Parsing and CFG vs RE

Narges Khakpour

Department of Computer Science,
Linnaeus University

Plan

Parser Types

- Slides on parsing by Jonas Lundberg and Narges Khakpour
- Input representation: a string
- Output representation: Parse (or syntax) tree
- Top-down or Bottom-up parsers

Parser Types

- Top-down parser
  - Starts from the start symbol (the tree root)
  - Transforms the start symbol into the input string to build the tree body
  - *Recursive descent parsing* vs *Table-driven parsing*
- Bottom-up parser
  - Starts from the input string that correspond to leaves
  - Constructs the rest of tree up to the start symbol (the tree root)
- We discuss top-down methods in this course.

Top-down Methods: An Example

Consider the grammar

$$(1) \quad S \rightarrow aABe \qquad (2) \quad A \rightarrow b \qquad (3) \quad A \rightarrow Abc \qquad (4) \quad B \rightarrow d$$

- Using a left-most derivation we can show that *abbcde* is in the language

$$S \stackrel{1}{\Longrightarrow} aABe \stackrel{3}{\Longrightarrow} aAbcBe \stackrel{2}{\Longrightarrow} abbcBe \stackrel{4}{\Longrightarrow} abbcde$$

  This is a *top-down method* since we start from the start symbol S (the syntax tree root) and work our way down to the Symbols *abbcde* (the leaves of the syntax tree).

- **Which non-terminal to apply a production rule to?**
- **What production to use when facing one (or k) Symbols?**
- AntLR uses a top-down parsing method.

Bottom-up Methods: An Example

$$(1) \quad S \rightarrow aABe \qquad (2) \quad A \rightarrow b \qquad (3) \quad A \rightarrow Abc \qquad (4) \quad B \rightarrow d$$

- Bottom-up approaches starts with the leaves and uses the grammar productions to reduce the input to the start symbol S.

$$abbcde \xrightarrow{2} aAbcde \xrightarrow{3} aAde \xrightarrow{4} aABe \xrightarrow{1} S$$

- That is, bottom-up parsing is a backward rightmost derivation.

$LL(k)$ Parsers

- A top-down parser
- Is predictive, i.e. predicts next production rule to be used
- Derivation: a non-terminal symbol to apply a production rule on.
- First L means read input string from left to right
- Second L means produce *leftmost derivation*
- k is the number of lookahead symbols
- An LL(k) grammar is a grammar that can be parsed by an LL(k) parser.

$LL(k)$ Parser: An Example

- Consider the following grammar

$$A \to aAB \mid ba \mid bb \quad \text{(Rules } 1-3\text{)}, \qquad B \to Ba \mid b \quad \text{(Rules } 4-5\text{)}$$

- Left-most derivation to show that *ababaa* is in the language

Status	Left-most	Remaining (to be parsed)	Rule/Comment
A	A	$ababaa$	Rule 1, $k=1$
aAB	A	$babaa$	Rule 2, $k=2$, Left-factorization
$abaB$	B	baa	Rule 4, $k=2$, Left-recursion
$abaBa$	B	baa	Rule 4
$abaBaa$	B	baa	Rule 5
$ababaa$	—	—	OK!

$LL(k)$ Parser: An Example

- Consider the following grammar

$$A \to aAB \mid ba \mid bb \quad (\text{Rules } 1-3), \qquad B \to Ba \mid b \quad (\text{Rules } 4-5)$$

- Left-most derivation to show that *ababaa* is in the language

Status	Left-most	Remaining (to be parsed)	Rule/Comment
A	A	$ababaa$	Rule 1, $k=1$
aAB	A	$babaa$	Rule 2, $k=2$, Left-factorization
$abaB$	B	baa	Rule 4, $k=2$, Left-recursion
$abaBa$	B	baa	Rule 4
$abaBaa$	B	baa	Rule 5
$ababaa$	–	–	OK!

