Recapitulation from Previous Lectures

Recap: Naive Derivation of Regular Expressions

$$
\begin{aligned}
\delta^c(\varnothing) &= \varnothing \\
\delta^c(\varepsilon) &= \varnothing \\
\delta^c(d) &= \begin{cases} \varepsilon & \textbf{if } d = c \\ \varnothing & \textbf{if } d \neq c \end{cases} \\
\delta^c(e_1 \mid e_2) &= \delta^c(e_1) \mid \delta^c(e_2) \\
\delta^c(e_1\, e_2) &= \begin{cases} \delta^c(e_1)\, e_2 \mid \delta^c(e_2) & \textbf{if } \mathit{nullable}(e_1) \\ \delta^c(e_1)\, e_2 & \textbf{otherwise} \end{cases} \\
\delta^c(e_1^*) &= \delta^c(e_1)\, e_1^*
\end{aligned}
$$

Problem: produces “dumb” repeated patterns, resulting in poor performance:

$$\delta^{\mathrm{aaaa}}(\mathrm{a}^*) = \varnothing\mathrm{a}^* \mid \varnothing\mathrm{a}^* \mid \varnothing\mathrm{a}^* \mid \varepsilon\,\mathrm{a}^*$$

Recap: Naive Derivation in Scala

```
def derive(expr: RegExp, c: Character): RegExp =
  expr match
    case Failure | EmptyStr ⇒ Failure
    case CharWhere(pred) ⇒
      if pred(c) then EmptyStr else Failure
    case Union(left, right) ⇒ derive(left, c) | derive(right, c)
    case Concat(left, right) ⇒
      val w = derive(left, c) ~ right
      if left.acceptsEmpty then w | derive(right, c) else w
    case Star(inner) ⇒ derive(inner, c) ~ expr
```

Recap: Normalizing Derivation

Idea: while differentiating,

- "flatten" disjunction structures (take them out of sequences)
- remove duplicates from generated disjunctions

Recap: Normalizing Derivation

Idea: while differentiating,

- "flatten" disjunction structures (take them out of sequences)
- remove duplicates from generated disjunctions

Example: consider a^*

$\delta^{\mathsf{a}}(\mathsf{a}^*) = \varepsilon \mathsf{a}^*$

$\delta^{\mathsf{aa}}(\mathsf{a}^*) = \varnothing \mathsf{a}^* \mid \varepsilon \mathsf{a}^*$

Recap: Normalizing Derivation

Idea: while differentiating,

- "flatten" disjunction structures (take them out of sequences)
- remove duplicates from generated disjunctions

Example: consider a^*

$\delta^{a}(a^*) = \varepsilon a^*$
$\delta^{aa}(a^*) = \emptyset a^* \mid \varepsilon a^*$
$\delta^{aaa}(a^*) = \emptyset a^* \cancel{\mid \emptyset a^*} \mid \varepsilon a^*$

Recap: Normalizing Derivation

Idea: while differentiating,

- "flatten" disjunction structures (take them out of sequences)
- remove duplicates from generated disjunctions

Example: consider a*

$\delta^{a}(a^*) = \varepsilon a^*$

$\delta^{aa}(a^*) = \varnothing a^* \mid \varepsilon a^*$

$\delta^{aaa}(a^*) = \varnothing a^* \cancel{\mid \varnothing a^*} \mid \varepsilon a^*$

$\delta^{aaaa}(a^*) = \varnothing a^* \cancel{\mid \varnothing a^*} \mid \varepsilon a^*$

...

Recap: Normalizing Derivation in Scala

```
def deriveNorm(char: Character): RegExp =
  val disjuncted = collection.mutable.Set[RegExp]()
  def work(expr: RegExp, rest: RegExp): Unit = expr match
    case CharWhere(pred) ⇒ if pred(char) then disjuncted += rest
    case Union(left, right) ⇒ work(left, rest); work(right, rest)
    case Concat(left, right) ⇒
      work(left, right ~ rest)
      if left.acceptsEmpty then work(right, rest)
    case Star(inner) ⇒ work(inner, expr ~ rest)
    case Failure | EmptyStr ⇒ ()
  work(this, EmptyStr) // register unions into `disjuncted`
  disjuncted.foldLeft[RegExp](Failure)(_ | _) // rebuild regexp
```

Theory of Normalizing Derivation

Sets of Regular Expressions

We will now be deadline with sets E of regular expressions.

The ascribed semantics will be that of a disjunction of all e in E.

Since $\cdot | \cdot$ is commutative, associative, and idempotent (just like sets),

this is a good representation.

Sets of Regular Expressions

We will now be deadline with sets E of regular expressions.

The ascribed semantics will be that of a disjunction of all e in E.

Since $\cdot|\cdot$ is commutative, associative, and idempotent (just like sets), this is a good representation.

Lexicographic Sorting

We can totally order any inductive data type *lexicographically*.
For example, $\varnothing < \varepsilon < \varnothing\varnothing < \varnothing\varepsilon < \varepsilon\varepsilon < \varnothing|\varnothing < \ldots$ etc.

Sets of Regular Expressions

We will now be deadline with sets E of regular expressions.

