Review for the Final

Stephen A. Edwards

Columbia University

Fall 2013
Table of Contents I

The Final

Structure of a Compiler

Scanning

Languages and Regular Expressions

NFAs

Translating REs into NFAs

Building a DFA from an NFA: Subset Construction

Parsing

Resolving Ambiguity
Table of Contents II
Rightmost and Reverse-Rightmost Derivations

Building the LR(0) Automaton
Building an SLR Parsing Table
Shift/Reduce Parsing

Name, Scope, and Bindings

Activation Records
Static Links for Nested Functions

Types

C’s Types
Layout of Records and Unions
Arrays
Name vs. Structural Equivalence
Table of Contents III

Control Flow

Order of Evaluation
Structured Programming
Multi-Way Branching
Recursion vs. Iteration
Applicative vs. Normal-Order Evaluation
Nondeterminism

Code Generation

Intermediate Representations
Optimization and Basic Blocks
Separate Compilation and Linking
Shared Libraries and Dynamic Linking
Table of Contents IV

Logic Programming in Prolog

Prolog Execution

The Prolog Environment

Unification

The Searching Algorithm

Prolog as an Imperative Language
<table>
<thead>
<tr>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda Expressions</td>
</tr>
<tr>
<td>Beta-reduction</td>
</tr>
<tr>
<td>Alpha-conversion</td>
</tr>
<tr>
<td>Reduction Order</td>
</tr>
<tr>
<td>Normal Form</td>
</tr>
<tr>
<td>The Y Combinator</td>
</tr>
</tbody>
</table>
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

```c
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;
}
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
Parser gives an abstract syntax tree.

```c
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```
func gcd(args arg a arg b) seq while a != b if a > b a b -= a b -= b a return a
Translation into 3-Address Code

\[ L0: \text{sne} \; \$1, \; a, \; b \]
\[ \text{seq} \; \$0, \; \$1, \; 0 \]
\[ \text{btrue} \; \$0, \; L1 \quad # \text{while} (a \neq b) \]
\[ \text{sl} \; \$3, \; b, \; a \]
\[ \text{seq} \; \$2, \; \$3, \; 0 \]
\[ \text{btrue} \; \$2, \; L4 \quad # \text{if} (a < b) \]
\[ \text{sub} \; a, \; a, \; b \quad # \text{a} -= \text{b} \]
\[ \text{jmp} \; \text{L5} \]
\[ L4: \text{sub} \; b, \; b, \; a \quad # \text{b} -= \text{a} \]
\[ L5: \text{jmp} \; \text{L0} \]
\[ L1: \text{ret} \; a \]

\[
\text{int} \; \text{gcd}(\text{int} \; a, \; \text{int} \; b) \\
\quad \{ \\
\quad \quad \text{while} \; (a \neq b) \{ \\
\quad \quad \quad \text{if} \; (a > b) \; a \; -= \; b; \\
\quad \quad \quad \text{else} \; b \; -= \; a; \\
\quad \quad \} \\
\quad \quad \text{return} \; a; \\
\quad \}\n\]

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      # while (a != b)
            je    .L3         # if (a < b)
          jle   .L5
          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: \( \epsilon \) (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 \( L = \{ \epsilon, \text{wo} \} \), \( M = \{ \text{man, men} \} \)

**Concatenation**: Strings from one followed by the other

\( LM = \{ \text{man, men, woman, women} \} \)

**Union**: All strings from each language

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

**Kleene Closure**: Zero or more concatenations

\( M^* = \{ \epsilon \} \cup M \cup MM \cup MMM \cdots = \{ \epsilon, \text{man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, manmenmen, ...} \} \)
A standard way to express languages for tokens.

1. \( \epsilon \) 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 \( \cup_{i=0}^{\infty} L^i \) \( (L^0 = \{\epsilon\} \text{ and } L^i = LL^{i-1}) \)
“All strings containing an even number of 0’s and 1’s”

1. Set of states
   \[ S : \{ A, B, C, D \} \]

2. Set of input symbols \( \Sigma : \{0, 1\} \)

3. Transition function \( \sigma : S \times \Sigma \epsilon \rightarrow 2^S \)

<table>
<thead>
<tr>
<th>state</th>
<th>( \epsilon )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>( \emptyset )</td>
<td>{B}</td>
<td>{C}</td>
</tr>
<tr>
<td>B</td>
<td>( \emptyset )</td>
<td>{A}</td>
<td>{D}</td>
</tr>
<tr>
<td>C</td>
<td>( \emptyset )</td>
<td>{D}</td>
<td>{A}</td>
</tr>
<tr>
<td>D</td>
<td>( \emptyset )</td>
<td>{C}</td>
<td>{B}</td>
</tr>
</tbody>
</table>

