Final Exam Review Session

Ronghui Gu

Spring 2024

Columbia University

* Course website: https://verigu.github.io/4115Spring2024/

Announcements

The Virtual Final Exam

Exam Duration: 75 minutes via Zoom, cameras must be on.

Exam Type: Closed book, except for one double-sided sheet of notes prepared by the student.

Materials Needed: 10 white A4 papers for writing answers.

Submission Instructions:

- Write each problem's answer on a separate paper sheet.
- Take photographs of your answers.
- Submit a PDF file through the Gradescope platform immediately after the exam.
- Submission window: 15 minutes post-exam.

Final Project

Final Report: May 9th, 11:59 pm via coursework

Video Submission: 10 mins video

- May 13th, 11:59 pm via coursework
- May 9th, 11:59 pm for graduating students

Instructions:

https://verigu.github.io/4115Spring2024/assignments/project.html

The Big Picture

What is a Programming Language?

A programming language is a notation that a person and a computer can both understand.

- It allows you to express what is the **task** to compute
- It allows a computer to **execute** the computation task

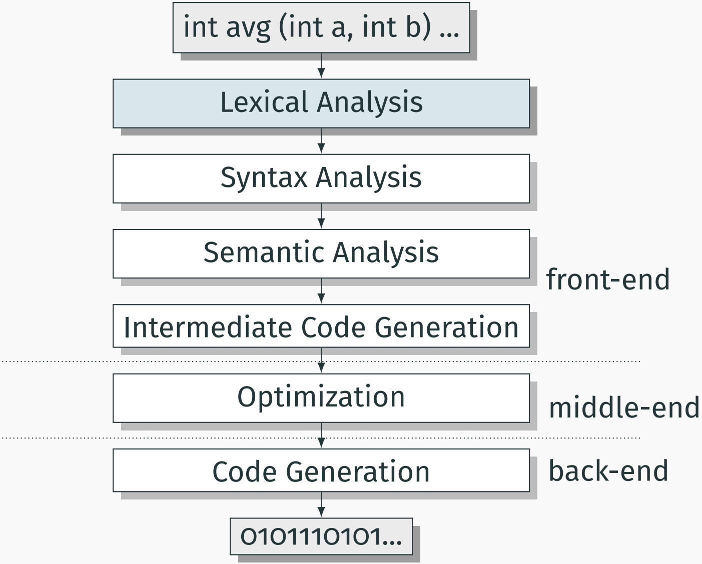
What is a Translator?

A programming language is a notation that a person and a computer can both understand.

- It allows you to express what is the **task** to compute
- It allows a computer to **execute** the computation task

A translator translates what you express to what a computer can execute.

Scanner


int avg (int a, int b) ...
Lexical Analysis
Syntax Analysis
Semantic Analysis
front-end
Intermediate Code Generation
Optimization
middle-end
Code Generation
back-end
0101110101...

Lexical Analysis

Lexical Analysis (Scanning)

Translate a stream of characters to a stream of tokens

$\mathrm{f\ o\ o}$ ␣ $=$ ␣ a $+$ ␣ bar $($ 0 $,$ ␣ 42 $,$ ␣ q $)$ $;$

ID EQUALS ID PLUS ID LPAREN NUM COMMA ID LPAREN SEMI

Token	Lexemes	Pattern
EQUALS	$=$	an equals sign
PLUS	$+$	a plus sign
ID	a foo bar	letter followed by letters or digits
NUM	0 42	one or more digits

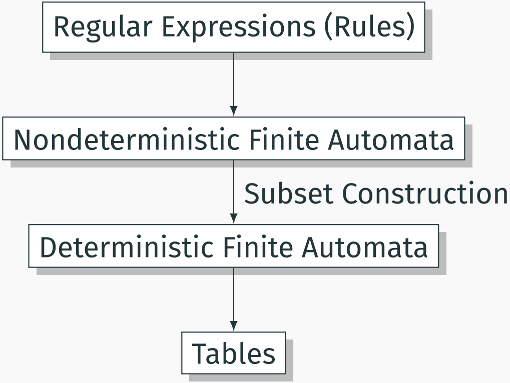
Regular Expressions over an Alphabet Σ

A standard way to express tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote sets $L(r)$ and $L(s)$,

$(r) \mid (s)$	denotes	$L(r) \cup L(s)$
$(r)(s)$		$\{tu : t \in L(r), u \in L(s)\}$
$(r)^*$		$\cup_{i=0}^{\infty} L(r)^i$
	where	$L(r)^0 = \{\epsilon\}$
	and	$L(r)^i = L(r)L(r)^{i-1}$

Implementing Scanners Automatically

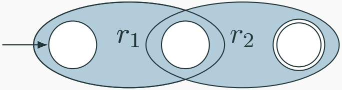


Translating REs into NFAs (Thompson's algorithm)

a

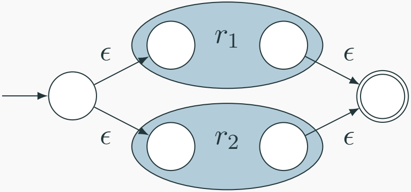
Symbol

$r_1 r_2$



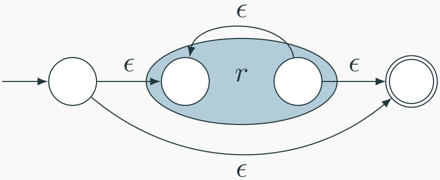
Sequence

$r_1 \mid r_2$



Choice

$(r)^*$



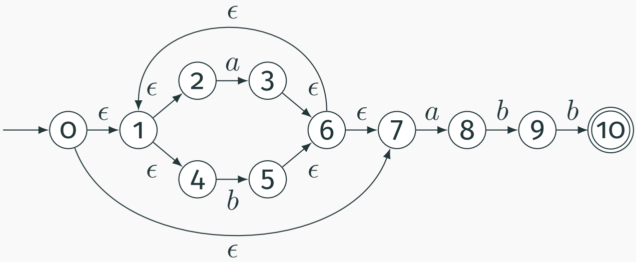
Kleene Closure

Translating REs into NFAs

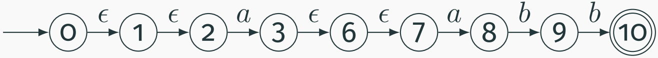
Example: Translate $(a \mid b)^*abb$ into an NFA. Answer:

Translating REs into NFAs

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



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



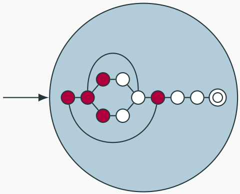
Building a DFA from an NFA

Subset construction algorithm

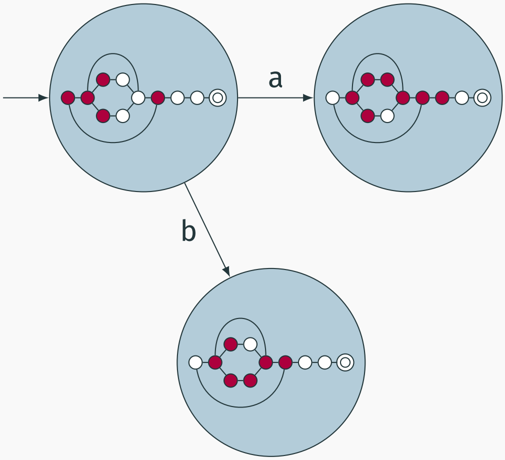
Simulate the NFA for all possible inputs and track the states that appear.

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

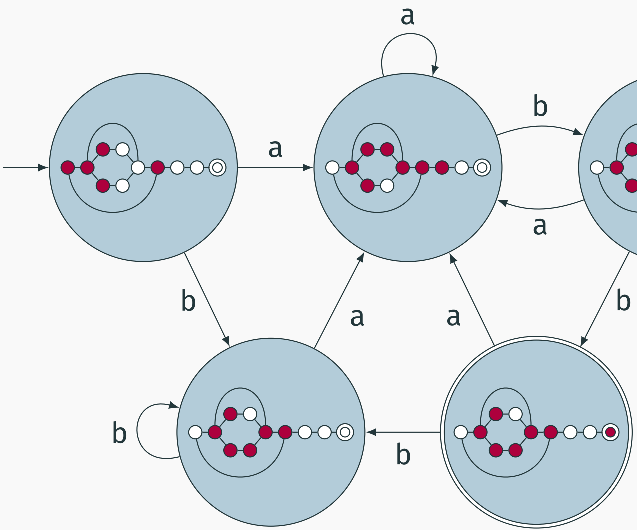
Subset construction for $(a \mid b)^* abb$



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



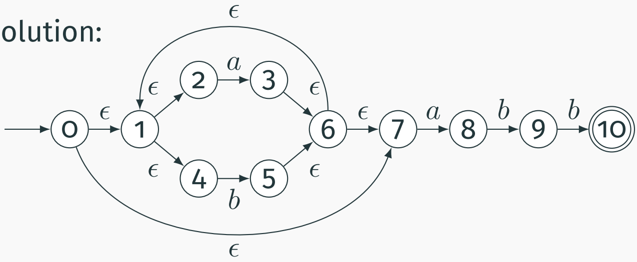
Subset construction for $(a \mid b)^*abb$



Transition Table Used In the Dragon Book

Problem: Translate $(a \mid b)^* abb$ into a DFA.

Solution:

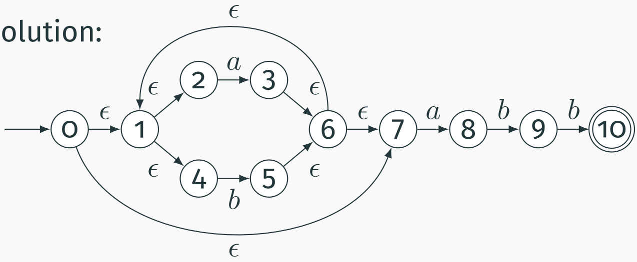


NFA State	DFA State	a	b

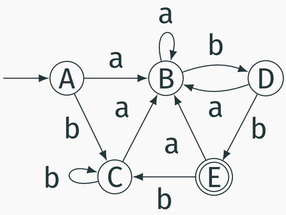
Transition Table Used In the Dragon Book

Problem: Translate $(a \mid b)^*abb$ into a DFA.

Solution:

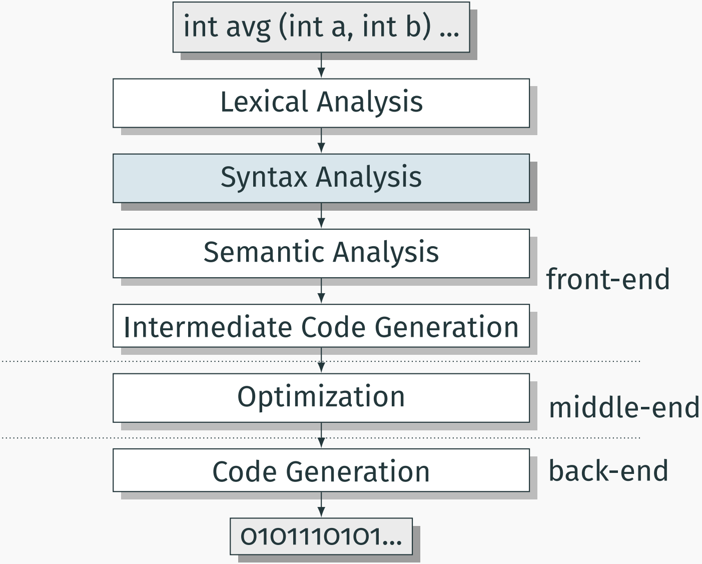


NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10}	E	B	C



Syntax Analysis

Parser



int avg (int a, int b) ...
Lexical Analysis
Syntax Analysis
Semantic Analysis
front-end
Intermediate Code Generation
Optimization
middle-end
Code Generation
back-end
0101110101...