$LL(k)$ Parser: An Example

- Consider the following grammar

$$A \to aAB \mid ba \mid bb \quad (\text{Rules } 1-3), \qquad B \to Ba \mid b \quad (\text{Rules } 4-5)$$

- Left-most derivation to show that *ababaa* is in the language

Status	Left-most	Remaining (to be parsed)	Rule/Comment
A	A	$ababaa$	Rule 1, $k = 1$
aAB	A	$babaa$	Rule 2, $k = 2$, Left-factorization
$abaB$	B	baa	Rule 4, $k = 2$, Left-recursion
$abaBa$	B	baa	Rule 4
$abaBaa$	B	baa	Rule 5
$ababaa$	–	–	OK!

$LL(k)$ Parser: An Example

- Consider the following grammar

$$A \to aAB \mid ba \mid bb \quad \text{(Rules } 1-3\text{)}, \qquad B \to Ba \mid b \quad \text{(Rules } 4-5\text{)}$$

- Left-most derivation to show that *ababaa* is in the language

Status	Left-most	Remaining (to be parsed)	Rule/Comment
A	A	$ababaa$	Rule 1, $k=1$
aAB	A	$babaa$	Rule 2, $k=2$, Left-factorization
$abaB$	B	baa	Rule 4, $k=2$, Left-recursion
$abaBa$	B	baa	Rule 4
$abaBaa$	B	baa	Rule 5
$ababaa$	–	–	OK!

$LL(k)$ Parser: An Example

- Consider the following grammar

$$A \to aAB \mid ba \mid bb \quad (\text{Rules } 1-3), \qquad B \to Ba \mid b \quad (\text{Rules } 4-5)$$

- Left-most derivation to show that *ababaa* is in the language

Status	Left-most	Remaining (to be parsed)	Rule/Comment
A	A	$ababaa$	Rule 1, $k=1$
aAB	A	$babaa$	Rule 2, $k=2$, Left-factorization
$abaB$	B	baa	Rule 4, $k=2$, Left-recursion
$abaBa$	B	baa	Rule 4
$abaBaa$	B	baa	Rule 5
$ababaa$	–	–	OK!

$LL(k)$ Parser: An Example

- Consider the following grammar

$$A \to aAB \mid ba \mid bb \quad \text{(Rules } 1-3\text{)}, \qquad B \to Ba \mid b \quad \text{(Rules } 4-5\text{)}$$

- Left-most derivation to show that *ababaa* is in the language

Status	Left-most	Remaining (to be parsed)	Rule/Comment
A	A	$ababaa$	Rule 1, $k = 1$
aAB	A	$babaa$	Rule 2, $k = 2$, Left-factorization
$abaB$	B	baa	Rule 4, $k = 2$, Left-recursion
$abaBa$	B	baa	Rule 4
$abaBaa$	B	baa	Rule 5
$ababaa$	–	–	OK!

- Is the above grammar an LL(1)?
- Rewrite the grammar

$$A \to aAB \mid bA', \quad A' \to a \mid b, \quad B \to bB', \quad B' \to aB' \mid \varepsilon$$

- Given a left-most non-terminal A and the remaining string, decide which production $A \to \alpha$ to chose.

Removing Ambiguity

- Derivations are not unique, even if the grammar is unambiguous
- In an unambiguous grammar, leftmost derivations are unique and right-most derivation are unique
- Consider the following grammar with the start symbol S

$$S \rightarrow S + S \mid S * S \mid a \mid b$$

$$S \xrightarrow{1} S + S \xrightarrow{3} a + S \xrightarrow{2} a + S * S \xrightarrow{3} a + a * S \xrightarrow{4} a + a * b$$

$$S \xrightarrow{2} S * S \xrightarrow{1} S + S * S \xrightarrow{3} a + S * S \xrightarrow{3} a + a * S \xrightarrow{4} a + a * b$$

- Removing left recursion and left-factoring sometimes can help to obtain an LL(1) grammar.

BonusPoint Is an ambiguous grammar an LL(1) grammar? Argue why.

