DECO3801 Test Plan Document

THEM - Typed HTML5 Evaluation Machine
Carl Hattenfels, Scott Heiner, Shen Yong Lau, Robert Meyer, Brendan Miller, David Uebergang

Functional Test Plan

Testing Strategy

There are three major testable components of the Typed HTML5 Evaluation Machine, our web application: the front-end website, back-end parser and database. While it was easy to write Python test cases for the back-end parser, it was more difficult to test our front-end website and database with a suite of computer-run tests. Instead, we wrote up a series of scenarios that we would undertake to ensure that the web application was running correctly and as expected. Clearly, all of these scenario tests can be "implemented" as they are merely actions performed by us. This means that a test fails when some functionality is not yet implemented, or when fixing one error creates another error.

In terms of the parser, - talk about error tests being added periodically - add code for error tests that are currently implemented into appendix.

Test Case Transcript

Implications of Functional Testing

Our functional testing highlighted some issues with all aspects of our application.

User Experience Goals

We had a clear user experience in mind while developing this website. Through its ease of use and minimal effort on the part of the user, we have aimed to create a very surgical, ambient, passive experience. The tool should give users immediate insight into the issues with their HTML and websites. This is where the user's experience with our tool ends, for this session. The user now can go and fix their file externally, return to our program and almost instantly receive another assessment of their code's validity. We do not aim to get the user invested in our system. However, we wish to create a reliable and worthwhile experience, brief as it is. The user should not be frustrated by the errors the program reveals, with the focus on helping the user learn and develop better web practices. It is meant to be a program that a user just “touches”, that is, they upload their file they want to check, and then go back and fix it, and then come back to this to validate again, in a cyclic process.

Our priorities are on quick and easy use, which is why everything is instantly accessible and requires very few clicks to navigate. We have designed the website to require as few as five clicks to access the primary functionality of the system.

User Testing Plan

Our web application will eventually be utilised by two user groups - students of DECO1400, and students of DECO7140. As such, we determined four major user testing groups:

- Undergraduate students who have already completed DECO1400
- Undergraduate students who have not completed DECO1400 but have worked with computers
- Masters students who have already completed DECO7140
- Masters students who have not already completed DECO7140

However, poor initial consideration due to this highly targeted user base caused us to primarily focus on students who hadn't done DECO1400, as these students represented students "new" to DECO1400. Focusing also on past DECO1400 students would have allowed us to get a feel for people who had previously done the course, and could determine whether the tool would have been worthwhile to them. Poor communication on our part lead to us getting very few in this category. Ultimately, we got information from ten users - five were undergraduates who had not done DECO1400, one was an undergradute who had done DECO1400 and four were masters students who had done DECO7140.

- in general, users had no trouble navigating the system - the result of the validation (i.e. highlighted tags) was not well understood by users

Summarise the results of your tests. For each scenario-based test: tabulate your metric results against each task describe or present your users feedback during/after the test Close with a general discussion that: summarises the issues raised identifies areas for improvement and design suggestions outlines any redefinition of functional and user test plans for final prototype (We dont care whether your prototype passes the tests. What is important is that your prototype is sufficiently broad to validate your test plan for the final product.)