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

Symbol

Sequence

Choice

Kleene Closure
Translating REs into NFAs

- **Symbol**: $a$
  - Transition: $a \rightarrow a$

- **Sequence**: $r_1 r_2$
  - Transition: $r_1 \rightarrow r_1$
  - Transition: $r_2 \rightarrow r_2$

- **Choice**: $r_1 | r_2$
  - Transition: $\epsilon \rightarrow r_1$
  - Transition: $\epsilon \rightarrow r_2$

- **Kleene Closure**: $(r)^*$
  - Transition: $\epsilon \rightarrow r$
  - Transition: $r \rightarrow \epsilon$
  - Transition: $\epsilon \rightarrow \epsilon$

Translating REs into NFAs

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

Show that the string “aabb” is accepted. Answer:
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 $\epsilon$-closure of the start state
2. For each character $c$,
   - New states: follow all transitions labeled $c$
   - Form the $\epsilon$-closure of the current states
3. Accept if any final state is accepting
Simulating an NFA: \( aabb \), Start

- State 0
- State 1
- State 2
- State 3
- State 4
- State 5
- State 6
- State 7
- State 8
- State 9
- State 10

Transitions:
- \( \epsilon \rightarrow 1 \)
- \( 0 \rightarrow 2 \) on \( a \)
- \( 1 \rightarrow 3 \) on \( a \)
- \( 2 \rightarrow 3 \) on \( a \)
- \( 3 \rightarrow 6 \) on \( a \)
- \( 6 \rightarrow 7 \) on \( a \)
- \( 4 \rightarrow 5 \) on \( b \)
- \( 5 \rightarrow 6 \) on \( b \)
- \( 1 \rightarrow 4 \) on \( b \)
- \( 4 \rightarrow 5 \) on \( b \)
- \( 6 \rightarrow 7 \) on \( a \)
- \( 7 \rightarrow 8 \) on \( a \)
- \( 8 \rightarrow 9 \) on \( b \)
- \( 9 \rightarrow 10 \) on \( b \)
- \( 6 \rightarrow 7 \) on \( a \)
- \( 7 \rightarrow 8 \) on \( a \)
- \( 8 \rightarrow 9 \) on \( b \)
- \( 9 \rightarrow 10 \) on \( b \)
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 $\epsilon$
- 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

```plaintext
{  
  type token = ELSE | ELSEIF
}

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

Diagram:

![DFA Diagram](image-url)
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 \((a | b)^* abb\)
Subset construction for \((a | b)^* abb\)
Result of subset construction for $(a \mid b)^* abb$

Is this minimal?
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

\[ 3 - 4 \times 2 + 5 \]

with the grammar

\[ e \rightarrow e + e | e - e | e \times e | e / e | N \]
Operator Precedence

Defines how “sticky” an operator is.

$$1 \times 2 + 3 \times 4$$

* at higher precedence than +:

$$(1 \times 2) + (3 \times 4)$$

+ at higher precedence than *:

$$1 \times (2 + 3) \times 4$$
Associativity

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

Most operators are left-associative

\[ 1 - 2 - 3 - 4 \]

(left associative)

\[ 1 - (2 - (3 - 4)) \]

(right associative)
A grammar specification:

\[
\text{expr} : \quad \text{expr PLUS expr} \\
\quad | \quad \text{expr MINUS expr} \\
\quad | \quad \text{expr TIMES expr} \\
\quad | \quad \text{expr DIVIDE expr} \\
\quad | \quad \text{NUMBER}
\]

Ambiguous: no precedence or associativity.

Ocamlyacc’s complaint: “16 shift/reduce conflicts.”
Assigning Precedence Levels

Split into multiple rules, one per level

\[
expr : \quad expr \text{ PLUS } expr \\
| \quad expr \text{ MINUS } expr \\
| \quad term \\
\]

\[
term : \quad term \text{ TIMES } term \\
| \quad term \text{ DIVIDE } term \\
| \quad atom \\
\]

\[
atom : \quad NUMBER
\]

Still ambiguous: associativity not defined

Ocamlyacc’s complaint: “8 shift/reduce conflicts.”
Assigning Associativity

Make one side the next level of precedence

\[
\begin{align*}
expr & : \quad expr \ PLUS \ term \\
 & \ | \quad expr \ MINUS \ term \\
 & \ | \quad term \\

term & : \quad term \ TIMES \ atom \\
 & \ | \quad term \ DIVIDE \ atom \\
 & \ | \quad atom \\
atom & : \quad NUMBER
\end{align*}
\]

This is left-associative.

No shift/reduce conflicts.
Rightmost Derivation of \( \text{Id} \ast \text{Id} + \text{Id} \)

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

At each step, expand the rightmost nonterminal.

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
Rightmost Derivation: What to Expand

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

Expand here ↑

Terminals only
Reverse Rightmost Derivation

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

viable prefixes

terminals
Shift/Reduce Parsing Using an Oracle

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

