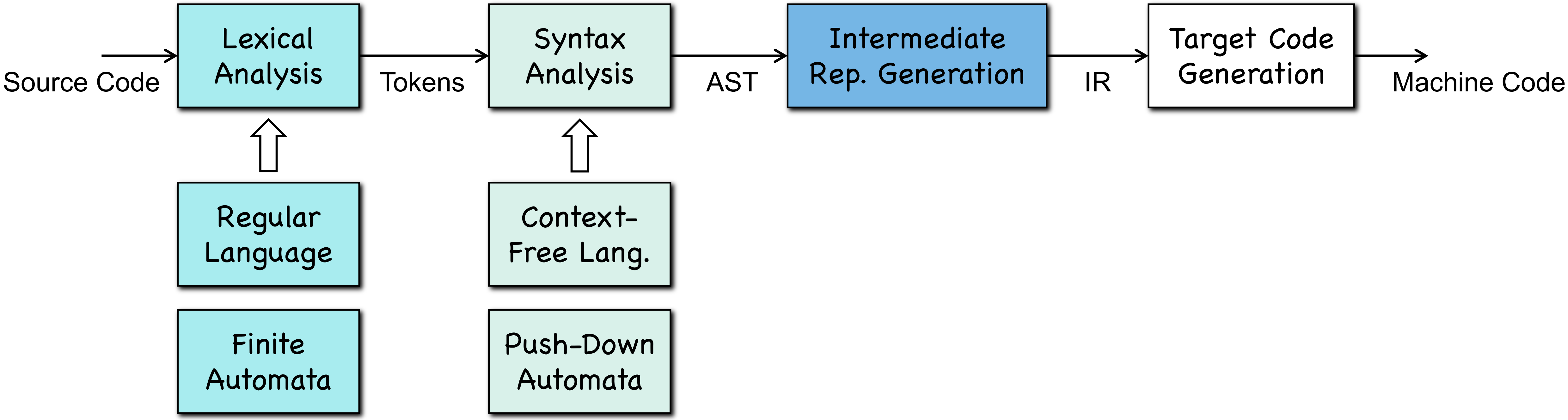
Recap-1
The Compilers’ Front End

Front End of Compilers



PART I: Regex → NFA → (Min) DFA

Regular Expression (Regex)

- Regex describes a language

Regular Expression (Regex)

- Regex describes a language
- Primitive regex
    - $\emptyset, \epsilon, a$

Regular Expression (Regex)

- Regex describes a language
- Primitive regex
  - $\emptyset, \epsilon, a$
- Given two regex: r_1, r_2, the following are regex
  - $r_1|r_2$
  - r_1r_2
  - r_1^*
  - (r_1)

Regular Expression (Regex)

- Regex describes a language
- Primitive regex
  - $\emptyset, \epsilon, a$
- Given two regex: r_1, r_2, the following are regex
  - $r_1 | r_2$
  - $r_1 r_2$
  - r_1^*
  - (r_1)
- **Example:** $(a|b)^* c$

Language by Regex

- The language represented by regex are defined below
- Primitive regex
  - $L(\emptyset) = \emptyset$; $L(\epsilon) = \{\epsilon\}$; $L(a) = \{a\}$

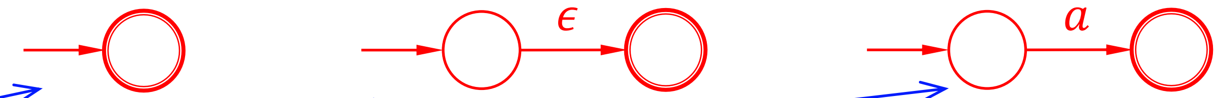
Language by Regex

- The language represented by regex are defined below
- Primitive regex
  - $L(\emptyset) = \emptyset;\ \ L(\epsilon) = \{\epsilon\};\ \ L(a) = \{a\}$
- Given two regex: r_1, r_2, the following are regex
  - $L(r_1|r_2) = L(r_1) \cup L(r_2)$
  - $L(r_1 r_2) = L(r_1)L(r_2)$
  - $L(r_1^*) = (L(r_1))^*$
  - $L((r_1)) = L(r_1)$

Regex to NFA (DFA)

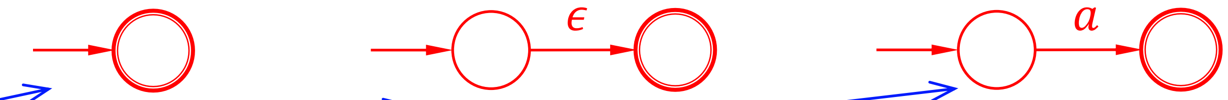
- Primitive regex
  - $L(\emptyset) = \emptyset$; $L(\epsilon) = \{\epsilon\}$; $L(a) = \{a\}$

Regex to NFA (DFA)



- Primitive regex
  - $L(\emptyset) = \emptyset$; $L(\epsilon) = \{\epsilon\}$; $L(a) = \{a\}$

Regex to NFA (DFA)



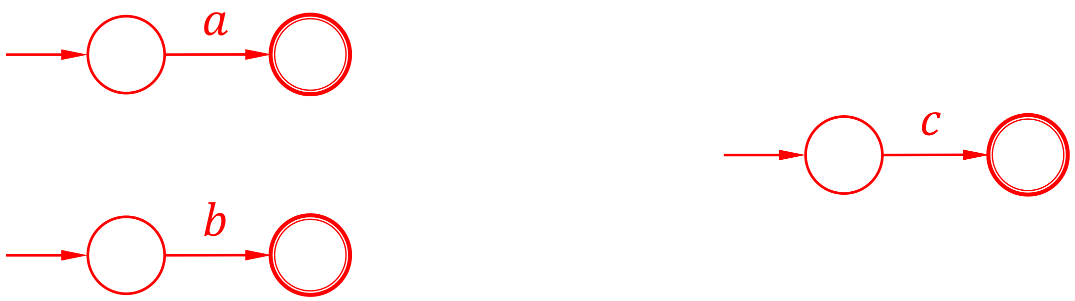
- Primitive regex
  - $L(\emptyset) = \emptyset$; $L(\epsilon) = \{\epsilon\}$; $L(a) = \{a\}$
- Given two regex: r_1, r_2, the following are regex
  - $L(r_1|r_2) = L(r_1) \cup L(r_2)$
  - $L(r_1r_2) = L(r_1)L(r_2)$
  - $L(r_1^*) = (L(r_1))^*$
  - $L((r_1)) = L(r_1)$

Example

- Build the NFA for the regex:$(a|b)^*c$
- $L\left((a|b)^*c\right) = \left(L(a|b)\right)^* L(c) = \left(L(a) \cup L(b)\right)^* L(c)$

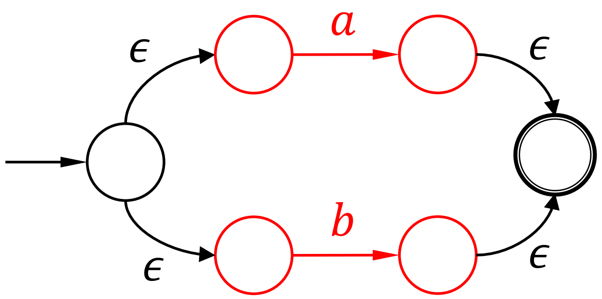
Example

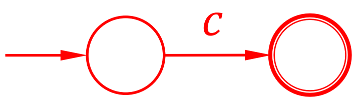
- Build the NFA for the regex:$(a|b)^*c$
- $L\big((a|b)^*c\big) = \big(L(a|b)\big)^*L(c) = \big(L(a) \cup L(b)\big)^*L(c)$



