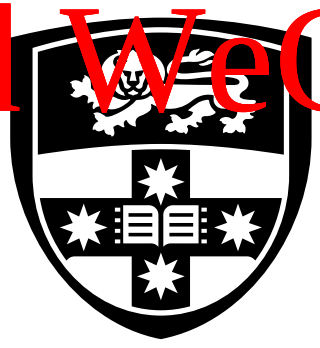
COMP2022: Formal Languages and Logic

2018, Semester 2, Week 8

Kalina Yacef, Joseph Godbehere

20th September, 2018



THE UNIVERSITY OF SYDNEY

Outline

- Revision

- Push Down Automata

- LL(k) Table-Descent Parsing

Revision

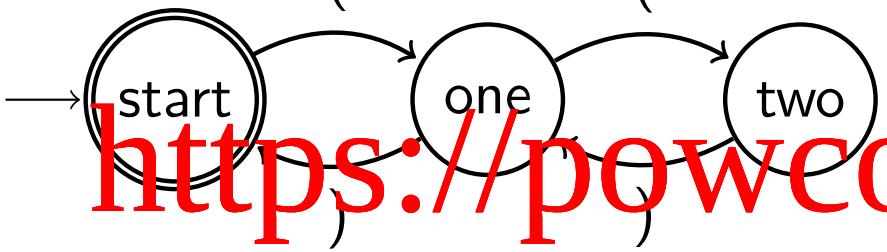
- Generate Context-Free Languages
  - Very important class of languages in CS (compilers, NLP, etc.)
- All rules are in the form $A \rightarrow \alpha$
- Closed under Union, Concatenation and Star Closure
- String derivation (left-most, right-most)
- Ambiguous grammars
- Clean grammars

Next lecture:

- Push-Down Automata
- Parsing

Limits of DFA/NFA

Recall this NFA for balanced parentheses



It accepts strings of balanced parentheses, nested up to two deep.

What if we wanted a depth of 3?

What if we want any level of nesting?

If we added a *stack* to the automata, we could accept any level of nesting

Push Down Automata

Finite automata *with a stack*

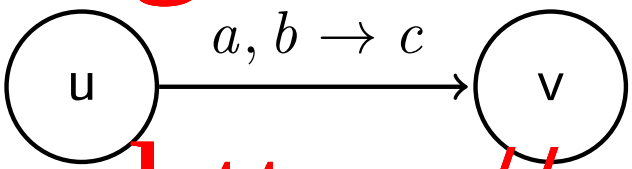
A PDA is a machine with the ability to:

- Read the letters of input string (like FA)
- Make state changes (like FA)
- Perform stack operations (new!)

In addition to storing terminals and variables, the stack will accept a special end of string symbol, which we will denote $.

PDA Transitions

PDA Transitions include stack operations



$a, b \to c$ means:

- Read (and remove) symbol a from the front of the input
- Pop symbol b from the top of the stack
- Push c onto the stack

We follow a transition if it is possible to perform these operations

PDA Transitions

We use ε to denote "no operation". For example:

- $a, \varepsilon \to c$
  - read (and remove) a from the front of the input
  - do not pop anything from the stack
  - push c onto the stack

PDA Transitions

We use ε to denote "no operation". For example:

- $a, \varepsilon \to c$
  - read (and remove) a from the front of the input
  - do not pop anything from the stack
  - push c onto the stack

- $\varepsilon, \varepsilon \to c$
  - do not read anything from the input
  - do not pop anything from the stack
  - push c onto the stack

PDA Transitions

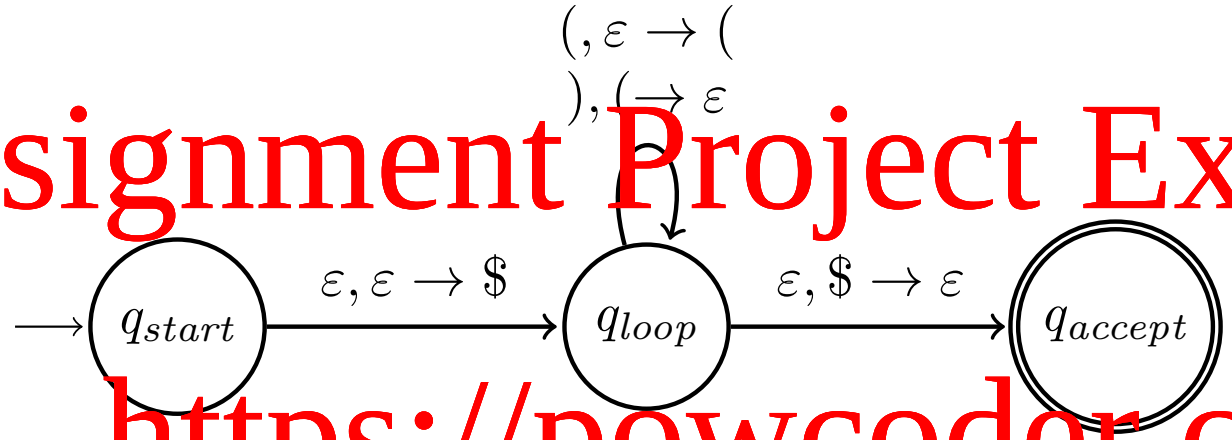
We use ε to denote "no operation". For example:

- $a, \varepsilon \to c$
  - read (and remove) a from the front of the input
  - do not pop anything from the stack
  - push c onto the stack

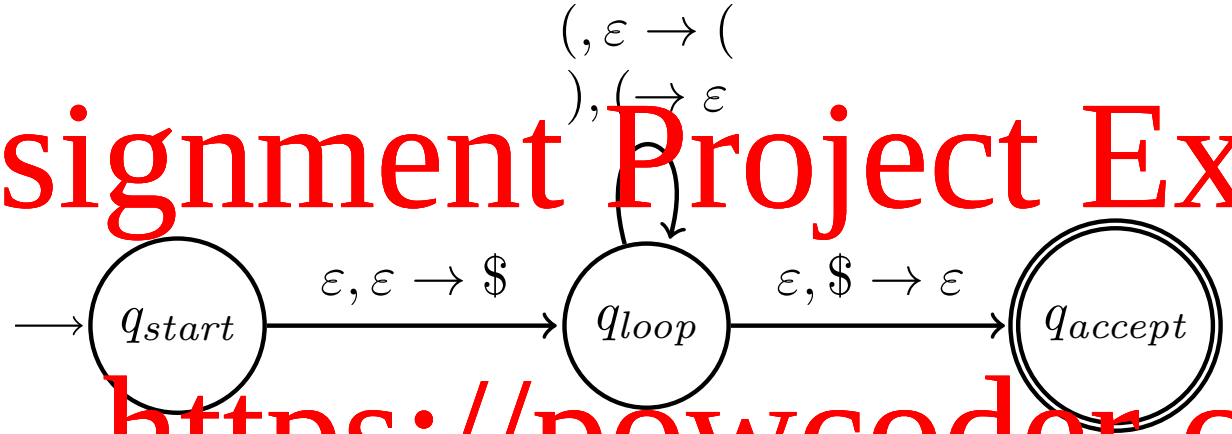
- $\varepsilon, \varepsilon \to c$
  - do not read anything from the input
  - do not pop anything from the stack
  - push c onto the stack

