COMP2022: Formal Languages and Logic

2017, Semester 1, Week 5

Joseph Godbehere

Adapted from slides by A/Prof Kalina Yacef

April 4, 2017

THE UNIVERSITY OF SYDNEY

Announcements

Mid-semester survey

- On eLearning, until Friday
- Help us improve the course for you
- Anonymous

Quizzes!

- Content: always similar to questions from the prior tutorial
- How to do well: revise the tutorial content

Outline

- Revision
- Push Down Automata
- LL(k) Table-Descent Parsing

Revision

A grammar G is sets of variables, terminals, production rules, and a start variable.

The production rules in *context-free grammars* always have the form $A \Rightarrow \alpha$, where A is a variable and α is a string of variables and/or terminals (or ε)

w can be derived by G if there exists a sequence of substitutions such that $S \Rightarrow^{+} w$

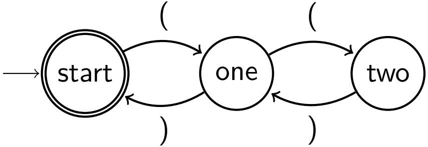
- Leftmost derivation: substitute the leftmost *variable* in w
- Rightmost derivation: substitute the rightmost *variable* in w

Language of G is the set of strings which can be derived from S

Ambiguous strings have multiple parse trees (equivalently, multiple left derivations, or multiple right derivations.)

Limits of DFA/NFA

Recall this NFA for balanced parentheses:



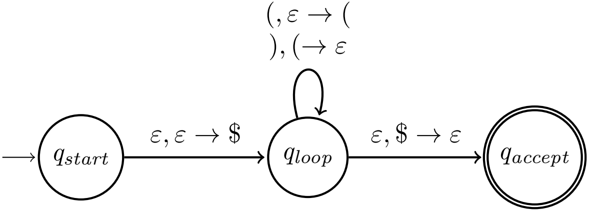
It accepts strings of balanced parentheses, nested up to two deep.

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

Parsing ((()))



$(, \varepsilon \rightarrow ($
$), (\rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \$$
$\varepsilon, \$ \rightarrow \varepsilon$
q_{start}
q_{loop}
q_{accept}

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

Push Down Automata

Finite automata *with a stack*.

Revision:
A *stack* is a structure with a LIFO property (Last In, First Out)

Stack operations:

- PEEK: read the top symbol without removing it
- POP: read the top symbol and remove it
- PUSH: write a symbol onto the top of the stack
- NOP: perform no operation on the stack

Push Down Automata

Finite automata *with a stack*.

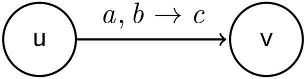
A PDA is a machine with the ability to:

- 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:

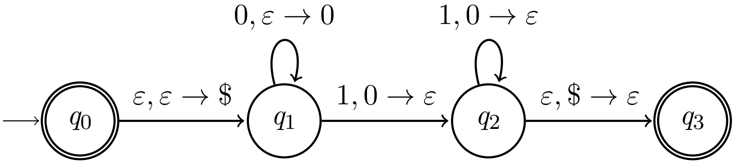
- $a, \varepsilon \rightarrow c$
  - read (and remove) a from the front of the input
  - do not pop anything from the stack
  - push c onto the stack
- $\varepsilon, \varepsilon \rightarrow c$
  - do not read anything from the input
  - do not pop anything from the stack
  - push c onto the stack

PDA Transitions

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

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

Example: PDA M_2 for $\{0^n1^n \mid n \geq 0\}$



1. Initialise the stack with $\$$
2. As each 0 is scanned, push it onto the stack
3. As each 1 is scanned, pop a 0 from the stack
4. If all the 0's have been popped, $\$$ will be on top the stack. If so, pop it and transition to q_3
5. If we have read all the input, accept the string

PDA: formal definition

A pushdown automata is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

1. Q is a finite set called the states
2. Σ is a finite set called the input alphabet
3. Γ is a finite set called the stack alphabet
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
5. $q_0 \in Q$ is the start state
6. $F \subseteq Q$ is the set of accept states

PDA: formal description of M_2

1. $Q = \{q_0, q_1, q_2, q_3\}$
2. $\Sigma = \{0, 1\}$
3. $\Gamma = \{0, \$\}$
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is

	ε	0	\$
q_0, ε	$\{(q_1, \$)\}$		
$q_1, 0$	$\{(q_1, 0)\}$		
$q_1, 1$		$\{(q_2, \varepsilon)\}$	
$q_2, 1$		$\{(q_2, \varepsilon)\}$	
q_2, ε			$\{(q_3, \varepsilon)\}$

5. q_0 is the start state
6. $F = \{q_0, q_3\}$

Language recognised by a PDA

A string is *accepted* by a PDA if there is some path from the start state to a final state, which consumes all the letters of the string. Otherwise the string is *rejected*.

