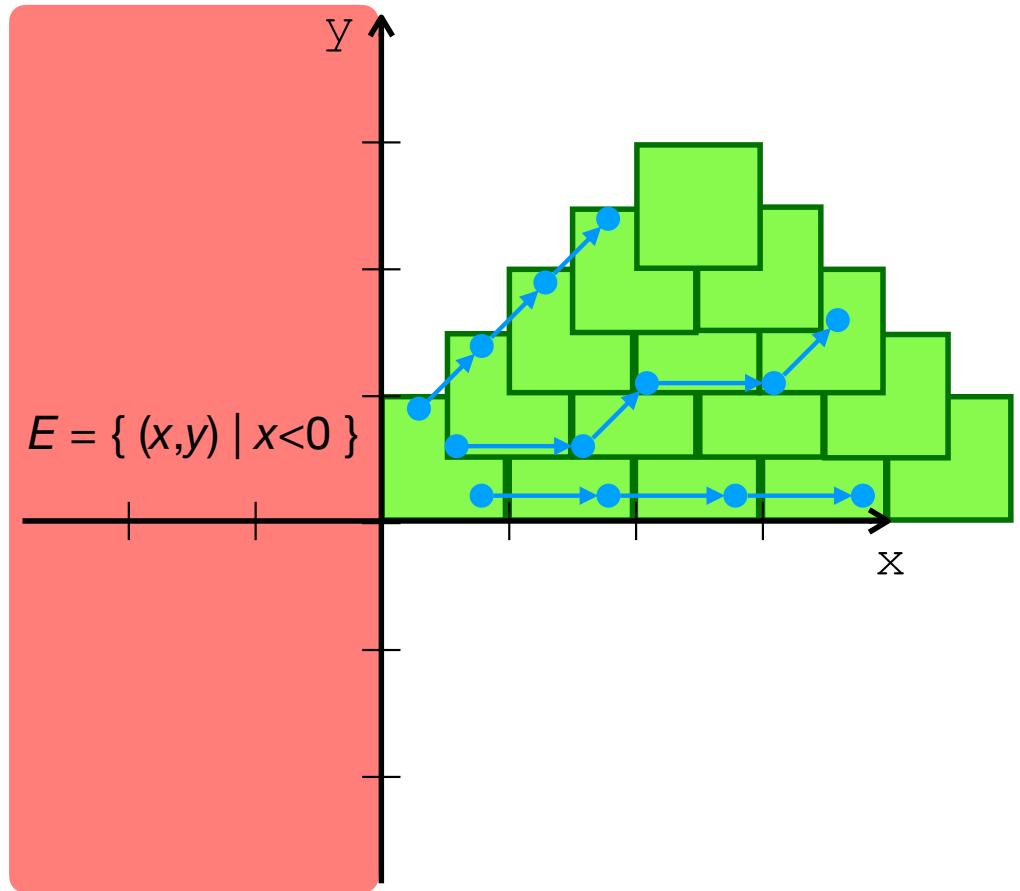
Convex Polyhedra abstraction

If the region is already a convex polyhedron,
take it as exact approximation

Otherwise, use the interval abstraction to find
the best enclosing box and take it as abstraction
(any interval element is also a convex polyhedron)

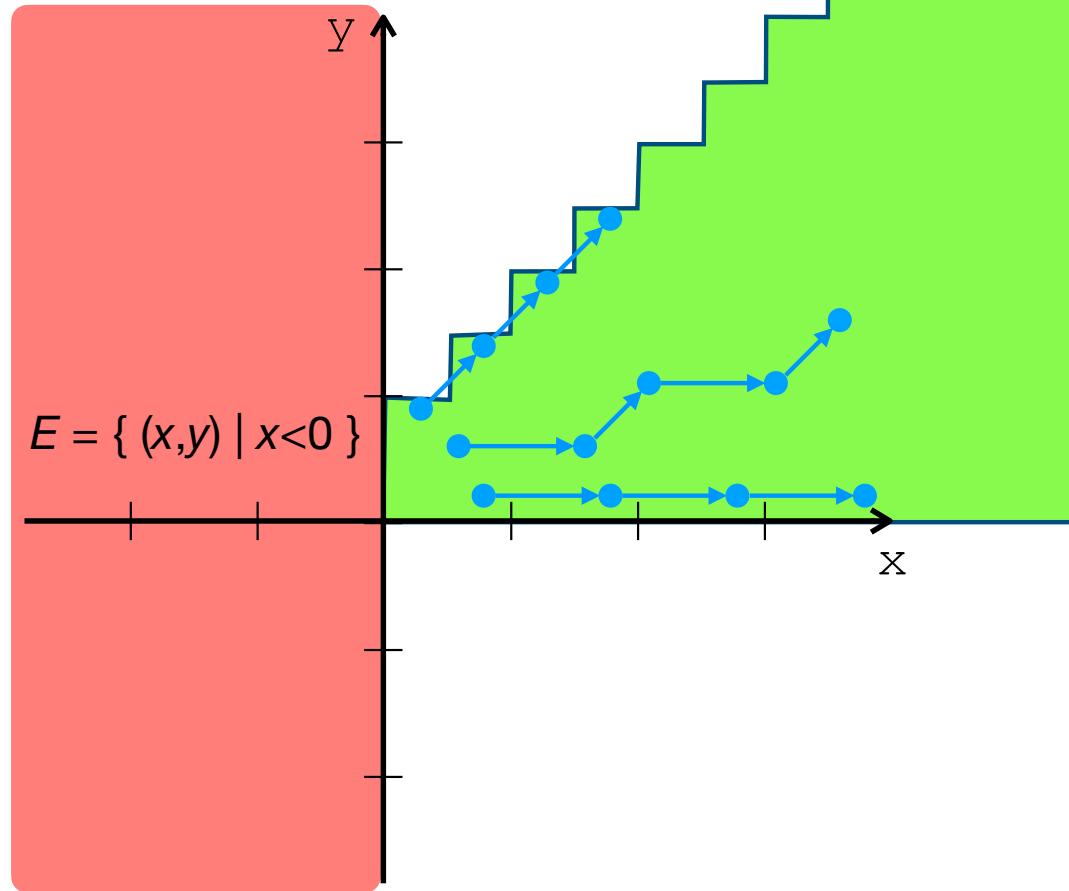
Reachable states

```
init([0,1]x[0,1]);  
iter {  
    translate(1,0)  
} or {  
    translate(0.5,0.5)  
}  
}
```



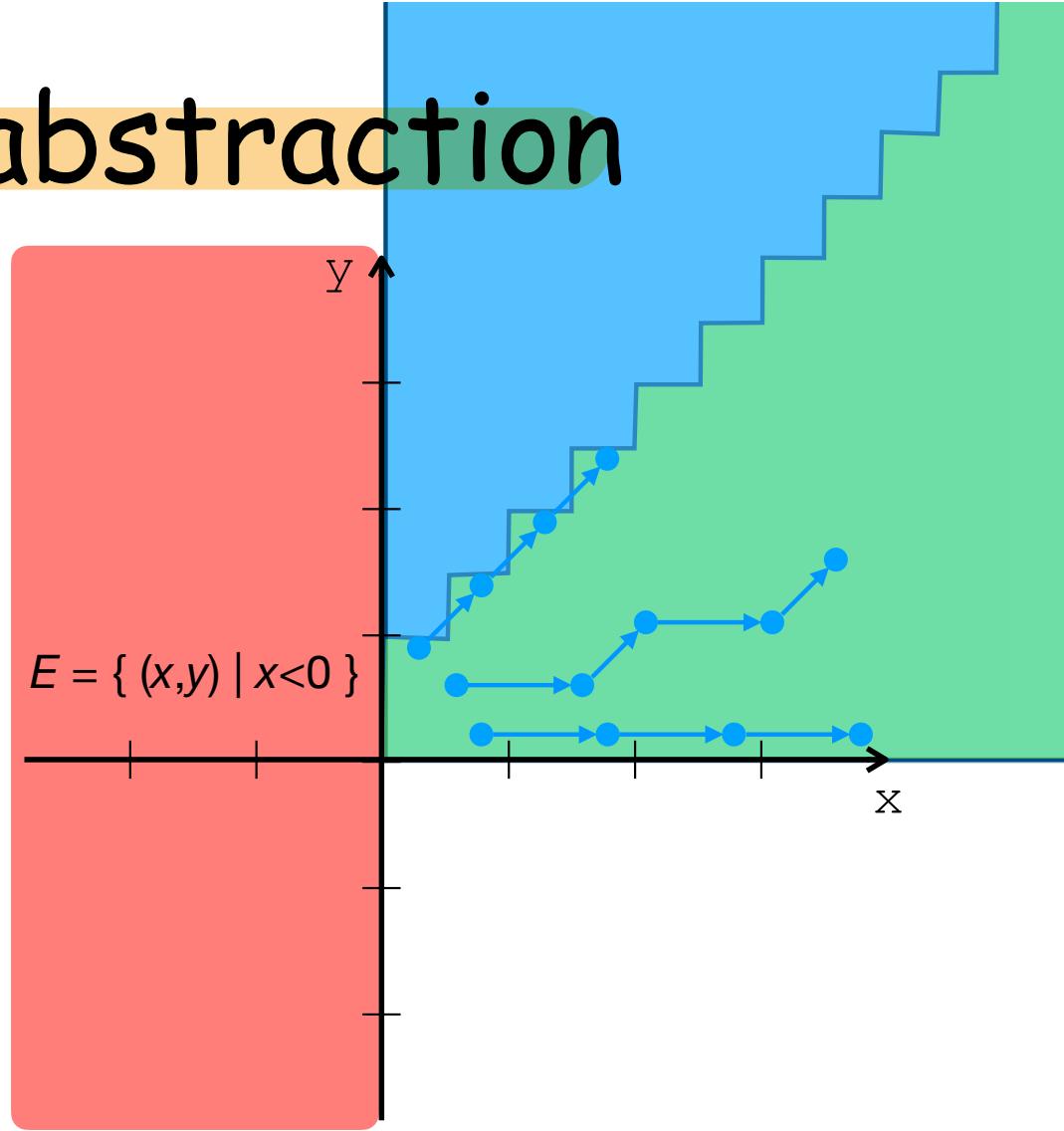
Reachable states

```
init([0,1]x[0,1]);  
iter {  
    translate(1,0)  
} or {  
    translate(0.5,0.5)  
}  
}
```



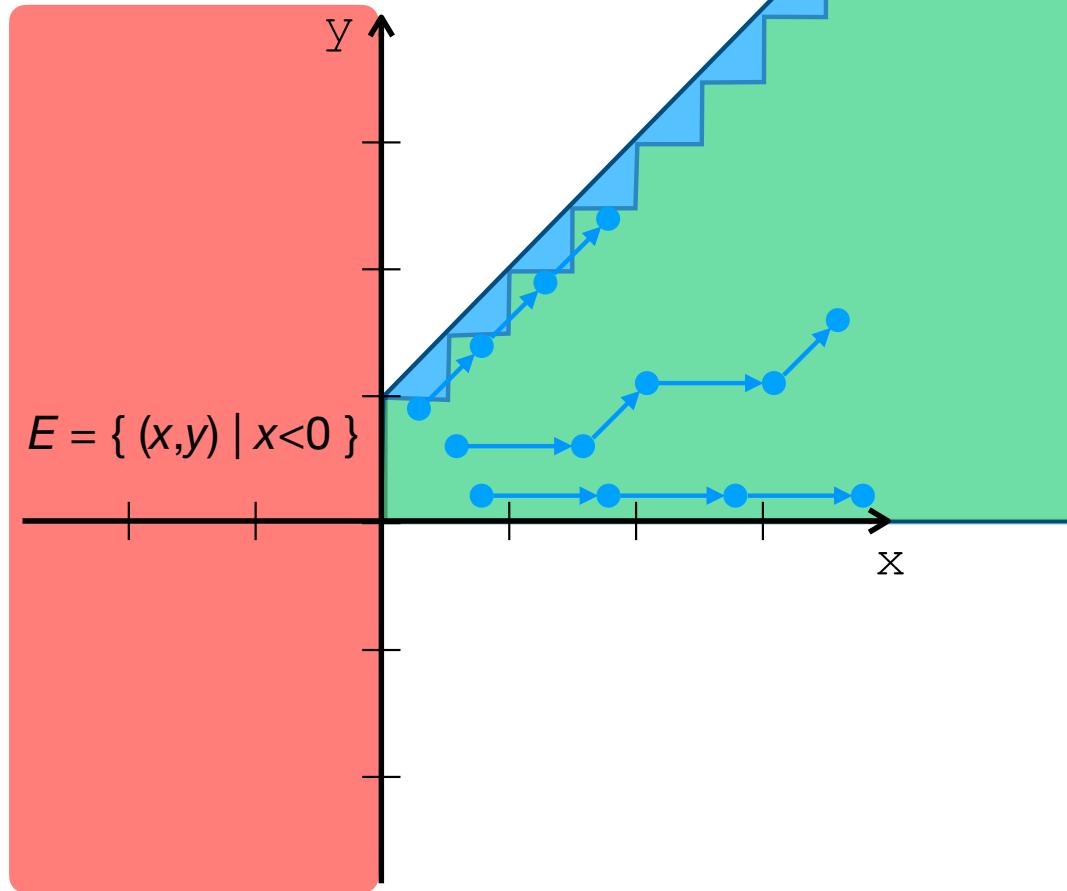
Intervals abstraction

```
init([0,1]x[0,1]);  
iter {  
    translate(1,0)  
} or {  
    translate(0.5,0.5)  
}  
}
```



Convex Polyhedra abstraction

```
init([0,1]x[0,1]);  
iter {  
    translate(1,0)  
} or {  
    translate(0.5,0.5)  
}  
}
```



Abstract semantics computation

Compositionality principle:

compute a sound analysis of a program
by computing sound abstract semantics
of program's components

Analysis function

analysis function:

input: a program p , an abstract element a (pre-state)

output: an abstract element a' (post-state)

$$\begin{array}{c} \text{analysis}(p, a) \\ = \\ [[p]]^{\#} a \end{array}$$

$\llbracket \cdot \rrbracket^{\#}$ ABSTRACT SEM.
a ABS. ELEMENT

Sound analysis

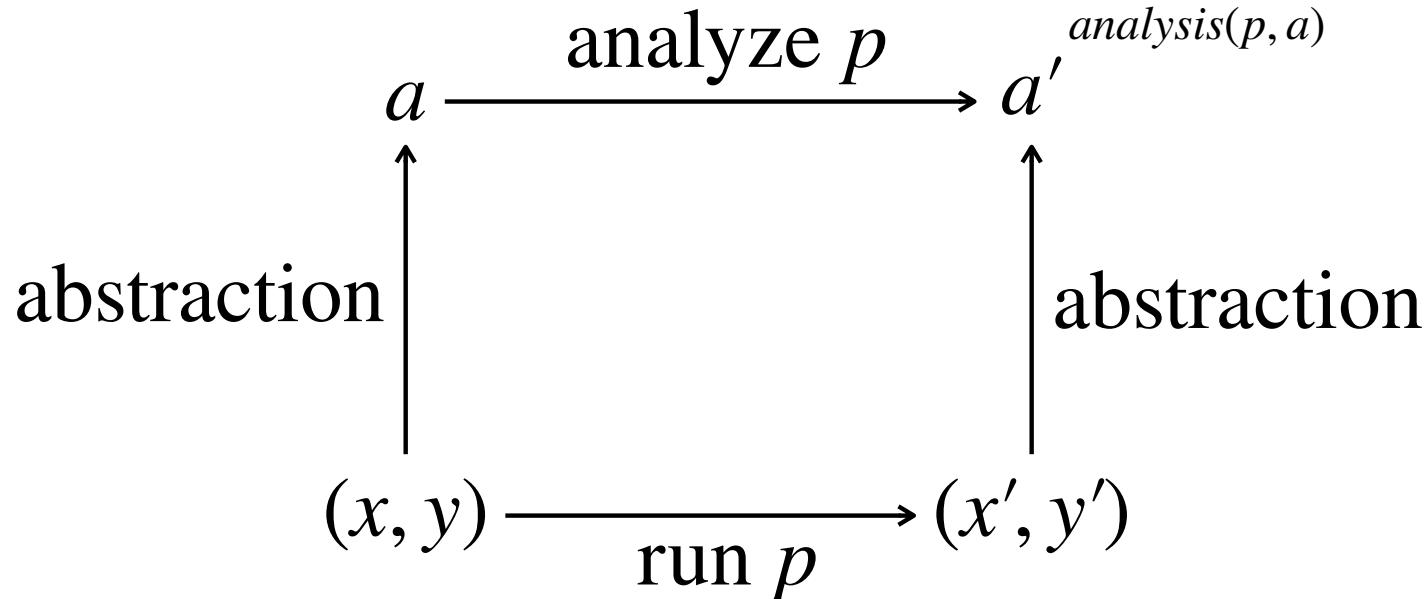
a sound *analysis* function must capture the real executions:

if p moves a point (x, y) to (x', y')

then for any abstract element a s.t. $(x, y) \in \gamma(a)$

it must be the case that $(x', y') \in \gamma(\underbrace{\text{analysis}(p, a)}_{\|p\|^{\#}a})$

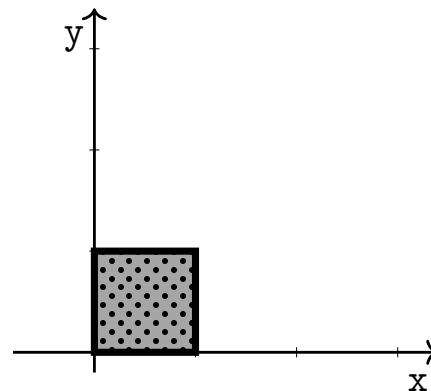
Sound analysis



Init

$\text{analysis}(\text{init}(\text{Region})) = \text{an abstraction of } \text{Region}$

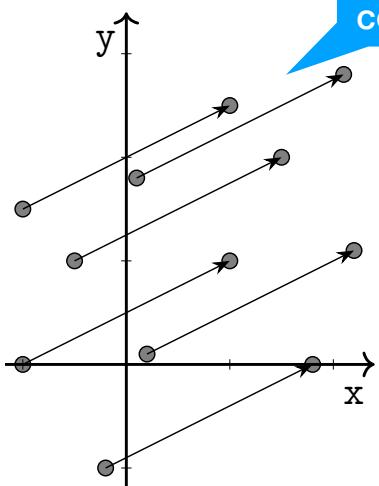
init($[0, 1] \times [0, 1]$)



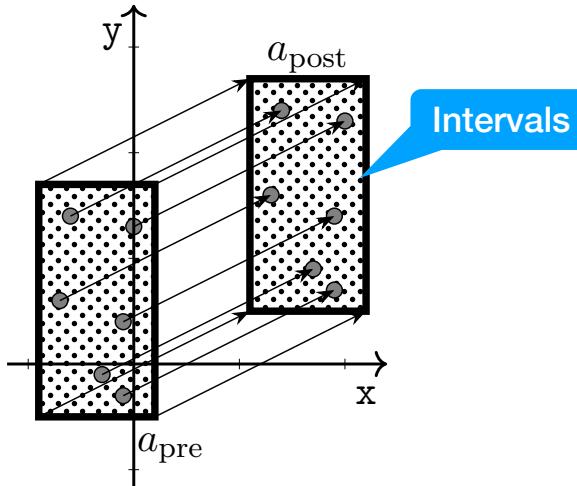
Translate

analysis(translate(u, v), a) =

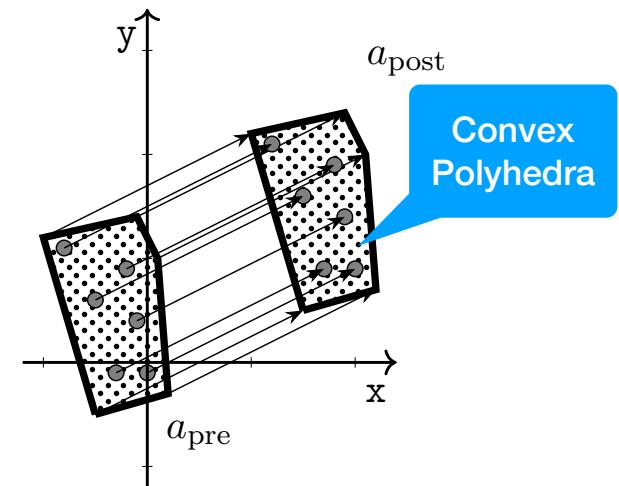
an abstraction that contains the translation of $\gamma(a)$



concrete



Intervals

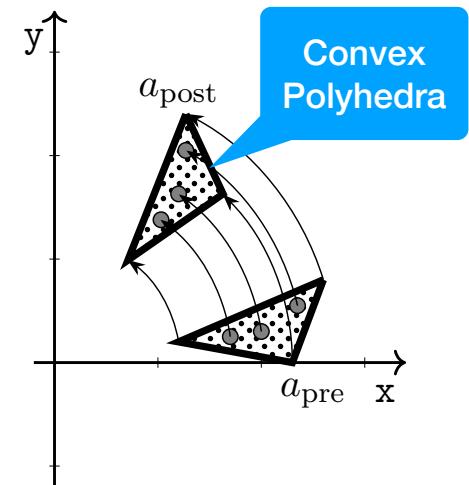
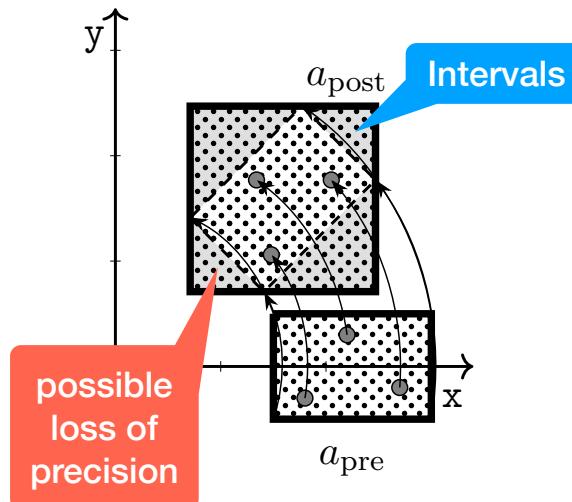
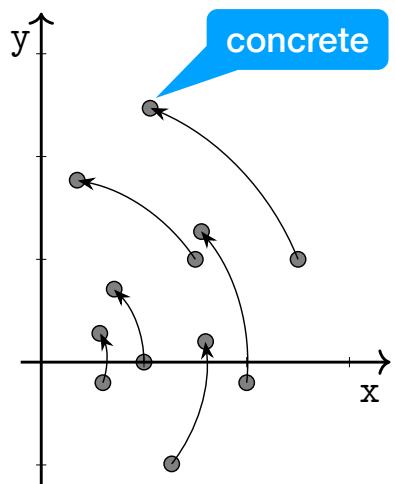


Convex
Polyhedra

Rotate

$\text{analysis}(\text{rotate}(u, v, \theta), a) =$

an abstraction that contains the rotation of $\gamma(a)$



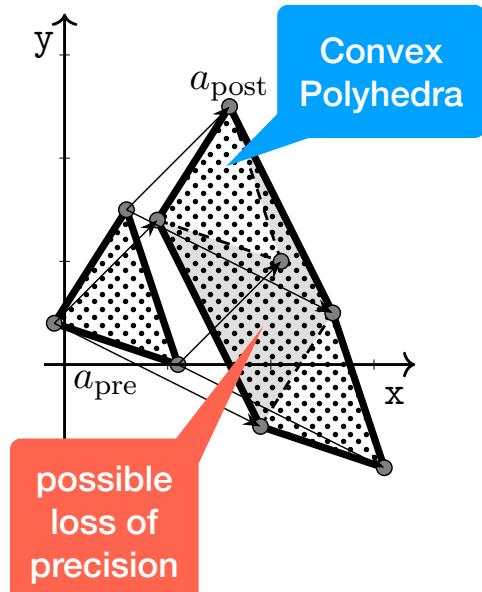
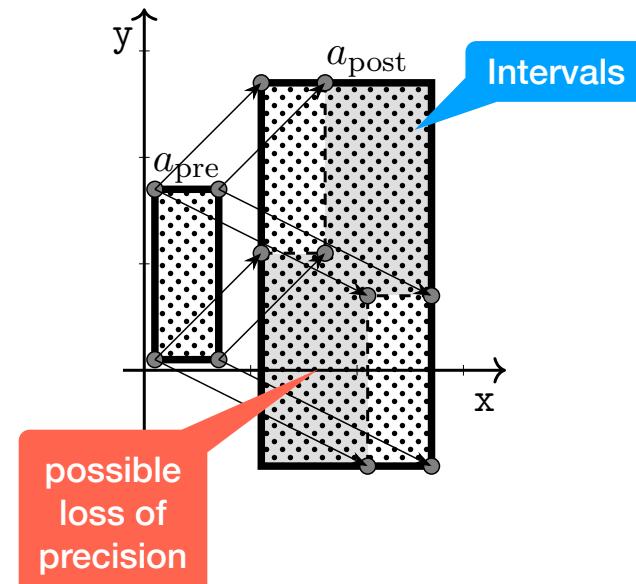
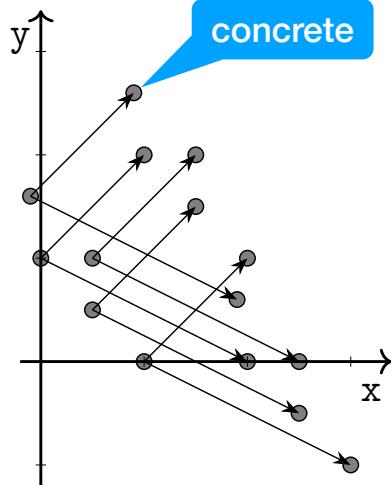
Sequence

$$\text{analysis}(p_0; p_1, a) = \text{analysis}(p_1, \text{analysis}(p_0, a))$$

Choice

over-approximates
two abstract
elements

$\text{analysis}(\{p_0\} \text{ or } \{p_1\}, a) =$
 $\text{union}(\text{analysis}(p_0, a), \text{analysis}(p_1, a))$



Iteration

```
iter {  
    p  
}
```

≈

```
{ }  
or { p }  
or { p;p }  
or { p;p;p }  
or { p;p;p;p }  
or { p;p;p;p;p }
```

...

Iterations

p0: { }

p1: { } or { p }

p2: { } or { p } or {p;p}

p3: { } or { p } or {p;p} or {p;p;p}

.....

