Review for the Final Exam

Stephen A. Edwards

Columbia University

Fall 2021
Table of Contents

The Final Exam

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
- Rightmost and Reverse-Rightmost Derivations
- Building the LR(0) Automaton
- First and Follow
- Building an SLR Parsing Table
- Shift/Reduce Parsing
Table of Contents II

Runtime Environments
  Storage Classes and Memory Layout
  The Stack and Activation Records
  The Heap
  Automatic Garbage Collection
  Objects and Inheritance

Code Generation
  Intermediate Representations
  Optimization and Basic Blocks
The Final

150 minutes
1:10 - 3:40, Friday, December 10th
Closed book, notes, Internet
One double-sided 8.5” × 11” sheet of notes of your own devising.
Comprehensive: Anything discussed in class is fair game
Little, if any, programming. This is not a test on OCaml
Details of OCaml/C/C++/Java syntax not required
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

*Text file is a sequence of characters*
Lexical Analysis Gives Tokens

```c
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.
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

```
func gcd(args arg a, arg b) seq
  while a != b
    if a > b
      a -= b
    else
      b -= a
    return a
```
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;
}
Generation of 80386 Assembly

gcd:

```
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: cmp %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: \(\epsilon\) (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
Let $L = \{ \epsilon, \text{wo} \}$, $M = \{ \text{man, men} \}$

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

$L M = \{ \text{man, men, woman, women} \}$

**Union:** All strings from each language

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

**Kleene Closure:** Zero or more concatenations

$M^* = \{ \epsilon \} \cup M \cup MM \cup MMM \cdots = \{ \epsilon, \text{man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, ...} \}$
Regular Expressions over an Alphabet $\Sigma$

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 $\cup_{i=0}^{\infty} L^i$ ($L^0 = \{\epsilon\}$ 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 \( \sigma : S \times \Sigma^* \rightarrow 2^S \)

<table>
<thead>
<tr>
<th>state</th>
<th>( \varepsilon )</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**
  - $a$

- **Sequence**
  - $r_1 r_2$

- **Choice**
  - $r_1 | r_2$

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

\[\text{Show that the string "} aabb\text{" is accepted. Answer:} \]
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
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 }
```
Deterministic Finite Automata

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

rule token =
parse "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 \((a | b)^* abb\)
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

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

with the grammar

\[ e \rightarrow e + e \mid e - e \mid e \times e \mid e / e \mid N \]
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

\[ 1 - 2 - 3 - 4 \]

\[
\begin{array}{c}
\phantom{1} - \\
\phantom{2} - \\
\phantom{3} - \\
\phantom{4} -
\end{array}
\]

\[
\begin{array}{c}
1 \phantom{-} - \\
2 \phantom{-} - \\
3 \phantom{-} - \\
4
\end{array}
\]

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

left associative

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

right associative
Fixing Ambiguous Grammars

A grammar specification:

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

Ambiguous: no precedence or associativity.

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

Split into multiple rules, one per level

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

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

\[
\text{atom} : \quad \text{NUMBER}
\]

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 $\text{Id} \times \text{Id} + \text{Id}$

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

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

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

- 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.*

\[
\begin{align*}
\text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \\
\text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \\
\text{t} + \text{t} + \cdots + \text{t} + e \\
\text{t} + \text{t} + \cdots + \text{t} + \text{Id} \\
\text{t} + \text{t} + \cdots + \text{t} + \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \\
\text{t} + \text{t} + \cdots + \text{t} \\
\text{t} + \text{t} + \cdots + \text{t} + \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \\
\text{t} + \text{t} + \cdots + \text{t}
\end{align*}
\]
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 “$e' \rightarrow \cdot e$”
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

\[
\begin{align*}
1: & e \rightarrow t + e \\
2: & e \rightarrow t \\
3: & t \rightarrow \text{id} \ast t \\
4: & t \rightarrow \text{id}
\end{align*}
\]

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 \( t \rightarrow \cdot \text{id} \).
Building the LR(0) Automaton

\[\begin{align*}
\text{S0: } & \quad e' \to \cdot e \\
& \quad e \to \cdot t + e \\
& \quad e \to \cdot t \\
& \quad t \to \cdot \text{Id} * t \\
& \quad t \to \cdot \text{Id}
\end{align*}\]

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

**S7**: $e' \rightarrow e \cdot$