The language *recognised* by a PDA is the set of strings which it accepts.

Language recognised by a PDA

Formal definition:
Let M be a PDA and $w = w_1 w_2 ... w_n$ be a string in $\Sigma^\star$. M accepts w if and only if there exists a string path:
$(r_0, s_0) \rightarrow_{a_1} (r_1, s_1) \rightarrow_{a_2} ... \rightarrow_{a_n} (r_n, s_n)$ such that:

1. $r_0 = q_0$ and $s_0 = \varepsilon$
2. $\delta(r_i, a_{i+1}, b) = (r_{i+1}, c)$ for $i = 0, ..., n-1$ where $s_i = bt$ and $s_{i+1} = ct$ with $t \in \Gamma^\star$
3. $r_n \in F$

Language recognised by a PDA

Formal definition:
Let M be a PDA and $w = w_1 w_2 ... w_n$ be a string in $\Sigma^\star$. M accepts w if and only if there exists a string path:
$(r_0, s_0) \rightarrow_{a_1} (r_1, s_1) \rightarrow_{a_2} ... \rightarrow_{a_n} (r_n, s_n)$ such that:

1. $r_0 = q_0$ and $s_0 = \varepsilon$
2. $\delta(r_i, a_{i+1}, b) = (r_{i+1}, c)$ for $i = 0, ..., n-1$ where $s_i = bt$ and $s_{i+1} = ct$ with $t \in \Gamma^\star$
3. $r_n \in F$

- This is slightly oversimplified! Actually we just need there to be some sequence of transitions which only consume one input symbol (i.e. we can also follow as many $\delta(\varepsilon, x, y)$ transitions as needed.)

Equivalent forms of acceptance

Final-state acceptance:
A string is accepted if there is some path from the start state to a final state which consumes all of the letters of the string. This is the most common definition, and the one we used earlier.

Empty-stack acceptance:
A string is accepted if the input string has been consumed and the stack is empty (no requirement about the machine being in any particular state.)

Determinism

A PDA is *deterministic* if for all states there is at most one possible move, given the unscanned input and stack contents.

More formally, suppose that:

- The PDA is in state q
- The next input symbol is a
- The symbol on top of the stack is b

Then we can transition to any pair (next state, symbol to push) in:

$$\delta(q, a, b) \cup \delta(q, \varepsilon, b) \cup \delta(q, a, \varepsilon) \cup \delta(q, \varepsilon, \varepsilon)$$

The PDA is *non-deterministic* if this set contains more than one pair.

Determinism

DFA and NFA have the same recognition power

Non-deterministic PDA are *more* powerful than deterministic PDA.

- Some context-free languages can be recognised by a non-deterministic PDA but *not* by a deterministic one.
    - Example: Language of palindromes (why?)

Context-free grammars and PDA

Theorem:
Context-free languages are exactly the languages accepted by PDAs

Proof:

1. For any PDA, there is a CFG generating the same language
   - We will not show this construction (Sipser 2.20 – 2.31)
   - The method usually produces very large grammars
2. For any CFG, there is a PDA recognising the same language
   - We will explore this algorithm
   - The PDA non-deterministically performs a leftmost derivation

CFG → PDA algorithm

The PDA will non-deterministically attempt a leftmost derivation.

Push the empty stack marker $, then the start symbol on the stack.

Repeat the following steps:

CFG → PDA algorithm

The PDA will non-deterministically attempt a leftmost derivation.

Push the empty stack marker $, then the start symbol on the stack.

Repeat the following steps:

- If the top of the stack is a variable A, then non-deterministically choose one of the rules for A. Substitute A on the stack with the string produced by the rule.

CFG → PDA algorithm

The PDA will non-deterministically attempt a leftmost derivation.

Push the empty stack marker \$, then the start symbol on the stack.

Repeat the following steps:

- If the top of the stack is a variable A, then non-deterministically choose one of the rules for A. Substitute A on the stack with the string produced by the rule.
- If the top of the stack is a terminal symbol a, then read the next symbol from the input. If they match, pop the stack. Otherwise, reject this branch of the non-determinism.

CFG → PDA algorithm

The PDA will non-deterministically attempt a leftmost derivation.