Stack:

- \( \text{Id} \ast \text{Id} + \text{Id} \)
- \( \text{Id} \ast \text{Id} + \text{Id} \)
- \( \text{Id} \ast \text{Id} + \text{Id} \)
- \( \text{Id} \ast \text{Id} + \text{Id} \)
- \( \text{Id} \ast \text{Id} + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( e \)

Input:

- \( e \)
- \( t + e \)
- \( t + t \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + \text{Id} \)
- \( e \)

Actions:

- shift
- shift
- shift
- reduce 4
- reduce 3
- shift
- shift
- reduce 4
- reduce 2
- reduce 1
- 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.*

\[ \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots \]
\[ \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \cdots \]
\[ t + t + \cdots + t + e \]
\[ t + t + \cdots + t + \text{Id} \]
\[ t + t + \cdots + t + \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \]
\[ t + t + \cdots + t \]

\[ \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \]
\[ \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \]
\[ t + e \]
\[ t + e \]
Building the Initial State of the LR(0) Automaton

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

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

1. \( e \rightarrow t + e \)
2. \( e \rightarrow t \)
3. \( t \rightarrow \text{Id} \ast t \)
4. \( t \rightarrow \text{Id} \)

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

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e' \rightarrow \cdot e$</td>
</tr>
<tr>
<td>$e \rightarrow \cdot t + e$</td>
</tr>
<tr>
<td>$e \rightarrow \cdot t$</td>
</tr>
<tr>
<td>$t \rightarrow \cdot \text{Id} \ast t$</td>
</tr>
<tr>
<td>$t \rightarrow \cdot \text{Id}$</td>
</tr>
</tbody>
</table>

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

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 $\text{Id} \ast t$ or $\text{Id}$, so $t \rightarrow \cdot \text{Id} \ast t$ and $t \rightarrow \cdot \text{Id}$.

This reasoning is a closure operation like $\epsilon$-closure in subset construction.
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 $\text{Id}$.

$$
\begin{align*}
e' & \rightarrow \cdot e \\
e & \rightarrow \cdot t + e \\
S0 : e & \rightarrow \cdot t \\
t & \rightarrow \cdot \text{Id} \ast t \\
t & \rightarrow \cdot \text{Id}
\end{align*}
$$
Building the LR(0) Automaton

“Just passed a string derived from e”

\[ S0 : e' \rightarrow e \cdot \]

“Just passed a prefix ending in a string derived from t”

\[ S1 : t \rightarrow \text{Id} \cdot \star t \]
\[ t \rightarrow \text{Id} \cdot \]

The first state suggests a viable prefix can start as any string derived from \( e \), any string derived from \( t \), or \( \text{Id} \).

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 $\text{Id} \ast \cdot t$.
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 $\text{Id} * \cdot t$ and two others.
Building the LR(0) Automaton

$S_0$: $e' \rightarrow e$

$e' \rightarrow e$

$e \rightarrow t + e$

$t \rightarrow \text{Id} \ast t$

$t \rightarrow \text{Id}$

$S_1$: $t \rightarrow \text{Id} \ast t$

$S_2$: $e \rightarrow t \cdot + e$

$e \rightarrow t$

$e \rightarrow \text{Id} \ast t$

$t \rightarrow \text{Id}$

$S_3$: $t \rightarrow \text{Id} \ast t$

$S_4$: $e \rightarrow t + e$

$e \rightarrow t$

$e \rightarrow \text{Id} \ast t$

$t \rightarrow \text{Id}$

$S_5$: $t \rightarrow \text{Id} \ast t$

$S_6$: $e \rightarrow t + e$

$e \rightarrow t$

$e \rightarrow \text{Id} \ast t$

$t \rightarrow \text{Id}$

$S_7$: $e' \rightarrow e$

$Just passed a prefix ending in a string derived from t$.

$Just passed a prefix that ended in an Id$.

$Just passed a string derived from e$.

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).

In $S_2$, a $+$ may be next. This gives $t + e$.

Closure adds 4 more items.

In $S_1$, $\ast$ may be next, giving $\text{Id} \ast t$ and two others.
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

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.
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: $t \rightarrow \text{Id} \ast t$

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

From S3, shift an \textbf{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: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

From S4, shift an \text{Id} and go to S1; or cross an \text{e} or a \text{t}.
Converting the LR(0) Automaton to an SLR Parsing Table

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s1</td>