The ascribed semantics will be that of a disjunction of all e in E.

Since $\cdot|\cdot$ is commutative, associative, and idempotent (just like sets),

this is a good representation.

Lexicographic Sorting

We can totally order any inductive data type *lexicographically*.
For example, $\varnothing \;<\; \varepsilon \;<\; \varnothing\varnothing \;<\; \varnothing\varepsilon \;<\; \varepsilon\varepsilon \;<\; \varnothing|\varnothing \;<\; \ldots$ etc.

Definition (disjunction of set of regexp)

Define $\mathit{disj}(E)$ as the (left-associated) disjunction of the elements in E,
taken in lexicographic order.

Example: $\mathit{disj}(\{\varepsilon, \mathrm{ab}, \mathrm{a}\}) \;=\; \varepsilon\,|\,\mathrm{a}\,|\,\mathrm{ab}$

Normalizing Derivation, Formally

Let the *normalizing derivation* of e w.r.t. c be defined as: $\underline{\delta}^c(e) = \mathit{disj}(\mathit{derive}_c(e, \varepsilon))$

$$
\begin{aligned}
\mathit{derive}_c(\varnothing, e) &= \emptyset \\
\mathit{derive}_c(\varepsilon, e) &= \emptyset \\
\mathit{derive}_c(c, e) &= \{e\} \\
\mathit{derive}_c(d, e) &= \emptyset \qquad (d \neq c) \\
\mathit{derive}_c(e_1 \,|\, e_2, e) &= \mathit{derive}_c(e_1, e) \cup \mathit{derive}_c(e_2, e) \\
\mathit{derive}_c(e_1\, e_2, e) &= \mathit{derive}_c(e_1, e_2\, e) \cup \begin{cases} \mathit{derive}_c(e_2, e) & \text{if } \mathit{nullable}(e_1) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{derive}_c({e_1}^*, e) &= \mathit{derive}_c(e_1, {e_1}^* e)
\end{aligned}
$$

Normalizing Derivation, Formally

Let the *normalizing derivation* of e w.r.t. c be defined as: $\underline{\delta}^c(e) = \mathit{disj}(\mathit{derive}_c(e, \varepsilon))$

$$
\begin{aligned}
\mathit{derive}_c(\varnothing, e) &= \emptyset \\
\mathit{derive}_c(\varepsilon, e) &= \emptyset \\
\mathit{derive}_c(c, e) &= \{e\} \\
\mathit{derive}_c(d, e) &= \emptyset \qquad (d \neq c) \\
\mathit{derive}_c(e_1 \,|\, e_2, e) &= \mathit{derive}_c(e_1, e) \cup \mathit{derive}_c(e_2, e) \\
\mathit{derive}_c(e_1\, e_2, e) &= \mathit{derive}_c(e_1, e_2\, e) \cup \begin{cases} \mathit{derive}_c(e_2, e) & \text{if } \mathit{nullable}(e_1) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{derive}_c({e_1}^*, e) &= \mathit{derive}_c(e_1, {e_1}^*\, e)
\end{aligned}
$$

Easy to verify that $L(\underline{\delta}^c(e)) = L(\delta^c(e))$.

Normalizing Derivation, Formally

Let the *normalizing derivation* of e w.r.t. c be defined as: $\underline{\delta}^c(e) = \mathit{disj}(\mathit{derive}_c(e, \varepsilon))$

$$
\begin{aligned}
\mathit{derive}_c(\varnothing, e) &= \emptyset \\
\mathit{derive}_c(\varepsilon, e) &= \emptyset \\
\mathit{derive}_c(c, e) &= \{e\} \\
\mathit{derive}_c(d, e) &= \emptyset \qquad (d \neq c) \\
\mathit{derive}_c(e_1 \mid e_2, e) &= \mathit{derive}_c(e_1, e) \cup \mathit{derive}_c(e_2, e) \\
\mathit{derive}_c(e_1\, e_2, e) &= \mathit{derive}_c(e_1, e_2\, e) \cup \begin{cases} \mathit{derive}_c(e_2, e) & \text{if } \mathit{nullable}(e_1) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{derive}_c(e_1{}^*, e) &= \mathit{derive}_c(e_1, e_1{}^*\, e)
\end{aligned}
$$

Easy to verify that $L(\underline{\delta}^c(e)) = L(\delta^c(e))$.

Notations: $\mathit{derive}_\varepsilon(e) = \{e\}$ and $\mathit{derive}_{cw}(e) = \bigcup\{\mathit{derive}_w(e_0) \mid e_0 \in \mathit{derive}_c(e)\}$

We can show that $\mathit{disj}(\mathit{derive}_w(e)) = \underline{\delta}^w(e)$

Max of Regular Expressions

Intuition: *over-approximate* the set of *any* successive derivations of a regular expression

Max of Regular Expressions

Intuition: *over-approximate* the set of *any* successive derivations of a regular expression

$$
\begin{aligned}
max(\varnothing, e) &= \emptyset \\
max(\varepsilon, e) &= \emptyset \\
max(c, e) &= \{e\} \\
max(e_1 \mid e_2, e) &= max(e_1, e) \cup max(e_2, e) \\
max(e_1\, e_2, e) &= max(e_1, e_2\, e) \cup max(e_2, e) \\
max({e_1}^*, e) &= max(e_1, {e_1}^*\, e)
\end{aligned}
$$