Example

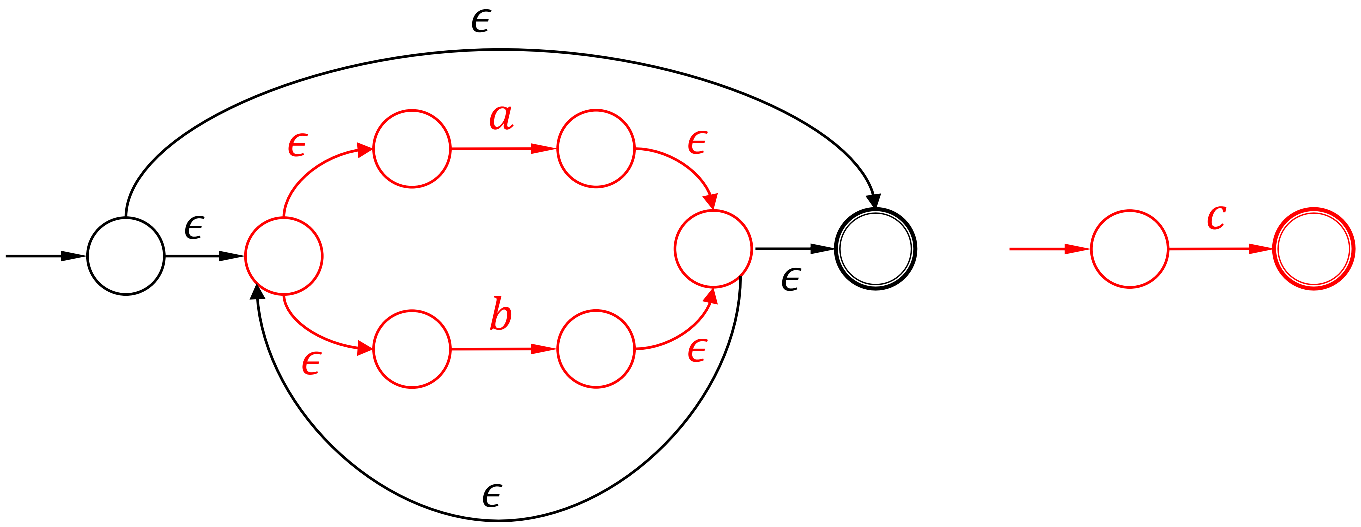
- Build the NFA for the regex:$(a|b)^*c$
- $L\left((a|b)^*c\right) = \left(L(a|b)\right)^* L(c) = \left(L(a) \cup L(b)\right)^* L(c)$





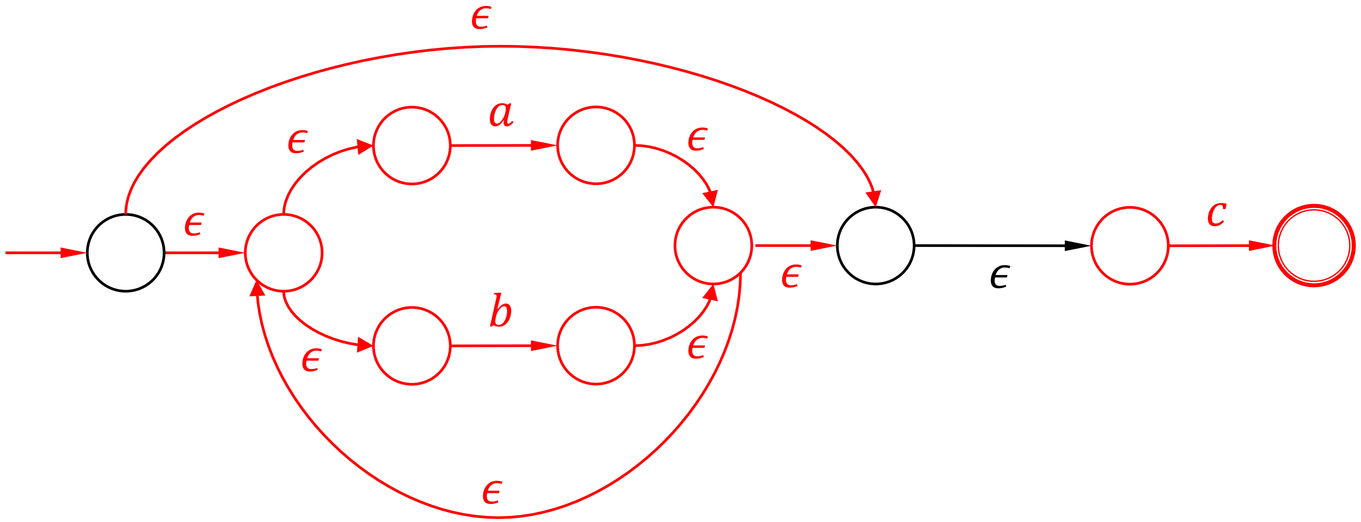
Example

- Build the NFA for the regex:$(a|b)^*c$
- $L\left((a|b)^*c\right) = \left(L(a|b)\right)^* L(c) = \left(L(a) \cup L(b)\right)^* L(c)$



Example

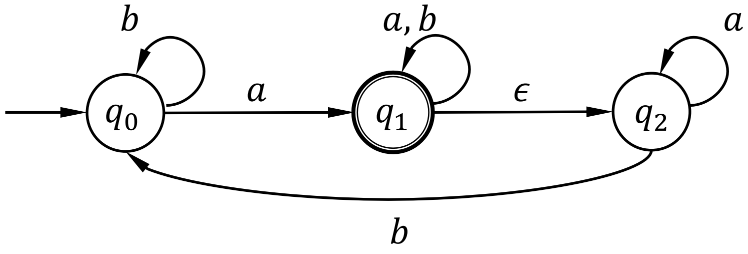
- Build the NFA for the regex: $(a|b)^*c$
- $L\big((a|b)^*c\big) = \big(L(a|b)\big)^*L(c) = \big(L(a) \cup L(b)\big)^*L(c)$

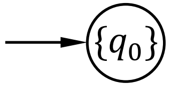


From NFA to DFA

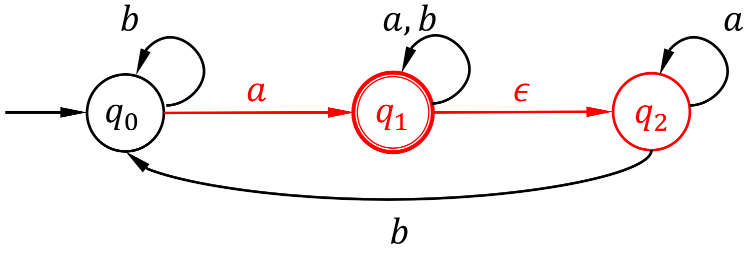
- **Subset Construction**
- A subset of NFA states is a DFA state