Appendix - Python Test Code

Syntax Tests

```python
from __future__ import absolute_import, division, unicode_literals

#from . import support
import unittest, html5lib
from html5lib import treebuilders

class TestSyntax(unittest.TestCase):
    """
    Provides a number of test cases to test the syntax used
    in the document.
    """

    def setUp(self):
        self.parser = html5lib.HTMLParser(tree=treebuilders.
            getTreeBuilder("etree"))

    def test_malformed_tag_name(self):
        """
        Test that the tag name isn't an invalid symbol.

        Input:
        A HTML fragment containing a tag with an invalid tag name.

        Expected Results:
        An error should be thrown reporting an invalid tag name.
        """

        inputFragmentEmptyName = "<html>< ></body>"
        inputFragmentQuestionMark = "<html><?></body>"
        inputFragmentRightBracket = "<html><></html>"

        self.parser.parse(inputFragmentEmptyName)

        self.assertIn(((6, 6), u'expected-tag-name', {u'data': u' '}),
            self.parser.errors, "Failed to report invalid tag name. Get "
                )

        self.parser.reset()
        self.parser.parse(inputFragmentQuestionMark)

        self.assertIn(((6, 6), u'expected-tag-name-but-got-question-mark'
            , {}),
            self.parser.errors, "Failed to report valid tag name. Got
                question mark instead.")

        self.parser.reset()
        self.parser.parse(inputFragmentRightBracket)

        self.assertIn(((6, 7), u'expected-tag-name-but-got-right-bracket'
            , {}),
            self.parser.errors, "Failed to report valid tag name. Got
                question mark instead.")

    def test_self_closing_end_tag(self):
        """
        Test that a closing tag with a misplaced forwardslash
        raises an error.

        Input:
        A HTML fragment containing a closing tag with a misplaced
            forwardslash.

        Expected Results:
        An error should be thrown reporting an invalid tag name.
        """

        inputFragment = "<html><a></a /></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((9, 14), u'self-closing-flag-on-end-tag', {}),
            self.parser.errors, "Failed to report misplaced forwardslash
                in closing tag.")

    def test_invalid_self_closing_tag(self):
        """
        Test that the use of a self closing tag for a tag
        which isn't considered a self closing tag returns
        an error.

        Input:
        A HTML fragment containing a start tag with a trailing
            forwardslash
        (self-closing) for a tag type which isn't a self closing tag.

        Expected Results:
        An error should be thrown reporting the given tag type isn't a
            self-closing
        tag.
        """

        inputFragment = "<html><a /></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((6, 10), u'non-void-element-with-trailing-solidus'
            , {u'name': u'a'}),
            self.parser.errors, "Failed to report invalid self-closing
                tag.")

    def test_attributes_in_end_tag(self):
        """
        Test that attributes occuring in a closing tag are
        reported as an error.

        Input:
        A HTML fragment containing a closing tag which contains
        at least one attribute.

        Expected Results:
        An error should be thrown reporting that the closing tag shouldn't contain
        attributes.
        """

        inputFragment = '<html><a></a src="blah"></html>'

        self.parser.parse(inputFragment)

        self.assertIn(((9, 23), u'attributes-in-end-tag', {}),
            self.parser.errors, "Failed to report attributes in closing tag.")

if __name__ == '__main__':
    unittest.main()
```

