2 Document Grammars and Instances

A look at the foundations of hierarchical document structures

2.1 Language-Theoretic Basis (a quick review)
- regular expressions
- extended context-free grammars (as document schemas) and
- parse trees (as document instances)

2.2 Review of XML Basics
- Practical realisation of the general model: XML document instances and DTDs

Regular Expressions: Syntax
- A regular expression over an alphabet \( \Sigma \) is either
  - \( \emptyset \) an empty set,
  - \( \lambda \) lambda (sometimes epsilon),
  - \( a \) any alphabet symbol \( a \in \Sigma \),
  - \((R | S)\) choice; sometimes \((R \cup S)\),
  - \((R S)\) concatenation, or
  - \( R^* \) Kleene closure or iteration,
where \( R \) and \( S \) are regular expressions
N.B: different syntaxes exist but the idea is same

Regular Expressions: Grouping
- Conventions to simplify operator expressions
  - outermost parentheses may be eliminated:
    \( E \) = \( (E) \)
  - binary operations are associative:
    \( (A \cup (B \cup C)) = ((A \cup B) \cup C) \)
  - operations have priorities:
    - iteration first, concatenation next, choice last
    - for example, \( (A \cup (B \cup C)^*)) = A \cup (B \cup C) \)

Regular Expressions: Semantics
- A regular expression \( E \) denotes a language (set of strings) \( L(E) \), defined inductively as follows:
  - \( L(\emptyset) = \{\} \) (empty set)
  - \( L(\lambda) = \{\lambda\} \) (singleton set of empty string \( \lambda \))
  - \( L(a) = \{a\} \) (singleton set of word \( a \))
  - \( L(R | S) = L(R) \cup L(S) = \{w \mid w \in L(R) \text{ or } w \in L(S)\} \)
  - \( L(R S) = L(R) L(S) = \{xy \mid x \in L(R) \text{ and } y \in L(S)\} \)
  - \( L(R^*) = L(R)^* = \{x_1 \ldots x_n \in L(R) \mid n \geq 0\} \)

Regular Expressions: Examples
- \( L(A | B (C D)^*) = ? \)
  - \( L(A) \cup L(B (C D)^*) \)
  - \( \{A\} \cup L(B) L(C (D)^*) \)
  - \( \{A\} \cup \{\lambda, CD, CDCCD, CDCCCD, \ldots\} \)
  - \( \{A, B, BCD, BCDCCD, BCDCCCD, \ldots\} \)
Regular Expressions: Examples

- Simplified top-level structure of a document:
  - $\Sigma = \{\text{title, auth, date, sect}\}$
  - title followed by an optional list of authors, fby an optional date, fby one or more sections: $\text{title auth* (date | \lambda) sect*}$

- Commonly used abbreviations:
  - $E^* = (E | \lambda)$; $E^+ = E E^*$
  - the above more compactly: $\text{title auth* date? sect+}$

Context-Free Grammars (CFGs)

- Used widely to syntax specification (programming languages, XML, …) and to parser/compiler generation (e.g. YACC/GNU Bison)
- CFG $G$ formally a quadruple $(V, \Sigma, P, S)$
  - $V$ is the alphabet of the grammar $G$
  - $\Sigma \subseteq V$ is a set of terminal symbols
  - $N = V - \Sigma$ is a set of nonterminal symbols
  - $P$ set of productions
  - $S \in V$ the start symbol

Productions and Derivations

- Productions: $A \rightarrow \alpha$, where $A \in N, \alpha \in V^*$
  - E.g. $A \rightarrow aBa$ (production 1)

- Let $\gamma, \delta \in V^*$. String $\gamma$ derives $\delta$ directly, $\gamma \Rightarrow \delta$, if
  - $\gamma = \alpha A \beta, \delta = \alpha \omega \beta$ for some $\alpha, \beta \in V^*$, and $A \rightarrow \omega \in P$
  - E.g. $A \Rightarrow AaBa$ (assuming production 1 above)

- NB: CFGs are often given simply by listing the productions ($P$); The start symbol ($S$) is then conventionally the left-hand-side of the first production

Language Generated by a CFG

- $\gamma$ derives $\delta$, $\gamma \Rightarrow \delta$, if there’s a sequence of (0 or more) direct derivations that transforms $\gamma$ to $\delta$

- The language generated by a CFG $G$:
  - $L(G) = \{w \in \Sigma^* | S \Rightarrow^* w \}$

- NB: $L(G)$ is a set of strings;
  - To model document structures, we consider syntax trees

Syntax Trees

- Also called parse trees or derivation trees
- Ordered trees
  - consist of nodes that may have child nodes which are ordered left-to-right
- nodes labelled by symbols of $V$:
  - internal nodes by nonterminals, root by start symbol
  - leaves by terminal symbols (or empty string $\lambda$)
- A node with label $A$ can have children labelled by $X_1, \ldots, X_k$ only if $A \rightarrow X_1, \ldots, X_k \in P$

