codeproject.com

Simple CSS Parser - CodeProject

BoneSoft

19-23 minutes

- Download v2 demo project - 36.09 KB
- Download v2 source files - 241.76 KB
- Download v1 demo project - 25.68 KB
- Download v1 source files - 51.14 KB

A Quick Note on the Downloads

If you have previously seen this article, the v2 downloads contain a complete rewrite of the parser, which required a completely new model. The v1 downloads are provided for those who are dependant on the previous model (the v1 parser was last updated on 11/25/2008). If this is the first time you are seeing this article, use the v2 downloads that this article now describes.

Introduction

My motivation for this project came from an interest in

importing color information. I needed to import from CSS files into an application I was working on for managing color sets of Web and other projects. However, on import I didn't just want the color, but the elements it is associated with as well. So, Regex was out. This left me with the need for a full CSS parser that could be used for lots of things, not just my nutty little color manager. So all I need is a grammar that describes the structure of the CSS 2.1 language. Unfortunately, the only grammar definition for CSS I could find was the W3C grammar written for Lex/Yacc. Due to my inexperience with Yacc, and my apathy towards learning it, I had to write a grammar from scratch. Ultimately, I chose to write a Coco/R attributed grammar. With version 2, the parser is now based on the 2.1 spec, with support for some 3.0 elements. Because it now handles "@" rules generically, most browser specific extensions should also now be supported. The object model was changed completely for the 2nd version of this parser. Hopefully, this will cover most uses you may have for a CSS parser. If you have further requirements, it may be a good base to work from.

Parsers

I don't claim to be an expert in the fine arts of deterministic finite automaton. Having said that, the basic concept of parsing text, interpreting the contents and building some useful result from it can be understood without a deep understanding of the nitty gritty of how it is accomplished.

When working with a parser generator like Coco, it's important to understand the basic concepts.

What Parsers Do

A parser is a working piece of software that 'knows' how to recognize a set of language rules. While in the process of recognizing a language, a parser usually acts on the information recognized. A parser doesn't have to act on the information, but it wouldn't be very interesting if all it did was say "yep, that's my language all right..." Some parsers build an object graph as they parse. Some call methods on another piece of software passing them parameters from the language. And some parsers just fire events as parts are recognized. With Coco, you are free to use any of these methods through **attributed code.** The beauty of tools like Coco is that you only need to specify your language in a format they recognize (the grammar), and they produce the parser for you.

Parsers usually don't work alone. Most parsers require a scanner (sometimes called a lexer or tokenizer) to supply them with lexical chunks or tokens. When a parser starts, it asks the scanner for the first token, which it checks against its rules to determine if that token was expected. If the token does fit a rule, it uses it to determine what should be expected next. If the grammar that produced the parser says:

Hide Copy Code

```
Doc = Word [ ":" Word ] ";" .
```

The parser will report an error if the first token is not a `Word`. If it is a `Word`, then the parser will get the next token from the scanner, expecting it to be the optional `":"` or the required `";"`. If the next token is `":"` then the parser knows to expect another `Word`. But if the next token is `";"` it skips the optional `[ ":" Word ]`.

Understanding how the parser uses tokens from the scanner will help you better understand how to express your language's grammar.

Grammars

A grammar is the textual representation of a language, its pieces, its rules, and how they all work together. A grammar defines how text should be broken up into **tokens** (lexical analysis), with such details as whether or not to consider underscores as letters, or to treat multi-symbol operators as a single entity. It also defines more complex language structures called **productions** which make up the rules for your language (what may follow what, how many times, and in what order). With Coco, you may also define **attributed code** that will be embedded in the parser code at the location it is defined in the grammar. A simple example of a production might be like C# variable declarations, where you could say:

Hide Copy Code

```
variableDecl means a TypeName followed by a
Name
    followed optionally by an Initialization.
```

In EBNF, which I'll detail more below, this would look more like:

Hide Copy Code

```
varialbeDecl = TypeName Name [ "=" ( Literal |
"new" TypeName ) ] ";" .
```

In this example, `TypeName`, `Name` & `Literal` are either defined token types, or other productions. Productions are defined by specifying a name, followed by an `"="` sign, then the definition, and must end with a period. Several things can enter into your decisions on how to break the language into productions: avoiding LL(1) conflicts, recursion and or reuse (also sometimes related to avoiding LL(1) conflicts), and allowing attributed code to have access to needed elements.

