Московский государственный университет имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра алгоритмических языков

Зверев Дмитрий Андреевич

Конвертер XML в JSON на Haskell

Практическое задание

Преподаватели:
Шамаева Елена Денисовна
Шитик Артём Иванович

Москва, 2025

Содержание

1. Постановка задачи

Цель работы

Конвертировать заданный текстовый файл из формата XML в формат JSON. Имя исходного файла и имя результирующего файла задаются как аргументы командной строки. В программе необходимо определить внутреннее представление для описания основных объектов выбранных форматов.

Основные задачи

- Определение внутренних типов данных для XML и JSON;
- Парсинг XML-файла с проверкой корректности синтаксиса (включая обработку атрибутов, дочерних элементов, текстовой информации, комментариев, деклараций);
- Преобразование XML-структуры во внутреннее представление на Haskell, соответствующее формату JSON;
- Сериализация преобразованной структуры в формате JSON;
- Обработка ошибочных ситуаций (например, незакрытые теги).

Входные данные

Имя входного файла (XML) и имя выходного файла (JSON) — передаются через аргументы командной строки.

Выходные данные

Сгенерированный JSON-файл или сообщение об ошибке.

2. Теоретическая часть

XML (eXtensible Markup Language) – расширяемый язык разметки, рекомендованный W3C для описания иерархических структур данных. Он основан на SGML и предназначен для хранения и передачи данных в текстовом формате, который может быть читаем как человеком, так и машиной.

JSON (JavaScript Object Notation) – облегченный формат обмена данными, основанный на синтаксисе JavaScript, но независимый от конкретного языка. Он представляет данные в виде объектов (пар «ключ–значение») и массивов, поддерживает базовые типы (строки, числа, булевы значения и `null`) и требует экранирования специальных символов в строках.

При преобразовании XML в JSON возникают следующие особенности:

- **Смешанное содержимое**

  XML-элемент может содержать одновременно текст и дочерние элементы, тогда как в JSON последовательность текстовых и структурных узлов напрямую не сохраняется, и для решения этой проблемы вводится специальное поле `#text` для хранения текста, а дочерние элементы группируются в объекты или массивы.

- **Атрибуты**
  В XML атрибуты и дочерние элементы являются разными сущностями. В JSON все данные представлены в виде полей объекта, поэтому атрибуты преобразуются в поля с символом `@` для избежания конфликтов имен.
- **Повторяющиеся элементы**
  Повторяющиеся одноимённые теги XML конвертируются в JSON-массив.
- **Экранирование строк**
  JSON-спецификация требует экранировать специальные символы внутри строк (", \, управляющие символы и т.д.) для обеспечения корректности.

3. Описание программы

Программа генерирует JSON-файл по следующим правилам:

1. Атрибуты XML конвертируются в поля JSON с символом `@`:

```
<tag attr="value"> =>
=> {"@attr": "value"}
```

2. Текстовые узлы после открывающего тега с атрибутом сохраняются как строки с ключом `#text`:

```
<tag a="1">hello</tag> =>
=> {"tag": {"@a":"1", "#text":"hello"}}
```

3. Одиночные элементы преобразуются в объекты:

```
<tag><child>data</child></tag> =>
=> {"tag": {"child":"data"}}
```

4. Повторяющиеся элементы группируются в массивы:

```
<list><item>1</item><item>2</item></list> =>
{"list": {"item": ["1", "2"]}}
```

5. Корректно обрабатывается смешанное содержимое (текст + элементы) с использованием ключа `#text`:

```
<a>Hello_ <b>WoW</b> world!</a> =>
=> {"a": {"#text":"Hello_world!", "b":"WoW"}}
```

6. Экранирование спецсимволов в строках, например:

```
" => \"
```

3.1. Модуль XMLTypes.hs

Данный модуль задает единый тип `XMLNode`, который позволит хранить исходную структуру и содержимое XML-файла (теги, атрибуты, текст, вложенность). Именно на этом типе строится парсинг (модуль `XmlParser.hs`) и конвертация в JSON (модуль `JsonConverter.hs`).