Max of Regular Expressions

Intuition: *over-approximate* the set of *any* successive derivations of a regular expression

$$
\begin{aligned}
max(\varnothing, e) &= \emptyset \\
max(\varepsilon, e) &= \emptyset \\
max(c, e) &= \{e\} \\
max(e_1 \mid e_2, e) &= max(e_1, e) \cup max(e_2, e) \\
max(e_1\, e_2, e) &= max(e_1, e_2\, e) \cup max(e_2, e) \\
max({e_1}^*, e) &= max(e_1, {e_1}^*\, e)
\end{aligned}
$$

Theorem

For any e, $max(e, \varepsilon)$ over-approximates all successive derivations of e:

$$\forall c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

Max of Regular Expressions

Intuition: *over-approximate* the set of *any* successive derivations of a regular expression

$$
\begin{aligned}
max(\varnothing, e) &= \emptyset \\
max(\varepsilon, e) &= \emptyset \\
max(c, e) &= \{e\} \\
max(e_1 \mid e_2, e) &= max(e_1, e) \cup max(e_2, e) \\
max(e_1\, e_2, e) &= max(e_1, e_2\, e) \cup max(e_2, e) \\
max(e_1{}^*, e) &= max(e_1, e_1{}^*\, e)
\end{aligned}
$$

Theorem

For any e, $max(e, \varepsilon)$ over-approximates all successive derivations of e:

$$\forall c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

Not tight: $max(\varnothing \mathrm{a}, \varepsilon) = \{\mathrm{a}\}$ but for any c and w we have $derive_{cw}(\varnothing \mathrm{a}, \varepsilon) = \varnothing$

Max of Regular Expressions, Proof

Theorem

$$\forall e, c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

Proof.

By induction on w, showing that: $\forall e.\ derive_{w}(e) \subseteq max(e, \varepsilon) \cup \{e\}$

Max of Regular Expressions, Proof

Theorem

$$\forall e, c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

Proof.

By induction on w, showing that: $\forall e.\ derive_w(e) \subseteq max(e, \varepsilon) \cup \{e\}$

- When $w = \varepsilon$, we have

$$derive_w(e) = derive_\varepsilon(e) = \{e\} \subseteq max(e, \varepsilon) \cup \{e\}$$

Max of Regular Expressions, Proof

Theorem

$$\forall e, c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

Proof.

By induction on w, showing that: $\forall e.\ derive_w(e) \subseteq max(e, \varepsilon) \cup \{e\}$

- When $w = \varepsilon$, we have

$$derive_w(e) = derive_\varepsilon(e) = \{e\} \subseteq max(e, \varepsilon) \cup \{e\}$$

- When $w = cw'$, assuming $\forall e'.\ derive_{w'}(e') \subseteq max(e', \varepsilon) \cup \{e'\}$, we have

$$\begin{aligned} derive_w(e) &= \bigcup \{ derive_{w'}(e') \mid e' \in derive_c(e) \} \\ &\subseteq \bigcup \{ max(e', \varepsilon) \mid e' \in derive_c(e) \} \cup \{e\} \end{aligned}$$

and we can show by induction on e that the latter is $\subseteq max(e, \varepsilon) \cup \{e\}$

$\square$

Max of Regular Expressions, Consequence

Theorem

$$\forall e, c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

$\Rightarrow$ Number of distinct regexps generated by normalizing derivation of any w **is *bounded*!**

Max of Regular Expressions, Consequence

Theorem $$\forall e, c, w.\ derive_{cw}(e) \subseteq max(e, \varepsilon)$$

$\Rightarrow$ Number of distinct regexps generated by normalizing derivation of any w **is *bounded*!**

More specifically, for any e, we have: (remember $\underline{\delta}^{w}(e) \;=\; disj(derive_{w}(e))$)

$$|\{\underline{\delta}^{w}(e) \mid w \in A^{*}\}| \;\leq\; 1 + 2^{|max(e,\varepsilon)|}$$

(Better bounds can be derived.)

Max of Regular Expressions, Consequence

Theorem $$\forall e, c, w.\ \mathit{derive}_{cw}(e) \subseteq \mathit{max}(e, \varepsilon)$$

$\Rightarrow$ Number of distinct regexps generated by normalizing derivation of any w **is *bounded*!**

More specifically, for any e, we have: (remember $\underline{\delta}^w(e) = \mathit{disj}(\mathit{derive}_w(e))$)

$$|\{\underline{\delta}^w(e) \mid w \in A^*\}| \leq 1 + 2^{|\mathit{max}(e,\varepsilon)|}$$

(Better bounds can be derived.)

