|  |  |  |
| --- | --- | --- |
| **Microsoft Ajax Minifier Documentation** | | |
| Development | ronlo | Last saved: 12/13/2009 10:59 AM |
| Program Management |  | Document state: **Draft** |
| Test |  | |

**Table of Contents**

[**Microsoft Ajax Minifier Documentation** 1](#_Toc248404067)

[1. Introduction to JavaScript Minification Concepts 1](#_Toc248404068)

[2. Command-Line Usage 2](#_Toc248404069)

[3. Default JavaScript Minification 4](#_Toc248404070)

[4. Local Variable and Function Renaming 9](#_Toc248404071)

[5. Conditional Compilation Comments 12](#_Toc248404072)

[6. Variable Renaming With and Eval Statements – Not! 13](#_Toc248404073)

[7. Analyzing Your Script 14](#_Toc248404074)

[8. Specifying External Globals 16](#_Toc248404075)

[9. Maximizing Your Local Variables and Functions 16](#_Toc248404076)

[10. Cross-Browser Peculiarities 18](#_Toc248404077)

[Ambiguous Try/Catch Variables 18](#_Toc248404078)

[Ambiguous Named Function Expressions 19](#_Toc248404079)

[11. Other Coding Tips 20](#_Toc248404080)

[12. Introduction to CSS Minification 20](#_Toc248404081)

[13. Default CSS Minification 21](#_Toc248404082)

[14. CSS Minification Options 21](#_Toc248404083)

[15. CSS Comment-Based Hacks 22](#_Toc248404084)

[16. Merging Localized Resource Files 22](#_Toc248404085)

[17. Pretty Print 24](#_Toc248404086)

[18. DLL Version of Microsoft Ajax Minifier 25](#_Toc248404087)

[19. Error Ouput 28](#_Toc248404088)

# Introduction to JavaScript Minification Concepts

JavaScript has the dubious quality of not being compiled. The source code text for the program is downloaded by the client, parsed, and the interpreted by the browser. Verbose, easy-to-maintain coding styles promoted by responsible software engineers actually hurts performance with JavaScript, as it greatly increases the page weight transmitted to the client.

There have been a number of utilities out there for “minifying” JavaScript. The first level of minification is to remove whitespace and comments. This is a minimum amount to be expected for a website – comments and whitespace can be extracted from the sources without changing the semantics of the code. The third-party tool Jsmin from Douglas Crockford (<http://www.crockford.com/javascript/jsmin.html>) is an excellent example of this kind of minification.

The second level of minification is to also remove excessive semicolons and curly-braces. In JavaScript, semicolons are not statement terminators as in C or C#, they are *separators*. Therefore the final statement in a block does not require a semicolon after it. Curly braces, although highly-recommended for properly-maintained JavaScript code, also don’t need to be around single-statement blocks. We don’t have to send those bytes to the user, however, and they can and should be removed before deployment. Developers also tend to over-parenthesize their expressions in order to make them more readable or to ensure that the intended precedence will unfold as expected. Extra parentheses can also be removed from the output. The danger is that this level of minification could alter the semantics of the code if not done properly.

The next level of minification is to recognize the fact that local variables and functions are referenced only within a known scope, and can therefore have their identifiers shrunk down to shorter names. This is highly useful because developers can code with longer, semantic names in their local functions and variables in order to increase readability and maintainability without having to penalize the client with large downloads. These modifications must be very carefully made because this level of change to the sources is greatly increased. Scope chains must be respected or the code will cease to work properly, and global variables cannot be touched (since they may be referenced from other script files).

The next level is to analyze the code and recognize unreachable statements and functions, and unused variables. Eliminating these bytes from the download can go a long way to shrinking your page weight, without having to require constant manual reviews of the code base. Again, this must be very carefully done to ensure that a function or variable really isn’t referenced.