Richer Sentences Are Harder

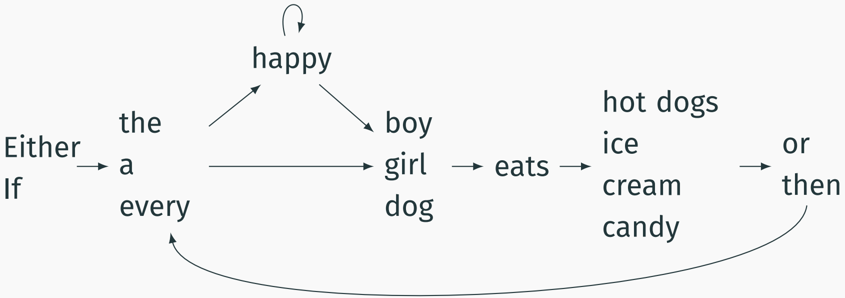
If the boy eats hot dogs, then the girl eats ice cream.

Either the boy eats candy, or every dog eats candy.

Richer Sentences Are Harder

If the boy eats hot dogs, then the girl eats ice cream.

Either the boy eats candy, or every dog eats candy.

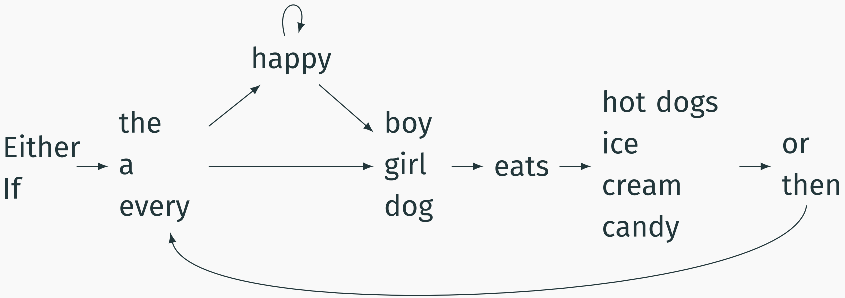


Does this work?

Richer Sentences Are Harder

If the boy eats hot dogs, then the girl eats ice cream.

Either the boy eats candy, or every dog eats candy.

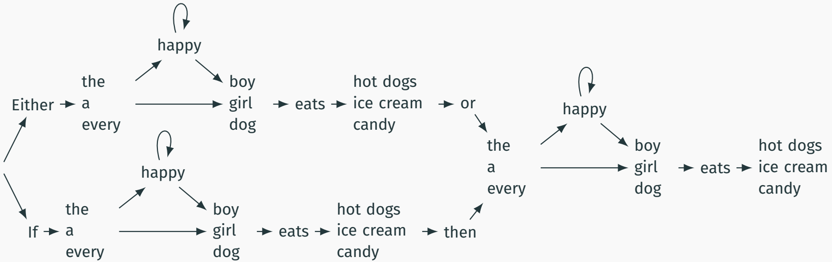


Does this work?

Want to remember the state?

Automata Have Poor Memories

Want to “remember” whether it is an “either-or” or “if-then” sentence. Only solution: duplicate states.



Automata in the form of Production Rules

Problem: automata do not remember where they've been

$S \to \text{Either } A$

$S \to \text{If } A$

$A \to \text{the } B$

$A \to \text{the } C$

$A \to \text{a } B$

$A \to \text{a } C$

$A \to \text{every } B$

$A \to \text{every } C$

$B \to \text{happy } B$

$B \to \text{happy } C$

$C \to \text{boy } D$

$C \to \text{girl } D$

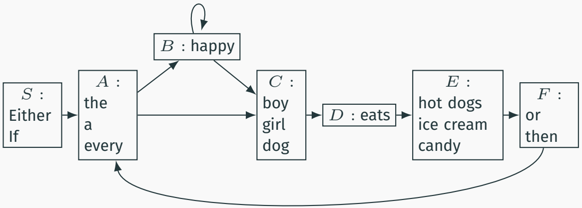
$C \to \text{dog } D$

$D \to \text{eats } E$

$E \to \text{hot dogs } F$

$E \to \text{ice cream } F$

$E \to \text{candy } F$



Solution: Context-Free Grammars

Context-Free Grammars have the ability to "call subroutines:"

$S \to$ Either P, or P. Exactly two Ps

$S \to$ If P, then P.

$P \to A\ H\ N$ eats O One each of A, H, N, and O

$A \to$ the

$A \to$ a

$A \to$ every

$H \to$ happy H H is "happy" zero or more times

$H \to \epsilon$

$N \to$ boy

$N \to$ girl

$N \to$ dog

$O \to$ hot dogs

$O \to$ ice cream

$O \to$ candy

An Example

n 0's followed by n 1's, e.g., 000111, 01

An Example

n 0's followed by n 1's, e.g., 000111, 01

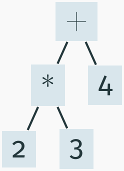
$$S \to 0\ S\ 1.$$
$$S \to \epsilon.$$

Constructing Grammars and Ocamlyacc

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

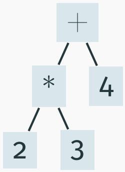
$$2 * 3 + 4 \qquad \Rightarrow$$



Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

$$2 * 3 + 4 \qquad \Rightarrow$$



Goal: verify the syntax of the program, discard irrelevant information, and “understand” the structure of the program.

Parentheses and most other forms of punctuation removed.

The Dangling Else Problem

Who owns the *else*?

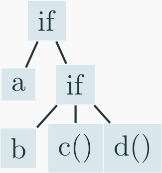
$$\text{if (a) if (b) c(); else d();}$$

```
stmt : IF expr THEN stmt
     | IF expr THEN stmt ELSE stmt
```

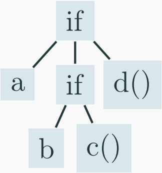
Problem comes after matching the first statement. Question is whether an “else” should be part of the current statement or a surrounding one since the second line tells us “stmt ELSE” is possible.

The Dangling Else Problem

Should this be



or



?

Grammars are usually ambiguous; manuals give disambiguating rules such as C’s:

> *As usual the “else” is resolved by connecting an else with the last encountered elseless if.*

The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling “then” (“dstmt”) and those that do not (“cstmt”). A statement may be either, but the statement just before an “else” must not have a dangling clause because if it did, the “else” would belong to it.

The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling "then" ("dstmt") and those that do not ("cstmt"). A statement may be either, but the statement just before an "else" must not have a dangling clause because if it did, the "else" would belong to it.

```
stmt  : dstmt
      | cstmt

dstmt : IF expr THEN stmt
      | IF expr THEN cstmt ELSE dstmt

cstmt : IF expr THEN cstmt ELSE cstmt
      | other statements...
```

The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling "then" ("dstmt") and those that do not ("cstmt"). A statement may be either, but the statement just before an "else" must not have a dangling clause because if it did, the "else" would belong to it.

```
stmt  : dstmt
      | cstmt

dstmt  : IF expr THEN stmt
       | IF expr THEN cstmt ELSE dstmt

cstmt  : IF expr THEN cstmt ELSE cstmt
       | other statements...
```

$$\text{if (a) if (b) c(); else d();}$$

The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling "then" ("dstmt") and those that do not ("cstmt"). A statement may be either, but the statement just before an "else" must not have a dangling clause because if it did, the "else" would belong to it.

```
stmt  : dstmt
      | cstmt

dstmt  : IF expr THEN stmt
       | IF expr THEN cstmt ELSE dstmt

cstmt  : IF expr THEN cstmt ELSE cstmt
       | other statements ...
```

$$\text{if (a) } \underbrace{\text{if (b) c();}}_{\text{cstmt?}} \text{ else d();}$$

Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

$$3 \text{ - } 4 \text{ * } 2 + 5$$

with the grammar

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

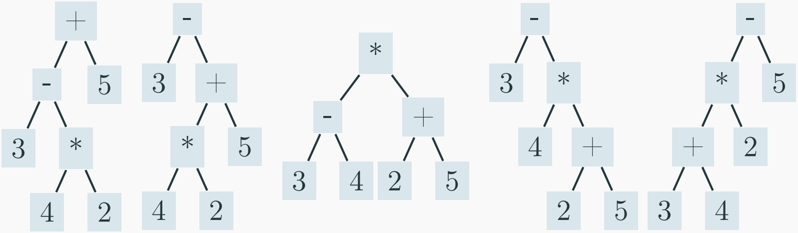
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

$$3 \text{ - } 4 * 2 + 5$$

with the grammar

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



Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

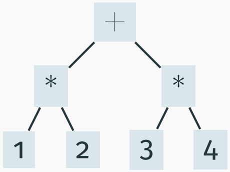
Operator Precedence

Defines how sticky an operator is.

$$1 * 2 + 3 * 4$$

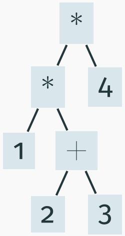
$*$ at higher precedence than $+$:

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



$+$ at higher precedence than $*$:

$1 * (2 + 3) * 4$

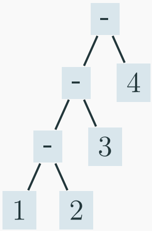


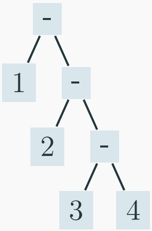
Associativity

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

Most operators are left-associative

1 - 2 - 3 - 4





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

left associative

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

right associative

Fixing Ambiguous Grammars

A grammar specification:

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

Ambiguous: no precedence or associativity.

Ocamlyacc's complaint: "16 shift/reduce conflicts."

$$1 * 2 + 3?$$

expr TIMES $\underline{\text{expr PLUS}}$ *shift*?

$\underline{\text{expr TIMES expr}}$ PLUS *reduce*?

Assigning Precedence Levels

Split into multiple rules, one per level

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

Assigning Precedence Levels

Split into multiple rules, one per level

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

$$1 * 2 + 3?$$

term TIMES term PLUS

Assigning Precedence Levels

Split into multiple rules, one per level

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

$$1 * 2 + 3?$$

term TIMES term PLUS *cannot shift*!

Assigning Precedence Levels

Split into multiple rules, one per level

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

$$1 * 2 + 3?$$

term TIMES term PLUS *cannot shift*!

term TIMES term PLUS

Assigning Precedence Levels

Split into multiple rules, one per level

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

$$1 * 2 + 3?$$

term TIMES term PLUS *cannot shift*!

term TIMES term PLUS *cannot reduce*!