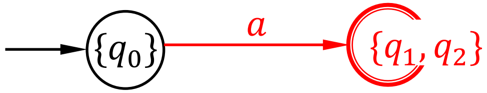
From NFA to DFA



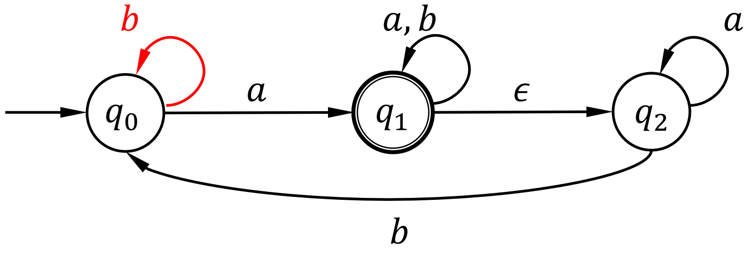


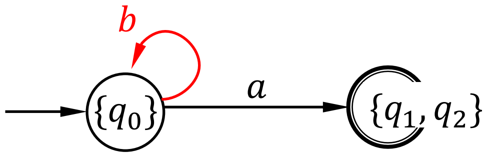
From NFA to DFA



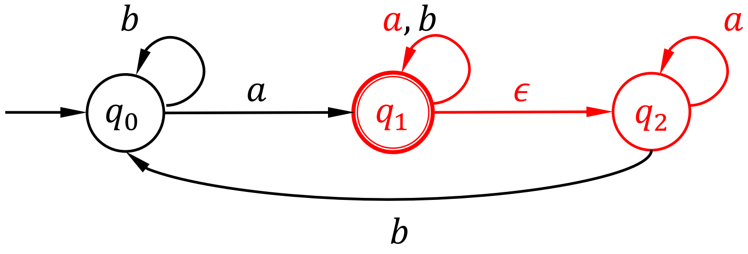


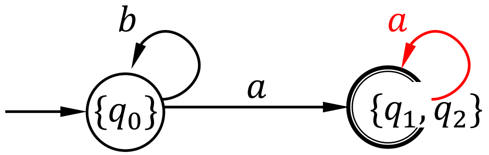
From NFA to DFA



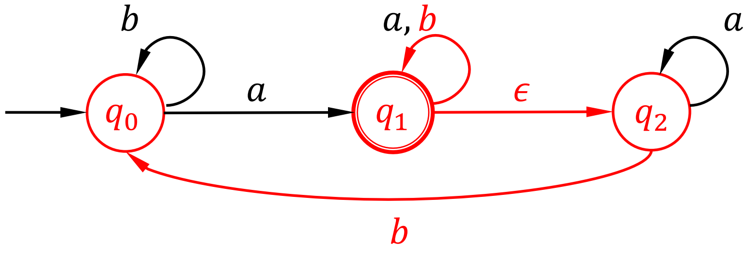


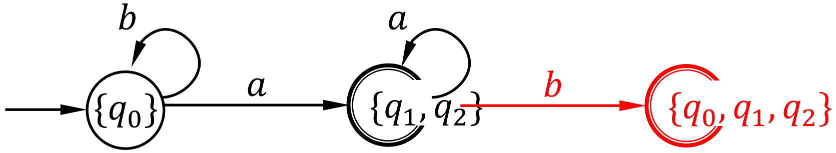
From NFA to DFA



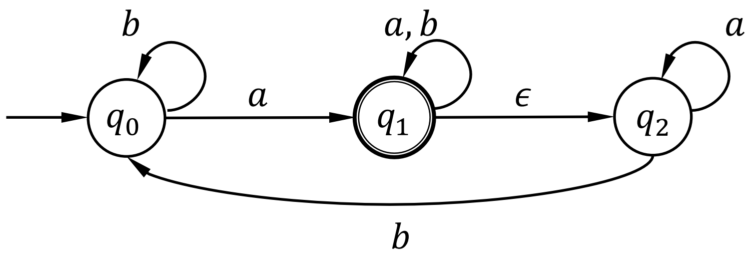


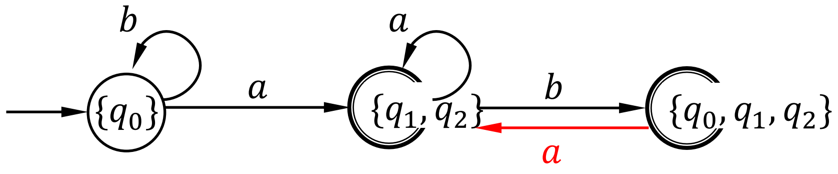
From NFA to DFA



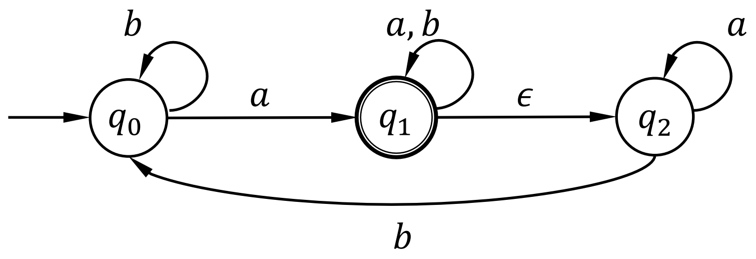


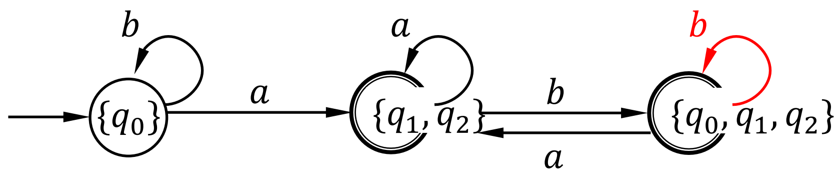
From NFA to DFA





From NFA to DFA



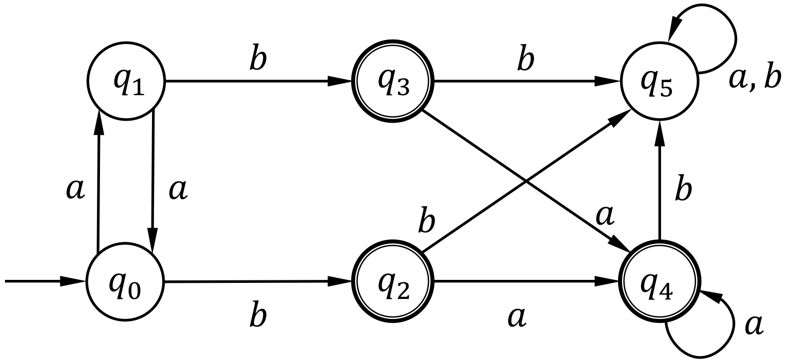


DFA Minimization

- Minimizing DFA can improve the efficiency of computation

DFA Minimization

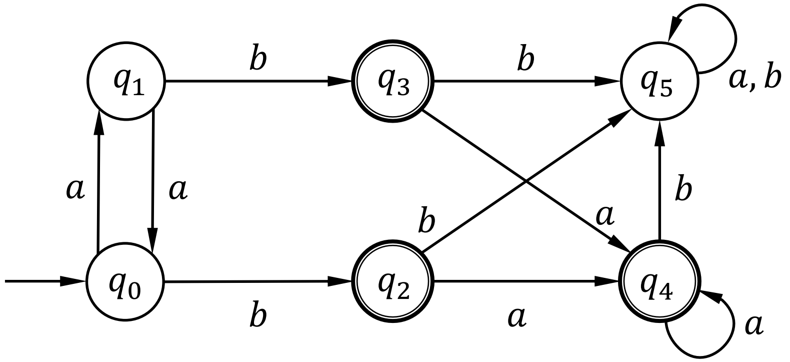
- Minimizing DFA can improve the efficiency of computation



	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

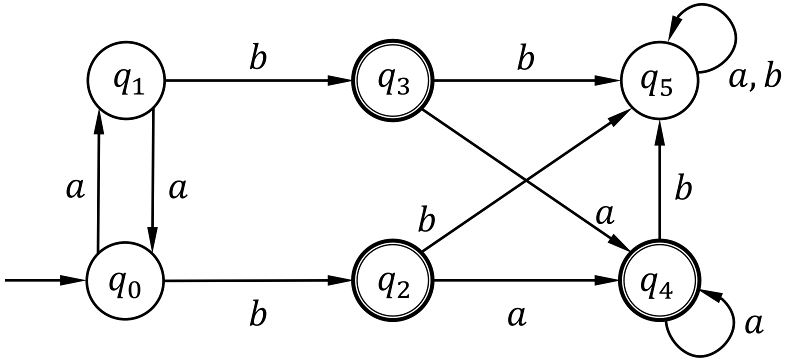


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2		
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