Iterations

$$p_0 \approx \{ \ }$$

$$p_{k+1} \approx \{ p_k \} \text{ or } \{ p_k; p \}$$

analysis(p_{k+1}, a) =

union(*analysis*(p_k, a), *analysis*($p, \text{analysis}(p_k, a)$))

idea: iterate the following assignment until it stabilizes

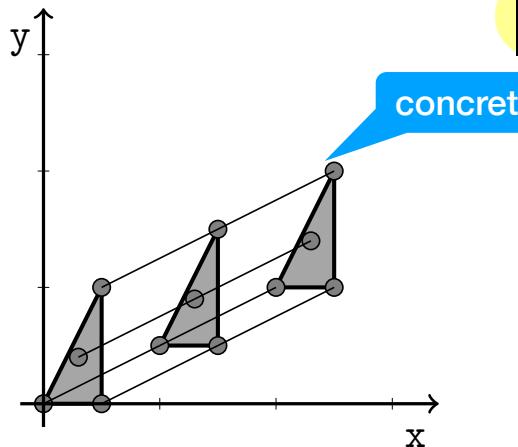
$R \leftarrow \text{union}(R, \text{analysis}(p, R))$

Iterations

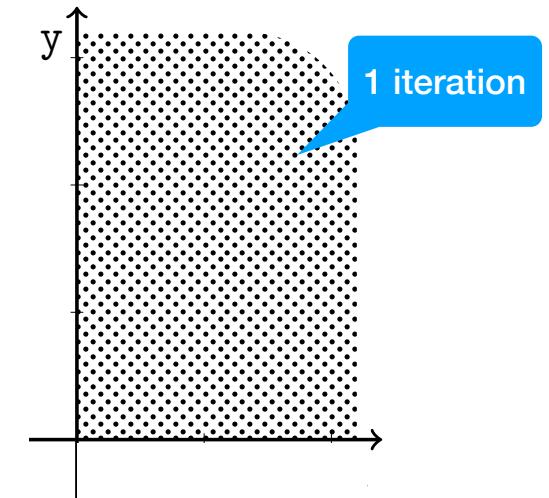
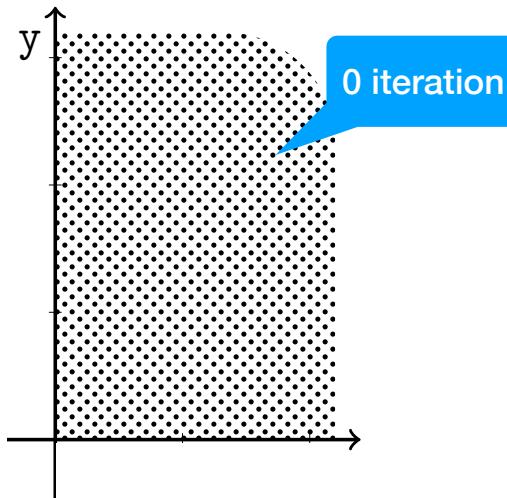
```

analysis(iter{p}, a) =   R ← a
                        do
                            T ← R
                            R ← union(R, analysis(p, R))
                            while R notin T
                        return T
    
```

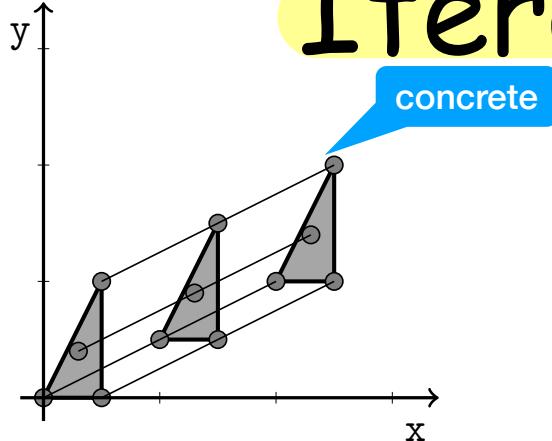
Iteration in Signs



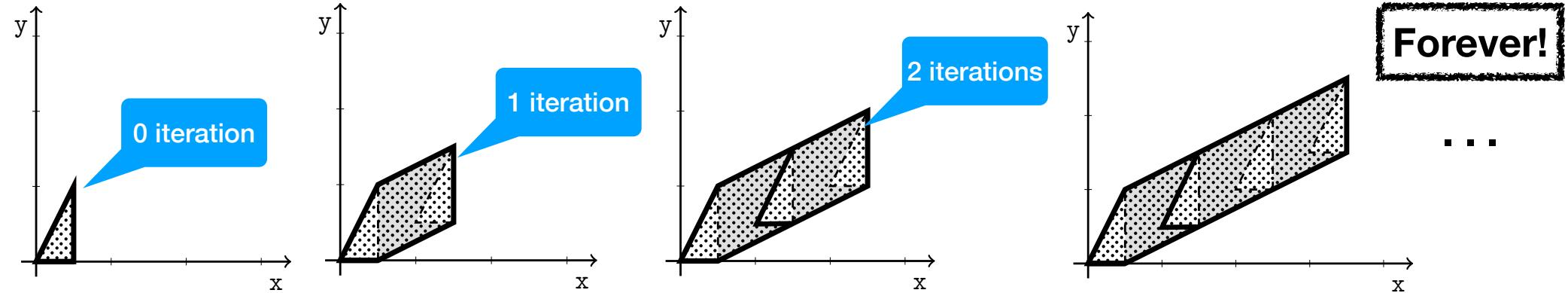
```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```



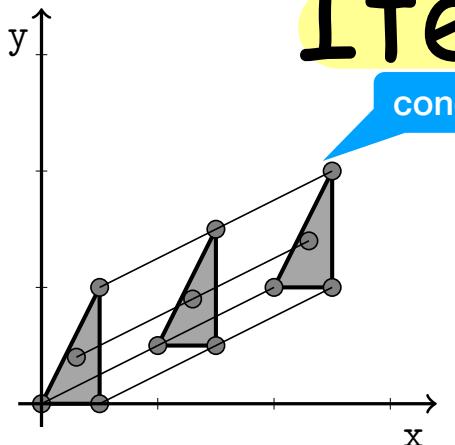
Iteration in Convex Poly



```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```

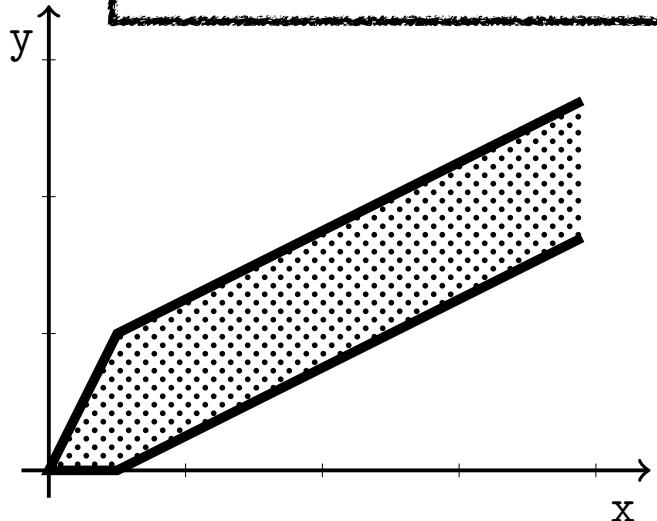


Iteration in Convex Poly



```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```

We want to converge to this state in finite time!



Widening

To enforce termination of the analysis
we need to enforce convergence of iterations
(not necessary if finitely many abstract elements are
traversed)

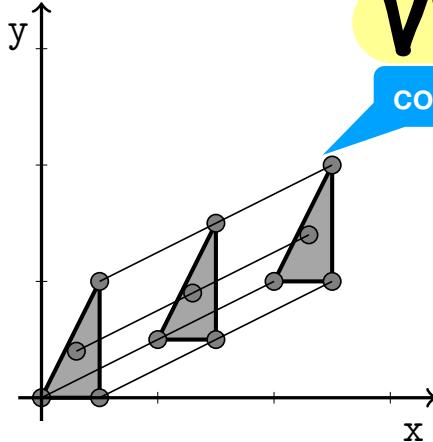
widening amounts to over-approximate
the union of two abstract elements and
further loose precision to gain convergence

Widening Convex Poly

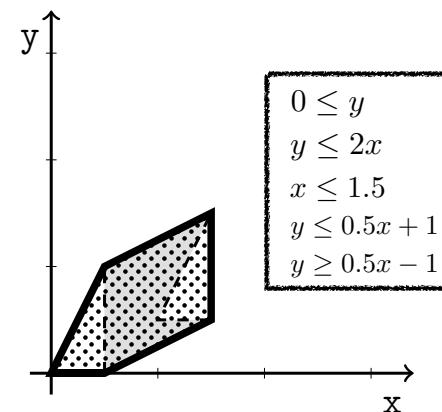
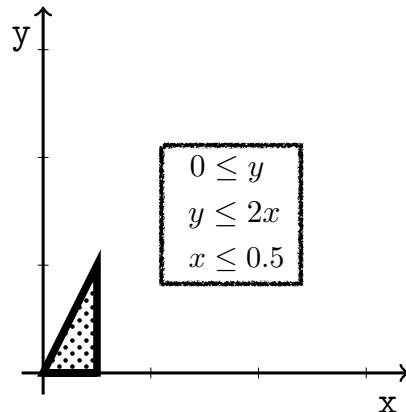
Abstract element = finitely many inequalities

Make sure the number of inequalities
is decreased at each iteration!

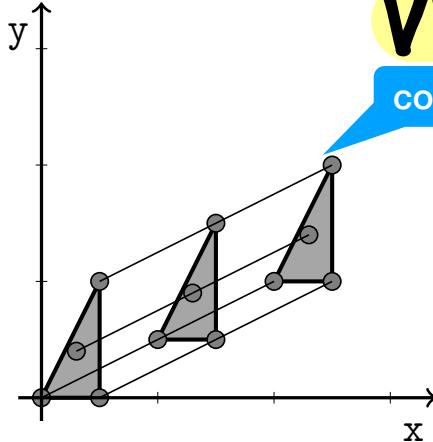
Widening Convex Poly



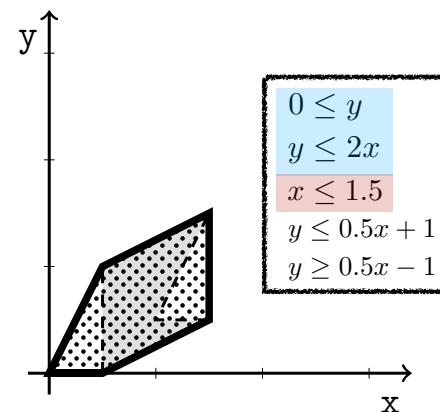
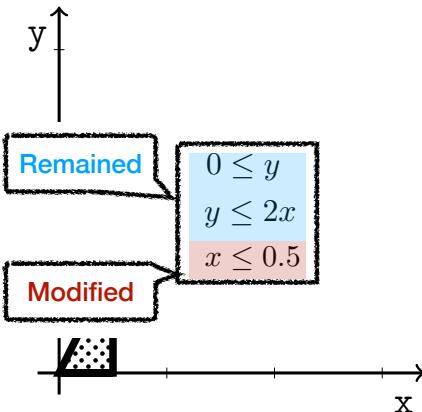
```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```



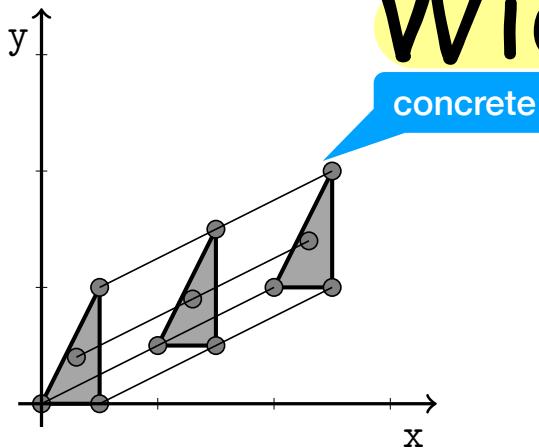
Widening Convex Poly



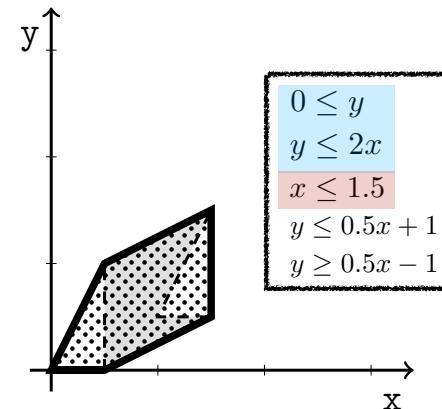
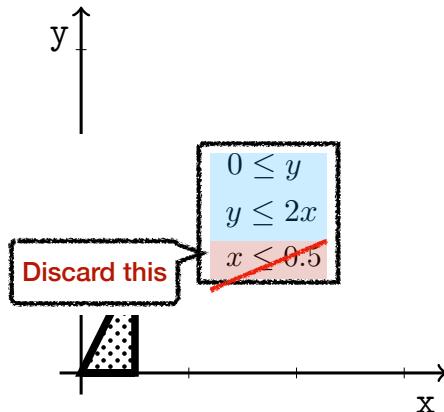
```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```



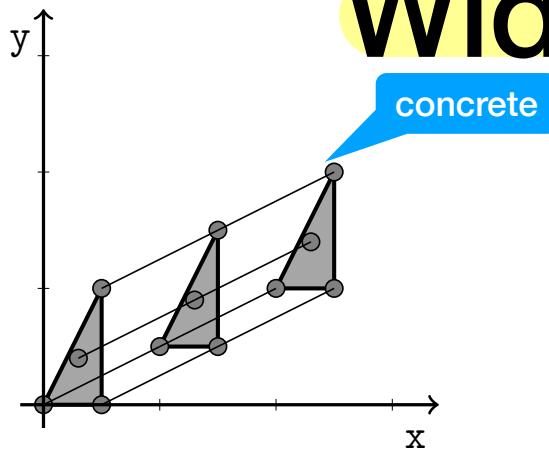
Widening Convex Poly



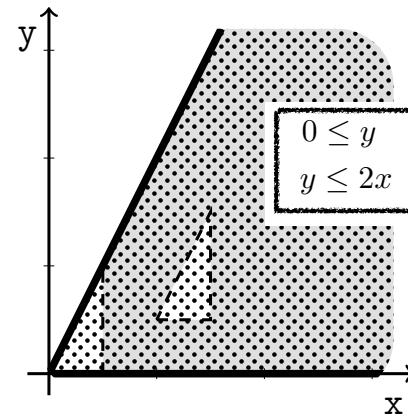
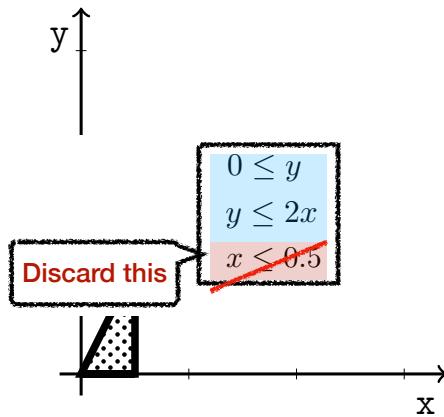
```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```



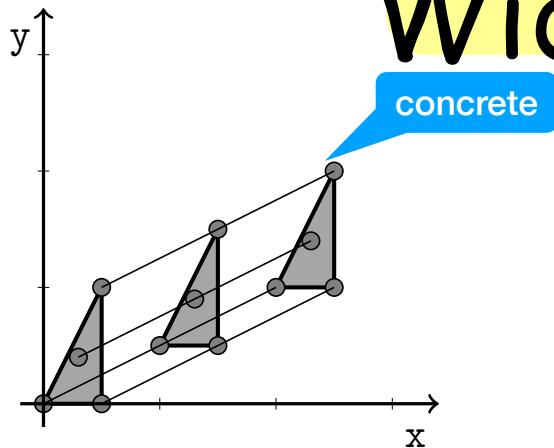
Widening Convex Poly



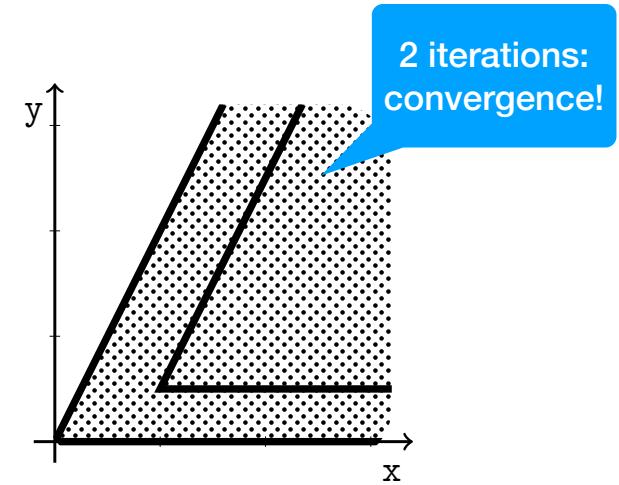
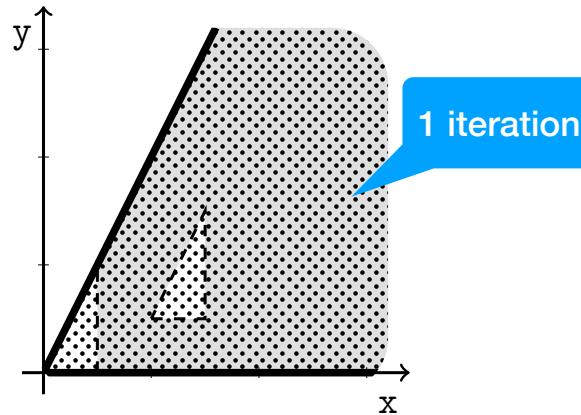
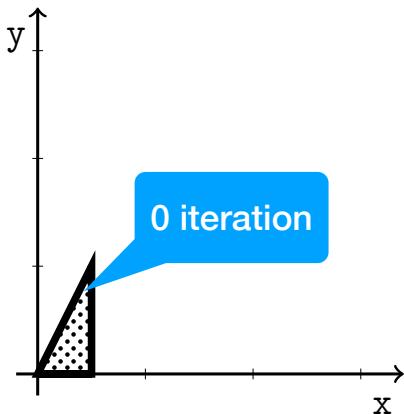
```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```



Widening Convex Poly



```
init({(x,y) | x ≤ 0.5 ∧ 0 ≤ y ≤ 2x}) ;  
iter { translate(1, 0.5) }
```



Iteration without widening

```
analysis(iter{p}, a) =   R ← a
                           do
                               T ← R
                               R ← union(R, analysis(p, R))
                           while R notin T
                           return T
```

Iteration with widening

```
analysis(iter{p}, a) =   R ← a
                           do
                               T ← R
                               R ← widen(R, analysis(p, R))
                           while R notin T
                           return T
```

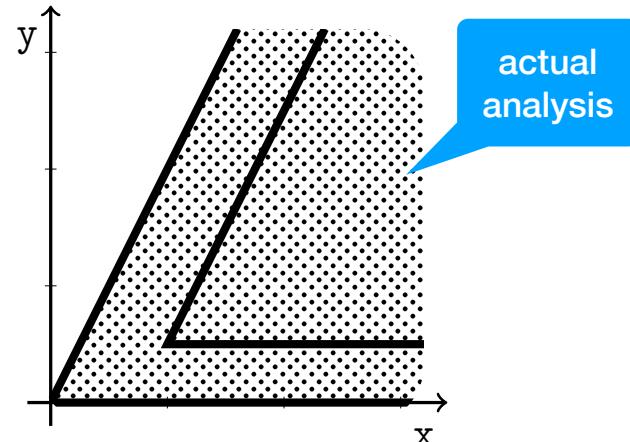
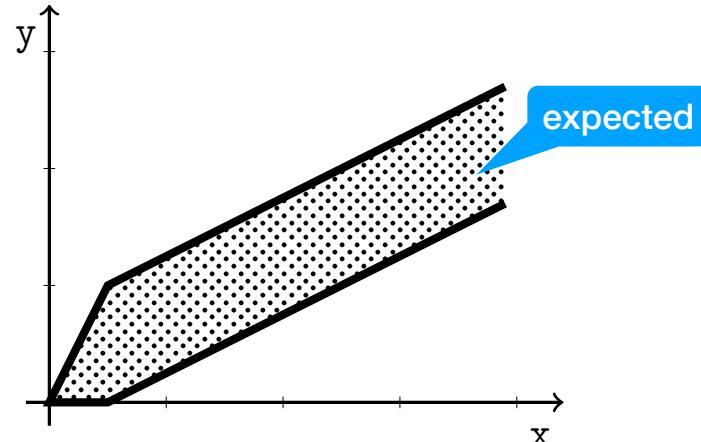
Imprecision due to widening

Widening:

over-approximate UNION

enforces convergence

may cause significant loss of precision



Analysis, at a glance

<code>analysis(init(\mathfrak{R}), a)</code>	$=$	best abstraction of the region \mathfrak{R}
<code>analysis(translation(u, v), a)</code>	$=$	{ return an abstract state that contains the translation of a
<code>analysis(rotation(u, v, θ), a)</code>	$=$	{ return an abstract state that contains the rotation of a
<code>analysis({p₀} or {p₁}, a)</code>	$=$	<code>union(analysis(p₁, a), analysis(p₀, a))</code>
<code>analysis(p₀; p₁, a)</code>	$=$	<code>analysis(p₁, analysis(p₀, a))</code>
<code>analysis(iter{p}, a)</code>	$=$	{ $R \leftarrow a;$ repeat $T \leftarrow R;$ $R \leftarrow widen(R, analysis(p, R));$ until inclusion(R, T) return T;

Trace verification

Problem statement:

can the program compute a point inside error region E ?

need to collect the set of reachable points

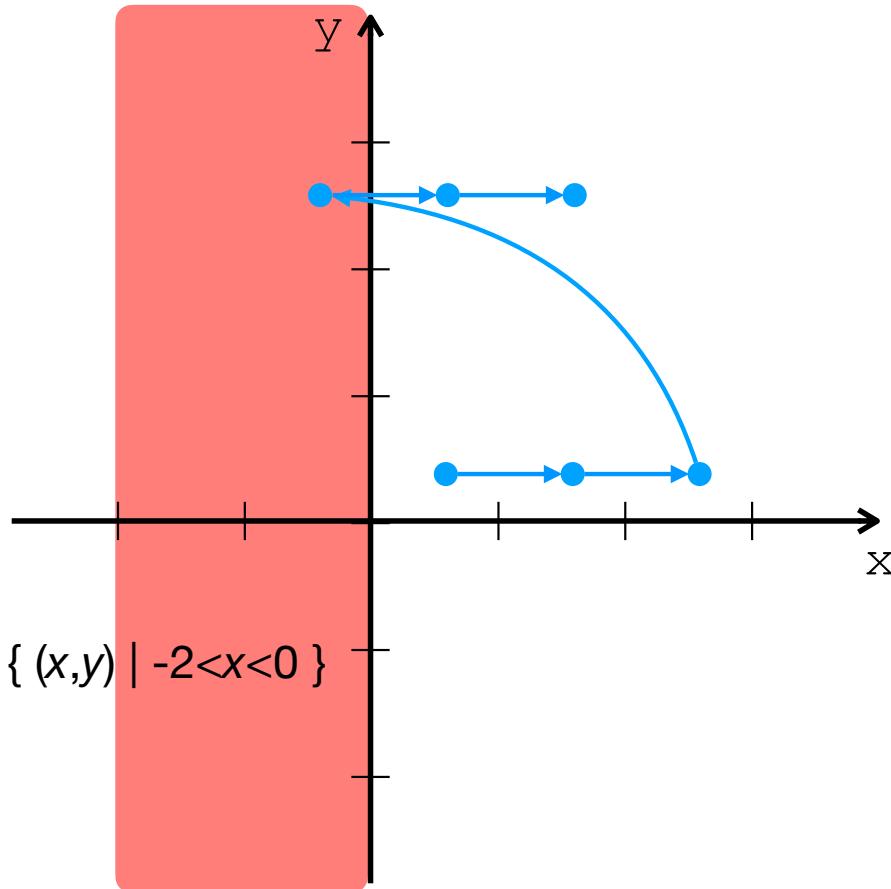
run analysis and collect all points R from every call to analysis.

The analysis is sound: R over-approximates reachable points

if $R \cap E = \emptyset$ the program is verified

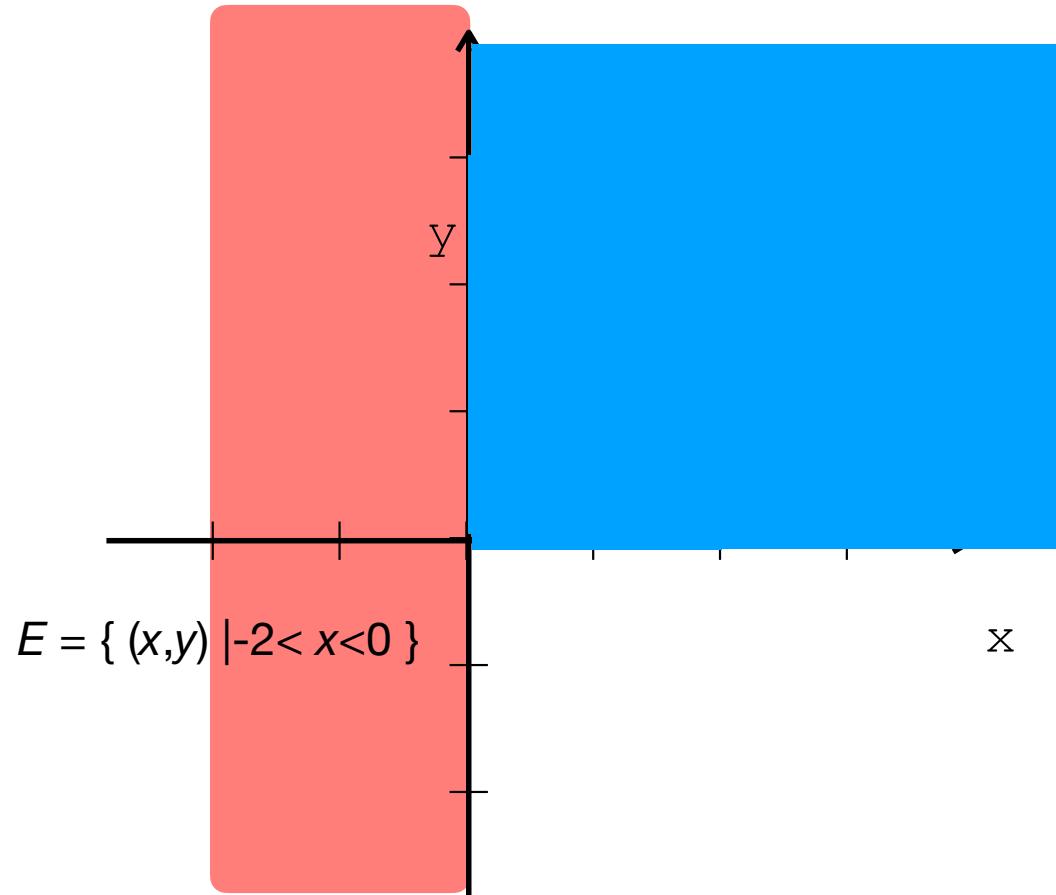
otherwise... we don't know

An incorrect execution



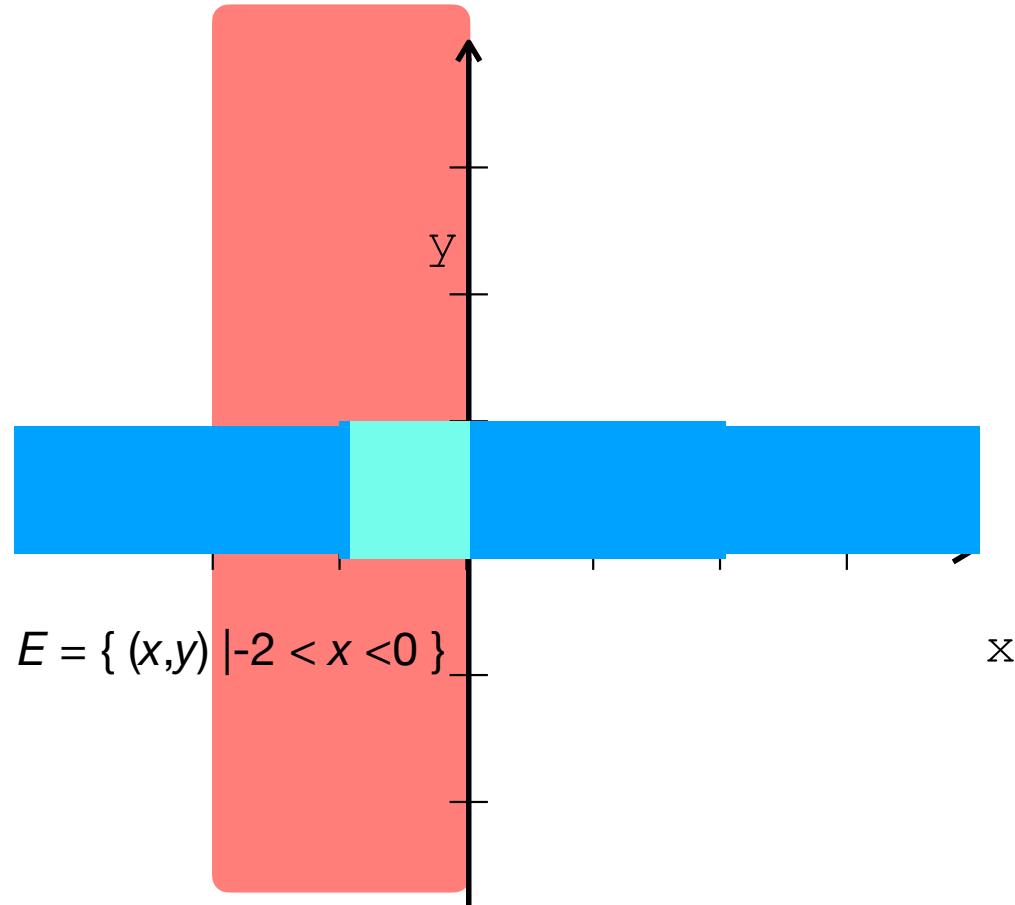
Abstract execution: verification

