Review for the Final Exam

Stephen A. Edwards

Columbia University

Fall 2017

Table of Contents I

Table of Contents II

Table of Contents III

Table of Contents IV

The Final

75 minutes

Closed book

One double-sided sheet of notes of your own devising

Comprehensive: Anything discussed in class is fair game, including things from before the midterm

Little, if any, programming

Details of O'Caml/C/C++/Java/Prolog syntax not required

Broad knowledge of languages discussed

Compiling a Simple Program

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

What the Compiler Sees

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

```
i  n  t sp  g  c  d  (  i  n  t sp  a  , sp  i
n  t sp  b  ) nl  { nl sp sp  w  h  i  l  e sp
(  a sp  !  = sp  b  ) sp  { nl sp sp sp sp  i
f sp  (  a sp  > sp  b  ) sp  a sp  -  = sp  b
; nl sp sp sp sp  e  l  s  e sp  b sp  -  = sp
a  ; nl sp sp  } nl sp sp  r  e  t  u  r  n sp
a  ; nl  } nl
```

Text file is a sequence of characters

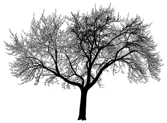
Lexical Analysis Gives Tokens

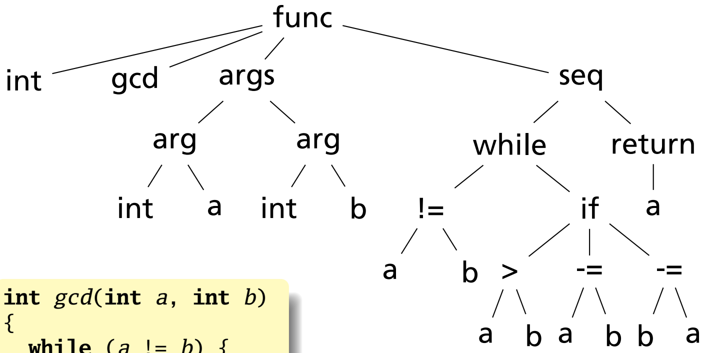
```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

```
int  gcd  (  int  a  ,  int  b  )  {  while  (  a
!=  b  )  {  if  (  a  >  b  )  a  -=  b  ;  else
b  -=  a  ;  }  return  a  ;  }
```

A stream of tokens. Whitespace, comments removed.

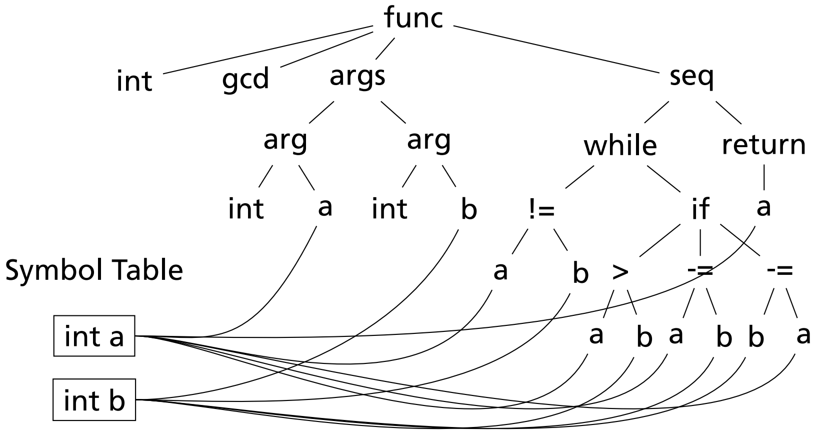
Parsing Gives an Abstract Syntax Tree





```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Semantic Analysis Resolves Symbols and Checks Types



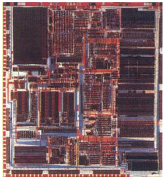
Translation into 3-Address Code

```
L0: sne   $1,  a, b
    seq   $0, $1, 0
    btrue $0, L1    # while (a != b)
    sl    $3,  b, a
    seq   $2, $3, 0
    btrue $2, L4    # if (a < b)
    sub   a,   a, b # a -= b
    jmp   L5
L4: sub   b,   b, a # b -= a
L5: jmp   L0
L1: ret    a
```

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Idealized assembly language w/ infinite registers

Generation of 80386 Assembly



```
gcd:  pushl %ebp            # Save BP
      movl  %esp,%ebp
      movl  8(%ebp),%eax    # Load a from stack
      movl  12(%ebp),%edx   # Load b from stack
.L8:  cmpl  %edx,%eax
      je    .L3             # while (a != b)
      jle   .L5             # if (a < b)
      subl  %edx,%eax       # a -= b
      jmp   .L8
.L5:  subl  %eax,%edx       # b -= a
      jmp   .L8
.L3:  leave                 # Restore SP, BP
      ret
```

Describing Tokens

Alphabet: A finite set of symbols

Examples: $\{$ 0, 1 $\}$, $\{$ A, B, C, . . . , Z $\}$, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Stephen, $\alpha\beta\gamma$

Language: A set of strings over an alphabet

Examples: $\emptyset$ (the empty language), $\{$ 1, 11, 111, 1111 $\}$, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let $L = \{\ \epsilon$, wo $\}$, $M = \{$ man, men $\}$

Concatenation: Strings from one followed by the other

$LM = \{$ man, men, woman, women $\}$

Union: All strings from each language

$L \cup M = \{\epsilon$, wo, man, men $\}$

Kleene Closure: Zero or more concatenations

$M^* = \{\epsilon\} \cup M \cup MM \cup MMM \cdots =$
$\{\epsilon$, man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, ...$\}$

Regular Expressions over an Alphabet Σ

A standard way to express languages for tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote languages $L(r)$ and $L(s)$,
    - $(r) \mid (s)$ denotes $L(r) \cup L(s)$
    - $(r)(s)$ denotes $\{tu : t \in L(r), u \in L(s)\}$
    - $(r)^*$ denotes $\cup_{i=0}^{\infty} L^i$ ($L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$)

Nondeterministic Finite Automata

"All strings containing an even number of 0's and 1's"

$$\begin{tikzcd}
\arrow[r] & A \arrow[r, bend left, "0"] \arrow[d, bend left, "1"] & B \arrow[l, bend left, "0"] \arrow[d, bend left, "1"] \\
& C \arrow[u, bend left, "1"] \arrow[r, bend left, "0"] & D \arrow[l, bend left, "0"] \arrow[u, bend left, "1"]
\end{tikzcd}$$

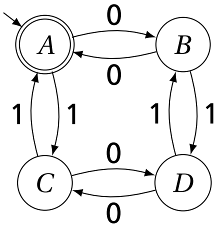
1. Set of states
   $S : \{ A \quad B \quad C \quad D \}$
2. Set of input symbols $\Sigma : \{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \rightarrow 2^S$

state	ϵ	0	1
A	$\emptyset$	$\{B\}$	$\{C\}$
B	$\emptyset$	$\{A\}$	$\{D\}$
C	$\emptyset$	$\{D\}$	$\{A\}$
D	$\emptyset$	$\{C\}$	$\{B\}$

4. Start state $s_0 : A$
5. Set of accepting states
   $F : \{ A \}$

The Language induced by an NFA

An NFA accepts an input string x iff there is a path from the start state to an accepting state that “spells out” x.

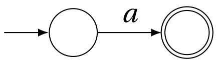


Show that the string “010010” is accepted.

$$\begin{tikzcd} A \arrow[r, "0"] & B \arrow[r, "1"] & D \arrow[r, "0"] & C \arrow[r, "0"] & D \arrow[r, "1"] & B \arrow[r, "0"] & A \end{tikzcd}$$

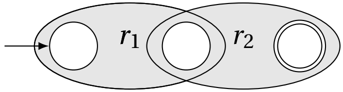
Translating REs into NFAs

a



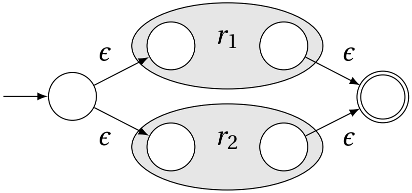
Symbol

$r_1 r_2$



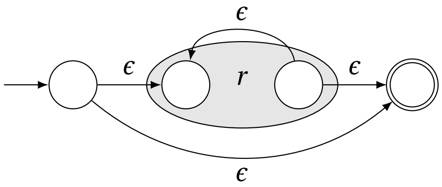
Sequence

$r_1 \mid r_2$



Choice

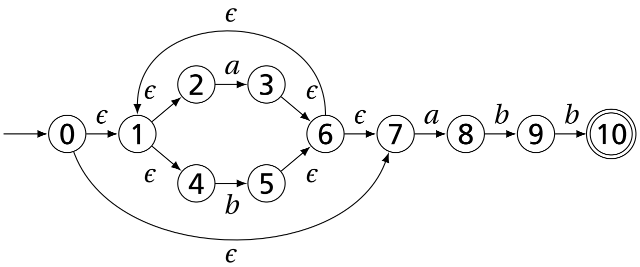
$(r)^*$



Kleene Closure

Translating REs into NFAs

Example: Translate $(a \mid b)^* abb$ into an NFA. Answer:



Show that the string "$aabb$" is accepted. Answer:

$$\begin{tikzcd} {} \arrow[r] & 0 \arrow[r, "\epsilon"] & 1 \arrow[r, "\epsilon"] & 2 \arrow[r, "a"] & 3 \arrow[r, "\epsilon"] & 6 \arrow[r, "\epsilon"] & 7 \arrow[r, "a"] & 8 \arrow[r, "b"] & 9 \arrow[r, "b"] & 10 \end{tikzcd}$$

Simulating NFAs

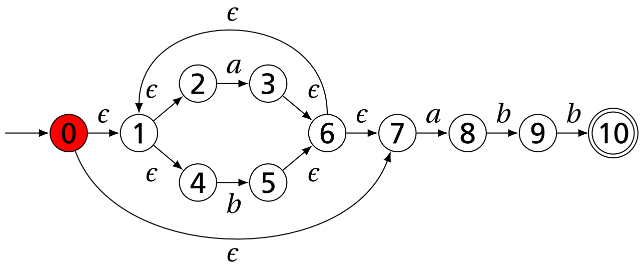
Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?

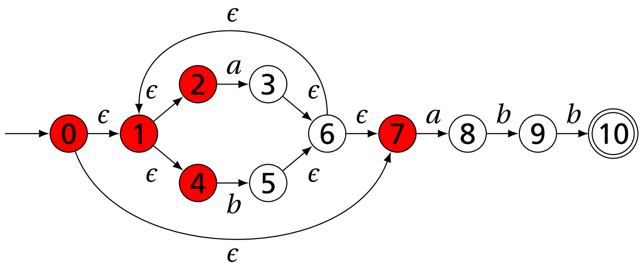
Solution: follow them all and sort it out later.

“Two-stack” NFA simulation algorithm:

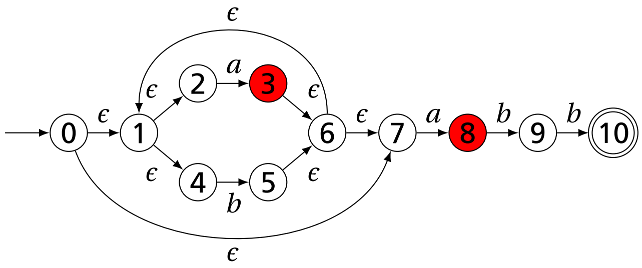
1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

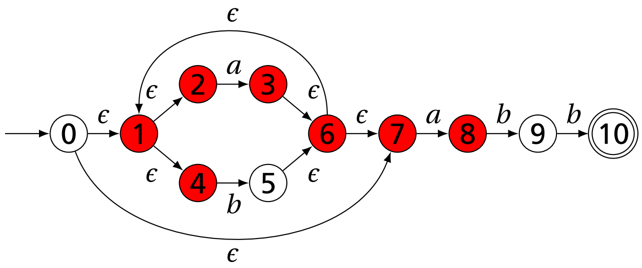
Simulating an NFA: $\cdot aabb$, Start



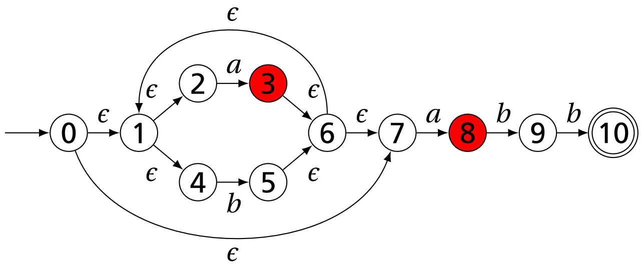


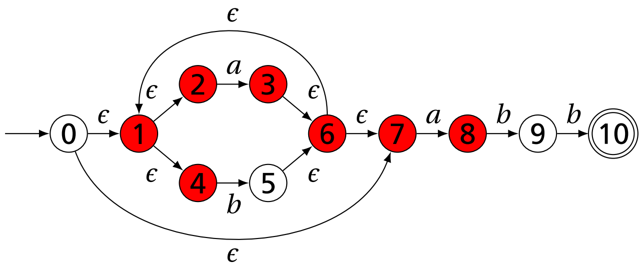
Simulating an NFA: $a{\cdot}abb$



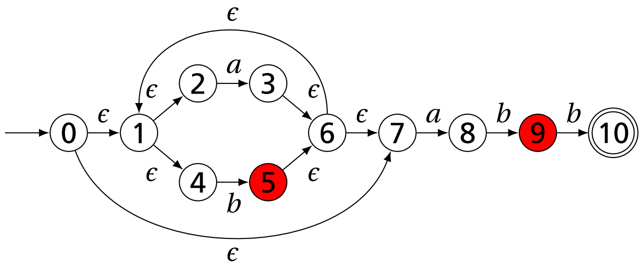


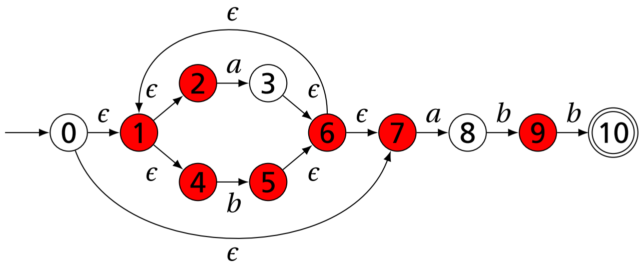
Simulating an NFA: $aa \cdot bb$



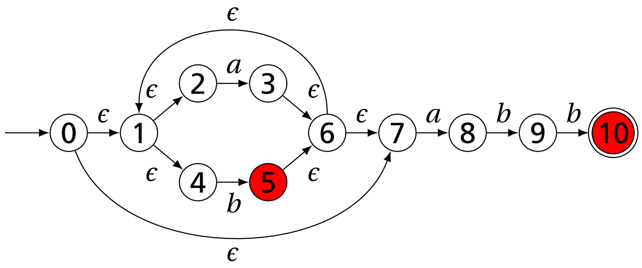


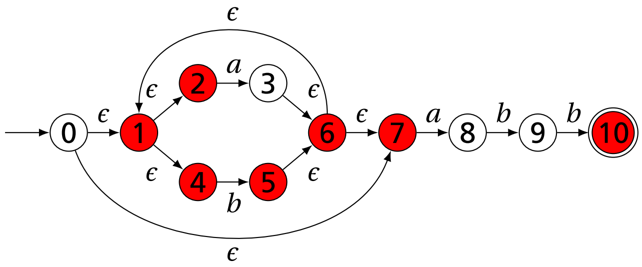
Simulating an NFA: $aab{\cdot}b$





Simulating an NFA: $aabb\cdot$, Done





Deterministic Finite Automata

Restricted form of NFAs:

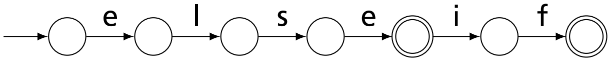
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

Differs subtly from the definition used in COMS W3261 (Sipser, *Introduction to the Theory of Computation*)