Plan

The Recursive Descent Parsing

- In a parse tree
  - Leaves are terminals
  - Interior nodes correspond to non-terminals
  - Root is the start symbol.
- Input string: $s_1 s_2 \ldots s_n$
- We associate each non-terminal A with one procedure `pA()`.
- Procedure `pA()` is used to select A productions ($A \to \alpha$).

The Recursive Descent Parsing

- Consider $A \to bCd \mid eF \quad C \to cCF|e \quad F \to b$ where $A, C, F \in V$ and $b, d, e \in \Sigma$

```
pA() {
 Node(A);
 if lookahead = b then
    consume(b); pC(); consume(d);
  elsif lookahead = e then
    consume(e); pF();
  else
    reportError();
  end if;
        // Can't make decision,
       // no suitable A production
}
```

```
consume(Symbol t) {

 if lookahead = t then
    Node(t); lookahead = nextSymbol
  else
       reportError();
         // Error in prediction
  end if;
              }
```

- Initializing the parser

```
lookahead = nextSymbol();           // Read first Symbol from input st
pStartSymbol();                    // Resolve start symbol
```

Recursive Descent: An Example

$A \to bCd \mid eF \quad C \to cCF|e \quad F \to b$

Construct parse tree for $bcebd$

A

Recursive Descent: An Example

$A \rightarrow bCd \mid eF \quad C \rightarrow cCF|e \quad F \rightarrow b$

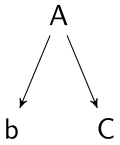
Construct parse tree for $bcebd$

$$\begin{tikzcd} A \arrow[d] \\ b \end{tikzcd}$$

Recursive Descent: An Example

$A \rightarrow bCd \mid eF \quad C \rightarrow cCF|e \quad F \rightarrow b$

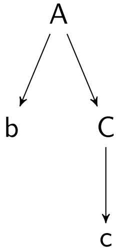
Construct parse tree for *bcebd*



Recursive Descent: An Example

$A \rightarrow bCd \mid eF \quad C \rightarrow cCF|e \quad F \rightarrow b$

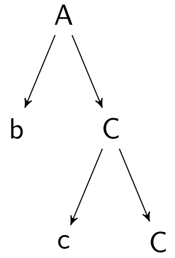
Construct parse tree for *bcebd*



Recursive Descent: An Example

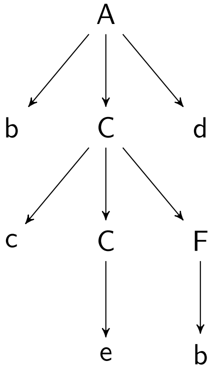
$A \rightarrow bCd \mid eF \quad C \rightarrow cCF|e \quad F \rightarrow b$

Construct parse tree for *bcebd*



Recursive Descent: An Example

If we continue, the final result is



Example: Arithmetic Expressions

An LL(1) grammar for arithmetic expressions:

$$\begin{aligned} G &= \{T, N, P, S\} \\ T &= \{id, +, *, (,),\} \\ N &= \{E, E', T, T', F\} \\ S &= E \end{aligned}$$

where P is defined as

$$\begin{array}{llll} (1) & E & \rightarrow & TE', \quad E' \rightarrow +TE' \mid \varepsilon, \\ (2) & T & \rightarrow & F\ T', \quad T' \rightarrow *F\ T' \mid \varepsilon, \\ (3) & F & \rightarrow & id \mid (E) \end{array}$$

Notice: Ambiguity, left-factoring, and left-recursion are all removed.

Example: Arithmetic Expressions

The RD procedure associated with $T' \to *F\ T' \mid \varepsilon$

```
pTprime() {
   if lookahead = * then
      consume(*); pF(); pTprime();
   elsif lookahead = X then
      ;   //Do nothing, the epsilon branch
   else
      reportError();
   end if;
}
```

- The ε-production for T' is the tricky part.
- We must determine on what input T' should do nothing and when to report error.

Example: Arithmetic Expressions

The RD procedure associated with $T' \rightarrow *F\ T' \mid \varepsilon$

```
pTprime() {
if lookahead = * then
consume(*); pF(); pTprime();
elsif lookahead = X then
;   //Do nothing, the epsilon branch
else
reportError();
end if;
}
```

BonusPoint What should X be in the above?