Assigning Precedence Levels

Split into multiple rules, one per level

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

$$1 * 2 + 3?$$

term TIMES term PLUS *cannot shift*!

term TIMES term PLUS *cannot reduce*!

term TIMES term PLUS *reduce*!

Assigning Precedence Levels

Split into multiple rules, one per level

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

$$1 * 2 + 3?$$

term TIMES term PLUS *cannot shift*!

term TIMES term PLUS *cannot reduce*!

term TIMES term PLUS *reduce*!

Still ambiguous: associativity not defined

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

Assigning Precedence Levels

Split into multiple rules, one per level

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

Still ambiguous: associativity not defined

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

$$1 * 2 * 3?$$

term TIMES term TIMES *shift*?

term TIMES term TIMES *reduce*?

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.

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.

$$1 * 2 * 3?$$

term TIMES <u>atom TIMES</u>

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.

$$1 * 2 * 3?$$

term TIMES <u>atom TIMES</u> *cannot shift*!

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.

$$1 * 2 * 3?$$

term TIMES <u>atom TIMES</u> *cannot shift*!

term TIMES <u>atom</u> TIMES

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.

$$1 * 2 * 3?$$

term TIMES <u>atom TIMES</u> *cannot shift*!

term TIMES <u>atom</u> TIMES *cannot reduce*!

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.

$$1 * 2 * 3?$$

term TIMES <u>atom TIMES</u> *cannot shift*!

term TIMES <u>atom</u> TIMES *cannot reduce*!

<u>term TIMES atom</u> TIMES

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.

$$1 * 2 * 3?$$

term TIMES <u>atom TIMES</u> *cannot shift*!

term TIMES <u>atom</u> TIMES *cannot reduce*!

<u>term TIMES atom</u> TIMES *reduce*!

Parsing Algorithms

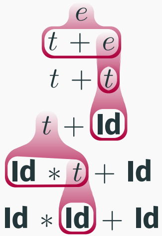
Shift/Reduce Parsing Using an Oracle

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



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

The Handle-Identifying Automaton

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

$\textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id} * t} \cdots$

$\textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id}} \cdots$

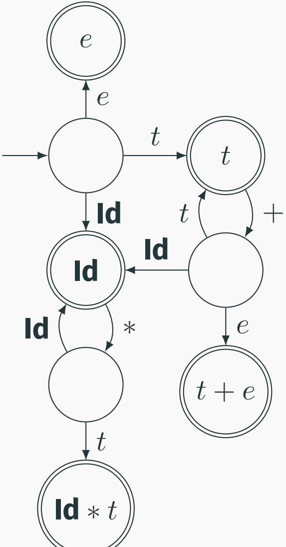
$t + t + \cdots + \underline{t + e}$

$t + t + \cdots + t + \underline{\textbf{Id}}$

$t + t + \cdots + t + \textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id} * t}$

$t + t + \cdots + \underline{t}$

e



Building the Initial State of the LR(0) Automaton

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

$$e' \rightarrow \bullet e$$

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 "$e' \rightarrow \bullet e$"

Building the Initial State of the LR(0) Automaton

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$e' \rightarrow \bullet e$
$e \rightarrow \bullet t + e$
$e \rightarrow \bullet t$

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 "$e' \rightarrow \bullet e$"

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

Building the Initial State of the LR(0) Automaton

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

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

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 "$e' \rightarrow \bullet e$"

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

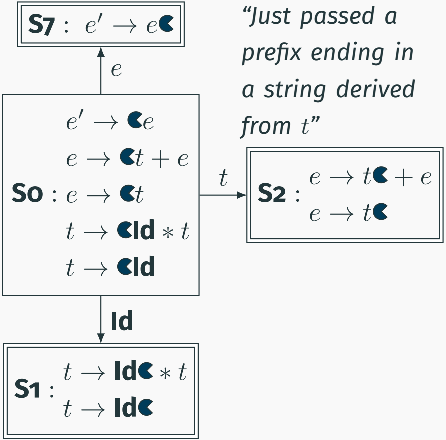
Also, t must be $\textbf{Id} * t$ or **Id**, so $t \rightarrow \bullet \textbf{Id} * t$ and $t \rightarrow \bullet \textbf{Id}$.

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

Building the LR(0) Automaton

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

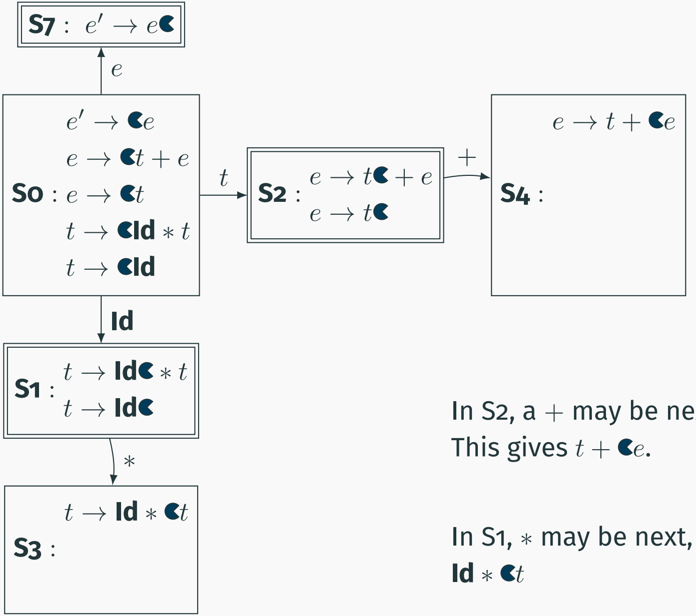
The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**.



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

The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**. The items for these three states come from advancing the $\bullet$ across each thing, then performing the closure operation (vacuous here).

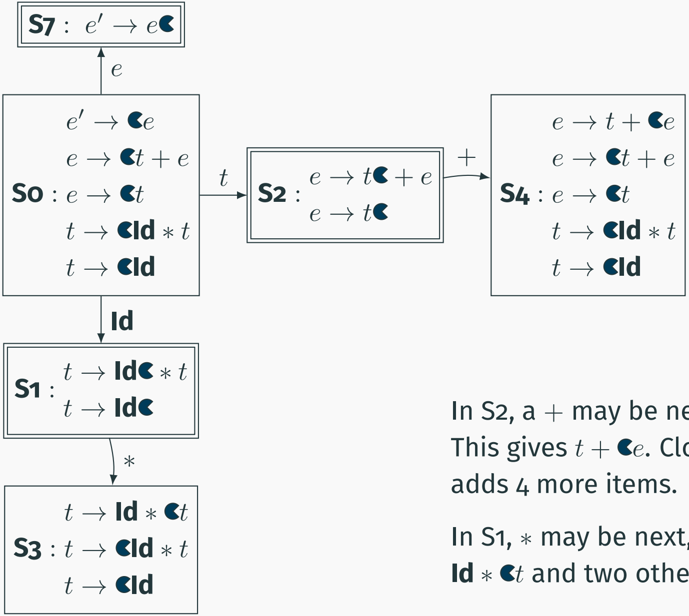
Building the LR(0) Automaton



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

In S1, $*$ may be next, giving
Id $* \bullet t$

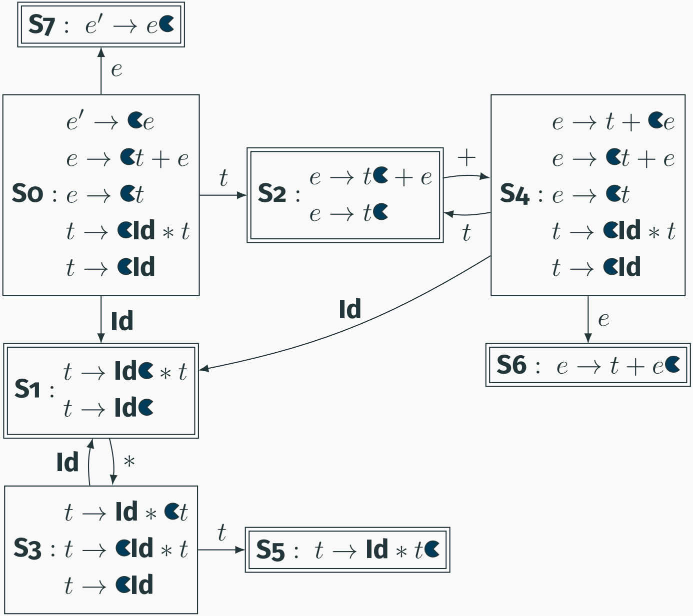
Building the LR(0) Automaton



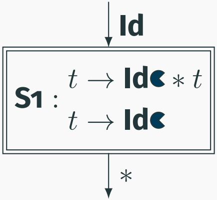
In S2, a $+$ may be next. This gives $t + \bullet e$. Closure adds 4 more items.

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

Building the LR(0) Automaton



What to do in each state?



$1 : e \to t + e$

$2 : e \to t$

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

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

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

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

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

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

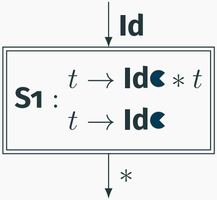
$t + t + \cdots + t + \mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t}$

$t + t + \cdots + \underline{t}$

e

Stack	Input	Action
$\mathbf{Id} * \mathbf{Id} * \cdots * \mathbf{Id}$	$* \cdots$	Shift

What to do in each state?



$1 : e \to t + e$

$2 : e \to t$

$3 : t \to \mathbf{Id} \; * t$

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

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

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

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

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

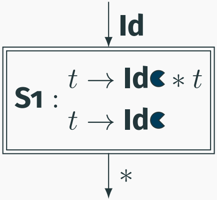
$t + t + \cdots + t + \mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t}$

$t + t + \cdots + \underline{t}$

e

Stack	Input	Action
$\mathbf{Id} * \mathbf{Id} * \cdots * \mathbf{Id}$	$* \cdots$	Shift
$\mathbf{Id} * \mathbf{Id} * \cdots * \mathbf{Id}$	$+ \cdots$	Reduce 4
$\mathbf{Id} * \mathbf{Id} * \cdots * \mathbf{Id}$		Reduce 4

What to do in each state?



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

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

Stack	Input	Action
$\textbf{Id} * \textbf{Id} * \cdots * \textbf{Id}$	$* \cdots$	Shift
$\textbf{Id} * \textbf{Id} * \cdots * \textbf{Id}$	$+ \cdots$	Reduce 4
$\textbf{Id} * \textbf{Id} * \cdots * \textbf{Id}$		Reduce 4
$\textbf{Id} * \textbf{Id} * \cdots * \textbf{Id}$	$\textbf{Id} \cdots$	Syntax Error

The FIRST function

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

1. If X is a terminal, $\text{FIRST}(X) = \{X\}$.
2. If $X \to \epsilon$, then add ϵ to $\text{FIRST}(X)$.
3. If $X \to Y_1 \cdots Y_k$ and $\epsilon \in \text{FIRST}(Y_1)$, $\epsilon \in \text{FIRST}(Y_2)$, …, 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 $\text{FIRST}(Y_1) \subset \text{FIRST}(X)$
4. If $X \to Y_1 \cdots Y_K$ and $\epsilon \in \text{FIRST}(Y_1)$, $\epsilon \in \text{FIRST}(Y_2)$, …, and $\epsilon \in \text{FIRST}(Y_k)$, add ϵ to $\text{FIRST}(X)$
   If all of X can be empty, X can be empty

The FIRST function

If you can derive a string that starts with terminal t from a sequence of terminals and nonterminals α, then $t \in$ FIRST(α).