Page Structure Tests

```python
from __future__ import absolute_import, division, unicode_literals

#from . import support
import unittest, html5lib
from html5lib import treebuilders

class TestPageStructure(unittest.TestCase):
    """
    Provides a number of test cases related to basic page structure
    for html5 documents.
    """

    def setUp(self):
        self.parser = html5lib.HTMLParser(tree=treebuilders.getTreeBuilder("etree"))

    def test_singular_tags(self):
        """
        Test that the multiple-instance-singular-tag error is thrown
        for cases where more than one instance of a singular tag block is
        present.

        Input:
        Nested blocks of singular tags (html, body, head).
        eg. <html><html></html></html>

        Ouput:
        All three test cases should report a multiple instance of both
        the start and closing tags for each of the three singular tags.
        """
        multipleHTMLInstances = "<html><html></html></html>"
        multipleHeadInstances = "<html><head><head></head></head><body></body></html>"
        multipleBodyInstances = "<html><head></head><body><body></body></body></html>"

        self.parser.parse(multipleHTMLInstances)

        self.assertIn(((6, 11), u'multiple-instance-singular-tag', {u'name': u'html'}),
            self.parser.errors, "Multiple instances of starting HTML tag not reported.")

        self.assertIn(((12, 18), u'incorrect-placement-html-end-tag', {u'name': u'html'}),
            self.parser.errors, "Multiple instances of closing HTML tag not reported.")

        self.parser.reset()
        self.parser.parse(multipleHeadInstances)

        self.assertIn(((12, 17), u'multiple-instance-singular-tag', {u'name': u'head'}),
            self.parser.errors, "Multiple instances of starting HTML tag not reported.")

        self.assertIn(((25, 31), u'incorrect-placement-singular-end-tag', {u'name': u'head'}),
            self.parser.errors, "Multiple instances of closing head tag not reported.")

        self.parser.reset()
        self.parser.parse(multipleBodyInstances)

        self.assertIn(((25, 30), u'multiple-instance-singular-tag', {u'name': u'body'}),
            self.parser.errors, "Multiple instances of starting HTML tag not reported.")

        self.assertIn(((38, 44), u'unexpected-end-tag-after-body', {u'name': u'body'}),
            self.parser.errors, "Multiple instances of closing body tag not reported.")

    def test_missing_doctype(self):
        """
        Test that the expected-doctype-but-got-start-tag error is thrown
        for cases where no DOCTYPE is declared.

        Input:
        Nested blocks of singular tags (html, body, head), all of which
        are missing the DOCTYPE declaration.
        eg. <html><html></html></html>

        Expected Results:
        All test cases should report a missing DOCTYPE declaration.
        """
        startTagBeforeDoctype = "<html><html></html></html>"
        endTagBeforeDoctype = "</head></head>"
        eofBeforeDoctype = ""

        self.parser.parse(startTagBeforeDoctype)

        self.assertIn(((0, 5), u'expected-doctype-but-got-start-tag', {u'name': u'html'}),
            self.parser.errors, "Failed to report missing DOCTYPE declaration (start tag before doctype.")

        self.parser.reset()
        self.parser.parse(endTagBeforeDoctype)

        self.assertIn(((0, 6), u'expected-doctype-but-got-end-tag', {u'name': u'head'}),
            self.parser.errors, "Failed to report missing DOCTYPE declaration (closing tag before doctype.")

        self.parser.reset()
        self.parser.parse(eofBeforeDoctype)

        self.assertIn(((-1, -1), u'expected-doctype-but-got-eof', {}),
            self.parser.errors, "Failed to report missing DOCTYPE declaration (EOF before doctype.")


    def test_closing_html(self):
        """
        Test that a missing HTML closing tag is reported when none
        are present in the document.

        Input:
        Nested blocks of singular tags (head, body).

        Expected Results:
        Report whether the the closing HTML tag is present.
        """
        multipleHeadInstances = "<head><head></head></head>"
        multipleBodyInstances = "<body><body></body></body>"

        self.parser.parse(multipleHeadInstances);

        self.assertIn(((-1, -1), u’no-closing-html-tag’, {}),
            self.parser.errors, "Failed to report missing closing HTML tag.")

        self.parser.reset()
        self.parser.parse(multipleBodyInstances)

        self.assertIn(((-1, -1), u’no-closing-html-tag’, {}),
            self.parser.errors, "Failed to report missing closing HTML tag.")

    def test_misplaced_tags_before_head(self):
        """
        Test that both start and closing tags occuring before the head
        section are reported as being misplaced.

        Input:
        A number of instances of start and closing tags being placed before
        the head section.

        Expected Results:
        Report whether or not the tags preceding the head section are reported
        as being misplaced.
        """
        misplacedHeadTags = "<html><body></body><head></head></html>"
        misplacedLinkTags = "<html><a></a><head></head><body></body></html>"

        self.parser.parse(misplacedHeadTags)

        self.assertIn(((6, 11), u’incorrect-start-tag-placement-before-head’, {u’name’: u’body’}),
            self.parser.errors, "Failed to report start body tag before head section.")