Opens the door to efficient memoization (caching)
allowing regexp matching in ***constant space*** and ***linear time*** w.r.t. size of words

Memoizing successive derivations

Algorithm regexp matching:

- Start with empty mapping $M := \emptyset$ and with regexp e
- For each i^{th} character c_i in w,
  - If $(e, c_i) \notin domain(M)$, set $M(e, c_i) := \underline{\delta}^{c_i}(e)$
  - Set $e := M(e, c_i)$
- Test whether $nullable(e)$

In Scala types $M :$ mutable.Map[(RegExp, Char), RegExp]

Memoizing successive derivations

Algorithm regexp matching:

- Start with empty mapping $M := \emptyset$ and with regexp e
- For each i^{th} character c_i in w,
  - If $(e, c_i) \notin \mathit{domain}(M)$, set $M(e, c_i) := \underline{\delta}^{c_i}(e)$
  - Set $e := M(e, c_i)$
- Test whether $\mathit{nullable}(e)$

In Scala types $M :$ mutable.Map[(RegExp, Char), RegExp]

Conceptually builds a deterministic finite-state automaton on the fly

Memoizing successive derivations

Algorithm regexp matching:

- Start with empty mapping $M := \emptyset$ and with regexp e
- For each i^{th} character c_i in w,
  - If $(e, c_i) \notin \mathit{domain}(M)$, set $M(e, c_i) := \underline{\delta}^{c_i}(e)$
  - Set $e := M(e, c_i)$
- Test whether $\mathit{nullable}(e)$

In Scala types $M :$ mutable.Map[(RegExp, Char), RegExp]

Conceptually builds a deterministic finite-state automaton on the fly

Note: to match *several tokens* in a row using longest match (*tokenization*),
we need either linear-space memoization or some *backtracking*

Memoizing successive derivations

Algorithm regexp matching:

- Start with empty mapping $M := \emptyset$ and with regexp e
- For each i^{th} character c_i in w,
  - If $(e, c_i) \notin domain(M)$, set $M(e, c_i) := \underline{\delta}^{c_i}(e)$
  - Set $e := M(e, c_i)$
- Test whether $nullable(e)$

In Scala types M : mutable.Map[(RegExp, Char), RegExp]

Conceptually builds a deterministic finite-state automaton on the fly

Note: to match *several tokens* in a row using longest match (*tokenization*),

we need either linear-space memoization or some *backtracking*

More or less how **Silex** is implemented! (library used in the Amy project)

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathsf{ab}} = \{\, \mathsf{a}^n\mathsf{b}^n \mid n \geq 0 \,\}$

Is L_{ab} regular?

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathsf{ab}} = \{\, \mathsf{a}^n \mathsf{b}^n \mid n \geq 0 \,\}$

Is L_{ab} regular? If not, how to prove it?

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathrm{ab}} = \{\, \mathrm{a}^n \mathrm{b}^n \mid n \geq 0 \,\}$

Is L_{ab} regular? If not, how to prove it?

Proof: By contradiction.

Assume there is a regexp e_{ab} such that $L(e_{\mathrm{ab}}) = L_{\mathrm{ab}}$.

Let $K = 1 + 2^{|max(e_{\mathrm{ab}}, \varepsilon)|}$

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathrm{ab}} = \{\, \mathrm{a}^n \mathrm{b}^n \mid n \geq 0 \,\}$

Is L_{ab} regular? If not, how to prove it?

Proof: By contradiction.

Assume there is a regexp e_{ab} such that $L(e_{\mathrm{ab}}) = L_{\mathrm{ab}}$.

Let $K = 1 + 2^{|max(e_{\mathrm{ab}},\, \varepsilon)|}$

There must be an $i \leq K$ such that $\underline{\delta}^{\mathrm{a}^{K+1}}(e_{\mathrm{ab}}) = \underline{\delta}^{\mathrm{a}^i}(e_{\mathrm{ab}})$

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathrm{ab}} = \{\, \mathrm{a}^n \mathrm{b}^n \mid n \geq 0 \,\}$

Is L_{ab} regular? If not, how to prove it?

Proof: By contradiction.

Assume there is a regexp e_{ab} such that $L(e_{\mathrm{ab}}) = L_{\mathrm{ab}}$.

Let $K = 1 + 2^{|max(e_{\mathrm{ab}}, \varepsilon)|}$

There must be an $i \leq K$ such that $\underline{\delta}^{\mathrm{a}^{K+1}}(e_{\mathrm{ab}}) = \underline{\delta}^{\mathrm{a}^i}(e_{\mathrm{ab}})$

By definition, $\underline{\delta}^{\mathrm{a}^i}(e_{\mathrm{ab}})$ must accept b^i because $\mathrm{a}^i\mathrm{b}^i \in L_{\mathrm{ab}}$
which also means $\underline{\delta}^{\mathrm{a}^{K+1}}(e_{\mathrm{ab}})$ must accept b^i

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathsf{ab}} = \{ \mathsf{a}^n \mathsf{b}^n \mid n \geq 0 \}$

Is L_{ab} regular? If not, how to prove it?

Proof: By contradiction.

Assume there is a regexp e_{ab} such that $L(e_{\mathsf{ab}}) = L_{\mathsf{ab}}$.