- $\varepsilon, b \to \varepsilon$
  - do not read anything from the input
  - pop b from the top of the stack
  - do not push anything onto the stack

Parsing ((()))

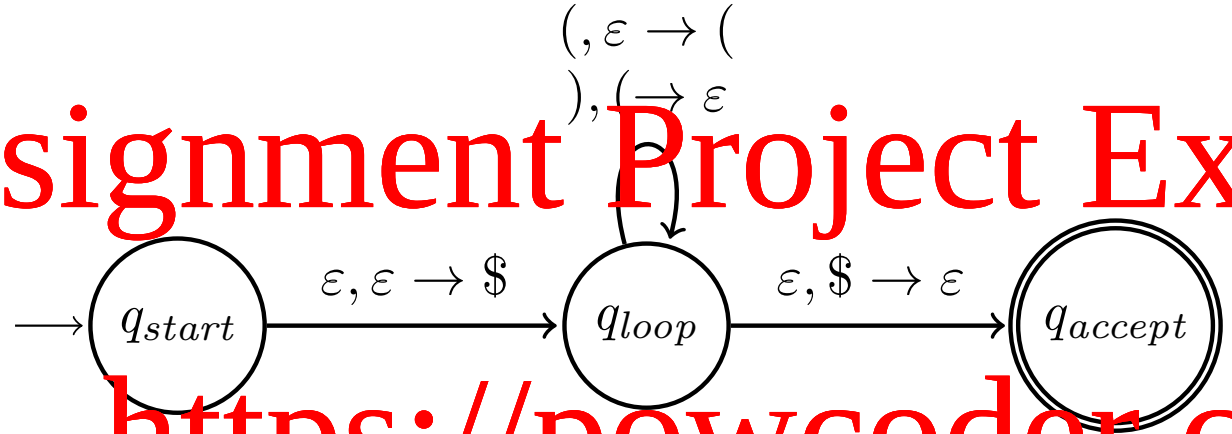


Parsing ((()))



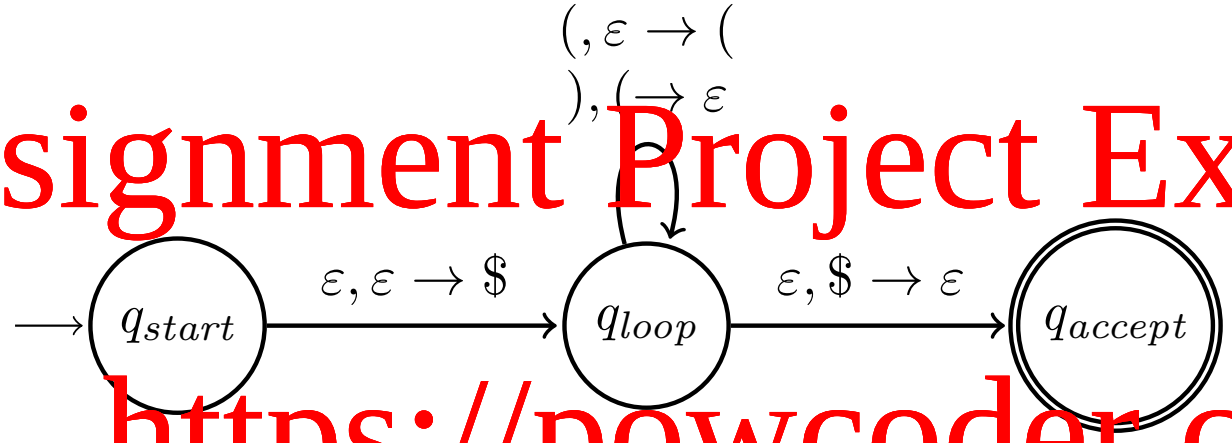
Current state	Unscanned input	Stack contents
q_{start}	$((()))$	$\emptyset$

Parsing ((()))



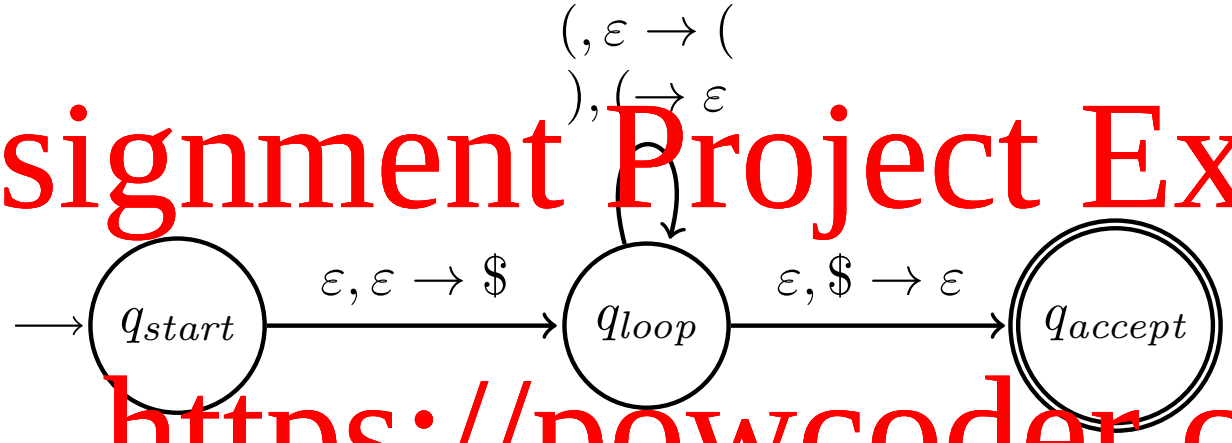
Current state	Unscanned input	Stack contents
q_{start}	(()())	$\emptyset$
q_{loop}	(()())	$

Parsing ((()))



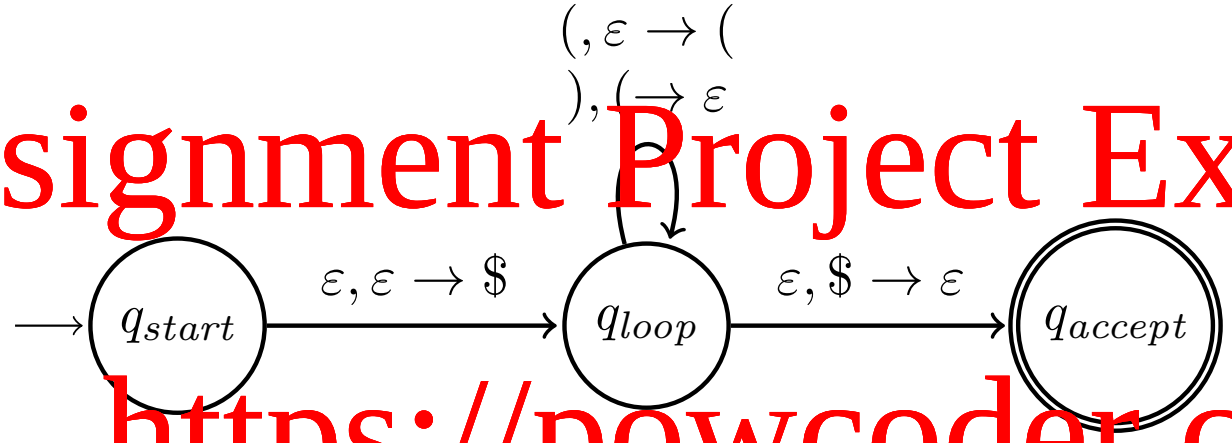
Current state	Unscanned input	Stack contents
q_{start}	(((())))	$\emptyset$
q_{loop}	(((())))	$
q_{loop}	((()))	($