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

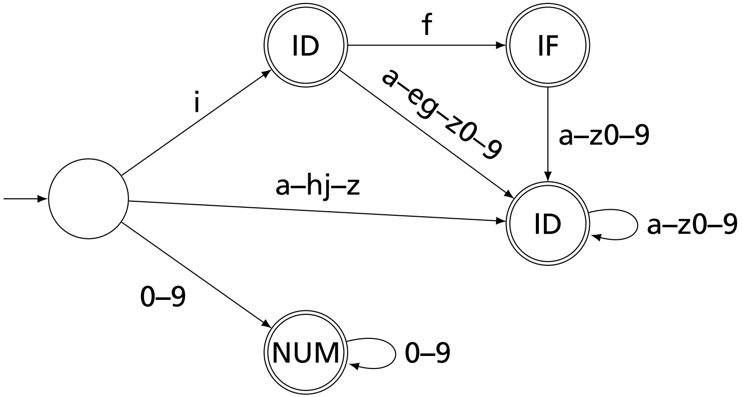
Deterministic Finite Automata

```
{
   type token = ELSE | ELSEIF
}

rule token =
  parse "else"   { ELSE }
      | "elseif" { ELSEIF }
```


$$\begin{tikzcd}
{} \arrow[r] & \bigcirc \arrow[r, "e"] & \bigcirc \arrow[r, "l"] & \bigcirc \arrow[r, "s"] & \bigcirc \arrow[r, "e"] & \circledcirc \arrow[r, "i"] & \bigcirc \arrow[r, "f"] & \circledcirc
\end{tikzcd}$$

Deterministic Finite Automata

```
{ type token = IF | ID of string | NUM of string }

rule token =
  parse "if"                                       { IF }
      | ['a'-'z'] ['a'-'z' '0'-'9']* as lit { ID(lit) }
      | ['0'-'9']+                         as num { NUM(num) }
```



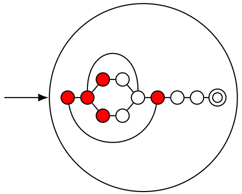
Building a DFA from an NFA

Subset construction algorithm

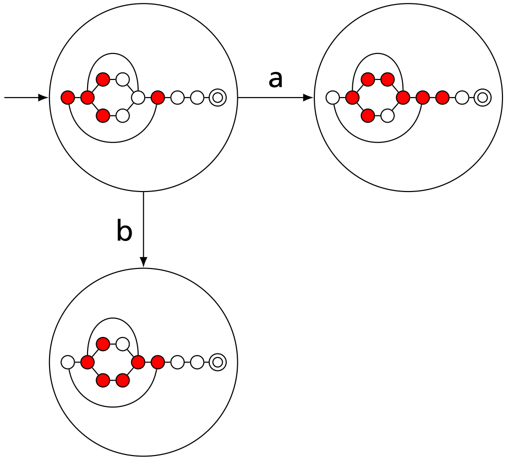
Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

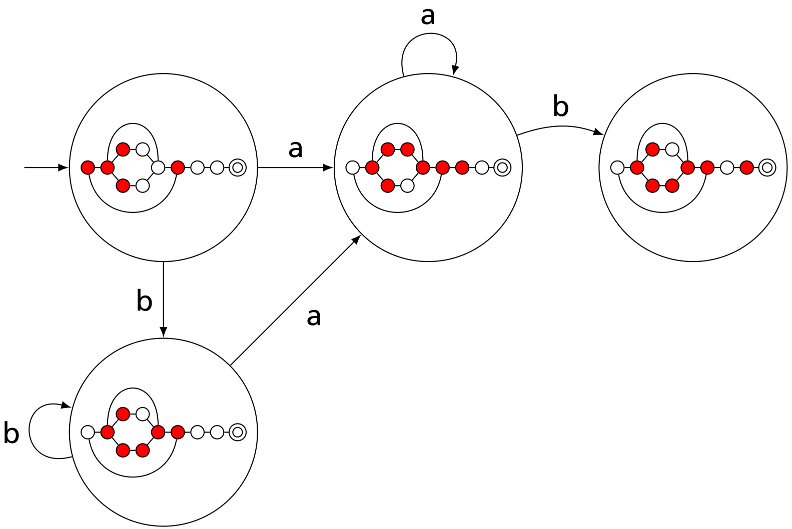
Subset construction for $(a \mid b)^* abb$



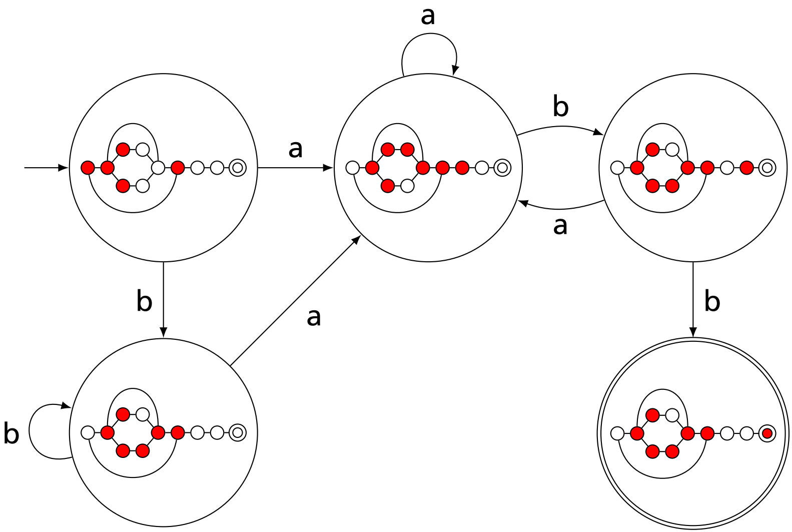
Subset construction for $(a \mid b)^* abb$



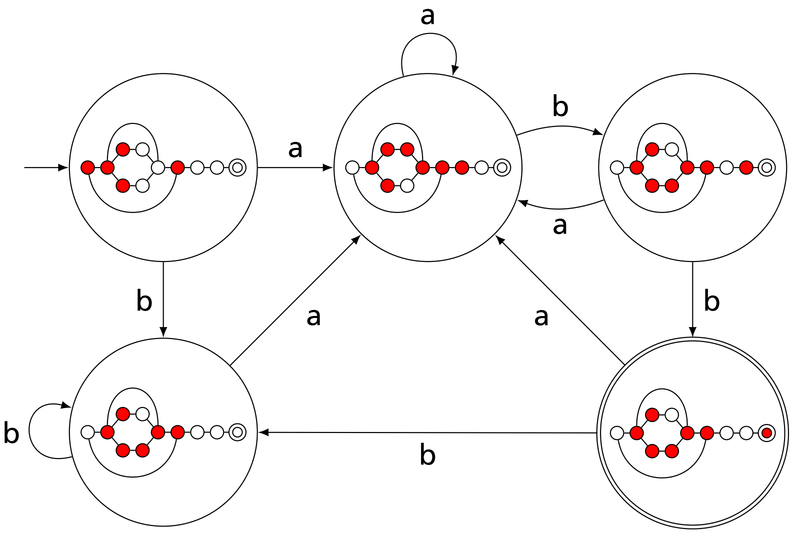
Subset construction for $(a \mid b)^* abb$



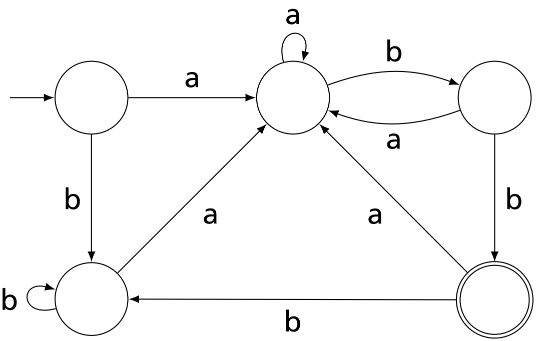
Subset construction for $(a \mid b)^* abb$



Subset construction for $(a \mid b)^* abb$



Result of subset construction for $(a \mid b)^* abb$



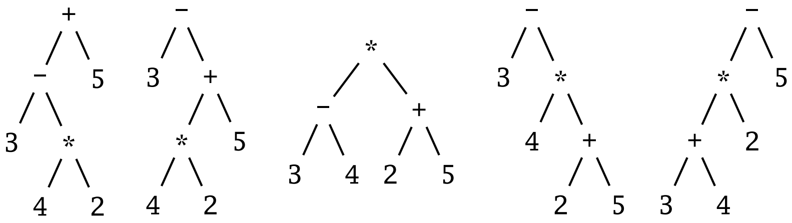
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

```
3 - 4 * 2 + 5
```

with the grammar

$$e \rightarrow e + e \mid e - e \mid e * e \mid e \,/\, e \mid N$$



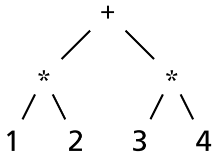
Operator Precedence

Defines how "sticky" an operator is.

```
1 * 2 + 3 * 4
```

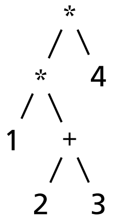
`*` at higher precedence than `+`:

```
(1 * 2) + (3 * 4)
```



`+` at higher precedence than `*`:

```
1 * (2 + 3) * 4
```

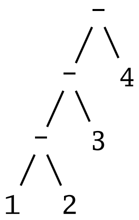


Associativity

Whether to evaluate left-to-right or right-to-left

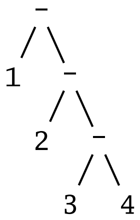
Most operators are left-associative

```
1 - 2 - 3 - 4
```



$((1-2)-3)-4$

left associative



$1-(2-(3-4))$

right associative

Fixing Ambiguous Grammars

A grammar specification:

```
expr :
    expr PLUS expr
  | expr MINUS expr
  | expr TIMES expr
  | expr DIVIDE expr
  | NUMBER
```

Ambiguous: no precedence or associativity.

Ocamlyacc's complaint: “16 shift/reduce conflicts.”

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr PLUS expr
     | expr MINUS expr
     | term

term : term TIMES term
     | term DIVIDE term
     | atom

atom  : NUMBER
```

Still ambiguous: associativity not defined

Ocamlyacc's complaint: "8 shift/reduce conflicts."

Assigning Associativity

Make one side the next level of precedence

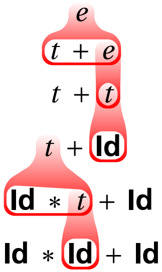
```
expr : expr PLUS term
     | expr MINUS term
     | term