Ниже приведен **полный код XMLTypes.hs**:

```
module XMLTypes (XMLNode(..)) where

--  Узел XML: текстовый элемент или элемент с атрибутами/дочерними узлами
data XMLNode
  = TextNode String          -- текстовое содержимое
  | ElementNode
      String                 -- имя тега
      [(String, String)]     -- атрибуты
      [XMLNode]              -- дочерние узлы
  deriving (Show, Eq)
```

Описание:

- `TextNode String` хранит обычный текст между тегами.
- `ElementNode name attrs children`:
  - `name` – имя тега;
  - `attrs :: [(String,String)]` – список атрибутов вида (`"id"`, `"1"`);
  - `children :: [XMLNode]` – вложенные узлы (текстовые/элементные).
- `deriving (Show, Eq)` позволяет выводить узлы в консоль и сравнивать их.

3.2. Модуль JSONTypes.hs

Данный модуль определяет алгебраические типы данных для представления структуры JSON. Предоставляемые в этом модуле типы позволяют структурировать данные, полученные после конвертации из XML, и обеспечить сериализацию в строковое представление JSON. Модуль используется другими модулями программы, а именно `JsonConverter.hs` и `JsonSerializer.hs`.

Ниже приведен **полный код JSONTypes.hs**:

```
module JSONTypes (JSONValue(..), JSONObject(..)) where

--  JSON-значение: строка, объект, массив
data JSONValue
  = JSONString String
  | JSONObj    JSONObject
  | JSONArray  [JSONValue]
  deriving (Show, Eq)

--  JSON-объект: список пар вида ключ-значение
newtype JSONObject = JSONObject [(String, JSONValue)]
```

```
  deriving (Show, Eq)
```

Описание:

- `JSONValue` — тип, описывающий возможные значения в JSON:
  - `JSONString String` — строка,
  - `JSONObj JSONObject` — вложенный объект,
  - `JSONArray [JSONValue]` — массив элементов типа `JSONValue`.
- `JSONObject` — новый тип специально для списка пар `(ключ, значение)`.
- `deriving (Show, Eq)` позволяет печатать и сравнивать JSON-структуры.

3.3. Модуль XmlParser.hs

Данный модуль отвечает за парсинг XML-текста в древовидную структуру `XMLNode`.