Parsing ((()))



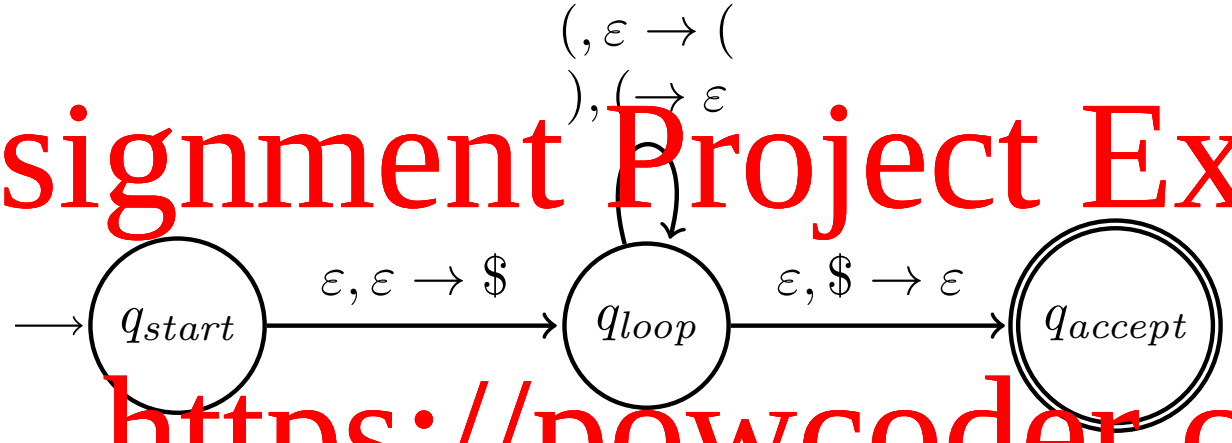
Current state	Unscanned input	Stack contents
q_{start}	((()))	$\emptyset$
q_{loop}	((()))	$
q_{loop}	(())	($
q_{loop}	())	(($

Parsing ((()))



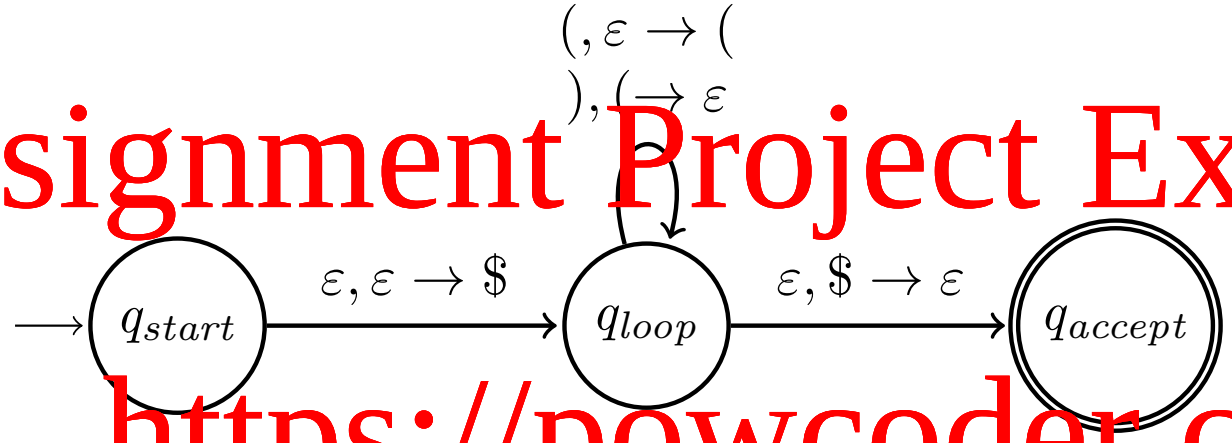
Current state	Unscanned input	Stack contents
q_{start}	((()))	$\emptyset$
q_{loop}	((()))	$
q_{loop}	(())	($
q_{loop}	())	(($
q_{loop}	))	(($

Parsing ((()))



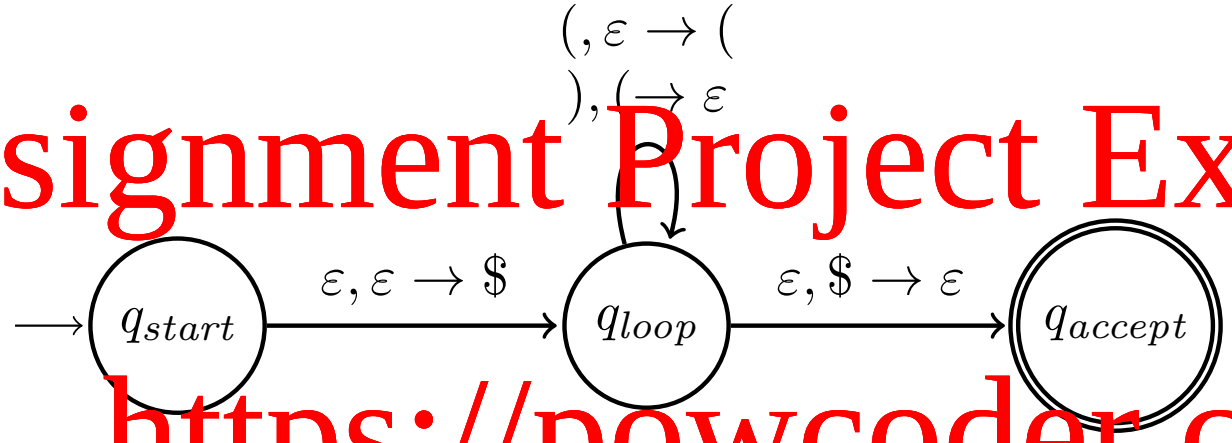
Current state	Unscanned input	Stack contents
q_{start}	((()))	$\emptyset$
q_{loop}	((()))	$
q_{loop}	(()))	($
q_{loop}	()))	(($
q_{loop}	)))	(((($
q_{loop}	))	((($

Parsing ((()))



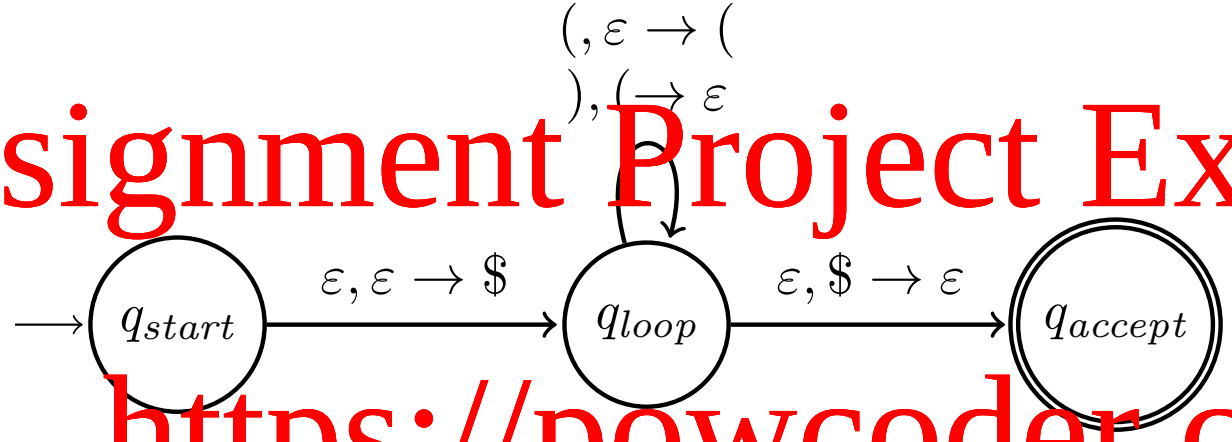
Current state	Unscanned input	Stack contents
q_{start}	((()))	$\emptyset$
q_{loop}	((()))	$
q_{loop}	(()))	($
q_{loop}	()))	(($
q_{loop}	)))	(((($
q_{loop}	))	((($
q_{loop}	)	(($