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

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

FIRST(**Id**) $= \{$**Id**$\}$

FIRST$(t) = \{$**Id**$\}$ because $t \rightarrow$ **Id** $* \, t$ and $t \rightarrow$ **Id**

FIRST$(e) = \{$**Id**$\}$ because $e \rightarrow t + e$, $e \rightarrow t$, and FIRST$(t) = \{$**Id**$\}$.

The FOLLOW function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in$ 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 $\epsilon \in$ FIRST(α), 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

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

FIRST$(t) = \{$**Id**$\}$

FIRST$(e) = \{$**Id**$\}$

FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{ \quad \}$

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 $t \in$ 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 $\epsilon \in$ FIRST(α), 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

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

FIRST$(t) = \{$**Id**$\}$

FIRST$(e) = \{$**Id**$\}$

FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{ + \quad \}$

2. *Because* $e \rightarrow \underline{t} + e$ *and* FIRST$(+) = \{+\}$

The FOLLOW function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in$ 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 $\epsilon \in$ FIRST(α), 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

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

FIRST$(t) = \{$**Id**$\}$

FIRST$(e) = \{$**Id**$\}$

FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{ + , \$\}$

3. Because $e \rightarrow \underline{t}$ and $\$ \in$ FOLLOW(e)

The FOLLOW function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in$ 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 $\epsilon \in$ FIRST(α), 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

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

FIRST$(t) = \{$**Id**$\}$

FIRST$(e) = \{$**Id**$\}$

FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{ + , \$\}$

Fixed-point reached: applying any rule does not change any set

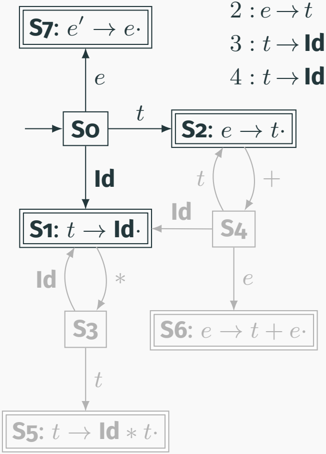
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



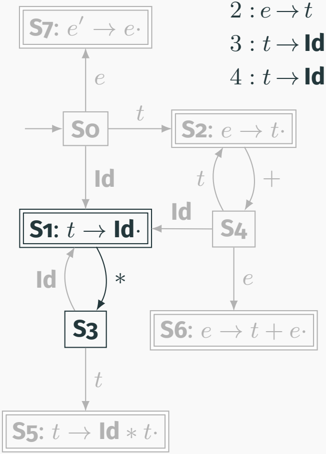
FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{+, \$\}$

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

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

Converting the LR(0) Automaton to an SLR Table



$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{+, \$\}$

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

From S1, shift a $*$ and go to S3; or, if the next input $\in$ FOLLOW(t), reduce by rule 4.

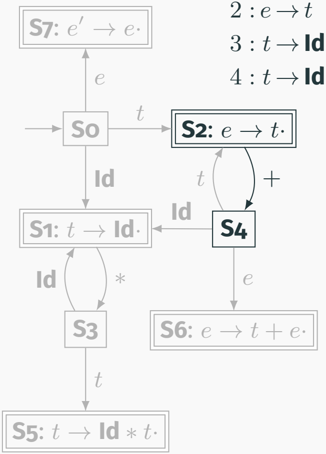
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



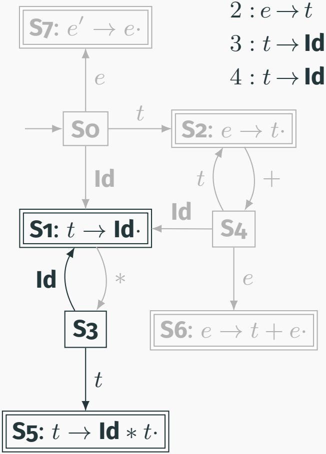
FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{+, \$\}$

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

From S2, shift a $+$ and go to S4; or, if the next input $\in$ FOLLOW(e), reduce by rule 2.

Converting the LR(0) Automaton to an SLR Table



$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{+, \$\}$

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

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

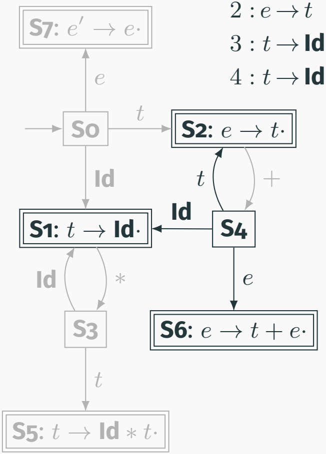
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{+, \$\}$

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

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

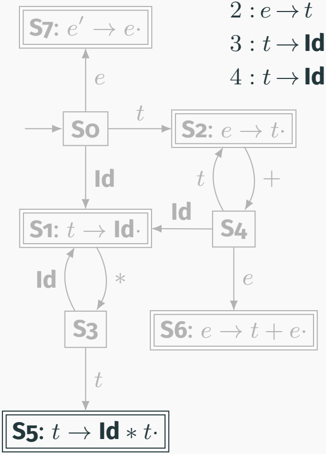
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



FOLLOW$(e) = \{\$\}$

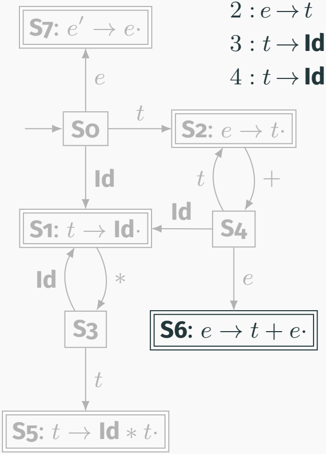
FOLLOW$(t) = \{+, \$\}$

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

From S5, reduce using rule 3 if the next symbol $\in$ FOLLOW(t).

Converting the LR(0) Automaton to an SLR Table

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



FOLLOW$(e) = \{\$\}$
FOLLOW$(t) = \{+, \$\}$

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

From S6, reduce using rule 1 if the next symbol $\in$ FOLLOW(e).

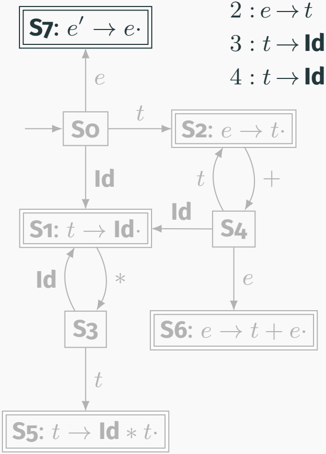
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



FOLLOW$(e) = \{\$\}$

FOLLOW$(t) = \{+, \$\}$

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

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

Shift/Reduce Parsing with an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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

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

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

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

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

Shift/Reduce Parsing with an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

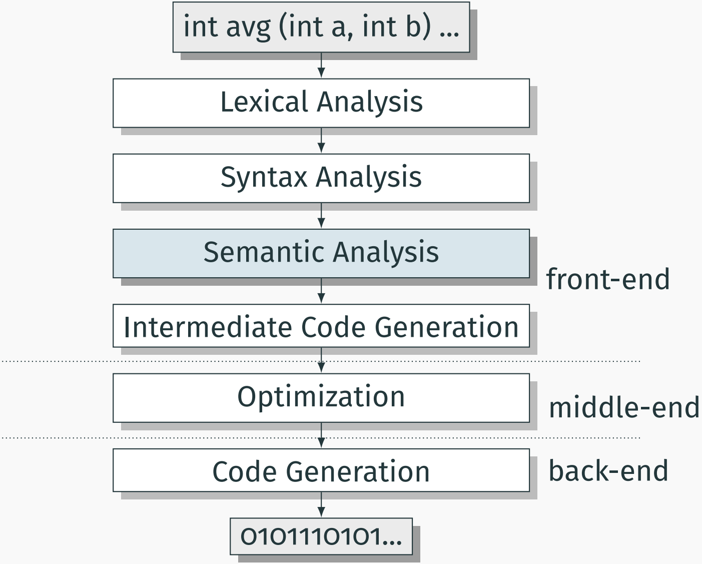
$4 : t \rightarrow \textbf{Id}$

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

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

Semantic Analysis

Semantic Analysis



Static Semantic Analysis

Lexical analysis: Each token is valid?

```
for #a1123                      /* invalid tokens */
for break                       /* valid Java tokens */
```

Syntactic analysis: Tokens appear in the correct order?

```
for break          /* invalid syntax */
return 3 + "f";    /* valid Java syntax */
```

Semantic analysis: Names used correctly? Types consistent?

```
return 3 + "f";     /* invalid */
return 3 + 13;      /* valid in Java */
```

What's Wrong With This?

$$a + f(b, c)$$

Scope questions:

Is a defined?

Is f defined?

Are b and c defined?

Type questions:

Is f a function of two arguments?

Can you add whatever a is to whatever f returns?

Does f accept whatever b and c are?

Scope - What names are visible?

Scope

Scope: where/when a name is bound to an object

Useful for modularity: want to keep most things hidden

Scoping Policy	Visible Names Depend On
Static	Textual structure of program Names resolved by compile-time symbol tables Faster, more common, harder to break programs
Dynamic	Run-time behavior of program Names resolved by run-time symbol tables, e.g., walk the stack looking for names Slower, more dynamic

Static vs. Dynamic Scope

C

```c
int a = 0;

int foo() {
  return a;
}

int bar() {
  int a = 10;

  return foo();
}
```

OCaml

```ocaml
let a = 0 in
let foo x = a in
let bar =
  let a = 10 in
  foo 0
```

Bash

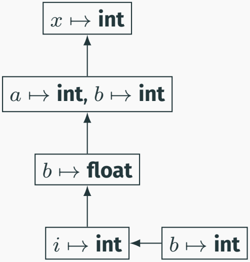
```bash
a=0

foo ()
{
  echo $a
}

bar ()
{
  local a=10
  foo
}

bar
echo $a
```

Symbol Tables by Example: C-style

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous
- Reach an identifier: lookup in chain of tables

```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```



Types - What operations are allowed?

Types

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

Two uses:

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

Optimization: eliminates certain runtime decisions

Type Systems

- A language's type system specifies which operations are valid for which types.
- The goal of type checking is to ensure that operations are used with the correct types.
- Three kinds of languages:
  - Statically typed: All or almost all checking of types is done as part of compilation (C, Java)
  - Dynamically typed: Almost all checking of types is done as part of program execution (Python)
  - Untyped: No type checking (machine code)

Strongly-typed Languages

Strongly-typed: the type of a value does not change in unexpected ways.

Is C strongly-typed?