Ниже приведен **полный код XmlParser.hs**:

```
module XmlParser (parseXml) where

import XMLTypes
import Data.Char (isSpace, isAlphaNum)
import Data.List (isPrefixOf, dropWhileEnd, isInfixOf)

-- Удаление всех комментариев вида <!-- ... -->
skipComments :: String -> String
skipComments [] = []
skipComments ('<':'!':'-':'-':cs) =
  case breakOn "-->" cs of
    (_, '-':'-':'>':xs) -> skipComments xs
    _                   -> []
skipComments (c:cs) = c : skipComments cs -- обработка обычных символов


-- Разбиение строки по первому вхождению подстроки через механизм аккумулятора
breakOn :: String -> String -> (String, String)
breakOn pat = go []
  where
    go acc [] = (reverse acc, [])
    go acc xs@(y:ys)
      | pat `isPrefixOf` xs = (reverse acc, xs)
      | otherwise           = go (y:acc) ys

-- Убирает начальные пробельные символы
skipSpaces :: String -> String
skipSpaces = dropWhile isSpace

-- ТОЧКА ВХОДА: парсит весь XML-документ:
-- возвращает Left с текстом ошибки или Right корневой узел
```

```
parseXml :: String -> Either String XMLNode
parseXml input =
    -- проверка на незакрытый комментарий
    if "<!--" `isInfixOf` input && not ("-->" `isInfixOf` input)
      then Left "Незакрытый комментарий"
      else
        -- удаление комментариев
        let cleaned = skipComments input
            input'  = skipSpaces cleaned
        in case input' of
             -- пропускаем XML-декларацию
             ('<':'?':'x':'m':'l':rest) ->
               case span (/= '>') rest of
                 (_, '>':rest') ->
                    let afterParseXml = skipSpaces rest'
                    in case afterParseXml of
                        [] -> Left "Остаток после парсинга XML!"
                        _ -> case parseElement afterParseXml of
                                Right (node, rem') | null (skipSpaces rem') ->
                                        Right node
                                Right _ -> Left "Остаток после парсинга XML"
                                Left err -> Left err
                 _              -> Left "Некорректная декларация XML"
             -- иначе сразу элемент
             _ -> case parseElement input' of
                    Right (node, rem')
                      | null (skipSpaces rem') -> Right node
                      | otherwise              -> Left "Остаток после парсинга XML"
                    Left err -> Left err

-- Парсинг одного элемента
parseElement :: String -> Either String (XMLNode, String)
parseElement ('<':'/':_) =
    Left "Неожиданный закрывающий тег"
parseElement ('<':rest0) =      -- парсим имя тега, атрибуты
    case span isAlphaNum rest0 of
      (name, rest1) ->
        let (attrs, rest2)  = parseAttributes (skipSpaces rest1)
            restClean       = skipSpaces rest2
        in case restClean of

             -- самозакрывающийся тег <tag/>
             '/':'>':rem' ->
               Right (ElementNode name attrs [], rem')

             -- битый атрибут
             c:_
               | isAlphaNum c -> Left "Некорректный синтаксис атрибута"

             -- открывающий '>' и далее содержимое
```

```
            '>':content ->
              case parseChildren content of
                Right (children, after) ->
                  let rest4 = skipSpaces after
                  in case rest4 of
                       ('<':'/':closing) ->
                         case span isAlphaNum closing of
                           (cn, '>':final)
                             | cn == name ->
                                   Right (ElementNode name attrs children, final)
                             | otherwise  -> Left $
                                 "Ожидался </" ++ name ++ ">, получен </" ++ cn ++ ">"
                           _ -> Left "Некорректный закрывающий тег"
                       _ ->
                         Left $
                           "Ожидался закрывающий тег </" ++ name ++ ">, но получено: "
                           ++ if null rest4 then "<пусто>" else take 10 rest4
                Left err -> Left err

            -- всё остальное: ожидался '>' или '/>'
            _ -> Left "Ожидался '>' после тега"
parseElement _ =
    Left "Ожидался открывающий тег '<'"


-- Считывает все атрибуты key="value" до '>' или '/>'
parseAttributes :: String -> ([(String, String)], String)
parseAttributes input =
    case input of
      ('>':_)     -> ([], input)
      ('/':'>':_) -> ([], input)
      _ -> case parseAttribute input of
             Just (attr, rem') ->
               let (attrs, rem'') = parseAttributes (skipSpaces rem')
               in (attr : attrs, rem'')
             Nothing -> ([], input)

-- Парсит один атрибут (key="value")
parseAttribute :: String -> Maybe ((String, String), String)
parseAttribute input =
    case span isAlphaNum input of
      (key, '=':'"':rest) ->
        case span (/= '"') rest of
          (val, '"':rem') -> Just ((key, val), rem')
          _               -> Nothing
      _ -> Nothing

-- Рекурсивно парсит дочерние узлы: текстовые и/или элементные
parseChildren :: String -> Either String ([XMLNode], String)
parseChildren input =
```

```
let input' = skipSpaces input
in if null input'
     then Right ([], "")
     else case input' of

            -- закрывающий тег: конец списка детей
            ('<':'/':_) -> Right ([], input')

            -- вложенный элемент
            ('<':_) ->
              case parseElement input' of
                Right (node, rem') ->
                  case parseChildren rem' of
                    Right (nodes, rem'') -> Right (node : nodes, rem'')
                    Left err             -> Left err
                Left err -> Left err

            -- текстовый узел
            _ ->
              let (raw, rem') = span (/= '<') input'
                  trimmed     = dropWhileEnd isSpace (dropWhile isSpace raw)
              in if null trimmed
                    then parseChildren rem'
                    else case parseChildren rem' of
                           Right (nodes, rem'') ->
                                Right (TextNode trimmed : nodes, rem'')
                           Left err             -> Left err
```