<td>7 2</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>6 2</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

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: $t \rightarrow \text{Id} * t$
4: $t \rightarrow \text{Id}$

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: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

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: $t \rightarrow \text{Id} * t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>e t</td>
</tr>
<tr>
<td>0</td>
<td>s1</td>
<td>7 2</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>6 2</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id * Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
</tbody>
</table>

0

- **Shift, goto 1**
### Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} \ast t$

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

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1</td>
<td>7 2</td>
</tr>
<tr>
<td>1</td>
<td>r4 s3 r4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4 r2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>6 2</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r3 r3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id * Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>$</td>
<td>Shift, goto 3</td>
</tr>
</tbody>
</table>

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: \( t \rightarrow \text{Id} * t \)
4: \( t \rightarrow \text{Id} \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Id}</td>
<td>+</td>
<td>*</td>
<td>$</td>
<td>e</td>
<td>t</td>
</tr>
<tr>
<td>0</td>
<td>s1</td>
<td></td>
<td></td>
<td>0</td>
<td>\text{Id} * \text{Id} + \text{Id}$</td>
</tr>
<tr>
<td>0</td>
<td>Id</td>
<td>1</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>0</td>
<td>Id</td>
<td>1</td>
<td>3</td>
<td></td>
<td>\text{Id} + \text{Id}$</td>
</tr>
<tr>
<td>0</td>
<td>Id</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>+ \text{Id}$</td>
</tr>
</tbody>
</table>

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: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>$e$</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>$t$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>0</td>
</tr>
</tbody>
</table>

0  s1  7  2
1  r4  s3  r4
2  s4  r2
3  s1
4  s1  6  2
5  r3  r3
6  r1
7  ✓

Stack Input Action

| 0 | Id * Id + Id $ | Shift, goto 1 |
| 0 | Id $ | Shift, goto 3 |
| 0 | Id * Id $ | Shift, goto 1 |
| 0 | Id + Id $ | Shift, goto 1 |
| 0 | Id * Id | + Id $ | Reduce 4 |
| 0 | Id * Id | + Id $ |

Remove the RHS of the rule (here, just \text{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: $t \rightarrow \text{Id} * t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>e t</td>
</tr>
<tr>
<td>0</td>
<td>s1</td>
<td>7 2</td>
</tr>
<tr>
<td>1</td>
<td>r4 s3 r4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4 r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>6 2</td>
</tr>
<tr>
<td>5</td>
<td>r3 r3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\text{Id} \ast \text{Id} \ast \text{Id}$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>1</td>
<td>$\text{Id} + \text{Id}$</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>3</td>
<td>$\text{Id} + \text{Id}$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>1</td>
<td>$\text{Id} \ast \text{Id}$</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0</td>
<td>$\text{Id} \ast \text{Id}$</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0</td>
<td>$\text{Id} \ast \text{Id}$</td>
<td>Shift, goto 4</td>
</tr>
</tbody>
</table>
### Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>$* Id + Id$</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>$* Id + Id$</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Shift, goto 4</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Reduce 2</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>$Id + Id$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Names, Objects, and Bindings

Object 1

Object 2

Object 3

Object 4

binding

binding

binding

binding

Name 1

Name 2

Name 3

Name 4
Typical Stack Layout

↑ higher addresses

argument 2
argument 1
return address
old frame pointer

saved registers
local variables
temporaries/arguments

← frame pointer
← stack pointer

↓ growth of stack
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;
}
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)