```
init([0,1]x[0,1]); ←  
iter {  
    {  
        translate(1, 0)  
    } or {  
        translate(1, 0.5)  
    }  
}
```



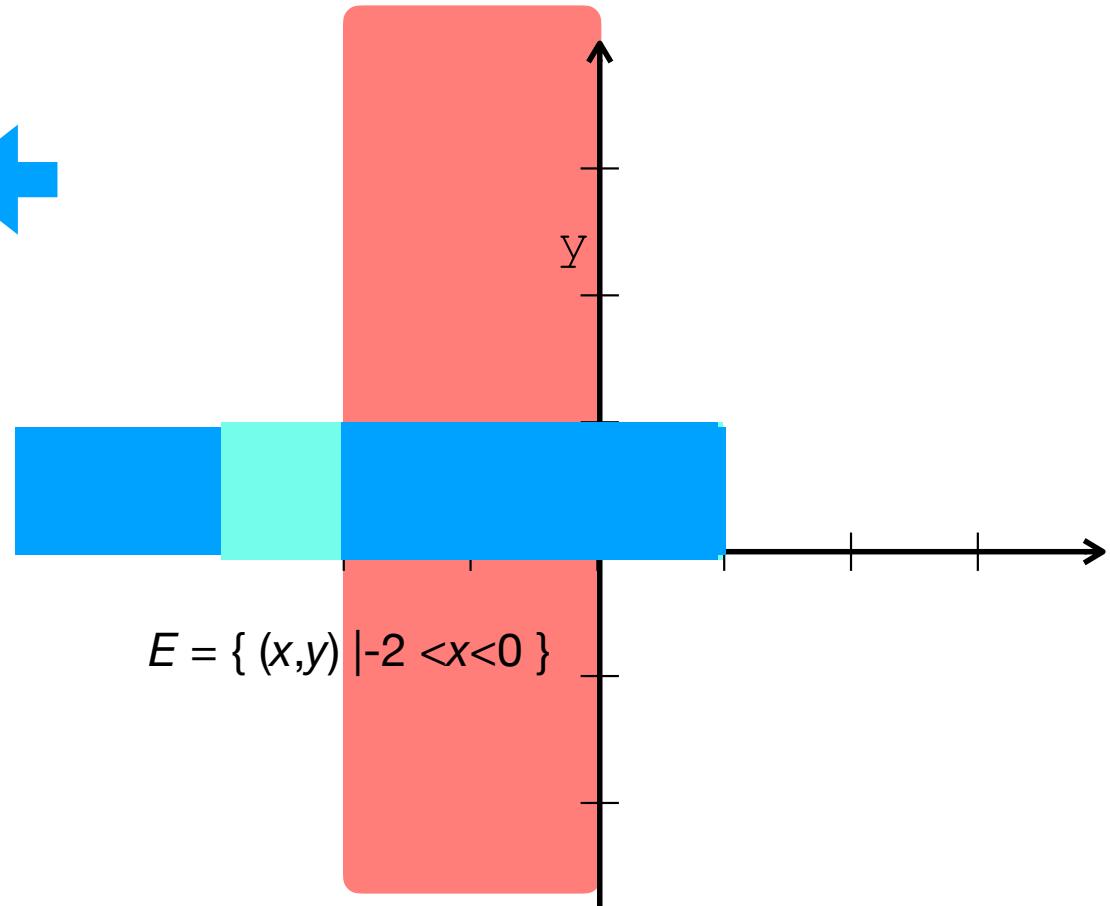
Abstract execution: true alarms

```
init([0,1]x[0,1]);  
iter {  
    translate(1,0)  
} or {  
    translate(-1,0)  
}  
}
```



Abstract execution: false alarm