Let $K = 1 + 2^{|max(e_{\mathsf{ab}}, \varepsilon)|}$

There must be an $i \leq K$ such that $\underline{\delta}^{\mathsf{a}^{K+1}}(e_{\mathsf{ab}}) = \underline{\delta}^{\mathsf{a}^i}(e_{\mathsf{ab}})$

By definition, $\underline{\delta}^{\mathsf{a}^i}(e_{\mathsf{ab}})$ must accept b^i because $\mathsf{a}^i\mathsf{b}^i \in L_{\mathsf{ab}}$
which also means $\underline{\delta}^{\mathsf{a}^{K+1}}(e_{\mathsf{ab}})$ must accept b^i ...

Expressiveness Limitation of Regular Expressions

Consider the language $L_{\mathrm{ab}} = \{\, \mathrm{a}^n \mathrm{b}^n \mid n \geq 0 \,\}$

Is L_{ab} regular? If not, how to prove it?

Proof: By contradiction.

Assume there is a regexp e_{ab} such that $L(e_{\mathrm{ab}}) = L_{\mathrm{ab}}$.

Let $K = 1 + 2^{|max(e_{\mathrm{ab}},\, \varepsilon)|}$

There must be an $i \leq K$ such that $\underline{\delta}^{\mathrm{a}^{K+1}}(e_{\mathrm{ab}}) = \underline{\delta}^{\mathrm{a}^i}(e_{\mathrm{ab}})$

By definition, $\underline{\delta}^{\mathrm{a}^i}(e_{\mathrm{ab}})$ must accept b^i because $\mathrm{a}^i \mathrm{b}^i \in L_{\mathrm{ab}}$
which also means $\underline{\delta}^{\mathrm{a}^{K+1}}(e_{\mathrm{ab}})$ must accept b^i ... ***but*** $\mathrm{a}^{K+1} \mathrm{b}^i \notin L_{\mathrm{ab}}$

$\Rightarrow$ ***Contradiction.***

$\square$

Introduction to Grammars

Regular *Grammars*

An equivalent way of defining regular languages – name intermediate productions:

$$\begin{aligned}
\text{start} &\rightarrow \text{letter}(\text{letter} \,|\, \text{digit})^* \\
\text{letter} &\rightarrow \text{a} \,|\, \text{b} \,|\, \text{c} \,|\, \ldots \,|\, \text{z} \\
\text{digit} &\rightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9
\end{aligned}$$

Regular *Grammars*

An equivalent way of defining regular languages – name intermediate productions:

$$\begin{aligned}\text{start} &\rightarrow \text{letter}(\text{letter}\,|\,\text{digit})^* \\ \text{letter} &\rightarrow \text{a}\,|\,\text{b}\,|\,\text{c}\,|\ldots|\,\text{z} \\ \text{digit} &\rightarrow 0\,|\,1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9\end{aligned}$$

"start", "letter", "digit" are part of a distinct set of identifiers called *non-terminals*,

often denoted abstractly by letters P, Q, R, S, ...

where S is often used as the Start symbol

Regular *Grammars*

An equivalent way of defining regular languages – name intermediate productions:

$$\begin{aligned} \text{start} &\rightarrow \text{letter}(\text{letter} \,|\, \text{digit})^* \\ \text{letter} &\rightarrow \text{a} \,|\, \text{b} \,|\, \text{c} \,|\, \ldots \,|\, \text{z} \\ \text{digit} &\rightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9 \end{aligned}$$

"start", "letter", "digit" are part of a distinct set of identifiers called *non-terminals*,

often denoted abstractly by letters P, Q, R, S, ...

where S is often used as the *S*tart symbol

Regularity requirement: *no recursion!*

Definitions should form a *directed acyclic graph* (DAG)

Context-Free Grammars (CFG)

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \;\rightarrow\; \varepsilon \mid \mathrm{a}\,S\,\mathrm{b}$$

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \rightarrow \varepsilon \mid \mathrm{a}\,S\,\mathrm{b}$$

Semantics given by *rewriting derivations* (note: not to be confused with *derivative*!)

Example derivation:

$$S$$

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \;\rightarrow\; \varepsilon \mid \mathsf{a}\,S\,\mathsf{b}$$

Semantics given by *rewriting derivations* (note: not to be confused with *derivative*!)

Example derivation:

$$S \;\Rightarrow\; \mathsf{a}\,S\,\mathsf{b}$$

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \rightarrow \varepsilon \mid \mathrm{a}\,S\,\mathrm{b}$$

Semantics given by *rewriting derivations* (note: not to be confused with *derivative*!)

Example derivation:

$$S \Rightarrow \mathrm{a}\,S\,\mathrm{b} \Rightarrow \mathrm{a}\,\mathrm{a}\,S\,\mathrm{b}\,\mathrm{b}$$

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \;\rightarrow\; \varepsilon \mid \mathrm{a}\,S\,\mathrm{b}$$

Semantics given by *rewriting derivations* (note: not to be confused with *derivative*!)

Example derivation:

$$S \;\Rightarrow\; \mathrm{a}\,S\,\mathrm{b} \;\Rightarrow\; \mathrm{a}\,\mathrm{a}\,S\,\mathrm{b}\,\mathrm{b} \;\Rightarrow\; \mathrm{a}\,\mathrm{a}\,\mathrm{a}\,S\,\mathrm{b}\,\mathrm{b}\,\mathrm{b}$$

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \rightarrow \varepsilon \mid \mathsf{a}\, S\, \mathsf{b}$$

Semantics given by *rewriting derivations* (note: not to be confused with *derivative*!)