```
float g;
union { float f; int i } u;
u.i = 3;
g = u.f + 3.14159; /* u.f is meaningless */
```

Is Java strongly-typed?

What about Python?

Solution: Type Environment

Put more information in the rules!

A type environment gives types for free variables .

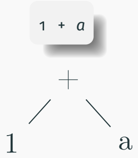
$$\frac{}{\mathcal{E} \vdash \text{NUMBER} : \ \textbf{int}}$$

$$\frac{\mathcal{E}(x) = \textbf{T}}{\mathcal{E} \vdash x : \ \textbf{T}}$$

$$\frac{\mathcal{E} \vdash \text{expr}_1 : \ \textbf{int} \qquad \mathcal{E} \vdash \text{expr}_2 : \ \textbf{int}}{\mathcal{E} \vdash \text{expr}_1 + \text{expr}_2 : \ \textbf{int}}$$

How To Check Symbols

check: environment $\rightarrow$ node $\rightarrow$ typedNode



check(+, E)

check(1, E) = 1 : int

check(a, E) = a : E.lookup(a) = a : int

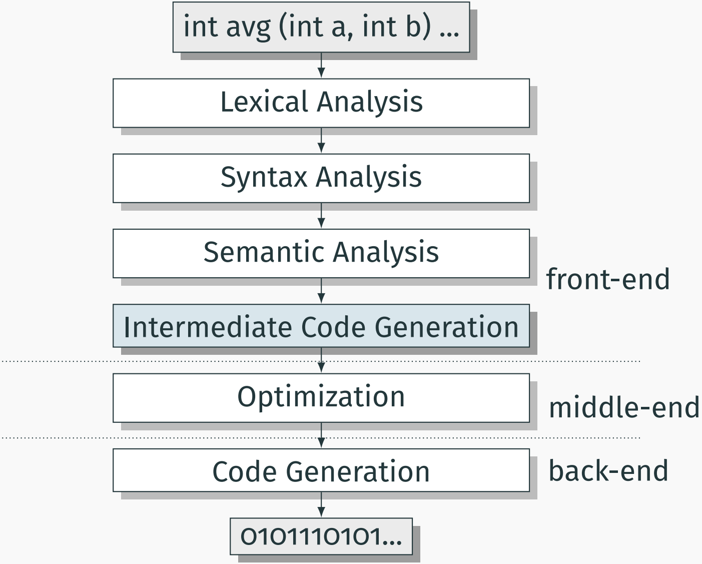
int + int = int

= 1 $+$ a : int

The environment provides a “symbol table” that holds information about each in-scope symbol.

IR Generation

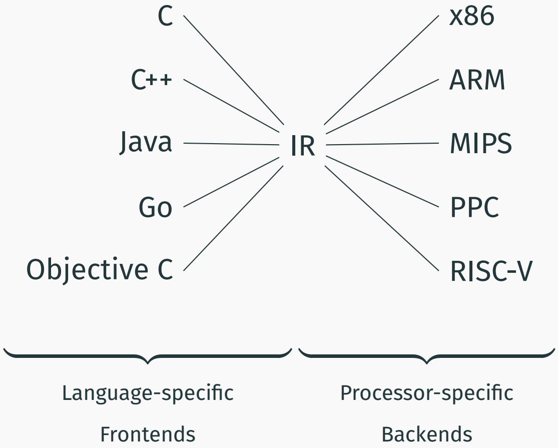
Intermediate Code Generation



Intermediate Representation

Suppose we wish to build compilers for n source languages and m target machines.

Case 2: IR present. Need just n front-ends and m back ends.



Three-Address Code & Static Single Assignment

Most register-based IRs use three-address code:
Arithmetic instructions have (up to) three operands: two sources and one destination.

SSA Form: each variable in an IR is assigned exactly once

C code:

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

Three-Address:

```
WHILE:  t = sne a, b
        bz DONE, t
        t = slt a, b
        bz ELSE, t
        b = sub b, a
        jmp LOOP
ELSE:   a = sub a, b
LOOP:   jmp WHILE
DONE:   ret a
```

SSA:

```
WHILE:  t1 = sne a1, b1
        bz DONE, t1
        t2 = slt a1, b1
        bz ELSE, t2
        b1 = sub b1, a1
        jmp LOOP
ELSE:   a1 = sub a1, b1
LOOP:   jmp WHILE
DONE:   ret a1
```

Address

What is an "Address" in Three-Address Code?

- Name: (from the source program) e.g., x, y, z
- Constant: (with explicit primitive type) e.g., 1, 2, 'a'
- Compiler-generated temporary: ("register") e.g., t1, t2, t3

Instructions of Three-Address Code

- x = op y, z: where op is a binary operation
- x = op y: where op is a unary operation
- x = y: copy operation
- jmp L: unconditional jump to label L
- bz L, x: jump to L if x is zero
- bnz L, x: jump to L if x is not zero
- param x, call L, y, return z: function calls

Three-Address Code (TAC) Generation

Goal: take statements (AST) and produce a sequence of TAC.

Example:

a := b + c * d;

TAC:

```
t1 = mul c, d
t2 = add b, t1
a = t1
```

Translate expressions and statements

Algorithm: Syntax-Directed Translation (SDT)

For each expression E, we'll synthesize two attributes:

- E.addr: the name of the variable (often a temporary variable)
- E.code: the IR instructions generated from E

SDT: each semantic rule corresponds to actions computing two attributes with the following auxiliary functions:

- Call NewTemp to create a new temporary variable
- Call Gen: to print a new three-address instruction

  Gen(t, "=", op, x, ",", y) $\Rightarrow$ "t = op x, y"

Syntax-Directed Translation (SDT)

CFG rule: $E_0 \to$ **id**

Actions:

E_0.addr := **id**

E_0.code := “” empty string

We do not consider scopes here.

Example: E_0 = ID(“a”)

E_0.addr := “a”

E_0.code := “” empty string

Syntax-Directed Translation (SDT)

CFG rule: $E_0 \to E_1 + E_2$

Actions:

E_0.addr := NewTemp()

E_0.code :=E_1.code || E_2.code ||

Gen(E_0.addr, "=", "add", E_1.addr, ",", E_2.addr)

Example: a + b

E_0 = PLUS (E_1, E_2) E_1 = ID("a") E_2 = ID("b")

E_1.addr := "a" E_1.code := ""

E_2.addr := "b" E_2.code := ""

E_0.addr := "t1"

Translating Statements

Assignment

CFG rule: $S \rightarrow$ **id** $:= E$

Actions:

S.code :=E.code || Gen(**id**, “=”, E.addr)

Example: a := b + c

S = ASG (ID(“a”), E) E =PLUS(ID(“b”), ID(“c”))

E.code := “t1 = add b, c” E.addr := “t1’

S.code := “t1 = add b, c” || “a = t1”

IF Statement

AST: IF(E, S)

Generated IR:

```
        E.code
        bz Label_End, E.addr
        S.code
    Label_End:
```

Example: if (a > b) { a -= b }

```
        t1 = slt a, b
        bz Label_End, t1
        a = sub a, b
    Label_End:
```

IF-ELSE Statement

AST: IFELSE(E, S_1, S_2)

Generated IR:

E.code
bz Label_Else, E.addr
S_1.code
jmp Label_End
Label_Else:
S_2.code
Label_End:

Loop

AST: WHILE(E, S)

Generated IR:

Label_While:

E.code

bz Label_End, E.addr

S.code

jmp Label_While

Label_End:

Function Calls

$f\ (E_1, \cdots, E_n)$

Generated IR:

E_n.code
E_{n-1}.code
$\ldots$
E_1.code
param E_n.addr
$\ldots$
param E_1.addr
call f, n

Basic Blocks

A Basic Block is a sequence of IR instructions with two properties:

1. The first instruction is the only entry point (no other branches in; can only start at the beginning)
2. Only the last instruction may affect control (no other branches out)

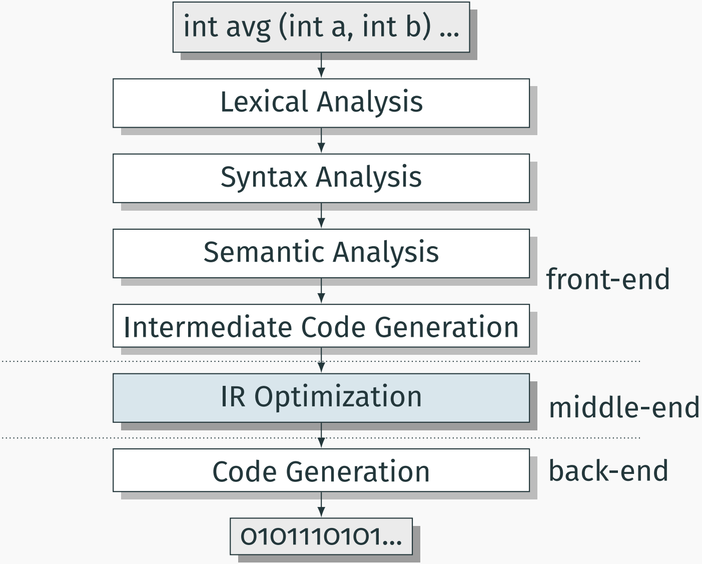
∴ If any instruction in a basic block runs, they all do

Typically "arithmetic and memory instructions, then branch"

```
ENTER:  t2 = add t1, 1
        t3 = slt t2, 10
        bz NEXT, t3
```

IR Optimization

IR Optimization



IR Optimization Discussion

Optimal? Undecidable!

Soundness: semantics-preserving

IR optimization v.s. code optimization:

`x * 0.5` ⇒ `x » 1`

Local optimization v.s. global optimization

Local Optimization

Common Subexpression Elimination

Purpose: remove the duplicate computation of "`a op b`" in Three-Address code.