```
init([0,1]x[0,1]); ←  
iter {  
    translate(-3, 0)  
}  
}
```



Three staged approach: 1

Selection of the semantics and properties of interest

fix the goal of the analysis

describe program behaviour

describe properties to be verified

formal description (we did it in prose, by examples)

Three staged approach: 2

Choice of the abstraction

describe properties manipulated by the analysis

must express properties of interest

must express useful invariants

Three staged approach: 3

Derivation of the analysis algorithm

follow the choices made in first two phases
the analysis closely follows the semantics

Abstract domain

We need the ability to
compute (possibly best) abstract representations

apply sound transformations to abstract elements

compare abstract elements for inclusion

compute the union/widening of abstract elements

Principles of Abstract Interpretation

15

Abstract Interpretation Framework

Abstract interpretation Framework

real execution

$$\llbracket P \rrbracket = \text{fix } F \in D \quad A \text{ domain of concrete states} \\ (\text{e.g. sets of integers})$$

abstract execution

$$\llbracket \widehat{P} \rrbracket = \text{fix } \widehat{F} \in D \quad A \text{ domain of abstract states} \\ (\text{e.g. sets of intervals})$$

correctness

$$\llbracket P \rrbracket \approx \llbracket \widehat{P} \rrbracket$$

implementation computation of $\llbracket \widehat{P} \rrbracket$



Abstract interpretation Framework

The framework requires:

- a relation between D and \hat{D}
- A relation between $F : D \rightarrow D$ and $\hat{F} : \hat{D} \rightarrow \hat{D}$

A function corresponding to
one-step abstract execution

The framework guarantees:

- correctness and implementation
- freedom: any such \hat{D} and \hat{F} are fine

A function corresponding to
one-step concrete execution

Recipe for the construction of an abstract interpreter

Step 1 : Define the language and a concrete semantics

Step 2 : Select an abstraction describing the set of properties

Step 3 : Derive the abstract semantics

The language

Assume a syntax for arithmetic expressions E and Boolean expression B, the syntax for the command is the following

$C ::=$	commands
skip	command that “does nothing”
$C; C$	sequence of commands
$x := E$	assignment command
input(x)	command reading of a value
if(B){C}else{C}	conditional command
while(B){C}	loop command
$P ::= C$	program

Step 1: Define concrete Semantics

Formalization of a single program execution

Operational semantics (transitional style)

- Big-step / small-step

Denotational semantics (compositional style)

- State → State

Semantics Style: Compositional vs. Transitional

Compositional semantics is defined by the semantics of sub-parts of a program

$$\llbracket AB \rrbracket = \dots \llbracket A \rrbracket \dots \llbracket B \rrbracket$$

For some realistic languages, even defining their compositional ("denotational") semantics is a hurdle

- goto, exceptions, function calls

Transitional-style ("operational") semantics avoids the hurdle

$$\llbracket AB \rrbracket = \{ s_1 \rightarrow s_2 \rightarrow \dots \}$$

Step 1: Define concrete Semantics

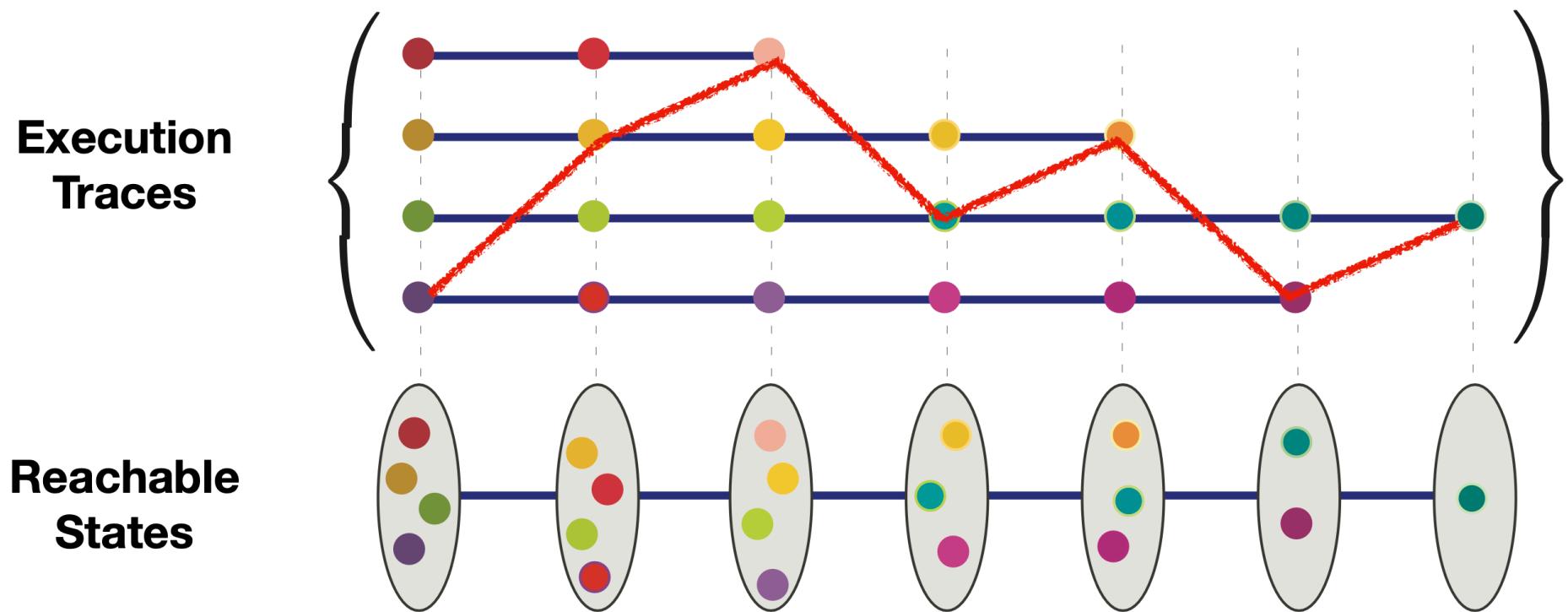
Formalization of all possible program executions

Also called **collecting** semantics

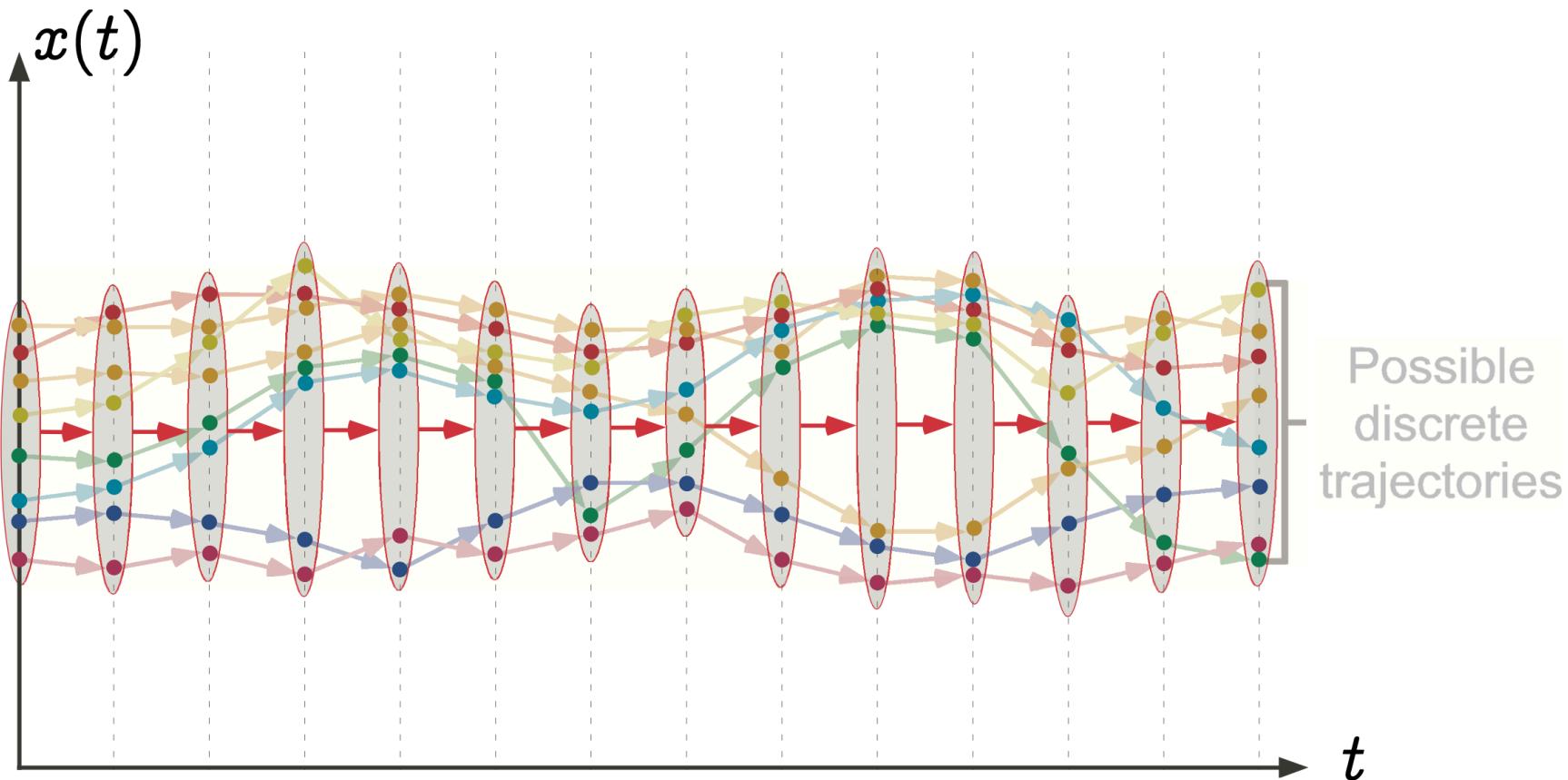
Simple extension of the standard semantics in general

$$2^{States} \rightarrow 2^{States}$$

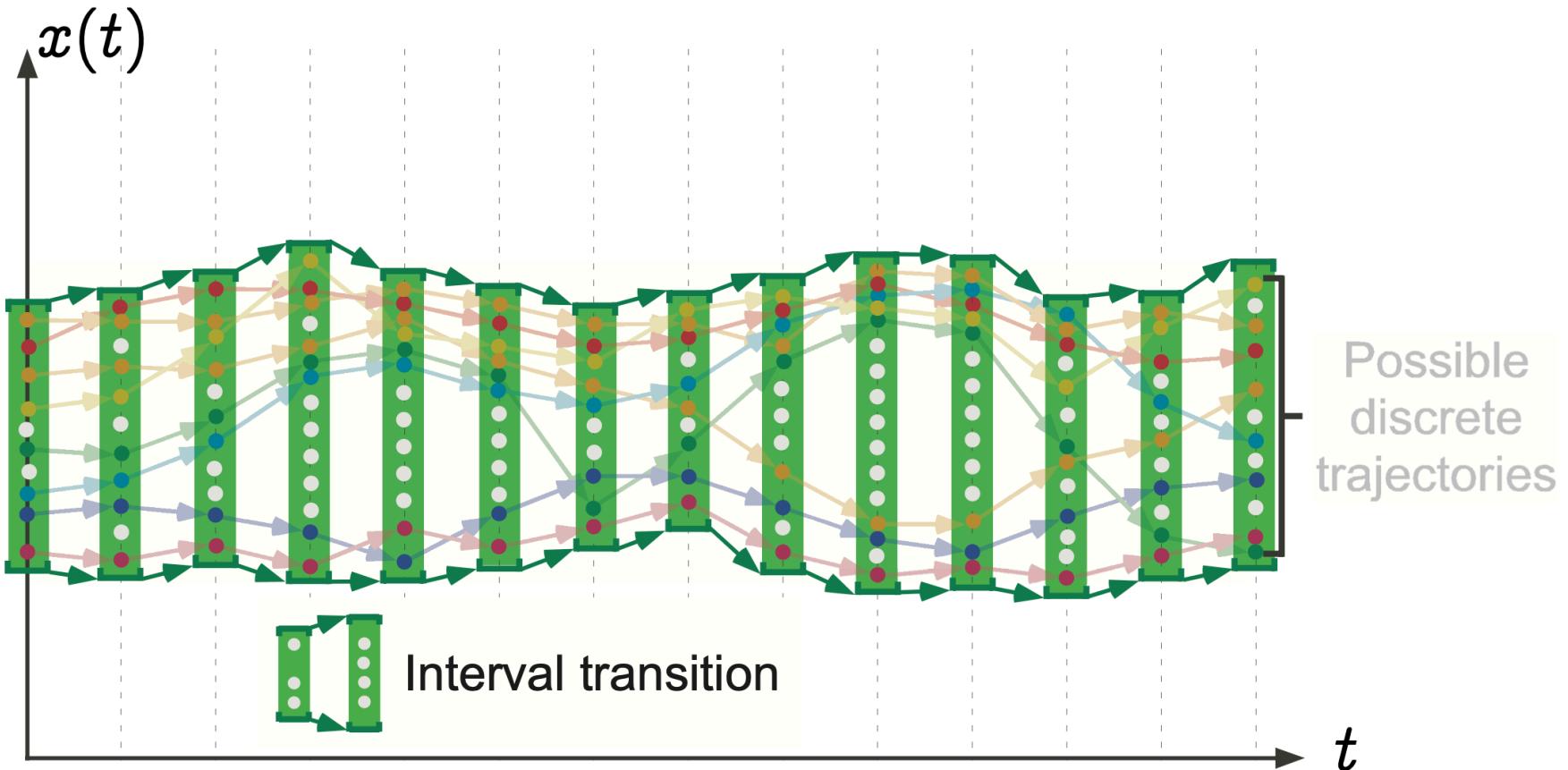
Traces vs. Reachable States



Transitions of sets of States



Transitions of Abstract States



Collecting Semantics

$x \in \mathbb{X} = \text{ProgramVariables}$

$\mathbb{V} = \mathbb{Z}$

Memories

$m \in \mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$

Assume

$\llbracket E \rrbracket : \mathbb{M} \rightarrow \mathbb{V}$ and $\llbracket B \rrbracket : \mathbb{M} \rightarrow \mathbb{B}$

$\llbracket C \rrbracket : \wp(\mathbb{M}) \rightarrow \wp(\mathbb{M})$

$M \in \wp(\mathbb{M})$

$\llbracket \text{skip} \rrbracket(M) = M$

$\llbracket x := E \rrbracket(M) = \{ m[x \mapsto \llbracket E \rrbracket(m)] \mid m \in M\}$

$\llbracket C_0; C_1 \rrbracket(M) = \llbracket C_1 \rrbracket(\llbracket C_0 \rrbracket(M))$

$\llbracket \text{input} \rrbracket(M) = \{ m[x \mapsto n] \mid n \in \mathbb{V}, m \in M\}$

Filtering function for the conditional

Since M is a set of states, the conditional filters the memories for which the condition is true and for them evaluate the first branch, do the same for the memories for which the condition is false and take the union

For each Boolean expression B , the filtering function

$$\mathcal{F}_B(M) = \{ m \in M \mid \llbracket B \rrbracket(m) = \text{true} \}$$

Collecting semantics for the conditional

$$\mathcal{F}_B(M) = \{ m \in M \mid \llbracket B \rrbracket(m) = \text{true} \}$$

Syntactic negation
E.g. $\neg(x > 3) = x \leq 3$

$$\mathcal{F}_{\neg B}(M) = \{ m \in M \mid \llbracket \neg B \rrbracket(m) = \text{true} \} = \{ m \in M \mid \llbracket B \rrbracket(m) = \text{false} \}$$

$$\llbracket \text{if } (B)\{C_0\} \text{ else } \{C_1\} \rrbracket(M) = \llbracket C_0 \rrbracket \mathcal{F}_B(M) \cup \llbracket C_1 \rrbracket \mathcal{F}_{\neg B}(M)$$

Collecting semantics $\llbracket \text{while}(B)\{C\} \rrbracket(M)$

We can partition executions based on the number of iterations they spend inside the loop before exit

M_i denotes the memories that are produced by program executions that went thought the loop body exactly i times starting from M

$$M_1 = \mathcal{F}_{\neg B}(\llbracket C \rrbracket \mathcal{F}_B(M))$$

$$M_2 = \mathcal{F}_{\neg B}(\llbracket C \rrbracket \mathcal{F}_B \llbracket C \rrbracket \mathcal{F}_B(M)) = \mathcal{F}_{\neg B}((\llbracket C \rrbracket \mathcal{F}_B)^2(M))$$

$$M_i = \mathcal{F}_{\neg B}((\llbracket C \rrbracket \mathcal{F}_B)^i(M))$$

Collecting semantics $\llbracket \text{while}(B)\{C\} \rrbracket(M)$

Thus, the set of output states of the loop is

$$\bigcup_{i \geq 0} M_i = \bigcup_{i \geq 0} \mathcal{F}_{\neg B}((\llbracket C \rrbracket \mathcal{F}_B)^i(M))$$

Since \mathcal{F}_B commutes with the union

$$\bigcup_{i \geq 0} M_i = \mathcal{F}_{\neg B}(\bigcup_{i \geq 0} (\llbracket C \rrbracket \mathcal{F}_B)^i(M))$$

Definition as fix-point

$$\llbracket \text{while}(B)\{C\} \rrbracket(M) = \mathcal{F}_{\neg B}(\bigcup_{i \geq 0} (\llbracket C \rrbracket \mathcal{F}_B)^i(M))$$

This can be rewritten as

$$\llbracket \text{while}(B)\{C\} \rrbracket(M) = \mathcal{F}_{\neg B}(\text{fix } F_M)$$

where

$$F_M = \lambda M'.M \cup \llbracket C \rrbracket \mathcal{F}_B(M')$$

Definition as fix-point

$$[\![\text{while}(B)\{C\}]\!](M) = \mathcal{F}_{\neg B}(\text{fix } F_M)$$

$$F_M = \lambda M'. M \cup [\![C]\!] \mathcal{F}_B(M')$$

F_M is continuous then we can apply the Kleene's theorem to compute
the invariant

$$\begin{aligned} F_M^0 &= F_M(\emptyset) = M \\ F_M^1 &= F_M(F_M^0) = M \cup [\![C]\!] \mathcal{F}_B(M) \\ F_M^2 &= F_M(F_M^1) = M \cup ([\![C]\!] \mathcal{F}_B)^2(M) \\ &\vdots \\ F_M^i &= M \cup ([\![C]\!] \mathcal{F}_B)^i(M) \end{aligned}$$

$$[\![\text{while}(B)\{C\}]\!](M) = \mathcal{F}_{\neg B}(\cup_{i<0} F_M^i)$$

Toward abstraction

Our concrete domain $(\wp(\mathbb{M}), \subseteq)$

We abstract each concrete element with an abstract element

$c \models a$ when the abstract element a describes c

$$M_0 = \{m \in \mathbb{M} \mid 0 \leq m(x) \leq m(y) \leq 8\} \models M^\# = \{x \mapsto [0, 10], y \mapsto [0, 80]\}$$

$$M_1 = \{m \in \mathbb{M} \mid 1 \leq m(x)\} \not\models M^\# = \{x \mapsto [0, 10], y \mapsto [0, 80]\}$$

Abstract relation

Given a concrete domain (C, \subseteq) an abstraction is defined by an abstract domain (A, \sqsubseteq) and an abstract relation $\models \subseteq C \times A$ such that

- if $a_0 \sqsubseteq a_1$ and $c \models a_0$ then also $c \models a_1$

$$a_0 = \{x \mapsto [0, 10], y \mapsto [0, 80]\} \sqsubseteq a_1 = \{y \mapsto [0, 100]\}$$

$$c_1 = \{m \in \mathbb{M} \mid 0 \leq m(x) \leq m(y) \leq 8\} \models a_0 \implies c_1 \models a_1$$

- if $c_0 \subseteq c_1$ and $c_1 \models a$ then also $c_0 \models a$

$$c_0 = \{m \in \mathbb{M} \mid 0 \leq m(x) \leq 4, m(y) = 6\} \subseteq c_1$$

$$c_1 \models a_0 \implies c_0 \models a_0$$

Concretization function

A common way to describe the abstract relation \models is by defining a function that maps each abstract element to the largest concrete element it describes

Definition

Concretization function $\gamma : A \rightarrow C$ is a monotone function that maps abstract a into the **greatest** concrete c that satisfies a ($c \models a$).

$$c \models a \Leftrightarrow c \subseteq \gamma(a)$$

$$\gamma(a_0) = \gamma(\{x \mapsto [0, 10], y \mapsto [0, 80]\}) = \{m \in \mathbb{M} \mid 0 \leq m(x) \leq 10, 0 \leq m(y) \leq 80\}$$

$$c_1 = \{m \in \mathbb{M} \mid 0 \leq m(x) \leq m(y) \leq 8\} \models a_0 \text{ since } c_1 \subseteq \gamma(a_0)$$

Abstraction function

Another way to describe the abstract relation \models is by defining a function that maps each concrete element to the smallest abstract element that describes it

Definition

Abstraction function $\alpha : C \rightarrow A$ (if it exists) is a monotone function

that maps concrete c into the most precise abstract a that describes c ($c \models a$). $c \models a \Leftrightarrow \alpha(c) \sqsubseteq a$

$$\alpha(c_1) = \alpha(\{m \in \mathbb{M} \mid 0 \leq m(x) \leq m(y) \leq 8\}) = \{x \mapsto [0, 8], y \mapsto [0, 8]\}$$

$$c_1 \models a_1 = \{y \mapsto [0, 100]\} \text{ since } \alpha(c_1) \sqsubseteq a_1$$

Galois connection

α and γ should agree on a same abstraction relation \sqsubseteq

$$c \models a \Leftrightarrow c \subseteq \gamma(a) \Leftrightarrow \alpha(c) \sqsubseteq a$$

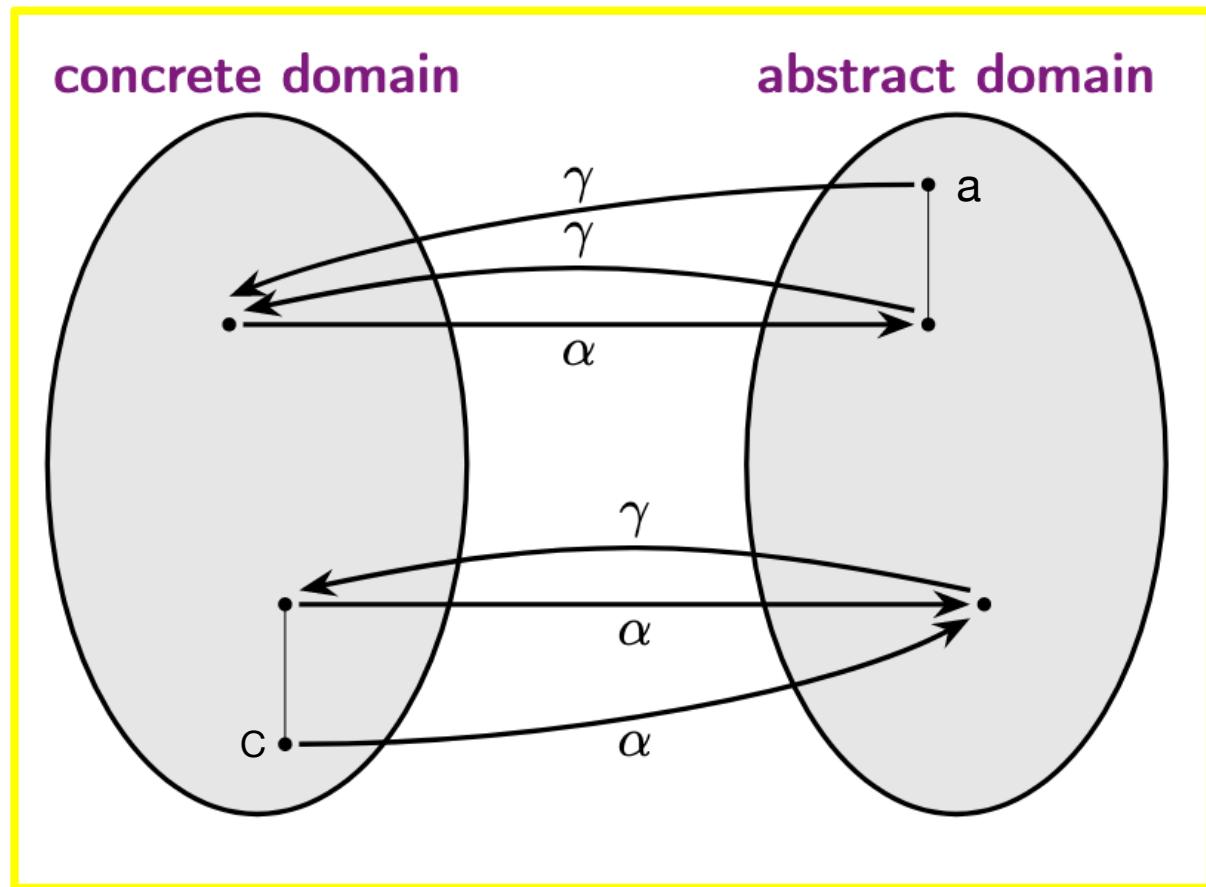
Definition

Galois connection: a pair of concretization function $\gamma : A \rightarrow C$ and an abstraction function $\alpha : C \rightarrow A$ such that

$$c \subseteq \gamma(a) \Leftrightarrow \alpha(c) \sqsubseteq a$$

Properties of Galois connections

- α and γ are monotone
- $c \subseteq \gamma(\alpha(c))$
- $\alpha(\gamma(a)) \sqsubseteq a$



Step 2: Non-relational abstractions

Non-relational abstractions: they forget relations among program variables

All the values for variables are abstracted independently

They proceed in two steps:

1. Collect the values a variables may take across a set of states
2. Over-approximate the set of values for each variable with an abstract element of a domain of value abstraction

Abstract states

$$\underbrace{(\wp(\mathbb{M}), \subseteq)}_{\text{CONCRETE DOM.}} \xrightleftharpoons[\alpha_M]{\gamma_M} \underbrace{(\mathbb{M}^\#, \sqsubseteq_M)}_{\text{ABS. DOM.}}$$

$M^\# \in \mathbb{M}^\# = \mathbb{X} \rightarrow \mathbb{V}^\#$

| |
IDE ABS. VALUES

$$\alpha_M(M)(x) = \alpha_V(\{m(x) \mid m \in M\})$$

$$\gamma_M(M^\#) = \{m \mid \forall x . m(x) \in \gamma_V(M^\#(x))\}$$

“MEM ABSTRACTIONS”

Signs

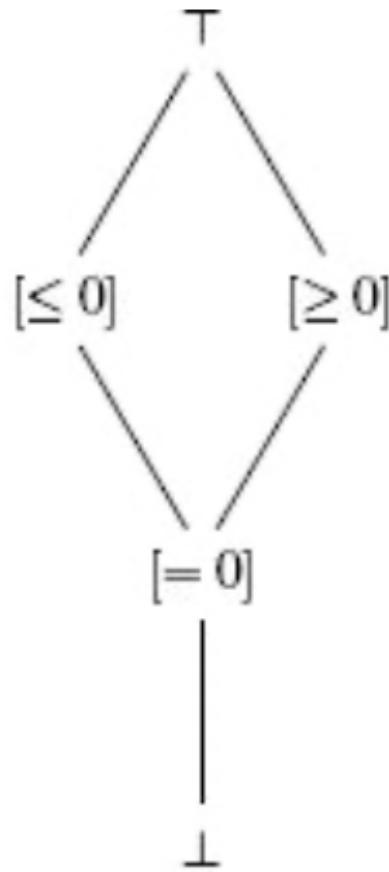
$$\gamma([\geq 0]) = \{n \in \mathbb{V} \mid n \geq 0\}$$

$$\gamma([\leq 0]) = \{n \in \mathbb{V} \mid n \leq 0\}$$

$$\gamma([= 0]) = \{0\}$$

$$\gamma(\top) = \mathbb{V}$$

$$\gamma(\perp) = \{\}$$



$$\alpha(\{2, 4, 8, 16, \dots\}) = [\geq 0]$$

$$\alpha(\{0\}) = [0]$$

$$\alpha(\{-1, 1\}) = \top$$

Intervals

Elements of A :

- \perp the empty set of values
- $(n_0, n_1), n_0 \in (\mathbb{Z} \cup \{-\infty\}), n_1 \in (\mathbb{Z} \cup \{+\infty\}), n_0 \leq n_1$

\sqsubseteq is the interval inclusion

A SET OF NWMS THAT WE ABSTRACT AS AN INTERVAL: $C = \{1, 3, 2, \emptyset, \dots\}$

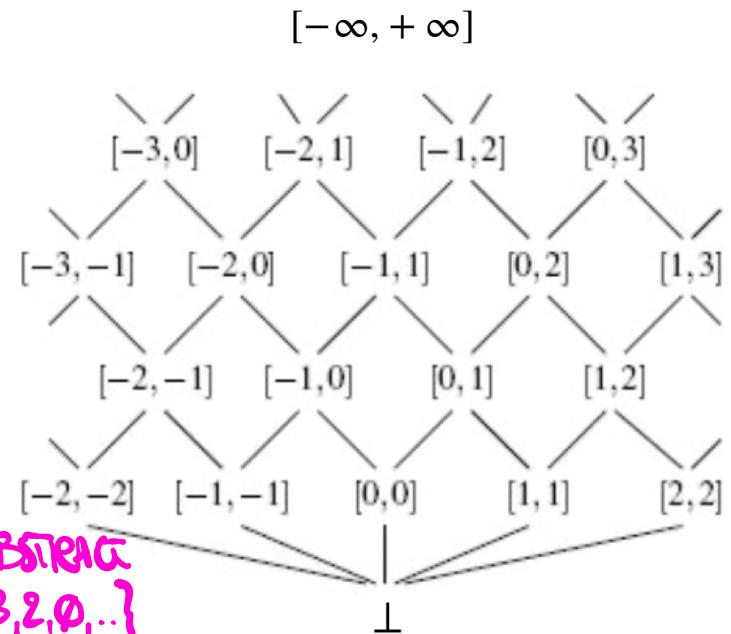
$$\gamma(\perp) = \{\}$$

$$\gamma([n_0, n_1]) = \{ n \in \mathbb{V} \mid n_0 \leq n \leq n_1 \}$$

$$\gamma([-\infty, n_1]) = \{ n \in \mathbb{V} \mid n \leq n_1 \}$$

$$\gamma([n_0, +\infty]) = \{ n \in \mathbb{V} \mid n_0 \leq n \}$$

$$\gamma([-\infty, +\infty]) = \mathbb{V}$$



$$\alpha(c) = \perp \text{ if } c = \emptyset,$$

$$\alpha(c) = [min(c), max(c)] \text{ if } c \neq \emptyset, min(c) \text{ and } max(c) \text{ exists}$$

$$\alpha(c) = [min(c), +\infty] \text{ if } c \neq \emptyset, min(c) \text{ exists}$$

$$\alpha(c) = [-\infty, max(c)] \text{ if } c \neq \emptyset, max(c) \text{ exists}$$

$$\alpha(c) = [-\infty, +\infty] \text{ otherwise}$$

Example

Consider the following set of memories M

$$m_0 : \begin{array}{l} x \mapsto 25 \\ y \mapsto 7 \\ z \mapsto -12 \end{array}$$

$$m_1 : \begin{array}{l} x \mapsto 28 \\ y \mapsto -7 \\ z \mapsto -11 \end{array}$$

$$m_2 : \begin{array}{l} x \mapsto 20 \\ y \mapsto 0 \\ z \mapsto -10 \end{array}$$

$$m_3 : \begin{array}{l} x \mapsto 35 \\ y \mapsto 8 \\ z \mapsto -9 \end{array}$$

With the Sign abstraction

$$M^\# : \begin{array}{l} x \mapsto [\geq 0] \\ y \mapsto \top \\ z \mapsto [\leq 0] \end{array}$$

With the interval abstraction

$$M^\# : \begin{array}{l} x \mapsto [25,35] \\ y \mapsto [-7,8] \\ z \mapsto [-12,-9] \end{array}$$

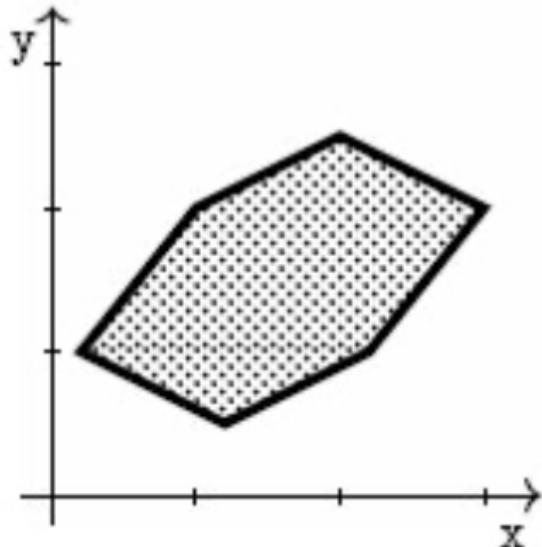
Relational domain

Convex Polyhedra domain

sets of numerical constraints of the form

$$c_1x + c_2y \leq c$$

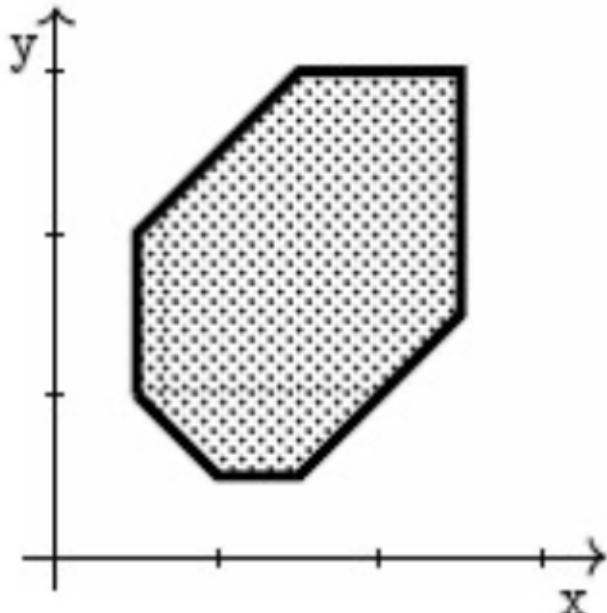
(at most two variables per constraint,
with unit coefficients)



does not admit a best abstraction

Relational domain

Octagon domain



sets of numerical constraints of
the form

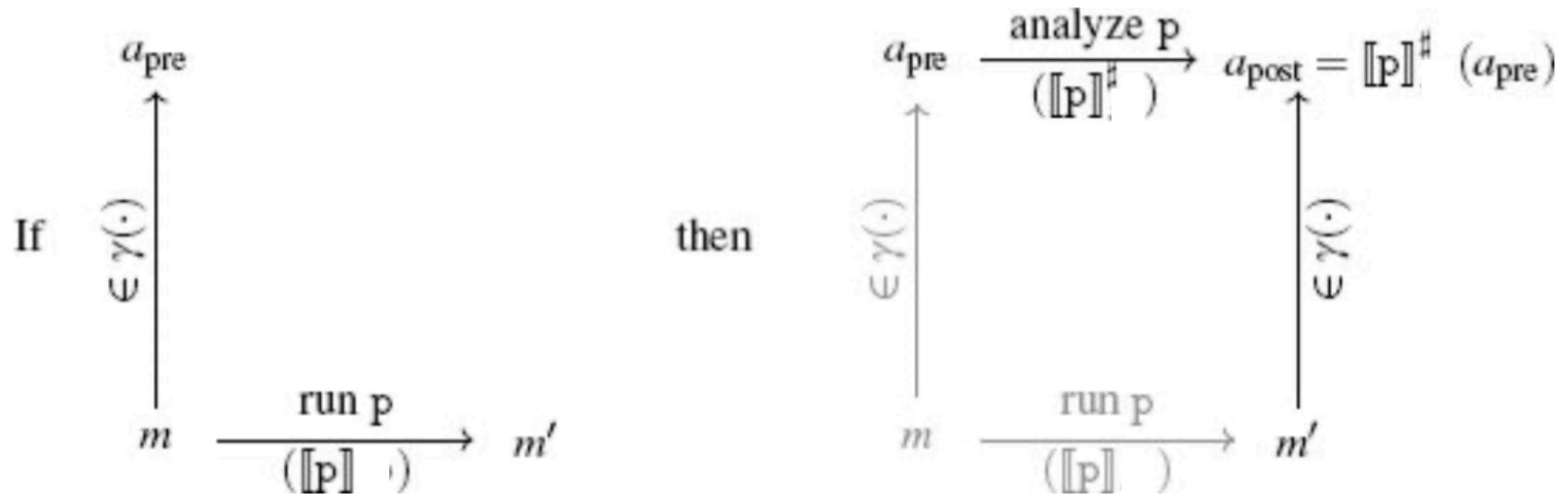
$$\pm x \pm y \leq c$$

(at most two variables per
constraint, with unit
coefficients)

A admits the best abstraction

Step 3: Abstract semantics

We want to define a **sound** abstract semantics



Abstract semantics of command

It will be defined by induction on the syntax

$$\llbracket C \rrbracket^\# \perp = \perp$$

$$\llbracket \text{skip} \rrbracket^\# M^\# = M^\#$$

$$\llbracket C_0; C_1 \rrbracket^\# M^\# = \llbracket C_1 \rrbracket^\# (\llbracket C_0 \rrbracket^\# (M^\#))$$

This and all inductive construction rely on the following result:

Let

$$F_0, F_1 : \wp(\mathbb{M}) \rightarrow \wp(\mathbb{M})$$

$$F_0^\#, F_1^\# : \mathbb{A} \rightarrow \mathbb{A}.$$

If $F_i \gamma \subseteq \gamma F_i^\#$,
then

$$F_0 F_1 \gamma \subseteq \gamma F_0^\# F_1^\#$$

Abstract interpretation of expressions

$$[\![E]\!]^\# : \mathbb{M}^\# \rightarrow \mathbb{V}^\#$$

$$[\![n]\!]^\# M^\# = \alpha(\{n\})$$

$$[\![x]\!]^\# M^\# = M^\#(x)$$

$$[\![E_0 + E_1]\!]^\# M^\# = [\![E_0]\!]^\# +^\# [\![E_1]\!]^\#$$

Sign domain

$$[\geq 0] +^\# [\leq 0] = \top$$

$$[\geq 0] +^\# [\geq 0] = [\geq 0]$$

Interval domain

$$[0, 6] +^\# [-2, 3] = [-2, 9]$$

$$[-\infty, -2] +^\# [4, 18] = [-\infty, 16]$$

Analysis of assignment

$$\llbracket x := E \rrbracket^{\#} M^{\#} = M^{\#}[x \mapsto (\llbracket E \rrbracket^{\#}(M^{\#}))]$$

Sign domain

$$\begin{aligned}\llbracket x := x + 6 + y \rrbracket &\{x \mapsto [\geq 0], y \mapsto \top\} \\ &= \{x \mapsto \top, y \mapsto \top\}\end{aligned}$$

$$\llbracket \text{input}(x) \rrbracket^{\#} M^{\#} = M^{\#}[x \mapsto \top]$$

Interval domain

$$\begin{aligned}\llbracket x := x + 6 + y \rrbracket &\{x \mapsto [3, 8], y \mapsto [-3, 5]\} \\ &= \{x \mapsto [6, 19], y \mapsto [-3, 5]\}\end{aligned}$$

Abstract interpretation of the conditional branching

$$[\![\text{if } (B) \{C_0\} \text{ else } \{C_1\}]\!](M) = [\![C_0]\!] \mathcal{F}_B(M) \cup [\![C_1]\!] \mathcal{F}_{\neg B}(M)$$

We use the compositional principle and we need to define over approximations of

- \mathcal{F}_B and of $\mathcal{F}_{\neg B}$
- the join operator \cup

Analysis of conditions

$$\underline{F_B} = \{ M \in M \text{ s.t } B = \text{TRUE} \}$$

For all $M^\#$, $\mathcal{F}_B(\gamma(M^\#)) \subseteq \gamma(\mathcal{F}_B^\#(M^\#))$

Sign domain $\mathcal{F}_{x < 0}^\#(M^\#) = \begin{cases} (y \in \mathbb{X}) \mapsto \perp & \text{if } M^\#(x) = [\geq 0] \text{ or } [= 0] \text{ or } \perp \\ M^\#[x \mapsto [\leq 0]] & \text{if } M^\#(x) = [\leq 0] \text{ or } \top \end{cases}$

Interval domain $\mathcal{F}_{x < n}^\#(M^\#) = \begin{cases} (y \in \mathbb{X}) \mapsto \perp & \text{if } a > n \\ M^\#[x \mapsto [a, n]] & \text{if } a \leq n \leq b \\ M^\# & \text{if } b \leq n \end{cases}$

Analysis of conditions

```
if(x > 7){  
    y := x - 7  
}else{  
    y := 7 - x  
}
```

Interval domain

$$\mathcal{F}_{x>7}^{\#}(\{x \mapsto \top, y \mapsto \top\}) = \{x \mapsto [8, +\infty], y \mapsto \top\}$$

$$\mathcal{F}_{x \leq 7}^{\#}(\{x \mapsto \top, y \mapsto \top\}) = \{x \mapsto [-\infty, 7], y \mapsto \top\}$$

Analysis of flow joins

We need to define a correct over approximation of the join \cup , that is, an abstract join $\cup^\#$ s.t. $\gamma(M_0^\#) \cup \gamma(M_1^\#) \subseteq \gamma(M_0^\# \sqcup^\# M_1^\#)$

$$\begin{aligned} M_0^\# &= \{x \mapsto [0, 3], y \mapsto [6, 7], z \mapsto [4, 8]\} \\ M_1^\# &= \{x \mapsto [5, 6], y \mapsto [0, 2], z \mapsto [6, 9]\} \end{aligned}$$

For the interval domain is defined in terms of min and max of intervals

$$M_0^\# \sqcup^\# M_1^\# = \{x \mapsto [0, 6], y \mapsto [0, 7], z \mapsto [4, 9]\}$$

Analysis of Conditional Command

$$[\text{if } (B)\{C_0\} \text{ else } \{C_1\}]^\#(M^\#) = [C_0]^\# \mathcal{F}_B^\#(M) \cup^\# [C_1]^\# \mathcal{F}_{\neg B}^\#(M^\#)$$