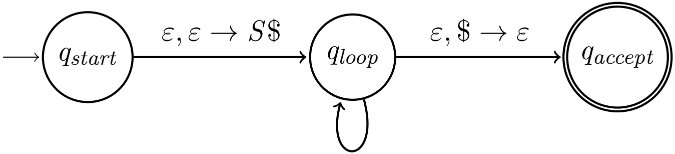
Push the empty stack marker \$, then the start symbol on the stack.

Repeat the following steps:

- If the top of the stack is a variable A, then non-deterministically choose one of the rules for A. Substitute A on the stack with the string produced by the rule.
- If the top of the stack is a terminal symbol a, then read the next symbol from the input. If they match, pop the stack. Otherwise, reject this branch of the non-determinism.
- If the top of the stack is \$, enter the accept state. This will accept the input if it has all been read.

CFG → PDA construction method

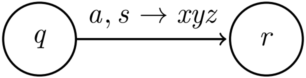
First build a "reduced PDA" as follows:


q_{start}
$\varepsilon, \varepsilon \to S\$$
q_{loop}
$\varepsilon, \$ \to \varepsilon$
q_{accept}

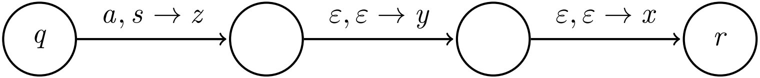
$\varepsilon, A \to \alpha$ for each rule $A \to \alpha$
$a, a \to \varepsilon$ for each terminal a

CFG → PDA construction method

Now expand the transitions in the “reduced PDA” to a normal one, such that only one symbol is push or popped to the stack in each transitions:



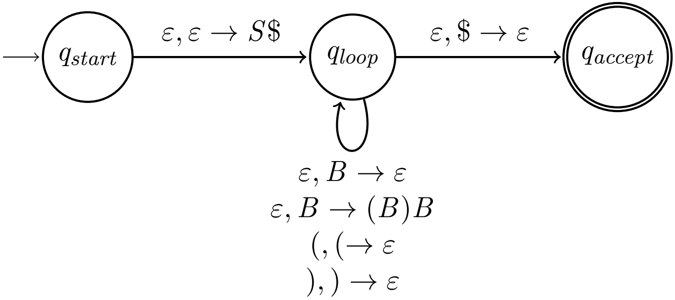
becomes:


q
$a, s \to z$
$\varepsilon, \varepsilon \to y$
$\varepsilon, \varepsilon \to x$
r

i.e. we push the string of symbols xyz onto the stack in reverse order, with each transition after the first one reading nothing from the input or the stack.

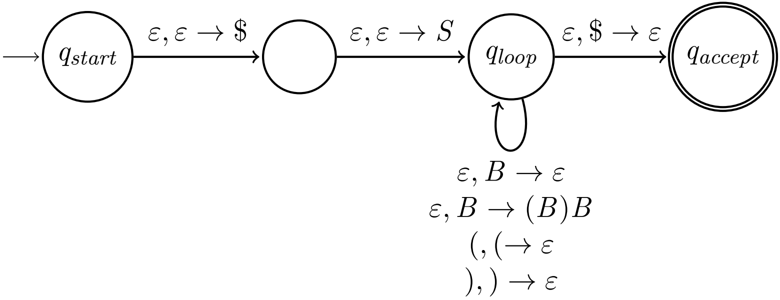
CFG → PDA example for $B \to (B)B \mid \varepsilon$

First build a “reduced PDA” as follows:


q_{start}
$\varepsilon, \varepsilon \to S\$$
q_{loop}
$\varepsilon, \$ \to \varepsilon$
q_{accept}
$\varepsilon, B \to \varepsilon$
$\varepsilon, B \to (B)B$
$(, (\to \varepsilon$
$),) \to \varepsilon$

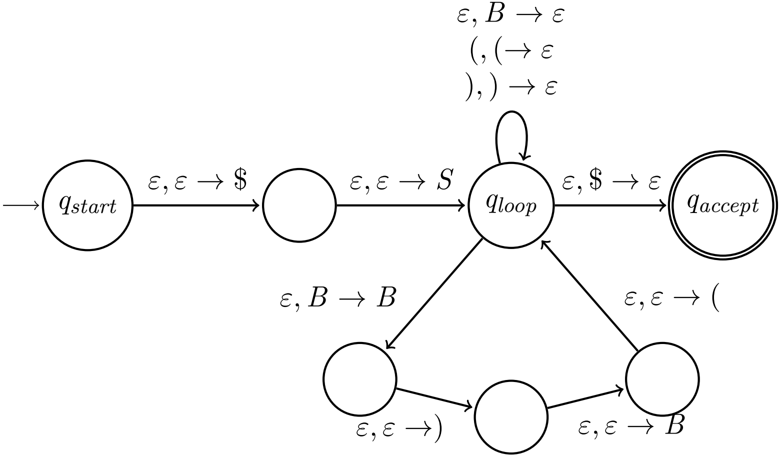
CFG → PDA example for $B \to (B)B \mid \varepsilon$