The next level of minification is to analyze the usage of global variables and object properties, and to provide shortcuts in the code the replace the longer, repeatedly used constructs. For example, if the algorithm detects that a script repeated accesses the “window” global object, a one-letter variable can be set to the window object, and every other instance of “window” in the code can be replaced with that single letter. This not only modifies the existing sources, but also injects new code into the JavaScript. What can be replaced must also be carefully determined to make sure that the values and objects cached in the short local variables will remain the same value for the life of the variable. This level of minification is not for the faint of heart.

Microsoft Ajax Minifier by default will try to reduce the code as much as possible: removal of comments, whitespace, and unnecessary semicolons and curly-braces; renaming of local variables and functions to shorter names; and removal of unused or unnecessary code. The -rename and –unused switches can be used to alter this default behavior.

# Command-Line Usage

Simply running AjaxMin without any parameters, or with the -? or /? switches, will display the command line usage. The bare minimum needed to minify a source file is the input file name:

ajaxmin inputfile.js

The type of minification (JS or CSS) will be switched depending on the file extension of the source file. Alternatively, either the –JS or –CSS switch can be used to explicitly put the tool in one of the two modes. If the combination of switched or input file extensions leads to an ambiguous JS or CSS state, an error will be thrown.

Not specifying an output file will send the minified code to the standard output stream. If you want to save the output directly to a file, use the –OUT option:

ajaxmin inputfile.js –out outputfile.js

By default, if the output file already exists, an error will be thrown and the file will not be over-written. To always write the output file, regardless of whether or not it already exists, use the –CLOBBER switch:

ajaxmin inputfile.js –out outputfile.js –clobber

By default, the tool will rename all local variables and function. If you do not wish to rename any local variables or functions, use the –RENAME option with the “none” value:

ajaxmin –rename:none inputfile.js –out outputfile.js

Another option has to do with a practice here at Microsoft that goes back many years and is supported by localization tools. In this practice, variables that contain strings and other data that need to be localized for different markets and languages have their names prefaced with “L\_”. For example, if a message is displayed in an alert box, the code might be written like this:

var L\_hello\_text = "Hello there! ";

alert(L\_hello\_text);

Tools can then be employed to search the code for variables that start with “L\_”, show them to the localizer (who typically is not a programmer and does not understand JavaScript code), who then translates the string and the tool replaces it in the code. Sites that use this scheme need to keep L\_-variable names intact so they can be localized. To keep only those variable names unchanged, specify the “localization” value to the -RENAME parameter:

ajaxmin –rename:localization inputfile.js –out outputfile.js

To generate analysis output on your code in order to find possible bugs earlier in the development process, use the –ANALYZE option. The –GLOBAL option is for specifying known global entities, which is useful when analyzing your code with the –ANALYZE switch:

ajaxmin –analyze –global:Msn,jQuery inputfile.js –out outputfile.js

The warning level determines what levels of warnings are displayed when minifying your code. The default warning level is zero, meaning only serious errors are displayed. The -ANALYZE option by default shows all warning levels. The cutoff can be specified manually with the –WARN option. For example, if you wish to analyze your code but don’t want to see the “suggestion” level of warnings (level 4), specify a warning level of 3:

ajaxmin inputfile.js –analyze –warn:3

The other options are less used. The -SILENT option is useful only when saving to a file without analyzing and you don’t want any output sent to the standard output stream. The encoding option (-ENC) is for specifying an encoding scheme for input (-ENC:IN) and output (-ENC:OUT) files other than the default of UTF-8 for input and ASCII for output. This mainly affects the way string literals are encoded. With the ASCII encoding scheme, extended characters are encoded such that all browsers can read them. A UNICODE encoding scheme, for example UTF8, would insert actual UNICODE characters directly in the strings.

The –TERM option adds a semicolon to the end of the minified stream. This can be useful if you need to later append multiple minified files together into a single file using a separate process.

