Review for the Final

Stephen A. Edwards

Columbia University

Summer 2014

Table of Contents I

Table of Contents II

Table of Contents III

Table of Contents IV

Table of Contents V

The Lambda Calculus

The Final

70 minutes

4–5 problems

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 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

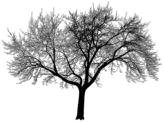
Lexical Analysis Gives Tokens

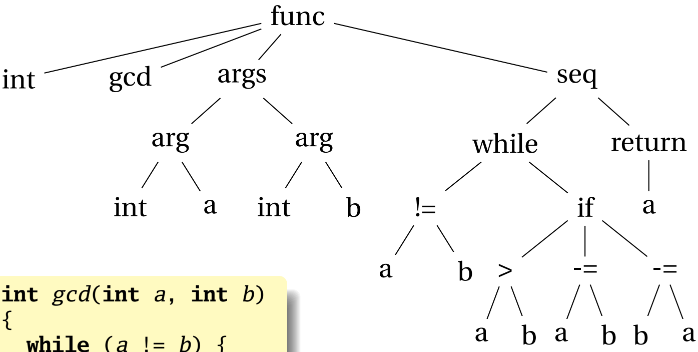
```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

`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.

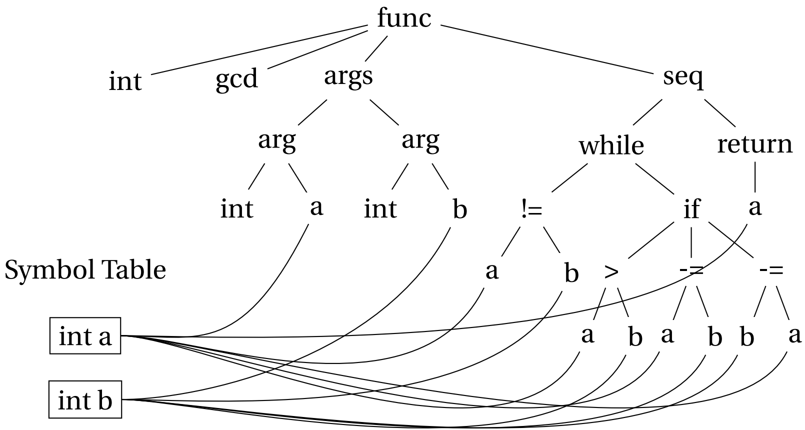
Parsing Gives an Abstract Syntax Tree






```
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



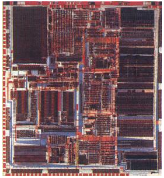
Translation into 3-Address Code

```
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: $\{ 0, 1 \}$, $\{ A, B, C, \ldots, Z \}$, 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 $L = \{ \epsilon, \text{wo} \}$, $M = \{ \text{man}, \text{men} \}$

Concatenation: Strings from one followed by the other

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

Union: All strings from each language

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

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, $\ldots\}$

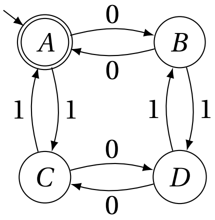
Regular Expressions over an Alphabet Σ

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"



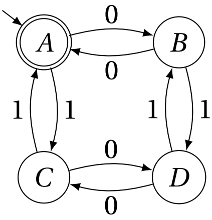
1. Set of states
   $S : \{ A \quad B \quad C \quad D \}$
2. Set of input symbols $\Sigma : \{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \to 2^S$

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

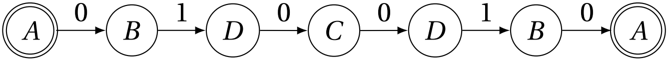
4. Start state $s_0 : A$
5. Set of accepting states $F : \{ A \}$

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.

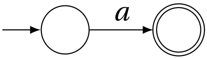


Show that the string "010010" is accepted.



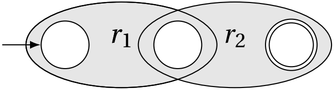
Translating REs into NFAs

a



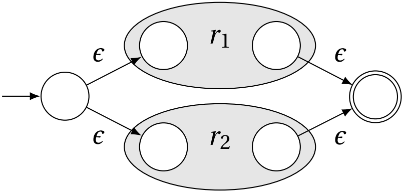
Symbol

$r_1 r_2$



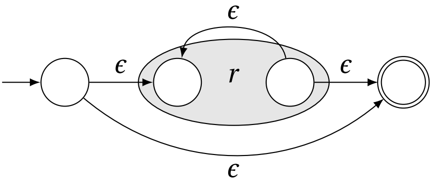
Sequence

$r_1 \mid r_2$



Choice

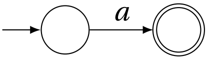
$(r)^*$



Kleene Closure

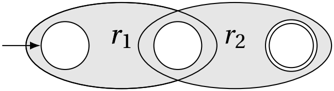
Translating REs into NFAs

a



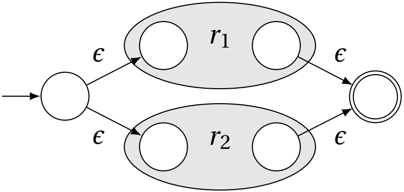
Symbol

$r_1 r_2$



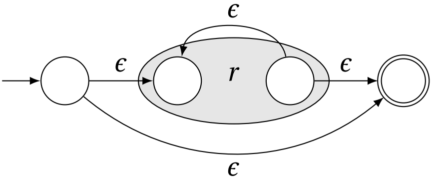
Sequence

$r_1 \mid r_2$



Choice

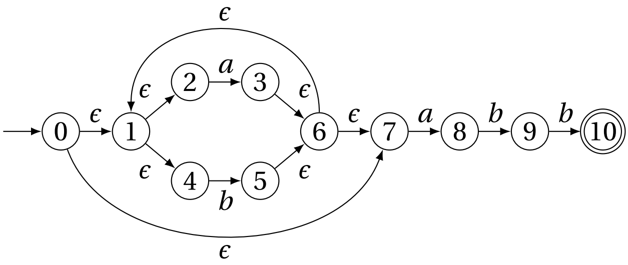
$(r)^*$



Kleene Closure

Translating REs into NFAs

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



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

$$\rightarrow 0 \xrightarrow{\epsilon} 1 \xrightarrow{\epsilon} 2 \xrightarrow{a} 3 \xrightarrow{\epsilon} 6 \xrightarrow{\epsilon} 7 \xrightarrow{a} 8 \xrightarrow{b} 9 \xrightarrow{b} (10)$$

Simulating NFAs

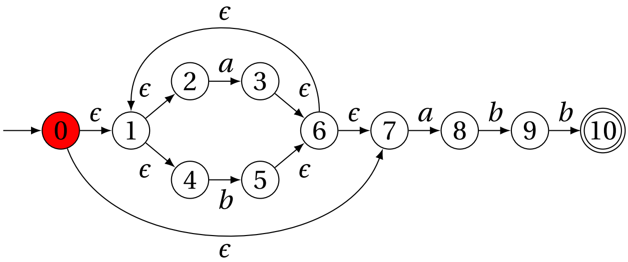
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

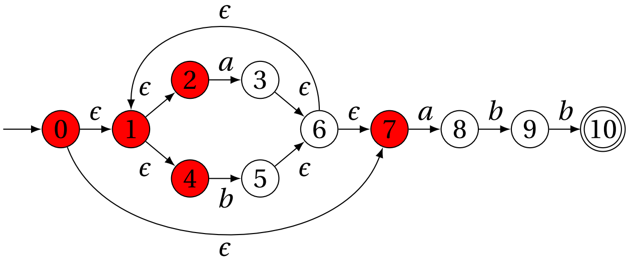
Solution: follow them all and sort it out later.

"Two-stack" NFA simulation algorithm:

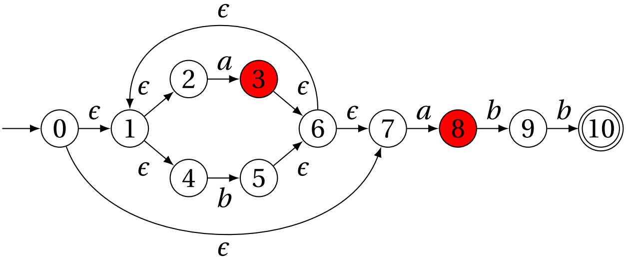
1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

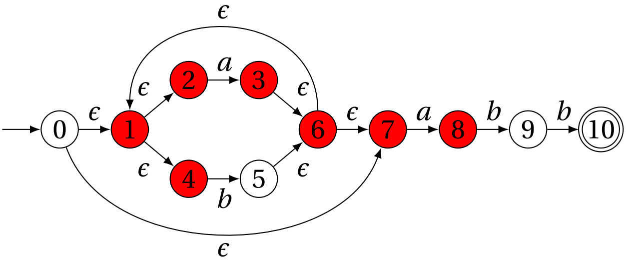
Simulating an NFA: $\cdot aabb$, Start



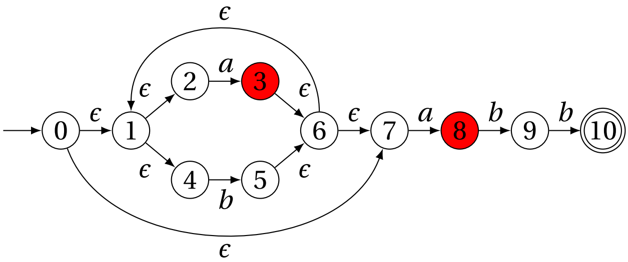


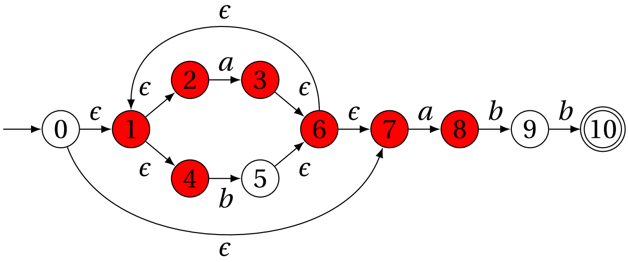
Simulating an NFA: $a \cdot abb$



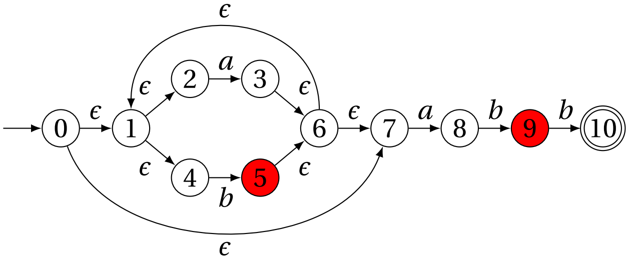


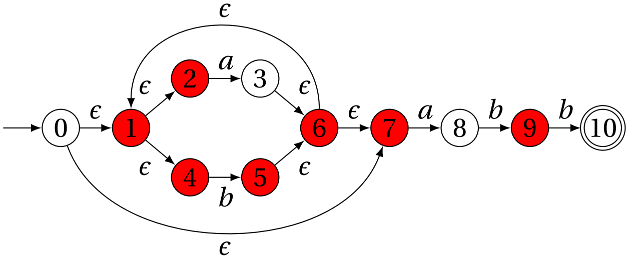
Simulating an NFA: $aa \cdot bb$



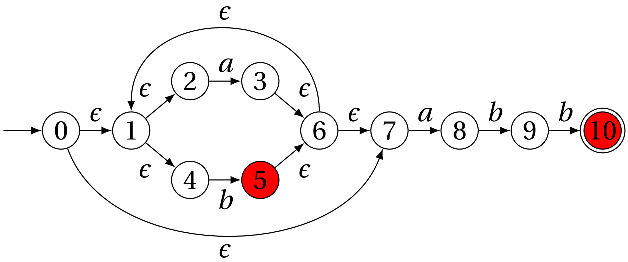


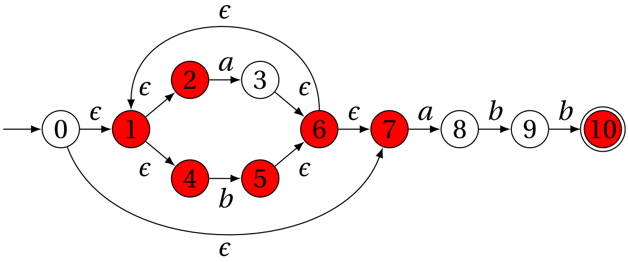
Simulating an NFA: $aab \cdot b$





Simulating an NFA: $aabb\cdot$, Done





Deterministic Finite Automata

Restricted form of NFAs:

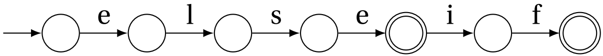
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

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.

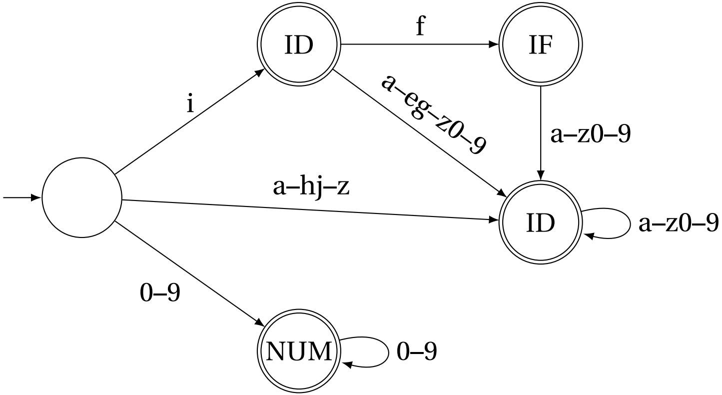
Deterministic Finite Automata