Ключевые моменты:

- `skipComments` в `parseXml` удаляет все XML-комментарии перед парсингом;
- `parseXml` обрабатывает опциональную декларацию `<?xml ...?>`, затем вызывает `parseElement`.
- `parseElement` распознает:
  - самозакрывающиеся теги `<tag/>`,
  - открывающие теги с атрибутами,
  - содержимое и соответствующий закрывающий тег.

  Пример:

  ```
  parseElement "<tag id=\"1\">text</tag>rest" →
  → Right (ElementNode "tag" [("id", "1")] [TextNode "text"], "rest")
  ```

- `parseAttributes`, используя `parseAttribute`, поочерёдно извлекает пары `key="value"`.
- `parseChildren` парсит дочерние узлы элемента (текст или вложенные элементы): если встречает $</$, возвращает пустой список дочерних узлов. Если начало с $<$, парсит элемент с помощью `parseElement`; иначе парсит текст до $<$, создавая `TextNode` и рекурсивно обрабатывает остаток строки.

Все функции возвращают остаток недоразобранной строки, что позволяет контролировать корректность: в конце парсинга оставаться ничего не должно. В случае ошибок парсинга функции завершают выполнение с ошибкой через Left.

3.4. Модуль JsonConverter.hs

Модуль `JsonConverter.hs` преобразует уже распарсенный `XMLNode` в дерево `JSONValue`, при этом сохраняя логику атрибутов, текстов, повторяющихся элементов и смешанного содержимого.

Ниже приведен **полный код JsonConverter.hs**:

```
module JsonConverter (xmlToJsonRoot) where

import XMLTypes
import JSONTypes
import Data.List (groupBy, sortOn)

-- Рекурсивное преобразование узла XML в JSONValue
xmlToJson :: XMLNode -> JSONValue
xmlToJson (TextNode s) = JSONString s
xmlToJson (ElementNode _ attrs children) =
  let
    -- разделяем: всю текстовую часть и группы дочерних элементов
    -- groups есть список пар (имя_тега, [XMLNode]) для каждого вида элемента
    (text, groups) = separateChildren children

    -- каждый XML-атрибут key="value" превращается в JSON-поле "@key": "value"
    attrPairs  = [("@" ++ k, JSONString v) | (k, v) <- attrs]

    -- если есть текст, складываем его с ключом "#text"
    textPair   = if null text then [] else [("#text", JSONString text)]

    -- генерация списка, которая превращает каждый тег из groups в JSON-пару
    elemPairs  =
      [ (name, toJsonValue nodes)
      | (name, nodes) <- groups
      ]

    allPairs   = attrPairs ++ textPair ++ elemPairs -- объединение
  in
    case (attrs, groups, text) of
      -- если только текст без атрибутов и детей, возвращаем строку
      ([], [], t) | not (null t) -> JSONString t
      _                          -> JSONObj (JSONObject allPairs)

-- Основная Функция: для корневого элемента сохраняем имя тега
xmlToJsonRoot :: XMLNode -> JSONValue
xmlToJsonRoot node@(ElementNode name _ _) =
  JSONObj (JSONObject [(name, xmlToJson node)])
xmlToJsonRoot (TextNode s) = JSONString s
```

```
-- Преобразует список XML-узлов в JSON-значение
-- (при обработке групп элементов с одинаковыми именами)
toJsonValue :: [XMLNode] -> JSONValue
toJsonValue [x] = xmlToJson x    -- если один элемент
toJsonValue xs  = JSONArray (map xmlToJson xs) -- если несколько

--  Разделяет дочерние узлы на текст и группы элементов по имени тега
separateChildren :: [XMLNode] -> (String, [(String, [XMLNode])])
separateChildren children =
  let
    -- собираем весь текст
    textElems  = [s | TextNode s <- children]
    text       = concat textElems

    -- оставляем только ElementNode
    elems      = [n | n@(ElementNode _ _ _) <- children]

    -- группируем по имени тега сохраняя исходный порядок элементов
    -- сортировка нужна ибо groupBy работает только с последовательными элементами
    groups     = groupBy sameName (sortOn getName elems) where
        sameName x y = getName x == getName y

    -- превращаем каждую непустую группу в пару (имя, список узлов с этим именем).
    keyed      = [ (getName first, grp) | grp@(first:_) <- groups ]
  in
    (text, keyed)

-- Извлечение имени из ElementNode
getName :: XMLNode -> String
getName (ElementNode name _ _) = name
getName _ = error "Не удалось извлечь имя из ElementNode"
```