Coco/R

Coco is a relatively new parser generator that has been ported to several languages, including our favorite C#. If you're not familiar with Coco, go download a copy and read the manual, it's a wonderful tool.

Coco takes ATG (attributed grammar) files containing language grammar definitions written in EBNF, and generates a Scanner (sometimes called a Lexer) and a Parser that understand your language.

ATG EBNF

EBNF is a very simple language with a small number of simple rules. However, it does take some work to express a language with it. When doing this, there are several things to consider... First of all, Coco is an LL(1) recursive decent parser, which means that it only looks ahead one token at a time, so all token declarations and all productions must start with a unique token so that Coco can determine at any given point which path to take when recognizing your language. If you define something that causes an LL(1) conflict, the compiler will tell you. Technically, Coco is an LL(1), but it does provide a couple of mechanisms for you to resolve conflicts on your own, in effect making it an LL(n) parser. I won't get into too much detail with these, because this article is not specifically about how to use Coco... But, you can do a manual multi-token look ahead in your attributed code (more on attributed code later), and you can also provide a call in your grammar to a custom method where you can analyze tokens to decide which path to take, (which you can find a few examples of in the v2 grammar).

The rules for defining productions are simple, similar in fact to regular expressions but simpler. Most texts on the subject of parsers recognize three basic non-terminal structures:

- `Sequences`: meaning X followed by Y
- `Alternatives`: meaning X or Y

- `Repetitions`: meaning zero or more Xs

  Many parser discussions will also add:

- `Options`: meaning zero or one X

  A grammar, and by extension a parser, represents a tree of rules (an abstract syntax tree to be exact). It's a maze of rules through which there are many valid paths. Every step of the way, the parser has to determine if the current token fits one of its options for moving forward. `Alternatives`, `Repetitions` and `Options` provide the conditionals that cause the tree to branch.

  In EBNF, `sequences` are represented simply by specifying one entity after another.

  Hide Copy Code

  ```
  Decl = TypeName Name .  // means a Decl is a
  TypeName followed by a Name.
  ```

  `Alternatives` are separated by pipes (`|`), sometimes grouped by parenthesis.

  Hide Copy Code

  ```
  Init = ( Literal | "new" TypeName ) .
      // means Init is a Literal or "new"
  followed by TypeName.
  ```

  `Repetitions` are enclosed within curly braces.

  Hide Copy Code

```
Sentence = { Word } "." .
    // means a sentence is any number of Words
followed by a period.
```

And `Options` are enclosed within square brackets.

Hide Copy Code

```
Decl = Type Name [ Init ] .
    // means a Decl is a Type followed by a
Name followed
    // optionally by an Init.
```

Coco's EBNF has some specific rules for the files structure, how and where to include lexical structures and token definitions, and some other things like comment characters and pragmas. I won't get into the full requirements of a Coco *ATG* file, but I would suggest downloading Coco and reading its manual if you are interested in knowing more.

Attributed Code

As stated earlier, parsers usually do something with the information they recognize. At certain points in your grammar, you will want to fill the properties of an object or objects, fire events, or call external methods. That's where attributed code comes in. Coco's grammar files are named with the *ATG* extension, which is the abbreviation for **Attributed Grammar**. Attribution code is enclosed within booby `(. .)` tags. As a very simple example of how this works...

Hide Copy Code

```
CSSDoc =
    { SelectorProd
      (. Console.WriteLine("SelectorProd
recognized"); .)
    }
.
```

Coco would produce a method that would look similar to this...