term : term TIMES atom
     | term DIVIDE atom
     | atom

atom  : NUMBER
```

This is left-associative.

No shift/reduce conflicts.

Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**

$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \textbf{Id} * t$
$4: t \rightarrow \textbf{Id}$


e
$t + e$
$t + t$
$t + \textbf{Id}$
$\textbf{Id} * t + \textbf{Id}$
$\textbf{Id} * \textbf{Id} + \textbf{Id}$

At each step, expand the *rightmost* nonterminal.

nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambigious.

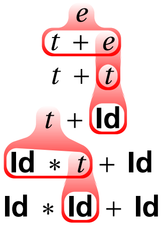
Rightmost Derivation: What to Expand

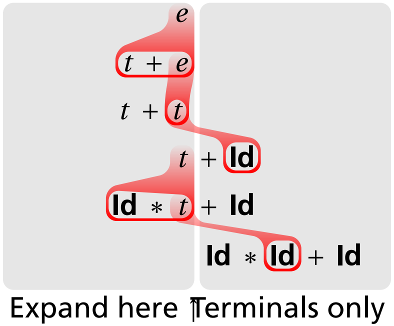
$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$





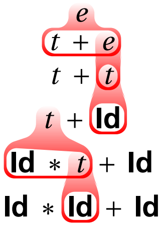
Reverse Rightmost Derivation

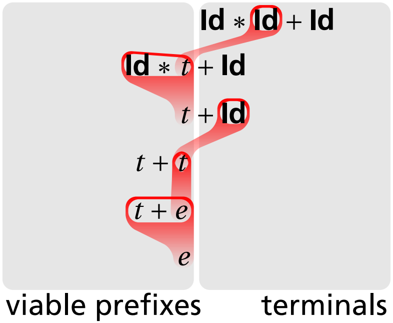
$1: e \rightarrow t + e$

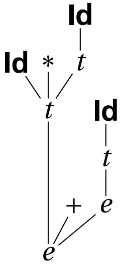
$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$







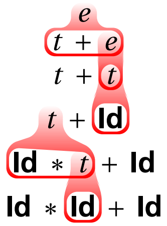
Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

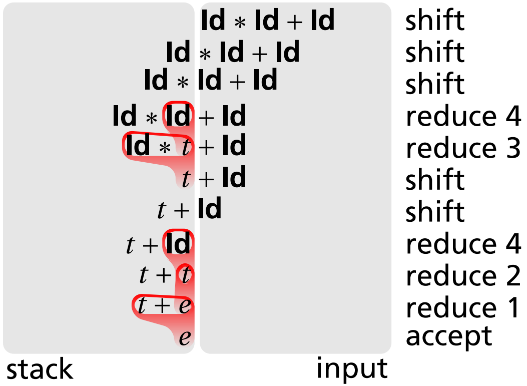
$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$



e
$t + e$
$t + t$
$t + \textbf{Id}$
$\textbf{Id} * t + \textbf{Id}$
$\textbf{Id} * \textbf{Id} + \textbf{Id}$



stack	input	
	$\textbf{Id} * \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id}$	$* \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} *$	$\textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} * \textbf{Id}$	$+ \textbf{Id}$	reduce 4
$\textbf{Id} * t$	$+ \textbf{Id}$	reduce 3
t	$+ \textbf{Id}$	shift
$t +$	$\textbf{Id}$	shift
$t + \textbf{Id}$		reduce 4
$t + t$		reduce 2
$t + e$		reduce 1
e		accept

Handle Hunting

Right Sentential Form: any step in a rightmost derivation

Handle: in a sentential form, a RHS of a rule that, when rewritten, yields the previous step in a rightmost derivation.

The big question in shift/reduce parsing:

When is there a handle on the top of the stack?

Enumerate all the right-sentential forms and pattern-match against them? *Usually infinite in number, but let's try anyway.*

The Handle-Identifying Automaton

Magical result, due to Knuth: *An automaton suffices to locate a handle in a right-sentential form.*

$\mathbf{Id} * \mathbf{Id} * \cdots * \boxed{\mathbf{Id} * t} \cdots$

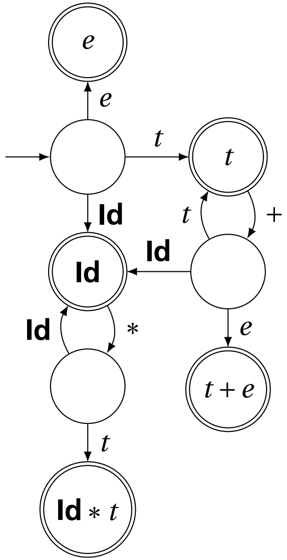
$\mathbf{Id} * \mathbf{Id} * \cdots * \boxed{\mathbf{Id}} \cdots$

$t + t + \cdots + \boxed{t + e}$

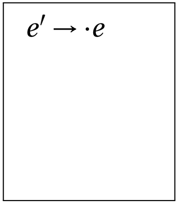
$t + t + \cdots + t + \boxed{\mathbf{Id}}$

$t + t + \cdots + t + \mathbf{Id} * \mathbf{Id} * \cdots * \boxed{\mathbf{Id} * t}$

$t + t + \cdots + \boxed{t}$



Building the Initial State of the LR(0) Automaton



$1: e \to t + e$
$2: e \to t$
$3: t \to \textbf{Id} * t$
$4: t \to \textbf{Id}$

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \to \cdot e$"

Building the Initial State of the LR(0) Automaton

$$\boxed{\begin{array}{l} e' \rightarrow \cdot e \\ e \rightarrow \cdot t + e \\ e \rightarrow \cdot t \\ \\ \\ \end{array}}$$

$1 : e \rightarrow t + e$
$2 : e \rightarrow t$
$3 : t \rightarrow \textbf{Id} \; * \, t$
$4 : t \rightarrow \textbf{Id}$

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \rightarrow \cdot e$"

There are two choices for what an e may expand to: $t + e$ and t. So when $e' \rightarrow \cdot e$, $e \rightarrow \cdot t + e$ and $e \rightarrow \cdot t$ are also true, i.e., it must start with a string expanded from t.

Building the Initial State of the LR(0) Automaton

$$\boxed{\begin{array}{l} e' \rightarrow \cdot e \\ e \rightarrow \cdot t + e \\ e \rightarrow \cdot t \\ t \rightarrow \cdot \textbf{Id} * t \\ t \rightarrow \cdot \textbf{Id} \end{array}}$$

$1 : e \rightarrow t + e$
$2 : e \rightarrow t$
$3 : t \rightarrow \textbf{Id} \ * t$
$4 : t \rightarrow \textbf{Id}$

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \rightarrow \cdot e$"

There are two choices for what an e may expand to: $t + e$ and t. So when $e' \rightarrow \cdot e$, $e \rightarrow \cdot t + e$ and $e \rightarrow \cdot t$ are also true, i.e., it must start with a string expanded from t.

Similarly, t must be either $\textbf{Id} * t$ or **Id**, so $t \rightarrow \cdot \textbf{Id} * t$ and $t \rightarrow \cdot \textbf{Id}$.

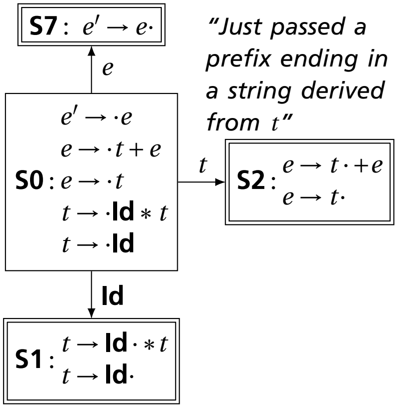
Building the LR(0) Automaton

$$\boxed{\begin{array}{rl} & e' \to \cdot e \\ & e \to \cdot t + e \\ \textbf{S0}: & e \to \cdot t \\ & t \to \cdot \textbf{Id} * t \\ & t \to \cdot \textbf{Id} \end{array}}$$

The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**.

Building the LR(0) Automaton

"Just passed a string derived from e"



S7 : $e' \rightarrow e\cdot$
e
S0 : $e' \rightarrow \cdot e$, $e \rightarrow \cdot t + e$, $e \rightarrow \cdot t$, $t \rightarrow \cdot \mathbf{Id} * t$, $t \rightarrow \cdot \mathbf{Id}$
t
S2 : $e \rightarrow t \cdot + e$, $e \rightarrow t\cdot$
Id
S1 : $t \rightarrow \mathbf{Id} \cdot * t$, $t \rightarrow \mathbf{Id}\cdot$

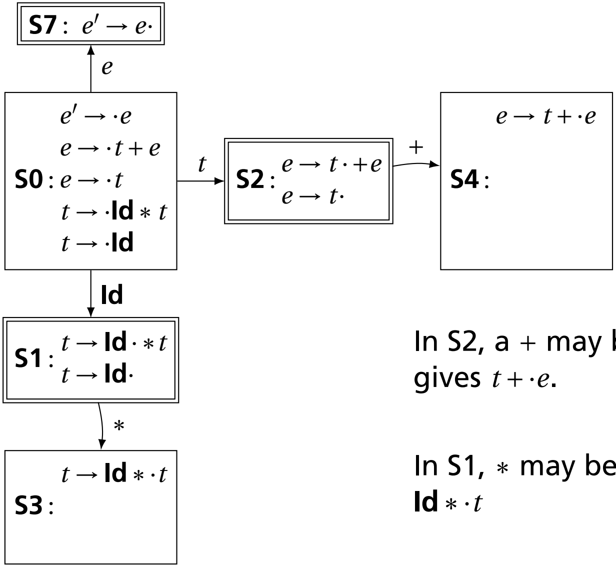
"Just passed a prefix ending in a string derived from t"

*"Just passed a prefix that ended in an **Id**"*

The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**.

The items for these three states come from advancing the $\cdot$ across each thing, then performing the closure operation (vacuous here).

Building the LR(0) Automaton

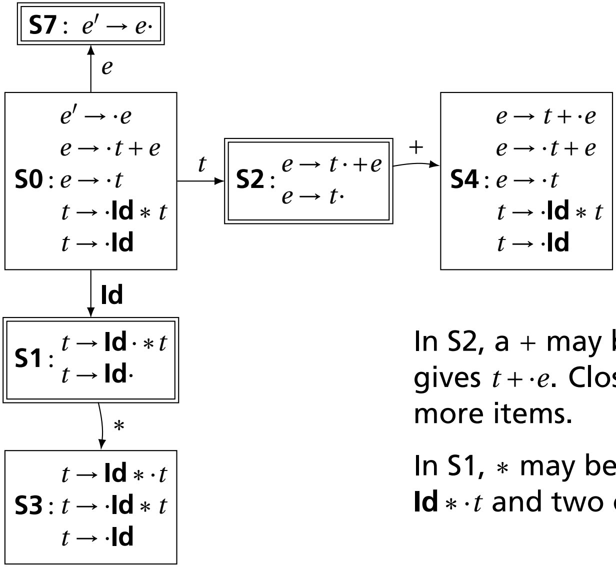


$\mathbf{S7}: e' \rightarrow e\cdot$
e
$e' \rightarrow \cdot e$
$e \rightarrow \cdot t + e$
$\mathbf{S0}: e \rightarrow \cdot t$
$t \rightarrow \cdot \mathbf{Id} * t$
$t \rightarrow \cdot \mathbf{Id}$
t
$\mathbf{S2}: e \rightarrow t \cdot + e$
$e \rightarrow t\cdot$
$+$
$e \rightarrow t + \cdot e$
$\mathbf{S4}:$
$\mathbf{Id}$
$\mathbf{S1}: t \rightarrow \mathbf{Id} \cdot * t$
$t \rightarrow \mathbf{Id}\cdot$
$*$
$t \rightarrow \mathbf{Id} * \cdot t$
$\mathbf{S3}:$

In S2, a $+$ may be next. This gives $t + \cdot e$.

In S1, $*$ may be next, giving $\mathbf{Id} * \cdot t$

Building the LR(0) Automaton

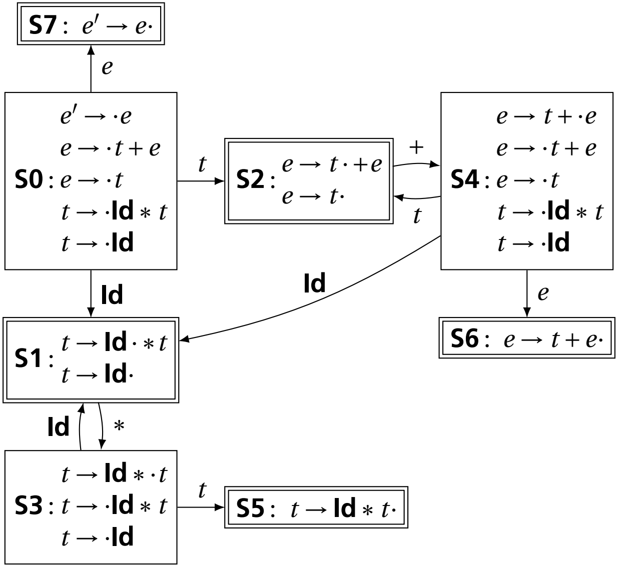


S7 : $e' \rightarrow e\cdot$
e
S0 : $e' \rightarrow \cdot e$
$e \rightarrow \cdot t + e$
$e \rightarrow \cdot t$
$t \rightarrow \cdot \mathbf{Id} * t$
$t \rightarrow \cdot \mathbf{Id}$
t
S2 : $e \rightarrow t \cdot + e$
$e \rightarrow t\cdot$
$+$
S4 : $e \rightarrow t + \cdot e$
$e \rightarrow \cdot t + e$
$e \rightarrow \cdot t$
$t \rightarrow \cdot \mathbf{Id} * t$
$t \rightarrow \cdot \mathbf{Id}$
Id
S1 : $t \rightarrow \mathbf{Id} \cdot * t$
$t \rightarrow \mathbf{Id}\cdot$
$*$
S3 : $t \rightarrow \mathbf{Id} * \cdot t$
$t \rightarrow \cdot \mathbf{Id} * t$
$t \rightarrow \cdot \mathbf{Id}$

In S2, a $+$ may be next. This gives $t + \cdot e$. Closure adds 4 more items.

In S1, $*$ may be next, giving **Id** $* \cdot t$ and two others.

Building the LR(0) Automaton



S7 : $e' \to e\cdot$
e
S0 : $e' \to \cdot e$, $e \to \cdot t + e$, $e \to \cdot t$, $t \to \cdot \mathbf{Id} * t$, $t \to \cdot \mathbf{Id}$
t
S2 : $e \to t \cdot + e$, $e \to t\cdot$
$+$
t
S4 : $e \to t + \cdot e$, $e \to \cdot t + e$, $e \to \cdot t$, $t \to \cdot \mathbf{Id} * t$, $t \to \cdot \mathbf{Id}$
Id
Id
e
S6 : $e \to t + e\cdot$
S1 : $t \to \mathbf{Id} \cdot * t$, $t \to \mathbf{Id}\cdot$
Id
$*$
S3 : $t \to \mathbf{Id} * \cdot t$, $t \to \cdot \mathbf{Id} * t$, $t \to \cdot \mathbf{Id}$
t
S5 : $t \to \mathbf{Id} * t\cdot$

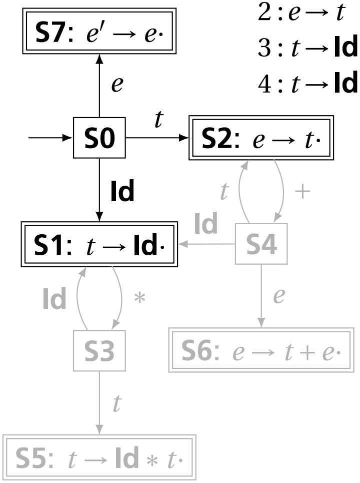
Converting the LR(0) Automaton to an SLR Parsing Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow$ **Id** $* \, t$

$4: t \rightarrow$ **Id**

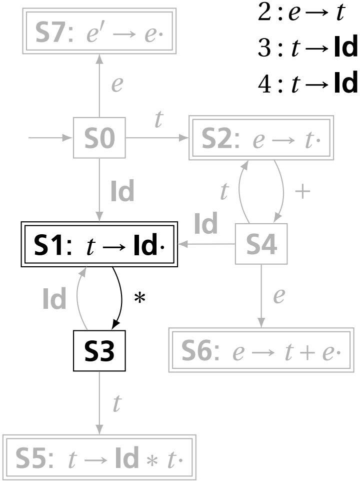


S7: $e' \rightarrow e\cdot$
e
S0
t
S2: $e \rightarrow t\cdot$
Id
t
$+$
Id
S1: $t \rightarrow$ Id$\cdot$
S4
Id
$*$
e
S3
S6: $e \rightarrow t + e\cdot$
t
S5: $t \rightarrow$ Id $* \, t\cdot$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2

From S0, shift an **Id** and go to S1; or cross a t and go to S2; or cross an e and go to S7.

Converting the LR(0) Automaton to an SLR Parsing Table



S7: $e' \rightarrow e\cdot$
e
S0
t
S2: $e \rightarrow t\cdot$
Id
t
$+$
Id
S1: $t \rightarrow \textbf{Id}\cdot$
S4
Id
$*$
e
S3
S6: $e \rightarrow t + e\cdot$
t
S5: $t \rightarrow \textbf{Id} * t\cdot$

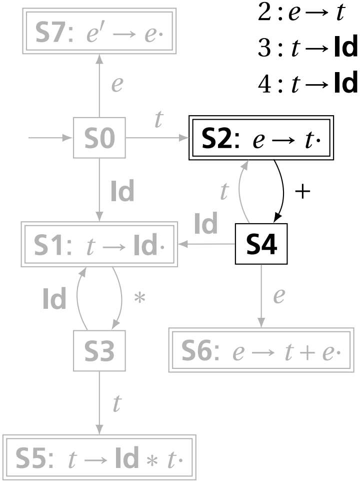
$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \textbf{Id} * t$
$4: t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	\$	e	t
0	s1				7	2
1		r4	s3	r4		

From S1, shift a $*$ and go to S3; or, if the next input could follow a t, reduce by rule 4. According to rule 1, $+$ could follow t; from rule 2, \$ could.

Converting the LR(0) Automaton to an SLR Parsing Table

$$1 : e \rightarrow t + e$$
$$2 : e \rightarrow t$$
$$3 : t \rightarrow \mathbf{Id} * t$$
$$4 : t \rightarrow \mathbf{Id}$$



S7: $e' \rightarrow e\cdot$
e
S0
t
S2: $e \rightarrow t\cdot$
Id
t
$+$
Id
S1: $t \rightarrow \mathbf{Id}\cdot$
S4
Id
$*$
e
S3
S6: $e \rightarrow t + e\cdot$
t
S5: $t \rightarrow \mathbf{Id} * t\cdot$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		

From S2, shift a $+$ and go to S4; or, if the next input could follow an e (only the end-of-input $), reduce by rule 2.

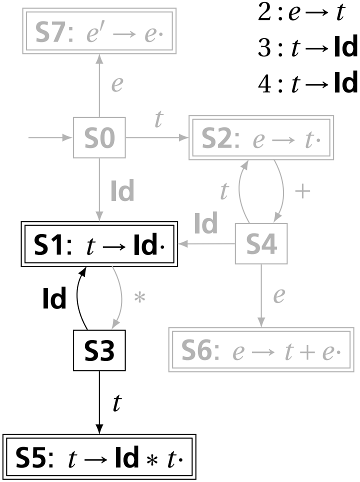
Converting the LR(0) Automaton to an SLR Parsing Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$

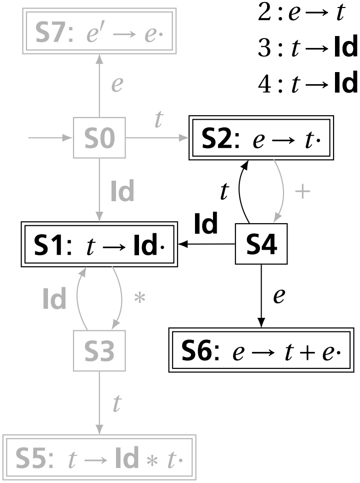


S7: $e' \rightarrow e\cdot$
e
t
S0
S2: $e \rightarrow t\cdot$
Id
t
$+$
Id
S1: $t \rightarrow \textbf{Id}\cdot$
S4
Id
$*$
e
S3
S6: $e \rightarrow t + e\cdot$
t
S5: $t \rightarrow \textbf{Id} * t\cdot$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5

From S3, shift an **Id** and go to S1; or cross a t and go to S5.

Converting the LR(0) Automaton to an SLR Parsing Table



$1 : e \to t + e$

$2 : e \to t$

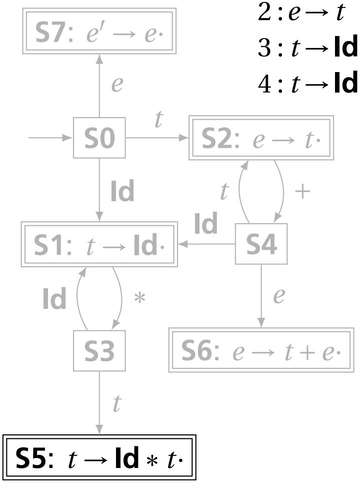
$3 : t \to \textbf{Id} * t$

$4 : t \to \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2

From S4, shift an **Id** and go to S1; or cross an e or a t.

Converting the LR(0) Automaton to an SLR Parsing Table

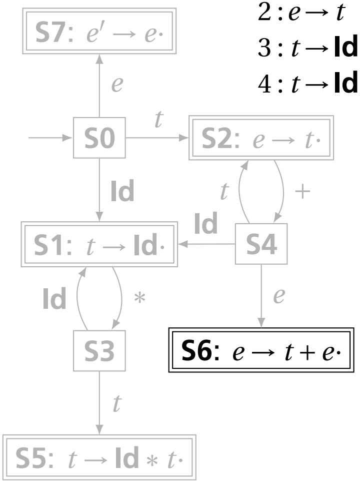


$1: e \to t + e$
$2: e \to t$
$3: t \to \textbf{Id} * t$
$4: t \to \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		

From S5, reduce using rule 3 if the next symbol could follow a t (again, $+$ and $).

Converting the LR(0) Automaton to an SLR Parsing Table

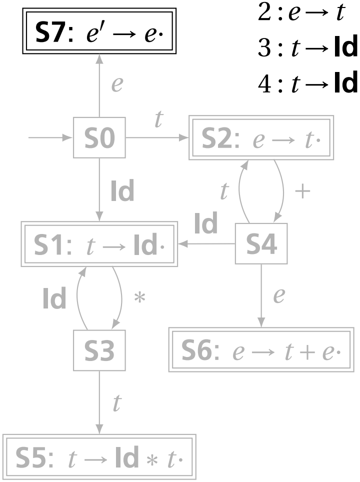


$1: e \to t + e$
$2: e \to t$
$3: t \to \textbf{Id} * t$
$4: t \to \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		

From S6, reduce using rule 1 if the next symbol could follow an e ($ only).

Converting the LR(0) Automaton to an SLR Parsing Table



$1: e \to t + e$

$2: e \to t$

$3: t \to \textbf{Id} * t$

$4: t \to \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

If, in S7, we just crossed an e, accept if we are at the end of the input.

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \textbf{Id} * t$
$4: t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	**Id** $*$ **Id** $+$ **Id** $	Shift, goto 1

Look at the state on top of the stack and the next input token.

Find the action (shift, reduce, or error) in the table.

In this case, shift the token onto the stack and mark it with state 1.

Shift/Reduce Parsing with an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

$3 : t \rightarrow \textbf{Id} * t$

$4 : t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	**\$**	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	$\textbf{Id} * \textbf{Id} + \textbf{Id}\,\$$	Shift, goto 1
0 $\overset{\textbf{Id}}{1}$	$* \textbf{Id} + \textbf{Id}\,\$$	Shift, goto 3

Here, the state is 1, the next symbol is $*$, so shift and mark it with state 3.

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	**$**	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	$\textbf{Id} * \textbf{Id} + \textbf{Id}\,\$$	Shift, goto 1
$0 \; \begin{matrix}\textbf{Id}\\1\end{matrix}$	$* \textbf{Id} + \textbf{Id}\,\$$	Shift, goto 3
$0 \; \begin{matrix}\textbf{Id}\\1\end{matrix} \; \begin{matrix}*\\3\end{matrix}$	$\textbf{Id} + \textbf{Id}\,\$$	Shift, goto 1
$0 \; \begin{matrix}\textbf{Id}\\1\end{matrix} \; \begin{matrix}*\\3\end{matrix} \; \begin{matrix}\textbf{Id}\\1\end{matrix}$	$+ \textbf{Id}\,\$$	Reduce 4

Here, the state is 1, the next symbol is $+$. The table says reduce using rule 4.

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	+	*	$	*e*	*t*
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	**Input**	**Action**
0	**Id** * **Id** + **Id** $	Shift, goto 1
0 **Id** 1	* **Id** + **Id** $	Shift, goto 3
0 **Id** 1 * 3	**Id** + **Id** $	Shift, goto 1
0 **Id** 1 * 3 **Id** 1	+ **Id** $	Reduce 4
0 **Id** 1 * 3	+ **Id** $	

Remove the RHS of the rule (here, just **Id**), observe the state on the top of the stack, and consult the “goto” portion of the table.

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \textbf{Id} * t$
$4: t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	+	*	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	**Id** * **Id** + **Id** $	Shift, goto 1
0 **Id**/1	* **Id** + **Id** $	Shift, goto 3
0 **Id**/1 */3	**Id** + **Id** $	Shift, goto 1
0 **Id**/1 */3 **Id**/1	+ **Id** $	Reduce 4
0 **Id**/1 */3 t/5	+ **Id** $	Reduce 3

Here, we push a t with state 5. This effectively “backs up” the LR(0) automaton and runs it over the newly added nonterminal.

In state 5 with an upcoming +, the action is “reduce 3.”

Shift/Reduce Parsing with an SLR Table

$1 : e \to t + e$

$2 : e \to t$

$3 : t \to \textbf{Id} * t$

$4 : t \to \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	**$**	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	$\textbf{Id} * \textbf{Id} + \textbf{Id}\,\$$	Shift, goto 1
$0\ \overset{\textbf{Id}}{1}$	$* \textbf{Id} + \textbf{Id}\,\$$	Shift, goto 3
$0\ \overset{\textbf{Id}}{1}\ \overset{*}{3}$	$\textbf{Id} + \textbf{Id}\,\$$	Shift, goto 1
$0\ \overset{\textbf{Id}}{1}\ \overset{*}{3}\ \overset{\textbf{Id}}{1}$	$+ \textbf{Id}\,\$$	Reduce 4
$0\ \overset{\textbf{Id}}{1}\ \overset{*}{3}\ \overset{t}{5}$	$+ \textbf{Id}\,\$$	Reduce 3
$0\ \overset{t}{2}$	$+ \textbf{Id}\,\$$	Shift, goto 4

This time, we strip off the RHS for rule 3, **Id** $* \, t$, exposing state 0, so we push a t with state 2.

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \textbf{Id} * t$
$4: t \rightarrow \textbf{Id}$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	**Id** $*$ **Id** $+$ **Id** $	Shift, goto 1
0 **Id** 1	$*$ **Id** $+$ **Id** $	Shift, goto 3
0 **Id** 1 $*$ 3	**Id** $+$ **Id** $	Shift, goto 1
0 **Id** 1 $*$ 3 **Id** 1	$+$ **Id** $	Reduce 4
0 **Id** 1 $*$ 3 t 5	$+$ **Id** $	Reduce 3
0 t 2	$+$ **Id** $	Shift, goto 4
0 t 2 $+$ 4	**Id** $	Shift, goto 1
0 t 2 $+$ 4 **Id** 1	$	Reduce 4
0 t 2 $+$ 4 t 2	$	Reduce 2
0 t 2 $+$ 4 e 6	$	Reduce 1
0 e 7	$	Accept

Types

A restriction on the possible interpretations of a segment of memory or other program construct.

Two uses:

Safety: avoids data being treated as something it isn't

Optimization: eliminates certain runtime decisions

Types of Types

Type	Examples
Basic	Machine words, floating-point numbers, addresses/pointers
Aggregate	Arrays, structs, classes
Function	Function pointers, lambdas

Basic Types

Groups of data the processor is designed to operate on.

On an ARM processor,

Type	**Width (bits)**
Unsigned/two's-complement binary	
Byte	8
Halfword	16
Word	32
IEEE 754 Floating Point	
Single-Precision scalars & vectors	32, 64, .., 256
Double-Precision scalars & vectors	64, 128, 192, 256

Derived types

Array: a list of objects of the same type, often fixed-length

Record: a collection of named fields, often of different types

Pointer/References: a reference to another object

Function: a reference to a block of code

Structs

Structs are the precursors of objects:

Group and restrict what can be stored in an object, but not what operations they permit.

Can fake object-oriented programming:

```
struct poly { ... };

struct poly *poly_create();
void        poly_destroy(struct poly *p);
void        poly_draw(struct poly *p);
void        poly_move(struct poly *p, int x, int y);
int         poly_area(struct poly *p);
```

Unions: Variant Records

A `struct` holds all of its fields at once. A `union` holds only one of its fields at any time (the last written).

```
union token {
  int i;
  float f;
  char *string;
};

union token t;
t.i = 10;
t.f = 3.14159;       /* overwrite t.i */
char *s = t.string;  /* return gibberish */
```

Applications of Variant Records

A primitive form of polymorphism:

```
struct poly {
  int x, y;
  int type;
  union { int radius;
          int size;
          float angle; } d;
};
```

If `poly.type == CIRCLE`, use `poly.d.radius`.

If `poly.type == SQUARE`, use `poly.d.size`.

If `poly.type == LINE`, use `poly.d.angle`.

Name vs. Structural Equivalence

```
struct f {
  int x, y;
} foo = { 0, 1 };

struct b {
  int x, y;
} bar;

bar = foo;
```

Is this legal in C? Should it be?

Type Expressions

C's declarators are unusual: they always specify a name along with its type.

Languages more often have *type expressions*: a grammar for expressing a type.

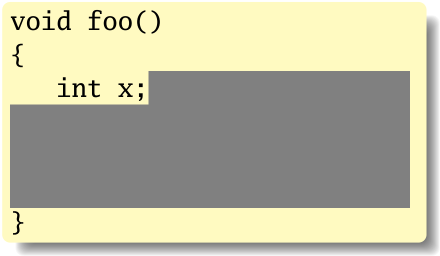
Type expressions appear in three places in C:

```
(int *) a                      /* Type casts */
sizeof(float [10])             /* Argument of sizeof() */
int f(int, char *, int (*)(int))  /* Function argument types */
```

Basic Static Scope in C, C++, Java, etc.

A name begins life where it is declared and ends at the end of its block.

From the CLRM, “The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.”



```
void foo()
{
    int x;



}
```

Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, “If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.”

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }


}
```

Static Scoping in Java

```
public void example() {
  // x, y, z not visible

  int x;
  // x visible

  for ( int y = 1 ; y < 10 ; y++ ) {
    // x, y visible

    int z;
    // x, y, z visible
  }

  // x visible
}
```

Basic Static Scope in O’Caml

A name is bound after the “in” clause of a “let.” If the name is re-bound, the binding takes effect *after* the “in.”

```
let x = 8 in


let x = x + 1 in
```

Returns the pair (12, 8):

```
let x = 8 in
  (let x = x + 2 in
    x + 2),
x
```

Let Rec in O'Caml

The "rec" keyword makes a name visible to its definition. This only makes sense for functions.

```
let rec fib i =
  if i < 1 then 1 else
    fib (i-1) + fib (i-2)
in
  fib 5
```

```
(* Nonsensical *)
let rec x = x + 3 in
```

Let...and in O’Caml

Let...and lets you bind multiple names at once. Definitions are not mutually visible unless marked “rec.”

```
let x = 8
and y = 9 in
```

```
let rec fac n =
     if n < 2 then
       1
     else
       n * fac1 n
and fac1 n = fac (n - 1)
in
fac 5
```

Nesting Function Definitions

```
let articles words =

  let report w =

    let count = List.length
      (List.filter ((=) w) words)
    in w ^ ": " ^
       string_of_int count

  in String.concat ", "
    (List.map report ["a"; "the"])

in articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

```
let count words w = List.length
  (List.filter ((=) w) words) in

let report words w = w ^ ": " ^
  string_of_int (count words w) in

let articles words =
  String.concat ", "
    (List.map (report words)
     ["a"; "the"]) in

articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

Produces “a: 1, the: 2”

Applicative- and Normal-Order Evaluation

```
int p(int i) {
   printf("%d ", i);
   return i;
}

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}

q( p(1), 2, p(3) );
```

What does this print?

Applicative- vs. and Normal-Order

Most languages use applicative order.

Macro-like languages often use normal order.

```
#define p(x) (printf("%d ",x), x)

#define q(a,b,c) total = (a), \
   printf("%d ", (b)), \
   total += (c)

q( p(1), 2, p(3) );
```

Prints 1 2 3.

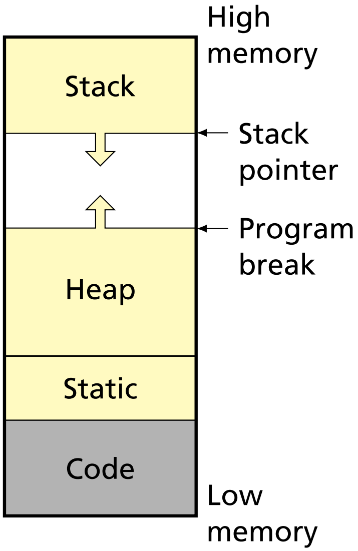
Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”

Storage Classes and Memory Layout

Stack: objects created/destroyed in last-in, first-out order

Heap: objects created/destroyed in any order; automatic garbage collection optional

Static: objects allocated at compile time; persist throughout run



Static Objects

```
class Example {
  public static final int a = 3;

  public void hello() {
    System.out.println("Hello");
  }
}
```

Examples

Static class variable

Code for hello method

String constant “Hello”

Information about the Example class

Advantages

Zero-cost memory management

Often faster access (address a constant)

No out-of-memory danger

Disadvantages

Size and number must be known beforehand

Wasteful if sharing is possible

Stack-Allocated Objects

Natural for supporting recursion.

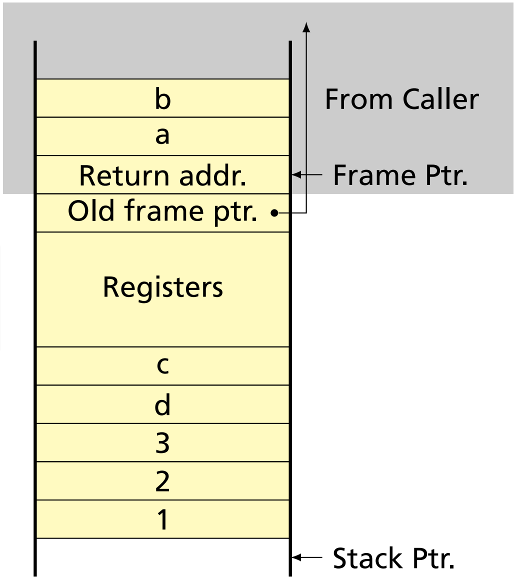
Idea: some objects persist from when a procedure is called to when it returns.

Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Each invocation of a procedure gets its own *frame* (*activation record*) where it stores its own local variables and bookkeeping information.

An Activation Record: The State Before Calling *bar*

```
int foo(int a, int b) {
  int c, d;
  bar(1, 2, 3);
}
```



Recursive Fibonacci

(Real C)

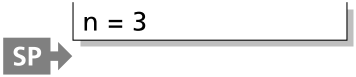
```
int fib(int n) {

  if (n<2)

    return 1;
  else
    return
        fib(n-1)
        +
        fib(n-2);

}
```

(Assembly-like C)

```
int fib(int n) {
     int tmp1, tmp2, tmp3;
     tmp1 = n < 2;
     if (!tmp1) goto L1;
     return 1;
L1:  tmp1 = n - 1;
     tmp2 = fib(tmp1);
L2:  tmp1 = n - 2;
     tmp3 = fib(tmp1);
L3:  tmp1 = tmp2 + tmp3;
     return tmp1;
}
```

$$\begin{tikzcd}
 & & \text{fib(3)} \arrow[dl, dash] \arrow[dr, dash] & \\
 & \text{fib(2)} \arrow[dl, dash] \arrow[dr, dash] & & \text{fib(1)} \\
\text{fib(1)} & & \text{fib(0)} &
\end{tikzcd}$$