The –DEBUG option is for specifying how to handle debug-related statements. By default, AjaxMin will remove debug-related statements: debugger, for instance. Statements constituting calls to the debug namespaces of a couple well-known libraries will also be culled. These namespaces are Web.Debug, Msn.Debug, Debug and $Debug. Calls to the WAssert function will also be stripped. If the calls are part of a constructor, the constructor will be replaced with an Object constructor. For example, this code:

debugger;

Web.Debug.Output("foo");

$Debug.Track("bar");

Debug.fail("debug fail");

WAssert(condition,message);

var wnd = new $Debug.DebugWindow();

Will be reduced to this:

var wnd={};

Notice that the debugger statement and the calls into the debug libraries have been totally removed, and the constructor call replaced with an empty object literal.

In order to mark a block of arbitrary code as debug-only, enclose it in comments of a special format:

///#DEBUG

Code between these two comments will be removed in normal mode,

and only shown when in debug mode

///#ENDDEBUG

The comments are three slashes (/), a pound-sign (#), followed by the word “DEBUG” or “ENDDEBUG” (respectively), with no whitespace between any of them.

To keep the debug statements, specify a Boolean value indicating true (true, t, yes, y, on, or 1) after the –DEBUG switch:

ajaxmin –debug:true inputfile.js –out outputfile.js

It is also possible to minify JavaScript sent in from the standard input stream. Microsoft Ajax Minifier will display the usage message if there are no parameters at all, so if you wish to minify from the standard input, at least one optional parameter must be specified.

# Default JavaScript Minification

Microsoft Ajax Minifier minifies script by parsing the source code into a JavaScript syntax parse tree using code based on the ROTOR sources published by Microsoft. Once the original sources have been parsed, the tree is manipulated, and then walked to output the minimum amount of JavaScript code required to reproduce a comparable parse tree. Microsoft Ajax Minifier does not alter the source file. Default minification will:

* Remove unnecessary white space.
* Remove comments (except for statement-level “important” comments).
* Remove unnecessary semicolons.
* Remove curly-braces around most single-statement blocks.
* Rename local variables and function.
* Determine best string delimiters (single- or double-quotes) based on which option will generate the fewer escaped characters within the string.
* Combine multiple adjacent variable declarations.
* Remove empty parameter lists on constructors.
* Remove unreferenced names for named function expressions.
* Remove unreferenced local functions.
* Remove many instances of unreachable code.

Whitespace and comment removal is pretty well-understood. There is nothing in the JavaScript specifications that requires the code to live on multiple lines; Microsoft Ajax Minifier outputs the entire resulting source into a single line, removes spaces between operators, and leaves spaces only where necessary to delimit keywords and identifiers. Comments are removed unless they are “important” statement-level comments. Important comments are multi-line comments in the format /\*! … \*/ (the exclamation mark must be immediately following the opening delimiter’s asterisk, with no white-space between them). Use the important-comment construct to add license or copyright notices to the top of your script and the comment will be sent to the minified output.

Because semicolons are *delimiters*, not terminators, extra semicolons are usually removed from the minified output. The last statement in a statement block does not need a semicolon, and the extra byte can be saved. This block:

if ( a == 0 )

{

a = 10;

alert(b/a);

}

Reduces to:

if(a==0){a=10;alert(b/a)}

There are exceptions to this rule. For example, Safari will throw an error if a throw statement is not terminated with a semi-colon, so one will be present in those cases, even if the statement is the last statement in a block. If this behavior is not desired, you can turn it off with the –MAC command-line option, specifying a false value.

Removal of unnecessary curly braces is equally simple. If a statement that normally expects a block of code (like if or for or while) only has a single statement within that block, the curly-braces are removed.

if ( a == 0 )

{

a = 10;

}

Reduces to:

if(a==0)a=10;

There are also exceptions to the curly-brace-single-statement rule. If an if statement contains an else block and the true block contains a single statement, that true block statement will be enclosed in curly braces if it ends with an if statement *without* an else block. Without the curly-braces, the else in the outer if would instead get parsed with the inner if statement. This code:

if ( a == 0 )

{

for(var o in opts)

{

if ( o > 0 )

{

a += o;

}

}

}

else

{

b = c / a;

}

Gets changed to:

if(a==0){for(var o in opts)if(o>0)a+=o}else b=c/a

Another exception is when a block contains a function declaration. Strictly speaking, according to the ECMA spec, function declarations are not allowed in the statement blocks of statements such as if, while, or for. However, browsers will parse those declarations in those spots. Safari will throw an error if the block is a single function declaration not enclosed in curly-braces, so curly-braces will always be added in those circumstances (unless the –MAC switch is specified with a false value).

