Inductive Sets of Data

Overview
- Language Processors
- Inductive Specification & Regular Expressions
- Context-Free Grammar
- Proof by Induction

References

A Language Interpreter
- An interpreter consists of two parts:
  - A front end that converts program text (a program in the source language) to an abstract syntax tree (the internal representation of the program text) and
  - An evaluator (the actual interpreter) that looks at a data structure and performs some associated actions, which depend on the actual data structure. In case of a language-processing system, the interpreter takes the abstract syntax tree and converts it, possibly using external inputs, to an answer.

Examples:
- A calculator
- Bash
- Perl, Python, sh, awk, Tcl
- JVM

Execution via Interpreter

```
Initial Program Text
  ^-----------------------------------------------------------^  ^
  |                                                             |
  | Front-End                                                     |
  |                                                             |
  | Abstract Syntax Tree                                         |
  |                                                             |
  | Interpreter                                                   |
  |                                                             |
  | Read-Translate-Run-Loop                                      |
  |                                                             |
  | Input                                                        |
  |                                                             |
  | Output                                                       |
```

1
A Language Compiler

- A compiler translates program text into some other language (the target language).
- The building blocks of a compiler are:
  - A front end that converts program text (a program in the source language) to an abstract syntax tree (the internal representation of the program text).
  - A set of independent compiler phases, each has assigned a particular task in the compilation process (e.g., semantic analysis, optimization, register allocation, code emission), and
  - The evaluator of a compiled language may be an interpreter (e.g., JVM) or simply a hardware machine (e.g., von Neumann computer).

Examples of compiled languages:
- C/C++, C#
- Pascal, Java

Programming Language Values

- In the specification of programming languages we have always at least two sets of values:
  - Expressed values – values that can be specified by means of (literal) expressions in the given programming language. Examples: numbers, pairs, characters, strings
  - Denoted values – values that are bound to names. Examples: variables, parameters, procedures
Source, Host, and Target Language

- The **source language** is the language in which we write programs that should be evaluated by an interpreter or compiled by a compiler.
- The **host language** is the language in which we specify the interpreter or compiler.
- The **target language** is the language a source language is translated into by a compiler. A target language may be a higher-level programming language (e.g., C) or assembly language (or machine language).

Lexical Tokens

- A lexical token is a sequence of characters, which are treated as a unit in the definition of a programming language. A programming language classifies lexical tokens into a finite set of token types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td><code>foo NIL symbol</code></td>
</tr>
<tr>
<td>Integer</td>
<td>0 239495</td>
</tr>
<tr>
<td>Real</td>
<td>.002 3.141 2E45 4.4E-67</td>
</tr>
<tr>
<td>Keywords</td>
<td><code>if while return class</code></td>
</tr>
<tr>
<td>Operators</td>
<td><code>+ - &gt;&gt; &amp;&amp; =</code></td>
</tr>
<tr>
<td>Punctuations</td>
<td><code>{} . ; { }</code></td>
</tr>
</tbody>
</table>

Specifying Token Recognition

- We can use any programming language to specify an ad hoc scanner.
- However, we will specify lexical tokens using the formal language of **regular expressions** and implement token recognition using **finite deterministic automata (DFA)**.
Finite Deterministic Automata

A deterministic automaton is a quintuple $(\Sigma, Q, q_0, \sigma, F)$ with:

- a set $\Sigma$ of actions (sometimes called an alphabet),
- a set $Q = \{q_0, q_1, \ldots\}$ of states,
- a subset $F \subseteq Q$ called the accepting states,
- a subset $\sigma$ of $Q \times \Sigma \times Q$ called the transitions,
- a designated start state $q_0$.

A transition $(q, a, q') \in \sigma$ is usually written $q \xrightarrow{a} q'$.

The automaton $A$ is said to be finite if $Q$ is finite.

Transition Graph

- An automaton is usually represented by a transition graph, whose nodes are states and whose arcs are transitions.