Hide Copy Code

```
void CSSDoc()
{
    while (la.kind == 5)
    {
        SelectorProd();
        Console.WriteLine("SelectorProd
recognized!");
    }
}
```

Coco turns all productions into a `void` method. So you cannot have one '`return`' value. However, you can pass them parameters with `ref` or `out` modifiers. To see an example of this, look at the *ATG* grammar file in the source package.

The Code

My entire language definition (minus attributed code) looks like

this:

Hide Shrink Copy Code

```
CSS2 =
    { ( ruleset | directive ) }
.
QuotedString =
    ( "'" {ANY} "'" | '"' {ANY} '"' )
.
URI =
    "url" [ "(" ] ( QuotedString | {ANY} ) [
")" ]
.
medium =
    (
        "all" | "aural" | "braille" |
"embossed"
        | "handheld" | "print" | "projection"
        | "screen" | "tty" | "tv"
    )
.
identity =
    (
        ident
        | "n" | "url" | "all" | "aural" |
"braille"
        | "embossed" | "handheld" | "print"
        | "projection" | "screen" | "tty" |
```

```
"tv"
    )
.
directive =
    '@' identity
    [ expr | medium ]
    (
        '{' [ {
            (
              declaration { ';' declaration }
[ ';' ]

            | ruleset
            | directive
            )
        } ] '}'
    |
        ';'
    )
.
ruleset =
    selector
    { ',' selector }
    '{' [ declaration { ';' declaration } [
';' ] ] '}'

.
```

```
selector =
    simpleselector { [ ( '+' | '>' | '~' ) ]
simpleselector }

.
simpleselector =
    ( identity | '*'
    | ('#' identity | '.' identity | attrib |
pseudo )

    )
    { ('#' identity | '.' identity | attrib |
pseudo ) }
.
attrib =

    '[' identity [
        ( '=' | "~=" | "|=" | "$=" | "^=" |
"*=" )
        ( identity | QuotedString )

    ] ']'
.
pseudo =
    ':' [ ':' ] identity [ '(' expr ')' ]

.
```

```
declaration =
    identity ':' expr [ "!important" ]
.
expr =
    term { [ ( '/' | ',' ) ] term }

.
term =
    (
      QuotedString
    | URI
    | "U\\" identity
    | HexValue
    | identity
        [ { (
            ':' [ ':' ] identity
          | '.' identity
          | '=' ( identity | { digit } )
        ) } ]

        [ '(' expr ')' ]
    |
        [ ( '-' | '+' ) ]

        { digit }
        [ '.' { digit } ]
        [ (
```