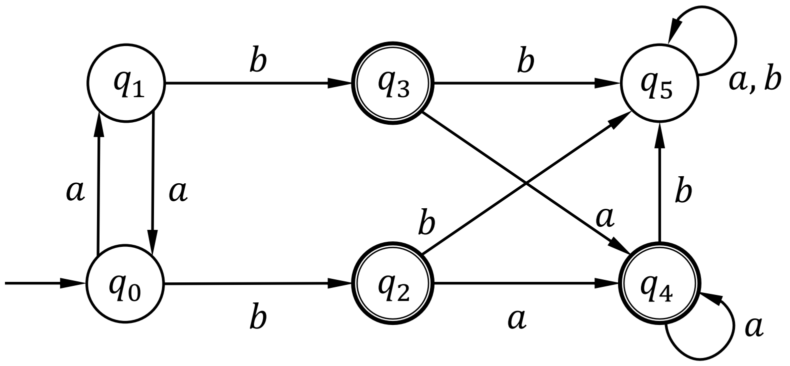


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

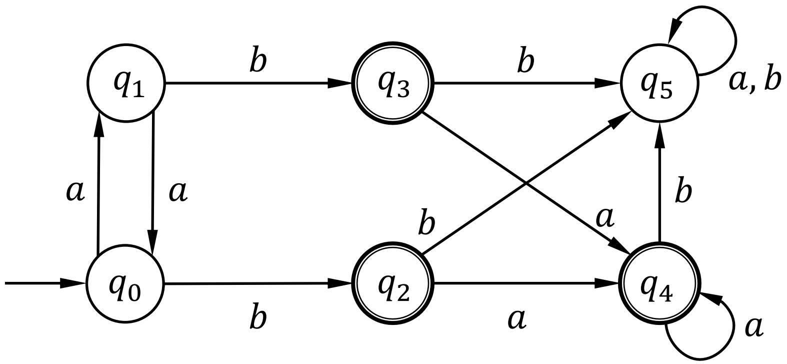


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

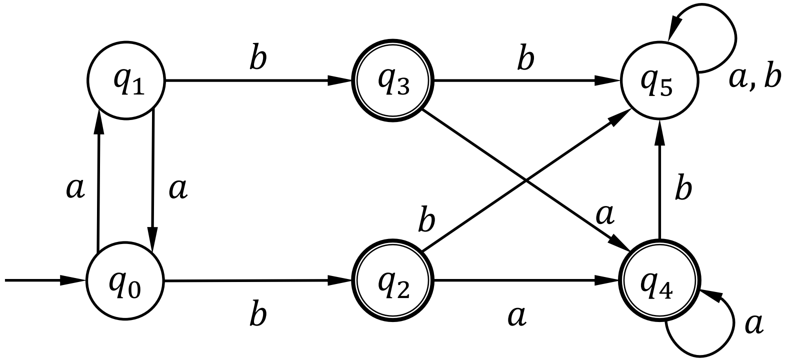


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

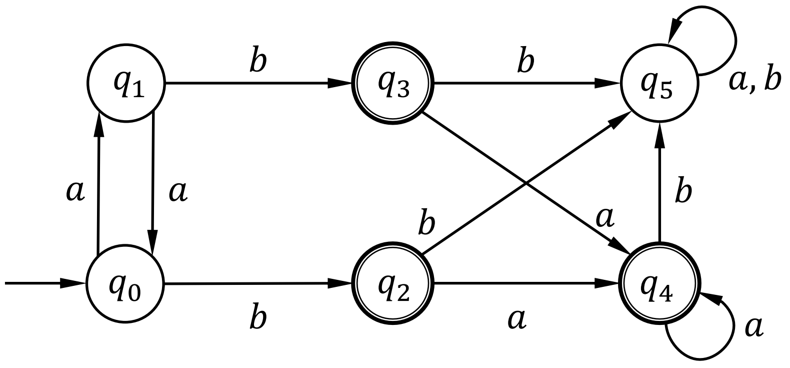


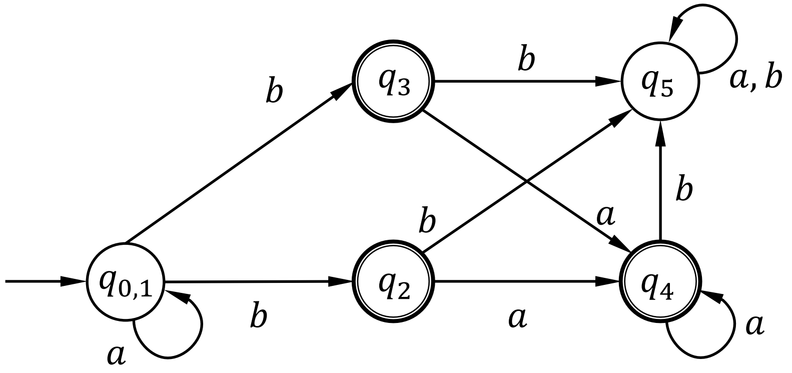
	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{q_0, q_1, q_5\}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	The result does not change and the algorithm completes

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

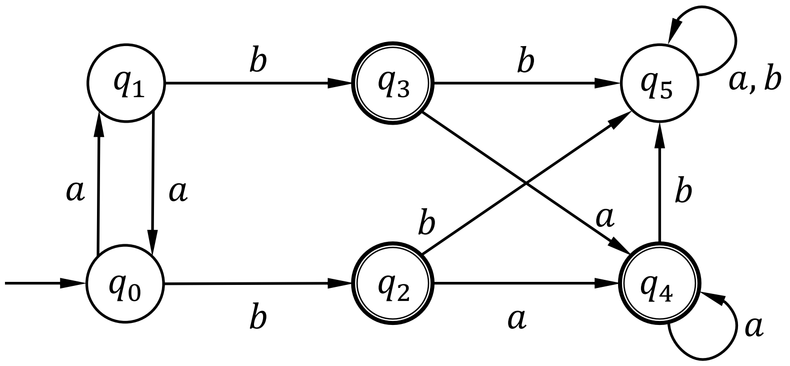


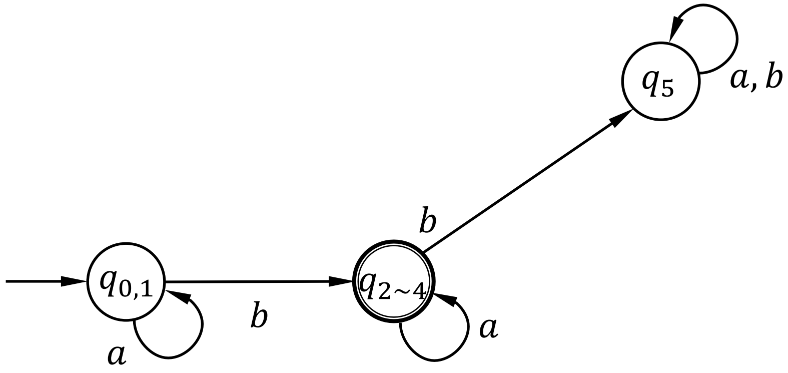


Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3	$\{ q_0, q_1 \}, \{ q_5 \}, \{q_2, q_3, q_4\}$	The result does not change and the algorithm completes

DFA Minimization

- Minimizing DFA can improve the efficiency of computation





Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3	$\{ q_0, q_1 \}, \{ q_5 \}, \{q_2, q_3, q_4\}$	The result does not change and the algorithm completes

PART II: CFG and Parsing

Context Free Grammar (CFG)

- A context-free grammar is a tuple $G = (N, T, S, P)$
  - N: a finite set of non-terminals
  - T: a finite set of terminals, such that $N \cap T = \emptyset$
  - $S \in N$: start non-terminals
  - P: production rules in the form of $A \rightarrow a$, where $A \in N$ and $a \in (N \cup T)^*$

```
assignment → identifier = expression
 expression → term + term
        term → identifier
        term → identifier * number
```

```
assignment → identifier = expression
 expression → term + term
        term → identifier
                 | identifier * number
```

Exercises

$S \rightarrow aSb \mid \epsilon$ is the grammar for $a^n b^n$

- Write a context-free grammar for the following languages
  - 0^*1^*

Exercises

$S \rightarrow aSb \mid \epsilon$ is the grammar for $a^n b^n$

- Write a context-free grammar for the following languages
  - $0^n 1^{2n}$

Exercises

$S \rightarrow aSb \mid \epsilon$ is the grammar for $a^n b^n$

- Write a context-free grammar for the following languages
  - L = { w | w is a string of balanced parentheses }

Exercises

$S \rightarrow aSb \mid \epsilon$ is the grammar for $a^n b^n$

- Write a context-free grammar for the following languages
  - $a^i b^j c^k$ where $i = j$ or $j = k$