Example: $A_0$ is a finite automaton over the alphabet $\Sigma = \{a,b,c\}$:

Machine Table

- We can describe $A_0$ formally also by writing $A = (\Sigma, Q, q_0, \sigma, F)$, where:
  - $\Sigma = \{a,b,c\}$,
  - $Q = \{q_0, q_1, q_2, q_3\}$,
  - $\sigma$ is described as
  
  $\begin{array}{c|ccc}
  \text{Input} & a & b & c \\
  \hline
  q_0 & q_1 & q_2 & q_3 \\
  q_1 & q_1 & q_2 & q_3 \\
  q_2 & q_1 & q_2 & q_3 \\
  q_3 & q_1 & q_2 & q_3 \\
  \end{array}$

- $q_0$ is the start state, and
- $F = \{q_2\}$.
Language of an Automaton

- A string over an alphabet $\Sigma$ is a finite sequence of symbols from that alphabet, usually written to one another and not separated by any characters. If $w$ is a string over $\Sigma$, the length of $w$, written $|w|$, is the number of symbols that it contains. The string of length zero is called the empty string and is denoted by $\varepsilon$.

- Let $A$ be an automaton over $\Sigma$, and $s = a_1 \ldots a_n$ a string over $\Sigma$. Then $A$ is said to accept $s$ if there is a path in $A$, from $q_0$ to an accepting state, whose arcs are labeled successively $a_1, \ldots, a_n$. The language of $A$, denoted by $L(A)$, is the set of strings accepted by $A$.

- The language $L(A)$ of any finite-state automaton $A$ is regular.

Sets

- A set is a collection of elements (or values), possibly empty.

- All elements satisfy a possibly complex characterizing property. Formally, we write:
  $$ \{ x \in A \mid P(x) = \text{True} \} $$
  to define a set, where all elements satisfy the property $P$.

- The basic axiom of set theory is that there exists an empty set, $\emptyset$, with no elements. Formally,
  $$ \forall x, x \notin \emptyset $$
  In words, "for every $x$, $x$ is not an element of $\emptyset$".

Inductive Specification

- Sometimes it is difficult to define a set explicitly, in particular if the elements of the set have a complex structure.

- However, it may be easy to define the set in terms of itself. This process is called inductive specification or recursion.

Example:

Let the set $S$ be the smallest set of natural numbers satisfying the following two properties:

- $0 \in S$, and
- Whenever $x \in S$, then $x + 3 \in S$.

The first property is called base clause and the second property is called inductive/recursive clause. An inductive specification may have multiple base and inductive clauses.
The “Smallest Set”

- If we use inductive specification, we always define the smallest set that satisfies all given properties. That is, inductive specification is free of redundancy.
- It is easy to see that there can only be one such set:
  If $S_1$ and $S_2$ both satisfy all given properties, and both are the smallest, then we have $S_1 \subseteq S_2$ (since $S_1$ is the smallest), and $S_2 \subseteq S_1$ (since $S_2$ is the smallest), hence $S_1 = S_2$.

Induction

- Having defined set inductively, we can use the inductive definition to prove properties about members of the set.
- The proof technique used is called mathematical induction.
- The most common forms are induction on the structure of expressions and induction on the length or structure of proofs.
- A simple and intuitive way to think of induction is that it is a method for writing down an infinite proof in a finite way.
  Note: We can construct infinitely many values from a given inductive specification.

Well-formed Formulae

- Well-formed formulae for compound Boolean propositions are defined as follows:
  - True and False are well-formed formulae;
  - $p$, where $p$ is a propositional variable, is a well-formed formula;
  - $(\neg p)$ is a well-formed formula, if $p$ is a well-formed formula,
  - $(p \land q)$, $(p \lor q)$, $(p \rightarrow q)$, $(p \leftrightarrow q)$ are well-formed formulae, if both $p$ and $q$ are well-formed formulae.

Examples:
  - $p \rightarrow q$, $(p \rightarrow q) \leftrightarrow ((\neg p) \lor q)$
Mathematical Induction

1. A proof by mathematical induction that a given property $P$ is true for every positive integer $n$, we write $P(n)$, consists of two steps:
   1) Basic step:
      The proposition $P(1)$ (or $P(0)$) is shown to be true.
   2) Inductive step:
      The implication $P(n) \Rightarrow P(n+1)$ is shown to be true for every positive integer $n$.

Note: In a proof by mathematical induction it is not assumed that $P(n)$ is true for all positive integers! It is only shown that if it is assumed that $P(n)$ is true, then $P(n+1)$ is also true. In general, we use an inference rule called "Modus ponens" ($P \Rightarrow Q \wedge P \vdash Q$) to show this.