```
{
   type token = ELSE | ELSEIF
}

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



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) }
```



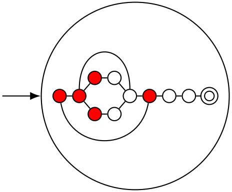
Building a DFA from an NFA

Subset construction algorithm

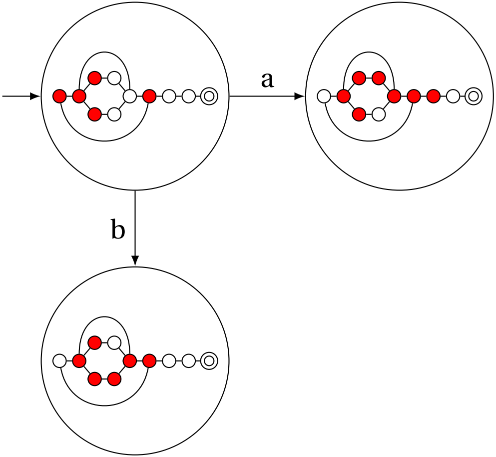
Simulate the NFA for all possible inputs and track the states that appear.

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

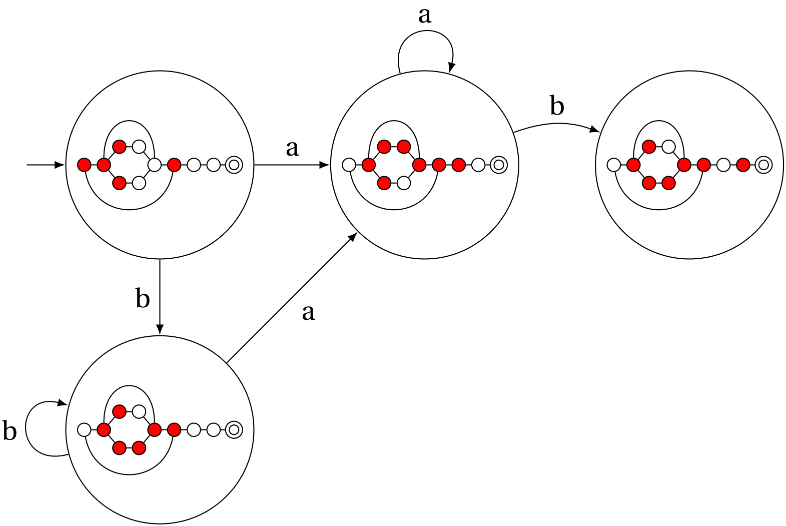
Subset construction for $(a \mid b)^* abb$



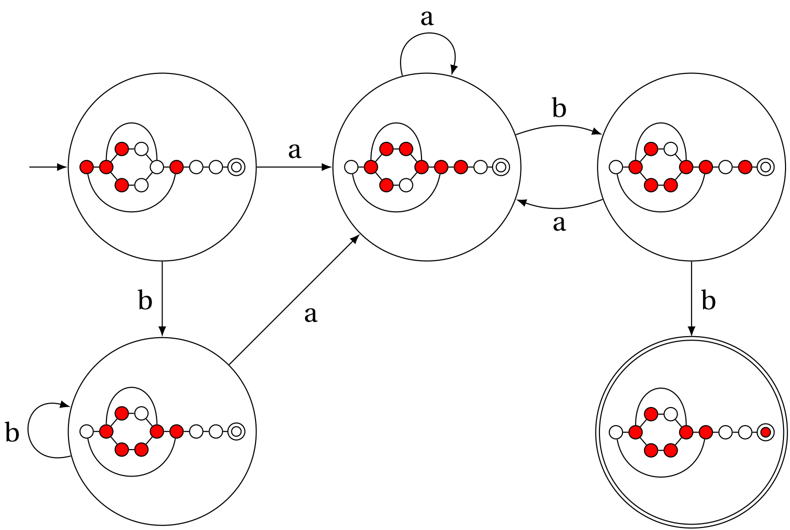
Subset construction for $(a \mid b)^* abb$



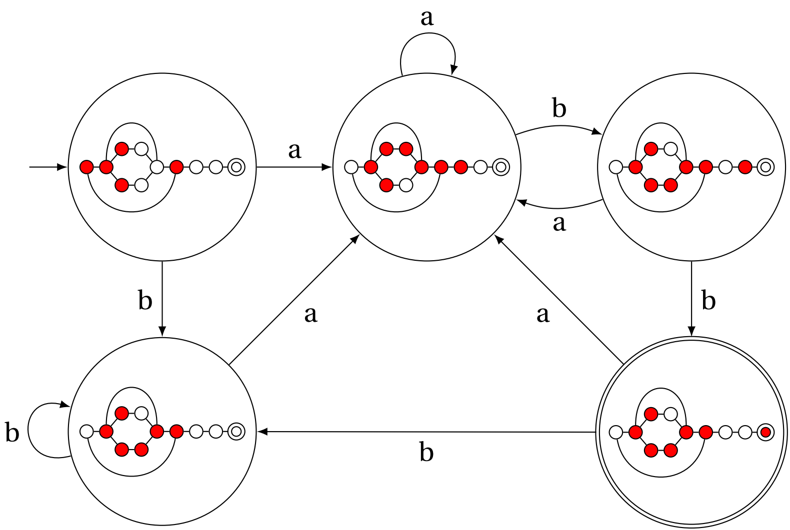
Subset construction for $(a \mid b)^* abb$



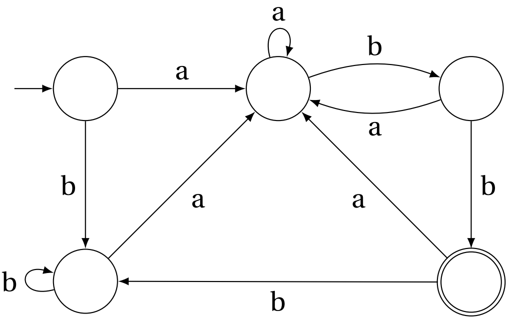
Subset construction for $(a \mid b)^* abb$



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



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



Is this minimal?

Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

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

with the grammar

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

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

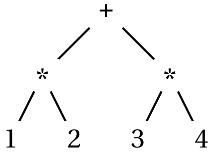
Operator Precedence

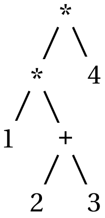
Defines how “sticky” an operator is.

`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
```

```
           -                     -
          / \                   / \
         -   4                 1   -
        / \                       / \
       -   3                     2   -
      / \                           / \
     1   2                         3   4
```