Recursive Descent Pros/Cons

- Easy to understand how parsing works
- Grammar updates are often difficult to handle: a minor update may get propagated to the whole parser.
- An alternative is to encode all predictions in a parsing table, and the parser uses that table to parse a grammar.

Plan

Table-Driven Approach

- Recursive calls are replaced by a stack.
- Which production branch to chose is given by a **parse table** $M[A, t]$.
- Given non-terminal A and lookahead t, $M[A, t]$ returns the appropriate production to use.
- There are algorithms for constructing parse tables

A Parse Table $M[A, t]$ for Grammar 3

	id	$+$	$*$	(	)
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Questions:

- How to construct a parse table? (not in this course)
- How to build a parser that uses a parse table?

Algorithm 1: Table driven LL(1)-parsing

stack.*push*(*StartSymbol*)
$LA = input.nextSymbol()$
repeat
  $X = stack.top()$
  if $X \in T$ **then**
    if $X = LA$ **then**
      Node(*stack*.*top*)
      stack.*pop*() (reduce)
      $LA = input.nextSymbol()$
    else
      error(stack,LA,input) (Symbol not in agreement with prediction)
    end if
  else
    if $M[X, LA] = X \to Y_1 \dots Y_n$ **then**
      stack.*pop*() (shift)
      push $Y_n \dots Y_1$ onto *stack*, with Y_1 on top
      add $X \to Y_1 \dots Y_n$ to parse tree
    else
      error(stack,LA,input) (Can't make a decision, empty slot in $M[X, LA]$)
    end if

Table driven LL(1)-parsing: An Example

	id	$+$	$*$	(	)
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \to TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \to FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \to id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \to \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \to +TE')$

Table driven LL(1)-parsing: An Example

	id	$+$	$*$	(	)
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \to TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \to FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \to id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \to \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \to +TE')$

Table driven LL(1)-parsing: An Example

	id	$+$	$*$	$($	$)$
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \to TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \to FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \to id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \to \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \to +TE')$

Table driven LL(1)-parsing: An Example

	id	$+$	$*$	$($	$)$
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \to TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \to FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \to id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \to \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \to +TE')$

Table driven LL(1)-parsing: An Example

	id	$+$	$*$	$($	$)$
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \to TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \to FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \to id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \to \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \to +TE')$

Table driven LL(1)-parsing: An Example

	id	+	*	(	)
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \rightarrow TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \rightarrow FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \rightarrow id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \rightarrow \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \rightarrow +TE')$

Table driven LL(1)-parsing: An Example

	id	$+$	$*$	(	)
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:** Push start symbol E on stack $\Rightarrow TOP = E$
  Lookahead is first input Symbol $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:** if $TOP = LA$ then **reduce** , else **shift**.
  shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side.
  reduce $\Rightarrow$ pop top element (a terminal) and set lookahead to next input.

Stack	TOP	LA	Remains	*Rule*
E	E	id	$+id\$$	shift with $M[E, id] = (E \to TE')$
TE'	T	id	$+id\$$	shift with $M[T, id] = (T \to FT')$
$FT'E'$	F	id	$+id\$$	shift with $M[F, id] = (F \to id)$
$idT'E'$	id	id	$+id\$$	reduce $\Rightarrow TOP = T', LA = +$
$T'E'$	T'	$+$	$id\$$	shift with $M[T', +] = (T' \to \varepsilon)$
E'	E'	$+$	$id\$$	shift with $M[E', +] = (E' \to +TE')$

Table driven LL(1)-parsing: An Example

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Success:** When lookahead is end-of-file ($LA = \$$)

Remains	LA	TOP	Stack
$+id\$$	id	E	E
$+id\$$	id	T	TE'
$+id\$$	id	F	$FT'E'$
$+id\$$	id	id	$idT'E'$
$id\$$	$+$	T'	$T'E'$
$id\$$	$+$	E'	E'

Remains	LA	TOP	Stack
$id\$$	$+$	$+$	$+TE'$
\$	id	T	TE'
\$	id	F	$FT'E'$
\$	id	id	$idT'E'$
	\$	T'	$T'E'$

Plan

RE to CFG

Theorem

For any arbitrary regular expression E defined over an alphabet Σ, there is a CFG G such that $L(E) = L(G)$.