Example: $\text{sum}(n) = \frac{n(n+1)}{2}$

As a young boy, the later mathematician Carl Friedrich Gauss was asked by his teacher to add up the first hundred numbers, in order to keep him quiet for a while. As we know today, this did not work out, since:

$\text{sum}(n) = \frac{n(n+1)}{2}$

Proof:
1) Base case: We must show that $\text{sum}(0) = 0(0+1)/2$. This is an easy calculation, and we have $\text{sum}(0) = 0$.

2) Inductive step: Assume $\text{sum}(n) = \frac{n(n+1)}{2}$ holds. We must show that $\text{sum}(n+1) = \frac{(n+1)(n+2)}{2}$ holds as well. First, $\text{sum}(n+1)$ is just the sum of the first $n$ numbers plus $(n+1)$. Therefore, we have

$\text{sum}(n+1) = \text{sum}(n) + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n^2 + n + 2n + 2)}{2} = \frac{(n+1)(n+2)}{2}$

as required.

Structural Induction

Structural induction uses the fact that substructures of a given object are always smaller than the object itself.

Structural induction is done as follows:
1) Base step: The induction hypothesis is true on simple structures (those without substructures).
2) Induction step: If the induction hypothesis is true on the substructures of a given object, say $s$, then it is true on $s$ itself.
Regular Sets

- Operations for building sets of strings:
  - Alternation
    $S_1 \cup S_2 = \{ s | s \in S_1 \lor s \in S_2 \}$
  - Concatenation
    $S_1 \cdot S_2 = \{ s_1 s_2 | s_1 \in S_1, s_2 \in S_2 \}$
  - Iteration
    $S^* = \{ \epsilon \} \cup S \cdot S \cdot S \cdot S \cdots = S_0 \cup S_1 \cup S_2 \cup S_3 \cup \cdots$

- A set of strings over $\Sigma$ is said to be regular if it can be built from the empty set $\emptyset$ and the singleton set $\{a\}$ (for each $a \in \Sigma$), using just the operations of alternation, concatenation, and iteration.

Arden’s Rule

- The following equations hold:
  - $(S_1 \cdot S_2) \cdot S_3 = S_1 \cdot (S_2 \cdot S_3)$
  - $(S_1 \cup S_2) \cdot T = S_1 \cdot T \cup S_2 \cdot T$
  - $S \cdot \epsilon = S$
  - $S \cdot (T \cdot S)^* = (S \cdot T)^* \cdot S$

- Note: $\emptyset$ means "no path", whereas $\epsilon$ means "empty path".

Arden’s Rule:

For any two sets of strings $S$ and $T$, the equation $X = S \cdot X \cup T$ has $X = S^* \cdot T$ as a solution. Moreover, this solution is unique if $\epsilon \notin S$.

Proof by induction on the length of $(T \cdot S)^*$:

1. Case 0:
   - We have $S \cdot \epsilon = S$.
   - We have $S \cdot \epsilon = S$. The result follows by a simple sub-induction to establish the fact that $\epsilon$ is a left-neutral element for $\cdot$.

2. Case n:
   - We assume $S \cdot (T \cdot S)^n = (S \cdot T)^n \cdot S$.
   - We need to show that $S \cdot (T \cdot S)^{n+1} = (S \cdot T)^{n+1} \cdot S$.

\[
\begin{align*}
S \cdot (T \cdot S)^n \cdot (T \cdot S) &= S \cdot (T \cdot S)^{n+1} \\
&= \epsilon \cdot (T \cdot S)^n \cdot T \cdot S \\
&= S \cdot T \cdot S^n \cdot T \cdot S \\
&= S \cdot T \cdot S^n \cdot T \\
&= S \cdot T \cdot (S \cdot T)^n \cdot S \\
&= S \cdot T \cdot (S \cdot T)^n \\
&= S \cdot T \cdot (S \cdot T)^n \cdot S \\
&= S \cdot (T \cdot S)^{n+1} \cdot S
\end{align*}
\]
## Regular Expression Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td>An ordinary character stands for itself.</td>
</tr>
<tr>
<td><code>ε</code></td>
<td>The empty string (we write <code>ε</code>).</td>
</tr>
<tr>
<td>`M</td>
<td>N`</td>
</tr>
<tr>
<td><code>M*</code></td>
<td>Repetition, zero or more times.</td>
</tr>
<tr>
<td><code>M+</code></td>
<td>Repetition, one or more times.</td>
</tr>
<tr>
<td><code>M?</code></td>
<td>Optional, zero or one occurrence of <code>M</code>.</td>
</tr>
<tr>
<td><code>[a-zA-Z]</code></td>
<td>Character set alternation.</td>
</tr>
<tr>
<td><code>\.</code></td>
<td>A period stands for any single character (except newline).</td>
</tr>
</tbody>
</table>