```
S0: e \rightarrow t
    t \rightarrow \cdot Id \ast t
    t \rightarrow \cdot Id

S1: t \rightarrow Id \ast t
    t \rightarrow Id.
```

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

**S2**: $e \rightarrow t \cdot + e$

```
S2: e \rightarrow t \cdot.
```

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

S0:
- \( e' \rightarrow e \cdot \)
- \( e \rightarrow t \cdot e \)
- \( t \rightarrow \cdot t \cdot e \)
- \( t \rightarrow \cdot \text{Id} \cdot t \)
- \( t \rightarrow \cdot \text{Id} \)

S1:
- \( t \rightarrow \text{Id} \cdot e \cdot t \)
- \( t \rightarrow \cdot \text{Id} \cdot e \)
- \( t \rightarrow \cdot \text{Id} \)

S2:
- \( e \rightarrow t \cdot e \)
- \( e \rightarrow t \cdot \cdot t \)

S3:
- \( t \rightarrow \text{Id} \cdot e \cdot t \)

S4:
- \( e \rightarrow t + e \)

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

In S1, * may be next, giving \( \text{Id} \cdot e \cdot t \).
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 $\text{ld} \cdot \cdot t$ and two others.
Building the LR(0) Automaton

\[ S_0 : \begin{align*}
    e' &\rightarrow e \\
    e &\rightarrow t + e \\
    t &\rightarrow \text{Id} * t \\
    \text{Id} &\rightarrow \text{Id}
\end{align*} \]

\[ S_1 : \begin{align*}
    t &\rightarrow \text{Id} * t \\
    \text{Id} &\rightarrow \text{Id}
\end{align*} \]

\[ S_2 : \begin{align*}
    e &\rightarrow t \cdot e \\
    e &\rightarrow t \\
    t &\rightarrow \text{Id} * t \\
    \text{Id} &\rightarrow \text{Id}
\end{align*} \]

\[ S_3 : \begin{align*}
    t &\rightarrow \text{Id} * t \\
    \text{Id} &\rightarrow \text{Id}
\end{align*} \]

\[ S_4 : \begin{align*}
    e &\rightarrow t + e \\
    e &\rightarrow t + e \\
    t &\rightarrow \text{Id} * t \\
    \text{Id} &\rightarrow \text{Id}
\end{align*} \]

\[ S_5 : \begin{align*}
    t &\rightarrow \text{Id} * t
\end{align*} \]

\[ S_6 : \begin{align*}
    e &\rightarrow t + e
\end{align*} \]

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). In \( S_2 \), a \( + \) may be next. This gives \( t + \cdot e \). Closure adds 4 more items. In \( S_1 \), \( * \) may be next, giving \( \text{Id} * \cdot t \) and two others.
The first function

If you can derive a string that starts with terminal $t$ from a sequence of terminals and nonterminals $\alpha$, then $t \in \text{first}(\alpha)$.

1. If $X$ is a terminal, $\text{first}(X) = \{ X \}$.
2. If $X \rightarrow \epsilon$, then add $\epsilon$ to $\text{first}(X)$.
3. If $X \rightarrow Y_1 \cdots Y_k$ and $\epsilon \in \text{first}(Y_1), \epsilon \in \text{first}(Y_2), \ldots$, and $\epsilon \in \text{first}(Y_{i-1})$ for $i = 1, \ldots, k$ for some $k$,
   add $\text{first}(Y_i) - \{ \epsilon \}$ to $\text{first}(X)$
   $X$ starts with anything that appears after skipping empty strings. Usually just $\text{first}(Y_1) \in \text{first}(X)$
4. If $X \rightarrow Y_1 \cdots Y_K$ and $\epsilon \in \text{first}(Y_1), \epsilon \in \text{first}(Y_2), \ldots$, and $\epsilon \in \text{first}(Y_k)$, add $\epsilon$ to $\text{first}(X)$
   If all of $X$ can be empty, $X$ can be empty

1: $e \rightarrow t + e$  \hspace{1cm} \text{first}(\text{Id}) = \{ \text{Id} \}$
2: $e \rightarrow t$  \hspace{1cm} \text{first}(t) = \{ \text{Id} \}$ because $t \rightarrow \text{Id} \ast t$ and $t \rightarrow \text{Id}$
3: $t \rightarrow \text{Id} \ast t$  \hspace{1cm} \text{first}(e) = \{ \text{Id} \}$ because $e \rightarrow t + e$, $e \rightarrow t$, and $\text{first}(t) = \{ \text{Id} \}$. 
4: $t \rightarrow \text{Id}$
First and $\epsilon$

$\epsilon \in \text{first}(\alpha)$ means $\alpha$ can derive the empty string.

1. If $X$ is a terminal, first($X$) = \{X\}.
2. If $X \rightarrow \epsilon$, then add $\epsilon$ to first($X$).
3. If $X \rightarrow Y_1 \cdots Y_k$ and