Example derivation:

$$S \Rightarrow \mathsf{a}\,S\,\mathsf{b} \Rightarrow \mathsf{a\,a}\,S\,\mathsf{b\,b} \Rightarrow \mathsf{a\,a\,a}\,S\,\mathsf{b\,b\,b} \Rightarrow \mathsf{a\,a\,a}\,\varepsilon\,\mathsf{b\,b\,b}$$

Idea of Context-Free Grammars (CFG)

What if we allowed *recursion* in grammar definitions?

$$S \rightarrow \varepsilon \mid \mathrm{a}S\mathrm{b}$$

Semantics given by *rewriting derivations* (note: not to be confused with *derivative*!)

Example derivation:

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathrm{aa}S\mathrm{bb} \Rightarrow \mathrm{aaa}S\mathrm{bbb} \Rightarrow \mathrm{aaa}\varepsilon\mathrm{bbb} = \mathrm{aaabbb}$$

Definition of Context-Free Grammars (CFG)

Formally: a tuple $G = (A, N, S, R)$

- A – terminals (alphabet for generated words $w \in A^*$), usually *tokens*
- N – non-terminals – symbols with (recursive) definitions
- R – grammar rules as pairs (n, v), written $n \rightarrow v$ where $n \in N$ is a non-terminal
  $v \in (A \cup N)^*$ – sequence of terminals and non-terminals
- A derivation in G starts from the starting symbol S
- Each step replaces a non-terminal with one of its right hand sides

Example from before: $S \;\rightarrow\; \varepsilon \mid \mathrm{a}\, S\, \mathrm{b}$

$$G \;=\; (A = \{a, b\},\; N = \{S\},\; S,\; R = \{(S, \varepsilon),\; (S, aSb)\})$$

$$G \;=\; (\{a, b\},\; \{S\},\; S,\; \{(S, \varepsilon),\; (S, aSb)\})$$

Parse Trees

Definition (parse tree)

A tree t is a *parse tree* of $G = (A, N, S, R)$ iff t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of A
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n
  whose children left to right are labelled by $p_1 \dots p_k$,
  there is a rule $(n \rightarrow p_1 \dots p_k) \in R$

Parse Trees

Definition (parse tree)

A tree t is a *parse tree* of $G = (A, N, S, R)$ iff t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of A
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n
  whose children left to right are labelled by $p_1 \dots p_k$,
  there is a rule $(n \;\rightarrow\; p_1 \dots p_k) \in R$

The *yield* of parse tree t: word obtained by reading the leaves of t from left to right

Parse Trees

Definition (parse tree)

A tree t is a *parse tree* of $G = (A, N, S, R)$ iff t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of A
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n
  whose children left to right are labelled by $p_1 \dots p_k$,
  there is a rule $(n \rightarrow p_1 \dots p_k) \in R$

The *yield* of parse tree t: word obtained by reading the leaves of t from left to right

The *language* of grammar G: defined as $L(G) = \{ \mathit{yield}(t) \mid t \text{ is a parse tree of } G \}$

Parse Trees

Definition (parse tree)

A tree t is a *parse tree* of $G = (A, N, S, R)$ iff t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of A
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n
  whose children left to right are labelled by $p_1 \dots p_k$,
  there is a rule $(n \rightarrow p_1 \dots p_k) \in R$

The *yield* of parse tree t: word obtained by reading the leaves of t from left to right

The *language* of grammar G: defined as $L(G) = \{\mathit{yield}(t) \mid t \text{ is a parse tree of } G\}$

Easy to check: "is t parse tree of G?"; **harder**: "are there parse trees for word w?"

Parse Tree Example

Given grammar:

$$\begin{aligned} S &\rightarrow PQ \\ P &\rightarrow \mathrm{a} \mid \mathrm{a}P \\ Q &\rightarrow \varepsilon \mid \mathrm{a}Q\mathrm{b} \end{aligned}$$

Show a parse tree for aaaabb

Balanced Brackets Grammar

Consider language L of words made up of square brackets “[” and “]” that are balanced (each opening bracket has a matching closing bracket)

Example sequences of brackets:

- [[[ ]] [ ]] – balanced, belongs to the language
- [ ] [ ]] – not balanced, does not belong

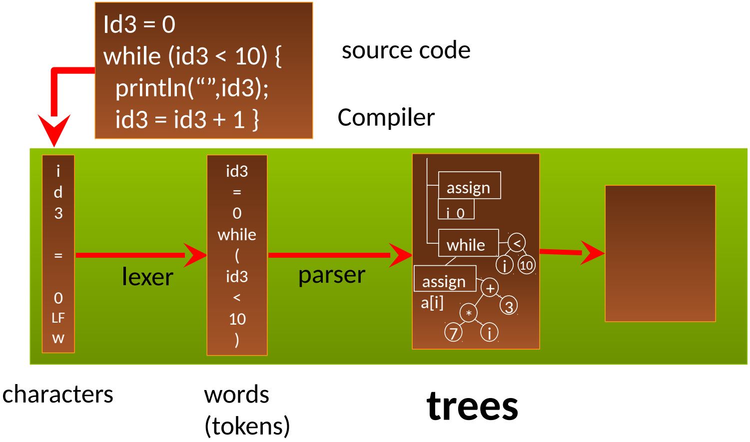
Exercise: give the grammar

Balanced Brackets Grammar

All these grammars define the same language of balanced brackets:

$$
\begin{array}{ll}
G_1 & S \rightarrow \varepsilon \mid S[S]S \\
G_2 & S \rightarrow \varepsilon \mid [S]S \\
G_3 & S \rightarrow \varepsilon \mid S[S] \\
G_4 & S \rightarrow \varepsilon \mid SS \mid [S]
\end{array}
$$