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

left associative $\qquad$ 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**

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

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

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

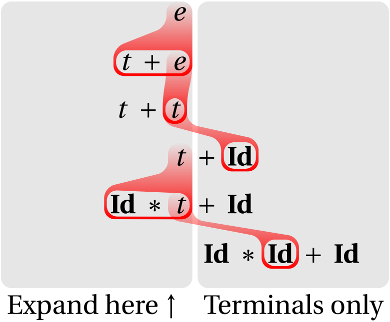
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
\boxed{t + e} \\
t + t \\
t + \boxed{\mathbf{Id}} \\
\boxed{\mathbf{Id} * t} + \mathbf{Id} \\
\mathbf{Id} * \boxed{\mathbf{Id}} + \mathbf{Id}
\end{array}
$$



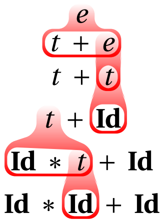
Reverse Rightmost Derivation

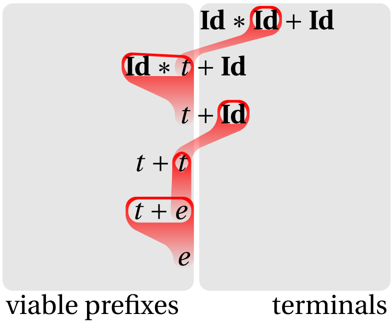
$1: e \rightarrow t + e$

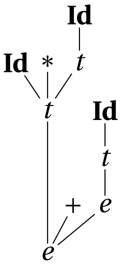
$2: e \rightarrow t$

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

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







Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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

stack	input	
	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id}$	$* \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} *$	$\mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} * \mathbf{Id}$	$+ \mathbf{Id}$	reduce 4
$\mathbf{Id} * t$	$+ \mathbf{Id}$	reduce 3
t	$+ \mathbf{Id}$	shift
$t +$	$\mathbf{Id}$	shift
$t + \mathbf{Id}$		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.*

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

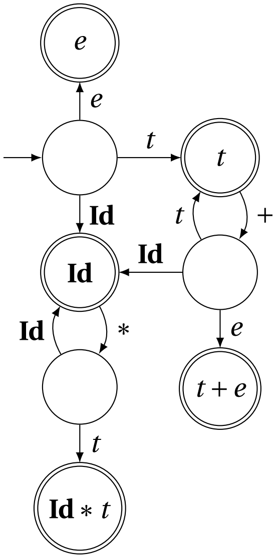
$\mathbf{Id} * \mathbf{Id} * \cdots * \boxed{\mathbf{Id}} \cdots$

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

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

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

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



Building the Initial State of the LR(0) Automaton

$$e' \rightarrow \cdot e$$

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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 "$e' \rightarrow \cdot e$"

Building the Initial State of the LR(0) Automaton

$$e' \rightarrow \cdot e$$
$$e \rightarrow \cdot t + e$$
$$e \rightarrow \cdot t$$

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

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

$$
\begin{array}{|l|}
\hline
e' \to \cdot e \\
e \to \cdot t + e \\
e \to \cdot t \\
t \to \cdot \mathbf{Id} * t \\
t \to \cdot \mathbf{Id} \\
\hline
\end{array}
$$

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

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 "$e' \to \cdot e$"

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

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

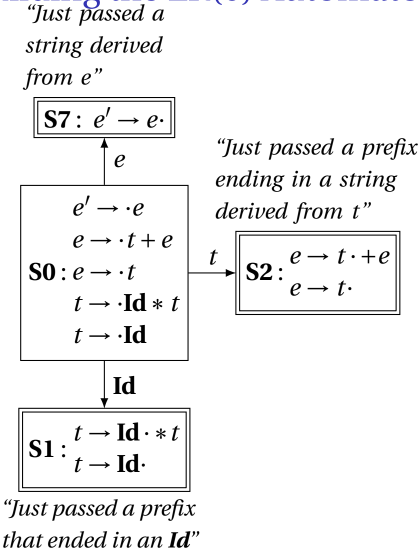
This reasoning is a *closure* operation like ϵ-closure in subset construction.

Building the LR(0) Automaton

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

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

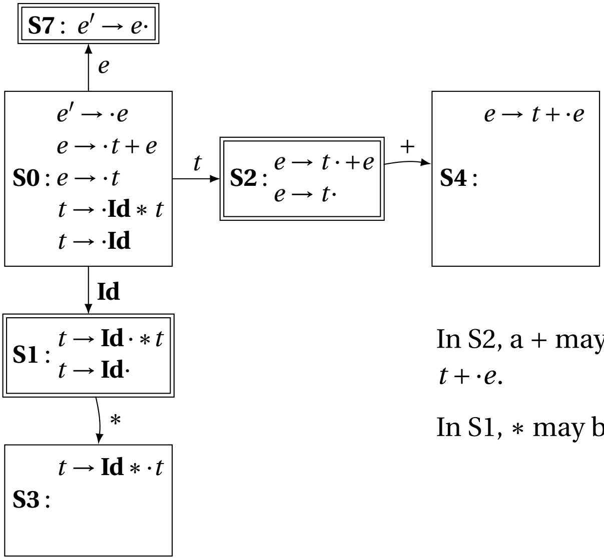
Building the LR(0) Automaton



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).

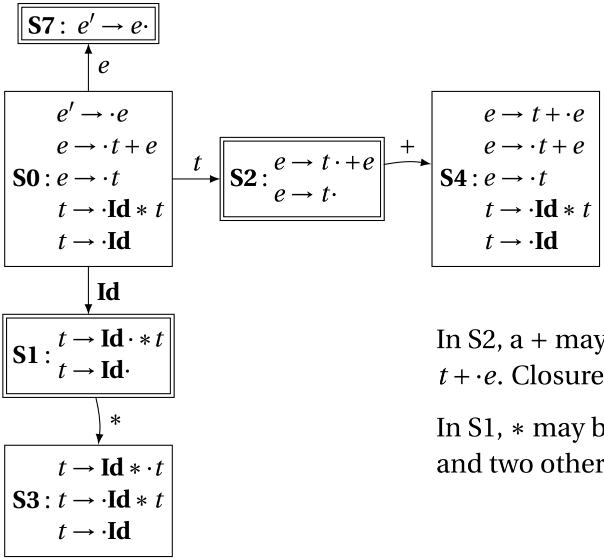
Building the LR(0) Automaton



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

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

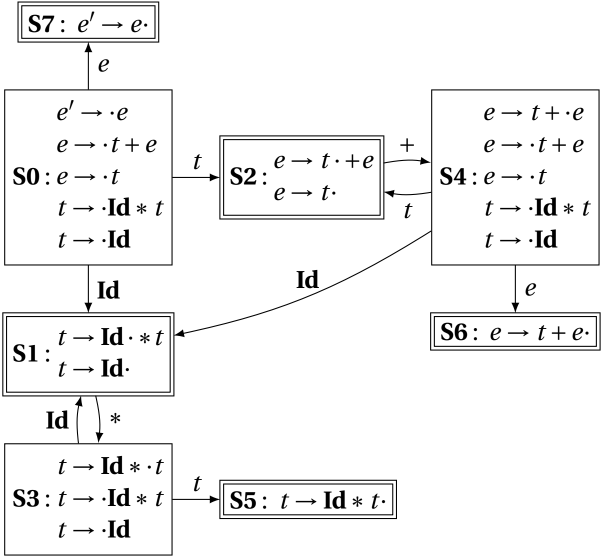
Building the LR(0) Automaton



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

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

Building the LR(0) Automaton



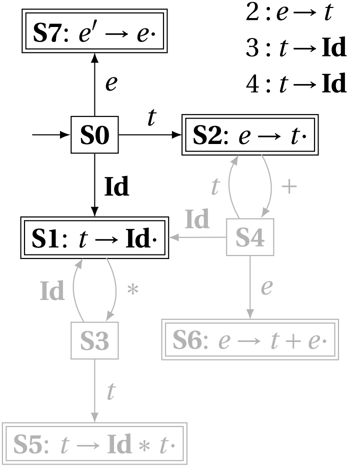
Converting the LR(0) Automaton to an SLR Parsing Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

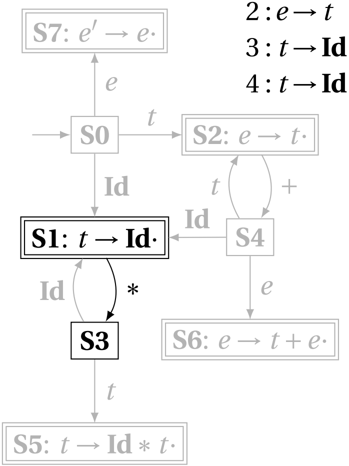
$4 : t \rightarrow \mathbf{Id}$



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



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

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.