```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;
}
```
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;
}

n = 3
return address
last frame pointer
tmp1 = 2
tmp2 =
tmp3 =
n = 2
return address
last frame pointer
tmp1 = 1
tmp2 =
tmp3 =
n = 1
return address
last frame pointer
tmp1 = 1
tmp2 =
tmp3 =
FP

SP
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;
}
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;
}
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)`

```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;
}
```
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;
}
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

```plaintext
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 *)
  in
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” evaluate to?

```
(a static link)
  x = 5
  s = 42
```

```
'a':
  x = 5
  s = 42
```
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) (* b *)
    in
    let e q = b (q+1) in
    e (x+1) (* a *)
```

What does “a 5 42” evaluate to?

```
(x = 5)
(s = 42)

(e = b)
(q = 6)
```
Implementing Nested Functions with Static Links

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

```
a:  
x = 5
s = 42
```
```
e:  
q = 6
```
```
b:  
y = 7
```
```
d:  
w = 8
```
```
c:  
z = 9
```
program example;
var a : integer; (* Outer a *)

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

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

begnin
    a := 2;
    if (readln() = 'b')
        locala
    else
        seta;
    writeln(a)
end
C’s Types: Base Types/Pointers

Base types match typical processor

**Typical sizes:**

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>char</td>
<td>short</td>
<td>int</td>
<td>long</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>double</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pointers (addresses)

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

```c
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

```c
/* function of two arguments returning a char */
char foo(int, double);
```
Structures: each field has own storage

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

Unions: fields share same memory

```c
union token {
    int i;
    double d;
    char *s;
};
```
A record is an object with a collection of fields, each with a potentially different type. In C,

```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:

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

```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:

```c
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`.
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.

16-bit integer: \[ \begin{array}{c} 1 \ 0 \end{array} \]

32-bit integer: \[ \begin{array}{c} 3 \ 2 \ 1 \ 0 \end{array} \]
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).

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

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

Most languages provide array types:

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

```fortran
character(10) i    ! FORTRAN
```

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

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

In C,

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

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

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

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

⇒ Array bounds must be known to access 2D+ arrays
Allocating Arrays in C++

```cpp
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 */
}
```
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.

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

return address $\leftarrow$ FP

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b[n-1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td>b[0]</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$\leftarrow$ FP – ?</td>
</tr>
</tbody>
</table>

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

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

Variables remain constant offset from frame pointer.
Name vs. Structural Equivalence

```c
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**

```c
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

\[
\begin{align*}
\text{tmp} &= b + c; \\
a &= \text{tmp} + d;
\end{align*}
\]

(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.

<table>
<thead>
<tr>
<th>Hindu-Arabic</th>
<th>Roman</th>
<th>Greek</th>
<th>Egyptian</th>
<th>Gravity</th>
<th>Babylonian</th>
<th>Chinese</th>
<th>Meyer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I</td>
<td>A</td>
<td>I</td>
<td>⌀</td>
<td>I</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>1</td>
<td>II</td>
<td>B</td>
<td>II</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>2</td>
<td>III</td>
<td>Γ</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>3</td>
<td>IV</td>
<td>Δ</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>4</td>
<td>V</td>
<td>E</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>5</td>
<td>VI</td>
<td>F</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>6</td>
<td>VII</td>
<td>Z</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>7</td>
<td>VIII</td>
<td>H</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>8</td>
<td>IX</td>
<td>Θ</td>
<td>III</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>I</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
<td>N</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>50</td>
<td>C</td>
<td>P</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
<td>⌀</td>
</tr>
</tbody>
</table>
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?
What does \texttt{bar()} return?

GCC returned 25.

Sun’s C compiler returned 20.

C says expression evaluation order is implementation-dependent.
Java prescribes left-to-right evaluation.

```java
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.
When you write

```c
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.