Описание:

- `xmlToJsonRoot` оборачивает результат `xmlToJson` в объект с единственным ключом-именем корневого тега;
- `xmlToJson` рекурсивно обходит XML-узел:
  - `TextNode` становится `JSONString`.
  - `ElementNode` собирает атрибуты, текст и дочерние элементы в единый список полей `[(String, JSONValue)]`.

  Пример:

  ```
  xmlToJson (ElementNode "tag" [("id", "1")] [TextNode "text"]) →
  → JSONObj (JSONObject [("@id", JSONString "1"), ("#text", JSONString
  "text")]).
  ```

- `toJsonValue` вызывает xmlToJson, если список содержит один узел, иначе создаёт JSONArray;

- `separateChildren` собирает весь текст дочерних узлов и группирует вложенные элементы по имени, чтобы потом корректно отобразить их в JSON;
- `getName` используется для сортировки и группировки элементов по имени.

3.5. Модуль JsonSerializer.hs

Этот модуль отвечает за преобразование внутреннего представления JSON в корректную текстовую строку.

Ниже приводится **полный код JsonSerializer.hs**:

```
module JsonSerializer (jsonValueToString) where

import JSONTypes
import Data.List (intercalate) -- для объединения списка строк с разделителем

-- Основная Функция: преобразует JSONValue в его строковое представление
jsonValueToString :: JSONValue -> String
jsonValueToString (JSONString s) =
  "\"" ++ escapeString s ++ "\""
jsonValueToString (JSONArray xs) =
  "[" ++ intercalate ", " (map jsonValueToString xs) ++ "]"  -- для случая массива
jsonValueToString (JSONObj (JSONObject pairs)) =    -- pairs :: [(String, JSONValue)]
  "{"                                                       -- для обекта JSONObj
  ++ intercalate ", "
       [ "\"" ++ k ++ "\":" ++ jsonValueToString v
       | (k, v) <- pairs
       ]
  ++ "}"

-- Экранизация спецсимволов в строке
escapeString :: String -> String
escapeString = concatMap escapeChar

escapeChar :: Char -> String        -- проход отдельно по символам
escapeChar c
  | c == '"'  = "\\\""  --   "   ->    \"
  | c == '\\' = "\\\\"  --   \   ->    \\
  | c == '\b' = "\\b"   -- Backspace
  | c == '\f' = "\\f"   -- Form feed
  | c == '\n' = "\\n"   -- новая строка
  | c == '\r' = "\\r"   -- возврат каретки
  | c == '\t' = "\\t"   -- табулция
  | c < ' '   = "\\u00" ++ toHex (fromEnum c) -- fromEnum для получения кода символа
  | otherwise = [c] -- иначе возвращает строку из одного символа

-- Перевод числа в 16-ю СС
toHex :: Int -> String
toHex n =
  let hex = "0123456789ABCDEF"
```

```
  in [ hex !! (n `div` 16)  -- !! - штука для доступа к эл-ту hex с индексом (n `div` 16
     , hex !! (n `mod` 16)
     ]
```

Описание:

- `jsonValueToString` обходит три случая: строку, массив и объект.