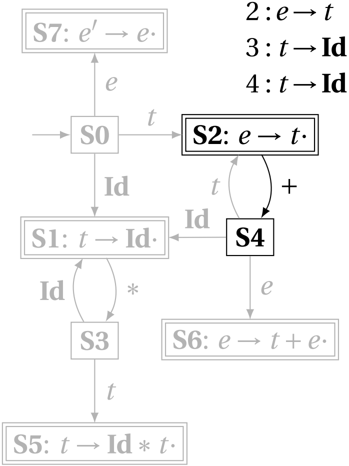
Converting the LR(0) Automaton to an SLR Parsing Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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

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.

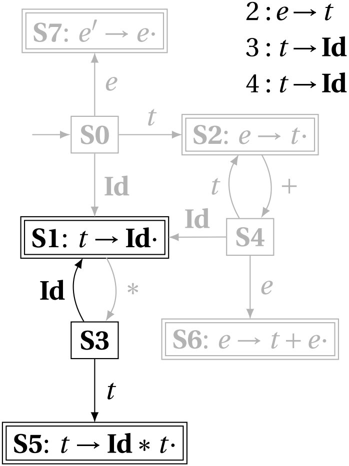
Converting the LR(0) Automaton to an SLR Parsing Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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.

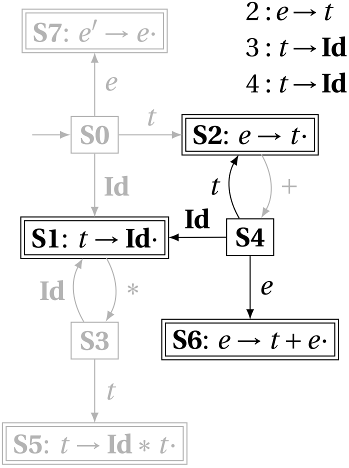
Converting the LR(0) Automaton to an SLR Parsing Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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

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

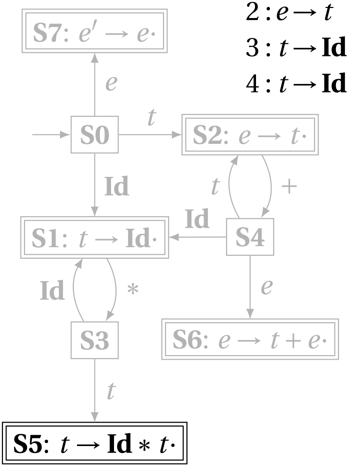
Converting the LR(0) Automaton to an SLR Parsing Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

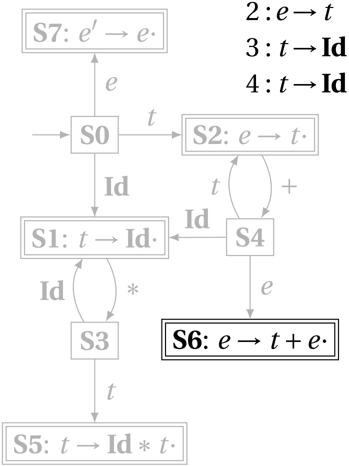
$4 : t \rightarrow \mathbf{Id}$



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

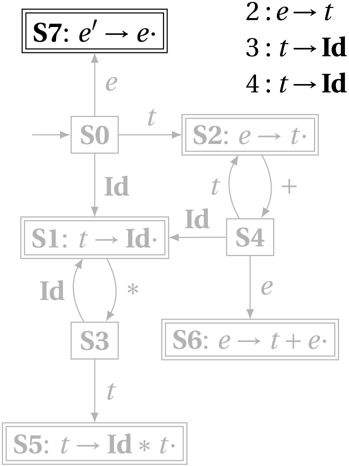


$1 : e \rightarrow t + e$
$2 : e \rightarrow t$
$3 : t \rightarrow \mathbf{Id} * t$
$4 : t \rightarrow \mathbf{Id}$

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



$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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

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

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	**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	**Id** $*$ **Id** + **Id** \$	Shift, goto 1
0 **Id** 1	$*$ **Id** + **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
$0\ \overset{\textbf{Id}}{1}$	$* \textbf{Id} + \textbf{Id}\,\$$	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

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

Shift/Reduce Parsing with an SLR Table

$1: e \to t + e$

$2: e \to t$

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

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

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	**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

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

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	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}\,\$$	Shift, goto 1
0 $\mathbf{Id}_1$	$* \mathbf{Id} + \mathbf{Id}\,\$$	Shift, goto 3
0 $\mathbf{Id}_1$ $*_3$	$\mathbf{Id} + \mathbf{Id}\,\$$	Shift, goto 1
0 $\mathbf{Id}_1$ $*_3$ $\mathbf{Id}_1$	$+ \mathbf{Id}\,\$$	Reduce 4
0 $\mathbf{Id}_1$ $*_3$ t_5	$+ \mathbf{Id}\,\$$	Reduce 3
0 t_2	$+ \mathbf{Id}\,\$$	Shift, goto 4

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

Shift/Reduce Parsing with an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

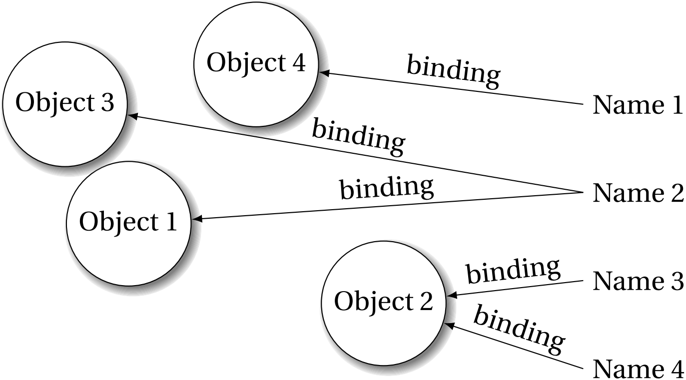
$3 : t \rightarrow \mathbf{Id} * t$

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

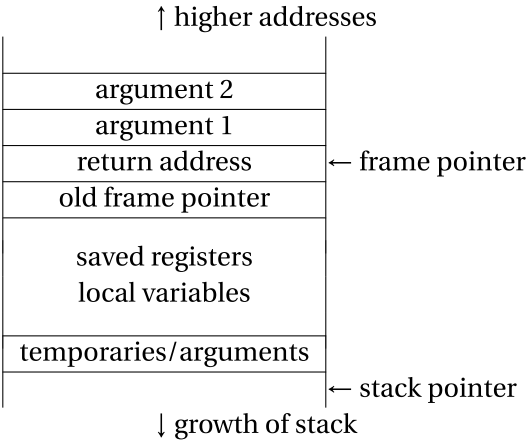
State	Action				Goto	
	Id	+	∗	\$	*e*	*t*
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}\ \$$	Shift, goto 1
$0\ \ \mathbf{Id}_{1}$	$* \mathbf{Id} + \mathbf{Id}\ \$$	Shift, goto 3
$0\ \ \mathbf{Id}_{1}\ \ *_{3}$	$\mathbf{Id} + \mathbf{Id}\ \$$	Shift, goto 1
$0\ \ \mathbf{Id}_{1}\ \ *_{3}\ \ \mathbf{Id}_{1}$	$+ \mathbf{Id}\ \$$	Reduce 4
$0\ \ \mathbf{Id}_{1}\ \ *_{3}\ \ t_{5}$	$+ \mathbf{Id}\ \$$	Reduce 3
$0\ \ t_{2}$	$+ \mathbf{Id}\ \$$	Shift, goto 4
$0\ \ t_{2}\ \ +_{4}$	$\mathbf{Id}\ \$$	Shift, goto 1
$0\ \ t_{2}\ \ +_{4}\ \ \mathbf{Id}_{1}$	$\$$	Reduce 4
$0\ \ t_{2}\ \ +_{4}\ \ t_{2}$	$\$$	Reduce 2
$0\ \ t_{2}\ \ +_{4}\ \ e_{6}$	$\$$	Reduce 1
$0\ \ e_{7}$	$\$$	Accept

Names, Objects, and Bindings



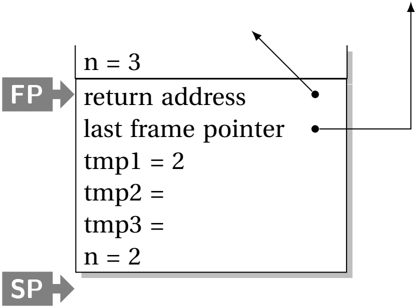
Typical Stack Layout



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;
}
```

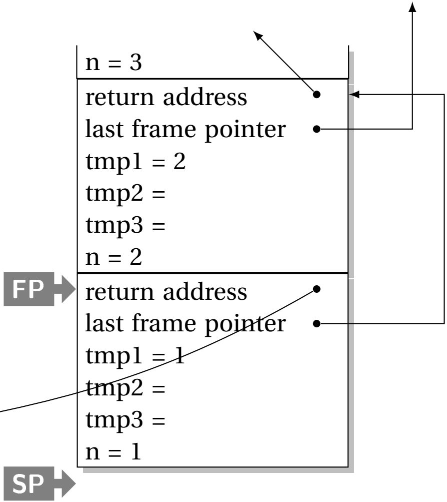
Executing fib(3)