```
v1 = a op b

.   .   .

v2 = a op b
```

If values of **`v1`**, **`a`**, and **`b`** have not changed, rewrite the code:

```
v1 = a op b

.   .   .

v2 = v1
```

Copy Propagation

If we have

```
v1 = v2
```

then as long as **`v1`** and **`v2`** have not changed, we can rewrite

```
a = ...  v1 ...
```

as

```
a = ...  v2 ...
```

Dead Code Elimination

An assignment to a variable **v** is called dead if its value is never read anywhere.

Implementing Local Optimization

Optimizations and Analyses

Most optimizations are only possible given some analysis of the program's behavior.

In order to implement an optimization, we will talk about the corresponding program analyses.

Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.
- An expression is called available if some variable in the program holds the value of that expression.
- In common subexpression elimination, we replace an available expression requiring computation by the variable holding its value.
- In copy propagation, we replace the use of a variable by the available expression it holds that does not require computation.

Finding Available Expressions

- Initially, no expressions are available
- Whenever we execute a statement

  `a = expr`

  - Any expression holding **`a`** is invalidated.
  - The expression **`a = expr`** becomes available.
- Algorithm: Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable.

Example: Available Expressions

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = a + b;

{ a = b, c = b, d = a + b, e = a + b }

d = b;

{ a = b, c = b, d = b, e = a + b }

f = a + b;

{ a = b, c = b, d = b, e = a + b, f = a + b }

Live Variables

- The analysis corresponding to dead code elimination is called liveness analysis.
- A variable is live at a point in a program if later in the program its value will be read before it is written to again.
- Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables.

Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a block in reverse order.
- Initially, some small set of values are known to be live (which ones depends on the particular program).
- When we see the statement: **`a = b op c`**
  - If **a** is not alive after the statement, skip it.
  - Otherwise, If **a** is alive after the statement
    - Just before the statement, **a** is not alive, since its value is about to be overwritten.
    - Just before the statement, both **b** and **c** are alive, since we're about to read their values.
  - (what if we have **`a = a op b`**?)

Example: Liveness Analysis

```
a = b;

c = a;

d = b + d;

e = d;

d = b;

f = e + c;
{ d, e }
```

Example: Dead Code Elimination

{ b, d }

a = b;

{ b, d }

c = a;

{ b, d }

d = b + d;

{ b, d }

e = d;

{ b, e }

d = b;

{ d, e }

f = e + c;

{ d, e }

Global Optimization

Global Constant Propagation

Replace each variable that is known to be a constant value with the constant.

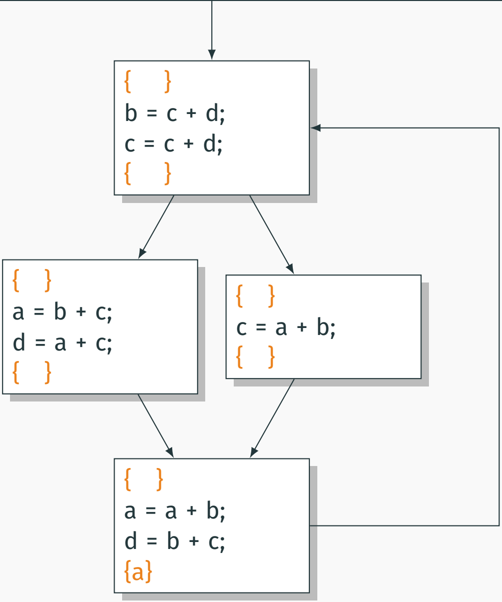
Global Dead Code Elimination

- Local dead code elimination needed to know what variables were live on exit from a basic block.
- This information can only be computed as part of a global analysis.
- How do we modify our liveness analysis to handle a CFG?

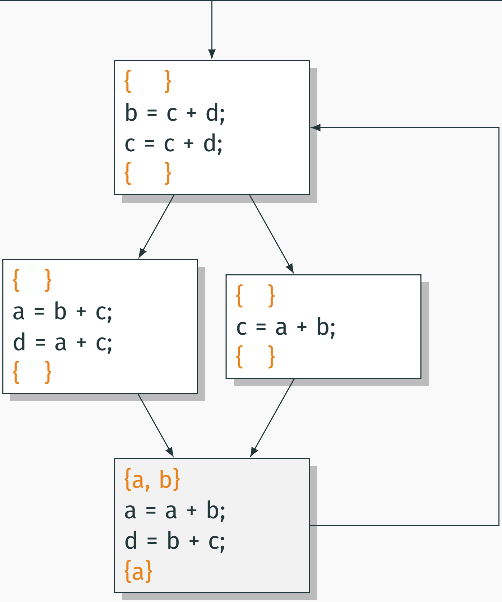
Global Dead Code Elimination

- In a local analysis, each statement has exactly one predecessor.
- In a global analysis, each statement may have multiple predecessors.
- A global analysis must combine information from all predecessors of a basic block.

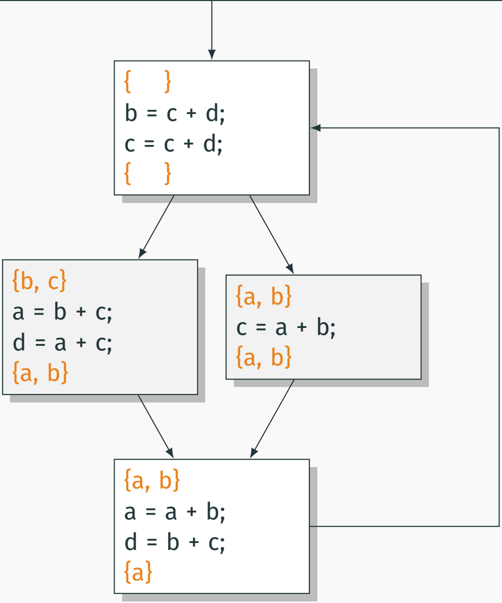
Global Dead Code Elimination with Loops



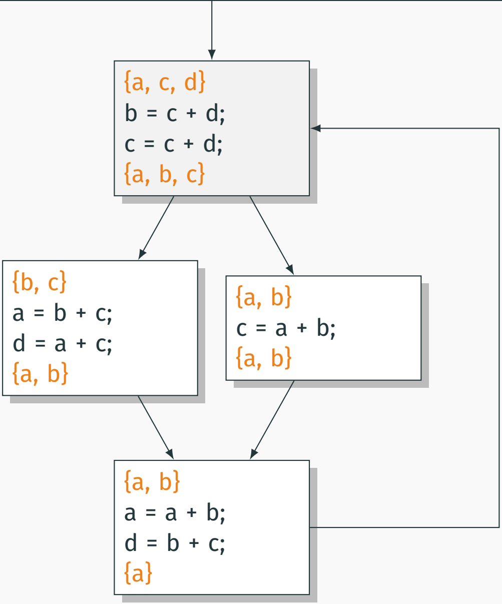
Global Dead Code Elimination with Loops



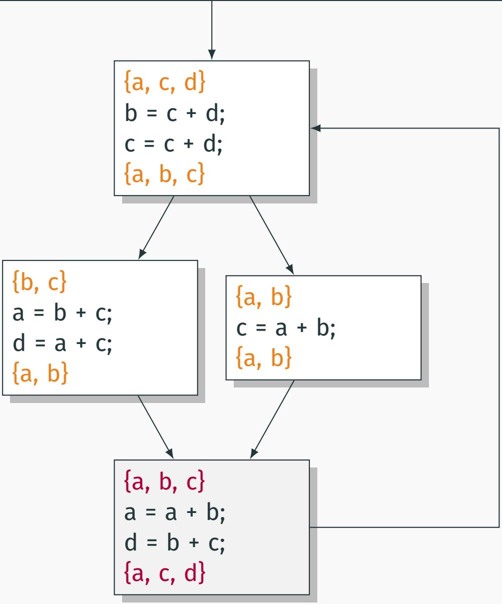
Global Dead Code Elimination with Loops



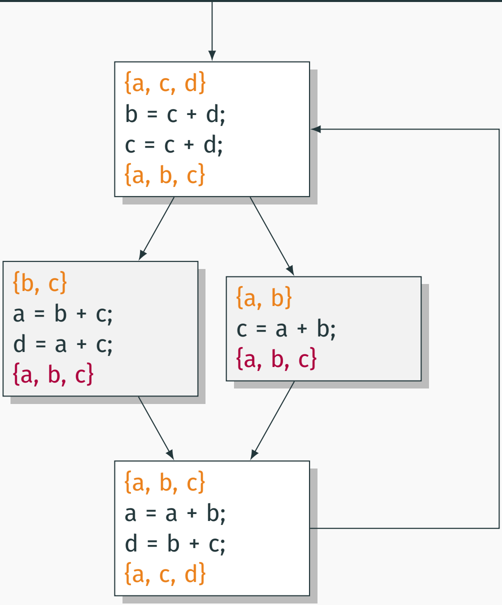
Global Dead Code Elimination with Loops



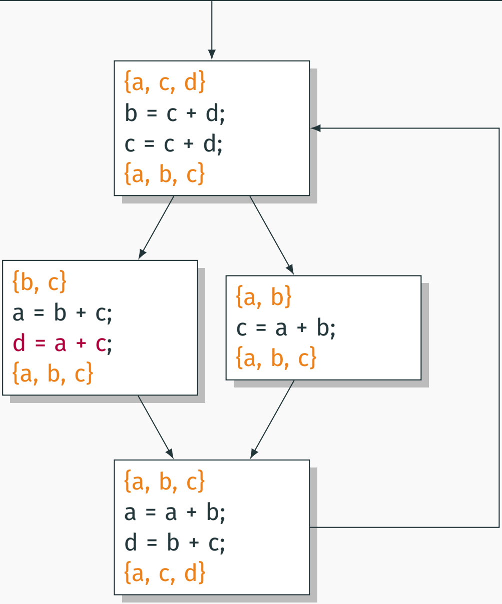
Global Dead Code Elimination with Loops



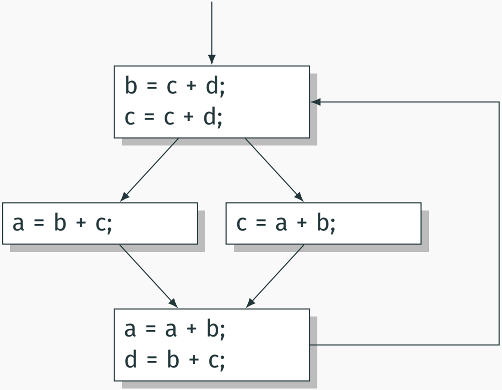
Global Dead Code Elimination with Loops



Global Dead Code Elimination with Loops

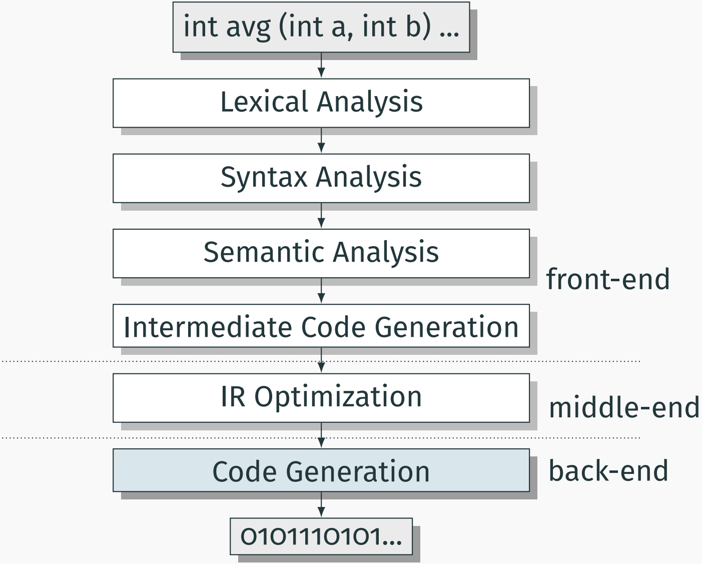


Global Dead Code Elimination with Loops



Code Generation

Code Generation



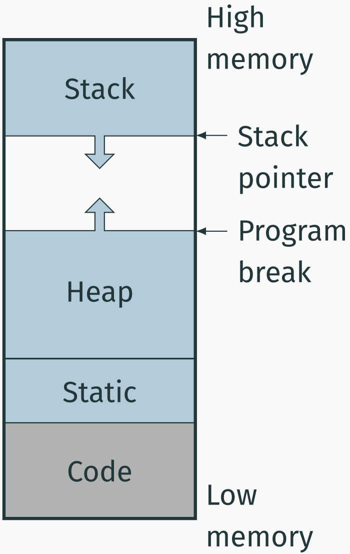
Runtime Environments

Storage Classes and Memory Layout

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

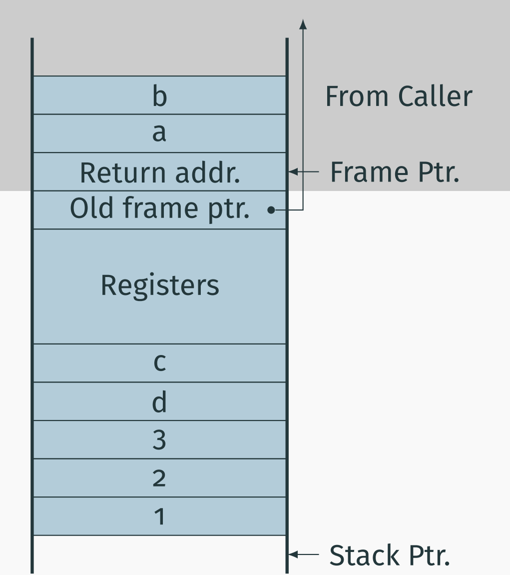
Heap: objects created/destroyed in any order; automatic garbage collection optional

Static: objects allocated at compile time; persist throughout run



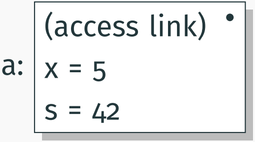
An Activation Record: The State Before Calling *bar*

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



Implementing Nested Functions with Access Links

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

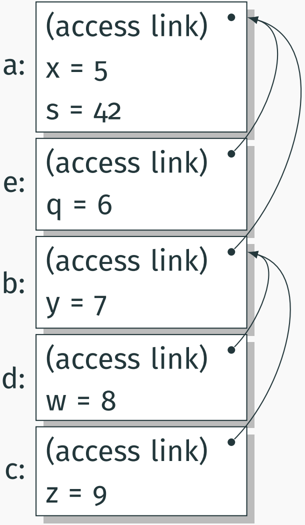


What does “a 5 42” give?

Implementing Nested Functions with Access Links

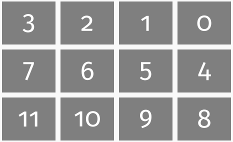
```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
   d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