  Пример:

  ```
  jsonValueToString (JSONObj (JSONObject [("name", JSONString
  "Alice")])) → {"name": "Alice"}
  ```

- `escapeString` и `escapeChar` обеспечивают экранирование всех специальных символов в строках в соответствии со спецификацией JSON (включая управляющие символы с кодами ниже 0x20).

3.6. Модуль Main.hs

Основной модуль программы: отвечает за взаимодействие с пользователем.

Ниже приведен **полный код Main.hs**:

```
module Main (main) where

import System.Environment (getArgs)
import XmlParser       (parseXml)
import JsonConverter   (xmlToJsonRoot)
import JsonSerializer  (jsonValueToString)

main :: IO ()
main = do
  args <- getArgs -- считывает аргументы командной строки
  case args of
    [inputFile, outputFile] -> do  -- проверка на корректность аргументов
      content <- readFile inputFile -- считывает содержимое XML-файла в строку
      case parseXml content of
        Left err      -> putStrLn $ "Ошибка парсинга: " ++ err
        Right xmlTree ->
          writeFile outputFile (jsonValueToString (xmlToJsonRoot xmlTree))
    _ ->
      putStrLn "Использование: data-project-exe <input.xml> <output.json>"
```

Таким образом, данный код:

- считывает аргументы командной строки (имена входного XML- и выходного JSON-файлов),
- запускает парсер XML (`parseXml`),
- при успешном разборе происходит конвертация в JSON (`xmlToJsonRoot`) и сериализация в строку (`jsonValueToString`),

- записывает результат в файл или выводит сообщение об ошибке.

4. Используемые библиотеки

В данном проекте были использованы следующие стандартные и вспомогательные библиотеки Haskell:

- `base`: основная библиотека языка, автоматически подключается в каждом модуле;
- `System.Environment` (`base`)
  - `getArgs` — чтение аргументов командной строки в `Main.hs`;
- `Data.Char` (`base`)
  - `isSpace`, `isAlphaNum` — проверка и пропуск пробельных и буквенно-цифровых символов (при парсинге XML),
  - `dropWhile`, `dropWhileEnd` — обрезка пробелов;
- `Data.List` (`base`)
  - `isPrefixOf`, `isInfixOf` — поиск и проверка наличия подстрок (комментарии, декларации),
  - `breakOn` (собственная реализация) + `span` — разбиение строк по шаблонам,
  - `sortOn`, `groupBy` — группировка одноимённых XML-элементов для формирования JSON-массивов,
  - `intercalate` — объединение строк при сериализации JSON.

5. Пример работы программы

5.1. Основной интерфейс

При запуске программа ожидает 2 файла-параметра:

```
data-project-exe <input.xml> <output.json>
```

В противном случае выводится следующее справочное сообщение:

```
Использование: data-project-exe <input.xml> <output.json>
```

5.2. Работа программы

1. Чтение содержимого входного XML-файла:

   ```
   content <- readFile inputFile
   ```

2. Парсинг XML-документа:

   ```
   case parseXml content of
     Left err      -> putStrLn $ "Ошибка парсинга: " ++ err
     Right xmlTree -> ...
   ```

3. Конвертация во внутреннее представление JSON:

```
let jsonVal = xmlToJsonRoot xmlTree
```

4. Сериализация и запись в выходной файл:

```
writeFile outputFile (jsonValueToString jsonVal)
```

Пример работы:

Файл *"input.xml"*:

```
<!-- COMMENT -->
<movie title="Titanic" director="James Cameron">
    <scene number="1">
        <title>Departure</title>
        <timestamp unit="min">5</timestamp>
        <timestamp>6</timestamp>
        <timestamp>7</timestamp>
    </scene>
    <scene number="2">
        <title>Iceberg Ahead</title>
        <timestamp>23:39</timestamp>
        <timestamp format="sec">3600</timestamp>
        <timestamp>23:40</timestamp>
    </scene>
</movie>
```