Parsing ((()))



Current state	Unscanned input	Stack contents
q_{start}	((()))	$\emptyset$
q_{loop}	((()))	$
q_{loop}	(()))	($
q_{loop}	()))	(($
q_{loop}	)))	(((($
q_{loop}	))	((($
q_{loop}	)	(($
q_{loop}	ε	($

Parsing ((()))



Current state	Unscanned input	Stack contents
q_{start}	(()())	$\emptyset$
q_{loop}	(()())	$
q_{loop}	()())	($
q_{loop}	())	(($
q_{loop}	))	(($
q_{loop}	)	($
q_{loop}	ε	$
q_{accept}	ε	$\emptyset$

From CFG → PDA → Table Driven parsers

The interesting part of the PDA is the stack and the Q_{loop} state

The stack remembers the part of the string that is yet to be derived.

This can be programmed as a descent table-driven parser

- Input: current token and variable on top of the stack
- Output: which rule to use

Non-determinism in the table is a problem: how do we choose which rule to use?

Try to construct a deterministic table whenever possible

- Not all CFG have an equivalent *deterministic* PDA

Parsing bcc with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	\$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing bcc with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing bcc with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	\$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing *bcc* with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing *bcc* with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing bcc with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing *bcc* with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$
$c\$$	$C\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing bcc with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$
$c\$$	$C\$$
$c\$$	$c\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing *bcc* with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$
$c\$$	$C\$$
$c\$$	$cC\$$
$\$$	$C\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing *bcc* with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$
$c\$$	$C\$$
$c\$$	$cC\$$
$\$$	$C\$$
$\$$	$\$$

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing *bcc* with a table-driven parser

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$
$c\$$	$C\$$
$c\$$	$cC\$$
$\$$	$C\$$
$\$$	$\$$
ACCEPTED	

$S \to BC$

$S \to a$

$B \to bB$

$B \to \varepsilon$

$C \to cC$

$C \to \varepsilon$

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Grammars are fundamental for constructing parsers.

Top-down parsing:

Grammars are fundamental for constructing parsers.

Top-down parsing: A parse tree is constructed from the root, proceeding downward towards the leaves

- Constructs the derivation by starting with the grammar's start symbol, then working towards the string, i.e. $S \Rightarrow^{+} w$
- LL(k) parsing
  - **L**eft-to-right, **L**eftmost derivation

Grammars are fundamental for constructing parsers.

Top-down parsing: A parse tree is constructed from the root, proceeding downward towards the leaves

- Constructs the derivation by starting with the grammar's start symbol then working towards the string. i.e. $S \Rightarrow^{+} w$
- LL(k) parsing
  - **L**eft-to-right, **L**eftmost derivation

Parsers should be efficient, so determinism is important.

LL(1) parsing

LL(1) grammar

- L: Left to right scanning of input
- L: Leftmost derivation

Deterministic derivation by looking ahead 1 symbols

Using less lookahead symbols is usually more efficient

Lookahead symbols

Suppose we have a grammar with two rules for S:

$$S \to XY \mid YZ \qquad \text{(other rules not shown)}$$

How do we choose which rule should replace S?

Suppose XY can only derive strings which start with a, and YZ can only derive strings which start with b or c.

Then if we look ahead one symbol, we know which rule to select:

- to derive abc we must choose $S \Rightarrow XY$
- to derive cab we must choose $S \Rightarrow YZ$

Table-driven LL(1) parsing

In a PDA: the stack contains the right hand side of the rules for a leftmost derivation

In a *descent table-driven parser*: Given the current input and the variable on top of the stack, the table specifies which production rule to use.

Descent table-driven LL(1) parser


```
loop
    T = symbol on top of the stack
    c = current input symbol
    if T == c == $ then accept
    else if T is a terminal or T == $ then
        if T == c then pop T and consume c
        else error
    else if P[T, c] = α is defined then
        pop T and push α onto the stack
                //(in reverse order)
    else error
endloop
```

Rules starting with non-terminals

$$S \to A \mid B$$
$$A \to aA \mid \varepsilon$$
$$B \to bB \mid c$$

The productions for A and B *seem* LL(1), but how can we choose which rule to use when deriving from S?

We look at the possible symbols which can start strings derived after $S \to A$, or after $S \to B$.

The set of symbols which could start any string derived from α is called the FIRST set of α

FIRST and FOLLOW sets

In order to fill in the entries of the table-drived parser we need to compute some FIRST and FOLLOW sets.

$FIRST(\alpha)$ is the set of all terminals which could start strings derived from α. We will need to calculate these for every production of G (i.e. the *right hand side* of each rule).

$FOLLOW(V)$ is the set of all terminals which which could follow the variable V at any stage of the derivation. Needed whenever V can derive ε.

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. If $\alpha \stackrel{+}{\Rightarrow} \varepsilon$ then $\varepsilon \in FIRST(\alpha)$.

Construction rules: Where a is a terminal, and α is some string of terminals and variables:

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. If $\alpha \stackrel{+}{\Rightarrow} \varepsilon$ then $\varepsilon \in FIRST(\alpha)$.

Construction rules: Where a is a terminal, and α is some string of terminals and variables:

1. $FIRST(\varepsilon) = \{\varepsilon\}$ (by definition)

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. If $\alpha \Rightarrow^+ \varepsilon$ then $\varepsilon \in FIRST(\alpha)$.

Construction rules: Where a is a terminal, and α is some string of terminals and variables:

1. $FIRST(\varepsilon) = \{\varepsilon\}$ (by definition)
2. $FIRST(a\alpha) = FIRST(a) = \{a\}$

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in FIRST(\alpha)$.

Construction rules: Where a is a terminal, and α is some string of terminals and variables:

1. $FIRST(\varepsilon) = \{\varepsilon\}$ (by definition)
2. $FIRST(a\alpha) = FIRST(a) = \{a\}$
3. If $A \to \alpha_1 \mid \ldots \mid \alpha_n$ then $FIRST(A) = FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n)$

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in FIRST(\alpha)$.