```c
cost = disaster_possible ? avoid_it() : cause_it();
```
Logical Operators

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

```java
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:

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

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

A cleaner version in C using structured control flow:

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

An even better version

```c
for (i = 0; i < 10; i++)
    printf("%d\n", i);
```
Gotos vs. Structured Programming

Break and continue leave loops prematurely:

```c
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:

```java
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:

```pascal
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:

```c
void consume_line(char *line) {
  if (line[0] == '%') return;
}
```
Multi-way Branching

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

```java
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

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

Obvious way:

```java
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:

```c
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:

```c
/* 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:

```c
double total = 0;
for (i = 0; i <= 10; i++)
    total += f(i);
```
Tail-Recursion and Iteration

```c
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

```ocaml
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

```c
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:

```c
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

```c
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;
    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.

```c
#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”
C does not define argument evaluation order:

```c
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.

```c
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

```java
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

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

- Folding constant expressions
  \[1 + 3 \rightarrow 4\]

- Removing dead code
  \[\text{if}(0)\{\ldots\} \rightarrow \text{nothing}\]

- Moving variables from memory to registers
  \[\text{ld} \ [\text{fp}+68], \%i1\]
  \[\text{sub} \ %i0, \%i1, \%i0 \rightarrow \text{sub} \ %o1, \%o0, \%o1\]
  \[\text{st} \ %i0, [\text{fp}+72]\]

- 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.

```plaintext
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

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.
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

ret a
Separate Compilation and Linking

- **Compiler**: foo.c, bar.c
  - foo.s, bar.s
  - foo.o, bar.o
  - printf.o, fopen.o, malloc.o

- **Assembler**: cc, as

- **Archiver**: libc.a

- **Linker**: ld

- **Output**: foo
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

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);
    }
    baz(a);
}
```

**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()

file1.o.a="Hello"
main()
baz()

file2.o.char b[6]
bar()

.a.out.text segment
main()
baz()
bar()

.data segment
a="Hello"

..bss segment
char b[6]
Linking

file1.o
  a="Hello"
  main()
  baz()

.a.out
  .text segment
  main()
  baz()
  bar()
  .data segment
  a="Hello"
  .bss segment
  char b[6]

file2.o
  .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); 
}
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:
<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>Offset</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>038</td>
<td>0</td>
<td>0</td>
<td>034</td>
<td>2**2</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>008</td>
<td>0</td>
<td>0</td>
<td>070</td>
<td>2**3</td>
</tr>
<tr>
<td>2</td>
<td>.bss</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>078</td>
<td>2**0</td>
</tr>
<tr>
<td>3</td>
<td>.rodata</td>
<td>008</td>
<td>0</td>
<td>0</td>
<td>078</td>
<td>2**3</td>
</tr>
</tbody>
</table>

SYMBOL TABLE:
<table>
<thead>
<tr>
<th>Addr</th>
<th>Type</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>g</td>
<td>O</td>
<td>.data 006 a</td>
</tr>
<tr>
<td>0000</td>
<td>g</td>
<td>F</td>
<td>.text 014 main</td>
</tr>
<tr>
<td>0000</td>
<td><em>UND</em></td>
<td>000</td>
<td>bar</td>
</tr>
<tr>
<td>0014</td>
<td>g</td>
<td>F</td>
<td>.text 024 baz</td>
</tr>
<tr>
<td>0000</td>
<td><em>UND</em></td>
<td>000</td>
<td>printf</td>
</tr>
</tbody>
</table>

RELOCATION RECORDS FOR [.text]:
<table>
<thead>
<tr>
<th>OFFSET</th>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004</td>
<td>R_SPARC_WDISP30</td>
<td>bar</td>
</tr>
<tr>
<td>001c</td>
<td>R_SPARC_HI22</td>
<td>.rodata</td>
</tr>
<tr>
<td>0020</td>
<td>R_SPARC_LO10</td>
<td>.rodata</td>
</tr>
<tr>
<td>0028</td>
<td>R_SPARC_WDISP30</td>
<td>printf</td>
</tr>
</tbody>
</table>
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

- Combine object files
- Relocate each function’s code
- Resolve previously unresolved symbols
Linking Resolves Symbols

**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);
}
```
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.

<table>
<thead>
<tr>
<th>Address 0:</th>
<th>libXaw.a</th>
<th>libX11.a</th>
<th>libXaw.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>xeyes</td>
<td>xterm</td>
<td>xclock</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Address 0:</th>
<th>libXaw.a</th>
<th>libXaw.a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libX11.a</td>
<td>libX11.a</td>
</tr>
<tr>
<td>xeyes</td>
<td>xterm</td>
<td>xclock</td>
</tr>
</tbody>
</table>

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:

Address 0:  

- libXaw.so  
- libXaw.so  
- libX11.so  
- xeyes  
- xterm  
- netscape

libXm.so  
libXaw.so  
libX11.so
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:

```
libXm.so
libXaw.so
libXaw.so
libX11.so
libX11.so
libX11.so

netscape
xterm

Address 0:  
exeyes
```

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.
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).

Query

?- nerd(stephen).

→ Search (Execution)

Result

yes
Simple Searching

Starts with the query:

?- nerd(stephen).

Can we convince ourselves that \texttt{nerd(stephen)} is true given the facts we have?

\texttt{techer(stephen).}
\texttt{nerd(X) :- techer(X).}

First says \texttt{techer(stephen)} is true. Not helpful.

Second says that we can conclude \texttt{nerd(X)} is true if we can conclude \texttt{techer(X)} is true. More promising.
Unifying \textit{nerd}(\textit{stephen}) with the head of the second rule, \textit{nerd}(X), we conclude that \(X = \text{stephen}\).

We’re not done: for the rule to be true, we must find that all its conditions are true. \(X = \text{stephen}\), so we want \textit{techer}(\textit{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.
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:

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

“Functor”

Paren must follow immediately.

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.
n0 % Mismatched constants

?- 5.3 = a.
n0 % Mismatched constants

?- 5.3 = X.
X = 5.3 ? ;
yes % Variables unify

?- foo(a,X) = foo(X,b).
n0 % 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).
n0 % 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
\[ e = \text{unify}(g, h, e) \]
if successful,
for each term \( t_1, \ldots, t_n \),
\[ e = \text{search}(t_k, e) \]
if all successful, return \( e \)
return no

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).

path(a,a)                          | path(a,a)=path(X,X)
|                                  | X=a
|                                yes

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

\[\begin{align*}
edge(a, b). & \quad edge(b, c). \\
edge(c, d). & \quad edge(d, e). \\
edge(b, e). & \quad edge(d, f).
\end{align*}\]

\[\begin{align*}
path(X, Y) :& \quad \text{edge}(X, Z), \ path(Z, Y).
\end{align*}\]

\[\begin{align*}
path(X, X).
\end{align*}\]

Consider the query

\[\begin{align*}
?\:- & \ path(a, a).
\end{align*}\]

Will eventually produce the right answer, but will spend much more time doing so.
Order Can Cause Infinite Recursion

\[\text{edge}(a, b). \text{edge}(b, c). \text{edge}(c, d). \text{edge}(d, e). \text{edge}(b, e). \text{edge}(d, f).\]

\[\text{path}(X, Y) :- \text{path}(X, Z), \text{edge}(Z, Y).\]

\[\text{path}(X, X).\]

Consider the query

\[\text{?- path}(a, a).\]
Prolog as an Imperative Language

A declarative statement such as

\[ P \text{ if } Q \text{ and } R \text{ and } S \]

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…”
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.

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

<table>
<thead>
<tr>
<th>?- nerd(X).</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = stephen</td>
</tr>
</tbody>
</table>

| yes |
Lambda Expressions

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

\[(+ \, 4 \, 5)\]

Evaluation: select a *redex* and evaluate it:

\[(+ \, (* \, 5 \, 6) \, (* \, 8 \, 3)) \rightarrow (+ \, 30 \, (* \, 8 \, 3))\]
\[\rightarrow (+ \, 30 \, 24)\]
\[\rightarrow 54\]

Often more than one way to proceed:

\[(+ \, (* \, 5 \, 6) \, (* \, 8 \, 3)) \rightarrow (+ \, (* \, 5 \, 6) \, 24)\]
\[\rightarrow (+ \, 30 \, 24)\]
\[\rightarrow 54\]

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:

\[ (+ 3 4) = ((+ 3) 4) \rightarrow 7 \]
Lambda Abstraction

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

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

That function of $x$ that adds $x$ to 1
The Syntax of the Lambda Calculus

\[
\text{expr} ::= \text{expr expr} \\
| \lambda \text{variable} . \text{expr} \\
| \text{constant} \\
| \text{variable} \\