In JavaScript, string literals can be delimited with single- or double-quotes. If a string is delimited with single-quotes, all instances of single-quotes within the string must be escaped with a backslash. Same goes for double-quotes. No matter what the programmer escapes the source string with, the output string will be delimited with whichever character produced the fewest escaped characters within the string, thereby reducing the number of bytes. The string:

var g = "what's his \"name\"?";

Gets automatically minified to:

var g='what\'s his "name"?'

Because double-quote delimiters requires two escapes, but single-quote delimiters only require one.

Well-written code defines each variable on a separate line. This allows for maximum maintainability and readability. However, this means multiple var statements, which means more bytes downloaded. Microsoft Ajax Minifier combines multiple adjacent var statements into a single comma-delimited var statement:

var a = 0;

var b = "some string";

var c = 3.14;

Gets reduced to:

var a=0,b="some string",c=3.14;

The object of a new operator is a function – either the name of a declared function, or a function expression. The new operator creates a new object, sets the this pointer to that object, then calls the function. The parameter list is actually optional; if the function is not passed any parameters, it is not necessary to add an empty set of parenthesis after it. So for instance:

var img = new Image();

becomes:

var img = new Image;

Unless the –NEW:KEEP option is specified, explicit calls to the Object and Array constructors are converted to object and array literals. For example, this code:

var obj = new Object();

var arr = new Array();

var lst = new Array(1, 2, 3);

Gets reduced to:

var obj={},arr=[],lst=[1,2,3];

The exceptions here are when the Object constructor is passed any parameters, or when the Array constructor is passed a single parameter that may be numeric (this indicates an array size, not an element list).

Because JavaScript only has function-level variable scopes, nested blocks do not add anything to the semantics of a function and therefore will be removed from the output. For example:

function foo(p)

{

var f = 10;

{

var g = 0;

f = p \* g;

}

}

Gets converted to:

function foo(a){var f=10,g=0;f=p\*g}

If an if statement contains an empty true-block and a non-empty false-block, the condition will be not-ed and the false-block moved to the true-block:

if (a >= b) {} else { alert("a!=b") }

if (foo.bar()) {} else { alert("not foo.bar()") }

if (!a) {} else { alert("a") }

gets converted to:

if(a<b)alert("a!=b");

if(!foo.bar())alert("not foo.bar()");

if(a)alert("a");

Although it may seem like a silly bit of code to write, sometimes the author of the source script will actually have statements in the true-block that get removed by AjaxMin; for instance, if the statements are debug-only statements.

If there are var statements immediately preceding a for statement, there are several situations where the statements will be combined, depending on the structure of the for statement’s initializer component. If the initializer is empty, the var statement will simply be moved into the for statement’s initializer:

var i = 5;

for (; i > 0; --i)

{

alert(i);

}

will be converted to:

for(var i=5;i>0;--i)alert(i)

If the for statement’s initializer is already using the var construct, the var statement and the initializer will be combined:

var n = 10;

for (var i=5; i > 0; --i)

{

n \*= i;

}

will be converted to:

for(var n=10,i=5;i>0;--i)n\*=i

If the for statement’s initializer is a simple assignment to a variable, and that variable is in an immediately-preceding var statement, the var statement and the initializer will again be combined:

var n=10, i;

for (i = 5; i > 0; --i)

{

n \*= i;

}

will be converted to:

for(var n=10,i=5;i>0;--i)n\*=i

If the for statement’s initializer does not use the var construct, and uses the comma operator to make multiple assignments, no preceding var statements will be combined into the for statement’s initializer. For example:

var i;

for (i = 5, n = 10; i > 0; --i)

{

n \*= i;

}

will become:

var i;for(i=5,n=10;i>0;--i)n\*=i

because the n variable being assigned to is not part of the preceding var statement, and could possibly be a different context if placed into a var construct.