```
if(x > 7){  
    y := x - 7  
}else{  
    y := 7 - x  
}
```

Starting with $\{x \mapsto \top, y \mapsto \top\}$

on the true branch we filter for condition $x > 7$

$$\mathcal{F}_{x>7}^\#(\{x \mapsto \top, y \mapsto \top\}) = \{x \mapsto [8, +\infty], y \mapsto \top\}$$

$$[y := x - 7]^\#(\{x \mapsto [8, +\infty], y \mapsto \top\}) = \{x \mapsto [8, +\infty], y \mapsto [1, +\infty]\}$$

on the false branch we filter for condition $x \leq 7$

$$\mathcal{F}_{x \leq 7}^\#(\{x \mapsto \top, y \mapsto \top\}) = \{x \mapsto [-\infty, 7], y \mapsto \top\}$$

$$[y := 7 - x]^\#(\{x \mapsto [-\infty, 7], y \mapsto \top\}) = \{x \mapsto [-\infty, 7], y \mapsto [0, +\infty]\}$$

Applying the abstract join we obtain



$$\boxed{\{x \mapsto \top, y \mapsto [0, +\infty]\}}$$

Abstract interpretation of the loop

Recall the concrete semantics of the loop

For $F = \llbracket C \rrbracket \mathcal{F}_B$

$$\llbracket \text{while}(B)\{C\} \rrbracket(M) = \mathcal{F}_{\neg B}(\bigcup_{i \geq 0} (\llbracket C \rrbracket \mathcal{F}_B)^i(M)) = \mathcal{F}_{\neg B}(\bigcup_{i \geq 0} F^i(M))$$

We can approximate \mathcal{F}_B and F so the problem we need to solve is how to compute an approximation of an infinite union $\bigcup_{i > 0} F^i(M)$

Concrete iterations

$$M_n = \bigcup_{i=0}^n F^i(M)$$

$$M_0 = M$$

$$M_{k+1} = M_k \cup F(M_k)$$

Abstract iterations

$$M_0^\# = M^\#$$

$$M_{k+1}^\# = M_k^\# \cup F^\#(M_k^\#)$$

Abstract iterations

$x := 0;$

while($x \geq 0$) {

$x := x + 1$

}

After the first assignment we have $M^\# = \{x \mapsto [0,0]\}$

$$M_0^\# = \{x \mapsto [0,0]\}$$

$$M_1^\# = \{x \mapsto [0,1]\}$$

$$M_2^\# = \{x \mapsto [0,2]\}$$

$$\vdots = \vdots$$

$$M_n^\# = \{x \mapsto [0,n]\}$$

$$\vdots = \vdots$$

$x := 0;$

while($x \leq 100$) {

if($x \geq 50$) {

$x := 10$

else {

$x := x + 1$

}

$$M_0^\# = \{x \mapsto [0,0]\}$$

$$M_1^\# = \{x \mapsto [0,1]\}$$

$$M_2^\# = \{x \mapsto [0,2]\}$$

$$\vdots = \vdots$$

$$M_{49}^\# = \{x \mapsto [0,49]\}$$

$$M_{50}^\# = \{x \mapsto [0,50]\}$$

$$M_{51}^\# = \{x \mapsto [0,50]\}$$

$$M_{52}^\# = \{x \mapsto [0,50]\}$$

$$\vdots = \vdots$$

Convergence of iterates

The computation of abstract iterations may not converge or it can converge too slowly

We can choose to use finite Height Domain

We can design widening operators

Finite height lattices

If the abstract domain has finite height the abstract iterations are finite

abs_iter(F^\sharp, M^\sharp)

$R \leftarrow M^\sharp;$

repeat

$T \leftarrow R;$

$R \leftarrow R \sqcup^\sharp F^\sharp(R);$

until $R = T$

return $M_{\lim}^\sharp = T;$

$x := 0;$

while($x \geq 0$) {

$x := x + 1$

}

$M_0^\sharp = \{x \mapsto [= 0]\}$

$M_1^\sharp = \{x \mapsto [\geq 0]\}$

$M_2^\sharp = \{x \mapsto [\geq 0]\}$

$x := 0;$

while($x \leq 100$) {

if($x \geq 50$) {

$x := 10$

else{

$x := x + 1$

}

}

Widening operator

Definition A widening operator over an abstract domain is a binary operator s.t.

- it holds $\gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \triangledown a_1)$
- for any sequence $(a_n)_{n \in \mathbb{N}}$, the sequence $(a'_n)_{n \in \mathbb{N}}$ defined as follows is ultimately stationary:

$$a'_0 = a_0$$

$$a'_{n+1} = a'_n \triangledown a_n$$

Widening operator for intervals

$$[n, p] \nabla_{\mathcal{F}} [n, q] = \begin{cases} [n, p] & \text{if } p \geq q \\ [n, +\infty) & \text{if } p < q \end{cases}$$

The same for the other bound

abs_iter(F^\sharp, M^\sharp)

$R \leftarrow M^\sharp;$

repeat

$T \leftarrow R;$

$R \leftarrow R \nabla F^\sharp(R);$

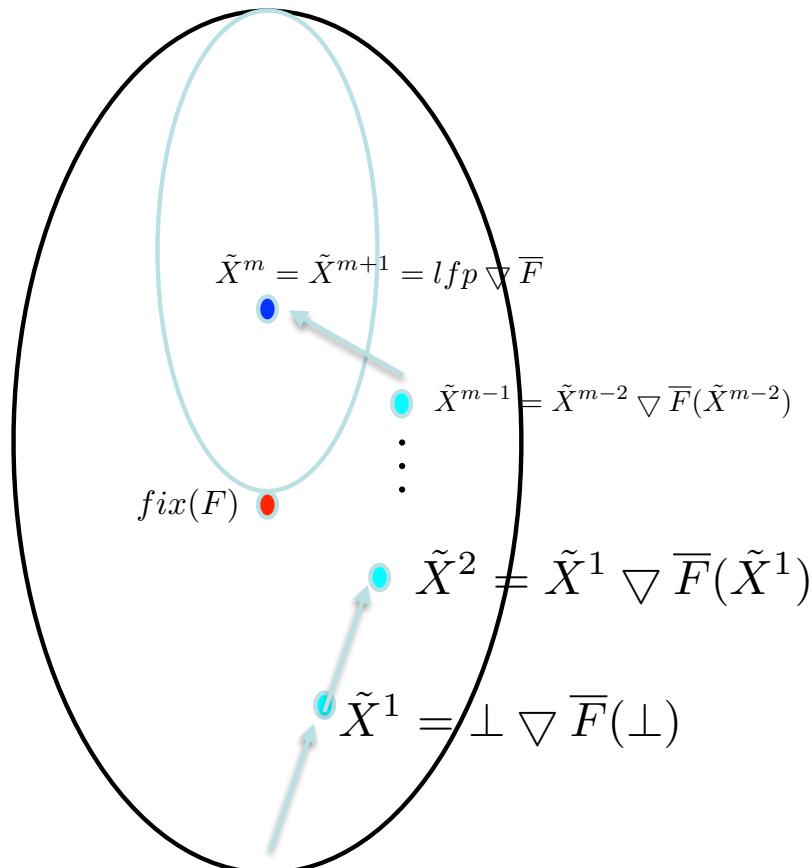
until $R = T$

return $M_{\lim}^\sharp = T;$

The abstract iterations become

Widening

$$\tilde{X}^m \sqsupseteq \overline{F}(\tilde{X}^m)$$



The analysis

$$\llbracket n \rrbracket^\# M^\# = \alpha(\{n\})$$

$$\llbracket x \rrbracket^\# M^\# = M^\#(x)$$

$$\llbracket E_0 + E_1 \rrbracket^\# M^\# = \llbracket E_0 \rrbracket^\# \textcolor{red}{+^\#} \llbracket E_1 \rrbracket^\#$$

$$\llbracket x := E \rrbracket^\# M^\# = M^\# [x \mapsto (\llbracket E \rrbracket^\#(M^\#))]$$

$$\llbracket \text{input}(x) \rrbracket^\# M^\# = M^\# [x \mapsto \top]$$

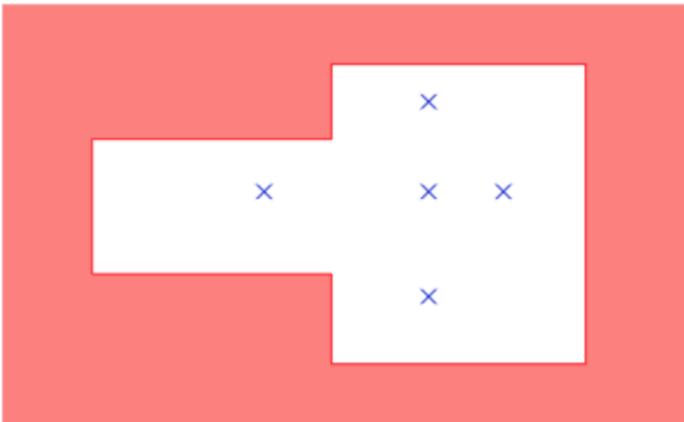
$$\llbracket \text{if } (B)\{C_0\} \text{ else } \{C_1\} \rrbracket^\#(M^\#) = \llbracket C_0 \rrbracket^\# \mathcal{F}_B^\#(M) \cup^\# \llbracket C_1 \rrbracket^\# \mathcal{F}_{\neg B}^\#(M^\#)$$

$$\llbracket \text{while}(B)\{C\} \rrbracket^\#(M^\#) = \mathcal{F}_{\neg B}^\#(\text{abs_iter}(\llbracket C \rrbracket^\# \mathcal{F}_B^\#, M^\#))$$

Design the
widening \triangledown

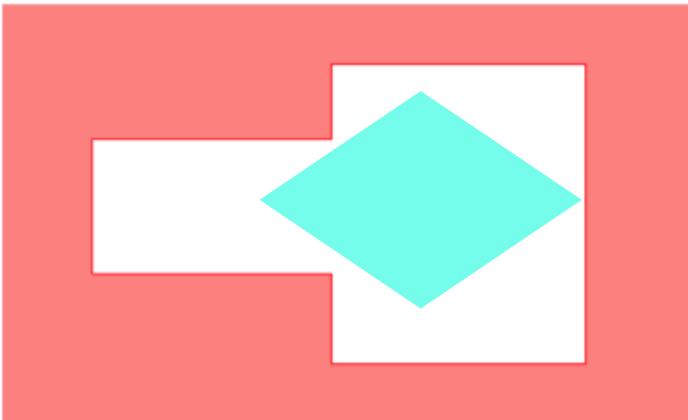
Theorem The computation of $\llbracket C \rrbracket^\# M^\#$ terminates and $\llbracket C \rrbracket \gamma(M^\#) \subseteq \gamma(\llbracket C \rrbracket^\#(M^\#))$

Using analysis' results



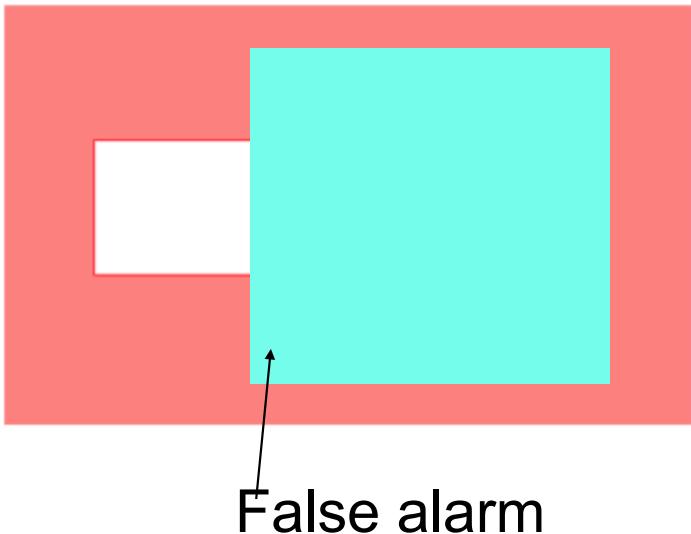
The program is correct

Using analysis' results



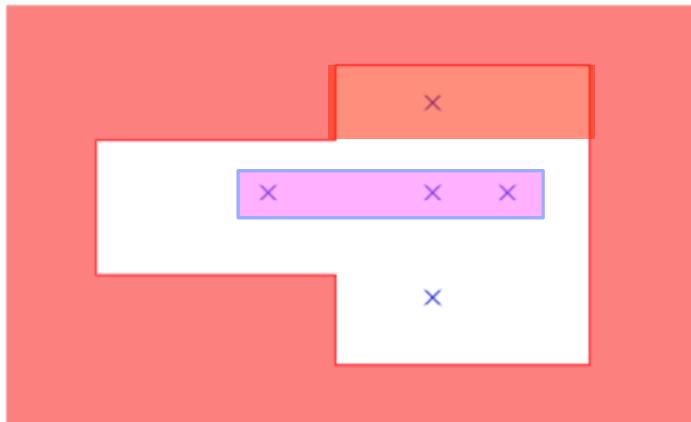
The program is correct and
our approximation can prove it

Using analysis' results



The program is correct and
our approximation can't prove it

Unsound analysis



The program is not correct and our approximation says it is correct



Trace-based operational semantics



```
p0 : while isEven(x) {  
    p1 : x = x div 2;  
}  
p2 : x = 4 * x;  
p3 : exit
```

The operational semantics updates a program-point, storage-cell pair, pp, x , using these four transition rules:

$$p_0, 2n \rightarrow p_1, 2n$$

$$p_1, n \rightarrow p_0, n/2$$

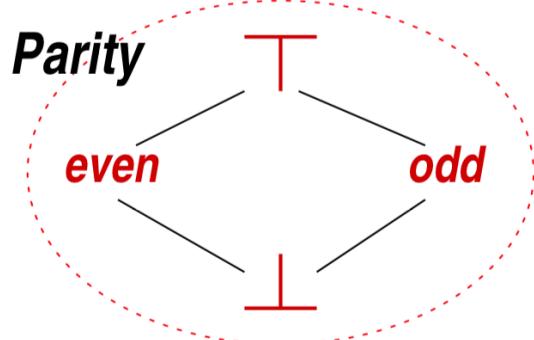
$$p_0, 2n + 1 \rightarrow p_2, 2n + 1$$

$$p_2, n \rightarrow p_3, 4n$$

A program's operational semantics is written as a trace:

$$p_0, 12 \rightarrow p_1, 12 \rightarrow p_0, 6 \rightarrow p_1, 6 \rightarrow p_0, 3 \rightarrow p_2, 3 \rightarrow p_3, 12$$

The parity domain



$$\gamma : \text{Parity} \rightarrow \mathcal{P}(\text{Int})$$

$$\gamma(\text{even}) = \{\dots, -2, 0, 2, \dots\}$$

$$\gamma(\text{odd}) = \{\dots, -1, 1, 3, \dots\}$$

$$\gamma(T) = \text{Int}, \quad \gamma(\perp) = \{\}$$

$$\alpha : \mathcal{P}(\text{Int}) \rightarrow \text{Parity}$$

$$\alpha(S) = \sqcup \{\beta(v) | v \in S\}, \text{ where } \beta(2n) = \text{even} \text{ and } \beta(2n + 1) = \text{odd}$$

The abstract transition rules are synthesized from the orginals:

$$p_i, a \longrightarrow p_j, \alpha(v'), \text{ if } v \in \gamma(a) \text{ and } p_i, v \longrightarrow p_j, v'$$

This recipe ensures that every transition in the original, “concrete” semantics is simulated by one in the abstract semantics.

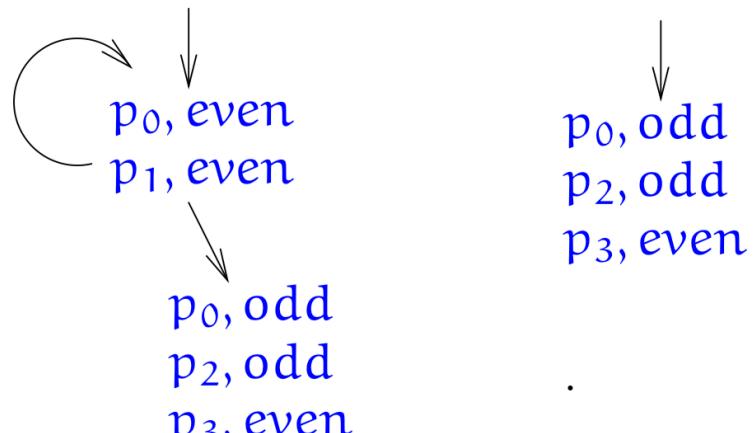
The abstraction rules

 $p_0, 2n \rightarrow p_1, 2n$ $p_0, 2n + 1 \rightarrow p_2, 2n + 1$ $p_1, n \rightarrow p_0, n/2$ $p_2, n \rightarrow p_3, 4n$

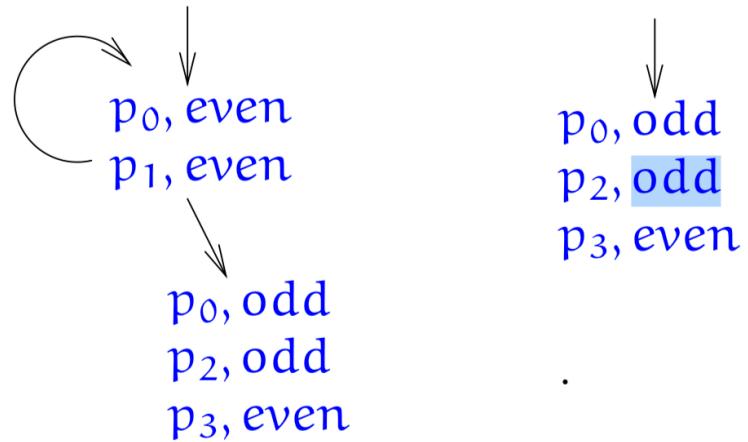
```
p0 : while isEven(x) {  
    p1 : x = x div 2;  
}  
p2 : x = 4 * x;  
p3 : exit
```

 $p_0, \text{even} \rightarrow p_1, \text{even}$ $p_0, \text{odd} \rightarrow p_2, \text{odd}$ $p_1, \text{even} \rightarrow p_0, \text{even}$ $p_1, \text{even} \rightarrow p_0, \text{odd}$ $p_2, \text{a} \rightarrow p_3, \text{even}$

Two trace trees cover the full range of inputs:



The interpretation of the program's semantics with the abstract values is an *abstract interpretation*:



We conclude that

- ◆ if the program terminates, x is even-valued
- ◆ if the input is odd-valued, the loop body, p_1 , will not be entered

Due to the loss of precision, we can not decide termination for almost all the even-valued inputs. (Indeed, only 0 causes nontermination.)

Another example: array bounds using intervals

Integer variables receive values from the *interval domain*,

$$I = \{[i, j] \mid i, j \in \text{Int} \cup \{-\infty, +\infty\}\}.$$

We define $[a, b] \sqcup [a', b'] = [\min(a, a'), \max(b, b')]$.

```
int a = new int[10];
i = 0; // i = [0,0]
while (i < 10) {
    ... a[i] ...
    i = i + 1;
}
```

Annotations:

- $i = [0,0]$ (initial value)
- p_1 : $i = [0,0] \sqcup [-\infty, 9] = [0,0]$ (loop invariant)
- $i = [0,0] \sqcup [1,1] \sqcup [-\infty, 9] = [0,1]$ (loop invariant after first iteration)
- p_2 : $i = [1,1]$ (loop invariant)
- $i = [1,1] \sqcup [2,2] = [1,2]$ (loop invariant after second iteration)

at $p_1 : [0..9]$

At convergence, i 's ranges are at $p_2 : [1..10]$

at loop exit : $[1..10] \sqcap [10, +\infty] = [10, 10]$

Constant Propagation analysis

```
p0 : x = 1; y = 2;  
p1 : while (x < y + z)  
      p2 : x = x + 1;  
      }  
p3 : exit
```

where $m + n$ is interpreted

$k_1 + k_2 \rightarrow \text{sum}(k_1, k_2),$

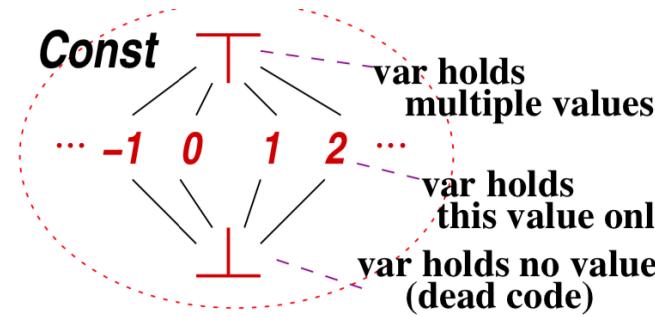
$\top \neq k_i \neq \perp, i \in 1..2$

$\top + k \rightarrow \top$

$k + \top \rightarrow \top$

Let $\langle u, v, w \rangle$ abbreviate

$\langle x : u, y : v, z : w \rangle$



Abstract trace: $p_0, \langle \top, \top, \top \rangle$

$p_1, \langle 1, 2, \top \rangle$

$\downarrow \Rightarrow p_3, \langle 1, 2, \top \rangle$

$p_2, \langle 1, 2, \top \rangle$

$p_1, \langle 2, 2, \top \rangle$

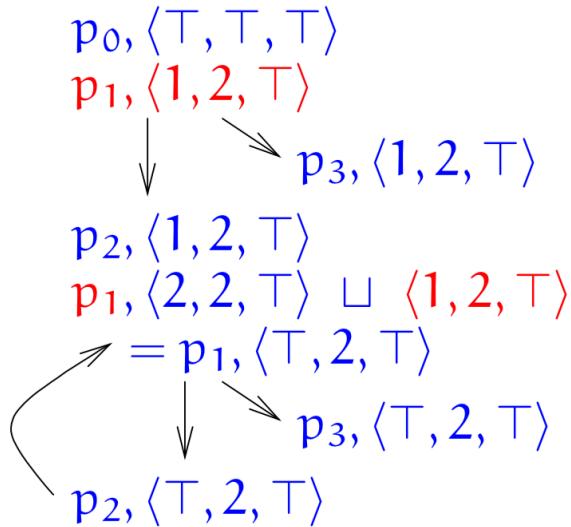
$\downarrow \Rightarrow p_3, \langle 2, 2, \top \rangle$

$p_2, \langle 2, 2, \top \rangle$

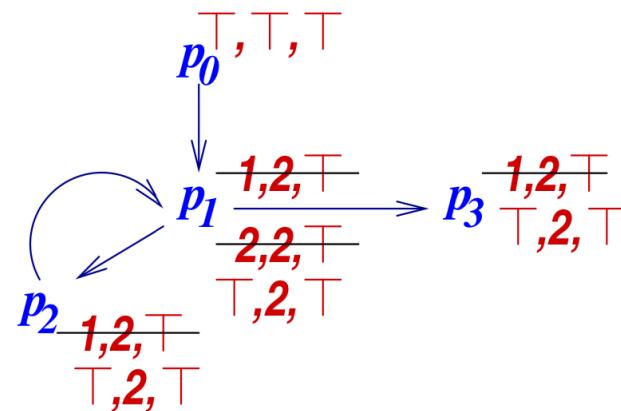
$p_1, \langle 3, 2, \top \rangle$

$\downarrow \Rightarrow \dots$

An acceleration is needed for finite convergence



Drawn as a data-flow analysis:



The analysis tells us to replace y at p_1 by 2:

$p_0 : x = 1; y = 2;$
 $p_1 : \text{while } (x < \cancel{y} + z) \{$
 $p_2 : x = x + 1;$
 $\}$
 $p_3 : \text{exit}$

2

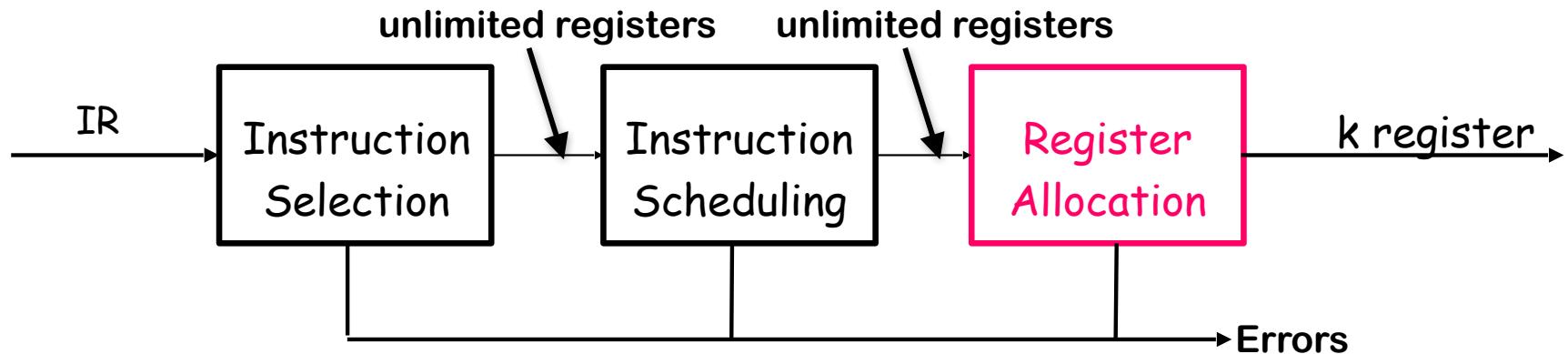
16

↑
TODO

Local Register Allocation

Register Allocation

Part of the compiler's back end



Critical properties

- Produce correct code that uses no more than k registers
- Minimize added work from loads and stores that spill values
- Minimize space used to hold spilled values
- Operate efficiently
 $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Spill code: Loads and Stores
inserted by the register allocator

Notation: The literature on register allocation consistently uses k as the number of registers available on the target system.

Register Allocation

r0 holds base address for local variables

@x is constant offset of x from r0

Consider a fragment of assembly code (or ILOC)

```
loadI  2      => r1    // r1 ← 2
loadAI r0, @b  => r2    // r2 ← b
mult   r1, r2  => r3    // r3 ← 2 · b
loadAI r0, @a  => r4    // r4 ← a
sub    r4, r3  => r5    // r5 ← a - (2 · b)
```

From the allocation perspective, these registers are virtual or pseudo-registers

The Problem

- At each instruction, decide which values to keep in registers
 - Note: each pseudo-register in the example is a value
- Simple if $|values| \leq |registers|$
- Harder if $|values| > |registers|$
- The compiler must automate this process

Register Allocation

The Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
 - No transformations (leave that to optimization & scheduling)
- Minimize inserted code – both dynamic & static measures
- Make good use of any extra registers

Allocation versus assignment

- Allocation is deciding which values to keep in registers
 - Assignment is choosing specific registers for values
 - This distinction is often lost in the literature
- The compiler must perform both allocation & assignment

Background issues

- The register allocator takes as input a code that is almost completely compiled
- It has been scanned, parsed, checked, analysed, optimised, rewritten as target machine code, and, perhaps, scheduled
- Many previously made decisions influence the task of the allocator:
 - Memory-to-memory versus register-to- register memory model

Additional complexity:

- Allocation vs Assignment
- Register Classes

Values that can be kept in registers: Unambiguous values

- A value that can be accessed with just one name is unambiguous
- Only unambiguous value can be kept registers