   $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, \ldots, and $\epsilon \in \text{first}(Y_{i-1})$
   for $i = 1, \ldots, k$ for some $k$,
   add first($Y_i$) − \{\epsilon\} to first($X$)
4. If $X \rightarrow Y_1 \cdots Y_K$ and
   $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, \ldots, and $\epsilon \in \text{first}(Y_K)$,
   add $\epsilon$ to first($X$)

\[
\begin{align*}
X & \rightarrow YZA \\
Y & \rightarrow b \\
Y & \rightarrow Z \\
Z & \rightarrow c \\
Z & \rightarrow W \\
W & \rightarrow W \\
W & \rightarrow d
\end{align*}
\]

\begin{align*}
\text{first}(b) &= \{b\} & \text{first}(c) &= \{c\} & \text{first}(d) &= \{d\} & (1) \\
\text{first}(W) &= \{\epsilon\} \cup \text{first}(d) = \{\epsilon, d\} & (2, 3) \\
\text{first}(Z) &= \text{first}(c) \cup (\text{first}(W) - \{\epsilon\}) \cup \{\epsilon\} = \{\epsilon, c, d\} & (3, 3, 4) \\
\text{first}(Y) &= \{\epsilon\} \cup \{b\} = \{\epsilon, b\} & (2, 3) \\
\text{first}(X) &= (\text{first}(Y) - \{\epsilon\}) \cup (\text{first}(Z) - \{\epsilon\}) \cup \text{first}(a) = \{a, b, c, d\} & (3, 3, 3)
\end{align*}
The follow function

If $t$ is a terminal, $A$ is a nonterminal, and $\ldots At\ldots$ can be derived, then $t \in \text{follow}(A)$.

1. Add $\$ (“end-of-input”) to follow($S$) (start symbol).
   End-of-input comes after the start symbol
2. For each prod. $\to \ldots A\alpha$, add first($\alpha$) $\setminus \{\epsilon\}$ to follow($A$).
   $A$ is followed by the first thing after it
3. For each prod. $A \to \ldots B$ or $A \to \ldots B\alpha$ where $\epsilon \in \text{first}(\alpha)$, then add everything in follow($A$) to follow($B$).
   If $B$ appears at the end of a production, it can be followed by whatever follows that production

1: $e \to t + e$           follow($e$) = {$\$\$
2: $e \to t$               follow($t$) = { }
3: $t \to \text{Id} \ast t$
4: $t \to \text{Id}$
first($t$) = {$\text{Id}$}
first($e$) = {$\text{Id}$}

1. Because $e$ is the start symbol
The follow function

If \( t \) is a terminal, \( A \) is a nonterminal, and \( \cdots At\cdots \) can be derived, then \( t \in \text{follow}(A) \).

1. Add $ (“end-of-input”) to follow\((S)\) (start symbol). 
   *End-of-input comes after the start symbol*

2. For each prod. \( \rightarrow \cdots A\alpha \), add first\((\alpha) – \{\epsilon\}\) to follow\((A)\).
   *A is followed by the first thing after it*

3. For each prod. \( A \rightarrow \cdots B \) or \( A \rightarrow \cdots B\alpha \) where \( \epsilon \in \text{first}(\alpha) \), then add everything in follow\((A)\) to follow\((B)\).
   *If \( B \) appears at the end of a production, it can be followed by whatever follows that production*

\[
\begin{align*}
1: e & \rightarrow t + e & \text{follow}(e) = \{\$\} \\
2: e & \rightarrow t & \text{follow}(t) = \{+\} \\
3: t & \rightarrow \text{Id} * t \\
4: t & \rightarrow \text{Id} & \text{first}(t) = \{\text{Id}\} \\
\end{align*}
\]

2. Because \( e \rightarrow t + e \) and first\((+\) = \{+\}
The follow function

If $t$ is a terminal, $A$ is a nonterminal, and $\cdots At\cdots$ can be derived, then $t \in \text{follow}(A)$.

1. Add $\$ ("end-of-input") to follow($S$) (start symbol).
   End-of-input comes after the start symbol

2. For each prod. $\rightarrow \cdots A\alpha$, add first($\alpha$) − {e} to follow($A$).
   A is followed by the first thing after it

3. For each prod. $A \rightarrow \cdots B$ or $A \rightarrow \cdots B\alpha$ where $e \in \text{first}(\alpha)$,
   then add everything in follow($A$) to follow($B$).
   If $B$ appears at the end of a production, it can be followed by whatever follows that production

1: $e \rightarrow t + e$

$\text{follow}(e) = \{\}$

2: $e \rightarrow t$

$\text{follow}(t) = \{+, \}$

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

3. Because $e \rightarrow t$ and $\$ \in \text{follow}(e)$