```
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;
}
```

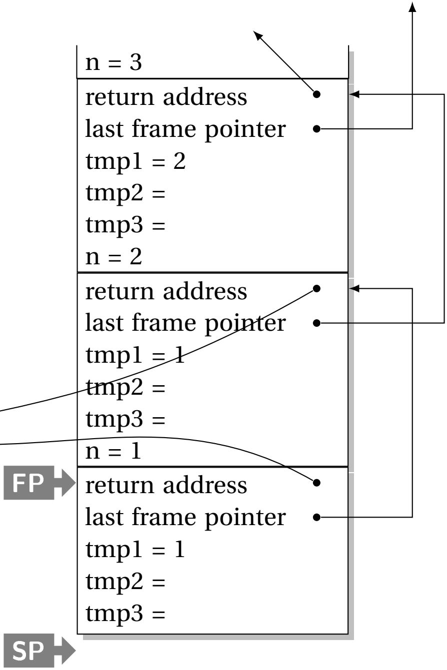
Executing fib(3)

```
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;
}
```



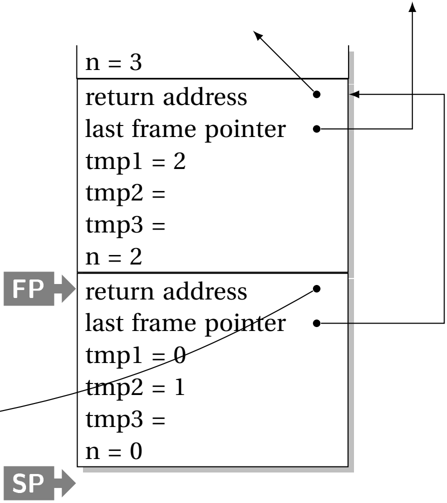
Executing fib(3)

```
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;
}
```



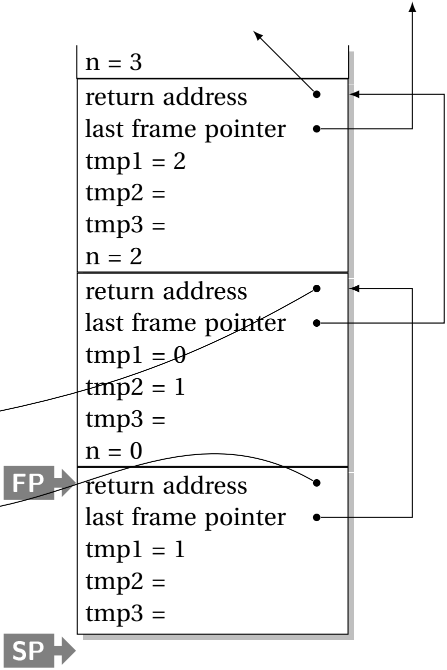
Executing fib(3)

```
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;
}
```



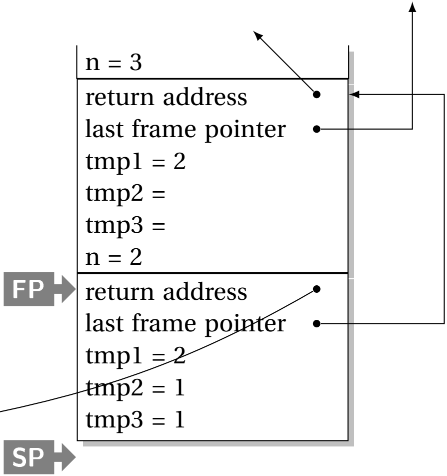
Executing fib(3)

```
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;
}
```

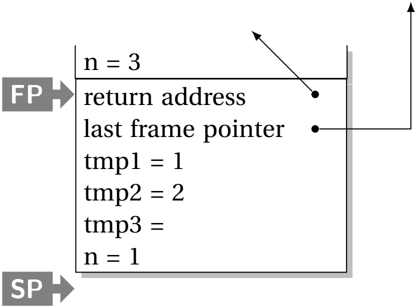


Executing fib(3)

```
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;
}
```



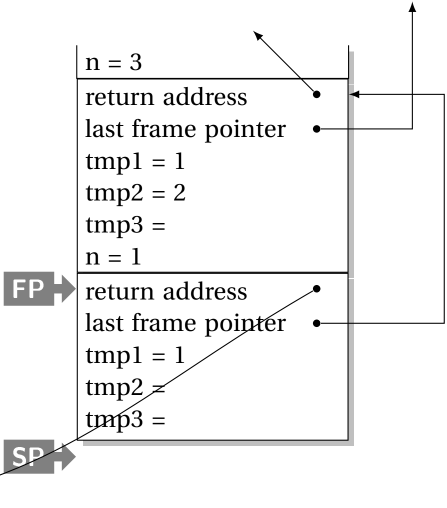
Executing fib(3)



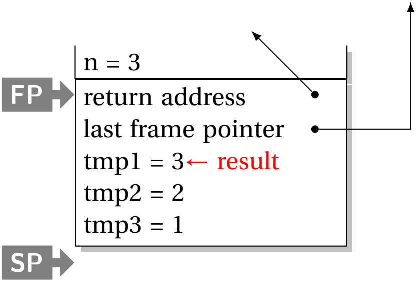

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

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;
}
```

Implementing Nested Functions with Static Links

a:

(static link) •

x = 5

s = 42

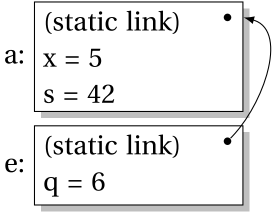
```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” evaluate to?

Implementing Nested Functions with Static 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 *)
```

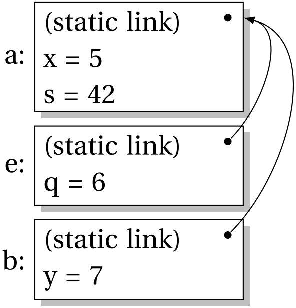
What does “a 5 42” evaluate to?



Implementing Nested Functions with Static 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 *)
```

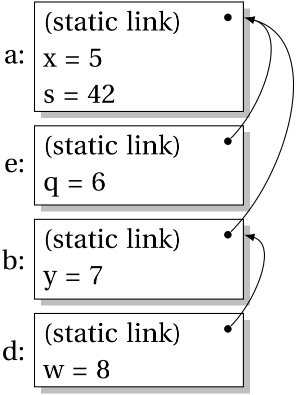
What does “a 5 42” evaluate to?



Implementing Nested Functions with Static 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 *)
```

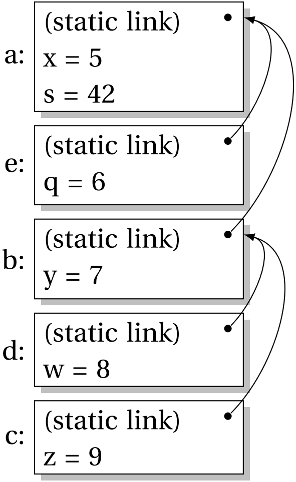
What does “a 5 42” evaluate to?



Implementing Nested Functions with Static 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” evaluate to?



Static vs. Dynamic Scope

```
program example;
var a : integer; (* Outer a *)

  procedure seta;
  begin
    a := 1  (* Which a does this change? *)
  end

  procedure locala;
  var a : integer; (* Inner a *)
  begin
    seta
  end

begin
  a := 2;
  if (readln() = 'b')
    locala
  else
    seta;
  writeln(a)
end
```

C's Types: Base Types/Pointers

Base types match typical processor

Typical sizes:	8	16	32	64
	char	short	int	long
			float	double

Pointers (addresses)

```
int *i;   /* i is a pointer to an int */
char **j; /* j is a pointer to a pointer to a char */
```

C's Types: Arrays, Functions

Arrays

```
char c[10];          /* c[0] ... c[9] are chars */
double a[10][3][2]; /* array of 10 arrays of 3 arrays of 2 doubles */
```

Functions

```
/* function of two arguments returning a char */
char foo(int, double);
```

C’s Types: Structs and Unions

Structures: each field has own storage

```
struct box {
  int x, y, h, w;
  char *name;
};
```

Unions: fields share same memory

```
union token {
  int i;
  double d;
  char *s;
};
```

Composite Types: Records

A record is an object with a collection of fields, each with a potentially different type. In C,

```
struct rectangle {
  int n, s, e, w;
  char *label;
  color col;
  struct rectangle *next;
};

struct rectangle r;
r.n = 10;
r.label = "Rectangle";
```

Applications of Records

Records 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);
```

Composite Types: Variant Records

A record object holds all of its fields. A variant record holds only one of its fields at once. In C,

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

union token t;
t.i = 10;
t.f = 3.14159;       /* overwrites t.i */
char *s = t.string;  /* returns 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`.

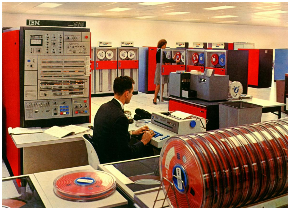
If `poly.type == SQUARE`, use `poly.d.size`.

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

Layout of Records and Unions

Modern processors have byte-addressable memory.

0
1
2
3



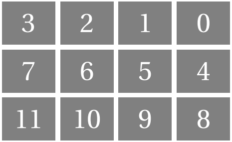
The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

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

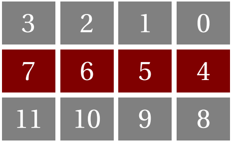
16-bit integer:			1	0
32-bit integer:	3	2	1	0

Layout of Records and Unions

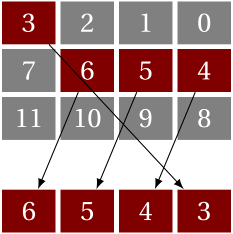
Modern memory systems read data in 32-, 64-, or 128-bit chunks:



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



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



SPARC prohibits unaligned accesses.

MIPS has special unaligned load/store instructions.

x86, 68k run more slowly with unaligned accesses.

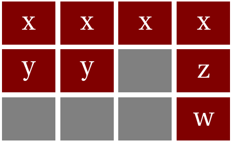
Padding

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

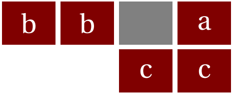
Rules:

- 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 */
};
```