Exercises

$S \rightarrow aSb \mid \epsilon$ is the grammar for $a^n b^n$

- Write a context-free grammar for the following languages
  - $a^i b^j c^k$ where $i \neq j$ or $j \neq k$

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1**: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again
- **Problem 2**: Backtracking may be necessary
  - when one derivation does not work, we may try others

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive

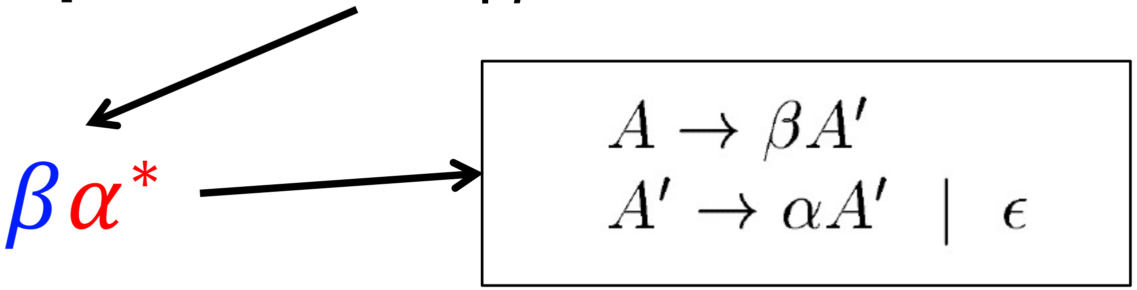
Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive, can be transformed into

$$\beta\alpha^{*}$$

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \to A\alpha \mid \beta$ is left-recursive, can be transformed into



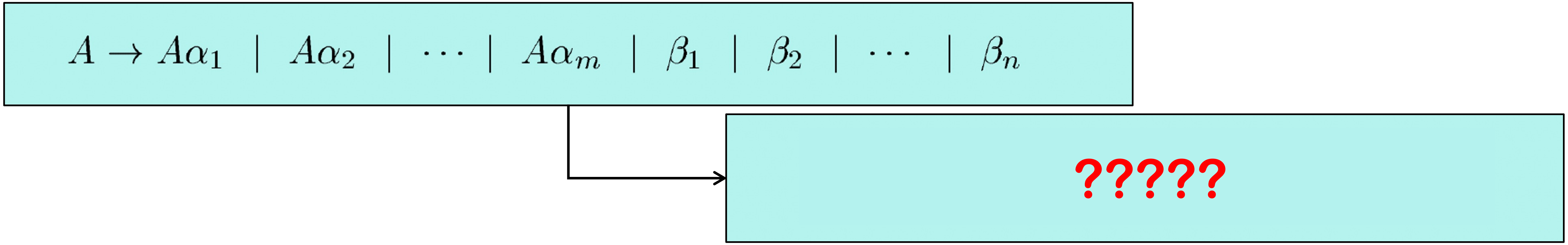
$\beta\alpha^{*}$

$$
\begin{aligned}
A &\to \beta A' \\
A' &\to \alpha A' \mid \epsilon
\end{aligned}
$$

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive, can be transformed into

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \quad \mid \quad \epsilon \end{aligned}$$



$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

?????

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \to A\alpha \mid \beta$ is left-recursive, can be transformed into

$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \epsilon \end{aligned}$$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1** ✓: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again
- **Problem 2**: Backtracking may be necessary
  - when one derivation does not work, we may try others

Predictive Parsing

- Predictive parsers are recursive descent parser w/o backtracking

Predictive Parsing

- Predictive parsers are recursive descent parser w/o backtracking
- LL(1)
  - L: scanning input from left to right
  - L: leftmost derivation
  - 1: Using one input symbol of lookahead at each step

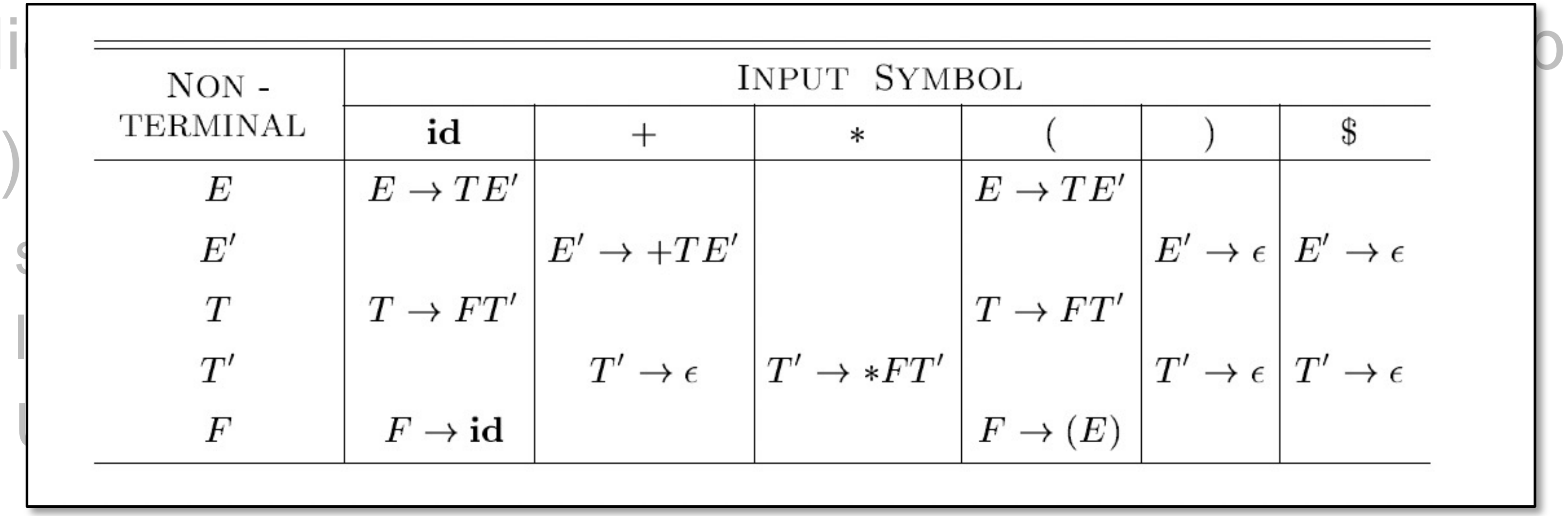
Predictive Parsing

- Predictive parsers are recursive descent parser w/o backtracking
- LL(1)
  - L: scanning input from left to right
  - L: leftmost derivation
  - 1: Using one input symbol of lookahead at each step

- LL(1) grammar (Not ambiguous! Not left-recursive!)
  - Rich enough to cover most programming constructs