### Examples of Regular Expressions

- `error`:
  ```
  \( //.*\n) | (|\n|\t) |
  ```
- `no token, just white space`
  ```
  \[0-9]+\.|[0-9]*\.|[0-9]+ |
  ```
- `token real`
  ```
  \[0-9]+\.|[0-9]*\.|[0-9]+ |
  ```
- `token integer`
  ```
  [a-zA-Z]\[a-zA-Z0-9]* |
  ```
- `token identifier`
  ```
  [a-zA-Z]\[a-zA-Z0-9]* |
  ```
- `keyword if`
  ```
  if |
  ```
- `token identifier`
  ```
  [a-zA-Z]\[a-zA-Z0-9]* |
  ```
- `token integer`
  ```
  [0-9]+ |
  ```
- `token real`
  ```
  [0-9]+\.|[0-9]*\.|[0-9]+ |
  ```
- `error`
  ```
  \[0-9]+\.|[0-9]*\.|[0-9]+ |
  ```
- `no token, just white space`
  ```
  \[0-9]+\.|[0-9]*\.|[0-9]+ |
  ```

## Finite Automata for Tokens

![Finite Automata for Tokens Diagram](image)
Disambiguation Rules

- Longest match:
  - The longest initial substring of the input that can match any regular expression is taken as the next token.
    - "596.354" => real, "85893a" => integer (85893)
- Rule priority:
  - For a particular longest initial substring, the first regular expression that can match determines its token type.
    - "if" => keyword if but only if we have defined the regular expression for keyword if before expression for identifier

Finite Nondeterministic Automata

A finite nondeterministic automaton is a quintuple \((\Sigma, Q, q_0, \sigma, F)\) with:
- a set \(\Sigma\) of actions (sometimes called an alphabet),
- a set \(Q = \{q_0, q_1, \ldots\}\) of states,
- a subset \(F\) of \(Q\) called the accepting states,
- a subset \(\sigma\) of \(\Sigma \times P(Q)\) called the transitions,
- a designated start state \(q_0\).

DFA’s versus NFA’s

An automaton is deterministic if for each pair \((q, a) \in Q \times \Sigma\) there is exactly one transition \(q \cdot a \rightarrow q'\).

Deterministic automaton: non-deterministic automaton:
Example

Deterministic system $S_1$:

Non-deterministic system $S_2$:

Are both systems equivalent?

Machine Tables

Deterministic system $S_1$:

Non-deterministic system $S_2$:

$$
\begin{array}{|c|c|c|c|}
\hline
S & q & a & b & c \\
\hline
S_1 & q_0 & q_1 & q_2 & q_3 \\
S_2 & q_0 & q_1 & q_2 & q_3 \\
\hline
\end{array}
$$

$b = b \\
c = c \\
c_0 = c_0 \\
S_1 = S_2$?

The systems $S_1$ and $S_2$ are language-equivalent. The accepted language is $L(S_1) = L(S_2) = (a \cdot (b \cup a \cdot c))^*$. 

11
**Language Equivalence**

- Language equivalence is blind for non-determinism. Thus, each non-deterministic automaton can be transformed into an equivalent deterministic one.

- However, language equivalence is blind for deadlocks. That is, the equivalence definition has only limited applicability.

**Combining DFA’s**

We can define a new NFA to combine the DFA’s defined for the lexical tokens. That is, we introduce a new start and end state and connect them with their corresponding states in the DFA’s through an ε transition.

Note that we also have to rename the states in the DFA’s in order to make them unique.

**Example**

---

---
Conversion of an NFA into a DFA

Multi-valued transitions make it hard to implement an NFA, since most computers do not have good "guessing" hardware.

The problem: ε transitions

The solution: We have to find a deterministic ε-free automaton

Proposition: For each finite automaton there exists an equivalent ε-free automaton.