It would be possible to move the var statement into the for statement if the preceding var statement(s) contained both variables, but for now Microsoft Ajax Minifier will simply not perform the combination logic if the for statement’s initializer uses the comma operator at all outside the var construct.

Another scenario involves a common construct with if statements. In analyzing various pieces of code, a pattern emerged. In order to call a method on an object that might or might not exist, code similar to this was being written:

if (obj.method)

{

obj.method();

}

The pattern is an if statement with a single statement in the true-block, which is a call statement. This pattern can be reduced to:

obj.method&&obj.method()

This works because the and-operator shortcuts; if the first expression does not evaluate to true, the second expression is not even executed. Technically, the single statement within the true-block simply needs to be an expression statement in order for this pattern to work, however, the other typical construct is a single assignment expression within the true-block:

if (obj.prop)

{

i += obj.prop;

}

Were Microsoft Ajax Minifier to apply the same reduction pattern to this code, it would not gain anything because the assignment operator expression would require parentheses around it in order for the operator precedence to remain valid:

obj.prop&&(i+=obj.prop)

In that case, it would not be saving any bytes at all over the standard behavior:

if(obj.prop)i+=obj.prop

Therefore the pattern Microsoft Ajax Minifier employs will perform the transition from the if statement to an and-operator only if the single statement within the true-block is a method call expression.

# Local Variable and Function Renaming

Local function and variable names are renamed to shorter names, while global functions are left alone. Function scope chains are respected, so global and outer variable references are carried through without interruption or interference. Within a local scope, variables and functions are renamed starting with lower-case letters, then upper-case letters, then combinations thereof. Those with the most references are named first, thereby attempting to make the biggest gains for those fields that are referenced most frequently.

Another aspect of AjaxMin is reference counting and tracking. This is used to remove some unused code. For example, if a function defines arguments that are never referenced, they are removed from the tail of the argument list:

function DivideTwoNumbers(numerator, denominator, unsedparameter )

{

return numerator / denominator;

}

Gets reduced to:

function a(a,b){return a/b}

Because the last parameter is never referenced, it is not part of the output. If an unused parameter is followed by a referenced parameter in the list, it is not removed because it affects the order of the used parameters. In JavaScript it is perfectly normal to pass fewer or more parameters than the function formally declares – only the order matters.

Scope chains are analyzed to find unreachable code in the form of local functions that are not actually called. Those functions are removed from the resulting output. In addition to the normal function that simply isn’t called from anywhere, the analysis also works with recursive functions (functions that call themselves) and chains of functions that call each other but aren’t called from anywhere else. A global function is never removed because it may be called from other modules outside the file being minified. Function expressions are assumed to be referenced by their containing scope.

The parameter to the catch statement provides an opportunity to write code that behaves differently in different browsers. In Internet Explorer, catch parameters are simply variables defined within the containing function scope, so code like this is perfectly acceptable:

function foo()

{

try

{

// do something that might error

}

catch(e)

{

// handle the error

}

alert(e);

}

All other browsers, however, would throw a script error because e is undefined outside the catch block. This may seem like a trivial difference, but when using the variable renaming feature, we have to be sure we’re pointing to the right field to produce the desired results. For instance, take the code below:

var e = "outer";

function foo()

{

try

{

// do something that might error

}

catch(e)

{

// handle the error

}

alert(e);

}

In all browsers but IE, the alert function in the above code will always display “outer,” whether or not there was an error and the catch block was executed. In IE, this code will display “[object Error]” if there was an error, and “undefined” is there wasn’t. If the outer “e” variable gets renamed to “a” and the catch argument gets renamed to “b,” then the alert parameter needs to be “a” to reference the proper value in non-IE browsers. But for Internet Explorer – and *only* for IE – the alert parameter needs to be “b.” It is generally a bad idea to code a function using try/catch statements such that the catch argument is the same name as a variables referenced *outside* the catch block (as in the above example). Such a pattern will behave differently in different browsers and should be avoided. As an aid to developers, an error will be thrown if Microsoft Ajax Minifier detects such a situation:

Possible coding error: Ambiguous catch identifier 'e'. Cross-browser behavior difference.