Syntax Trees: Example

- CFG for simplified arithmetic expressions:
  - $V = \{E, +, *, I\}; \Sigma = \{+, *, I\}; N = \{E\}; S = E$
  - $P = \{E \rightarrow E+E, E \rightarrow E*E, E \rightarrow I, E \rightarrow (E)\}$

  - Syntax tree for $2*(3+4)$?
Syntax Trees: Example

```
      E
     / \   
    /   \  
  E   E   E
     \   /  
      \ /   
        E
```

CFGs for Document Structures

- Nonterminals represent document elements
  - E.g. model for items (Ref) of a bibliography list:
    - `Ref` → `AuthorList Title PublData`
    - `AuthorList` → `Author AuthorList`
    - `AuthorList` → `λ`
- Notice:
  - right-hand-side of a production is a fixed string of grammar symbols
  - Repetition simulated using recursion
    - e.g. `AuthorList` above

Example: List of Three Authors

```
AuthorAuthorAuthorTitlePublData
Aho Hopcroft Ullman
```

Problems

- "Auxiliary nonterminals" (like `AuthorList`) obscure the model
  - the last `Author` several levels apart from its intuitive parent element `Ref`
  - awkward to access and to count Authors of a reference
  - avoided by extended context-free grammars

Extended CFGs (ECFGs)

- like CFGs, but right-hand-sides of productions are regular expressions over V
  - E.g.: `Ref` → `Author* Title PublData`
- Let γ, δ ∈ V+. String γ derives δ directly, γ => δ, if
  - γ = αAβ, δ = αωβ for some α, β ∈ V*, and A → E ∈ P such that ω ∈ L(E)
  - E.g. `Ref` => `Author Author Author Title PublData`

Language Generated by an ECFG

- `L(G)` defined similarly to CFGs:
  - γ derives δ, γ => δ, if
  - γ = α₁ => . . . => αₙ = δ (for n ≥ 0)
  - `L(G) = { w ∈ Σ* | S =>* w }
- Theorem: Extended and ordinary CFGs allow to generate the same languages.
- Syntax trees of ECFGs and CFGs differ! (Next)
Syntax Trees of an ECFG

- Similar to parse trees of an ordinary CFG, except that...
  - node with label A can have children labelled $X_1, \ldots, X_k$ when $A \rightarrow E \in P$ such that $X_1, \ldots, X_k \in L(E)$
  - an internal node may have arbitrarily many children (e.g., Authors below a Ref node)

Example: Three Authors of a Ref

```
Ref
  Author Author Author
  Aho Hopcroft Ullman
  The Design and Analysis ...

Ref -> Author* Title PublData ∈ P
Author Author Author Title PublData ∈ L(Author* Title PublData)
```

Terminal Symbols in Practise

- (Extended) CFGs:
  - Leaves of parse trees are labelled by single terminal symbols ($\in \Sigma$)
  - Too granular for practise; instead terminal symbols which stand for all values of a type
    - XML DTDs: #PCDATA for variable length string content
    - Proposed XML schema formalisms:
      - a string, byte, integer, boolean, date, ...
    - Explicit string constants rare in document grammars

2.2 XML, eXtensible Markup Language

- W3C Recommendation 10-Feb-1998
  - not an official standard, but a stable industry standard
    - a revision, not a new version of XML
  - what is said below about valid XML documents applies to SGML documents, too

What is XML?

- Extensible Markup Language is not a markup language!
  - does not fix semantics or a tag set (like, e.g., HTML does)
- A way to use markup to represent information
- A metalanguage
  - supports definition of specific markup languages
    - E.g. XHTML a reformulation of HTML using XML

Next: Essential Features of XML

- An overview of the essentials of XML
  - many details skipped
    - some to be discussed in exercises or with other topics when the need arises
  - learn to consult the original sources (specifications, documentation etc) for details
XML Encoding of Structure

- XML document essentially a parenthesized linear encoding of a parse tree (see next)
  - corresponds to a pre-order walk
  - start of inner node (or element) A denoted by a start tag `<A>`, end denoted by end tag `</A>`
  - leaves are strings (or empty elements)
- certain extensions (especially attributes)

XML Encoding of Structure: Example

```
<S>
  S
  W
  W
  world!
  Hello
</S>
```

An XML Processor (Parser)

- Reads XML documents
- Passes data to an application
- XML Recommendation
  - tells how to read, what to pass
  - check the XML Rec for details; quite readable!