Proof.

The operators in a RE include union ($\cup$), concatenation (.) and closure (R^*). We prove this by induction on the number of operators n.

Base Case If $n = 0$, then two cases can happen

- $E = \sigma$ where $\sigma \in \Sigma$: then the following CFG generates $L(E)$
  $G = \langle \{S\}, \Sigma, \{S \to \sigma\}, S \rangle$
- $E = \epsilon$: $G = \langle \{S\}, \Sigma, \{S \to \epsilon\}, S \rangle$

$\square$

RE to CFG

Inductive Step: For all regular expressions E_k with k operator, assume that there is a CFG G_k such that $L(G_k) = L(E_k)$. Let E_{k+1} be a regular expression. Three cases can happen:

- $E_{k+1} = E' \cup E''$: According to the induction step hypothesis, there are $G' = \langle V', \Sigma', R', S' \rangle$ and $G'' = \langle V'', \Sigma'', R'', S'' \rangle$ such that $L(G') = L(E')$, $L(G'') = L(E'')$ and $V' \neq V''$. Consider the following CFG where $S \notin V' \cup V''$:

$$G = \langle V' \cup V'' \cup \{S\}, \Sigma' \cup \Sigma'', \{S \to S', S \to S''\} \cup R' \cup R'', S \rangle$$

$$L(S) = L(S') \cup L(S'') = L(E') \cup L(E'') = L(E_{k+1})$$

RE to CFG

- $E_{k+1} = E'.E''$: According to the induction step hypothesis, there are $G' = \langle V', \Sigma', R', S' \rangle$ and $G'' = \langle V'', \Sigma'', R'', S'' \rangle$ such that $L(G') = L(E')$, $L(G'') = L(E'')$ and $V' \neq V''$. Consider the following CFG where $S \notin V' \cup V''$:

$$G = \langle V' \cup V'' \cup \{S\}, \Sigma' \cup \Sigma'', \{S \to S'S''\} \cup R' \cup R'', S \rangle$$

$$L(S) = L(S'S'') = L(E').L(E'') = L(E_{k+1})$$

RE to CFG

- $E_{k+1} = E'^{*}$: According to the induction step hypothesis, there are $G' = \langle V', \Sigma', R', S' \rangle$ such that $L(G') = L(E')$. Consider the following CFG where $S \notin V'$:

$$G = \langle V' \cup \{S\}, \Sigma', \{S \rightarrow S'S \mid \epsilon\} \cup R', S \rangle$$

$$\begin{aligned} L(S) = \;& \epsilon \cup L(S') \cup L(S'S') \cup L(S'S'S') \ldots = \\ & \epsilon \cup L(E') \cup L(E'E') \cup L(E'E'E') \ldots = L(E_{k+1}) \end{aligned}$$

CFGs $\not\subseteq$ REs

Theorem

There are CFGs that generates non-regular languages, i.e. CFGs $\not\subseteq$ REs.

Proof.

Consider a context-free grammar that produces $\{a^n b^n \mid 0 \leq n\}$.
We know that a RE has a corresponding FSA. The accepting DFA of should accept $a^n b^n$ and reject $a^m b^n$, where $n \neq m$. It means it should reach two different states where one is an accepting state (to accept $a^n b^n$) and the other is not (to reject $a^m b^n$). Since n is infinite, then we need to have infinite states while DFA only has a finite number of states.
Hence, we cannot represent it using a regular language. $\square$

Plan

Conclusions

- Recursive descent parsing and table-driven parsing
- Regular languages are context-free but not all context-free languages are regular
- Next lecture is on Turing Machines