The approach: We can compute the ε-closure for each reachable state. By combining the states in the ε-closure, that is, defining a new state, we can construct from an NFA a DFA that accepts the same language. (This is done automatically by lex, flex, etc.)

Syntax Analysis

The role of the parser:

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. That is, parsing the string of tokens succeeds if and only if there exists a derivation sequence from the start symbol of the grammar to the string of tokens.

The result of the syntax analysis is an abstract syntax tree, which is being used by the interpreter/compiler to verify program properties, perform program optimizations, and generate target code or execute the program.

Need for more Flexibility

Valid regular expression specification:

```
<Digits> = [0-9]  
<Sum> = (<Digits> +) <Digits>
```

```
28+301+9
```

Invalid regular expression specification:

```
<Digits> = [0-9]  
<Sum> = <Exp> + <Exp>  
<Exp> = "(" <Sum> ")" | <Digits>
```

We cannot construct a DFA for the second specification!
Pumping Lemma for Regular Languages

If \( L \) is a regular language (e.g., the language of a finite-state automaton), then there is a number \( p \) (the pumping length) where, if \( x \) is any string in \( L \) of length at least \( p \), then \( x \) may be divided into three pieces, \( x = xyz \), satisfying the following conditions:

1. For each \( i \geq 0 \), \( xy^i z \in L \),
2. \(|y| > 0 \), and
3. \(|xy| \leq p \).

When \( x \) is divided into \( xyz \), either \( x \) or \( z \) may be \( \epsilon \), but condition 2 says that \( y \neq \epsilon \). Condition 3 states that the pieces \( x \) and \( y \) together have length at most \( p \). This is an extra condition that is very useful when proving that a given language is not regular.

Is \( L = \{a^n b^n \mid n > 0\} \) regular?

Let \( L \) be the language \( \{a^n b^n \mid n > 0\} \). We use the pumping lemma to prove that \( L \) is not regular. The proof is by contradiction.

Assume \( L \) is regular. Let \( p \) be the pumping length given by the pumping lemma. Choose \( s \) to be the string \( a^p b^p \). Because \( s \) is a member of \( L \) and \( s \) has length more than \( p \), the pumping lemma guarantees that \( s \) can be divided into three pieces \( s = xyz \) where \( |xyz| \leq p \) and \( |y| > 0 \).

Consider the three cases to show that this is impossible.

- The string \( y \) consists only of \( a \)'s. Thus, \( a^p b^p = xyz \) with \( x = a^r \), \( y = a^s \), and \( z = a^{t}b^{p} \). According to the case 1 of the pumping lemma, it is not required that all strings \( a^r \) with \( i \geq 0 \) have to be elements of \( L \). But this is true for \( i = 0 \), hence a contradiction.

- The string \( y \) consists only of \( b \)'s. This case gives also a contradiction.

- The string \( y \) consists of both \( a \)'s and \( b \)'s. Thus, \( a^p b^p = xyz \) with \( x = a^r \), \( y = a^s b^t \), and \( z = b^{u} \). According to the case 1 of the pumping lemma, it is not required that all strings \( a^r b^s \) with \( i \geq 0 \) have to be elements of \( L \). But this is not true for \( i = 0 \), hence a contradiction. Q.E.D.

Grammar

Definition:
- A vocabulary (or alphabet) \( V \) is a finite, nonempty set of symbols. A sentence over \( V \) is a string of finite length of elements of \( V \). The empty string, denoted by \( \epsilon \), is the string containing no symbols. The set of all sentences over \( V \) is denoted by \( V^* \).

Definition:
- A grammar is a quadruple \( G = (V, T, S, P) \) with
  - \( V \), a vocabulary,
  - \( T \subset V \) consisting of terminal elements (tokens),
  - \( S \in (V - T) \), and
  - \( P \subseteq (V^* - T^*) \times V^* \).
Language

Definition: Let \( G = (V, T, S, P) \) be a grammar. The language generated by \( G \) (or the language of \( G \)), denoted by \( L(G) \), is the set of all strings of terminals that are derivable from the start symbol \( S \). In other words,

\[
L(G) = \{ w \in T^* | S \Rightarrow^+ w \}
\]