Соответствующий файл *"output.json"*:

```
{
  "movie": {
    "@title": "Titanic",
    "@director": "James Cameron",
    "scene": [
      {
        "@number": "1",
        "timestamp": [
          {
            "@unit": "min",
            "#text": "5"
          },
          "6",
          "7"
        ],
        "title": "Departure"
      },
      {
        "@number": "2",
        "timestamp": [
          "23:39",
```

```
          {
            "@format": "sec",
            "#text": "3600"
          },
          "23:40"
        ],
        "title": "Iceberg Ahead"
      }
    ]
  }
}
```

5.3. Обработка ошибочных ситуаций

- **Неправильное число аргументов:**

```
> data-project-exe
Использование: data-project-exe <input.xml> <output.json>
```

- **Нет закрывающих тегов:**

```
<root></notroot>
Ошибка парсинга: Ожидался </root>, но получен </notroot>

<root>
<child>Text
</root>
Ошибка парсинга: Ожидался </child>, но получен </root>
```

- **Нет единого корневого тега $=>$ программа выявляет остаток после парсинга:**

```
<root>Ok</root>
<another>Wrong</another>
Ошибка парсинга: Остаток после парсинга XML
```

- **Некорректный атрибут:**

```
<tag attr "missing_eq"></tag>
Ошибка парсинга: Некорректный синтаксис атрибута
```

- **Некорректный синтаксис XML:**

```
Just some text
Ошибка парсинга: Ожидался открывающий тег '<'
```

- **Нет открывающего тега:**

```
</orphan>
Ошибка парсинга: Неожиданный закрывающий тег
```

- **Незакрытый комментарий:**

```
<root>
<!-- this is broken
<child>text</child>
</root>
Ошибка парсинга: Незакрытый комментарий
```

6. Выводы

Полученный конвертер реализует полное и корректное преобразование файла из формата XML в JSON: модули `XMLTypes` и `JSONTypes` задают внутренние структуры, `XmlParser` парсит любой корректный XML и выдает описание ошибок при нарушении синтаксиса, `JsonConverter` обеспечивает отображение атрибутов, текста и вложенных элементов, а `JsonSerializer` генерирует валидный JSON с корректным экранированием. Таким образом, цель задания была достигнута.

7. Литература

[1] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0. Fifth Edition* [Электронный ресурс] // W3C Recommendation. – 10 February 1998. – Режим доступа: https://www.w3.org/TR/xml (дата обращения: 01.04.2025).

[2] Wikipedia. *XML* [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/XML (дата обращения: 06.04.2025).

[3] Wikipedia. *JSON* [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/JSON (дата обращения: 01.04.2025).

[4] Bray T., ed.; et al. *RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format* [Электронный ресурс] – IETF RFC 8259. – December 2017. – Режим доступа: https://datatracker.ietf.org/doc/html/rfc8259 (дата обращения: 10.04.2025).

[5] ENSI Tech. *Интеграции: JSON и XML/XSD* [Электронный ресурс]. – Режим доступа: https://docs.ensi.tech/analyst-guides/tools/json (дата обращения: 25.03.2025).

[6] Oxygenxml. *XML to JSON Converter* [Электронный ресурс]. – Режим доступа: https://www.oxygenxml.com/doc/ug-editor/topics/convert-XML-to-JSON-x-tools.html (дата обращения: 10.04.2025).

[7] Wullink E.; et al. *EPP XML to RPP JSON Conversion Rules (draft-wullink-rpp-json-00)* [Электронный ресурс] – IETF Internet-Draft. – Режим доступа: https://www.ietf.org/archive/id/draft-wullink-rpp-json-00.html (дата обращения: 20.04.2025).