Predictive Parsing

- Predi backtracking
- LL(1)
  - L: s
  - L: l
  - 1: U

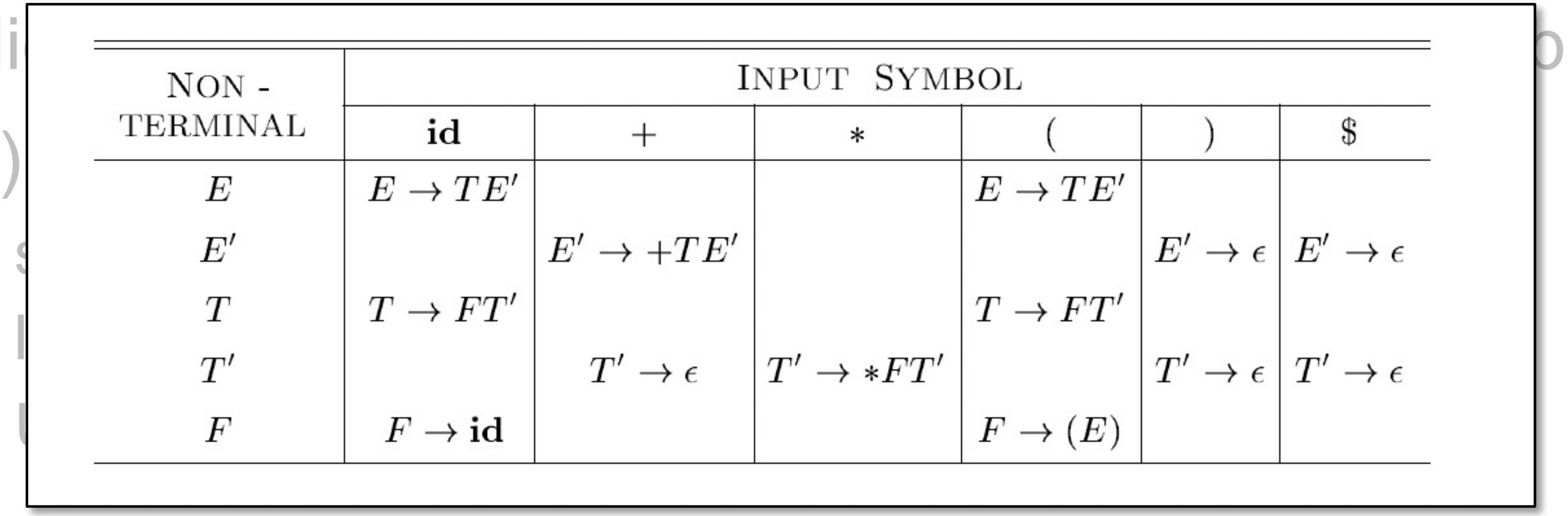


NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

- LL(1) grammar (Not ambiguous! Not left-recursive!)
  - Rich enough to cover most programming constructs

Predictive Parsing

- Predi backtracking
- LL(1)
  - L: s
  - L: l
  - 1: U



NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

- LL(1) grammar (Not ambiguous! Not left-recursive!)
  - Rich enough to cover most programming constructs
  - To build the predictive table, let's define $\mathrm{FIRST}(\alpha);\ \mathrm{FOLLOW}(\alpha)$

First() and Follow()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with

First() and Follow()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with
- $\mathrm{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α

First()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\mathrm{FIRST}(a) = \{a\}$
- $\mathrm{FIRST}(b) = \{b\}$
- $\mathrm{FIRST}(c) = \{c\}$

First()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\mathrm{FIRST}(a) = \{a\}$
- $\mathrm{FIRST}(b) = \{b\}$
- $\mathrm{FIRST}(c) = \{c\}$
- $\mathrm{FIRST}(S) = \{c\}$
- $\mathrm{FIRST}(A) = \{a\}$

Follow()

- $\mathrm{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α
  - $\$ \in \mathrm{FOLLOW}(S)$, where $\$$ is string's end marker, S the start non-terminal

Follow()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b\ |\ a$
- $\mathrm{FOLLOW}(S) = \{\$\}$
- $\mathrm{FOLLOW}(A) = \{b\}$

Follow()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) = \{b\}$

- **Example**: $S \to c\ A\ A;\ \ A \to a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$

Follow()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\mathrm{FOLLOW}(S) = \{\$\}$
- $\mathrm{FOLLOW}(A) = \{b\}$

- **Example**: $S \to c\ A\ A;\ \ A \to a\ b \mid a$
- $\mathrm{FOLLOW}(S) = \{\$\}$
- $\mathrm{FOLLOW}(A) \supseteq \mathrm{FIRST}(A) \backslash \{\epsilon\} = \{a\}$

Follow()

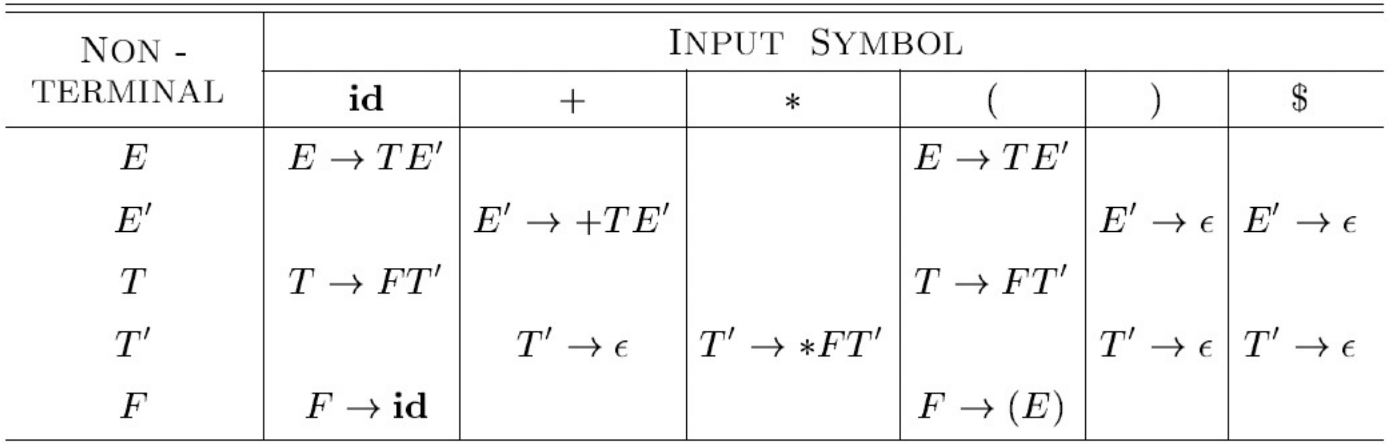
- **Example**: $S \rightarrow c\ A\ b;\ \ A \rightarrow a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) = \{b\}$

- **Example**: $S \rightarrow c\ A\ A;\ \ A \rightarrow a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) \supseteq \text{FIRST}(A) \backslash \{\epsilon\} = \{a\}$
- $\text{FOLLOW}(A) \supseteq \text{FOLLOW}(S) = \{\$\}$

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \rightarrow \alpha$
  - $\forall a \in \mathrm{FIRST}(\alpha){:}\ M[A, a] = A \rightarrow \alpha$
  - $\epsilon \in \mathrm{FIRST}(\alpha) \Rightarrow \ \forall b \in \mathrm{FOLLOW}(A){:}\ M[A, b] = A \rightarrow \alpha$

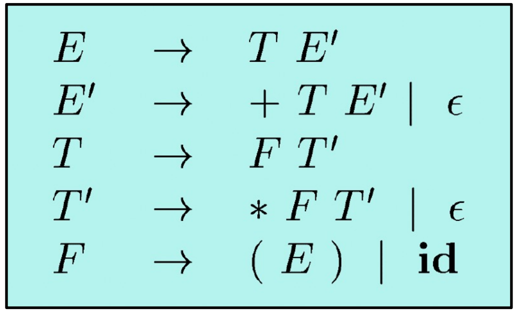
$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$



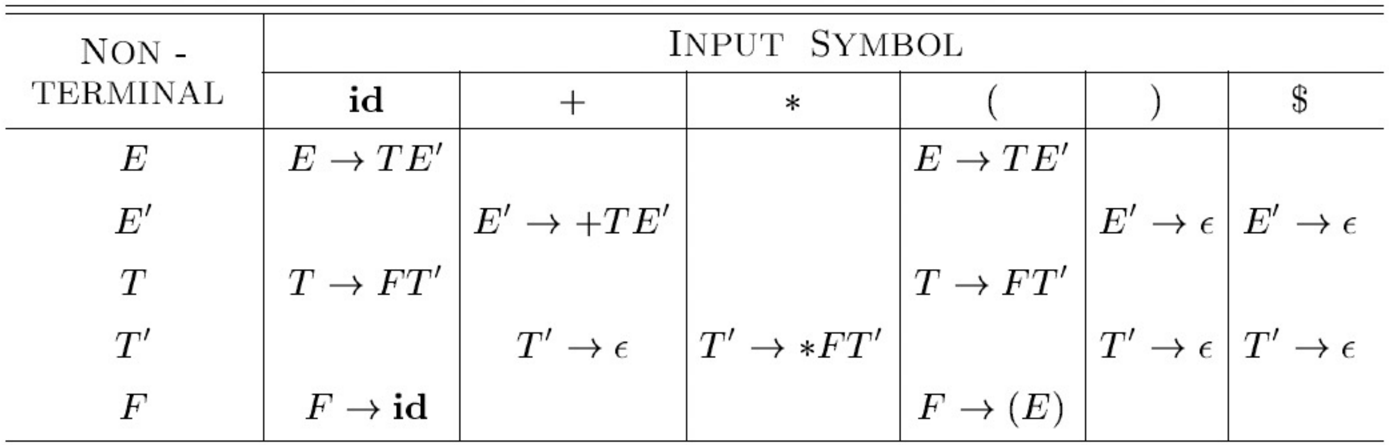
NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \rightarrow \alpha$
  - **$\forall a \in \mathbf{FIRST}(\alpha): M[A, a] = A \rightarrow \alpha$**
  - $\epsilon \in \text{FIRST}(\alpha) \Rightarrow \ \forall b \in \text{FOLLOW}(A): M[A, b] = A \rightarrow \alpha$