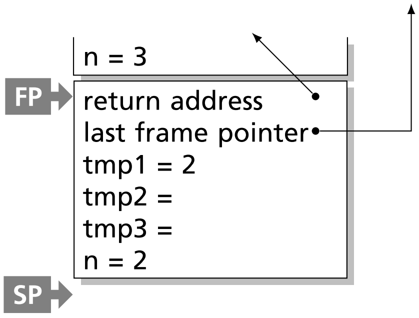
Executing fib(3)



```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

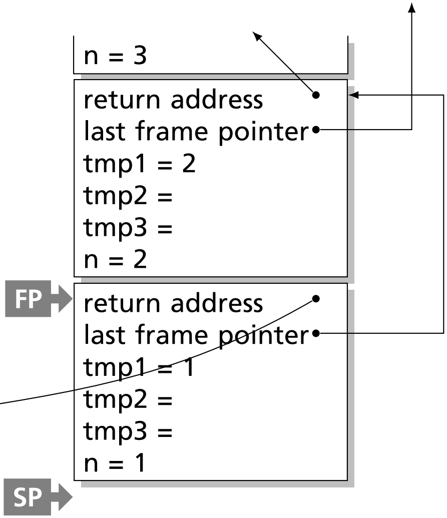
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



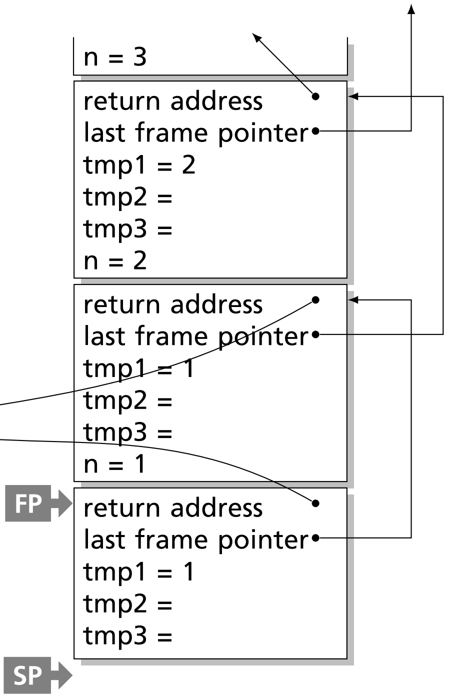
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



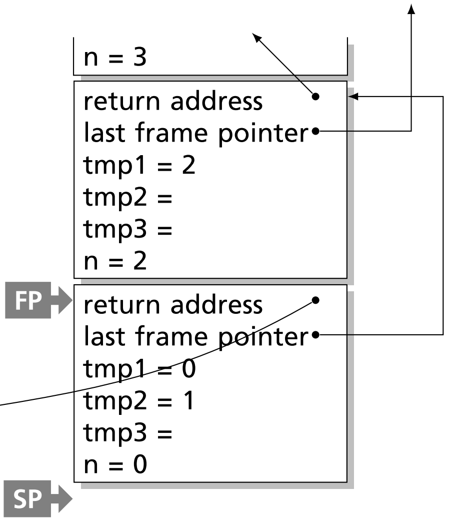
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



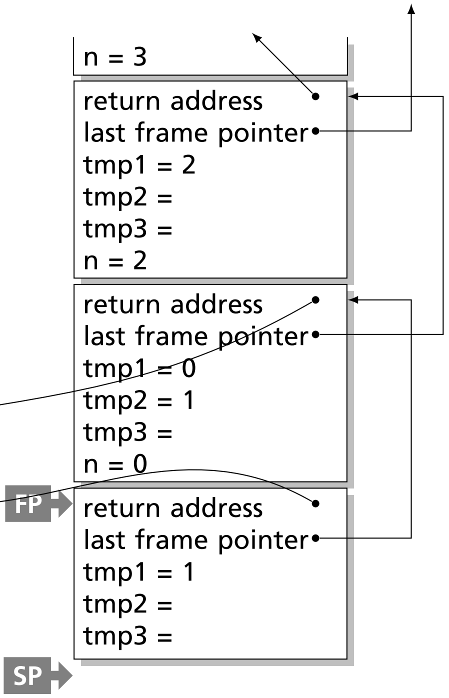
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



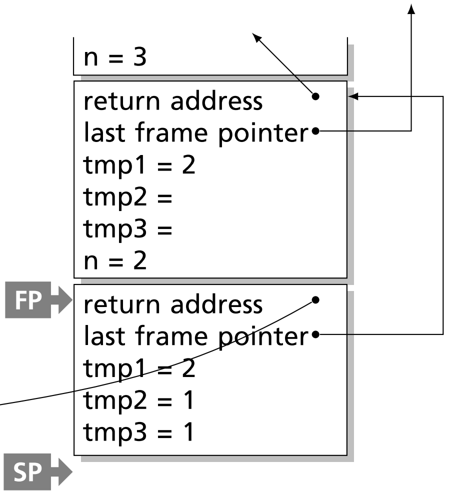
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

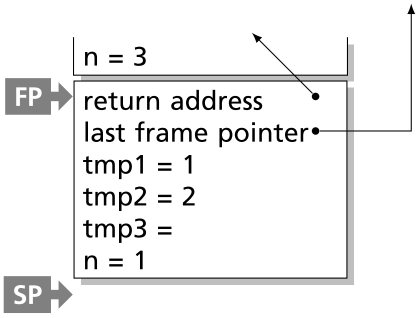


Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



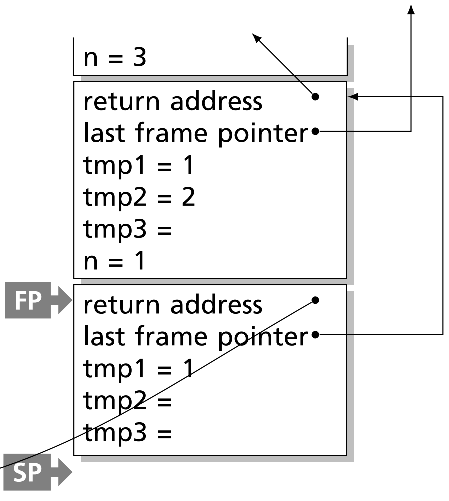
Executing fib(3)



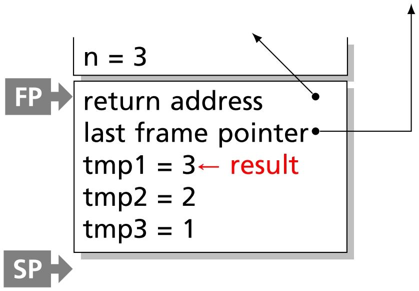
```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



Executing fib(3)

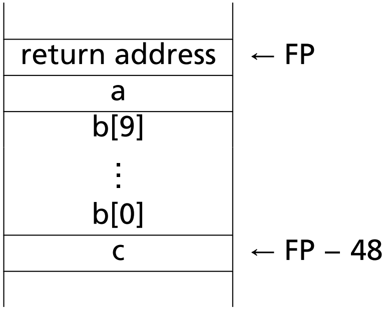


```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

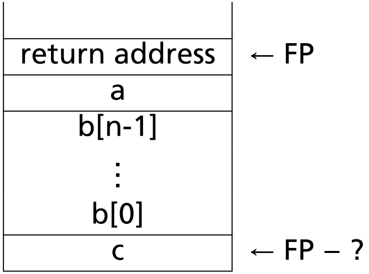
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```

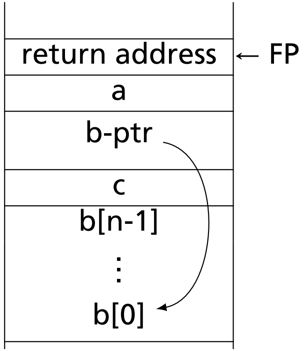


Doesn't work: generated code expects a fixed offset for c.
Even worse for multi-dimensional arrays.

Allocating Variable-Sized Arrays

As always:
add a level of indirection

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```



Variables remain constant offset from frame pointer.

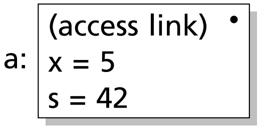
Nesting Function Definitions

```
let articles words =

  let report w =

    let count = List.length
      (List.filter ((=) w) words)
    in w ^ ": " ^
       string_of_int count

  in String.concat ", "
    (List.map report ["a"; "the"])

in articles
     ["the"; "plt"; "class"; "is";
      "a"; "pain"; "in";
      "the"; "butt"]
```

```
let count words w = List.length
  (List.filter ((=) w) words) in

let report words w = w ^ ": " ^
  string_of_int (count words w) in

let articles words =
  String.concat ", "
    (List.map (report words)
     ["a"; "the"]) in

articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

Produces "a: 1, the: 2"

Implementing Nested Functions with Access Links



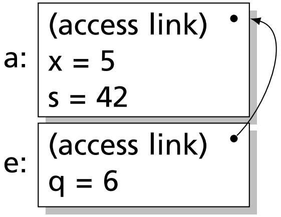
```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

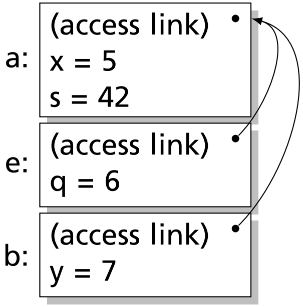
What does “a 5 42” give?



Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

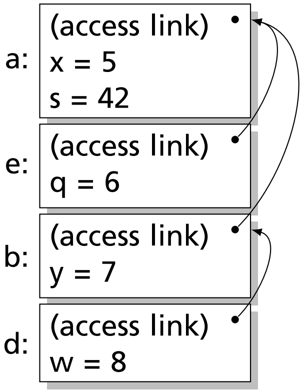
What does “a 5 42” give?



Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

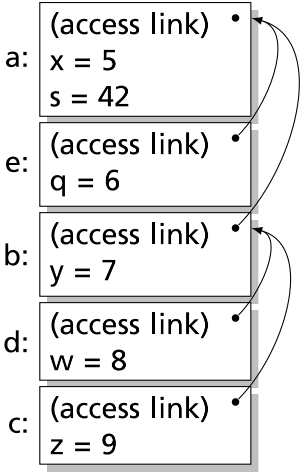
What does “a 5 42” give?



Implementing Nested Functions with Access Links

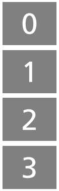
```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

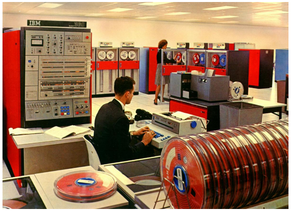
What does “a 5 42” give?



Layout of Records and Unions

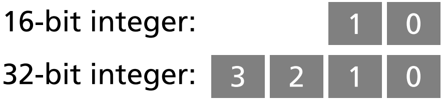
Modern processors have byte-addressable memory.





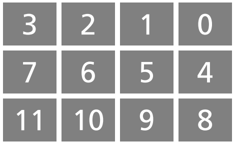
The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

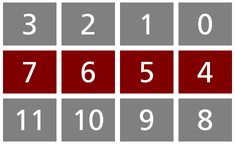


Layout of Records and Unions

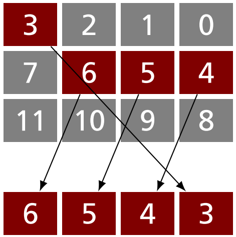
Modern memory systems read data in 32-, 64-, or 128-bit chunks:



Reading an aligned 32-bit value is fast: a single operation.



It is harder to read an unaligned value: two reads plus shifting



SPARC and ARM prohibit unaligned accesses

MIPS has special unaligned load/store instructions

x86, 68k run more slowly with unaligned accesses

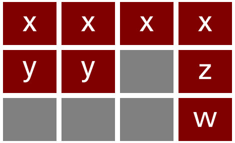
Padding

To avoid unaligned accesses, the C compiler pads the layout of unions and records.

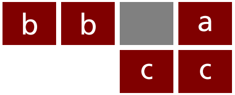
Rules:

- Each n-byte object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte object must be of size mn for some integer m (aligned even when arrayed).

```
struct padded {
  int x;    /* 4 bytes */
  char z;   /* 1 byte  */
  short y;  /* 2 bytes */
  char w;   /* 1 byte  */
};
```



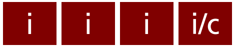
```
struct padded {
  char a;   /* 1 byte  */
  short b;  /* 2 bytes */
  short c;  /* 2 bytes */
};
```



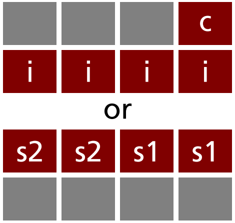
Unions

A C *struct* has a separate space for each field; a C *union* shares one space among all fields

```
union intchar {
  int i;    /* 4 bytes */
  char c;   /* 1 byte  */
};
```



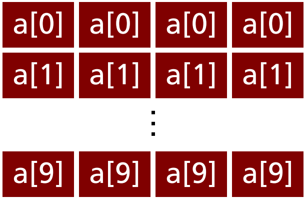
```
union twostructs {
  struct {
    char c;    /* 1 byte */
    int i;     /* 4 bytes */
  } a;
  struct {
    short s1; /* 2 bytes */
    short s2; /* 2 bytes */
  } b;
};
```



Arrays

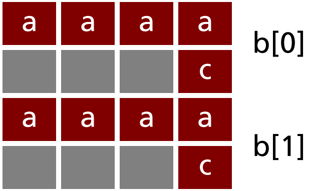
Basic policy in C: an array is just one object after another in memory.

```
int a[10];
```



This is why you need padding at the end of *structs*.

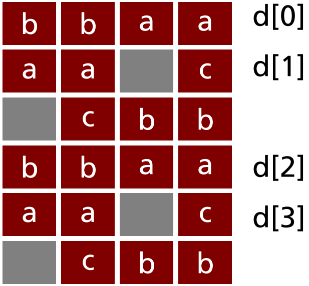
```
struct {
  int a;
  char c;
} b[2];
```



Arrays and Aggregate types

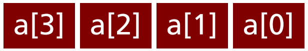
The largest primitive type dictates the alignment

```
struct {
  short a;
  short b;
  char c;
} d[4];
```

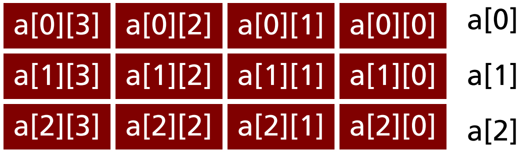


Arrays of Arrays

```
char a[4];
```



```
char a[3][4];
```



Heap-Allocated Storage

Static works when you know everything beforehand and always need it.

Stack enables, but also requires, recursive behavior.

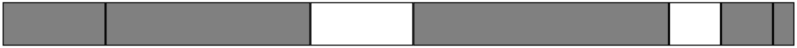
A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)

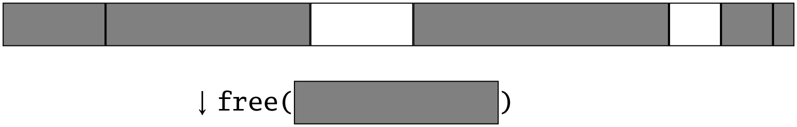
Dynamic Storage Allocation in C

```
struct point {
   int x, y;
};

int play_with_points(int n)
{
  int i;
  struct point *points;

  points = malloc(n * sizeof(struct point));

  for ( i = 0 ; i < n ; i++ ) {
    points[i].x = random();
    points[i].y = random();
  }

  /* do something with the array */

  free(points);
}
```

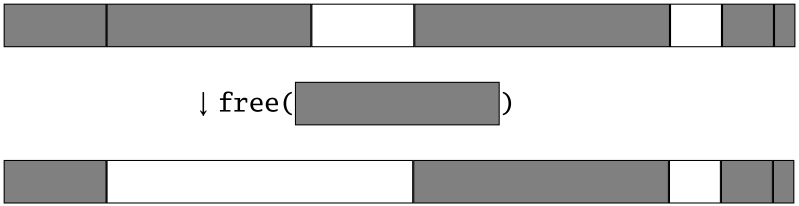
Dynamic Storage Allocation



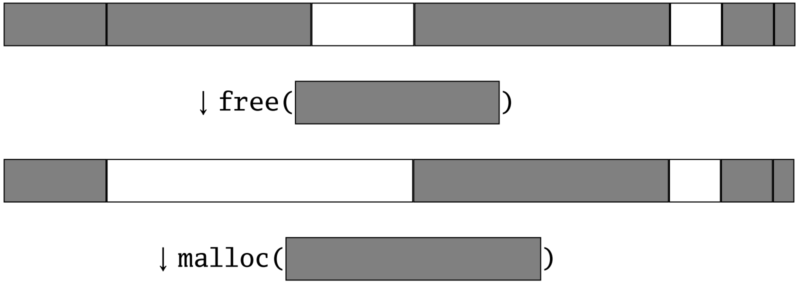
Dynamic Storage Allocation



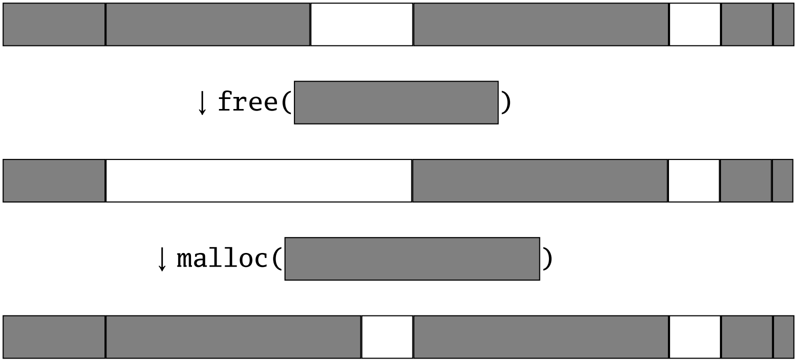
Dynamic Storage Allocation



Dynamic Storage Allocation



Dynamic Storage Allocation



Dynamic Storage Allocation

Rules:

- Each allocated block contiguous (no holes)
- Blocks stay fixed once allocated

`malloc()`

- Find an area large enough for requested block
- Mark memory as allocated

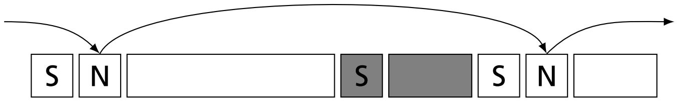
`free()`

- Mark the block as unallocated

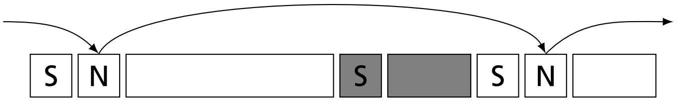
Simple Dynamic Storage Allocation

- Maintaining information about free memory
  - Simplest: Linked list
- The algorithm for locating a suitable block
  - Simplest: First-fit
- The algorithm for freeing an allocated block
  - Simplest: Coalesce adjacent free blocks

Simple Dynamic Storage Allocation

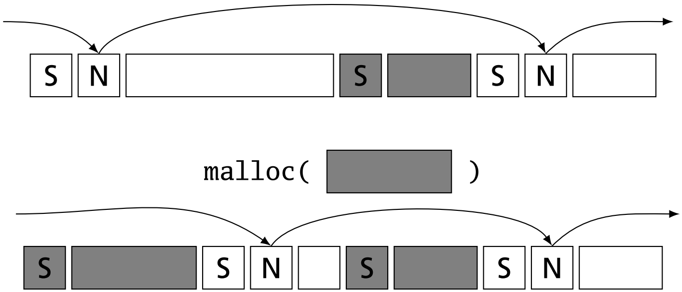


Simple Dynamic Storage Allocation

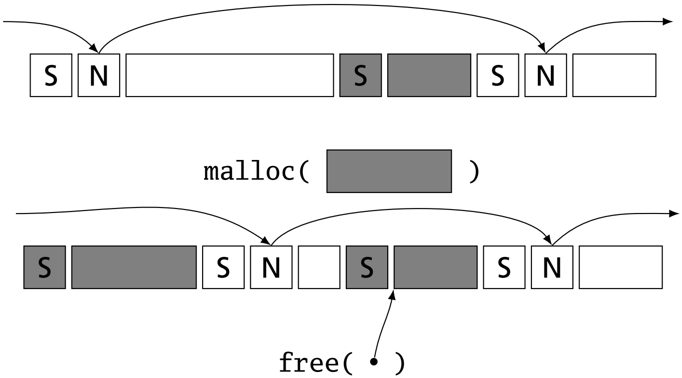


malloc( )

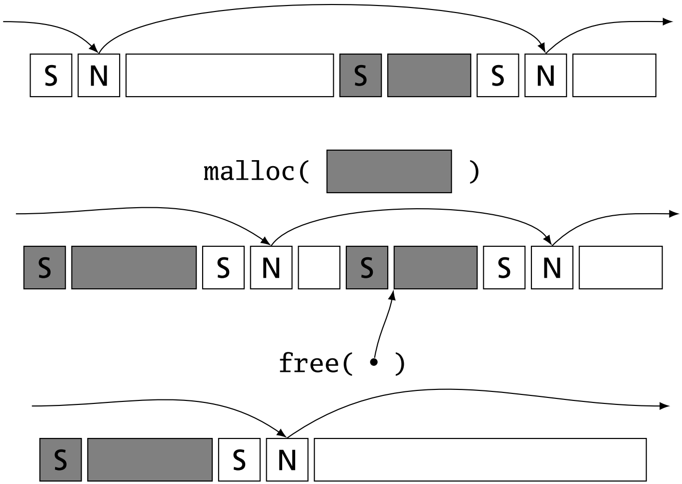
Simple Dynamic Storage Allocation



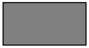
Simple Dynamic Storage Allocation

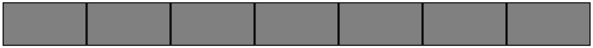


Simple Dynamic Storage Allocation

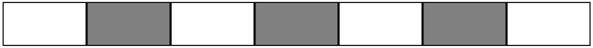


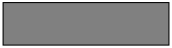
Fragmentation

`malloc(`  `)` seven times give



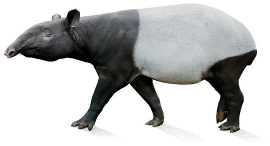
`free()` four times gives



`malloc(`  `)` ?

Need more memory; can’t use fragmented memory.

Zebra

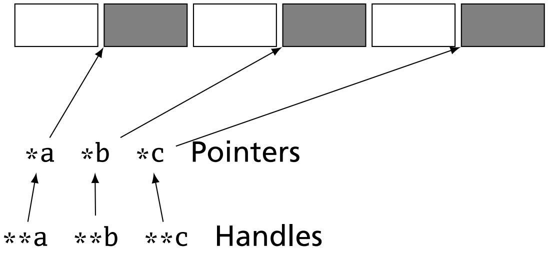


Tapir

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”

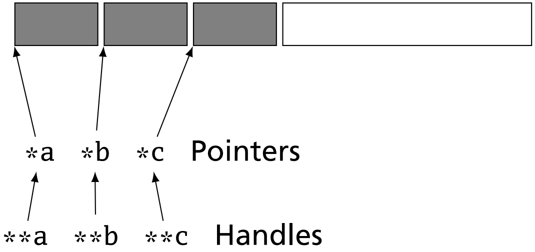


The original Macintosh did this to save memory.

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through "handles."



The original Macintosh did this to save memory.

Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

Now common: Java, C#, Javascript, Python, Ruby, OCaml and most functional languages

Advantages

Much easier for the programmer

Greatly improves reliability: no memory leaks, double-freeing, or other memory management errors

Disadvantages

Slower, sometimes unpredictably so

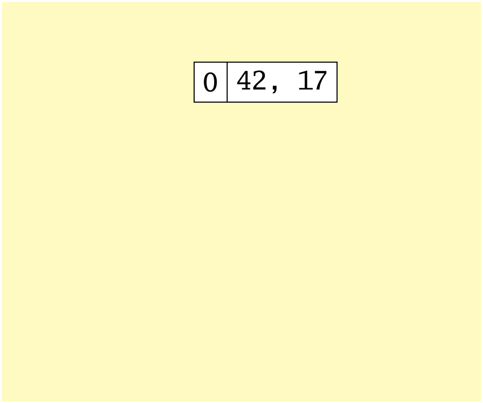
May consume more memory

Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

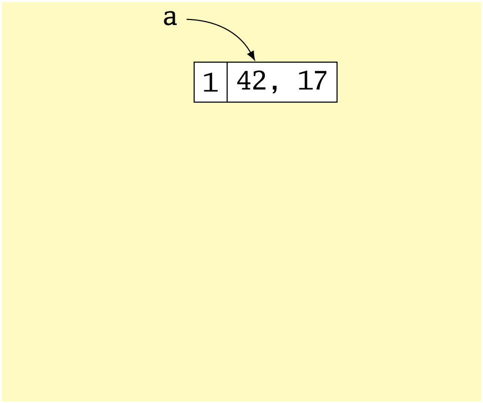


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

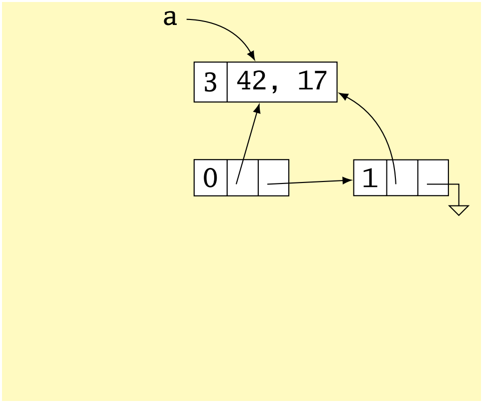


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

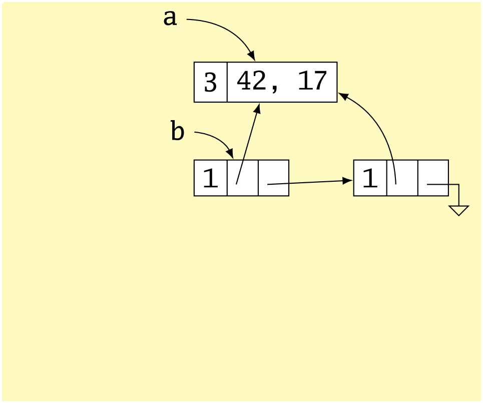


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

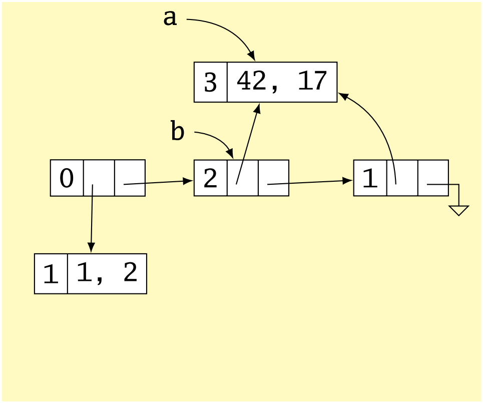


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

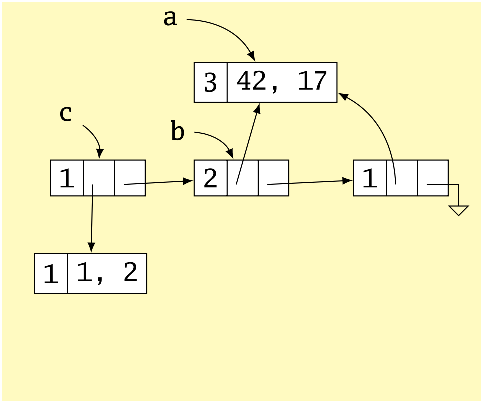


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

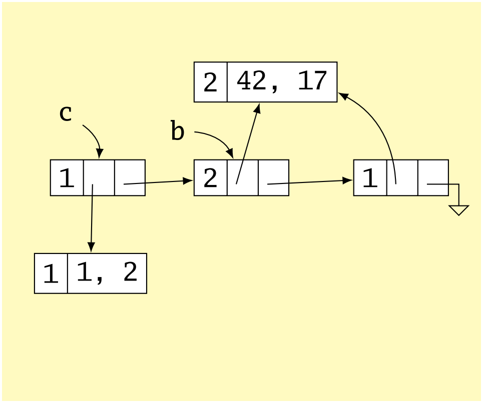


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

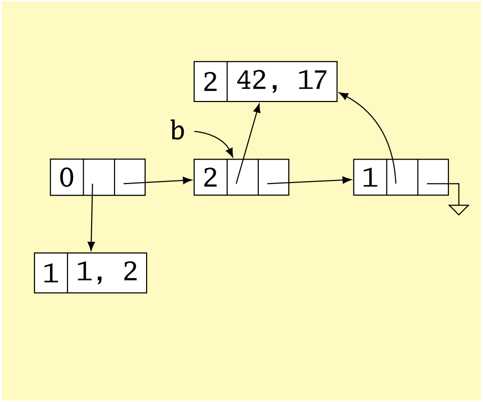


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

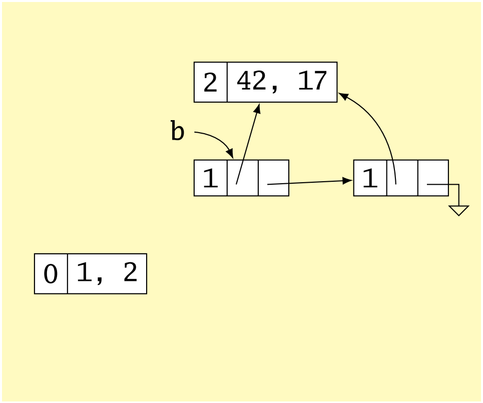


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

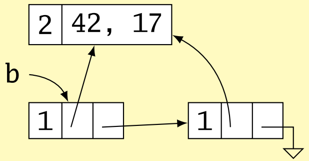


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```



Issues with Reference Counting

Circular structures defy reference counting:

$$\begin{tikzcd}
\boxed{a} \arrow[r, bend left] & \boxed{b} \arrow[l, bend left]
\end{tikzcd}$$

Neither is reachable, yet both have non-zero reference counts.

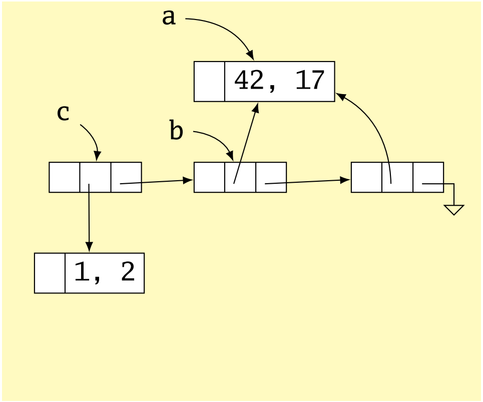
High overhead (must update counts constantly), although incremental

Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

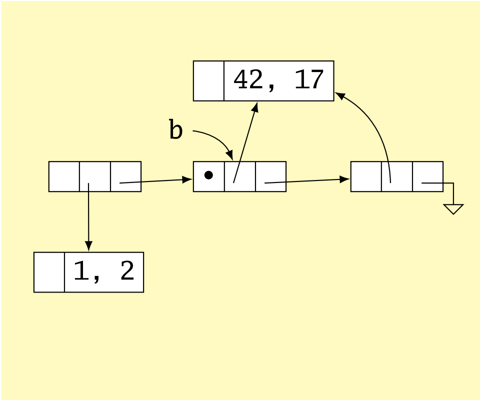


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

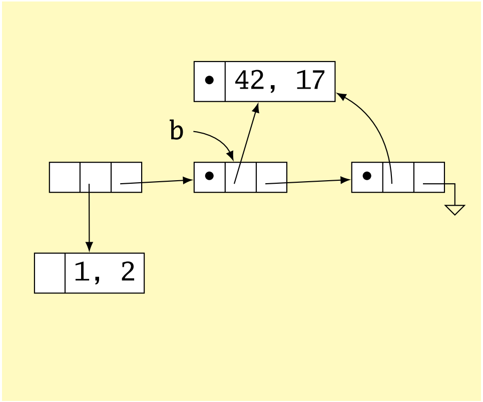


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

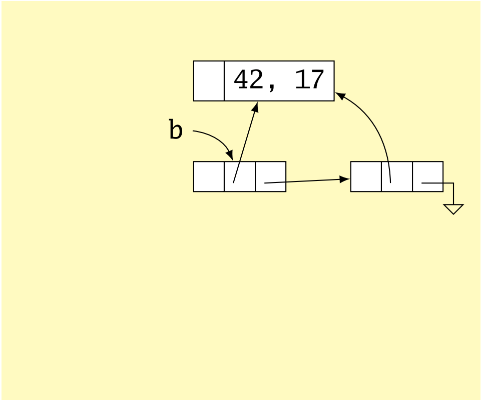


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```



Mark-and-Sweep

Mark-and-sweep is faster overall; may induce big pauses

Mark-and-compact variant also moves or copies reachable objects to eliminate fragmentation

Incremental garbage collectors try to avoid doing everything at once

Most objects die young; generational garbage collectors segregate heap objects by age

Parallel garbage collection tricky

Real-time garbage collection tricky

Single Inheritance

Simple: Add new fields to end of the object

Fields in base class always at same offset in derived class (compiler never reorders)

Consequence: Derived classes can never remove fields

C++

```
class Shape {
  double x, y;
};

class Box : Shape {
  double h, w;
};


class Circle : Shape {
  double r;
};
```

Equivalent C

```
struct Shape {
  double x, y;
};

struct Box {
  double x, y;
  double h, w;
};

struct Circle {
  double x, y;
  double r;
};
```

Virtual Functions

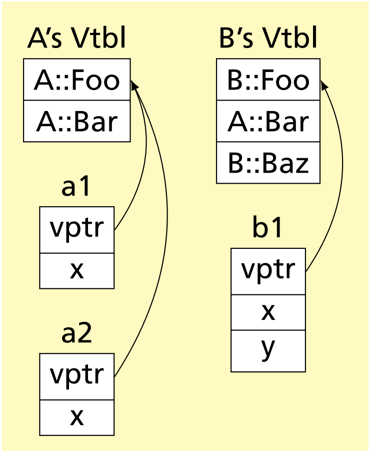
```
class Shape {
  virtual void draw(); // Invoked by object's run-time class
};                     // not its compile-time type.

class Line : public Shape {
  void draw();
}

class Arc : public Shape {
  void draw();
};

Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw();   // Invoke Line::draw()
s[1]->draw();   // Invoke Arc::draw()
```

Virtual Functions

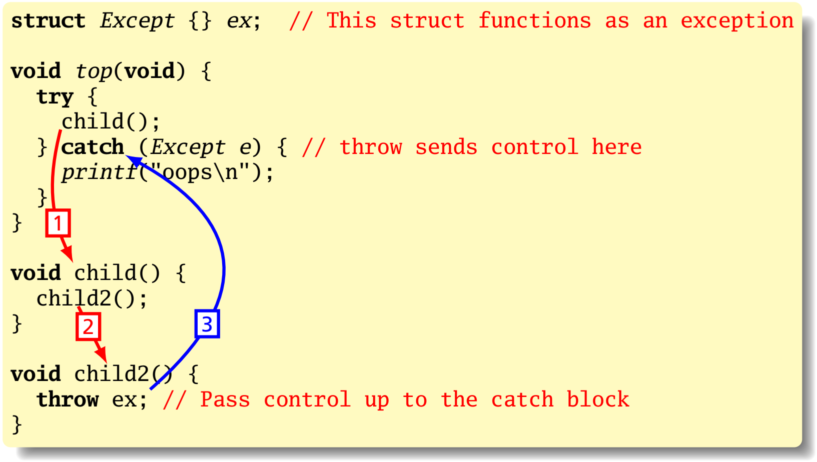
Trick: add to each object a pointer to the virtual table for its type, filled with pointers to the virtual functions.