Arrays

Most languages provide array types:

```
char i[10];                         /* C */

character(10) i                     ! FORTRAN

i : array (0..9) of character;      -- Ada

var i : array [0 .. 9] of char;     { Pascal }
```

Array Address Calculation

In C,

`struct foo a[10];`

`a[i]` is at $a + i *$ `sizeof(struct foo)`

`struct foo a[10][20];`

`a[i][j]` is at $a + (j + 20 * i) *$ `sizeof(struct foo)`

$\Rightarrow$ Array bounds must be known to access 2D+ arrays

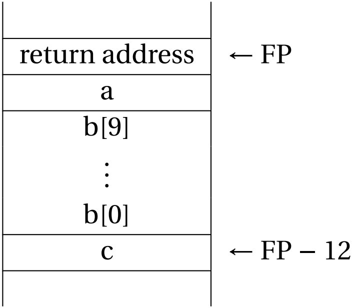
Allocating Arrays in C++

```
int a[10];              /* static */

void foo(int n)
{
  int b[15];            /* stacked */
  int c[n];             /* stacked: tricky */
  int d[];              /* on heap */
  vector<int> e;        /* on heap */

  d = new int[n*2]; /* fixes size */
  e.append(1);          /* may resize */
  e.append(2);          /* may resize */
}
```

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

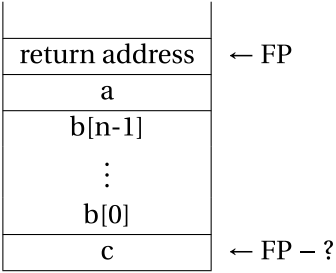
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

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

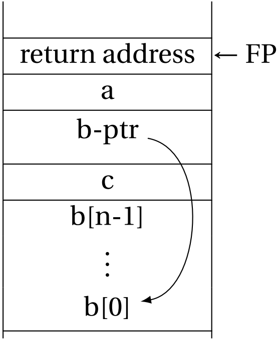


Doesn't work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.

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.

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?

Name vs. Structural Equivalence

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

typedef struct f f_t;

f_t baz;

baz = foo;
```

Legal because `f_t` is an alias for `struct f`.

Ordering Within Expressions

What code does a compiler generate for

```
a = b + c + d;
```

Most likely something like

```
tmp = b + c;
a = tmp + d;
```

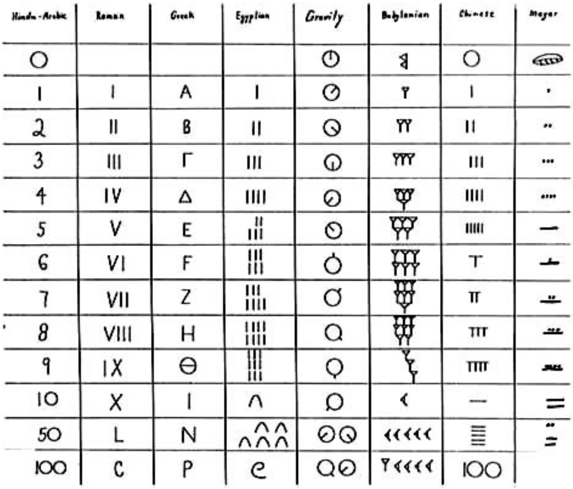
(Assumes left-to-right evaluation of expressions.)

Order of Evaluation

Why would you care?

Expression evaluation can have side-effects.

Floating-point numbers don't behave like numbers.



Side-effects

```
int x = 0;

int foo() {
  x += 5;
  return x;
}

int bar() {
  int a = foo() + x + foo();
  return a;
}
```

What does *bar()* return?

Side-effects

```c
int x = 0;

int foo() {
  x += 5;
  return x;
}

int bar() {
  int a = foo() + x + foo();
  return a;
}
```

What does *bar()* return?

GCC returned 25.

Sun's C compiler returned 20.

C says expression evaluation order is implementation-dependent.

Side-effects

Java prescribes left-to-right evaluation.

```
class Foo {

  static int x;

  static int foo() {
     x += 5;
     return x;
  }

  public static void main(String args[]) {
    int a = foo() + x + foo();
    System.out.println(a);
  }

}
```

Always prints 20.

Short-Circuit Evaluation

When you write

```
if (disaster_could_happen)
   avoid_it();
else
   cause_a_disaster();
```

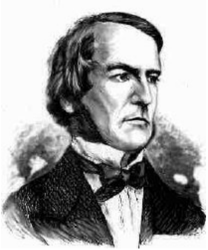
`cause_a_disaster()` is not called when `disaster_could_happen` is true.

The *if* statement evaluates its bodies lazily: only when necessary.

The section operator `? :` does this, too.

```
cost = disaster_possible ? avoid_it() : cause_it();
```

Logical Operators



In Java and C, Boolean logical operators "short-circuit" to provide this facility:

```
if (disaster_possible || case_it()) { ... }
```

`cause_it()` only called if `disaster_possible` is false.

The && operator does the same thing.

Useful when a later test could cause an error:

```
int a[10];

if (i => 0 && i < 10 && a[i] == 0) { ... }
```

Unstructured Control-Flow

Assembly languages usually provide three types of instructions:

Pass control to next instruction:

```
add, sub, mov, cmp
```

Pass control to another instruction:

```
jmp rts
```

Conditionally pass control next or elsewhere:

```
beq bne blt
```

Unstructured Control-Flow

```
  BEQ A
B:

  JMP C
A:

  BEQ D
C:

  BEQ B
D:

  BNE B
  RTS
```

Structured Control-Flow

The "object-oriented languages" of the 1960s and 70s.

Structured programming replaces the evil *goto* with structured (nested) constructs such as

for

while

break

return

continue

do .. while

if .. then .. else

Gotos vs. Structured Programming

A typical use of a goto is building a loop. In BASIC:

```
10 PRINT I
20 I = I + 1
30 IF I < 10 GOTO 10
```

A cleaner version in C using structured control flow:

```
do {
  printf("%d\n", i);
  i = i + 1;
} while ( i < 10 )
```

An even better version

```
for (i = 0 ; i < 10 ; i++)
  printf("%d\n", i);
```

Gotos vs. Structured Programming

Break and continue leave loops prematurely:

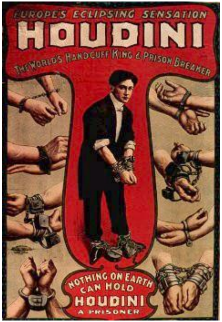
```
for ( i = 0 ; i < 10 ; i++ ) {
  if ( i == 5 ) continue;
  if ( i == 8 ) break;
  printf("%d\n", i);
}
```

```
  i = 0;
Again:
  if (!(i < 10)) goto Break;
  if ( i == 5 ) goto Continue;
  if ( i == 8 ) goto Break;
  printf("%d\n", i);
Continue: i++; goto Again;
Break:
```

Escaping from Loops

Java allows you to escape from labeled loops:

```
a: for (int i = 0 ; i < 10 ; i++)
     for ( int j = 0 ; j < 10 ; j++) {
        System.out.println(i + "," + j);
        if (i == 2 && j == 8) continue a;
        if (i == 8 && j == 4) break a;
     }
```



Gotos vs. Structured Programming

Pascal has no “return” statement for escaping from functions/procedures early, so goto was necessary:

```
procedure consume_line(var line : string);
begin
  if line[i] = '%' then goto 100;
  (* .... *)
100:
end
```

In C and many others, return does this for you:

```
void consume_line(char *line) {
  if (line[0] == '%') return;
}
```

Multi-way Branching

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

```
switch (s) {
  case 1: goto One;
  case 2: goto Two;
  case 3: goto Three;
  case 4: goto Four;
}
goto Break;

One:   one(); goto Break;
Two:   two(); goto Break;
Three: three(); goto Break;
Four:  four(); goto Break;
Break:
```

Switch sends control to one of the case labels. Break terminates the statement. Really just a multi-way *goto*:

Implementing multi-way branches

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

Obvious way:

```
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
```

Reasonable, but we can sometimes do better.

Implementing multi-way branches

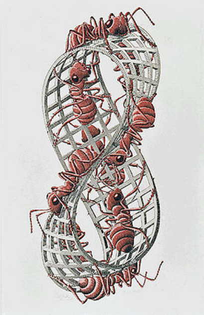
If the cases are *dense*, a branch table is more efficient:

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

A branch table written using a GCC extension:

```
  /* Array of addresses of labels */
  static void *l[] = { &&L1, &&L2, &&L3, &&L4 };

  if (s >= 1 && s <= 4)
     goto *l[s-1];
  goto Break;
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

Recursion and Iteration



To compute $\sum_{i=0}^{10} f(i)$ in C,

the most obvious technique is iteration:

```
double total = 0;
for ( i = 0 ; i <= 10 ; i++ )
  total += f(i);
```

Tail-Recursion and Iteration

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

Notice: no computation follows any recursive calls.

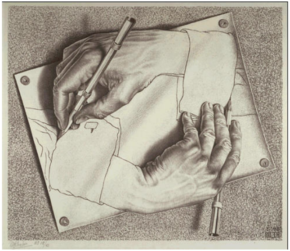
Stack is not necessary: all variables “dead” after the call.

Local variable space can be reused. Trivial since the collection of variables is the same.