        self.assertIn(((12, 18), u’incorrect-end-tag-placement-before-head’, {u’name’: u’body’}),
            self.parser.errors, "Failed to report closing body tag before head section.")

        self.parser.reset()
        self.parser.parse(misplacedLinkTags)

        self.assertIn(((6, 8), u'incorrect-start-tag-placement-before-head', {u'name': u'a'}),
            self.parser.errors, "Failed to report start link (a) tag before head section.")

        self.assertIn(((9, 12), u'incorrect-end-tag-placement-before-head', {u'name': u'a'}),
            self.parser.errors, "Failed to report closing link (a) tag before head section.")

    def test_incorrect_tags_in_head(self):
        """
        Test that tags which don't belong in the head section
        are reported as misplaced using the 'incorrect-start-tag-placement-in-head'
        and 'incorrect-end-tag-placement-in-head' errors.

        Input:
        A HTML fragment with a pair of head tags enclosing a tag
        pair which doesn't belong in the head phase.

        Expected Results:
        Inclusion of the 'incorrect-start-tag-placement-in-head'
        and 'incorrect-end-tag-placement-in-head' errors being reported
        as part of the returned array of error codes.
        """
        inputFragment = "<html><head><a></a></head></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((12, 14), u'incorrect-start-tag-placement-in-head', {u'name': u'a'}),
            self.parser.errors, "Failed to report starting tag which doesn't belong in the head section.")

        self.assertIn(((15, 18), u'incorrect-end-tag-placement-in-head', {u'name': u'a'}),
            self.parser.errors, "Failed to report closing tag which doesn't belong in the head section.")

    def test_tags_after_eof(self):
        """
        Tests that starting and closing tags occuring after the last
        instace of a closing HTML tag are reported as an error.

        Input:
        A HTML fragment with a start and closing tag pair occuring
        after the start and closing HTML pair.

        Expected Results:
```

```
        An error being thrown for both the start and closing tags
            occuring
        after the HTML tags.
        """

        inputFragment = "<html></html><a></a>"

        self.parser.parse(inputFragment)

        self.assertIn(((13, 15), u'expected-eof-but-got-start-tag', {u'
            name': u'a'}),
            self.parser.errors, "Failed to report start tag after closing
                 HTML tag.")

        self.assertIn(((16, 19), u'expected-eof-but-got-end-tag', {u'name
            ': u'a'}),
            self.parser.errors, "Failed to report closing tag after
                closing HTML tag.")

    def test_missing_start_tag(self):
        """
        Tests that a missing start tag is reported in the case
        that a closing tag is found without a matching start tag.

        Input:
        A HTML fragment containing a closing tag without a matching
        start tag.

        Expected Results:
        An error being thrown reporting that the matching start tag
        is missing.
        """

        inputFragment = "<html><head></head><body></a></body></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((25, 28), u'unexpected-end-tag', {u'name': u'a'}),
            self.parser.errors, "Failed to report the lack of a matching
                start tag.")

    def test_misplaced_tags_after_body(self):
        """
        Tests that any tags occuring after the body phase
        are reported as being incorrectly placed.

        Input:
        A HTML fragment with a pair of start and closing tags placed
        after the closing body tag.

        Expected Results:
        An error should be thrown for both the start and closing
        tags found after the closing body tag.
        """

        inputFragment = "<html><head></head><body></body><a></a></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((32, 34), u'unexpected-start-tag-after-body', {u'
            name': u'a'}),
            self.parser.errors, "Failed to report misplaced starting tag
                found after the closing body tag.")

        self.assertIn(((35, 38), u'unexpected-end-tag-after-body', {u'
            name': u'a'}),
            self.parser.errors, "Failed to report misplaced closing tag
                found after the closing body tag.")

    def test_missing_closing_html_tag(self):
        """
        Test that a missing closing HTML tag is reported.

        Input:
        A HTML fragment missing a closing HTML tag.

        Expected Results:
        An error should be thrown stating that the closing HTML tag is
            missing.
        """

        inputFragment = "<html><head></head><body></body>"

        self.parser.parse(inputFragment)

        self.assertIn(((-1, -1), u'no-closing-html-tag', {}),
            self.parser.errors, "Failed to report missing closing HTML
                tag.")

    def test_early_termination_before_head(self):
        """
        Test that an early closing HTML tag before the head phase
        is reported as an error.

        Input:
        A HTML fragment with the head and body sections placed after
        a closed set of HTML tags.

        Expected Results:
        An error should be thrown stating that the closing HTML tag
        has been found before the head phase.
        """