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

$\text{first}(t) = \{\text{Id}\}$

$\text{first}(e) = \{\text{Id}\}$
The follow function

If $t$ is a terminal, $A$ is a nonterminal, and $\cdots At\cdots$ can be derived, then $t \in \text{follow}(A)$.

1. Add $\text{”end-of-input”}$ to $\text{follow}(S)$ (start symbol).

   
   End-of-input comes after the start symbol

2. For each prod. $\rightarrow \cdots A\alpha$, add $\text{first}(\alpha) - \{\varepsilon\}$ to $\text{follow}(A)$.

   $A$ is followed by the first thing after it

3. For each prod. $A \rightarrow \cdots B$ or $A \rightarrow \cdots B\alpha$ where $\varepsilon \in \text{first}(\alpha)$, then add everything in $\text{follow}(A)$ to $\text{follow}(B)$.

   If $B$ appears at the end of a production, it can be followed by whatever follows that production

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
<th>follow(e)</th>
<th>follow(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e \rightarrow t + e$</td>
<td>{$}$</td>
<td>{$+ , $}$</td>
</tr>
<tr>
<td>2</td>
<td>$e \rightarrow t$</td>
<td>follow(e)</td>
<td>follow(t)</td>
</tr>
<tr>
<td>3</td>
<td>$t \rightarrow \text{Id} \ast t$</td>
<td>follow(e)</td>
<td>follow(t)</td>
</tr>
<tr>
<td>4</td>
<td>$t \rightarrow \text{Id}$</td>
<td>follow(e)</td>
<td>follow(t)</td>
</tr>
</tbody>
</table>

Fixed-point reached: applying any rule does not change any set
Converting the LR(0) Automaton to an SLR Parsing Table

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

From S1, shift a \( \ast \) 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

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

<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</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td></td>
</tr>
</tbody>
</table>

From S3, shift an \( \text{Id} \) 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: \(t \rightarrow \text{Id}\)

\[
\begin{array}{c|ccc|c}
\text{State} & \text{Action} & \text{Goto} \\
\hline
\text{Id} & + & \ast & $ & \text{e} \text{ t} \\
0 & s1 & & 7 & 2 \\
1 & r4 & s3 & r4 \\
2 & s4 & r2 \\
3 & s1 \\
4 & s1 & & 6 & 2 \\
\end{array}
\]

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

\[
\begin{align*}
S0: & \quad t \rightarrow \text{Id} \\
S1: & \quad t \rightarrow \text{Id} \\
S2: & \quad e \rightarrow t \\
S3: & \quad \text{Id} \rightarrow * \text{Id} \\
S4: & \quad \text{Id} \rightarrow t \\
S5: & \quad t \rightarrow \text{Id} * t \\
S6: & \quad e \rightarrow t + e \\
S7: & \quad e' \rightarrow e \\
\end{align*}
\]

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

S7: \( e' \rightarrow e \cdot \)

S1: \( t \rightarrow \text{Id} \cdot \)

S2: \( e \rightarrow t \cdot \)

S3

S4

S5: \( t \rightarrow \text{Id} \star t \cdot \)

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

<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</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>5 6</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>6 2</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id</td>
<td>$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Id Id $</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>t</td>
<td>Reduce 2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>Accept</td>
</tr>
</tbody>
</table>

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

Here, the state is 1, the next symbol is \(\ast\), so shift and mark it with state 3.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>Id * Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>(0) Id</td>
<td>* Id + Id$</td>
<td>Shift, goto 3</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>0</td>
<td>s1</td>
<td>7 2</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>6 2</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>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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