Works in O’Caml, too

```
let rec gcd a b =
  if a = b then a
  else if a > b then gcd (a - b) b
  else gcd a (b - a)
```

Tail-Recursion and Iteration



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

Can be rewritten into:

```
int gcd(int a, int b) {
start:
  if ( a==b ) return a;
  else if ( a > b ) a = a-b; goto start;
  else b = b-a; goto start;
}
```

Good compilers, especially those for functional languages, identify and optimize tail recursive functions.

Less common for imperative languages, but gcc -O was able to handle this example.

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”

Argument Evaluation Order

C does not define argument evaluation order:

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

int q(int a, int b, int c) {}

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

Might print 1 2 3, 3 2 1, or something else.

This is an example of *nondeterminism*.

Nondeterminism

Nondeterminism is not the same as random:

Compiler usually chooses an order when generating code.

Optimization, exact expressions, or run-time values may affect behavior.

Bottom line: don't know what code will do, but often know set of possibilities.

```
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}
q( p(1), p(2), p(3) );
```

Will *not* print 5 6 7. It will print one of

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

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

```c
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 →

```
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 →

```
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.

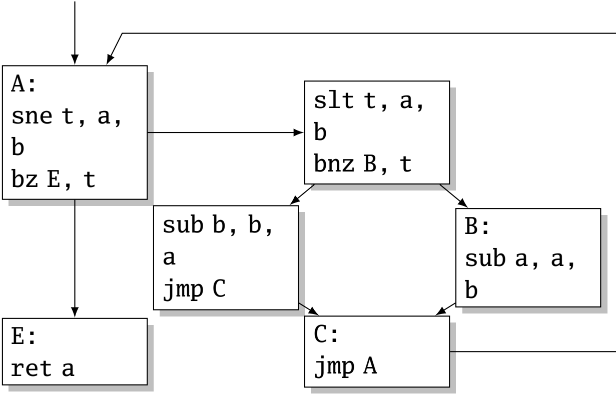
Starts with a statement following a conditional branch or is a branch target.

Usually ends with a control-transfer statement.

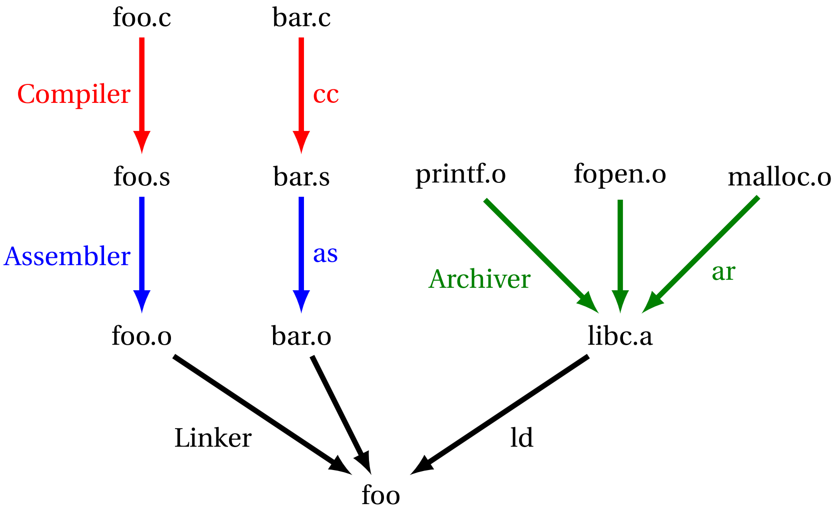
Control-Flow Graphs

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
```



Separate Compilation and Linking



Linking

Goal of the linker is to combine the disparate pieces of the program into a coherent whole.

file1.c:

```
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

file2.c:

```
#include <stdio.h>
extern char a[];

static char b[6];

void bar() {
  strcpy(b, a);
  baz(b);
}
```

libc.a:

```
int
printf(char *s, ...)
{
  /* ... */
}

char *
strcpy(char *d,
       char *s)
{
  /* ... */
}
```

Linking

Goal of the linker is to combine the disparate pieces of the program into a coherent whole.

file1.c:

```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

file2.c:

```c
#include <stdio.h>
extern char a[];

static char b[6];

void bar() {
  strcpy(b, a);
  baz(b);
}
```

libc.a:

```c
int
printf(char *s, ...)
{
  /* ... */
}

char *
strcpy(char *d,
       char *s)
{
  /* ... */
}
```

Linking

file1.o

a="Hello"

main()

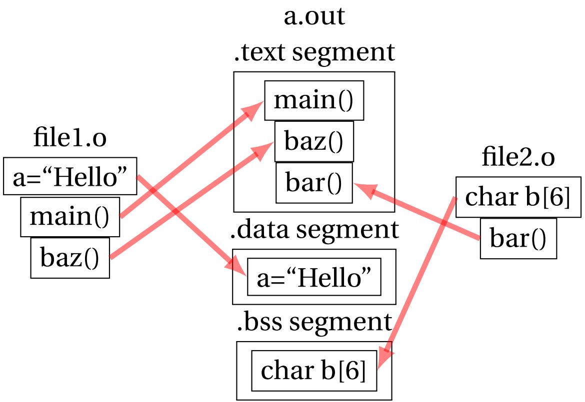
baz()

file2.o

char b[6]

bar()

Linking



.text
Code of program

.data
Initialized data

.bss
Uninitialized data
"Block Started by Symbol"

Object Files

Relocatable: Many need to be pasted together. Final in-memory address of code not known when program is compiled

Object files contain

- imported symbols (unresolved “external” symbols)
- relocation information (what needs to change)
- exported symbols (what other files may refer to)

Object Files

file1.c:

```
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

exported symbols

imported symbols

Object Files

file1.c:

```
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

```
# objdump -x file1.o
Sections:
Idx Name     Size VMA LMA Offset Algn
  0 .text    038  0   0   034    2**2
  1 .data    008  0   0   070    2**3
  2 .bss     000  0   0   078    2**0
  3 .rodata  008  0   0   078    2**3

SYMBOL TABLE:
0000 g O .data   006 a
0000 g F .text   014 main
0000     *UND*   000 bar
0014 g F .text   024 baz
0000     *UND*   000 printf

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE            VALUE
0004 R_SPARC_WDISP30   bar
001c R_SPARC_HI22      .rodata
0020 R_SPARC_LO10      .rodata
0028 R_SPARC_WDISP30   printf
```

Object Files

file1.c:

```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

```
# objdump -d file1.o
0000 <main>:
 0: 9d e3 bf 90 save  %sp, -112, %sp
 4: 40 00 00 00 call  4 <main+0x4>
    4: R_SPARC_WDISP30 bar
 8: 01 00 00 00 nop
 c: 81 c7 e0 08 ret
10: 81 e8 00 00 restore

0014 <baz>:
14: 9d e3 bf 90 save  %sp, -112, %sp
18: f0 27 a0 44 st  %i0, [ %fp + 0x44 ]
1c: 11 00 00 00 sethi  %hi(0), %o0
    1c: R_SPARC_HI22 .rodata
20: 90 12 20 00 mov  %o0, %o0
    20: R_SPARC_LO10 .rodata
24: d2 07 a0 44 ld  [ %fp + 0x44 ], %o1
28: 40 00 00 00 call  28 <baz+0x14>
    28: R_SPARC_WDISP30 printf
2c: 01 00 00 00 nop
30: 81 c7 e0 08 ret
34: 81 e8 00 00 restore
```

Before and After Linking

```
int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

- Combine object files
- Relocate each function's code
- Resolve previously unresolved symbols

Code starting address changed

```
0000 <main>:
 0: 9d e3 bf 90 save  %sp, -112, %sp
 4: 40 00 00 00 call  4 <main+0x4>
    4: R_SPARC_WDISP30 bar
 8: 01 00 00 00 nop
 c: 81 c7 e0 08 ret
10: 81 e8 00 00 restore

0014 <baz>:
14: 9d e3 bf 90 save  %sp, -112, %sp
18: f0 27 a0 44 st  %i0, [ %fp + 0x44 ]
1c: 11 00 00 00 sethi  %hi(0), %o0
    1c: R_SPARC_HI22 .rodata
20: 90 12 20 00 mov  %o0, %o0
    20: R_SPARC_LO10 .rodata
24: d2 07 a0 44 ld  [ %fp + 0x44 ], %o1
28: 40 00 00 00 call  28 <baz+0x14>
    28: R_SPARC_WDISP30 printf
2c: 01 00 00 00 nop
30: 81 c7 e0 08 ret
34: 81 e8 00 00 restore
```

Unresolved symbol

```
105f8 <main>:
105f8: 9d e3 bf 90 save %sp, -112, %sp
105fc: 40 00 00 0d call 10630 <bar>

10600: 01 00 00 00 nop
10604: 81 c7 e0 08 ret
10608: 81 e8 00 00 restore

1060c <baz>:
1060c: 9d e3 bf 90 save  %sp, -112, %sp
10610: f0 27 a0 44 st    %i0, [ %fp + 0x44 ]
10614: 11 00 00 41 sethi %hi(0x10400), %o0

10618: 90 12 23 00 or    %o0, 0x300, %o0

1061c: d2 07 a0 44 ld    [ %fp + 0x44 ], %o1
10620: 40 00 40 62 call  207a8

10624: 01 00 00 00 nop
10628: 81 c7 e0 08 ret
1062c: 81 e8 00 00 restore
```

Linking Resolves Symbols

file1.c:

```
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
  bar();
}

void baz(char *s) {
  printf("%s", s);
}
```

file2.c:

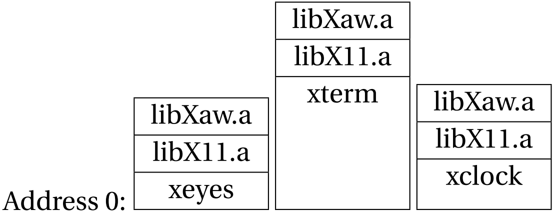
```
#include <stdio.h>
extern char a[];

static char b[6];

void bar() {
  strcpy(b, a);
  baz(b);
}
```

```
105f8 <main>:
105f8: 9d e3 bf 90 save %sp, -112, %sp
105fc: 40 00 00 0d call 10630 <bar>
10600: 01 00 00 00 nop
10604: 81 c7 e0 08 ret
10608: 81 e8 00 00 restore

1060c <baz>:
1060c: 9d e3 bf 90 save  %sp, -112, %sp
10610: f0 27 a0 44 st    %i0, [ %fp + 0x44 ]
10614: 11 00 00 41 sethi %hi(0x10400), %o0
10618: 90 12 23 00 or    %o0, 0x300, %o0   ! "%s"
1061c: d2 07 a0 44 ld    [ %fp + 0x44 ], %o1
10620: 40 00 40 62 call  207a8             ! printf
10624: 01 00 00 00 nop
10628: 81 c7 e0 08 ret
1062c: 81 e8 00 00 restore

10630 <bar>:
10630: 9d e3 bf 90 save  %sp, -112, %sp
10634: 11 00 00 82 sethi %hi(0x20800), %o0
10638: 90 12 20 a8 or    %o0, 0xa8, %o0    ! 208a8 <b>
1063c: 13 00 00 81 sethi %hi(0x20400), %o1
10640: 92 12 63 18 or    %o1, 0x318, %o1   ! 20718 <a>
10644: 40 00 40 4d call  20778             ! strcpy
10648: 01 00 00 00 nop
1064c: 11 00 00 82 sethi %hi(0x20800), %o0
10650: 90 12 20 a8 or    %o0, 0xa8, %o0    ! 208a8 <b>
10654: 7f ff ff ee call  1060c <baz>
10658: 01 00 00 00 nop
1065c: 81 c7 e0 08 ret
10660: 81 e8 00 00 restore
10664: 81 c3 e0 08 retl
10668: ae 03 c0 17 add  %o7, %l7, %l7
```

Shared Libraries and Dynamic Linking

The 1980s GUI/WIMP revolution required many large libraries (the Athena widgets, Motif, etc.)

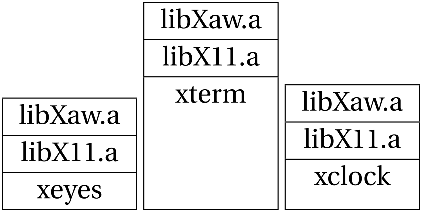
Under a *static linking* model, each executable using a library gets a copy of that library's code.



Shared Libraries and Dynamic Linking

The 1980s GUI/WIMP revolution required many large libraries (the Athena widgets, Motif, etc.)

Under a *static linking* model, each executable using a library gets a copy of that library's code.

Address 0: 

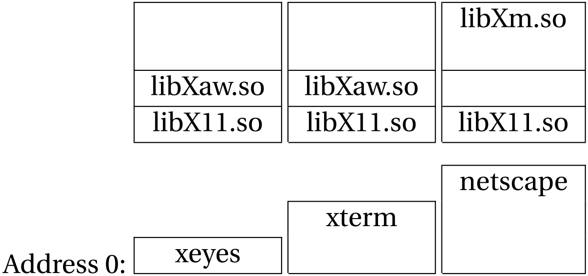
Wasteful: running many GUI programs at once fills memory with nearly identical copies of each library.

Something had to be done: another level of indirection.

Shared Libraries: First Attempt

Most code makes assumptions about its location.

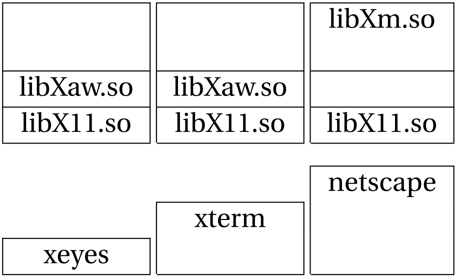
First solution (early Unix System V R3) required each shared library to be located at a unique address:



Shared Libraries: First Attempt

Most code makes assumptions about its location.

First solution (early Unix System V R3) required each shared library to be located at a unique address:



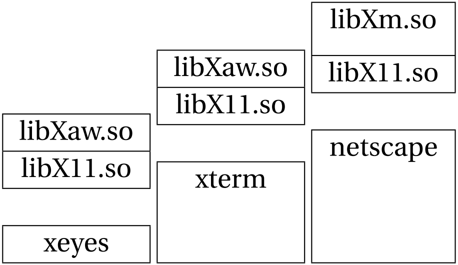
Address 0:

Obvious disadvantage: must ensure each new shared library located at a new address.

Works fine if there are only a few libraries; tended to discourage their use.

Shared Libraries

Problem fundamentally is that each program may need to see different libraries each at a different address.



Position-Independent Code

Solution: Require the code for libraries to be position-independent.
Make it so they can run anywhere in memory.

As always, add another level of indirection:

- All branching is PC-relative
- All data must be addressed relative to a base register.
- All branching to and from this code must go through a jump table.

Position-Independent Code for bar()

Normal unlinked code

```
save  %sp, -112, %sp
sethi  %hi(0), %o0
  R_SPARC_HI22 .bss
mov  %o0, %o0
  R_SPARC_LO10 .bss
sethi  %hi(0), %o1
  R_SPARC_HI22 a
mov  %o1, %o1
  R_SPARC_LO10 a
call  14
  R_SPARC_WDISP30 strcpy
nop
sethi  %hi(0), %o0
  R_SPARC_HI22 .bss
mov  %o0, %o0
  R_SPARC_LO10 .bss
call  24
  R_SPARC_WDISP30 baz
nop
ret
restore
```

gcc -fpic -shared

```
save  %sp, -112, %sp
sethi  %hi(0x10000), %l7
call  8e0   ! add PC to %l7
add  %l7, 0x198, %l7
ld  [ %l7 + 0x20 ], %o0
ld  [ %l7 + 0x24 ], %o1
```

Actually just a stub

```
call  10a24 ! strcpy

nop
ld  [ %l7 + 0x20 ], %o0
```

call is PC-relative

```
call  10a3c ! baz

nop
ret
restore
```

Prolog Execution

Facts

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

$\downarrow$

Query

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

$\rightarrow$ Search (Execution)

$\downarrow$

Result

yes

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

in the order they appear

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

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).
```

path(a,a)

path(a,a)=path(X,X)

X=a

yes

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.

path(a,a)

path(a,a)=path(X,Y)

X=a Y=a

edge(a,Z)

edge(a,Z) = edge(a,b)

Z=b

path(b,a)

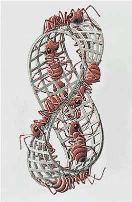
⋮

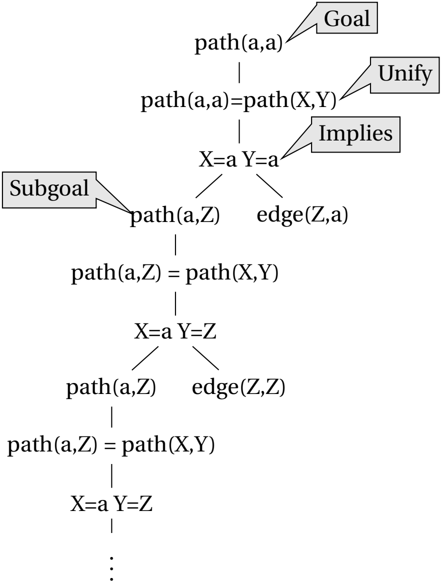
Order Can Cause Infinite Recursion

```
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.

This is the problem with the last path example.

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

"To solve P, solve P…"

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

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

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
```

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.

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

$$\begin{array}{cccccccc} (& \lambda & x & . & + & x & 1 &) \\ & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \end{array}$$

That function of x that adds x to 1

The Syntax of the Lambda Calculus

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

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.

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

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

$$(\lambda x \, . \, \lambda y \, . \, y) \left((\lambda z \, . \, z \; z) \; (\lambda z \, . \, z \; z)\right)$$

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

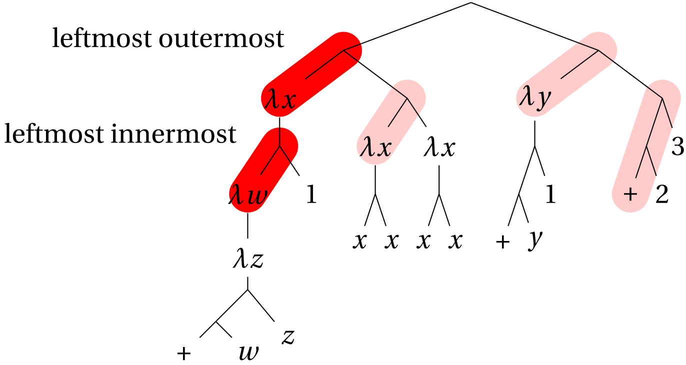
Not all expressions have normal forms, but is there a reliable way to find the normal form if it exists?

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

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



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., $FAC = Y\ H$:

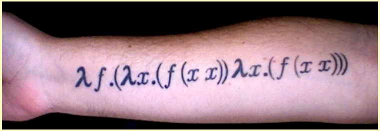
$$
\begin{aligned}
FAC &= H\ FAC \\
Y\ H &= H\ (Y\ H)
\end{aligned}
$$

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

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 $f(f(f(f(\cdots))))$"