Construction rules: Where a is a terminal, and α is some string of terminals and variables:

1. $FIRST(\varepsilon) = \{\varepsilon\}$ (by definition)
2. $FIRST(a\alpha) = FIRST(a) = \{a\}$
3. If $A \to \alpha_1 \mid \ldots \mid \alpha_n$ then $FIRST(A) = FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n)$
4. If $\alpha \neq \varepsilon$ then
   - If $\varepsilon \notin FIRST(A)$ then $FIRST(A\alpha) = FIRST(A)$
   - If $\varepsilon \in FIRST(A)$ then $FIRST(A\alpha) = FIRST(A) \setminus \{\varepsilon\} \cup FIRST(\alpha)$

Examples calculating FIRST sets

$$B \to (B)B \mid \varepsilon$$

$FIRST'((B)B') =$

Examples calculating FIRST sets

$$B \to (B)B \mid \varepsilon$$

$FIRST('(B)B') = \{(\}$ rule 2

$FIRST(B) =$

Examples calculating FIRST sets

$$B \to (B)B \mid \varepsilon$$

$$FIRST(\text{“}(B)B\text{”}) = \{(\} \quad \text{rule 2}$$

$$FIRST(B) = FIRST(\text{“}(B)B\text{”}) \cup FIRST(\varepsilon) \quad \text{rule 3}$$

$$=$$

Examples calculating FIRST sets

$$B \to (B)B \mid \varepsilon$$

$$\begin{aligned} FIRST(\text{“}(B)B'\text{”}) &= \{(\} && \text{rule 2} \\ FIRST(B) &= FIRST(\text{“}(B)B\text{”}) \cup FIRST(\varepsilon) && \text{rule 3} \\ &= FIRST(\text{“}(B)B\text{”}) \cup \{\varepsilon\} && \text{rule 1} \\ &= \end{aligned}$$

Examples calculating FIRST sets

$$B \to (B)B \mid \varepsilon$$

$$\begin{aligned}
FIRST(\text{“}(B)B\text{”}) &= \{(\} && \text{rule 2}\\
FIRST(B) &= FIRST(\text{“}(B)B\text{”}) \cup FIRST(\varepsilon) && \text{rule 3}\\
&= FIRST(\text{“}(B)B\text{”}) \cup \{\varepsilon\} && \text{rule 1}\\
&= \{(\} \cup \{\varepsilon\}\\
&= \{(, \varepsilon\}
\end{aligned}$$

Examples calculating FIRST sets

$$S \rightarrow BC \mid a$$
$$B \rightarrow bB \mid \varepsilon$$
$$C \rightarrow cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$FIRST(\varepsilon) =$

Examples calculating FIRST sets

$$S \rightarrow BC \mid a$$
$$B \rightarrow bB \mid \varepsilon$$
$$C \rightarrow cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$FIRST(\varepsilon) = \{\varepsilon\}$ rule 1

$FIRST(B) =$

Examples calculating FIRST sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$$FIRST(\varepsilon) = \{\varepsilon\} \qquad \text{rule 1}$$
$$FIRST(B) = FIRST(bB) \cup FIRST(\varepsilon) \qquad \text{rule 3}$$
$$= \{b, \varepsilon\}$$

Examples calculating FIRST sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$$FIRST(\varepsilon) = \{\varepsilon\} \quad \text{rule 1}$$
$$FIRST(B) = FIRST(bB) \cup FIRST(\varepsilon) \quad \text{rule 3}$$
$$= \{b, \varepsilon\} \quad \text{rules 1 and 2}$$
$$FIRST(C) =$$

Examples calculating FIRST sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$$FIRST(\varepsilon) = \{\varepsilon\} \quad \text{rule 1}$$
$$FIRST(B) = FIRST(bB) \cup FIRST(\varepsilon) \quad \text{rule 3}$$
$$= \{b, \varepsilon\} \quad \text{rules 1 and 2}$$
$$FIRST(C) = \{c, \varepsilon\} \quad \text{similarly}$$
$$FIRST(BC) =$$

Examples calculating FIRST sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$$FIRST(\varepsilon) = \{\varepsilon\} \qquad \text{rule 1}$$
$$FIRST(B) = FIRST(bB) \cup FIRST(\varepsilon) \qquad \text{rule 3}$$
$$= \{b, \varepsilon\} \qquad \text{rules 1 and 2}$$
$$FIRST(C) = \{c, \varepsilon\} \qquad \text{similarly}$$
$$FIRST(BC) = FIRST(B) \setminus \{\varepsilon\} \cup FIRST(C) \qquad \text{rule 4b}$$
$$=$$

Examples calculating FIRST sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

We want to calculate $FIRST(BC)$. We will need to calulate some other sets first to help us.

$$FIRST(\varepsilon) = \{\varepsilon\} \qquad \text{rule 1}$$
$$FIRST(B) = FIRST(bB) \cup FIRST(\varepsilon) \qquad \text{rule 3}$$
$$= \{b, \varepsilon\} \qquad \text{rules 1 and 2}$$
$$FIRST(C) = \{c, \varepsilon\} \qquad \text{similarly}$$
$$FIRST(BC) = FIRST(B) \setminus \{\varepsilon\} \cup FIRST(C) \qquad \text{rule 4b}$$
$$= \{b\} \cup \{c, \varepsilon\}$$
$$= \{b, c, \varepsilon\}$$

When rules can yield ε

Consider the grammar

$$S \to AB$$

$$A \to aA \mid \varepsilon$$

$$B \to bB \mid c$$

A can start with a or ε, i.e. $FIRST(A) = \{a, \varepsilon\}$

Suppose we are parsing the string bc, and so far we have derived $S \Rightarrow AB$. How does the parser know which production to use next, since b is not in $FIRST(A)$?

When rules can yield ε

Consider the grammar

$$S \to AB$$
$$A \to aA \mid \varepsilon$$
$$B \to bB \mid c$$

A can start with a or ε, i.e. $FIRST(A) = \{a, \varepsilon\}$

Suppose we are parsing the string bc, and so far we have derived $S \Rightarrow AB$. How does the parser know which production to use next, since b is not in $FIRST(A)$?

b is not in $FIRST(aA)$ or $FIRST(\varepsilon)$, but we can clearly see that using $A \to \varepsilon$ will give us $AB \Rightarrow B \Rightarrow bB \Rightarrow bc$

When rules can yield ε

Consider the grammar

$$S \to AB$$
$$A \to aA \mid \varepsilon$$
$$B \to bB \mid c$$

A can start with a or ε, i.e. $FIRST(A) = \{a, \varepsilon\}$

Suppose we are parsing the string bc, and so far we have derived $S \Rightarrow AB$. How does the parser know which production to use next, since b is not in $FIRST(A)$?

b is not in $FIRST(aA)$ or $FIRST(\varepsilon)$, but we can clearly see that using $A \to \varepsilon$ will give us $AB \Rightarrow B \Rightarrow bB \Rightarrow bc$

When a variable can derive ε, we need to look at the terminal symbols which can begin strings which could FOLLOW that variable in an derivation. These are called the FOLLOW sets.

Table construction: FOLLOW sets

Definition:

If A is a variable, then $FOLLOW(A)$ is the set of terminals which can be derived from any string which can appear immediately to the right of A in some stage of the derivation.

Construction rules:

Where α, β are strings of symbols, and S, X, Y are variables:

Table construction: FOLLOW sets

Definition:

If A is a variable, then $FOLLOW(A)$ is the set of terminals which can be derived from any string which can appear immediately to the right of A in some stage of the derivation.