<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} + \text{Id} $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>( 0 )</td>
<td>\text{Id}</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>( 0 )</td>
<td>\text{Id} \ast $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>( 0 )</td>
<td>\text{Id} + \text{Id} $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>( 0 )</td>
<td>+ \text{Id} $</td>
<td>Reduce 4</td>
</tr>
</tbody>
</table>

Shift/Reduce parses with an SLR 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>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>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

**Stack** | **Input** | **Action**
---|---|---
0 | \( \text{Id} \ast \text{Id} + \text{Id} \$ | Shift, goto 1
1 | \( \text{Id} \ast + \text{Id} \$ | Shift, goto 3
0 | \( \text{Id} \ast + \text{Id} \$ | Shift, goto 1
1 | \( + \text{Id} \$ | Reduce 4
0 | \( + \text{Id} \$ | Reduce 2

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: $t \rightarrow \text{Id} \ast t$
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>e t</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>

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.”

<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 3</td>
</tr>
<tr>
<td>0 1 3 Id</td>
<td>+ Id $</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 1 3 5 Id t</td>
<td>+ Id $</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0 1 3 1 Id * Id</td>
<td>+ Id $</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 1 3 Id</td>
<td>+ Id $</td>
<td>Reduce 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>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>r2</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>r3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

### Stack, Input, Action

<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</td>
<td>Id $</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>0</td>
<td>Id * Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0</td>
<td>Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0</td>
<td>+ Id $</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0</td>
<td>+ Id $</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0</td>
<td>+ Id $</td>
<td>Shift, goto 4</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.
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></td>
<td>Id</td>
<td>e</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>5</td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Stack | Input | Action
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Id</td>
<td>* Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>1 * Id</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>0 Id</td>
<td>1 * Id</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 t</td>
<td>1 + Id</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 t</td>
<td>1 + Id</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0 t</td>
<td>1 + t</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 e</td>
<td>+ e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 e</td>
<td>+ e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 e</td>
<td>+ e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 e</td>
<td>+ e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 e</td>
<td>+ e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 e</td>
<td>+ e</td>
<td>Reduce 1</td>
</tr>
</tbody>
</table>

Accept
Storage Classes and Memory Layout

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

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

Static: objects allocated at compile time; persist throughout run

Diagram:
- Stack
- Heap
- Static
- Code

Labels:
- High memory
- Stack pointer
- Program break
- Low memory
Static Objects