Syntax Trees



Parse Trees and Abstract Syntax Trees

Difference between parse trees and abstract syntax trees

- Node children in ***parse trees*** correspond precisely to RHS of grammar rules

  Definition of parse trees is fixed given the grammar

  Often, compilers never actually build parse trees in memory

- Nodes in ***abstract syntax tree*** (AST) contain only useful information

  We can choose our own syntax trees,
  to facilitate both construction and processing in later stages of compiler

  Compilers often directly builds ASTs

Parse Trees and Abstract Syntax Trees

Difference between parse trees and abstract syntax trees

- Node children in ***parse trees*** correspond precisely to RHS of grammar rules

  Definition of parse trees is fixed given the grammar

  Often, compilers never actually build parse trees in memory

- Nodes in ***abstract syntax tree*** (AST) contain only useful information

  We can choose our own syntax trees,
  to facilitate both construction and processing in later stages of compiler

  Compilers often directly builds ASTs

Pretty printer: takes AST and outputs the leaves of one possible parse tree

Abstract Syntax Trees for Expressions

An expression grammar:

$$
\begin{aligned}
\text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\
\text{op} &\rightarrow + \mid * \\
&\dots
\end{aligned}
$$

Abstract Syntax Trees for Expressions

An expression grammar:

$$
\begin{aligned}
\text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\
\text{op} &\rightarrow + \mid * \\
&\ldots
\end{aligned}
$$

A possible AST for it:

```
enum Expr:
  case IntLit(n: Int)
  case Var(name: String)
  case Plus(e1: Expr, e2: Expr)
  case Times(e1: Expr, e2: Expr)
```

Abstract Syntax Trees for Expressions

An expression grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\ \text{op} &\rightarrow + \mid * \\ &\ldots \end{aligned}$$

A possible AST for it:

```
enum Expr:
  case IntLit(n: Int)
  case Var(name: String)
  case Plus(e1: Expr, e2: Expr)
  case Times(e1: Expr, e2: Expr)
```

Notice: no parenthesis case; no "op"

Ambiguous Grammars

An expression grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\ \text{op} &\rightarrow + \mid * \\ &\ldots \end{aligned}$$

Ambiguous Grammars

An expression grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\ \text{op} &\rightarrow + \mid * \\ &\ldots \end{aligned}$$

Problem: how to parse "$x * 42 + y$"?

Ambiguous Grammars

An expression grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\ \text{op} &\rightarrow + \mid * \\ &\dots \end{aligned}$$

Problem: how to parse "$x * 42 + y$"?

Some token sequences have multiple parse trees ⇒ **ambiguous**

Ambiguous Grammars

An expression grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\ \text{op} &\rightarrow + \mid * \\ \ldots \end{aligned}$$

Problem: how to parse "$x * 42 + y$"?

Some token sequences have multiple parse trees ⇒ **ambiguous**

Multiple possible ASTs:

- `Times(Var("x"), Plus(IntLit(42), Var("x")))`
- `Plus(Times(Var("x"), IntLit(42)), Var("x"))`

Which is "correct"?

Ambiguous Grammars

An expression grammar:

$$
\begin{aligned}
\text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\
\text{op} &\rightarrow + \mid * \\
&\ldots
\end{aligned}
$$

Problem: how to parse "$x * 42 + y$"?

We want "$*$" to *bind stronger* than "$+$"

Ambiguous Grammars

An expression grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\ \text{op} &\rightarrow + \mid * \\ &\dots \end{aligned}$$

Problem: how to parse "$x * 42 + y$"?

We want "$*$" to *bind stronger* than "$+$",

meaning "$x * 42 + y$" is equivalent to (has the same AST as) "$(x * 42) + y$"

and *not* equivalent to "$x * (42 + y)$"

Ambiguous Grammars

An expression grammar:

$$
\begin{aligned}
\text{expr} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'} \\
\text{op} &\rightarrow + \mid * \\
&\ldots
\end{aligned}
$$

Problem: how to parse "$x * 42 + y$"?

We want "$*$" to *bind stronger* than "$+$",

meaning "$x * 42 + y$" is equivalent to (has the same AST as) "$(x * 42) + y$"

and *not* equivalent to "$x * (42 + y)$"

Can we change the grammar to reject the latter?