| (\text{expr})
\]

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

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

\[(\lambda x \cdot + x 1) 4 \rightarrow (+ 4 1) \rightarrow 5\]

The argument may appear more than once

\[(\lambda x \cdot + x x) 4 \rightarrow (+ 4 4) \rightarrow 8\]

or not at all

\[(\lambda x \cdot 3) 5 \rightarrow 3\]
Free and Bound Variables

Here, $x$ is like a function argument but $y$ is like a global variable. Technically, $x$ occurs bound and $y$ occurs free in

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

However, both $x$ and $y$ occur free in

$$((+ x y))$$
Beta-Reduction More Formally

\[(\lambda x \cdot E) F \rightarrow_\beta E'\]

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

\[(\lambda x \cdot + (- x \, 1)) \, x \, 3\]

so

\[(\lambda x \cdot (\lambda x \cdot + (- x \, 1)) \, x \, 3) \, 9 \rightarrow (\lambda x \cdot + (- x \, 1)) \, 9 \, 3 \rightarrow + (- 9 \, 1) \, 3 \rightarrow + 8 \, 3 \rightarrow 11\]
One way to confuse yourself less is to do $\alpha$-conversion: renaming a $\lambda$ argument and its bound variables.

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

\[
(\lambda x \cdot (\lambda x \cdot + (\neg x \ 1)) \ x \ 3) \ 9 \leftrightarrow (\lambda x \cdot (\lambda y \cdot + (\neg y \ 1)) \ x \ 3) \ 9 \\
\rightarrow ((\lambda y \cdot + (\neg y \ 1)) \ 9 \ 3) \\
\rightarrow (+ (\neg 9 \ 1) \ 3) \\
\rightarrow (+ \ 8 \ 3) \\
\rightarrow 11
\]
Running $\beta$-reduction in reverse, leaving the “meaning” of a lambda expression unchanged, is called \textit{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.

\[(\lambda x . \lambda y . y) \ ( (\lambda z . z \ z) \ (\lambda z . z \ z))\]

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 $\beta$-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.
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

\[ \left( \left( \lambda x . (\lambda w . \lambda z . + w z) 1 \right) \right) \left( (\lambda x . x x) (\lambda x . x x) \right) \left( (\lambda y . + y 1) (+ 2 3) \right) \]
Recursion

Where is recursion in the lambda calculus?

\[ FAC = \left( \lambda n . IF (= n 0) 1 \left( \ast n \left( FAC \left( - n 1 \right) \right) \right) \right) \]

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

\[ FAC = \left( \lambda n \ldots . FAC \ldots \right) \leftarrow_\beta \left( \lambda f . \left( \lambda n \ldots . f \ldots \right) \right) FAC \]

\[ = H \left( FAC \right) \]

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

\[ H = \lambda f . \lambda n . IF (= n 0) 1 \left( \ast n \left( f \left( - n 1 \right) \right) \right) \]
Recursion

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

\[
FAC = H \cdot FAC
\]

\[
Y H = H (Y H)
\]

\[
FAC 1 = Y H 1
= H (Y H) 1
= (\lambda f . \lambda n . IF (= n 0) 1 (\ast n (f (\neg n 1)))) (Y H) 1
\]

\[
\rightarrow (\lambda n . IF (= n 0) 1 (\ast n ((Y H) (\neg n 1)))) 1
\]

\[
\rightarrow IF (= 1 0) 1 (\ast 1 ((Y H) (\neg 1 1)))
\]

\[
\rightarrow * 1 (Y H 0)
\]

\[
= * 1 (H (Y H) 0)
\]

\[
= * 1 ((\lambda f . \lambda n . IF (= n 0) 1 (\ast n (f (\neg n 1)))) (Y H) 0)
\]

\[
\rightarrow * 1 ((\lambda n . IF (= n 0) 1 (\ast n (Y H (\neg n 1)))) 0)
\]

\[
\rightarrow * 1 (IF (= 0 0) 1 (\ast 0 (Y H (\neg 0 1))))
\]

\[
\rightarrow * 1 1
\]

\[
\rightarrow 1
\]
Here’s the eye-popping part: $Y$ can be a simple lambda expression.

\[
Y = \lambda f \cdot (\lambda x \cdot f(x x)) (\lambda x \cdot f(x x))
\]

\[
YH = \left( \lambda f \cdot (\lambda x \cdot f(x x)) (\lambda x \cdot f(x x)) \right) H
\]
\[
\rightarrow (\lambda x \cdot H(x x)) (\lambda x \cdot H(x x))
\]
\[
\rightarrow H \left( (\lambda x \cdot H(x x)) (\lambda x \cdot H(x x)) \right)
\]
\[
\leftrightarrow H \left( \left( \lambda f \cdot (\lambda x \cdot f(x x)) (\lambda x \cdot f(x x)) \right) H \right)
\]
\[
= H(YH)
\]

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