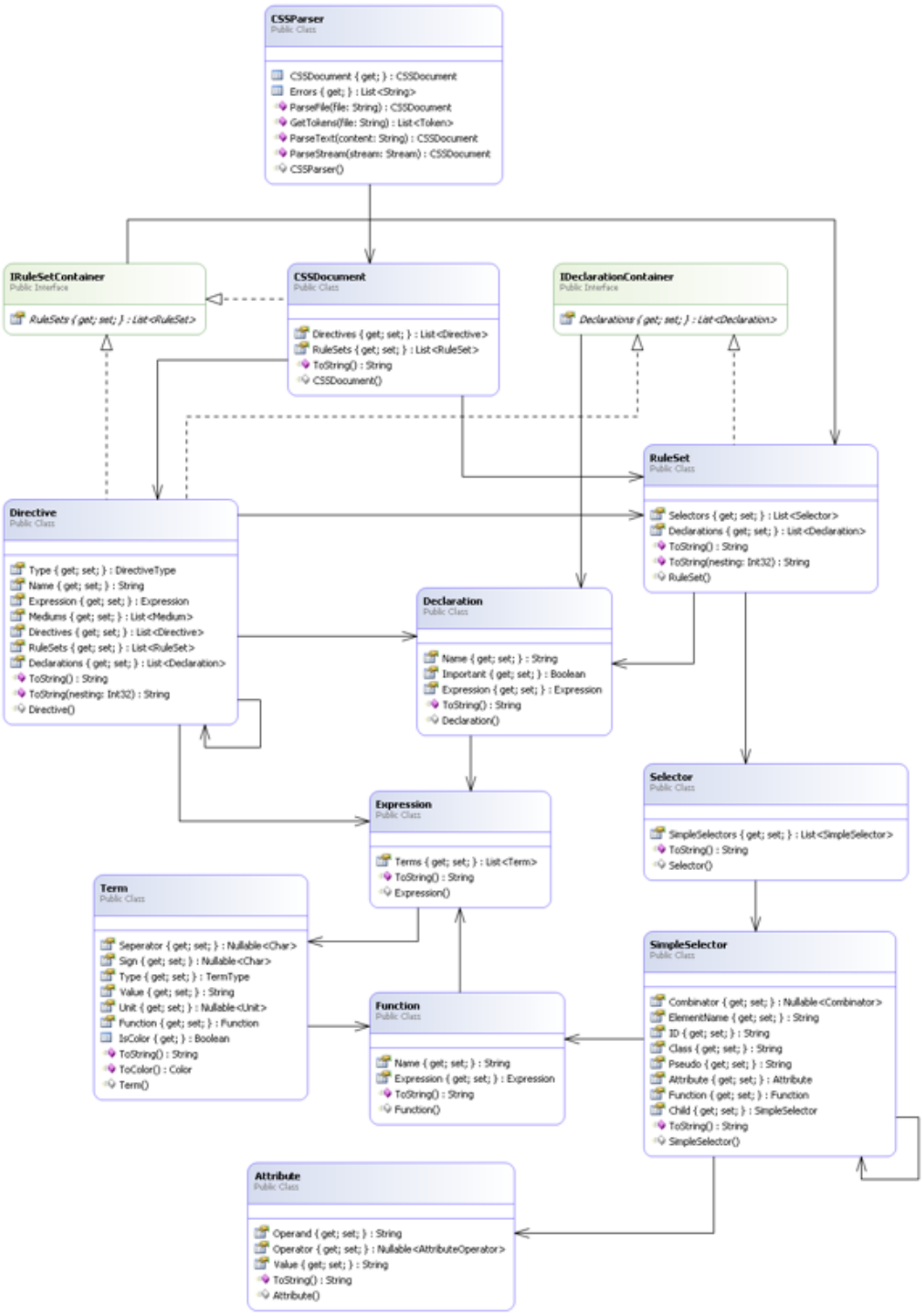
```
                "n" [ ( "+" | "-" ) digit {
digit } ]

            | "%"

            | identity
        ) ]
    )
.
HexValue =
    '#'

    [ { digit } ]
    [ ident ]
.
```

The object model that this parser builds is similar in structure to the grammar. To get the image down to the required width, the following class diagram excludes enums and some peripheral classes. I know it's difficult to read, I apologize, but you should be able to make out the details. The full diagram is included in the source download.



Describing this model is a little more difficult than the original model. It's much easier to show some examples of how the CSS structure fits into the model. A `RuleSet` is what now

represents the most common structure in CSS (selectors followed by a block of declarations):

Hide Copy Code

```
table tr td {
    color: Red;
}
```

A `RuleSet` can have one or more `Selectors`:

Hide Copy Code

```
table.one, tr#head, td {
    top: 0px;
}

span.one.two {
    color: #0066CC;
}
table tr td {
    color: Red;
}
```

A `Selector` contains one or more `SimpleSelectors` (which can also nest other `SimpleSelectors`):

Hide Copy Code

```
table, tr, td {
    top: 0px;
```

```
}

span.one.two {
    color: #0066CC;
}
table tr td {
        color: Red;
}
```

RuleSets can have zero or many Declarations representing it's block of rules.

Hide Copy Code

```
table tr td {
    border: 1px solid #FFFFFF;
}
```

A Declaration consists of a Name, and Important flag, and an Expression.