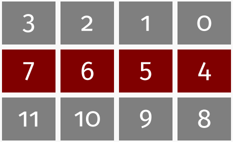


Layout of Records and Unions

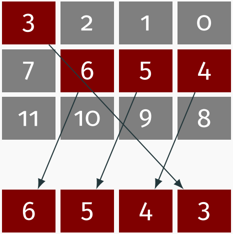
Modern memory systems read data in 32-, 64-, or 128-bit chunks:



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



How about reading an unaligned value?

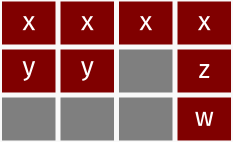


Padding

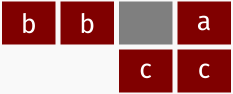
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each n-byte-aligned object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte-aligned 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 */
};
```

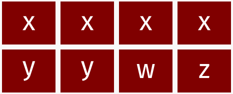


Padding

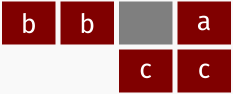
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each n-byte-aligned object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte-aligned 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  */
  char w;   /* 1 byte  */
  short y;  /* 2 bytes */
};
```

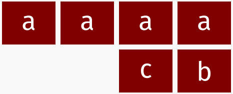



```c
struct padded {
  char a;   /* 1 byte  */
  short b;  /* 2 bytes */
  short c;  /* 2 bytes */
};
```

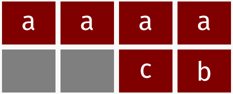


Padding: (1) or (2)?

```c
struct padded {
  int a;  /* 4 bytes  */
  char b; /* 1 byte */
  char c; /* 1 byte */
};
```



(1)

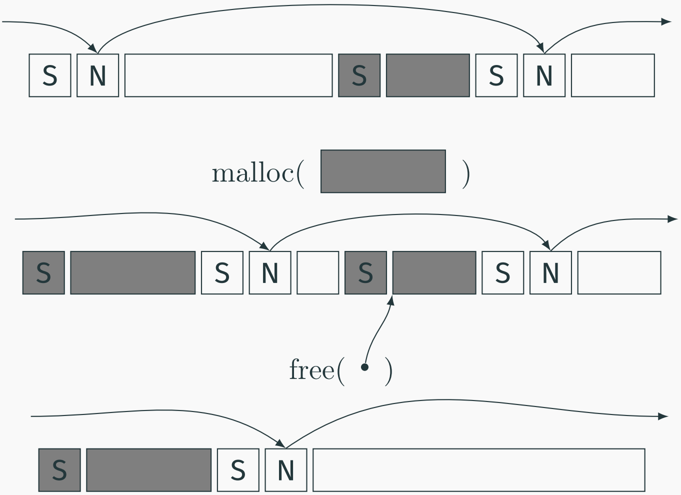


(2)

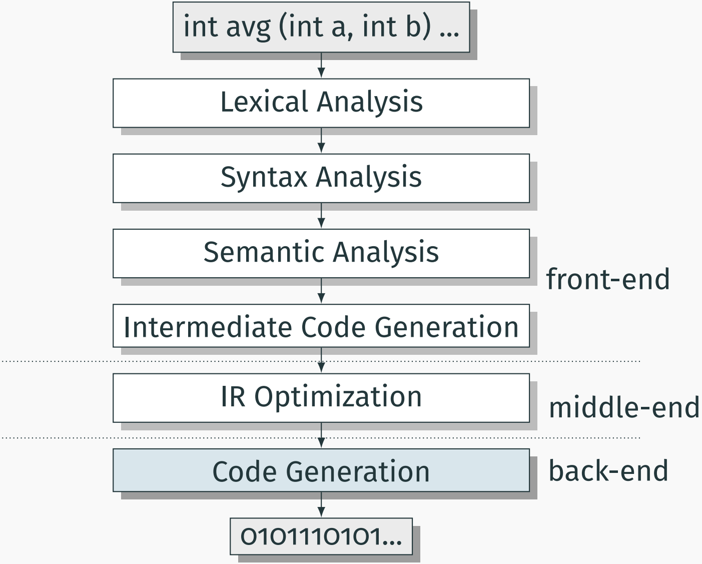
Heap-Allocated Storage

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

Simple Dynamic Storage Allocation



Code Generation



A Better Allocator

Goal: try to hold as many variables in registers as possible.

Register consistency:

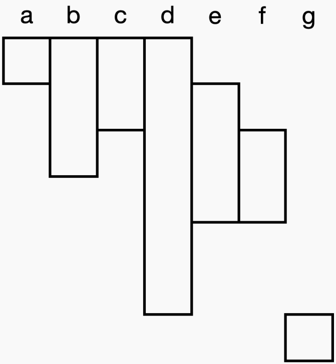
- At each program point, each variable must be in the same location.
- At each program point, each register holds at most one live variable.

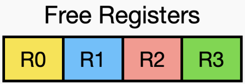
Register Allocation

Explore three algorithms for register allocation:

- Naive (“no”) register allocation.
- Linear scan register allocation.
- Graph-coloring register allocation.

Linear Scan

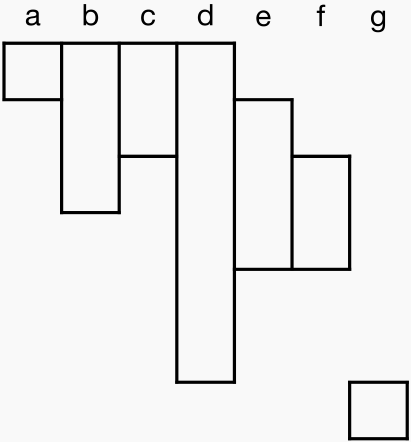




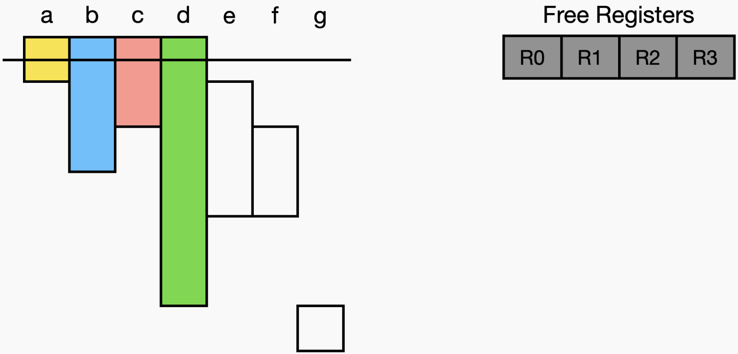
Live Intervals

Live interval: the smallest subrange of the IR code containing all a variable's live ranges.

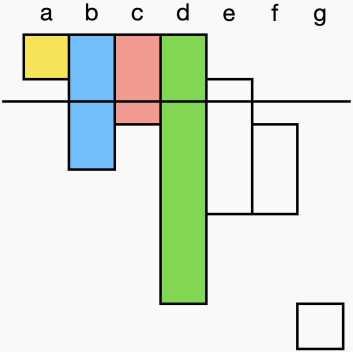
```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```

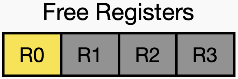


Linear Scan

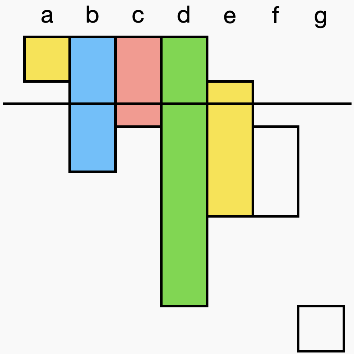


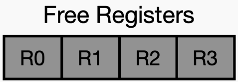
Linear Scan





Linear Scan





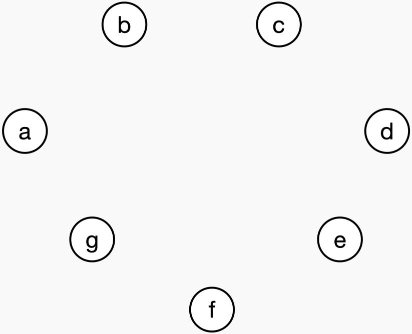
Graph-coloring Register Allocation

The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```

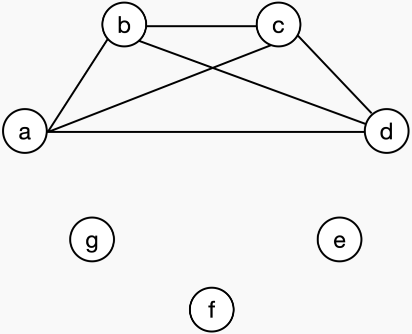
The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```



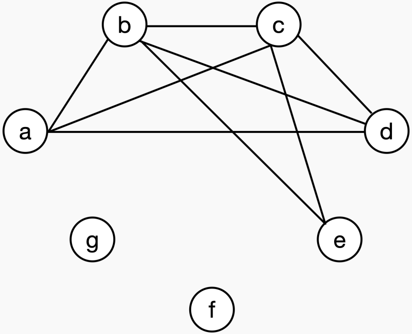
The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```



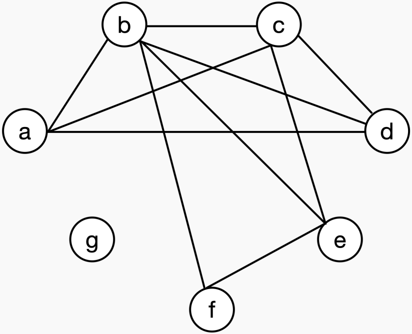
The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
  { e, b, c }
  f = b + c;
  { e, f, b }
  f = f + b;
   { e, f }
  d = e + f;
    { d }
   g = d;
    { g }
```