Expand $\varepsilon, \varepsilon \to S\$$


q_{start}
$\varepsilon, \varepsilon \to \$$
$\varepsilon, \varepsilon \to S$
q_{loop}
$\varepsilon, \$ \to \varepsilon$
q_{accept}
$\varepsilon, B \to \varepsilon$
$\varepsilon, B \to (B)B$
$(, (\to \varepsilon$
$),) \to \varepsilon$

CFG → PDA example for $B \to (B)B \mid \varepsilon$

Expand $\varepsilon, B \to (B)B$


$\varepsilon, B \to \varepsilon$
$(, (\to \varepsilon$
$),) \to \varepsilon$
q_{start}
$\varepsilon, \varepsilon \to \$$
$\varepsilon, \varepsilon \to S$
q_{loop}
$\varepsilon, \$ \to \varepsilon$
q_{accept}
$\varepsilon, B \to B$
$\varepsilon, \varepsilon \to ($
$\varepsilon, \varepsilon \to)$
$\varepsilon, \varepsilon \to B$

Grammars are fundamental for constructing parsers.

Parsers should be efficient, so determinism is important.

Top-down parsing:

Bottom-up parsing:

Grammars are fundamental for constructing parsers.

Parsers should be efficient, so determinism is important.

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

Bottom-up parsing:

Grammars are fundamental for constructing parsers.

Parsers should be efficient, so determinism is important.

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

Bottom-up parsing: The parse tree is constructed starting from the leaves, working up towards the root

- Constructs the derivation by starting with the string, and working backwards to the start symbol. i.e. $w \Leftarrow ... \Leftarrow S$
- LR(k) parsing
  - **L**eft-to-right, **R**ightmost derivation in reverse

From CFG → PDA → Table Driven parsers

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

The stack contains right-hand side of the production rules, in an order which allows a left-most derivation.

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?

We will see how to construct a deterministic table whenever possible

- Not all CFG have an equivalent *deterministic* PDA

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 $S \Rightarrow YZ$

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

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

$$S \rightarrow aSc \mid b$$

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

LL(1) grammar: example

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

$$S \rightarrow aSc \mid b$$

Derivation of $aabcc$

$$\begin{aligned} S &\Rightarrow aSc && \text{Use } S \rightarrow aSc \text{ because } aabcc \text{ begins with } a \\ &\Rightarrow aaScc && \text{Use } S \rightarrow aSc \text{ because } abcc \text{ begins with } a \\ &\Rightarrow aabcc && \text{Use } S \rightarrow b \text{ because } bcc \text{ begins with } b \end{aligned}$$

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

$$\begin{aligned} S &\to AB \\ A &\to aA \mid a \\ B &\to bB \mid c \end{aligned}$$

Derivation of $aabbc$

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

non-LL(k) grammar: example

$$S \to aS \mid T$$
$$T \to aBb \mid \varepsilon$$

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.