Hide Copy Code

```
table tr td {
    border: 1px solid #FFFFFF;
}
```

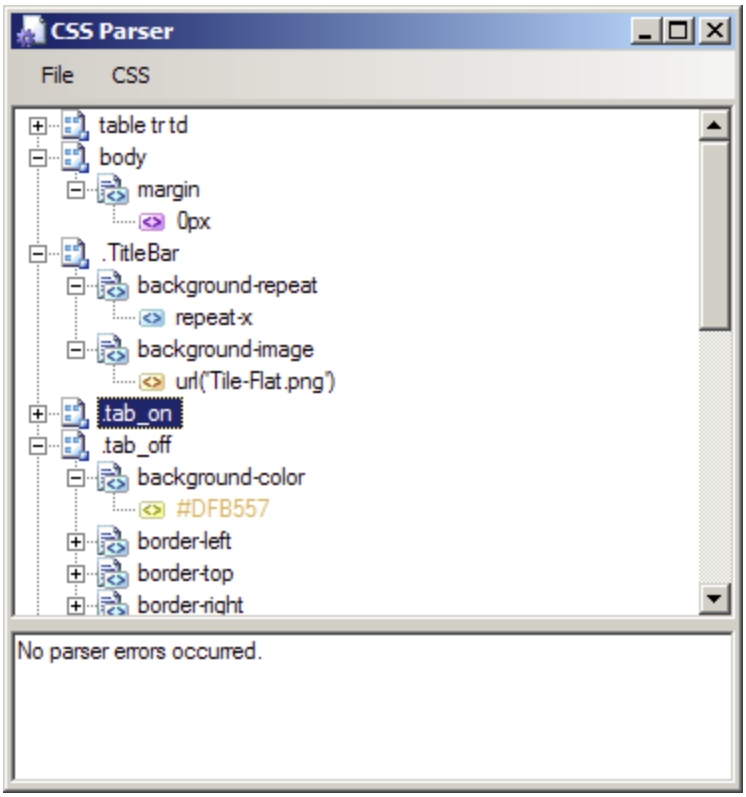
An Expression is a collection of Terms. A Term represents an actual property value.

Hide Copy Code

```
table tr td {
    border: 1px solid #FFFFFF;
}
```

The Demo Application

The demo application is very simple. Selecting "Open CSS" from the "File" menu allows you to browse for a *CSS* file. Once opened, it automatically uses the Coco generated parser to parse the *CSS* file, building a `CSSDocument` object, which it then displays in the `TreeView`. The "File" menu has a "Parse Text" option that opens a dialog window that you can enter CSS snippets to parse, and a "View Tokens" menu option that does just what it says. There is also a "CSS" menu with two options: "View CSS" and "View HTML". "View CSS" renders the model back to CSS text, which was handy for me for verifying the parser's recognition and the model's representation. "View HTML" does basically the same thing, but adds markup with `SPAN` tags so you can style it however you wish.



Conclusion

If you've been looking for a simple CSS parser, this may fill your requirements. If you are looking for a more robust CSS parser, this may give you a good start to producing one. If you are just interested in seeing a simple example of using Coco/R, this project is a light introduction with a realistic example. If nothing else, at least now you have a valid reason to use booby tags. Like I said at the beginning of this article, it is a very simple representation of CSS, but it did what I needed. I hope you can find a use for it, or can make improvements to share with us.

Ancient History

09/18/2007

Changed the scanner spec in the CSS grammar to not ignore whitespaces, to better handle selector names.

09/20/2007

At the end of the main production, I added `{whitespace}` because of an issue with comments at the end of single line selectors like...

Hide Copy Code

```
a:link { color: #DD79DD; text-decoration:
none; } /*comment*/
```

Previously in this situation, a parser error was reported halting parsing.

11/07/2007

I changed the `SelectorName` production to look for `(identifier | "*")` instead of `identifier` because of an issue with asterisks as tag identifiers...

Hide Copy Code

```
tr td * { color: #FF0000; }
```

Previously I was unaware of the wild card.