Construction rules:

Where α, β are strings of symbols, and S, X, Y are variables:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
   (i.e. the start symbol can be followed by the end of the string)

Table construction: FOLLOW sets

Definition:

If A is a variable, then $FOLLOW(A)$ is the set of terminals which can be derived from any string which can appear immediately to the right of A in some stage of the derivation.

Construction rules:

Where α, β are strings of symbols, and S, X, Y are variables:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
   (i.e. the start symbol can be followed by the end of the string)
2. If $X \to \alpha Y$ then $FOLLOW(X) \subset FOLLOW(Y)$
   (i.e. anything that can follow X can follow Y)

Table construction: FOLLOW sets

Definition:

If A is a variable, then $FOLLOW(A)$ is the set of terminals which can be derived from any string which can appear immediately to the right of A in some stage of the derivation.

Construction rules:

Where α, β are strings of symbols, and S, X, Y are variables:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
   (i.e. the start symbol can be followed by the end of the string)
2. If $X \to \alpha Y$ then $FOLLOW(X) \subset FOLLOW(Y)$
   (i.e. anything that can follow X can follow Y)
3. If $X \to \alpha Y \beta$ then $FIRST(\beta) \setminus \{\varepsilon\} \subset FOLLOW(Y)$
   (i.e. any terminal which can start β can follow Y)

Table construction: FOLLOW sets

Definition:

If A is a variable, then $FOLLOW(A)$ is the set of terminals which can be derived from any string which can appear immediately to the right of A in some stage of the derivation.

Construction rules:

Where α, β are strings of symbols, and S, X, Y are variables:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
   (i.e. the start symbol can be followed by the end of the string)
2. If $X \to \alpha Y$ then $FOLLOW(X) \subset FOLLOW(Y)$
   (i.e. anything that can follow X can follow Y)
3. If $X \to \alpha Y \beta$ then $FIRST(\beta) \setminus \{\varepsilon\} \subset FOLLOW(Y)$
   (i.e. any terminal which can start β can follow Y)
4. If $X \to \alpha Y \beta$, $\varepsilon \in FIRST(\beta)$ then
   $FOLLOW(X) \subset FOLLOW(Y)$
   (i.e. if X can derive a string ending in Y, anything that follows X can follow Y)

Examples calculating FOLLOW sets

$$S \to BC \mid a$$

$$B \to bB \mid \varepsilon$$

$$C \to cC \mid \varepsilon$$

$FOLLOW(S) =$

Examples calculating FOLLOW sets

$$S \to BC \mid a$$

$$B \to bB \mid \varepsilon$$

$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) =$

Examples calculating FOLLOW sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) = FOLLOW(S)$ only rule 4 applied

$= \{\$\}$

Examples calculating FOLLOW sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) = FOLLOW(S)$ only rule 4 applied

$= \{\$\}$

$FOLLOW(B) =$

Examples calculating FOLLOW sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) = FOLLOW(S)$ only rule 4 applied

$= \{\$\}$

$FOLLOW(B) = FIRST(C) \setminus \{\varepsilon\}$ rule 3

Examples calculating FOLLOW sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) = FOLLOW(S)$ only rule 4 applied

$= \{\$\}$

$FOLLOW(B) = FIRST(C) \setminus \{\varepsilon\}$ rule 3

$\cup\ FOLLOW(C)$ rule 4

$=$

Examples calculating FOLLOW sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) = FOLLOW(S)$ only rule 4 applied

$= \{\$\}$

$FOLLOW(B) = FIRST(C) \setminus \{\varepsilon\}$ rule 3

$\cup\, FOLLOW(C)$ rule 4

$= \{c\} \cup \{\$\}$

$=$

Examples calculating FOLLOW sets

$$S \to BC \mid a$$
$$B \to bB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

$FOLLOW(S) = \{\$\}$ only rule 1 applied

$FOLLOW(C) = FOLLOW(S)$ only rule 4 applied

$= \{\$\}$

$FOLLOW(B) = FIRST(C) \setminus \{\varepsilon\}$ rule 3

$\cup\, FOLLOW(C)$ rule 4

$= \{c\} \cup \{\$\}$

$= \{c, \$\}$

Examples calculating FOLLOW sets

$$B \to (B)B \mid \varepsilon$$

$$FOLLOW(B) =$$

Examples calculating FOLLOW sets

$$B \to (B)B \mid \varepsilon$$

$$FOLLOW(B) = \{\$\} \qquad \text{rule 1}$$

$$\cup\, FIRST(\text{“}\,)B\text{”}) \setminus \{\varepsilon\} \qquad \text{rule 3}$$

$$= \$$$

Examples calculating FOLLOW sets

$$B \to (B)B \mid \varepsilon$$

$$\begin{aligned} FOLLOW(B) &= \{\$\} && \text{rule 1} \\ &\quad \cup FIRST(\text{“})B\text{”}) \setminus \{\varepsilon\} && \text{rule 3} \\ &= \{\$\} \cup \{)\} \\ &= \{\$,)\} \end{aligned}$$

Constructing the parse table

Rows: one for each variable of the grammar

Columns: one for each terminal of the grammar, and for the end of string marker \$

Steps to fill the table T:

1. If there is a rule $R \to \alpha$ with $a \in FIRST(\alpha)$, then put α in $T[R, a]$
2. If there is a rule $R \to \alpha$ with $\varepsilon \in FIRST(\alpha)$ and $a \in FOLLOW(R)$, then put α in $T[R, a]$

Example

$F \to \alpha$	$FIRST(\alpha)$	$FOLLOW(F)$ if $\varepsilon \in FIRST(\alpha)$
$S \to BC$		
$S \to a$		
$B \to bB$		
$B \to \varepsilon$		
$C \to cC$		
$C \to \varepsilon$		

Parse table:

	a	b	c	\$
S				
B				
C				

Example

$F \to \alpha$	$FIRST(\alpha)$	$FOLLOW(F)$ if $\varepsilon \in FIRST(\alpha)$
$S \to BC$	$FIRST(BC) = \{b, c, \varepsilon\}$	
$S \to a$	$FIRST(a) = \{a\}$	
$B \to bB$	$FIRST(bB) = \{b\}$	
$B \to \varepsilon$	$FIRST(\varepsilon) = \{\varepsilon\}$	
$C \to cC$	$FIRST(cC) = \{c\}$	
$C \to \varepsilon$	$FIRST(\varepsilon) = \{\varepsilon\}$	

Parse table:

	a	b	c	\$
S				
B				
C				

Example

$F \to \alpha$	$FIRST(\alpha)$	$FOLLOW(F)$ if $\varepsilon \in FIRST(\alpha)$
$S \to BC$	$FIRST(BC) = \{b, c, \varepsilon\}$	$FOLLOW(S) = \{\$\}$
$S \to a$	$FIRST(a) = \{a\}$	
$B \to bB$	$FIRST(bB) = \{b\}$	
$B \to \varepsilon$	$FIRST(\varepsilon) = \{\varepsilon\}$	$FOLLOW(B) = \{c, \$\}$
$C \to cC$	$FIRST(cC) = \{c\}$	
$C \to \varepsilon$	$FIRST(\varepsilon) = \{\varepsilon\}$	$FOLLOW(C) = \{\$\}$