```
int first, second, *p, *q;  
...  
first = *p; // store the value from the variable referred to by p in first  
*q = 3;     // assign to the variable referred to by q  
second = *p; // store the value from the variable referred to by p in second
```

- After the assignment of first the compiler can keep the value of *p in a register only if it is sure that q and p points to different memory locations

Alias analysis

Register-to -register vs. memory-to-memory

- With a register-to -register earlier phases in the compiler directly encode the knowledge about ambiguous memory references: with this model unambiguous values are kept into virtual registers
- In a register-to -register the code produced by the previous step is not legal
- In a memory-to-memory model, the code is legal before allocation; allocation improve performance
- In a memory-to-memory model the allocator does not have any knowledge and this can limit its ability

Allocation

Allocation is an hard problem that in its general formulation is NP-complete.

The allocation of a single basic block with one size data value can be done in polynomial time under strong hypothesis:

- each value have to be stored to memory at the end of its lifetime (no constant,...)
- the spilling of value has uniform cost

any additional complexity makes the problem NP-complete

Allocation vs. Assignment

- Once we have reduced the demand for registers,
the assignment can be done in polynomial time for a machine with
one kind of registers

Register Classes

- General purpose registers
 - Integer values and memory addresses
 - Floating-point registers (single and double precision)
 - On some architectures also condition code, predicate registers or branch target registers
-
- If the compiler uses different kind of registers for different kinds of data, it can allocate each class independently: the problem can be simplified
 - If the different kinds of data overlap, the compiler must allocate them together: the allocation can become more complex (single and double precision registers for floating-point)

Basic Blocks in Assembly Code (or ILOC)

Definition

- A basic block is a maximal length segment of straight-line (i.e., branch free) code

Importance

- Strongest facts are provable for branch-free code
- If any statement executes, they all execute
- Execution is totally ordered

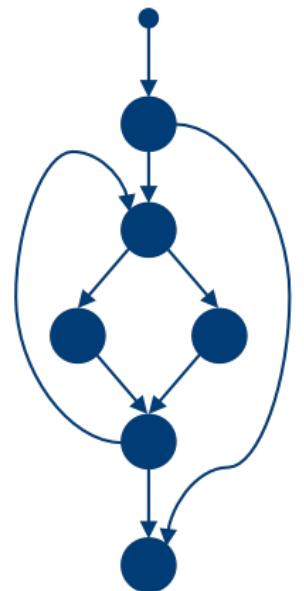
Ignore, for the moment, exceptions

Role of Basic Blocks in Optimization

- Many techniques for improving basic blocks
- Simplest problems
- Strongest methods

Local Register Allocation

- What is “local” ? (different from “regional” or “global”)
 - A local transformation operates on basic blocks
 - Many optimizations are done on a local scale or scope
- Does local allocation solve the problem?
 - It produces good register use inside a block
 - Inefficiencies can arise at boundaries between blocks
- How many passes can the allocator make?
 - This is an off-line problem
 - As many passes as it takes, within reason
 - You can do a fine job in a couple of passes



Blocks in a Control-flow Graph (CFG)

Register Allocation

Optimal register allocation is hard

Local Allocation

- Simplified cases $\Rightarrow O(n)$
- Real cases \Rightarrow NP-Complete

Local Assignment

- Single size, no spilling $\Rightarrow O(n)$
- Two sizes \Rightarrow NP-Complete

Global Allocation

- NP-Complete for 1 register
- NP-Complete for k registers
(most sub-problems are NPC, too)

Global Assignment

- NP-Complete

Real compilers face real problems

ILOC

- Pseudo-code for a simple, abstracted RISC machine
 - generated by the instruction selection process
- Simple, compact data structures

a - 2 × b

loadI	2	⇒ r ₁
loadAI	r ₀ , @b	⇒ r ₂
mult	r ₁ , r ₂	⇒ r ₃
loadAI	r ₀ , @a	⇒ r ₄
sub	r ₄ , r ₃	⇒ r ₅

Nearly assembly code

- simple three-address code
- RISC-like addressing modes
 - I, AI, AO
- unlimited virtual registers
(register-to-register vs memory-to-memory)

ILOC

- Pseudo-code for a simple, abstracted RISC machine
 - generated by the instruction selection process
- Simple, compact data structures

$a = 2 \times b$

loadI	2		r_1
loadAI	r_0	$@b$	r_2
add	r_1	r_2	r_3
loadAI	r_0	$@a$	r_4
sub	r_4	r_3	r_5

Quadruples:

- table of $k \times 4$ small integers
- simple record structure
- easy to reorder
- all names are explicit

Observations

The Register Allocator does not need to "understand" the code

- It needs to distinguish definitions from uses
 - Definitions might need to store a spilled value
 - Uses might need to load a spilled value
- ILOC makes definitions and uses pretty clear
 - The assignment arrow, \Rightarrow , separates uses from definitions
 - Except on the store operation, which uses all its register operands
 - store r8 \Rightarrow r1 // $\text{MEM(r1)} \leftarrow r8$
 - That is the point of the arrow!
- Your allocator needs to know, by opcode, how many definitions and how many uses it should see
 - Beyond that, the meaning of the ILOC is somewhat irrelevant to the allocator

Observations

A value is live between its definition and its uses

- Find definitions ($x \leftarrow \dots$) and uses ($y \leftarrow \dots x \dots$)
- From definition to last use is its live range
 - How does a second definition affect this?
- Can represent live range as an interval $[i, j]$ (in block)

Let MAXLIVE be the maximum, over each instruction i in the block, of the number of values (pseudo-registers) live at i .

- If $\text{MAXLIVE} \leq k$, allocation should be easy:
no need to reserve F registers for spilling
- If $\text{MAXLIVE} > k$, some values must be spilled to memory:
need to reserve F registers for spilling

Finding live ranges is harder in the global case

Concrete Example of MAXLIVE

Sample code sequence

loadI	1028	$\Rightarrow r1$	// $r1 \leftarrow 1028$
load	$r1$	$\Rightarrow r2$	// $r2 \leftarrow \text{MEM}(r1)$
mult	$r1, r2$	$\Rightarrow r3$	// $r3 \leftarrow 1028 \cdot y$
load	x	$\Rightarrow r4$	// $r4 \leftarrow x$
sub	$r4, r2$	$\Rightarrow r5$	// $r5 \leftarrow x - y$
load	z	$\Rightarrow r6$	// $r6 \leftarrow z$
mult	$r5, r6$	$\Rightarrow r7$	// $r7 \leftarrow z \cdot (x - y)$
sub	$r7, r3$	$\Rightarrow r8$	// $r8 \leftarrow z \cdot (x - y) - (1028 \cdot y)$
store	$r8$	$\Rightarrow r1$	// $\text{MEM}(r1) \leftarrow z \cdot (x - y) - (1028 \cdot y)$

Store uses this register & defines a memory location.

The code uses 1028 as both an address and as a constant in the computation.

The intent is to create a long live range for pedagogical purposes. Remember, the allocator does not need to understand the computation. It just needs to preserve the computation.

Concrete Example of MAXLIVE

Live ranges in the example

loadI	1028	$\Rightarrow r1$	// r1
load	r1	$\Rightarrow r2$	// r1 r2
mult	r1, r2	$\Rightarrow r3$	// r1 r2 r3
load	x	$\Rightarrow r4$	// r1 r2 r3 r4
sub	r4, r2	$\Rightarrow r5$	// r1 r3 r5
load	z	$\Rightarrow r6$	// r1 r3 r5 r6
mult	r5, r6	$\Rightarrow r7$	// r1 r3 r7
sub	r7, r3	$\Rightarrow r8$	// r1 r8
store	r8	$\Rightarrow r1$	//

Remember, r1 is a use,
not a definition

A pseudo-register is live
after an operation if it has
been defined & has a use in
the future

Concrete Example of MAXLIVE

Live ranges in the example

loadI	1028	$\Rightarrow r1$	// r1
load	r1	$\Rightarrow r2$	// r1 r2
mult	r1, r2	$\Rightarrow r3$	// r1 r2 r3
load	x	$\Rightarrow r4$	// r1 r2 r3 r4
sub	r4, r2	$\Rightarrow r5$	// r1 r3 r5
load	z	$\Rightarrow r6$	// r1 r3 r5 r6
mult	r5, r6	$\Rightarrow r7$	// r1 r3 r7
sub	r7, r3	$\Rightarrow r8$	// r1 r8
store	r8	$\Rightarrow r1$	//

Remember, r1 is a use,
not a definition

MAXLIVE is 4

Compute these "live" sets in a backward pass over the code.

Start with live as the empty set.

At each op, remove target & add operands

Local Allocation: Top-down Versus Bottom-up

Top-down allocator

- Work from external notion of what is important
- Assign registers in priority order
- Save some registers for the values relegated to memory

Bottom-up allocator

- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Handle all values uniformly

Top-down Allocator

The idea

- The most heavily used values should reside in a register
- Reserve registers for use in spills, say r registers

Algorithm

- Count the number of occurrences of each virtual register in the block (from 2 to $\text{maxlength}(\text{block})$)
 - Sort the registers according to the previous info
 - Allocate first $k - r$ values to registers
 - Rewrite code to reflect these choices
- { Move values with no register into memory
(add LOADs & STOREs)

Programmers applied this idea by hand in the 70's & early 80's

Top-down Allocator

How many registers must the allocator reserve?

- Need registers to compute spill addresses & load values
- Number depends on target architecture
 - Typically, must be able to load 2 values
- Reserve these registers for spilling

What if $k - r < |\text{values}| < k$?

- Remember that the underlying problem is NP-Complete
- The allocator can either
 - Check for this situation
 - Adopt a more complex strategy
 - Accept the fact that the technique is an approximation

Back to the Example

Top down (3 registers)

loadI	1028	$\Rightarrow r1$	// r1	r1 is used more often than r3
load	r1	$\Rightarrow r2$	// r1 r2	
mult	r1, r2	$\Rightarrow r3$	// r1 r2 r3	
load	x	$\Rightarrow r4$	// r1 r2 r3 r4	
sub	r4, r2	$\Rightarrow r5$	// r1 r3 r5	
load	z	$\Rightarrow r6$	// r1 r3 r5 r6	
mult	r5, r6	$\Rightarrow r7$	// r1 r3	r7
sub	r7, r3	$\Rightarrow r8$	// r1	r8
store	r8	$\Rightarrow r1$	//	

Note that this assumes that no extra register is needed for spilling

Back to the Example

Top down (3 registers, need 2 for operands)

loadI	1028	$\Rightarrow r1$	// r1
load	r1	$\Rightarrow r2$	// r1 r2
mult	r1, r2	$\Rightarrow r3$	// r1 r2 r3

r1 is used more often than r3

load	x	$\Rightarrow r4$	// r1 r2 r3 r4
sub	r4, r2	$\Rightarrow r5$	// r1 r3 r5
load	z	$\Rightarrow r6$	// r1 r3 r5 r6
mult	r5, r6	$\Rightarrow r7$	// r1 r3

spill r3

sub	r7, r3	$\Rightarrow r8$	// r1
store	r8	$\Rightarrow r1$	//

r8 restore r3

Note that this assumes that no extra register is needed for spilling

An Example

Top down (3 registers, need 2 for operands)

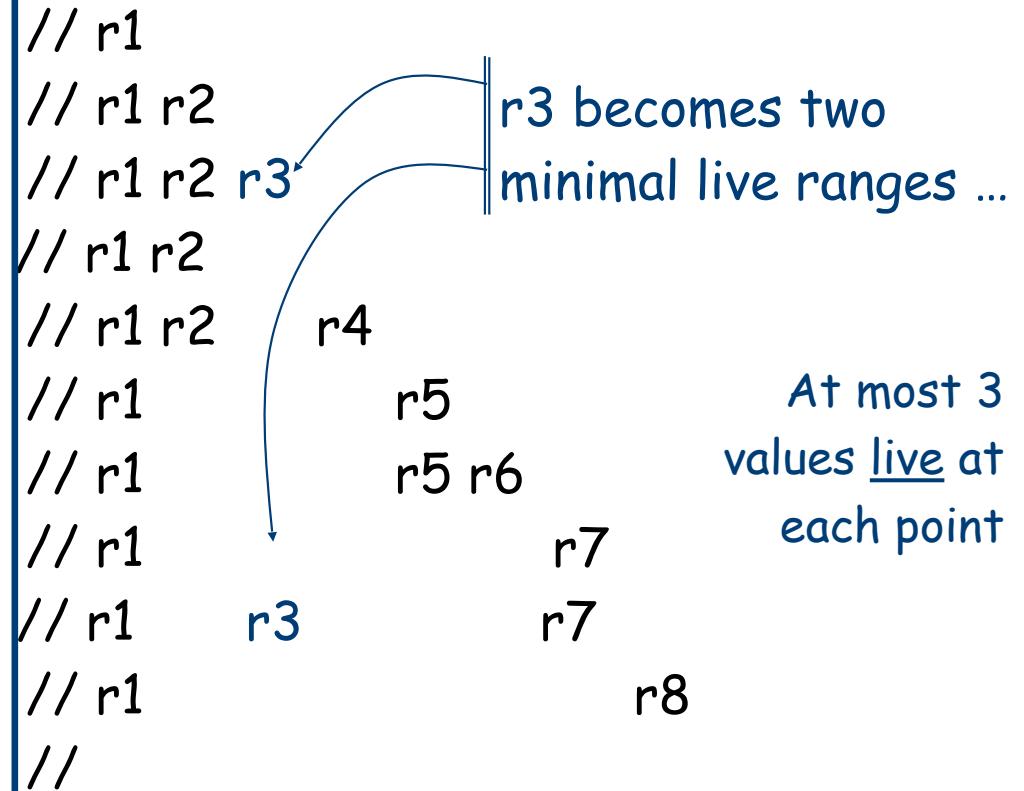
loadI	1028	$\Rightarrow r1$	// r1	
load	r1	$\Rightarrow r2$	// r1 r2	
mult	r1, r2	$\Rightarrow r3$	// r1 r2 r3	
store	r3	$\Rightarrow 16$	// r1 r2	r3 becomes two minimal live ranges ...
load	x	$\Rightarrow r4$	// r1 r2	r4
sub	r4, r2	$\Rightarrow r5$	// r1	r5
load	z	$\Rightarrow r6$	// r1	r5 r6
mult	r5, r6	$\Rightarrow r7$	// r1	r7
load	16	$\Rightarrow r3$	// r1	r3 r7
sub	r7, r3	$\Rightarrow r8$	// r1	r8
store	r8	$\Rightarrow r1$	//	

"spill" and "restore" become stores and loads

An Example

Top down (3 registers)

loadI	1028	$\Rightarrow r1$
load	r1	$\Rightarrow r2$
mult	r1, r2	$\Rightarrow r3$
store	r3	$\Rightarrow 16$
load	x	$\Rightarrow r4$
sub	r4, r2	$\Rightarrow r5$
load	z	$\Rightarrow r6$
mult	r5, r6	$\Rightarrow r7$
load	16	$\Rightarrow r3$
sub	r7, r3	$\Rightarrow r8$
store	r8	$\Rightarrow r1$



The two short versions of r3 each overlap with fewer values, which simplifies the allocation problem. Such “spilling” will (eventually) create a code where the allocator can succeed.

An Example

Top down (3 registers)

loadI	1028	$\Rightarrow r1$	// r1		
load	r1	$\Rightarrow r2$	// r1 r2		
mult	r1, r2	$\Rightarrow r3$	// r1 r2 r3		
store	r3	$\Rightarrow 16$	// r1 r2		
loadI	x	$\Rightarrow r4$	// r1 r2	r4	
sub	r4, r2	$\Rightarrow r5$	// r1		r5
loadI	z	$\Rightarrow r6$	// r1		r5 r6
mult	r5, r6	$\Rightarrow r7$	// r1		r7
load	16 $\Rightarrow r3$	// r1	<u>r3</u>	<u>r7</u>	possible delay
sub	r7, r3	$\Rightarrow r8$	// r1		r8
store	r8	$\Rightarrow r1$	//		

This code is slower than the original, but it works correctly on a target machine with only three (available) registers.
Correctness is a virtue.

Weakness of the top down approach to allocation

- A physical register is dedicated to a virtual register for an entire block

Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values used “soon” in registers

Algorithm (not optimal!):

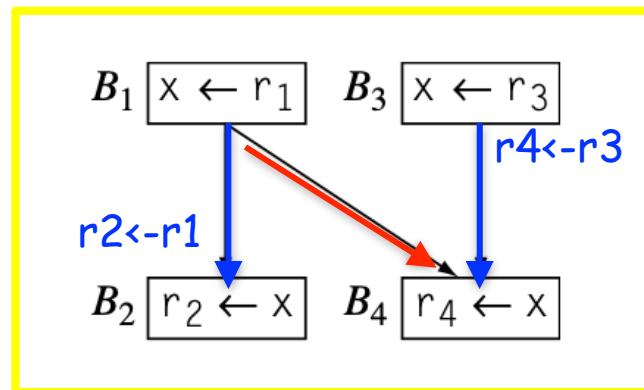
- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:

- Spill the value whose next use is farthest in the future
- Prefer clean values (not to be stored that are constant or values already in memory) to dirty values (that need to be stored).

From local algorithms to regional algorithms

- Extending local algorithms to regional ones can be difficult



- the only solution is to store back in memory the value of x at the end of B1 and B2
- while we could add register to register operation for blue arrow no possibility exists for the red arrow

17

Global Register Allocation via Graph Coloring

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Global Register Allocation

The Big Picture



Optimal global allocation is NP-Complete, under almost any assumptions.

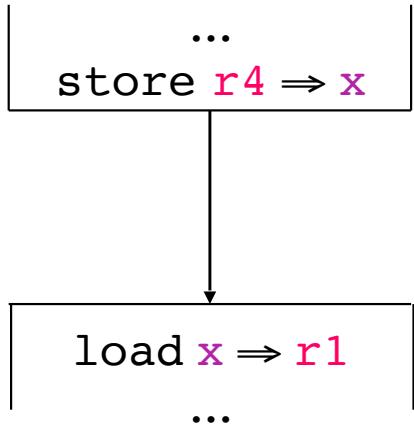
At each point in the code

- 1 Determine which values will reside in registers
- 2 Select a register for each such value

For local allocation we saw

- the frequency-count allocator (top down)
- the allocator based on distance to the next use (bottom up)

What Makes Global Register Allocation Hard?

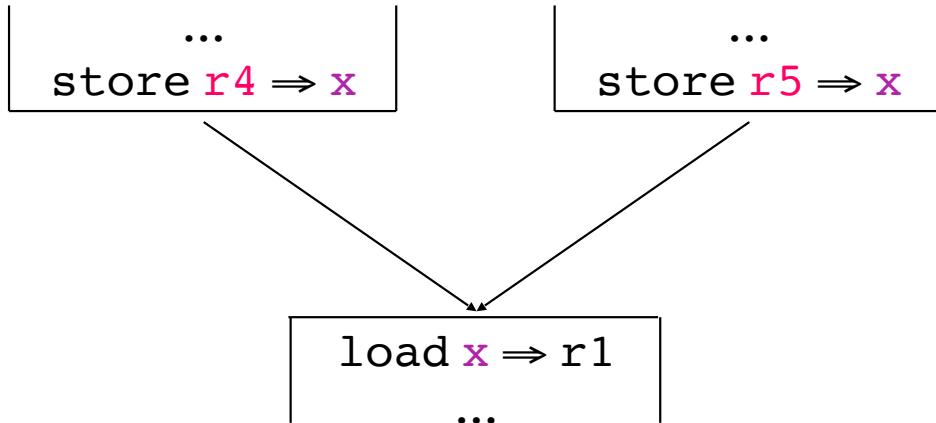


This is an assignment problem,
not an allocation problem !
If x is kept in the right register
we could just replace store-load
with a move

What's harder across multiple blocks?

- Could replace a load with a move
- Good assignment would obviate the move
- Must build a control-flow graph to understand inter-block flow
- Can spend an inordinate amount of time adjusting the allocation

What Makes Global Register Allocation Hard?



What if one block has x in a register, but not the other?

A more complex scenario

- Block with multiple predecessors in the control-flow graph
- Must get the “right” values in the “right” registers in each predecessor
- In a loop, a block can be its own predecessor

This adds tremendous complications

Global Register Allocation

Taking a global approach

- Abandon the distinction between local & global
- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Difference between two different allocations for the same code lies

- the number of loads and stores
- the placement operations (different blocks execute different times and this may vary at every run)

Global vs Local Register Allocation

- The structure of global live range can be more complex : a global live range is a web of definitions and uses
- In a local live rage all reference execute once per execution of the block . Thus the cost of spilling is uniform
- In a global allocator the cost of spilling depends on where the spilling occurs
 - Global allocators annotate each reference with an estimated execution frequency derived by static analysis or from profile data

Graph colouring paradigm

- 1 Build an interference graph G_I for the procedure
 - Computing LIVE is harder than in the local case
 - G_I is not an interval graph as in the local case
- 2 (try to) construct a k-coloring
 - Minimal coloring is NP-Complete
 - Spill placement becomes a critical issue
- 3 Map colors onto physical registers

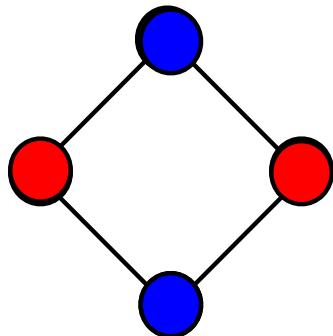
Graph Coloring

(A Background Digression)

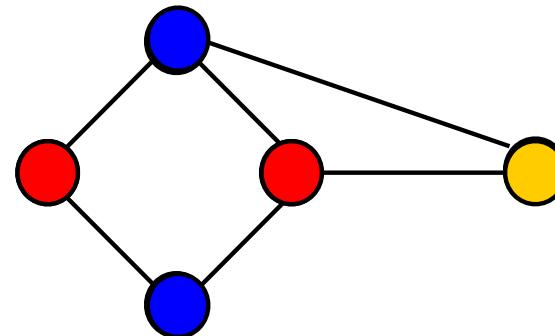
The problem

A graph G is said to be k -colorable iff the nodes can be labeled with integers $1 \dots k$ so that no edge in G connects two nodes with the same label

Examples



2-colorable



3-colorable

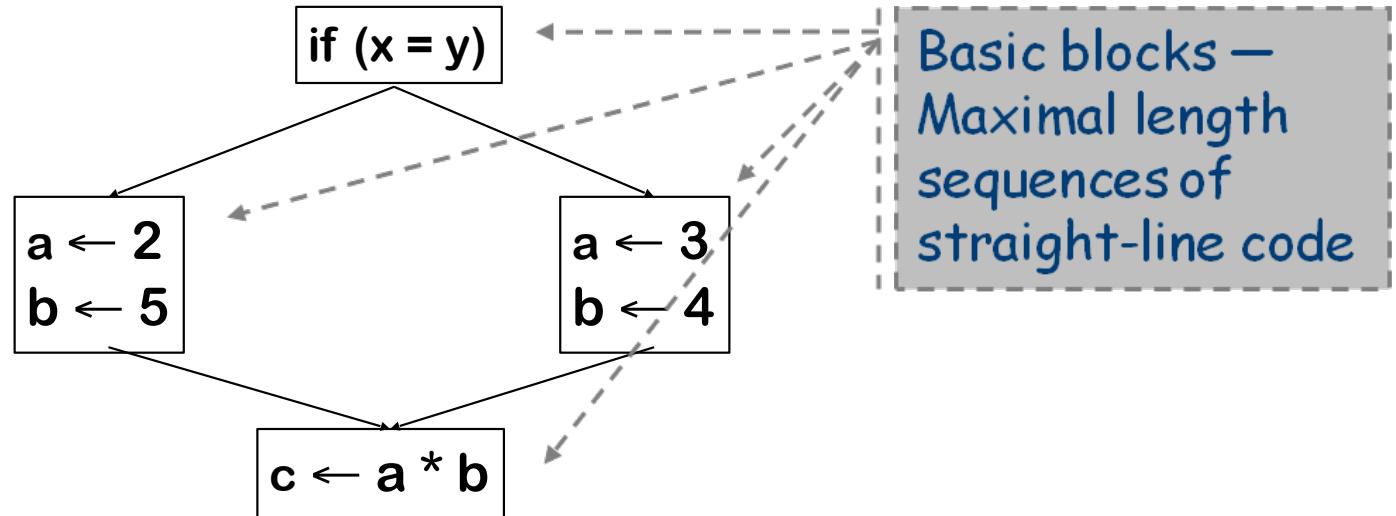
Each color can be mapped to a distinct physical register

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
 - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Static Single Assignment (SSA) form

- The main idea: each name is defined exactly once
- The name refers to the variable in some program point
- Encodes both control and value flow
- Introduce ϕ -functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
x0 ← ...
y0 ← ...
if (x0 >= k) goto next
loop:   x1 ←  $\phi(x_0, x_2)$ 
        y1 ←  $\phi(y_0, y_2)$ 
        x2 ← x1 + 1
        y2 ← y1 + x2
        if (x2 < k) goto loop
next:   ...
```

Strengths of SSA-form

- each use refers to a single definition
- (sometimes) faster algorithms

Building the Interference Graph

What is an “interference” ? (or conflict)

- Two values **interfere** if there exists an operation where both are simultaneously live
- If x and y interfere, they cannot occupy the same register

To compute interferences, we must know where values are “live”

The interference graph, $G_I = (N_I, E_I)$

- Nodes in G_I represent values, or live ranges
- Edges in G_I represent individual interferences
 - For $x, y \in N_I$, $\langle x, y \rangle \in E_I$ iff x and y interfere
- A k -coloring of G_I can be mapped into an allocation to k registers

Building the Interference Graph

To build the interference graph

1 Discover live ranges

- > Construct the **SSA form** of the procedure
- > At each ϕ -function, take the union of the arguments
- > Rename to reflect these new "live ranges"

2 Compute LIVE sets over live ranges for each block

- > Use an iterative data-flow solver
- > Solve equations for LIVE over domain of live range names

3 Iterate over each block, bottom-up

- > Track the current LIVE set
- > At each operation, add appropriate edges & update LIVE

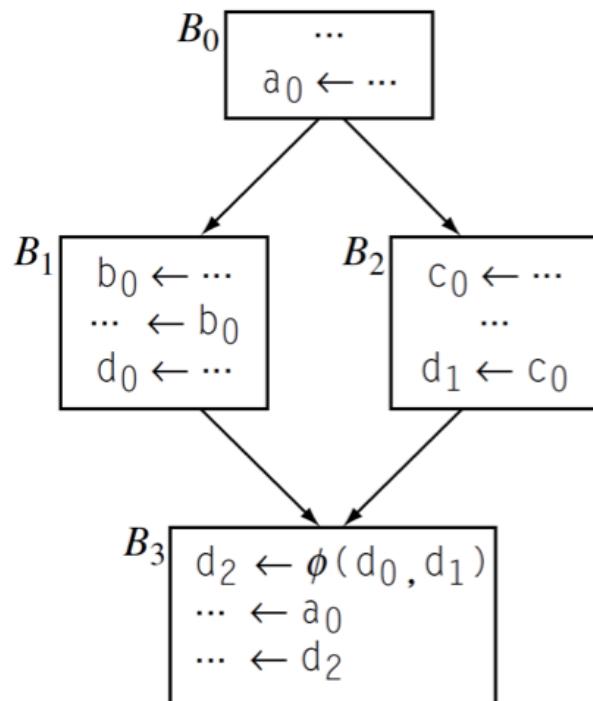
TODO



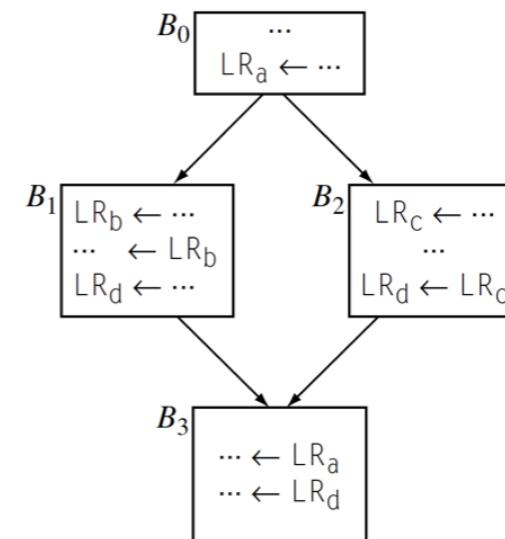
Point 1

Discover live ranges

- › Construct the **SSA form** of the procedure
- › At each ϕ -function, take the union of the arguments
- › Rename to reflect these new “live ranges”: arguments of the same phi functions has to be united together



(a) Code Fragment in
SSA Form



(b) Rewritten in Terms
of Live Ranges

Live ranges $\{LRa=a_0, LRb=b_0, LRC=c_0,$
 $LRd=d_0 \cup d_1 \cup d_2\}$

Point 2: Computing LIVE Sets

A value is **live** between its **definition** and its **uses**

- Find definitions ($x \leftarrow \dots$) and uses ($y \leftarrow \dots \times \dots$)
 - From definition to last use is its **live range**
 - How does a **second definition** affect this?
 - Can represent live range as an interval $[i, j]$ (in block)
-
- $LV_o(I) = \bigcup \{LV_o(I') \mid I' \text{ in } \text{post}(I)\}$
 - $LV_o(I) = (LV_o(I) \setminus \text{def}(B)) \cup \text{use}(B)$

$$\text{def}([x:=a]^l) = \{x\} \quad \text{and} \quad \emptyset \text{ elsewhere}$$

$$\text{use}([x:=a]^l) = \text{FV}(a)$$

$$\text{use}([b]^l) = \text{FV}(b) \quad \text{and} \quad \emptyset \text{ elsewhere}$$

$LV_o(I)$ are the variables live right before the block I

I $B: x := a$

$LV_o(I)$ are the variables live at the exit of the block I

Solve the equations using a fixed-point iterative scheme

Point 3:constructing the Interference graph

- LR_j interferences with LR_i if one is live at a definition of the other
- Once the allocator has built global variable ranges and annotate each basic block with its LiveOut set ($LV_{\bullet}(B)$), it can construct Interference graph with a linear pass over each block.