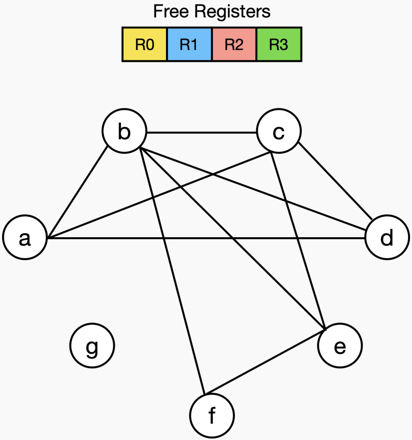
The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```



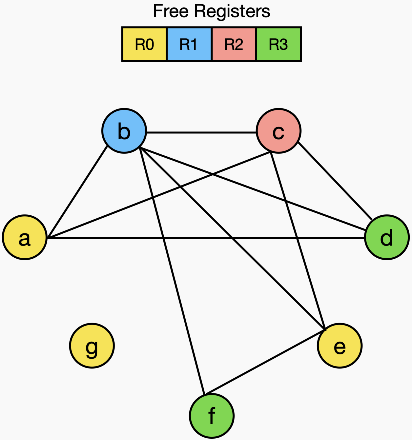
The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
   { e, f }
  d = e + f;
    { d }
   g = d;
    { g }
```



The Register Interference Graph (RIG)

```
{ d, b, c, a }
e = d + a;
{ e, b, c }
f = b + c;
{ e, f, b }
f = f + b;
{ e, f }
d = e + f;
{ d }
g = d;
{ g }
```



The Register Interference Graph

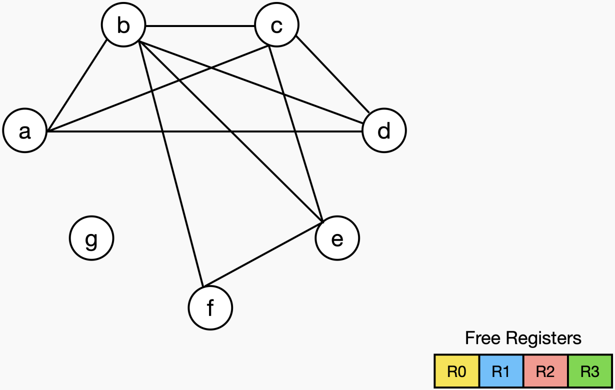
The register interference graph (RIG) of a control-flow graph is an undirected graph where

- Each node is a variable
- There is an edge between two variables that are live at the same point

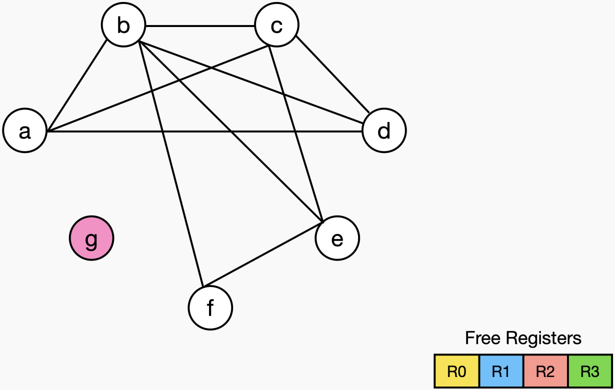
Perform register allocation by assigning each variable a different register from all of its neighbors.

This problem is equivalent to graph-coloring.

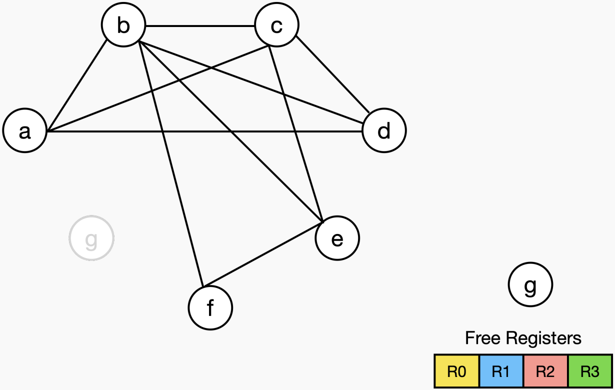
Chaitin's Algorithm



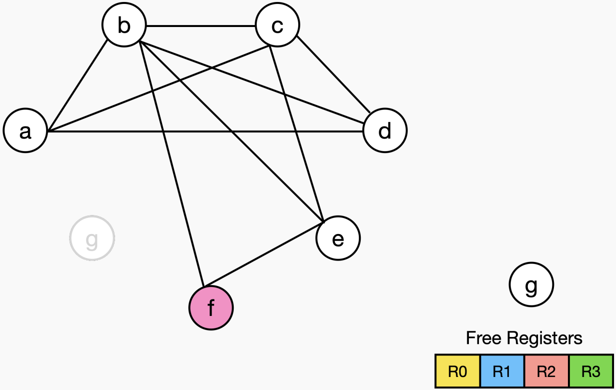
Chaitin's Algorithm



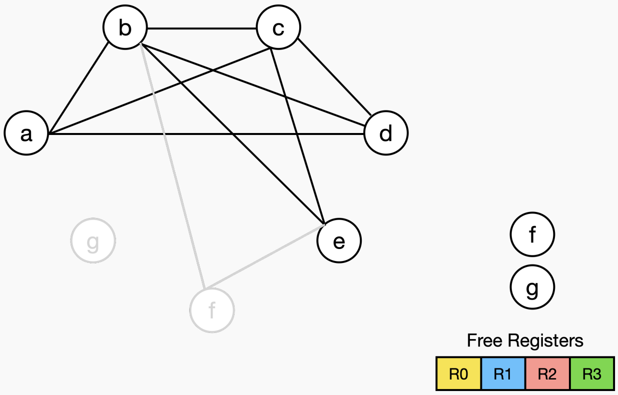
Chaitin's Algorithm



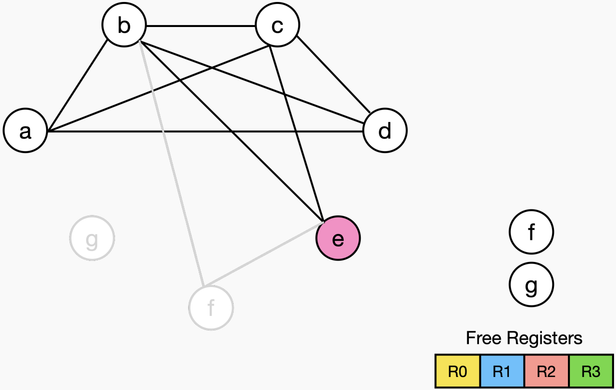
Chaitin's Algorithm



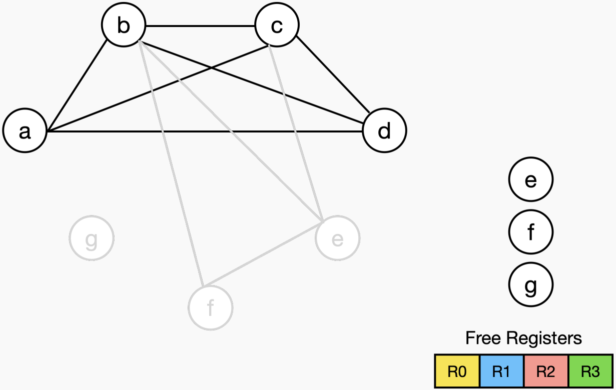
Chaitin's Algorithm



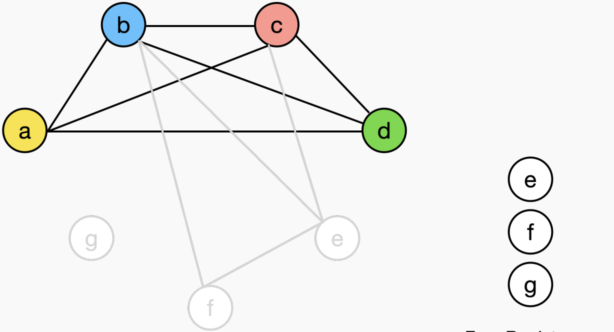
Chaitin's Algorithm



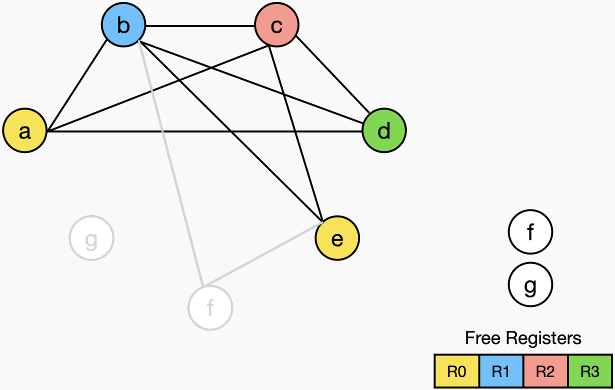
Chaitin's Algorithm



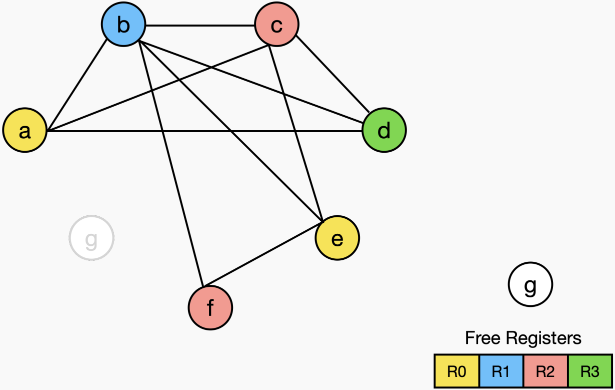
Chaitin's Algorithm



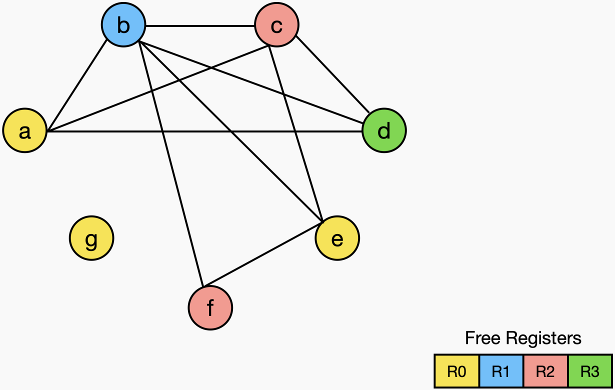
Chaitin's Algorithm



Chaitin's Algorithm



Chaitin’s Algorithm



Code Generation