Layering the Grammar by Priorities

Idea: go from

$$\text{expr} \;\rightarrow\; \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'}$$

to the *layered* form

$$\begin{aligned} \text{expr} \;&\rightarrow\; \text{expr} + \text{expr} \mid \text{multi} \\ \text{multi} \;&\rightarrow\; \text{intLiteral} \mid \text{ident} \mid \text{multi} * \text{multi} \mid \text{`('expr`)'} \end{aligned}$$

Layering the Grammar by Priorities

Idea: go from

$$\text{expr} \rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'}$$

to the *layered* form

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{expr} \mid \text{multi} \\ \text{multi} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{multi} * \text{multi} \mid \text{`('expr`)'} \end{aligned}$$

Now "$x * (42 + y)$" is no longer accepted

Layering the Grammar by Priorities

Idea: go from

$$\text{expr} \rightarrow \text{intLiteral} \mid \text{ident} \mid \text{expr op expr} \mid \text{`('expr`)'}$$

to the *layered* form

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{expr} \mid \text{multi} \\ \text{multi} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{multi} * \text{multi} \mid \text{`('expr`)'} \end{aligned}$$

Now "$x * (42 + y)$" is no longer accepted

Problem: how to parse "$x + 42 + y$"?

Associativity

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{expr} \mid \text{multi} \\ \text{multi} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{multi} * \text{multi} \mid \text{`('}\,\text{expr}\,\text{`)'} \end{aligned}$$

Problem: how to parse "$x + 42 + y$"?

Associativity

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{expr} \mid \text{multi} \\ \text{multi} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{multi} * \text{multi} \mid \text{`('}\,\text{expr}\,\text{`)'} \end{aligned}$$

Problem: how to parse "$x + 42 + y$"?

Some token sequences have multiple parse trees $\Rightarrow$ **ambiguous**

Associativity

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{expr} \mid \text{multi} \\ \text{multi} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{multi} * \text{multi} \mid \text{`('}\,\text{expr}\,\text{`)'} \end{aligned}$$

Problem: how to parse "$x + 42 + y$"?

Some token sequences have multiple parse trees ⇒ **ambiguous**

Multiple interpretations:

- "$(x + 42) + y$"
- "$x + (42 + y)$"

Left Associativity – Left Recursion

Problem: how to parse "$x + 42 + y$"?

We want $+$ and $*$ to be *left-associative*

Left Associativity – Left Recursion

Problem: how to parse "$x + 42 + y$"?

We want $+$ and $*$ to be *left-associative*

Idea: keep layering the syntax...

$$
\begin{aligned}
\text{expr} &\rightarrow \text{expr} + \text{multi} \mid \text{multi} \\
\text{multi} &\rightarrow \text{multi} * \text{factor} \mid \text{factor} \\
\text{factor} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{`('expr`)'}
\end{aligned}
$$

Left Associativity – Left Recursion

Problem: how to parse "$x + 42 + y$"?

We want $+$ and $*$ to be *left-associative*

Idea: keep layering the syntax...

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{multi} \mid \text{multi} \\ \text{multi} &\rightarrow \text{multi} * \text{factor} \mid \text{factor} \\ \text{factor} &\rightarrow \text{intLiteral} \mid \text{ident} \mid \text{`('expr`)'} \end{aligned}$$

Note: we say such grammars are ***left-recursive***

because they "recurse" immediately on the left of a rule: $\textbf{expr} \rightarrow \textbf{expr} + \text{multi}$

Parsing: From Grammars to Abstract Syntax Trees

Parsing – An Old Problem

Generally performed from context-free grammars (CFGs) or *simpler subclasses* of CFGs

- LL(k) grammars such as LL(1), LL($*$)
- LR(k) grammars, SLR(1), LALR...
- ...

More recently, parsing-expression grammars (PEGs)

Many algorithms have been devised!

- Automaton-based LL(1) parsing (Lewis and Stearns, 1968)
- CYK algorithm for general CFG (Cocke, Younger and Kasami, ca 1967)
- Earley parsing for general CFG (Earley, 1970)
- Generalized LR parsing (Tomita, 1987)
- Parsing by derivatives for general CFG (Might et al., 2011)
- Packrat for PEG (Ford, 2002)
- etc. etc.

In this course: focus on LL(1), Pratt, Packrat, mention LR(1) in passing

Recursive Descent LL(1) Parsing

Recursive *descent* is a *decent* parsing technique

- Can be **easily implemented** manually based on the grammar
- **Efficient** – linear in the size of the token sequence
- Direct correspondence between grammar and code

Recursive Descent LL(1) Parsing

Recursive *descent* is a *decent* parsing technique

- Can be **easily implemented** manually based on the grammar
- **Efficient** – linear in the size of the token sequence
- Direct correspondence between grammar and code

Yet, to enable simple *recursive descent* parsing,

one might need to transform their grammar to be "LL(1)"

Recursive Descent LL(1) Parsing

Recursive *descent* is a *decent* parsing technique

- Can be **easily implemented** manually based on the grammar
- **Efficient** – linear in the size of the token sequence
- Direct correspondence between grammar and code

Yet, to enable simple *recursive descent* parsing,

one might need to transform their grammar to be "LL(1)"

In next lecture: how to parse by recursive descent *manually* and then *automatically*