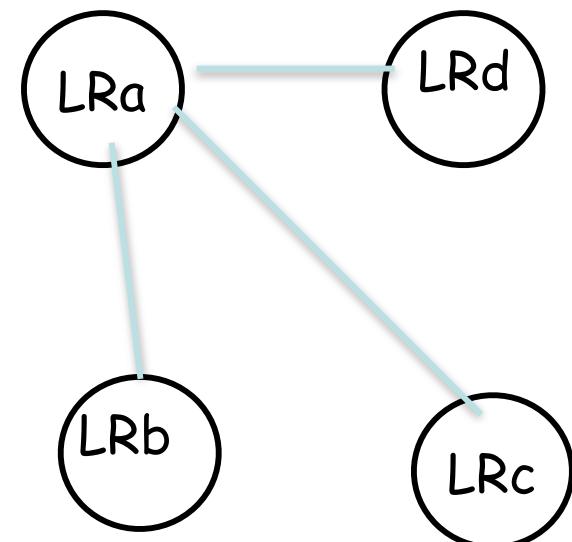
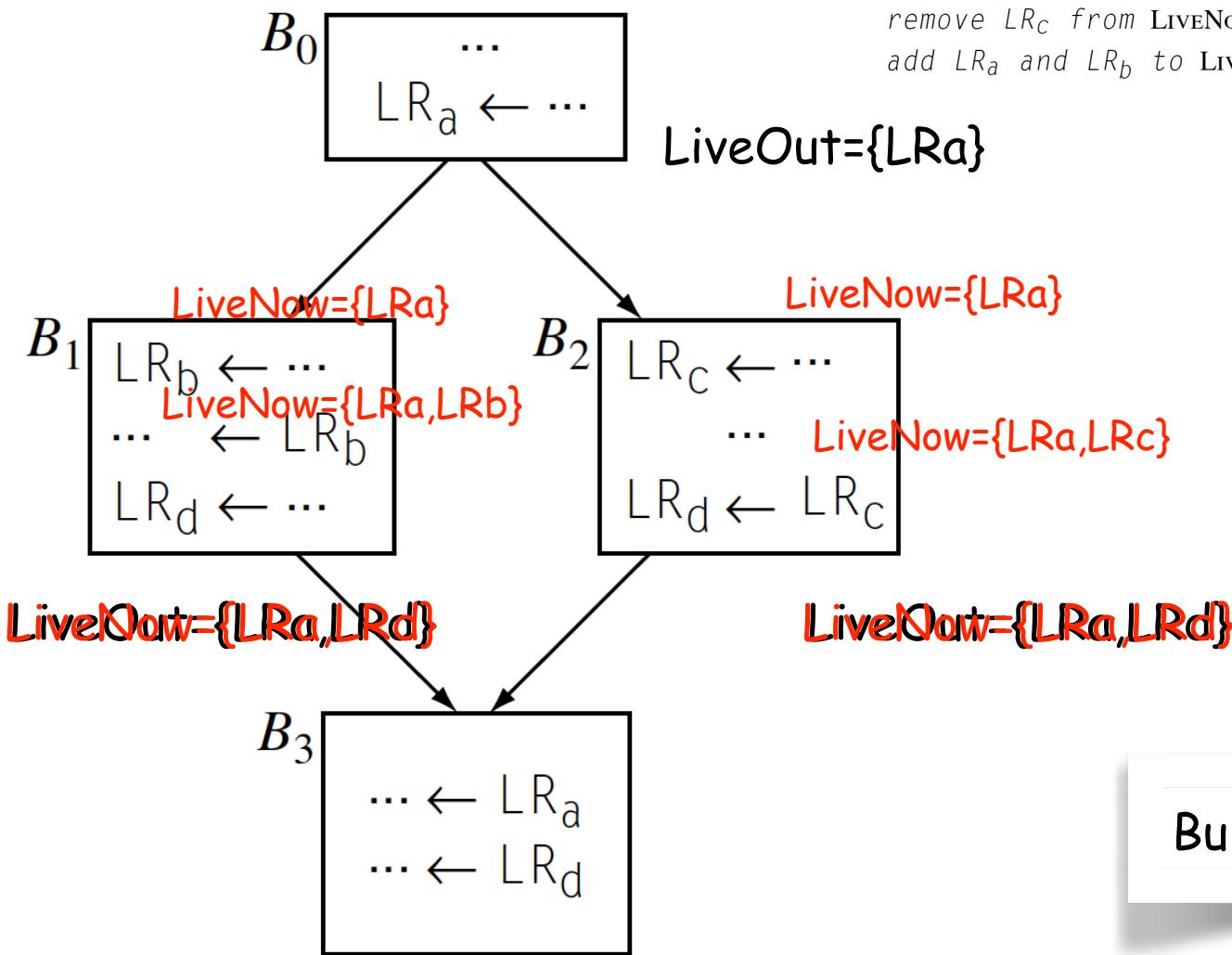
```
for each  $LR_j$ 
    create a node  $n_j \in N$ 
for each basic block  $b$ 
     $LIVENow \leftarrow LIVEOUT(b)$ 
    for each operation  $op_n, op_{n-1}, op_{n-2}, \dots, op_1$  in  $b$ 
        with form  $op_j LR_a, LR_b \Rightarrow LR_c$ 
        for each  $LR_j \in LIVENow$ 
            add  $(LR_c, LR_j)$  to  $E$ 
        remove  $LR_c$  from  $LIVENow$ 
        add  $LR_a$  and  $LR_b$  to  $LIVENow$ 
```

Inverse order!

```

for each  $LR_i$ 
    create a node  $n_i \in N$ 
for each basic block  $b$ 
     $LIVENow \leftarrow LIVEOut(b)$ 
    for each operation  $op_n, op_{n-1}, op_{n-2}, \dots, op_1$  in  $b$ 
        with form  $op_i : LR_a, LR_b \Rightarrow LR_c$ 
            for each  $LR_j \in LIVENow$ 
                add  $(LR_c, LR_j)$  to  $E$ 
            remove  $LR_c$  from  $LIVENow$ 
            add  $LR_a$  and  $LR_b$  to  $LIVENow$ 

```



Building interference Graph!

Account for Execution Frequency

The compiler annotates each block with estimated execution counts

These informations can be derived from

- profile data or from heuristics
- fixed assumptions, for example, a loop executes 10 times,
an unpredictable if-then-else divides by 2

To estimate the **cost of spilling** a single reference the allocator adds the cost of the address and memory operation and multiply by frequencies

For each live range it sum up the cost of individual references

Building an allocator

To build an allocator based on graph coloring on the interference graph, the compiler writer needs two additional mechanisms:

- the allocator needs an efficient technique to discover a k-coloring (remember finding is NP-complete)
 - Register allocator uses fast approximations that are not guaranteed to find a k-coloring
- The allocator needs a strategy that handles the case no color remain for a specific live range
 - The allocator chooses one or more live range to spill and reconsider the problem.

Now the interference graph may be colorable!

Observation on Coloring for Register Allocation

- Suppose you have k registers (not all the physical ones: some dedicated to keep, e.g. base addresses)—look for a k coloring
- Any vertex n that has fewer than k neighbours in the interference graph ($n^\circ < k$) can **always** be colored!
 - Pick any color not used by its neighbours — there must be one

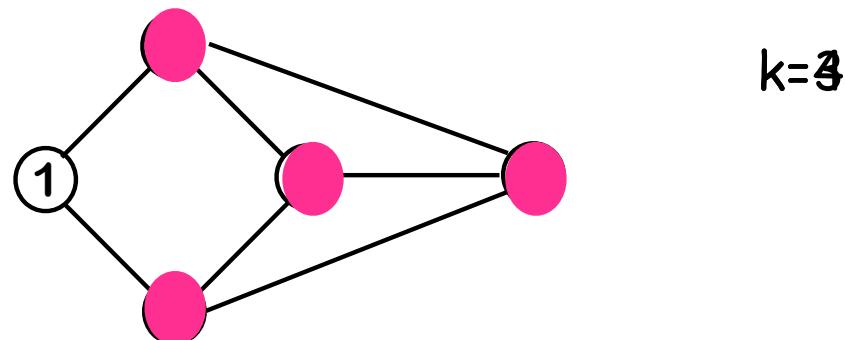
Top-Down Coloring

The Big Picture

- Use high-level priorities to rank live ranges
- Allocate registers for them in priority order
- Use coloring to assign specific registers to live ranges

Improving the Top-Down algorithm

1. Rank all live ranges with their estimated runtime saving,
(analogous of the spilling cost)
2. Separate constrained from unconstrained live ranges
 - > A live range is **constrained** if it has $\geq k$ neighbours in G_I



3. Constrained live ranges are coloured first in rank order

Handling Spills

- When the top down allocator encounters a live range that cannot be coloured it spills the live range to change the problem.
- Since the all previously coloured live ranges were ranked higher than the uncoloured one, it spills the uncoloured one.
- It could think of uncolor some of the previous one but it must exercise care to avoid the full cost of backtracking
- After the spilling the problem becomes easier and a new interference graph can be constructed.

Bottom-Up global allocator

- Ideas behind Chaitin's algorithm:
 - Pick any vertex n such that $n^{\circ} < k$ and put it on the stack
 - Remove that vertex and all edges incident from the interference graph
 - This may make additional nodes have fewer than k neighbours
 - At the end, if some vertex n still has k or more neighbours, then spill the live range associated with n
 - Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbour

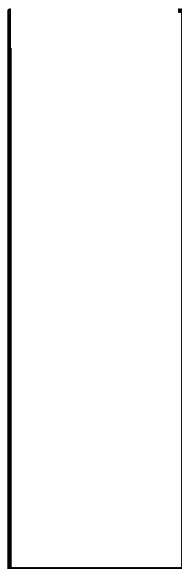
Chaitin's Algorithm

1. While \exists vertices with $< k$ neighbours in G_I
 - > Pick any vertex n such that $n^o < k$ and put it on the stack
 - > Remove that vertex and all edges incident to it from G_I

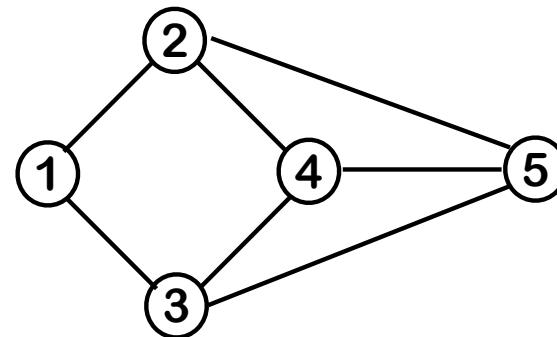
Lowers degree of
 n 's neighbours
2. If G_I is non-empty (all vertices have k or more neighbours) then:
 - > Pick a vertex n (using some heuristic) and spill the live range associated with n
 - > Remove vertex n from G_I , along with all edges incident to it and put it on the "spill list"
 - > If this causes some vertex in G_I to have fewer than k neighbours, then go to step 1; otherwise, repeat step 2
3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again
4. Otherwise, successively pop vertices off the stack and color them in the lowest color not used by some neighbour

Chaitin's Algorithm in Practice

3 Registers



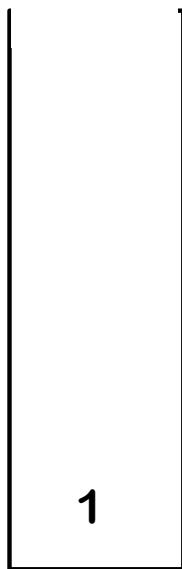
Stack



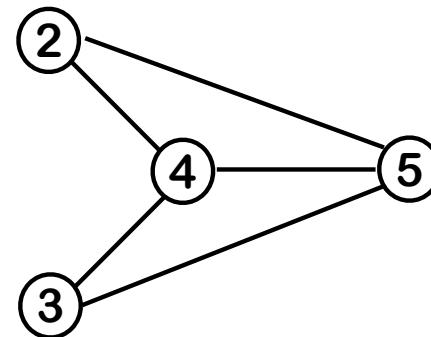
1 is the only node with degree < 3

Chaitin's Algorithm in Practice

3 Registers



Stack



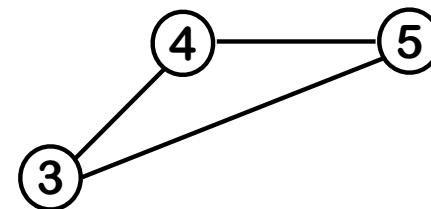
Now, 2 & 3 have degree < 3

Chaitin's Algorithm in Practice

3 Registers



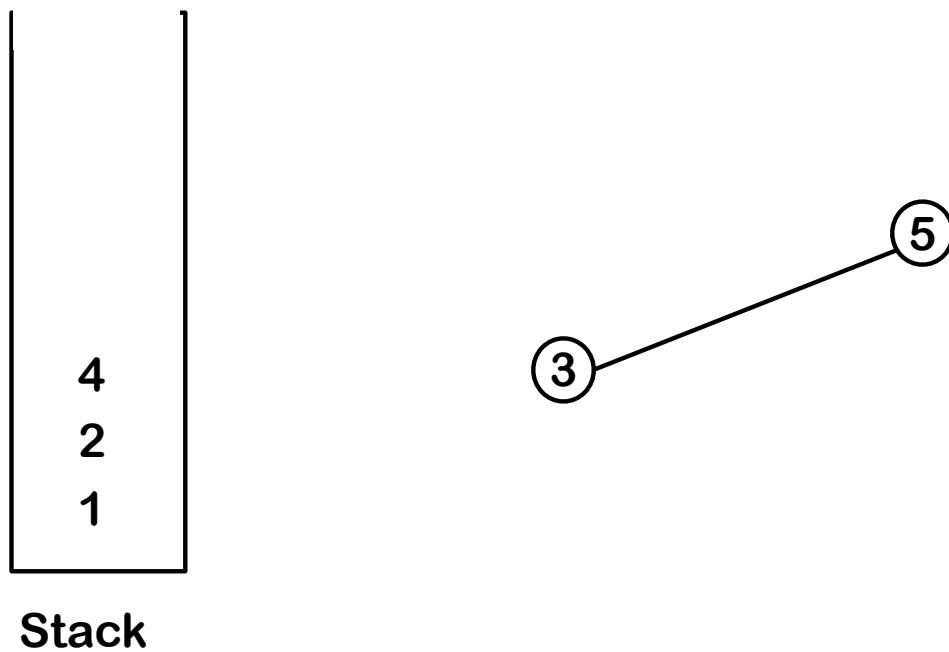
Stack



Now all nodes have degree < 3

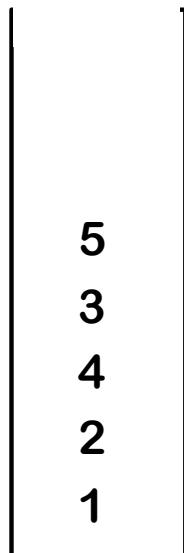
Chaitin's Algorithm in Practice

3 Registers

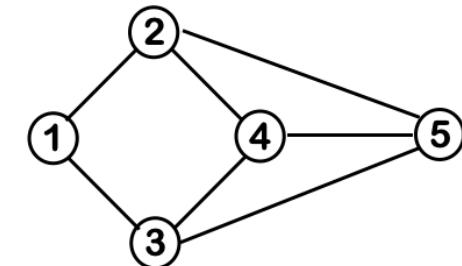


Chaitin's Algorithm in Practice

3 Registers



Stack

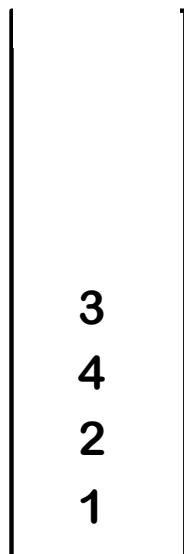


Colors:

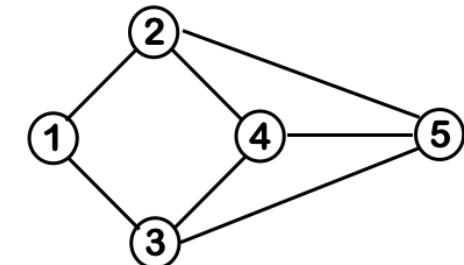


Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:



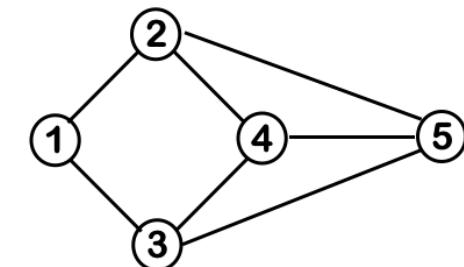
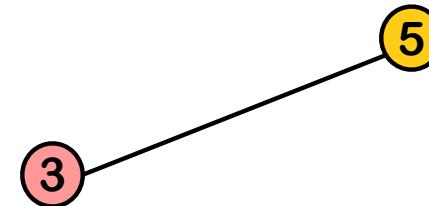
5

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

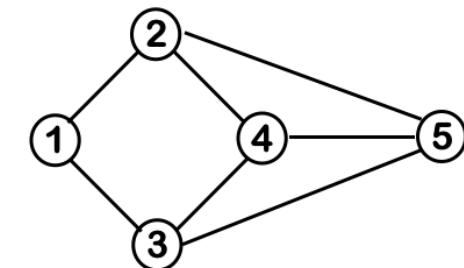
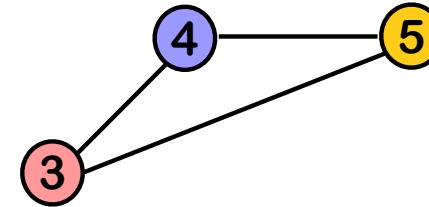


Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1:

2:

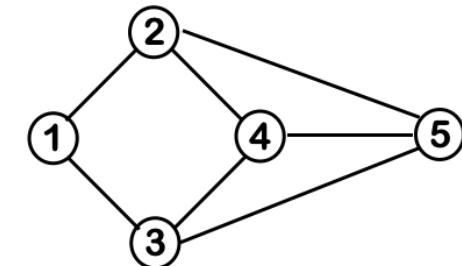
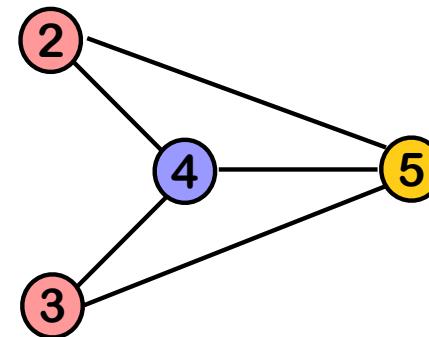
3:

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1:

2:

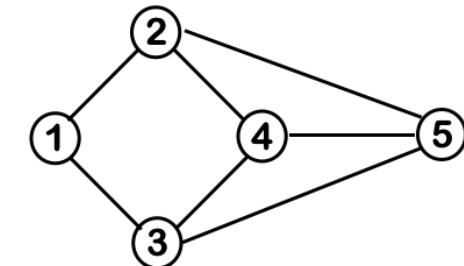
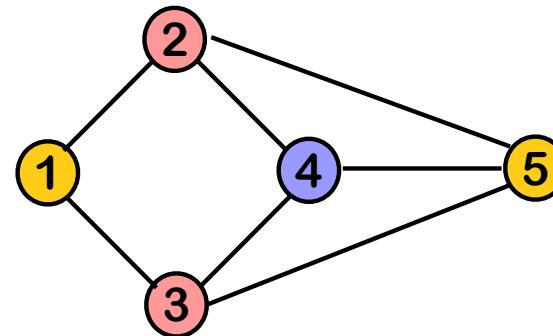
3:

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1:

2:

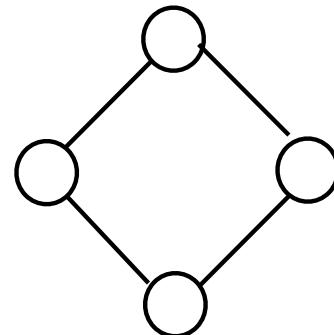
3:

Improvement in Coloring Scheme

Optimistic Coloring

- If Chaitin's algorithm reaches a state where every node has k or more neighbours, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
 - When you pop it off, a color might be available for it!

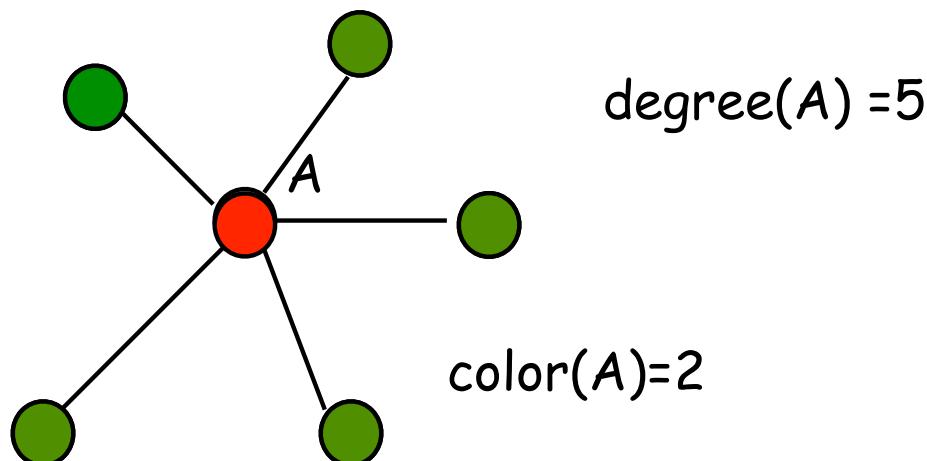
2 Registers:



Chaitin's algorithm
immediately spills one
of these nodes

Improvement in Coloring Scheme

- A node n might have $k+2$ neighbours, but those neighbours might only use $3 (< k)$ colors
- Degree is a loose upper bound on colorability



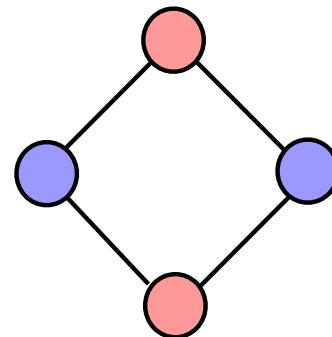
Improvement in Coloring Scheme

Optimistic Coloring

- If Chaitin's algorithm reaches a state where every node has k or more neighbours, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
 - When you pop it off, a color might be available for it!