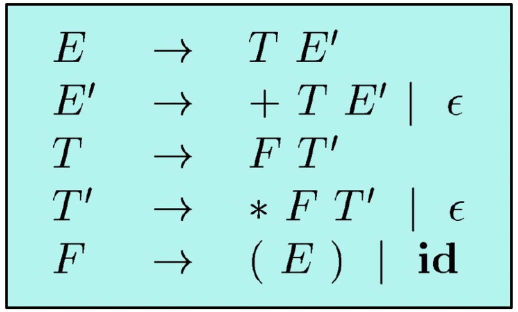
$$\begin{array}{lll} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \textbf{id} \end{array}$$



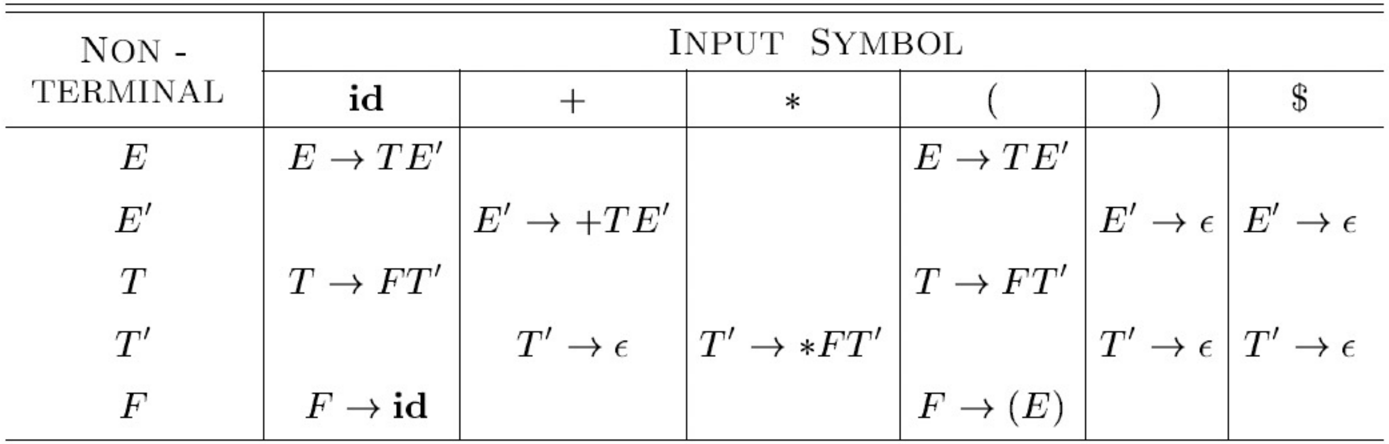
NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \to \alpha$
  - $\forall a \in \text{FIRST}(\alpha)$: $M[A, a] = A \to \alpha$
  - $\boldsymbol{\epsilon \in \textbf{FIRST}(\alpha) \Rightarrow \ \forall b \in \textbf{FOLLOW}(A)\text{: } M[A, b] = A \to \alpha}$



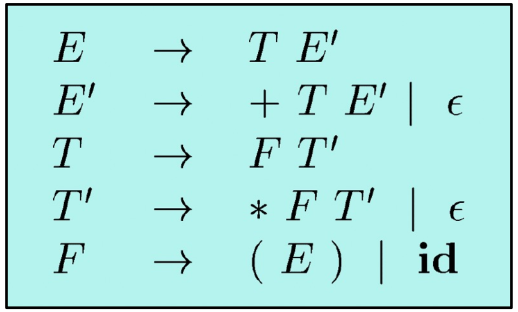
$$
\begin{array}{lcl}
E & \to & T\ E' \\
E' & \to & +\ T\ E'\ |\ \epsilon \\
T & \to & F\ T' \\
T' & \to & *\ F\ T'\ |\ \epsilon \\
F & \to & (\ E\ )\ |\ \textbf{id}
\end{array}
$$



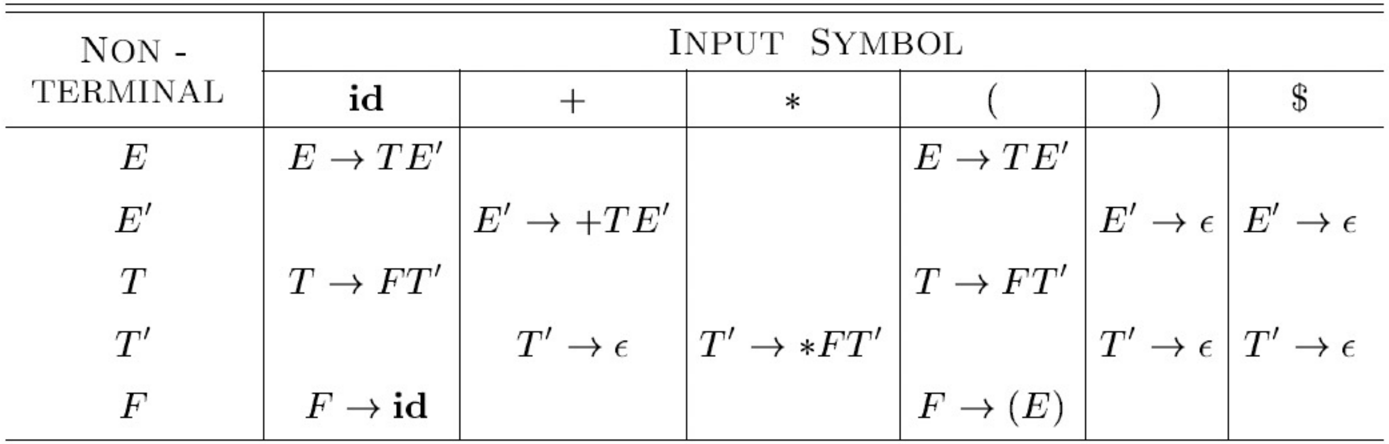
NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \rightarrow \alpha$
  - $\forall a \in \text{FIRST}(\alpha)\colon M[A, a] = A \rightarrow \alpha$
  - $\epsilon \in \text{FIRST}(\alpha) \Rightarrow \ \forall b \in \text{FOLLOW}(A)\colon M[A, b] = A \rightarrow \alpha$



$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \textbf{id} \end{array}$$



NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

- **Choose the production rule as per the table; empty means error**

Building a Parser in Practice

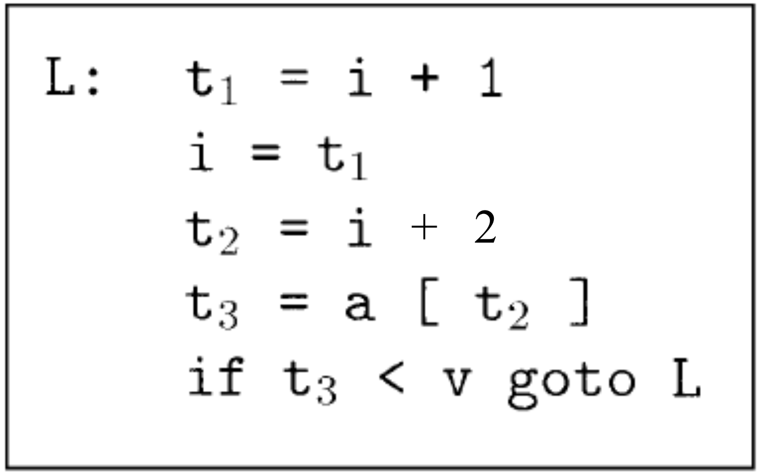
- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem** ✓ **1**: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again
- **Problem** ✓ **2**: Backtracking may be necessary
  - when one derivation does not work, we may try others