Parse table:

	a	b	c	$\$$
S				
B				
C				

Example

$F \to \alpha$	$FIRST(\alpha)$	$FOLLOW(F)$ if $\varepsilon \in FIRST(\alpha)$
$S \to BC$	$FIRST(BC) = \{b, c, \varepsilon\}$	$FOLLOW(S) = \{\$\}$
$S \to a$	$FIRST(a) = \{a\}$	
$B \to bB$	$FIRST(bB) = \{b\}$	
$B \to \varepsilon$	$FIRST(\varepsilon) = \{\varepsilon\}$	$FOLLOW(B) = \{c, \$\}$
$C \to cC$	$FIRST(cC) = \{c\}$	
$C \to \varepsilon$	$FIRST(\varepsilon) = \{\varepsilon\}$	$FOLLOW(C) = \{\$\}$

Parse table:

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing with a parsing table

1. Append the end of input marker \$ to the input and push \$ to the stack.
2. Put the *start variable* on the stack and scan the *first token*
3. Repeat the following:
   - 3.1 If *the top of the stack is a variable symbol* V, and the current token is a, then pop V and push the string from the table entry (V, a). If the entry was empty, reject the input.
   - 3.2 else *if the top of the stack is a terminal symbol* t, compare t to a. If they match, pop the stack and scan the next token. Otherwise reject the input.
   - 3.3 else *if the top of the stack and the token are both \$*, then accept the input (the stack is empty and we have used all the input.)
   - 3.4 else reject the input (the stack is empty but there is unread input.)

Parsing *bcc*

Remaining input	Stack

	a	b	c	\$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing bcc

Remaining input	Stack
$bcc\$$	$S\$$
$bcc\$$	$BC\$$
$bcc\$$	$bBC\$$
$cc\$$	$BC\$$
$cc\$$	$C\$$
$cc\$$	$cC\$$
$c\$$	$C\$$
$c\$$	$cC\$$
$\$$	$C\$$
$\$$	$\$$
ACCEPTED	

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing $bcbc$

Remaining input	Stack

	a	b	c	\$
S	a	bC	BC	BC
B		bB	ε	ε
C			cC	ε

Parsing $bcbc$

Remaining input	Stack
$bcbc\$$	$S\$$
$bcbc\$$	$BC\$$
$bcbc\$$	$bBC\$$
$cbc\$$	$BC\$$
$cbc\$$	$C\$$
$cbc\$$	$cC\$$
$bc\$$	$C\$$
REJECTED	

	a	b	c	$\$$
S	a	BC	BC	BC
B		bB	ε	ε
C			cC	ε

LL(k) parsing

LL(k) grammar

- L: Left to right scanning of input
- L: Leftmost derivation
- k: uses k look ahead symbols

Deterministic derivation by looking ahead k symbols

Using less lookahead symbols is usually more efficient

LL(1) grammar: example

$$L = \{a^n bc^n \mid n \geq 0\}$$

$$S \to aSc \mid b$$

This grammar is LL(1), because the right side of each production rule lead to strings beginning with different letters

Each step of the derivation can be deterministically determined by examining the current symbol (1 lookahead symbol)

- If the remaining input starts with a, use $S \to aSc$
- If the remaining input starts with b, use $S \to b$
- (If it starts with anything else, no derivation exists)

LL(1) grammar: example

$L = \{a^n b c^n \mid n \geq 0\}$

$$S \to aSc \mid b$$

Derivation of $aabcc$

$S \Rightarrow aSc$ Use $S \to aSc$ because $aabcc$ begins with a

$\Rightarrow aaScc$ Use $S \to aSc$ because $abcc$ begins with a

$\Rightarrow aabcc$ Use $S \to b$ because bcc begins with b

LL(2) grammar: example

$$L = \{a^m b^n c \mid n \geq 0\}$$

$$S \to AB$$

$$A \to aA \mid a$$

$$B \to bB \mid c$$

Each step of the derivation can be deterministically determined by examining the current symbol and the next one (2 lookahead symbols). e.g. if we need to replace a variable A and:

- If the remaining input starts with aa, use $A \to aA$
- If the remaining input starts with ab or ac, use $A \to a$
- (If it starts with anything else, no derivation exists)

LL(2) grammar: example

$$S \to AB$$
$$A \to aA \mid a$$
$$B \to bB \mid c$$

Derivation of $aabbc$

$S \Rightarrow AB$	No other choice
$\Rightarrow aAB$	Use $A \to aA$ because $aabbc$ begins with aa
$\Rightarrow aaB$	Use $A \to a$ because $abbc$ begins with ab
$\Rightarrow aabB$	Use $B \to bB$ because bbc begins with b
$\Rightarrow aabbB$	Use $B \to bB$ because bc begins with b
$\Rightarrow aabbc$	Use $B \to c$ because c begins with c

non-LL(k) grammar: example

$$S \to aS \mid T$$

$$T \to aTb \mid \varepsilon$$

Not LL(1): next symbol a is not enough to determine which production to use ($S \to aS$ and $S \to T$ can both generate strings starting with a)

Not LL(2): the input aa is not enough either

Not LL(k): we need to know how many b's there are. For any k we can choose $n > k$ such that $a^n b^n \in L(G)$ but we would need to lookahead $2n > k$ symbols to decide which rule to use first.

Identify if a grammar is LL(1)

Recall that a grammar is LL(1) if it is sufficient to look at the next symbol to determine which rule to follow next.

i.e. If every cell in the LL(1) parse table contains at most one rule, then the grammar is LL(1)

More formally, a grammar is LL(1) iff for every variable A:

- Let $A \to \alpha_1 \mid \ldots \mid \alpha_n$ be the production rules for A
- Let $X_i = FIRST(\alpha_i)$ if $\varepsilon \notin FIRST(\alpha_i)$
- Let $X_i = FIRST(\alpha_i) \cup FOLLOW(A)$ otherwise
- Then $X_i \cap X_j = \emptyset$ for all $i \neq j$

Transforming non-LL(1) grammars

When a grammar is not LL(1) we try to find an equivalent grammar which is, by applying the following techniques:

- Left factoring
- Elimination of left recursion

Transforming non-LL(1) grammars

When a grammar is not LL(1) we try to find an equivalent grammar which is, by applying the following techniques:

- Left factoring
- Elimination of left recursion

Recall: grammars are *equivalent* if they generate the same language

Transforming non-LL(1) grammars

When a grammar is not LL(1) we try to find an equivalent grammar which is, by applying the following techniques:

- Left factoring
- Elimination of left recursion

Recall: grammars are *equivalent* if they generate the same language

Such a grammar does not always exist. For example no LL(k) grammar exists for the language
$\{a^n b^n \mid n \geq 0\} \cup \{a^n b^{2n} \mid n \geq 0\}$

Left factoring: why?

Consider the grammar fragment $S \rightarrow abcC \mid abdD$

- The two rules both start with the same prefix ab
- i.e. their FIRST sets both include a
- The LL(1) parse table will have multiple entries at (S, a).

Left factoring: why?

Consider the grammar fragment $S \to abcC \mid abdD$

- The two rules both start with the same prefix ab
- i.e. their FIRST sets both include a
- The LL(1) parse table will have multiple entries at (S, a).