```java
class Example {
    public static final int a = 3;
    public void hello() {
        System.out.println("Hello");
    }
}
```

Examples
- Static class variable
- Code for hello method
- String constant “Hello”
- Information about the Example class

Advantages
- Zero-cost memory management
- Often faster access (address a constant)
- No out-of-memory danger

Disadvantages
- Size and number must be known beforehand
- Wasteful if sharing is possible
Stack-Allocated Objects

Natural for supporting recursion.

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

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

Each invocation of a procedure gets its own frame (activation record) where it stores its own local variables and bookkeeping information.
An Activation Record: The State Before Calling `bar`

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

(Real C)

```c
int fib(int n) {
    if (n<2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

(Assembly-like C)

```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;
}
```

```
fib(3)
  \  
  fib(2)  fib(1)
  \  
  fib(1)  fib(0)
```
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;
}
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;
}
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;
}
```

```
n = 3
return address
last frame pointer
  tmp1 = 1
tmp2 = 2
tmp3 =
n = 1
```
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;
}
Local arrays with fixed size are easy to stack.

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

<table>
<thead>
<tr>
<th>return address</th>
<th>← FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b[9]</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td>← FP - 48</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
Allocating Variable-Sized Arrays

Variable-sized local arrays aren’t as easy.

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

<table>
<thead>
<tr>
<th>return address</th>
<th>← FP</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>← 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;
}
```

return address ← FP

Variables remain constant offset from frame pointer.
Heap-Allocated Storage

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

Stack enables, but also requires, recursive behavior.

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

(These heaps are different than those in, e.g., heapsort)
struct point {
    int x, y;
};

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

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

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

    /* do something with the array */

    free(points);
}
Dynamic Storage Allocation
Dynamic Storage Allocation

↓ free( )

↓ malloc( )
Dynamic Storage Allocation

\[
\downarrow \text{free(}\quad\text{)}
\]
Dynamic Storage Allocation

\[
downarrow \text{free( } \\
\downarrow \text{malloc( }
\]
Dynamic Storage Allocation

↓ free( )

↓ malloc( )
Dynamic Storage Allocation

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

`malloc()`
- Find an area large enough for requested block
- Mark memory as allocated

`free()`
- Mark the block as unallocated
Simple Dynamic Storage Allocation

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

malloc(

malloc( [gray] )

free()
Simple Dynamic Storage Allocation

malloc(S)

malloc( S )

free(S)

free(S)
Simple Dynamic Storage Allocation

malloc(S S N S S N)

malloc(S S N S S N)

free(S S N S S N)
Simple Dynamic Storage Allocation

malloc( )

free( )
Fragmentation

```c
malloc( ) seven times give
```

```c
free() four times gives
```

```c
malloc( )?
```

Need more memory; can’t use fragmented memory.
Fragmentation and Handles

Standard CS solution: Add another layer of indirection. Always reference memory through “handles.”

The original Macintosh did this to save memory.
Fragmentation and Handles

Standard CS solution: Add another layer of indirection. Always reference memory through “handles.”

The original Macintosh did this to save memory.
Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

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

**Advantages**

Much easier for the programmer

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

**Disadvantages**

Slower, sometimes unpredictably so

May consume more memory
Reference Counting

What and when to free?

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

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

| Reference Counting | 0 | 42, 17 |
Reference Counting

What and when to free?

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

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

What and when to free?

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

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

What and when to free?

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

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

What and when to free?

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

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

What and when to free?

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

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

What and when to free?

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

```ocaml
def a = (42, 17) in
def b = [a;a] in
def c = (1,2)::b in
b
```
Reference Counting

What and when to free?

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

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

What and when to free?

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

```latex
def a = (42, 17) in
def b = [a;a] in
def c = (1,2)::b in
def b
```
Reference Counting

What and when to free?

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

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

Circular structures defy reference counting:

\[ a \xrightarrow{} b \]

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

High overhead (must update counts constantly), although incremental
Mark-and-Sweep

What and when to free?

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

```ocaml
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

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

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

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

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Mark-and-Sweep

What and when to free?

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

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

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

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

Incremental garbage collectors try to avoid doing everything at once

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

Parallel garbage collection tricky

Real-time garbage collection tricky
Single Inheritance

Simple: Add new fields to end of the object

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

Consequence: Derived classes can never remove fields

C++

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

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

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

Equivalent C

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

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

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

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

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

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

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

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

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

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

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

A a1;
A a2;
B b1;
```
```java
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

```java
# 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
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= 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
  \[ 1+3 \rightarrow 4 \]

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

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

- 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.

\[ a = c + 5 + 3; \]
\[ \text{if } (0 + 3) \{ \]
\[ \quad b = c + 8; \quad \rightarrow \quad b = a = c + 8; \]
\[ \}

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
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

B:
    sub a, a, b

C:
    jmp A