PART III: IR Generation

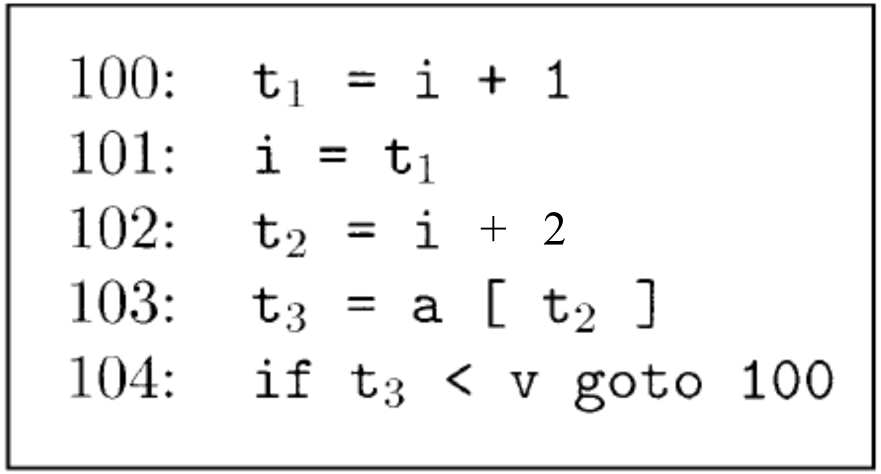
Three-Address Code

- do i = i + 1; while (a[i + 2] < v);



L: t_1 = i + 1
i = t_1
t_2 = i + 2
t_3 = a [t_2]
if t_3 < v goto L

Symbolic Labels



100: t_1 = i + 1
101: i = t_1
102: t_2 = i + 2
103: t_3 = a [t_2]
104: if t_3 < v goto 100

Numeric Labels

- Implementation methods: *quadruples*, *triples*, etc.

Static Single-Assignment

- **Feature 1**: Every variable has only one definition
- **Feature 2**: Using φ to merge definitions from multi paths
- => Direct def-use chains

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

if (flag) x_1 = -1; else x_2 = 1;

$x_3 = \varphi\ (x_1, x_2);$

$y\ = x_3\ *\ a$

Dominance Relations

- A **dom** B
  - if all paths from Entry to B goes through A
- A **post-dom** B
  - if all paths from B to Exit goes through A

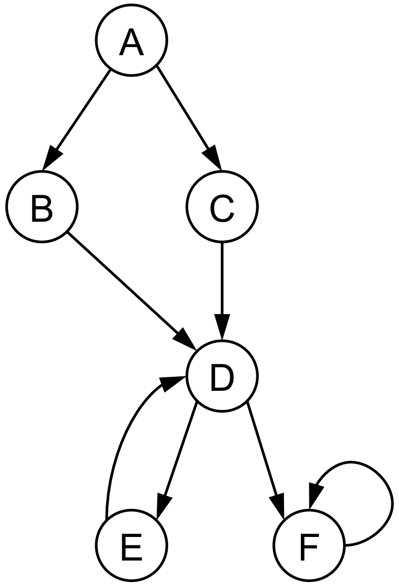
- Strict (post-)dominance – A (post-)**dom** B but A ≠ B

- Immediate dominance – A strict-dom B, but there’s no C, such that A strict-dom C, C strict-dom B

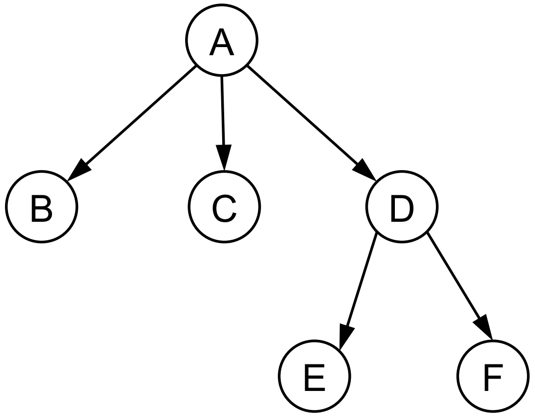
Dominator Tree

- Almost linear time to build a dominator tree.
  - Node: Block
  - Edge: Immediate dom relation

- Why is it a tree?



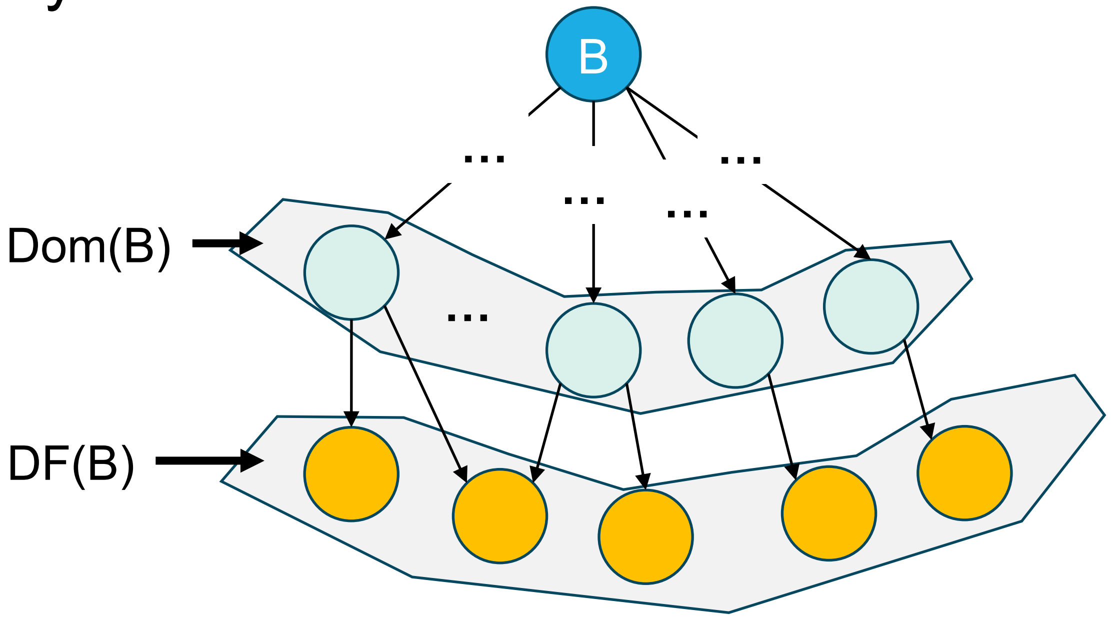
Flow Graph



Dominator Tree

Dominance Frontier

- DF(B) = { … } for the block B
  - The immediate successors of the blocks dominated by B
  - Not strictly dominated by B



Dominance Frontier

- DF(B) = { … } for the block B
  - The immediate successors of the blocks dominated by B
  - Not strictly dominated by B

- DF($\mathbb{B}$) = { … } for a set of blocks $\mathbb{B}$
  - DF($\mathbb{B}$) = $\cup_{B\in\mathbb{B}}$ DF(B)

Iterated Dominance Frontier

- Iterated DF of a block set $\mathbb{B}$
  - $DF_1 = DF(\mathbb{B}); \mathbb{B} = \mathbb{B} \cup DF_1$
  - $DF_2 = DF(\mathbb{B}); \mathbb{B} = \mathbb{B} \cup DF_2$
  - ……
  - until fixed point! (i.e., $DF_n = DF_{n-1}$)

THANKS!