11/08/2007

I changed the `PropertyValue` production to optionally include a minus sign before number values.

12/01/2007

There was an issue with URL properties. I was using the `ANY` token, which had some unexpected side effects. I'm now using a quoted `string` token.

12/02/2007

I originally omitted the `!important` attribute. I've never used it, it was easy to overlook. But it's not included.

03/14/2008

Apparently I didn't test too thoroughly with URL properties. They should work now for most reasonable relative paths.

03/19/2008

The parser and the model have been updated to support two CSS 2.0 constructs not previously supported.

- **The > & + combinators** - `Tag` now has a `ParentRelationship` property of type `char`. By default, it's set to a `null` character. When appropriate, it will contain a > or +. In this example...

Hide Copy Code

```
E > F {}
```

... the Tag representing the F element will have a `ParentRelationship` of `>`

- **Attribute Selectors** - Tag now also has an `Attributes` property of type `List<string>`. Currently I'm not parsing the contents of attribute selectors into a meaningful structure. I may change that in the future, but for now each attribute selector is held as a `string`. For now my interest is in parsing documents that contain them, and preserving the information that's parsed. Examples of attribute selectors currently supported are as follows...

  Hide Copy Code

```
DIV P *[href] {}

SPAN[hello="Cleveland"][goodbye="Columbus"] {
color: blue; }

A[rel~="copyright"] {}

*[LANG|="en"] { color : red; }

*[id^="content-supp"]#content-supp:lang(en) {
    left : 34.6%;
    margin-left : 0;
```

```
}
```

11/25/2008

As pointed out by Andrew Stellman, compound classes were originally overlooked. The parser is looking at the various types of selectors that can be chained together (i.e. id, class, pseudo-class) recursively. However, not expecting to see more than one of any of those in a single selector, with compound classes it was overwriting previous classes with the next. So the last class in a compound would be displayed. CSS compound classes look like this:

Hide Copy Code

```
.boldtext.redtext {
    font-style: italic;
}
```

Since this wasn't accounted for in the original design, the model doesn't have a representation of such a construct. As a temporary fix, I made a small change in the attribute code of the SelectorName production in the ATG grammar file to append class names on subsequent discoveries. (This didn't require a change to the grammar, just the attributed code in the grammar) So, when parsing the above example, you will have a selector named ".boldtext.redtext". This works, and renders correctly. I'm not completely satisfied with this solution

since it doesn't really represent the structure, but it does solve the problem. When time permits, I will see about working up a new parser that will handle CSS 3.0 as well as possibly the Mozilla and Microsoft extension.

12/05/2008

This update includes the release of an entirely re-written grammar and model. This version is much closer to the actual CSS 2.1 spec, with some support for CSS 3.0 elements. If you are dependant on the previous model, the original v1.0 downloads are still available. The parser now supports CSS 2.1 & 3 combinators, attribute selectors, any functions, any @ rules, CSS 3 units, and most extensions.

10/06/2009

Finally, I've taken the time to rewrite the grammar to observe whitespaces. Which should address the issues pointed out by brisemec, crlord, Marcus Bernander and wknopf.

Recent History

12/31/2010

Fixed (hopefully) an issue with negative number values. It wouldn't surprise me if I introduced new errors, but I have a good list of test files that all passed.

1/3/2011

Fixed an issue where a pseudo element was consuming trailing spaces causing some selectors to be combined.

3/21/2011

Fixed a couple of issues with the model that prevented it from being XML serialized.

7/8/2011

Fixed a couple minor issues like double commas in the re-render. Also fixed some issues with some complex filter examples.

09/02/2011

As reported by pagey47, there were still a couple of places that should have been expecting whitespace. Languages that require you to observe whitespace make a grammar cumbersome. I found a couple of places, there may still be more. This particular issue affected filter parameters with equal signs:

Hide Copy Code

```
.test {
    opacity: 0.2;
    filter: alpha(opacity = 20);
```

```
}
```

failed to parse because of the whitespaces on either side of the equal sign. This should be resolved.