Like the objects themselves, the virtual table for each derived type begins identically.

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};

struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1;
A a2;
B b1;
```

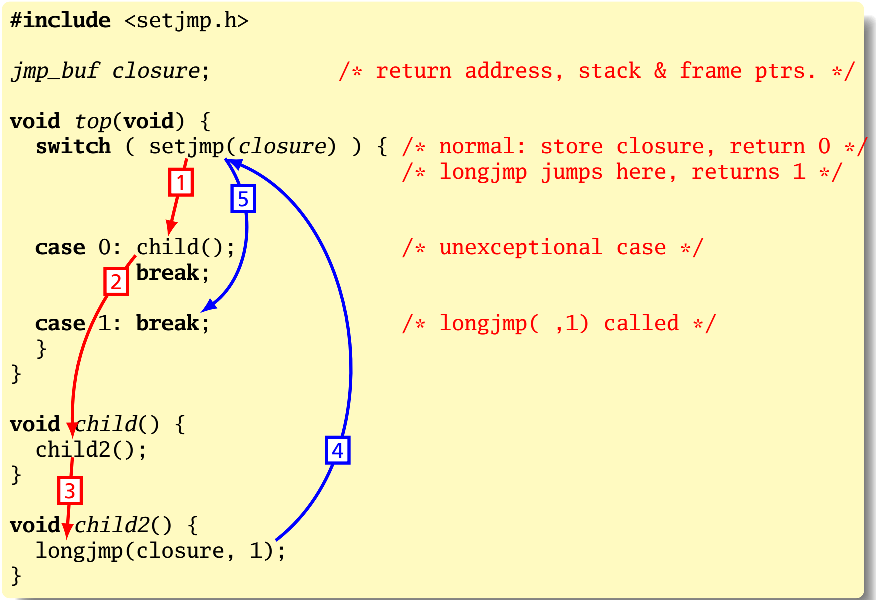


C++'s Exceptions



```
struct Except {} ex;  // This struct functions as an exception

void top(void) {
  try {
    child();
  } catch (Except e) { // throw sends control here
    printf("oops\n");
  }
}

void child() {
  child2();
}

void child2() {
  throw ex; // Pass control up to the catch block
}
```

C's setjmp/longjmp: Idiosyncratic Exceptions



```c
#include <setjmp.h>

jmp_buf closure;          /* return address, stack & frame ptrs. */

void top(void) {
  switch ( setjmp(closure) ) { /* normal: store closure, return 0 */
                               /* longjmp jumps here, returns 1 */


  case 0: child();             /* unexceptional case */
          break;

  case 1: break;               /* longjmp( ,1) called */
  }
}

void child() {
  child2();
}

void child2() {
  longjmp(closure, 1);
}
```

Implementing Exceptions

One way: maintain a stack of exception handlers

```
try {

  child();


} catch (Ex e) {
  foo();
}

void child() {
  child2();
}

void child2() {
  throw ex;
}
```

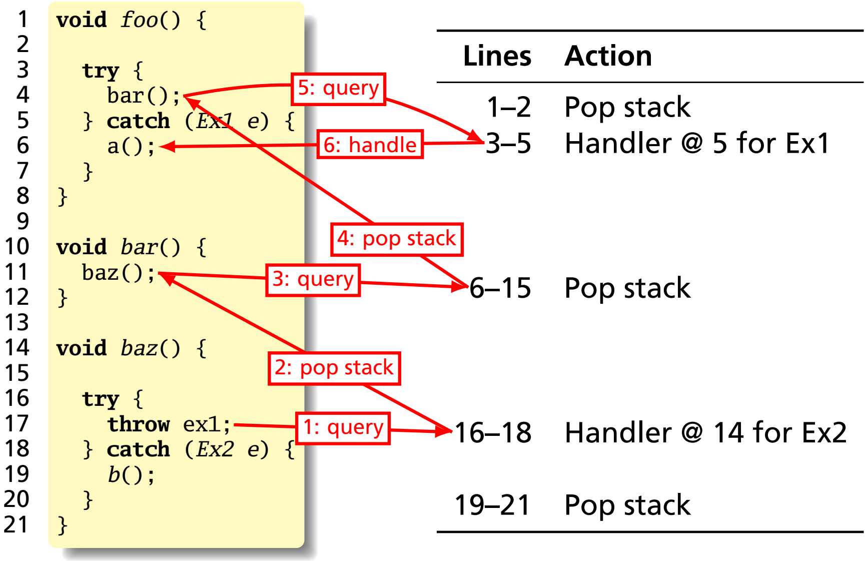
```
  push(Ex, Handler); // Push handler on stack

  child();
  pop();             // Normal termination
  goto Exit;         // Jump over "catch"
Handler:
  foo();             // Body of "catch"
Exit:

void child() {
  child2();
}

void child2() {
  throw(ex);       // Unroll stack; find handler
}
```

Incurs overhead, even when no exceptions thrown

Implementing Exceptions with Tables

Q: When an exception is *thrown*, where was the last *try*?

A: Consult a table: relevant handler or “pop” for every PC



```
 1  void foo() {
 2
 3    try {
 4      bar();
 5    } catch (Ex1 e) {
 6      a();
 7    }
 8  }
 9
10  void bar() {
11    baz();
12  }
13
14  void baz() {
15
16    try {
17      throw ex1;
18    } catch (Ex2 e) {
19      b();
20    }
21  }
```

Lines	Action
1–2	Pop stack
3–5	Handler @ 5 for Ex1
6–15	Pop stack
16–18	Handler @ 14 for Ex2
19–21	Pop stack

Stack-Based IR: Java Bytecode

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
# javap -c Gcd

Method int gcd(int, int)
   0 goto 19

   3 iload_1        // Push a
   4 iload_2        // Push b
   5 if_icmple 15   // if a <= b goto 15

   8 iload_1        // Push a
   9 iload_2        // Push b
  10 isub           // a - b
  11 istore_1       // Store new a
  12 goto 19

  15 iload_2        // Push b
  16 iload_1        // Push a
  17 isub           // b - a
  18 istore_2       // Store new b

  19 iload_1        // Push a
  20 iload_2        // Push b
  21 if_icmpne 3    // if a != b goto 3

  24 iload_1        // Push a
  25 ireturn        // Return a
```

Stack-Based IRs

Advantages:

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers
- Often compact

Disadvantages:

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization

Register-Based IR: Mach SUIF

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
gcd:
gcd._gcdTmp0:
  sne   $vr1.s32 <- gcd.a,gcd.b
  seq   $vr0.s32 <- $vr1.s32,0
  btrue $vr0.s32,gcd._gcdTmp1   // if !(a != b) goto Tmp1

  sl    $vr3.s32 <- gcd.b,gcd.a
  seq   $vr2.s32 <- $vr3.s32,0
  btrue $vr2.s32,gcd._gcdTmp4   // if !(a < b) goto Tmp4

  mrk   2, 4     // Line number 4
  sub   $vr4.s32 <- gcd.a,gcd.b
  mov   gcd._gcdTmp2 <- $vr4.s32
  mov   gcd.a <- gcd._gcdTmp2   // a = a - b
  jmp   gcd._gcdTmp5
gcd._gcdTmp4:
  mrk   2, 6
  sub   $vr5.s32 <- gcd.b,gcd.a
  mov   gcd._gcdTmp3 <- $vr5.s32
  mov   gcd.b <- gcd._gcdTmp3   // b = b - a
gcd._gcdTmp5:
  jmp   gcd._gcdTmp0

gcd._gcdTmp1:
  mrk   2, 8
  ret   gcd.a    // Return a
```

Register-Based IRs

Most common type of IR

Advantages:

- Better representation for register machines
- Dataflow is usually clear

Disadvantages:

- Slightly harder to synthesize from code
- Less compact
- More complicated to interpret

Optimization In Action

```
int gcd(int a, int b) {
  while (a != b) {
    if (a < b) b -= a;
    else a -= b;
  }
  return a;
}
```

GCC on SPARC

```
gcd:   save %sp, -112, %sp
       st   %i0, [%fp+68]
       st   %i1, [%fp+72]
.LL2:  ld   [%fp+68], %i1
       ld   [%fp+72], %i0
       cmp  %i1, %i0
       bne  .LL4
       nop
       b    .LL3
       nop
.LL4:  ld   [%fp+68], %i1
       ld   [%fp+72], %i0
       cmp  %i1, %i0
       bge  .LL5
       nop
       ld   [%fp+72], %i0
       ld   [%fp+68], %i1
       sub  %i0, %i1, %i0
       st   %i0, [%fp+72]
       b    .LL2
       nop
.LL5:  ld   [%fp+68], %i0
       ld   [%fp+72], %i1
       sub  %i0, %i1, %i0
       st   %i0, [%fp+68]
       b    .LL2
       nop
.LL3:  ld   [%fp+68], %i0
       ret
       restore
```

GCC -O7 on SPARC

```
gcd:   cmp    %o0, %o1
       be     .LL8
       nop
.LL9:  bge,a  .LL2
       sub    %o0, %o1, %o0
       sub    %o1, %o0, %o1
.LL2:  cmp    %o0, %o1
       bne    .LL9
       nop
.LL8:  retl
       nop
```

Typical Optimizations

- Folding constant expressions
  1+3 → 4
- Removing dead code
  if (0) {...} → nothing
- Moving variables from memory to registers

```
ld   [%fp+68], %i1
sub  %i0, %i1, %i0  → sub   %o1, %o0, %o1
st   %i0, [%fp+72]
```

- Removing unnecessary data movement
- Filling branch delay slots (Pipelined RISC processors)
- Common subexpression elimination

Machine-Dependent vs. -Independent Optimization

No matter what the machine is, folding constants and eliminating dead code is always a good idea.

```
a = c + 5 + 3;
if (0 + 3) {
  b = c + 8;            →   b = a = c + 8;
}
```

However, many optimizations are processor-specific:

- Register allocation depends on how many registers the machine has
- Not all processors have branch delay slots to fill
- Each processor’s pipeline is a little different

Basic Blocks

```
int gcd(int a, int b) {
  while (a != b) {
    if (a < b) b -= a;
    else a -= b;
  }
  return a;
}
```

lower $\longrightarrow$

```
A: sne t, a, b
   bz  E, t
   slt t, a, b
   bnz B, t
   sub b, b, a
   jmp C
B: sub a, a, b
C: jmp A
E: ret a
```

split $\longrightarrow$

```
A: sne t, a, b
   bz  E, t

   slt t, a, b
   bnz B, t

   sub b, b, a
   jmp C

B: sub a, a, b

C: jmp A

E: ret a
```

The statements in a basic block all run if the first one does.

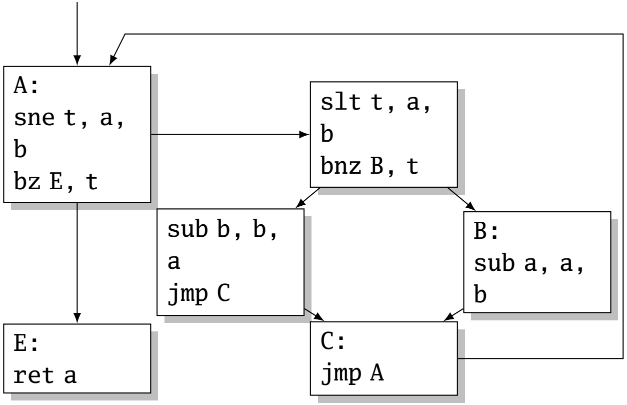
Starts with a statement following a conditional branch or is a branch target.

Usually ends with a control-transfer statement.

Control-Flow Graphs

A CFG illustrates the flow of control among basic blocks.