Types of Grammars

- Type 0: A grammar that has no restrictions on its productions.
- Type 1 – context-sensitive: A grammar can only have productions of the form \((w_1, w_2)\), where the length of \( w_2 \) is greater than or equal to the length of \( w_1 \), or the form \((w_1, \varepsilon)\).
- Type 2 – context-free: A grammar can only have productions of the form \((w_1, w_2)\), where \( w_1 \) is a single nonterminal.
- Type 3 - regular: A grammar can only have productions of the form \((w_1, w_2)\), with \( w_1 \) is a nonterminal and \( w_2 \) is either \( aB \), \( Ba \), \( a \), or \( \varepsilon \), where \( B \) is a nonterminal and \( a \) is a terminal.

\[ L = \{a^n b^n | n > 0\} \text{ Revisited} \]

Let \( G = (\{s, a, b\}, \{a, b\}, s, \{(s, asb), (s, ab)\}) \) be a grammar. Then the language generated by \( G \) is

\[
L(G) = \{a^n b^n | n > 0\}.
\]

Furthermore, the production \((s, asb)\) makes \( G \) a context-free grammar. Therefore, \( L(G) \) is a context-free language.
The Backus-Naur Form

- In general, we specify the grammar of a programming language using a context-free grammar (or type 2 grammar).
- We use a notation called Backus-Naur Form (BNF).
- The general rule format is:

```latex
\text{lhs} ::= \text{rhs}
```

where \text{lhs} is a nonterminal, and \text{rhs} may be a list, separated with "|" of strings of terminals and nonterminals.

Using BNF

In BNF, nonterminals are often enclosed in brackets.

Example:

```latex
\text{list-of-numbers} ::= () | (\text{number} : \text{list-of-numbers})
```

Note: In BNF, some nonterminals (e.g. \text{number}) are left undefined, when their meaning is sufficiently clear from the context.

Kleene Star

- The Kleene star, written (\ldots)^{\ast}, is used to specify a sequence of any number of instances of a given string.

Example:

```latex
\text{a-list} ::= (\text{symbol-expression})^{\ast}
\text{symbol-expression} ::= \text{symbol} | \text{a-list}
\text{a-list} ::= () | (a b c | \text{fun1} (\text{fun2 arg1 arg2 arg3 arg4})
```

-
Kleene Plus

The Kleene plus, written \( \{ \ldots \}^+ \), is used to specify a sequence of one or more instances of a given string.

Example:

\[
\begin{align*}
\text{<nonempty-list>} & ::= \{ \text{<datum>} \}^+ \\
\text{<datum>} & ::= \text{<number>} | \text{<symbol>} | \text{<string>} \\
\end{align*}
\]

\[(a \ b \ "HIT3315") \}
\]

Separated List Notation

The separated list notation, written \( \{ \ldots \}^* \) or \( \{ \ldots \}^+ \), can be used to specify any number of instances of a given string that are separated with a non-empty character sequence.

Example:

\[
\begin{align*}
\text{<list-of-expressions>} & ::= \{ \text{<expression>} \}^* \\
\end{align*}
\]

\[(1 , 2 , 3) \]

From BNF to a Program

With the help of BNF rules, starting with simple members of a data set, we are able to specify inductively complex data structures.

We can use the same approach to construct programs that manipulate these data structures.

First we define the program’s behavior on simple inputs, and then we use this behavior to build inductively programs that can process with more complex arguments.
Exponentiation

- Consider the problem of computing the integer exponential of a given integer number.

- A program that tackles this problem should take as arguments a base b and a positive integer exponent n and compute $b^n$.

- In general, $e(b, n) = \begin{cases} 1 & n = 0 \\ b \cdot e(b, n-1) & n > 0 \end{cases}$

Is $e(b, n) = b^n$ correct?

- To show that $e(b, n) = b^n$ is indeed correct, we proceed by induction on n:

  - Base step: $n = 0$
    Then we have $e(b, 0) = 1 = b^0$.

  - Induction Step:
    Assume $e(b, n) = b^n$ is correct.
    We must show that $e(b, n+1) = b^{n+1}$. By the definition of e, it holds that $e(b, n+1) = b \cdot e(b, n)$. Using the induction hypothesis, we have $e(b, n+1) = b \cdot b^n = b^{n+1}$, as desired.

Q.E.D.