We can "factor out" the string ab to obtain an equivalent grammar, where B is a new variable:

$$S \to abB$$

$$B \to cC \mid dD$$

This grammar fragment is equivalent, but is LL(1)

Left Factoring example

Ambiguous grammar:

$$S \to abC \mid abD$$

$$\ldots \to \ldots$$

After factorisation

$$S \to abB$$

$$B \to cC \mid dD$$

$$\ldots \to \ldots$$

	a	b	c	d
S	abC abD			
...	...	...	...	...

	a	b	c	d
S	abB			
B			cC	dD
...	...	...	...	...

Left factoring: definition

If a string w appears on the left of several rules for a variable A:

$$A \to wX_1 \mid \ldots \mid wX_1$$

Then we can factor out w and introduce a new variable A':

$$A \to wA'$$
$$A' \to X_1 \mid \ldots \mid X_n$$

Any other rules produced by A are unaffected.

Recursion (from last week)

If a variable X can generate a string containing X itself, then it is recursive

- left-recursive: it occurs at the start of the string $X \Rightarrow^+ X\beta$
- right-recursive: it occurs at the end of the string $X \Rightarrow^+ \alpha X$
- self-embedding: it occurs in between: $X \Rightarrow^+ \alpha X \beta$

A grammar is recursive if any of its variables is recursive

A grammar for an infinite language must contain at least one recursive variable

Eliminate left recursion: why?

Consider this simple grammar:

$$A \rightarrow c$$

$$A \rightarrow Ab$$

$$FIRST(c) = \{c\}$$

$$FIRST(Ab) = \{c\}$$

Eliminate left recursion: why?

Consider this simple grammar:

$$A \rightarrow c$$

$$A \rightarrow Ab$$

$$FIRST(c) = \{c\}$$

$$FIRST(Ab) = \{c\}$$

If we try to construct the parse table:

	b	c	$\$$
A		c or Ab	

Eliminate left recursion: why?

Consider this simple grammar:

$$A \rightarrow c$$

$$A \rightarrow Ab$$

$$FIRST(c) = \{c\}$$

$$FIRST(Ab) = \{c\}$$

If we try to construct the parse table:

	b	c	$\$$
A		c or Ab	

The base cases for the recursion must have FIRST sets which intersect with the left recursive rule!

Eliminating left recursion

Let α, β be arbitrary strings of terminals and/or variables.

Let A be a variable, and R a new variable

If A has left recursive rules:

$$A \to Ac \mid \beta$$

It can be replaced with:

$$A \to \beta R$$
$$R \to \alpha R \mid \varepsilon$$

Eliminating left recursion

Let α, β be arbitrary strings of terminals and/or variables.

Let A be a variable, and R a new variable

If A has left recursive rules:

$$A \to A\alpha \mid \beta$$

It can be replaced with:

$$A \to \beta R$$
$$R \to \alpha R \mid \varepsilon$$

What do the parse trees look like for $\beta\alpha\alpha\alpha$ using the original and transformed grammar?

Simple example

$$A \to c$$

$$A \to Ab$$

Then $\alpha =$

Simple example

$$A \to c$$

$$A \to Ab$$

Then $\alpha = b$, $\beta =$.

Simple example

$$A \to c$$

$$A \to Ab$$

Then $\alpha = b$, $\beta = c$, which gives us:

$$A \to cR$$

$$R \to bR \mid \varepsilon$$

Simple example

$$A \to c$$

$$A \to Ab$$

Then $\alpha = b$, $\beta = c$, which gives us:

$$A \to cR$$

$$R \to bR \mid \varepsilon$$

	b	c	\$
A		cR	
R	bR		ε

Complex example

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow a \mid b \mid c$$

Then $\alpha =$

Complex example

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow a \mid b \mid c$$

Then $\alpha = \ + T \mid -T$, $\beta =$

Complex example

$$E \to E + T$$
$$E \to E - T$$
$$E \to T$$
$$T \to a \mid b \mid c$$

Then $\alpha = +T \mid -T$, $\beta = T$, which gives us:

$$E \to TR$$
$$R \to +TR \mid -TR \mid \varepsilon$$
$$T \to a \mid b \mid c$$

Complex example

$$E \to TR$$

$$R \to +TR \mid -TR \mid \varepsilon$$

$$T \to a \mid b \mid c$$

$$FIRST(TR) =$$

Complex example

$$E \to TR$$

$$R \to +TR \mid -TR \mid \varepsilon$$

$$T \to a \mid b \mid c$$

$FIRST(TR) = \{a, b, c\}$

$FIRST(+TR) =$

Complex example

$$E \to TR$$

$$R \to +TR \mid -TR \mid \varepsilon$$

$$T \to a \mid b \mid c$$

$FIRST(TR) = \{a, b, c\}$

$FIRST(+TR) = \{+\}$

$FIRST(-TR) =$

Complex example

$$E \to TR$$

$$R \to +TR \mid -TR \mid \varepsilon$$

$$T \to a \mid b \mid c$$

$FIRST(TR) = \{a, b, c\}$

$FIRST(+TR) = \{+\}$

$FIRST(-TR) = \{-\}$

Because $\varepsilon \in FIRST(\varepsilon)$, we calculate $FOLLOW(R) =$

Complex example

$$E \to TR$$

$$R \to +TR \mid -TR \mid \varepsilon$$

$$T \to a \mid b \mid c$$

$FIRST(TR) = \{a, b, c\}$

$FIRST(+TR) = \{+\}$

$FIRST(-TR) = \{-\}$

Because $\varepsilon \in FIRST(\varepsilon)$, we calculate $FOLLOW(R) = \{\$\}$

Complex example

$$E \to TR$$

$$R \to +TR \mid -TR \mid \varepsilon$$

$$T \to a \mid b \mid c$$

$FIRST(TR) = \{a, b, c\}$

$FIRST(+TR) = \{+\}$

$FIRST(-TR) = \{-\}$

Because $\varepsilon \in FIRST(\varepsilon)$, we calculate $FOLLOW(R) = \{\$\}$

	a	b	c	$+$	$-$	\$
E	TR	TR	TR			
R				$+TR$	$-TR$	ε
T	a	b	c			

Proving that a grammar is *not* LL(1)

It is sufficient to show any one of the following:

- The grammar is left recursive

- The grammar needs left factoring

- The first sets of the production rules for a variable are not disjoint

Typical exam question

Consider the grammar G:

$$S \to ST \mid ab$$

$$T \to aTbb \mid ab$$

Show that the grammar G is not LL(1)

Transform G to obtain a grammar G' which is LL(1)

Give the LL(1) parse table for G'

Push-down Automata

- "NFA with a stack"
- CFG to PDA construction method
- Recognises the set of CFL
- Non-deterministic PDA are *more powerful* than D-PDA

Parsing

- LL(k) parsers look ahead up to k symbols
- Not all CFG are LL(k)
- $FIRST(\alpha)$: set of terminals (or ε) which start strings derived from α
- $FOLLOW(X)$: the set of terminals (or \$) which could start strings following X in a derivation
- How to build a parse table for an LL(1) CFG
- How to parse a string using an LL(1) parse table

Announcements

- Assignment 2
  - Will be on this topic (parsing)
  - Due Sunday 14th October (end of week 10)
  - Released this weekend

- Monday 1st October is a public holiday
  - Alternative tutorials for COMP2022 Monday students:
    - Tuesday 3pm in ABS 3100
    - Wednesday 9am in ABS 3090
      - Select a session here:
        `https://edstem.org/courses/2892/sway/`
  - COMP2922: normal tutorial covered in advanced session