        inputFragment = "<html></html><head></head><body></body>"

        self.parser.parse(inputFragment)

        self.assertIn(((6, 12), u'early-termination-before-head', {u'name
            ': u'html'}),
            self.parser.errors, "Failed to report early termination
                before head section.")

    def test_early_termination_in_head(self):
        """
        Test that an early closing HTML tag in the head phase
        is reported as an error.

        Input:
        A HTML fragment with the closing HTML tag placed within
        the set of head tags.

        Expected Results:
        An error should be thrown stating that the closing HTML tag
        has been found in the head phase.
        """

        inputFragment = "<html><head></html></head><body></body>"

        self.parser.parse(inputFragment)

        self.assertIn(((12, 18), u'early-termination-in-head', {u'name':
            u'html'}),
            self.parser.errors, "Failed to report early termination
                before head section.")

    def test_early_termination_before_body(self):
        """
        Test that an early closing HTML tag before the body phase
        is reported as an error.

        Input:
        A HTML fragment with the closing HTML tag placed before the body
        section.

        Expected Results:
        An error should be thrown stating that the closing HTML tag
        has been found before the body phase.
        """

        inputFragment = "<html><head></head></html><body></body>"

        self.parser.parse(inputFragment)

        self.assertIn(((19, 25), u'early-termination-before-body', {u'
            name': u'html'}),
            self.parser.errors, "Failed to report early termination
                before head section.")

    def test_early_termination_in_body(self):
        """
        Test that an early closing HTML tag in the body phase
        is reported as an error.

        Input:
        A HTML fragment with the closing HTML tag placed within
        the set of body tags.

        Expected Results:
        An error should be thrown stating that the closing HTML tag
        has been found in the head phase.
        """

        inputFragment = "<html><head></head><body></html></body>"

        self.parser.parse(inputFragment)

        self.assertIn(((25, 31), u'early-termination-in-body', {u'name':
            u'html'}),
            self.parser.errors, "Failed to report early termination
                before head section.")

    def test_tags_between_head_body(self):
        """
        Test that a set of tags placed after the head section
        but before the body section is reported as an error.

        Input:
        A HTML fragment with a set of tags between the head
        and body sections.

        Expected Results:
        An error should be thrown stating that the set of tags
        can't be placed between the head and body sections.
        """

        inputFragment = "<html><head></head><a></a><body></body></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((19, 21), u'start-tag-before-body-after-head', {u'
            name': u'a'}),
            self.parser.errors, "Failed to report start tag after head
                phase but before body phase.")

        self.assertIn(((22, 25), u'end-tag-before-body-after-head', {u'
            name': u'a'}),
            self.parser.errors, "Failed to report closing tag after head
                phase but before body phase.")

    def test_missing_starting_html_tag(self):
        """
        Test that a missing starting HTML tag is reported as an error.

        Input:
        A HTML fragment missing a starting HTML tag.

        Expected Results:
        An error should be thrown indicating that the fragment doesn't
        contain a starting HTML tag.
        """

        inputFragment = "<head></head><body></body></html>"

        self.parser.parse(inputFragment)

        self.assertIn(((-1, -1), u'no-starting-html-tag', {}),
            self.parser.errors, "Failed to report missing starting HTML
                tag.")
```

```python

    def test_unknown_doctype(self):
        """
        Test that a doctype with an invalid name is reported as being
        an unknown doctype.

        Input:
        A HTML fragment containing an invalid doctype name.

        Expected Results:
        An error should be thrown reporting that the doctype name is invalid.
        """

        inputFragment = '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">'

        self.parser.parse(inputFragment)

        self.assertIn(((0, 89), u'unknown-doctype', {}),
            self.parser.errors, "Failed to report unknown doctype.")

    def test_space_after_doctype(self):
        """
        Test that a doctype tag has a space between the doctype declaration
        and the doctype name.

        Input:
        A HTML fragment containing a doctype with no space between the doctype
        declaration and the doctype name.

        Expected Results:
        An error should be thrown reporting that there is no space between
        the doctype declaration and the doctype name.
        """

        inputFragment = '<!DOCTYPEhtml>'

        self.parser.parse(inputFragment)

        self.assertIn(((0, 8), u'need-space-after-doctype', {}),
            self.parser.errors, "Failed to report missing space after the doctype declaration.")

if __name__ == '__main__':
    unittest.main()
```