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] = $\alpha$ is defined then
        pop T and push $\alpha$ onto the stack
                //(in reverse order)
    else error
endloop
```

Rules starting with non-terminals

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aA \mid \varepsilon \\
B &\to bB \mid c
\end{aligned}
$$

It's clear that the productions for A and B are LL(1), but how do 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. Iff $\alpha \Rightarrow^{+} \varepsilon$ 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. Iff $\alpha \Rightarrow^{+} \varepsilon$ 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)

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. Iff $\alpha \Rightarrow^{+} \varepsilon$ 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. Iff $\alpha \Rightarrow^{+} \varepsilon$ 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 $A \rightarrow \alpha_1 \mid ... \mid \alpha_n$ then
   $FIRST(A) = FIRST(\alpha_1) \cup ... \cup FIRST(\alpha_n)$

Table construction: FIRST sets

If α is a string, then $FIRST(\alpha)$ is the set of terminals which can begin strings derived from α. Iff $\alpha \Rightarrow^{+} \varepsilon$ 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 $A \to \alpha_1 \mid ... \mid \alpha_n$ then $FIRST(A) = FIRST(\alpha_1) \cup ... \cup FIRST(\alpha_n)$
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

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

$FIRST(\text{“}(B)B\text{”}) =$

Examples calculating FIRST sets

$$B \rightarrow (B)B \mid \varepsilon$$

$$FIRST(``(B)B") = \{(\}$$ rule 2

$$FIRST(B) =$$

Examples calculating FIRST sets

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

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

Examples calculating FIRST sets

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

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

Examples calculating FIRST sets

$$B \rightarrow (B)B \mid \varepsilon$$

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

Examples calculating FIRST sets

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

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

$FIRST(\varepsilon) =$

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.

$FIRST(\varepsilon) = \{\varepsilon\}$ 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.

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

Examples calculating FIRST sets

$$
\begin{aligned}
S &\to BC \mid a \\
B &\to bB \mid \varepsilon \\
C &\to cC \mid \varepsilon
\end{aligned}
$$

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

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

Examples calculating FIRST sets

$$\begin{aligned} S &\to BC \mid a \\ B &\to bB \mid \varepsilon \\ C &\to cC \mid \varepsilon \end{aligned}$$

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

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

Examples calculating FIRST sets

$$
\begin{aligned}
S &\to BC \mid a \\
B &\to bB \mid \varepsilon \\
C &\to cC \mid \varepsilon
\end{aligned}
$$

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

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

Examples calculating FIRST sets

$$\begin{aligned} S &\to BC \mid a \\ B &\to bB \mid \varepsilon \\ C &\to cC \mid \varepsilon \end{aligned}$$

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

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

When rules can yield ε

Consider the grammar

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

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

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?

When rules can yield ε

Consider the grammar

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

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

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?

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

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

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

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?

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 $X \rightarrow \alpha Y$ then $FOLLOW(X) \subset FOLLOW(Y)$

   (i.e. anything that can follow Y can follow X)

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 Y can follow X)
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 Y can follow X)
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

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

$$\begin{aligned} FOLLOW(S) &= \{\$\} && \text{only rule 1 applied} \\ FOLLOW(C) &= FOLLOW(S) && \text{only rule 4 applied} \\ &= \end{aligned}$$

Examples calculating FOLLOW sets

$$
\begin{aligned}
S &\to BC \mid a \\
B &\to bB \mid \varepsilon \\
C &\to cC \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
FOLLOW(S) &= \{\$\} && \text{only rule 1 applied} \\
FOLLOW(C) &= FOLLOW(S) && \text{only rule 4 applied} \\
&= \{\$\} \\
FOLLOW(B) &=
\end{aligned}
$$

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

$$
\begin{aligned}
S &\to BC \mid a \\
B &\to bB \mid \varepsilon \\
C &\to cC \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
FOLLOW(S) &= \{\$\} && \text{only rule 1 applied} \\
FOLLOW(C) &= FOLLOW(S) && \text{only rule 4 applied} \\
&= \{\$\} \\
FOLLOW(B) &= FIRST(C) \setminus \{\varepsilon\} && \text{rule 3} \\
&\quad \cup FOLLOW(C) && \text{rule 4} \\
&=
\end{aligned}
$$

Examples calculating FOLLOW sets

$$
\begin{aligned}
S &\to BC \mid a \\
B &\to bB \mid \varepsilon \\
C &\to cC \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
FOLLOW(S) &= \{\$\} && \text{only rule 1 applied} \\
FOLLOW(C) &= FOLLOW(S) && \text{only rule 4 applied} \\
&= \{\$\} \\
FOLLOW(B) &= FIRST(C) \setminus \{\varepsilon\} && \text{rule 3} \\
&\quad \cup FOLLOW(C) && \text{rule 4} \\
&= \{c\} \cup \{\$\} \\
&=
\end{aligned}
$$

EXAMPLES CALCULATING FOLLOW SETS

$$
\begin{aligned}
S &\to BC \mid a \\
B &\to bB \mid \varepsilon \\
C &\to cC \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
FOLLOW(S) &= \{\$\} && \text{only rule 1 applied} \\
FOLLOW(C) &= FOLLOW(S) && \text{only rule 4 applied} \\
&= \{\$\} \\
FOLLOW(B) &= FIRST(C) \setminus \{\varepsilon\} && \text{rule 3} \\
&\quad \cup FOLLOW(C) && \text{rule 4} \\
&= \{c\} \cup \{\$\} \\
&= \{c, \$\}
\end{aligned}
$$

Examples calculating FOLLOW sets

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

$$FOLLOW(B) =$$

Examples calculating FOLLOW sets

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

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

Examples calculating FOLLOW sets

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

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

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 $R \rightarrow \alpha$ with $a \in FIRST(\alpha)$ then put α in $T[R, a]$
2. If there is a rule $R \rightarrow \alpha$ 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

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