Procedure Exponential

- In C, the procedure for $e(b, n)$ is defined as follows:

  ```c
  int exponential(int b, int n) {
    if (n == 0)
      return 1;
    else
      return b * exponential(b, n - 1);
  }
  ```

- The two branches of the if expression correspond to the two cases of the inductive definition of $e(b, n)$.

- If we can reduce a given problem to a sub-problem, then we can recursively call the procedure that solves the original problem to solve the sub-problem.
Exponentiation with negative Exponent

```c
int exponential(int b, int n) {
    if (n == 0)
        return 1;
    else
        if (n < 0)
            return (1 / b) * exponential(b, n + 1);
        else
            return b * exponential(b, n - 1);
}
```

- This procedure works on all integers (including negative integers).
- It holds that \( b^{-n} = \frac{1}{b^n} \) for all integers \( n \). Moreover, we can use inductive program specification, since \( b^{-n} = \frac{1}{b} \cdot \frac{1}{b^{n-1}} \).

Recursion

- If a procedure that contains within its body calls to itself, then this procedure is called to be recursively defined.
- This approach of program specification is called recursion and is found not only in programming.
- If we define a procedure recursively, then there must exist at least one sub-problem that can be solved directly, that is without calling the procedure again.
- Note: A recursively defined procedure must always contain a directly solvable sub-problem. Otherwise, this procedure does not terminate.

Rule of Thumb

- When defining a program based on structural induction, the structure of the program must be patterned according the structure of the data.
- In general, this means that we have to define one procedure for each syntactic category used to specify our data. Then each procedure has to examine the input to see, which right-hand-side it corresponds to. Furthermore, for every nonterminal that appears in the right-hand-side, there will be a recursive call to a procedure for that nonterminal. This approach is also called recursive-descent-parsing.
Always Remember

FOLLOW THE GRAMMAR

Syntax Tree

\[ G = ( V = \{ <Expression>, <Term>, <Number>, +, -, *, / \}, \]
\[ T = \{ <Number>, +, -, *, / \}, \]
\[ S = <Expression>, \]
\[ P = \{ (<expression> := <expression> (+|-) <term>), (<expression> := <term>), (<term := <term> (*|/) <number>), (<term> := <number>) \} \]

\[ <Expression> \rightarrow <Expression> + <Term> \]
\[ <Expression> \rightarrow <Term> \]
\[ <Term > \rightarrow <Term> (x|/) <Number> \]
\[ <Term> \rightarrow <Number> \]

Derivation

\[ 4x^2+1 \in L(G) \]
Ambiguous Grammars

A grammar is ambiguous if we can derive a sentence $s \in L(G)$ with two or more different syntax trees.

Consider:

- $\text{Expression} \ ::= \text{Identifier}$
- $\text{Expression} \ ::= \text{Number}$
- $\text{Expression} \ ::= \text{Expression} + \text{Expression}$
- $\text{Expression} \ ::= \text{Expression} - \text{Expression}$
- $\text{Expression} \ ::= \text{Expression} \times \text{Expression}$
- $\text{Expression} \ ::= \text{Expression} / \text{Expression}$
- $\text{Expression} \ ::= (\text{Expression})$

Consider:

- $4 \times 2 + 1$

Which syntax tree is the correct one?

Consider:

- $4 + 2 + 1$

Which syntax tree is the correct one?
**Precedence & Associativity**

**Precedence:**
An operator has a higher precedence if the syntax rule that defines its application occurs nested in another rule. That is, an operator binds tighter if its syntax rule is defined deeper in the hierarchy.

**Associativity:**
An operator associates to the left if the syntax rule that defines its application uses a left-recursive rule structure. An operator associates to the right if the syntax rule that defines its application uses a right-recursive rule structure. Non-recursive rules result in operators with no associativity.

---

**An Expression Language**

- `<Expressions> ::= <Expressions> + <Term>`
- `<Expressions> ::= <Expressions> - <Term>`
- `<Expressions> ::= <Term>`
- `<Term> ::= <Term> x <PrimaryExpression>`
- `<Term> ::= <Term> / <PrimaryExpression>`
- `<Term> ::= <PrimaryExpression>`
- `<PrimaryExpression> ::= <Identifier>`
- `<PrimaryExpression> ::= <Number>`
- `<PrimaryExpression> ::= { <expression> }

---

**4 x 2 + 1**

By precedence
4 + 2 + 1

By associativity