```
A:
sne t, a, b
bz  E, t

slt t, a, b
bnz B, t

sub b, b, a
jmp C

B:
sub a, a, b

C:
jmp A

E:
ret a
```



Lambda Expressions

Function application written in prefix form. “Add four and five” is

$$(+\ 4\ 5)$$

Evaluation: select a *redex* and evaluate it:

$$\begin{aligned}(+\ (*\ 5\ 6)\ (*\ 8\ 3)) &\rightarrow (+\ 30\ (*\ 8\ 3))\\ &\rightarrow (+\ 30\ 24)\\ &\rightarrow 54\end{aligned}$$

Often more than one way to proceed:

$$\begin{aligned}(+\ (*\ 5\ 6)\ (*\ 8\ 3)) &\rightarrow (+\ (*\ 5\ 6)\ 24)\\ &\rightarrow (+\ 30\ 24)\\ &\rightarrow 54\end{aligned}$$

Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

Function Application and Currying

Function application is written as juxtaposition:

$$f\ x$$

Every function has exactly one argument.
Multiple-argument functions, e.g., $+$, are represented by *currying*, named after Haskell Brooks Curry (1900–1982). So,

$$(+\ x)$$

is the function that adds x to its argument.

Function application associates left-to-right:

$$\begin{aligned}(+\ 3\ 4) &= ((+\ 3)\ 4)\\ &\rightarrow 7\end{aligned}$$

Lambda Abstraction

The only other thing in the lambda calculus is *lambda abstraction*: a notation for defining unnamed functions.

$$(\lambda x\ .\ +\ x\ 1)$$

$$\begin{array}{cccccccc} (& \lambda & x & . & + & x & 1 &) \\ & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \\ & \text{That function of} & x & \text{that} & \text{adds} & x & \text{to } 1 & \end{array}$$

The Syntax of the Lambda Calculus

$$
\begin{array}{lcl}
\textit{expr} & ::= & \textit{expr}\ \textit{expr} \\
 & | & \lambda\ \textit{variable}\ .\ \textit{expr} \\
 & | & \textit{constant} \\
 & | & \textit{variable} \\
 & | & (\textit{expr})
\end{array}
$$

Constants are numbers and built-in functions;
variables are identifiers.

Beta-Reduction

Evaluation of a lambda abstraction—*beta-reduction*—is just substitution:

$$\begin{aligned}(\lambda x\ .\ +\ x\ 1)\ 4 &\rightarrow (+\ 4\ 1)\\ &\rightarrow 5\end{aligned}$$

The argument may appear more than once

$$\begin{aligned}(\lambda x\ .\ +\ x\ x)\ 4 &\rightarrow (+\ 4\ 4)\\ &\rightarrow 8\end{aligned}$$

or not at all

$$(\lambda x\ .\ 3)\ 5 \rightarrow 3$$

Free and Bound Variables

$$(\lambda x\ .\ +\ x\ y)\ 4$$

Here, x is like a function argument but y is like a global variable.

Technically, x *occurs bound* and y *occurs free* in

$$(\lambda x\ .\ +\ x\ y)$$

However, both x and y occur free in

$$(+\ x\ y)$$

Beta-Reduction More Formally

$$(\lambda x\ .\ E)\ F \ \rightarrow_\beta\ E'$$

where E' is obtained from E by replacing every instance of x that appears free in E with F.

The definition of free and bound mean variables have scopes. Only the rightmost x appears free in

$$(\lambda x\ .\ +\ (-\ x\ 1))\ x\ 3$$

so

$$\begin{aligned}(\lambda x\ .\ (\lambda x\ .\ +\ (-\ x\ 1))\ x\ 3)\ 9 &\rightarrow (\lambda\ x\ .\ +\ (-\ x\ 1))\ 9\ 3\\ &\rightarrow\ +\ (-\ 9\ 1)\ 3\\ &\rightarrow\ +\ 8\ 3\\ &\rightarrow\ 11\end{aligned}$$

Alpha-Conversion

One way to confuse yourself less is to do α-conversion: renaming a λ argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

$$\begin{aligned}
(\lambda x \,.\, (\lambda x \,.\, + \ (- \ x \ 1)) \ x \ 3) \ 9 &\leftrightarrow (\lambda x \,.\, (\lambda y \,.\, + \ (- \ y \ 1)) \ x \ 3) \ 9 \\
&\to ((\lambda y \,.\, + \ (- \ y \ 1)) \ 9 \ 3) \\
&\to (+ \ (- \ 9 \ 1) \ 3) \\
&\to (+ \ 8 \ 3) \\
&\to 11
\end{aligned}$$

Beta-Abstraction and Eta-Conversion

Running β-reduction in reverse, leaving the "meaning" of a lambda expression unchanged, is called *beta abstraction*:

$$+\ 4\ 1 \leftarrow (\lambda x\ .\ +\ x\ 1)\ 4$$

Eta-conversion is another type of conversion that leaves "meaning" unchanged:

$$(\lambda x\ .\ +\ 1\ x) \leftrightarrow_{\eta} (+\ 1)$$

Formally, if F is a function in which x does not occur free,

$$(\lambda x\ .\ F\ x) \leftrightarrow_{\eta} F$$

Reduction Order

The order in which you reduce things can matter.

$$(\lambda x \,.\, \lambda y \,.\, y) \big((\lambda z \,.\, z\ z)\ (\lambda z \,.\, z\ z)\big)$$

Two things can be reduced:

$$(\lambda z \,.\, z\ z)\ (\lambda z \,.\, z\ z)$$

$$(\lambda x \,.\, \lambda y \,.\, y)\ (\cdots)$$

However,

$$(\lambda z \,.\, z\ z)\ (\lambda z \,.\, z\ z) \rightarrow (\lambda z \,.\, z\ z)\ (\lambda z \,.\, z\ z)$$

$$(\lambda x \,.\, \lambda y \,.\, y)\ (\cdots) \rightarrow (\lambda y \,.\, y)$$

Normal Form

A lambda expression that cannot be β-reduced is in *normal form*. Thus,

$$\lambda y \,.\, y$$

is the normal form of

$$(\lambda x \,.\, \lambda y \,.\, y)\ \big((\lambda z \,.\, z\ z)\ (\lambda z \,.\, z\ z)\big)$$

Not everything has a normal form. E.g.,

$$(\lambda z \,.\, z\ z)\ (\lambda z \,.\, z\ z)$$

can only be reduced to itself, so it never produces an non-reducible expression.

Normal Form

Can a lambda expression have more than one normal form?

Church-Rosser Theorem I: If $E_1 \leftrightarrow E_2$, then there exists an expression E such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$.

Corollary. No expression may have two distinct normal forms.

Proof. Assume E_1 and E_2 are distinct normal forms for E: $E \leftrightarrow E_1$ and $E \leftrightarrow E_2$. So $E_1 \leftrightarrow E_2$ and by the Church-Rosser Theorem I, there must exist an F such that $E_1 \rightarrow F$ and $E_2 \rightarrow F$. However, since E_1 and E_2 are in normal form, $E_1 = F = E_2$, a contradiction.

Normal-Order Reduction

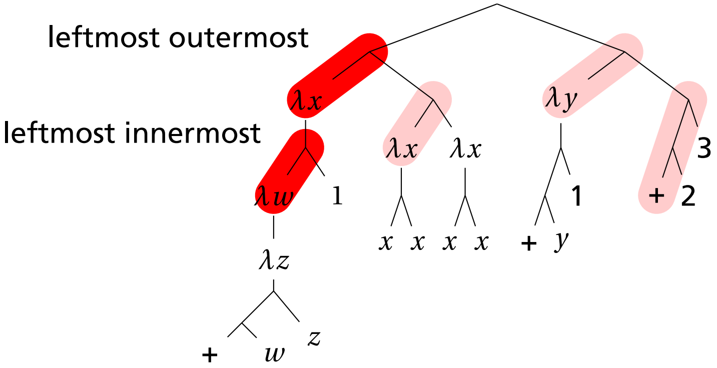
Not all expressions have normal forms, but is there a reliable way to find the normal form if it exists?

Church-Rosser Theorem II: If $E_1 \rightarrow E_2$ and E_2 is in normal form, then there exists a *normal order* reduction sequence from E_1 to E_2.

Normal order reduction: reduce the leftmost outermost redex.

Normal-Order Reduction

$$\Big(\big(\lambda x\,.\,((\lambda w\,.\,\lambda z\,.\,+\;w\;z)\;1)\big)\;\big((\lambda x\,.\,x\;x)\;(\lambda x\,.\,x\;x)\big)\Big)\;\big((\lambda y\,.\,+\;y\;1)\;(+\;2\;3)\big)$$



Recursion

Where is recursion in the lambda calculus?

$$FAC = \left(\lambda n\, .\, IF\ (=\ n\ 0)\ 1\ \left(*\ n\ \left(FAC\ (-\ n\ 1)\right)\right)\right)$$

This does not work: functions are unnamed in the lambda calculus. But it is possible to express recursion *as a function*.

$$\begin{array}{rcl} FAC & = & (\lambda n\, .\, \ldots\ FAC\ \ldots) \\ & \leftarrow_\beta & (\lambda f\, .\, (\lambda n\, .\, \ldots\ f\ \ldots))\ FAC \\ & = & H\ FAC \end{array}$$

That is, the factorial function, FAC, is a *fixed point* of the (non-recursive) function H:

$$H = \lambda f\, .\, \lambda n\, .\, IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$$

Recursion

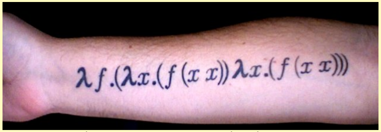
Let's invent a function Y that computes FAC from H, i.e., $FAC = Y\ H$:

$$\begin{array}{rcl} FAC & = & H\ FAC \\ Y\ H & = & H\ (Y\ H) \end{array}$$

$$\begin{array}{rcl}
FAC\ 1 & = & Y\ H\ 1 \\
 & = & H\ (Y\ H)\ 1 \\
 & = & (\lambda f\ .\ \lambda n\ .\ IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 1 \\
 & \rightarrow & (\lambda n\ .\ IF\ (=\ n\ 0)\ 1\ (*\ n\ ((Y\ H)\ (-\ n\ 1))))\ 1 \\
 & \rightarrow & IF\ (=\ 1\ 0)\ 1\ (*\ 1\ ((Y\ H)\ (-\ 1\ 1))) \\
 & \rightarrow & *\ 1\ (Y\ H\ 0) \\
 & = & *\ 1\ (H\ (Y\ H)\ 0) \\
 & = & *\ 1\ ((\lambda f\ .\ \lambda n\ .\ IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 0) \\
 & \rightarrow & *\ 1\ ((\lambda n\ .\ IF\ (=\ n\ 0)\ 1\ (*\ n\ (Y\ H\ (-\ n\ 1))))\ 0) \\
 & \rightarrow & *\ 1\ (IF\ (=\ 0\ 0)\ 1\ (*\ 0\ (Y\ H\ (-\ 0\ 1)))) \\
 & \rightarrow & *\ 1\ 1 \\
 & \rightarrow & 1
\end{array}$$

The Y Combinator

Here's the eye-popping part: Y can be a simple lambda expression.

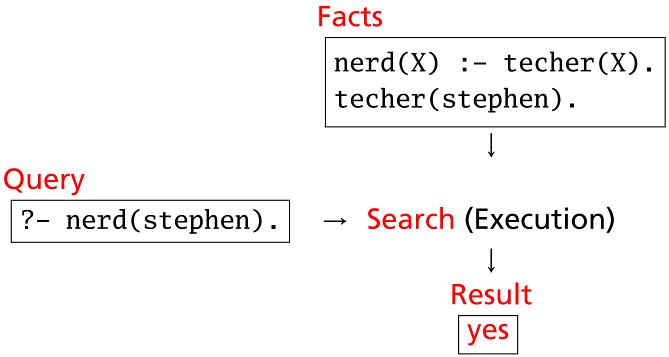
$Y =$ 

$$= \lambda f\,.\,\big(\lambda x\,.\,f\ (x\ x)\big)\ \big(\lambda x\,.\,f\ (x\ x)\big)$$

$$\begin{aligned}
Y\ H &= \Big(\lambda f\,.\,\big(\lambda x\,.\,f\ (x\ x)\big)\ \big(\lambda x\,.\,f\ (x\ x)\big)\Big)\ H \\
&\rightarrow \big(\lambda x\,.\,H\ (x\ x)\big)\ \big(\lambda x\,.\,H\ (x\ x)\big) \\
&\rightarrow H\ \Big(\big(\lambda x\,.\,H\ (x\ x)\big)\ \big(\lambda x\,.\,H\ (x\ x)\big)\Big) \\
&\leftrightarrow H\ \bigg(\Big(\lambda f\,.\,\big(\lambda x\,.\,f\ (x\ x)\big)\ \big(\lambda x\,.\,f\ (x\ x)\big)\Big)\ H\bigg) \\
&= H\ (Y\ H)
\end{aligned}$$

"Y: The function that takes a function f and returns $f(f(f(f(\cdots))))$"

Prolog Execution



```
nerd(X) :- techer(X).
techer(stephen).
```

```
?- nerd(stephen).
```

Simple Searching

Starts with the query:

```
?- nerd(stephen).
```

Can we convince ourselves that `nerd(stephen)` is true given the facts we have?

```
techer(stephen).
nerd(X) :- techer(X).
```

First says `techer(stephen)` is true. Not helpful.

Second says that we can conclude `nerd(X)` is true if we can conclude `techer(X)` is true. More promising.

Simple Searching

```
techer(stephen).
nerd(X) :- techer(X).
```

```
?- nerd(stephen).
```

Unifying `nerd(stephen)` with the head of the second rule, `nerd(X)`, we conclude that `X = stephen`.

We're not done: for the rule to be true, we must find that all its conditions are true. `X = stephen`, so we want `techer(stephen)` to hold.

This is exactly the first clause in the database; we're satisfied. The query is simply true.

More Clever Searching

```
techer(stephen).
techer(todd).
nerd(X) :- techer(X).
```

```
?- nerd(X).
```

“Tell me about everybody who’s provably a nerd.”

As before, start with query. Rule only interesting thing.

Unifying `nerd(X)` with `nerd(X)` is vacuously true, so we need to establish `techer(X)`.

Unifying `techer(X)` with `techer(stephen)` succeeds, setting `X = stephen`, but we’re not done yet.

Unifying `techer(X)` with `techer(todd)` also succeeds, setting `X = todd`, but we’re still not done.

Unifying `techer(X)` with `nerd(X)` fails, returning no.

The Prolog Environment

Database consists of Horn clauses. ("If a is true and b is true and ... and y is true then z is true".)

Each clause consists of terms, which may be constants, variables, or structures.

Constants: `foo my_Const + 1.43`

Variables: `X Y Everybody My_var`

Structures:
```
rainy(rochester)
teaches(edwards, cs4115)
```

Structures and Functors

A structure consists of a functor followed by an open parenthesis, a list of comma-separated terms, and a close parenthesis:

“Functor”

paren must follow immediately

```
bin_tree( foo, bin_tree(bar, glarch) )
```

What’s a structure? Whatever you like.

A predicate `nerd(stephen)`
A relationship `teaches(edwards, cs4115)`
A data structure `bin(+, bin(-, 1, 3), 4)`

Unification

Part of the search procedure that matches patterns.

The search attempts to match a goal with a rule in the database by unifying them.

Recursive rules:

- A constant only unifies with itself
- Two structures unify if they have the same functor, the same number of arguments, and the corresponding arguments unify
- A variable unifies with anything but forces an equivalence

Unification Examples

The = operator checks whether two structures unify:

```
| ?- a = a.
yes                          % Constant unifies with itself
| ?- a = b.
no                           % Mismatched constants
| ?- 5.3 = a.
no                           % Mismatched constants
| ?- 5.3 = X.
X = 5.3 ? ;                  % Variables unify
yes
| ?- foo(a,X) = foo(X,b).
no                           % X=a required, but inconsistent
| ?- foo(a,X) = foo(X,a).
X = a                        % X=a is consistent
yes
| ?- foo(X,b) = foo(a,Y).
X = a
Y = b                        % X=a, then b=Y
yes
| ?- foo(X,a,X) = foo(b,a,c).
no                           % X=b required, but inconsistent
```

The Searching Algorithm

```
search(goal $g$, variables $e$)
    for each clause $h$ :- $t_1, \ldots, t_n$ in the database
        $e$ = unify($g$, $h$, $e$)
        if successful,
            for each term $t_1, \ldots, t_n$,
                $e$ = search($t_k$, $e$)
        if all successful, return $e$
    return no
```

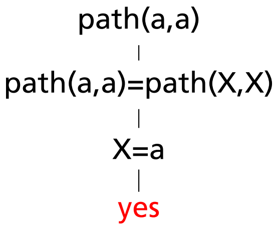
in the order they appear

in the order they appear

Note: This pseudo-code ignores one very important part of the searching process!

Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).

path(X, X).

path(X, Y) :-
    edge(X, Z), path(Z, Y).
```



Consider the query

```
| ?- path(a, a).
```

Good programming practice: Put the easily-satisfied clauses first.

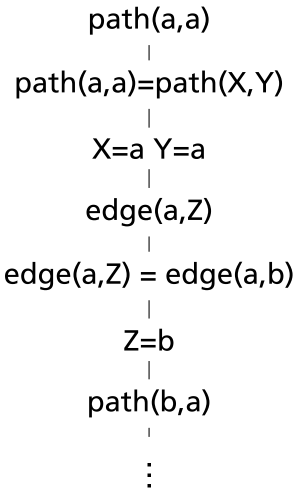
Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).

path(X, Y) :-
    edge(X, Z), path(Z, Y).

path(X, X).
```

Consider the query

```
| ?- path(a, a).
```

Will eventually produce the right answer, but will spend much more time doing so.

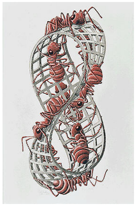


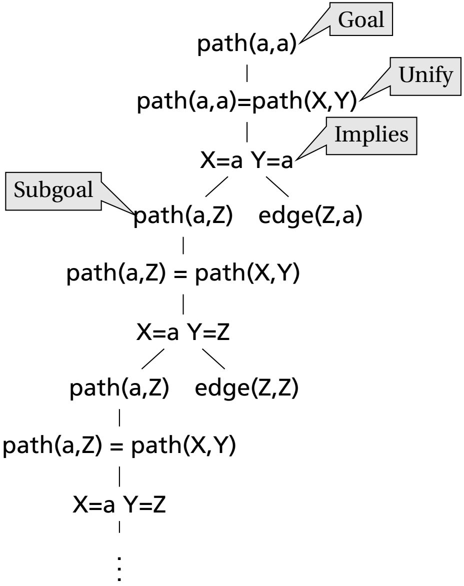
Order Can Cause Infinite Recursion

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).

path(X, Y) :-
    path(X, Z), edge(Z, Y).

path(X, X).
```

Consider the query

```
| ?- path(a, a).
```





Prolog as an Imperative Language

A declarative statement such as

P if Q and R and S

can also be interpreted procedurally as

To solve P, solve Q, then R, then S.

```
go :- print(hello_),
      print(world).
```

```
| ?- go.
hello_world
yes
```

This is the problem with the last path example.

```
path(X, Y) :-
   path(X, Z), edge(Z, Y).
```

“To solve P, solve P. . . ”

Cuts

Ways to shape the behavior of the search:

- Modify clause and term order.
  Can affect efficiency, termination.
- "Cuts"
  Explicitly forbidding further backtracking.

When the search reaches a cut (!), it does no more backtracking.

```
techer(stephen) :- !.
techer(todd).
nerd(X) :- techer(X).
```

```
| ?- nerd(X).

X = stephen

yes
```