2 Registers:

2-Colorable



Briggs algorithm finds
an available color

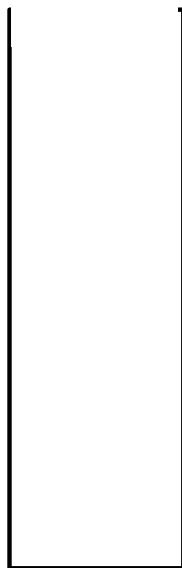
- For example, a node n might have $k+2$ neighbours, but those neighbours might only use just one color (or any number $< k$)
 - Degree is a loose upper bound on colorability

Chaitin-Briggs Algorithm

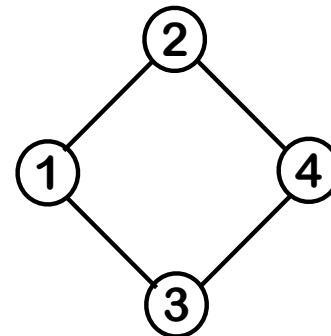
1. While \exists vertices with $< k$ neighbours in G_I
 - > Pick any vertex n such that $n^{\circ} < k$ and put it on the stack
 - > Remove that vertex and all edges incident to it from G_I
 - This action often creates vertices with fewer than k neighbours
2. If G_I is non-empty (all vertices have k or more neighbours) then:
 - > Pick a vertex n (using some **heuristic condition, spill metric as cost/degree**), push n on the stack and remove n from G_I , along with all edges incident to it
 - > If this causes some vertex in G_I to have fewer than k neighbours, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbour
 - > If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it, and restart at step 1

Chaitin-Briggs in Practice

2 Registers



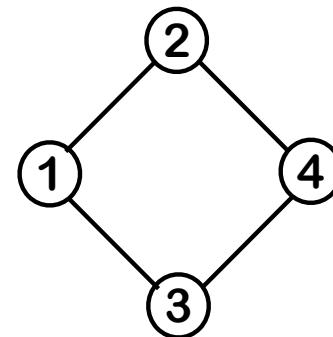
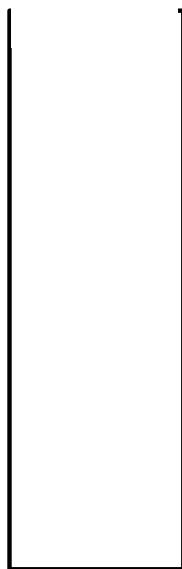
Stack



- No node has degree < 2
- Chaitin would spill a node
 - Briggs picks the same node & stacks it

Chaitin-Briggs in Practice

2 Registers

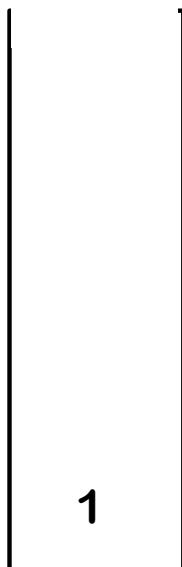


Stack

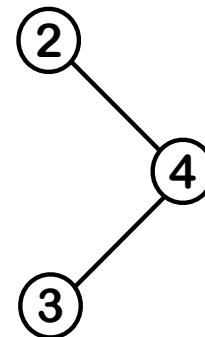
Pick a node, say 1

Chaitin-Briggs in Practice

2 Registers



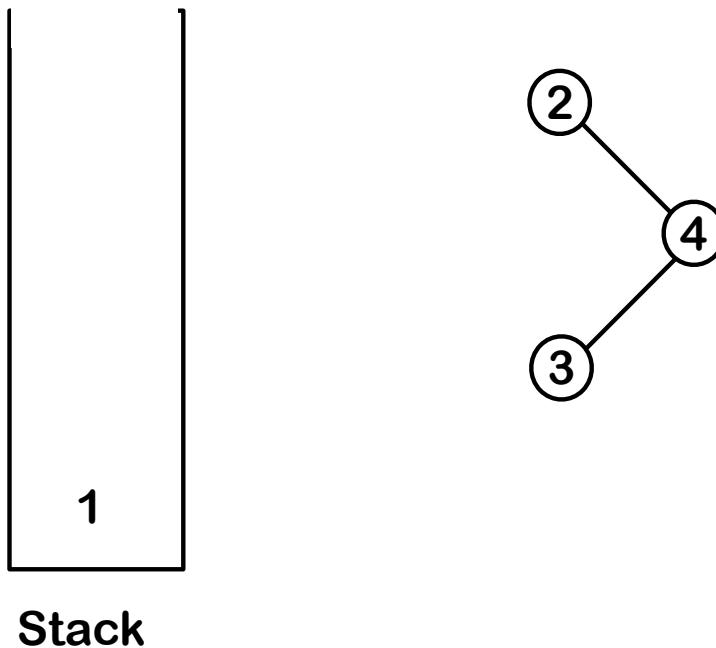
Stack



Pick a node, say 1

Chaitin-Briggs in Practice

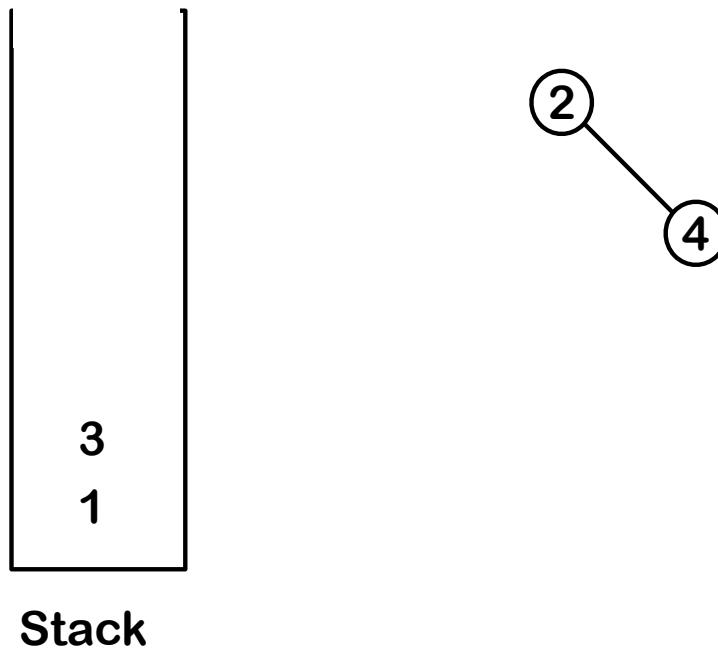
2 Registers



Now, both 2 & 3 have degree < 2
Pick one, say 3

Chaitin-Briggs in Practice

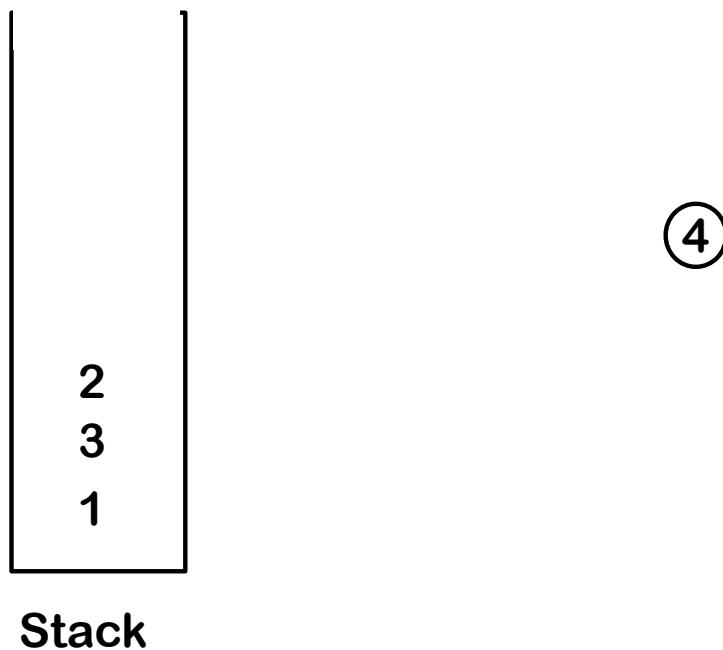
2 Registers



Both 2 & 4 have degree < 2.
Take them in order 2, then 4.

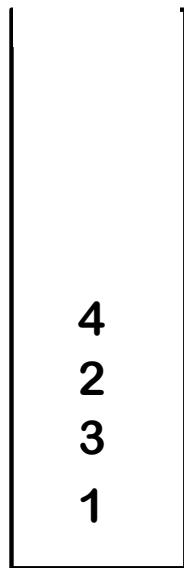
Chaitin-Briggs in Practice

2 Registers

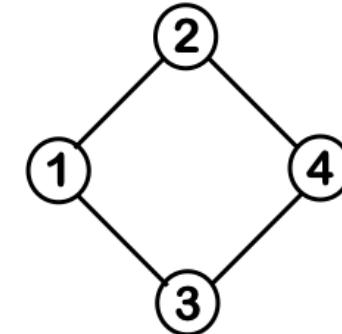


Chaitin-Briggs in Practice

2 Registers



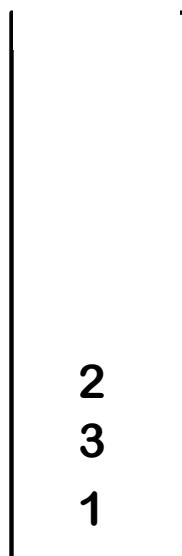
Stack



Now, rebuild the graph

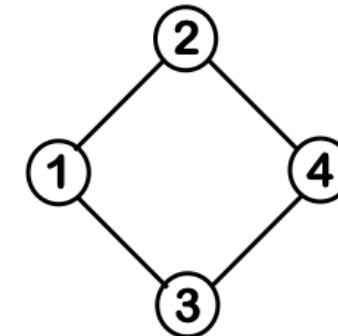
Chaitin-Briggs in Practice

2 Registers



Stack

4



Colors:

1:

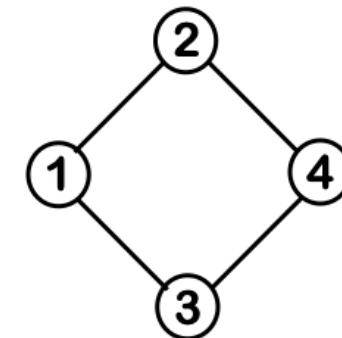
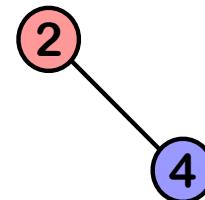
2:

Chaitin-Briggs in Practice

2 Registers



Stack

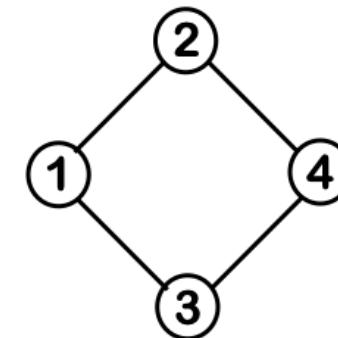
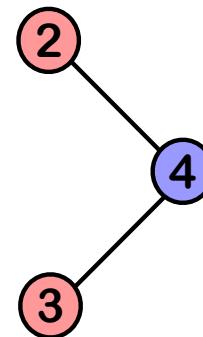


Colors:

- 1:
- 2:

Chaitin-Briggs in Practice

2 Registers

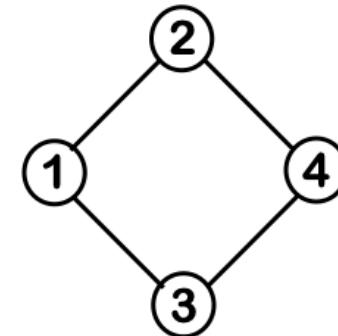


Colors:

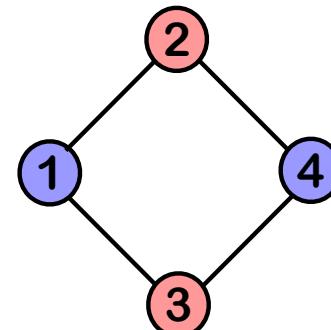
- 1: 
- 2: 

Chaitin-Briggs in Practice

2 Registers



Stack



Colors:

1:

2:

Comparing Top-Down and Bottom-Up approach

- Top-Down constrained nodes first
- Bottom-Up unconstrained nodes first and in this way some constrained becomes unconstrained
- No clear way to compare the results

Advanced Topics in Global Allocation

Coalescing Copies I

- To reduce the degree the compiler writer can use the interference graph to determine when two live ranges that are connected by a copy can be coalesced or combined.

i2i $LR_1 \Rightarrow LR_2$ Copy from register to register

- If LR_1 and LR_2 do not otherwise interfere, the operation can be eliminated and all references to LR_2 can be rewritten to use LR_1

Several advantages:

- It eliminates the copy operation
- It reduces the degree of any LR that interfered with both LR_1 and LR_2

Coalescing Copies II

add $LR_t, LR_u \Rightarrow LR_a$ a
...
i2i $LR_a \Rightarrow LR_b$ b
i2i $LR_a \Rightarrow LR_c$ c
...
add $LR_b, LR_w \Rightarrow LR_x$
add $LR_c, LR_y \Rightarrow LR_z$

(a) Before Coalescing

Both copy operations
are candidate for coalescing!

add $LR_t, LR_u \Rightarrow LR_{ab}$ ab
...
i2i $LR_{ab} \Rightarrow LR_c$ c
...
add $LR_{ab}, LR_w \Rightarrow LR_x$
add $LR_c, LR_y \Rightarrow LR_z$

(b) After Coalescing LR_a and LR_b

- Even if LR_a overlaps both LR_b and LR_c , it interferes with neither of them because **the source and destination of a copy do not interfere**

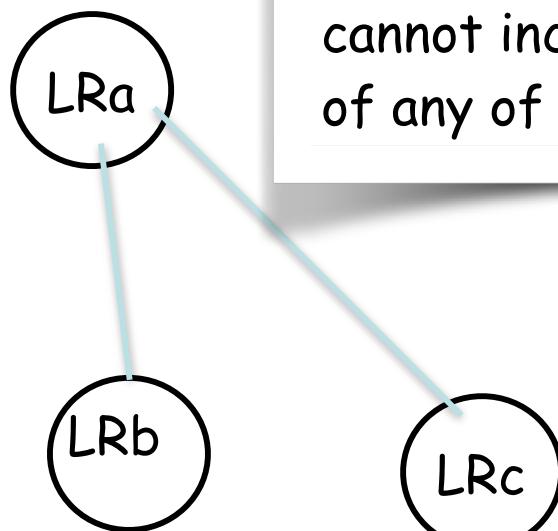
Coalescing Copies III

add $LR_t, LR_u \Rightarrow LR_a$ a
...
i2i $LR_a \Rightarrow LR_b$ b
i2i $LR_a \Rightarrow LR_c$ c
...
add $LR_b, LR_w \Rightarrow LR_x$
add $LR_c, LR_y \Rightarrow LR_z$

(a) Before Coalescing

add $LR_t, LR_u \Rightarrow LR_{ab}$ ab
...
i2i $LR_{ab} \Rightarrow LR_c$ c
...
add $LR_{ab}, LR_w \Rightarrow LR_x$
add $LR_c, LR_y \Rightarrow LR_z$

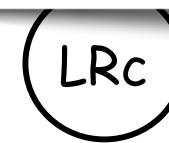
(b) After Coalescing LR_a and LR_b



Coalescing two live ranges
cannot increase the degrees
of any of their neighbours



but the resulting graph can be harder to color,
the degree of LR_{ab} can grow!!



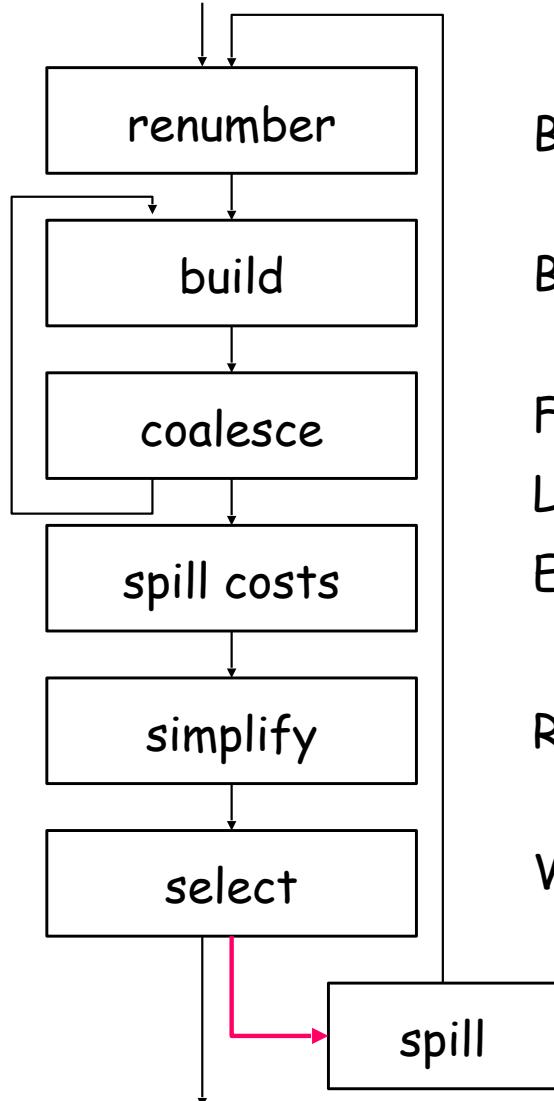
Safe Coalescing

- To perform coalescing, the allocator walks each block and examines each copy operation in the block
- When it finds $i2i \ LR_1 \Rightarrow LR_2$ with LR_1 and LR_2 that do not interfere the allocator combines them, eliminates the copy and update the Interference graph
- Coalescing two live range can prevent new coalescing: the order of coalescing matters

Several heuristics have been developed to decide when coalescing is safe, i.e. when it is guaranteed that it will not turn a K -colourable graph into one that is not K -colourable.

Using such heuristics, it is possible to interleave simplification steps with safe coalescing steps, thereby removing many useless move operations.

Chaitin-Briggs Allocator (Bottom-up Coloring)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

$LR_x \rightarrow LR_y$, and $\langle LR_x, LR_y \rangle \notin G_I \Rightarrow$ combine LR_x & LR_y

Estimate cost for spilling
each live range

Remove nodes from the graph

While stack is non-empty

pop n, insert n into G_I , & try to color it

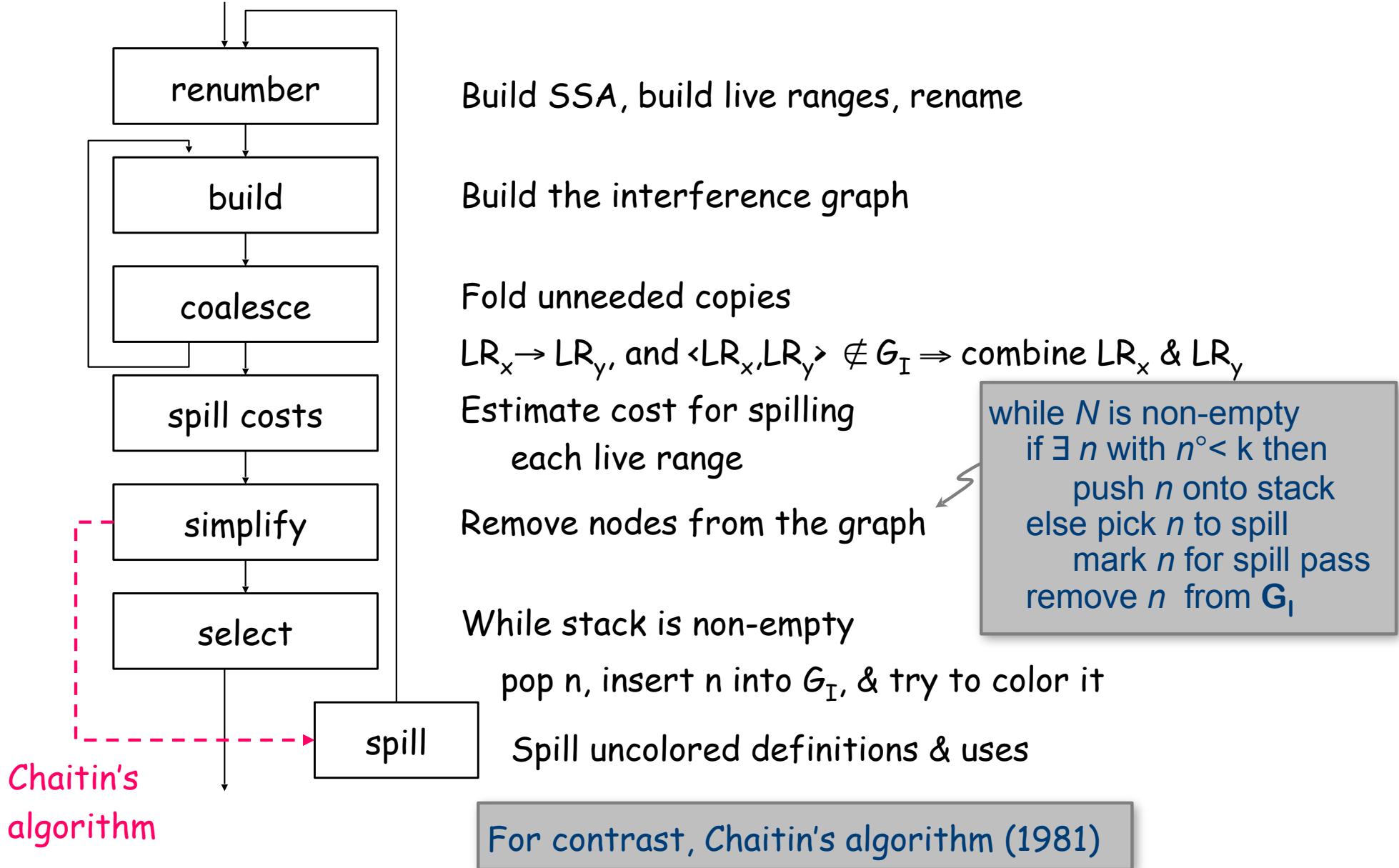
Spill uncolored definitions & uses

```
while N is non-empty
  if ∃ n with n° < k then
    push n onto stack
  else pick n to (maybe) spill
    push n onto stack
    remove n from G_I
```

Briggs' algorithm (1989)

Quick Aside ...

Chaitin's Allocator (Bottom-up Coloring)



Chaitin-Briggs Allocator

(Bottom-up Global)

Strengths & Weaknesses

- ↑ Precise interference graph
- ↑ Strong coalescing mechanism
- ↑ Handles register assignment well
- ↑ Runs fairly quickly
- ↓ Known to overspill in tight cases
- ↓ Interference graph has no geography
- ↓ Spills a live range everywhere

Is improvement still possible ?

⇒ yes, but the returns are getting rather small

Linear Scan Allocation

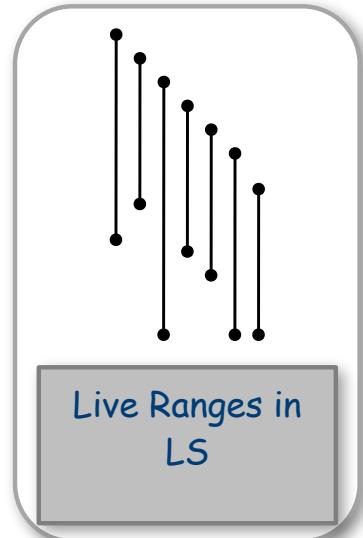
Coloring allocators are often viewed as too expensive for use in JIT environments, where compile time occurs at runtime

Linear scan allocators use an approximate interference graph and a version of the bottom-up local algorithm

Sun's HotSpot server compiler uses a complete Chaitin-Briggs allocator.

Live Interval

- $[i, j]$: live interval for variable v if there is no instruction with number $j' > j$ such that v is live at j' , and there is no instruction with number $i' < i$ such that v is live at i'
- conservative approximation of live ranges
- there may be sub ranges $[i, j]$ in which v is not live
- trivial live range for any variable – $[1, N]$



Linear Scan Allocation

Building the Interval Graph

- Consider the procedure as a linear list of operations
- A live range for some name is an interval (x,y)
- Intervals overestimates live ranges and therefore interference

The Algorithm

- Use bottom-up local algorithm
- Distance to next use is well defined
- Algorithm is fast & produces reasonable allocations

Variations have been proposed that build on this scheme

The linear scan algorithm

- compute the live intervals.
- live intervals are stored in a list sorted in order of increasing start point.
- At each step, the algorithm maintains a list, *active*, of live intervals that overlap the current point and have been placed in registers.
- *active* list is sorted in order of increasing end point.

The code

LinearScanRegisterAllocation

```
active ← {}
foreach live interval  $i$ , in order of increasing start point
    ExpireOldIntervals( $i$ )
    if length(active) =  $R$  then
        SpillAtInterval( $i$ )
    else
        register[ $i$ ] ← a register removed from pool of free registers
        add  $i$  to active, sorted by increasing end point
```

ExpireOldIntervals(i)

```
foreach interval  $j$  in active, in order of increasing end point
    if endpoint [ $j$ ] ≥ startpoint [ $i$ ] then
        return
    remove  $j$  from active
    add register[ $j$ ] to pool of free registers
```

SpillAtInterval(i)

```
spill ← last interval in active
if endpoint [spill] > endpoint [ $i$ ] then
    register[ $i$ ] ← register[spill]
    location[spill] ← new stack location
    remove spill from active
    add  $i$  to active, sorted by increasing end point
else
    location[ $i$ ] ← new stack location
```

Linear Scan example

Live ranges

a	b	c	d	e
orange				
orange	orange			
orange	orange	orange		
	orange	orange		
	orange	orange	orange	
		orange		orange
		orange	orange	
		orange	orange	
		orange		orange

Allocation

R1	R2
a	
a	b
a	b
	b
d	b
d	e
d	

c is spilled

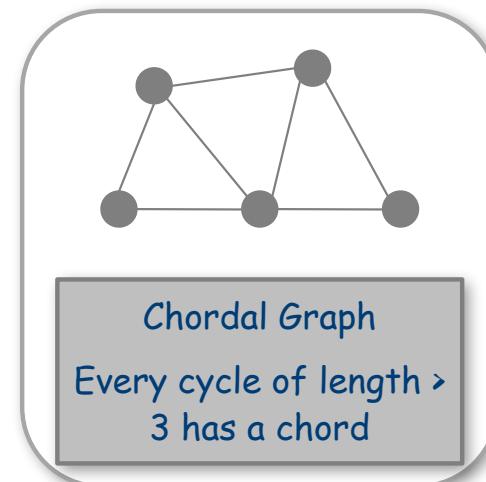
Global Coloring from SSA Form

- Chaitin-Briggs works from live ranges that are a coalesced version of SSA names

Observation: The interference graph of a program in SSA form is a chordal graph.

A **chordal graph** is a graph in which all cycles > 3 has a chord (an edge that is not part of the cycle but connects two vertices of the cycle)

Observation: Chordal graphs can be colored in $O(N)$ time.



Global Coloring from SSA Form

These two facts suggest allocation using an interference graph built from SSA Form

- SSA allocators use raw SSA names as live ranges

A-based allocation has created a lot of excitement in the last couple of years

Global Coloring from SSA Form

Coloring from SSA Names has its advantages

- If graph is k-colorable, it finds the coloring
 - (Opinion) An SSA-based allocator will find more k-colorable graphs than a live-range based allocator because SSA names are shorter and, thus, have fewer interferences.
- Allocator should be faster than a live-range allocator
 - Cost of live analysis folded into SSA construction, where it is amortised over other passes
 - Biggest expense in Chaitin-Briggs is the Build-Coalesce phase, which SSA allocator avoids, as it destroys the chordal graph

Global Coloring from SSA Form

Coloring from SSA Names has its disadvantages

- Coloring is rarely the problem
 - Most non-trivial codes spill; on trivial codes, both SSA allocator and classic Chaitin-Briggs are overkill. (Try linear scan?)
- SSA form provides no obvious help on spilling
- After allocation, code is still in SSA form
 - Need out-of-SSA translation
 - Introduce copies after allocation
 - Must run a post-allocation coalescing phase
 - Algorithms exist that do not use an interference graph
 - They are not as powerful as the Chaitin-Briggs coalescing phase

Hybrid Approach ?

How can the compiler attain both speed and precision?

Observation: lots of procedures are small & do not spill

Observation: some procedures are hard to allocate

Possible solution:

- Try different algorithms
- First, try linear scan
 - It is cheap and it may work
- If linear scan fails, try heavyweight allocator of choice
 - Might be Chaitin-Briggs, SSA, or some other algorithm
 - Use expensive allocator only when cheap one spills

This approach would not help with the speed of a complex compilation, but it might compensate on simple compilations