XML: Logical Document Structure

- Elements
  - correspond to internal nodes of the parse tree
  - unique root element `->` document a single parse tree
  - indicated by matching (case-sensitive!) tags
  - can contain text and/or subelements
  - can be empty:
    `<elem-type></elem-type>` OR (e.g.)
    `<br/>`

Logical document structure (2)

- Attributes
  - name-value pairs attached to elements
  - "metadata", usually not treated as content
  - in start-tag after the element type name
    `<div class="preface" date='990126'>`...
- Also:
  - `<!-- comments outside other markup -->`
  - `<?note this would be passed to the application as a processing instruction named 'note'?>`

CDATA Sections

- "CDATA Sections" to include XML markup characters as textual content
  ```
  <![CDATA[
  Here we can easily include markup characters and, for example, code fragments:
  <example>if (Count < 5 && Count > 0) {
    <example>
  ]]>```

Two levels of correctness

- **Well-formed** documents
  - roughly: follows the syntax of XML,
  markup correct (elements properly nested, tag
  names match, attributes of an element have
  unique names, ...)
  - violation is a fatal error
- **Valid** documents
  - (in addition to being well-formed)
  obey their **document type definition**

Document Type Declaration

- Provides a grammar (**document type definition, **
  **DTD**) for a class of documents
- Syntax
  ```xml
  <!DOCTYPE rootElemType SYSTEM "ex.dtd"[
    <!-- "external subset" in file ex.dtd -->
    ! [ <!-- "internal subset" may come here -->
    ]>
  ]>
  ```
  - **DTD** is the union of the external and internal
  subset; internal subset has higher precedence
  - can override entity and attribute declarations (see next)

Markup Declarations

- **DTD** consists of markup declarations
  - element type declarations
    - similar to productions of ECFGs
  - **attribute-list declarations**
    - for declared element types
  - **entity declarations** (see later)
  - **notation declarations**
    - to pass information about external (binary) objects
      to the application

Element type declarations

- The general form is
  ```xml
  <!ELEMENT elementTypeName (E)> where E is a content model
  ```
  - regular expression of element names
  - Content model operators:
    - \( E | F \) : alternation
    - \( E, F \) : concatenation
    - \( E^? \) : optional
    - \( E^* \) : zero or more
    - \( E^+ \) : one or more
    - \( E^* \) : grouping

Attribute-List Declarations

- Can declare attributes for elements:
  - Name, data type and possible default value
- Example:
  ```xml
  <!ATTLIST FIG id ID #IMPLIED
  descr CDATA #REQUIRED
  class (a | b | c) "a">
  ```
  - Semantics mainly up to the application
  - processor checks that ID attributes are unique and that
    targets of IDREF attributes exist

Mixed, Empty and Arbitrary Content

- **Mixed content**: \( <!ELEMENT P (#PCDATA | I | IMG)*> 
  - may contain text (#PCDATA) and elements
- **Empty content**: \( <!ELEMENT IMG EMPTY> 
- **Arbitrary content**: \( <!ELEMENT HTML ANY> 
  (= <!ELEMENT HTML (#PCDATA | choice-of-all-declared-element-types) >) 
  ```
Entities (1)

- Multiple uses:
  - character entities:
    » &lt; &gt; and &amp; expand to '<' and '
  - other predefined entities:
    &amp;amp; &lt; &apos; &amp;quot;
  - general entities are shorthand notations:
    &lt;!ENTITY UKU "University of Kuopio"&gt;

Entities (2)

- physical storage units comprising a document
  - parsed entities
    &lt;!ENTITY chap1 SYSTEM "http://myweb/ch1"&gt;
  - document entity is the starting point of processing
  - entities and elements must nest properly:
    &lt;!DOCTYPE doc [
      &lt;!ENTITY chap1 ...
    ]&gt;

Unparsed Entities

- External (binary) files
- Declarations:
  &lt;!NOTATION TIFF "bin/xv"&gt;
  &lt;!ENTITY fig123 SYSTEM "figs/f123.tif" NDATA TIFF&gt;
- Usage:
  &lt;IMG file="fig123"&gt;

Parameter entities

- Way to parameterize and modularize DTDs
  &lt;!ENTITY % table-dtd SYSTEM "dtds/tab.dtd"><%%table-dtd...&gt;
  &lt;!ENTITY % stAttr "status (draft | ready)
  "draft"; &lt;!ENTITY chap %stAttr; &lt;!ENTITY sect %stAttr; &lt;
  &lt;!ATTLIST IMG file ENTITY #REQUIRED; &lt;

Speculations about XML Parsing

- Parsing involves two things:
  1. Checking the syntactic correctness of the input
  2. Building a parse tree for the input (a la DOM), or otherwise passing the document content to the application (e.g. a la SAX)
- Task 2 is simple, thanks to the simplicity of XML markup (see next)
- Slightly more difficult (?) to implement are
  - pulling the entities together
  - checking the well-formedness
  - checking the validity w.r.t the DTD (or a Schema)

Building an XML Parse Tree

<S><W>Hello</W><E><E><W>world!</W></E></W></S>