At line 8, col 10-11: e

Switch case statements will also be manipulated. For instance, if the final case/default statement in the switch ends with a “break” statement, that statement will be removed, since control flow will fall out of the switch block after the last statement in the last case is executed anyway. If there is a “default” case and it only contains a break statement, it will be removed. If there is no default case (or if an insignificant default case is removed), then any other cases that only contain a break statement will also be removed as insignificant. The exceptions are if the break statements break to a label that is not the containing switch statement.

Frequently-used literals can be combined into local variables when the –LITERALS:COMBINE option is used. For instance:

function foo(p)

{

p[0].style.display = "block";

p[1].style.display = "block";

p[2].style.display = "block";

p[3].style.display = "block";

p[4].style.display = "block";

}

Would get reduced to:

function foo(p)

{

var a="block";

p[0].style.display=a;

p[1].style.display=a;

p[2].style.display=a;

p[3].style.display=a;

p[4].style.display=a

}

The literals that may be combined are: true, false, null, numeric values, and string values. Be aware that only literals within a local scope or tree of local scopes will be combined in this way. No global variables will be created by Microsoft Ajax Minifier to eliminate the possibility of colliding with other global values. If the same string literal is frequently used across global functions, for example, the code will not create a global variable shared by all the functions.

One possible problem with using the –LITERALS:COMBINE option is that the algorithm affects the entropy of the resulting JS file in such a way that further compression by the web server (for example, using over-the-wire gzip) may not be as effective as if applied against code that doesn’t combine the literals. If you use the –LITERALS:COMBINE option, be sure to test your file sizes after compression against the normal, un-combined results as well. Although the –LITERALS:COMBINE results may be significantly smaller before compression, they might be *larger* than the regular results after compression.

One aspect of minification can be seen as either good or bad, depending on whether or not you’re trying to debug the deployed code. Variable-renamed code is very difficult to read – it could be seen as a sort of mild (but maddening) form of obfuscation. First, all the comments and semantic names are removed and all the code is on a single line. Then add that each function scope starts its renamed variables over again from “a,” and trust me – stepping through this kind of code in a debugger will drive you insane.

# Conditional Compilation Comments

AjaxMin supports only a subset of conditional compilation comments. The @cc\_on, @if, and @set statements are supported – it’s *where* they are in your code that determines whether or not they will be passed through to the output.

In general, statement-level conditional comments are supported. For instance, if you had the following code:

var ie = false;

/\*@cc\_on

ie = true;

@\*/

alert(ie ? "Internet Explorer" : "NOT Internet Explorer");

It will get minified to:

var ie=false;/\*@cc\_on ie=true;@\*/alert(ie?"Internet Explorer":"NOT Internet Explorer")

This is because the conditional-compilation comment starts at a statement boundary and only contains whole statements within it.

There is also a special-case scenario in which the var-statement can specify the initializer of a variable within conditional-compilation comments. This is useful for defining IE-specific values, like:

var ie/\*@cc\_on = 1 @\*/;

alert(ie ? "Internet Explorer" : "NOT Internet Explorer");

which reduces to:

var ie/\*@cc\_on=1@\*/;alert(ie?"Internet Explorer":"NOT Internet Explorer")

The rule here is that both the equals-sign and the initializer – but nothing else – must be included in the conditional-compilation comment. Anything else will be ignored.

Conditional-compilation comments in general are *not* supported by AjaxMin within JavaScript expressions. However, if a conditional-compilation comment is encountered inside an expression, and the comment contains only a single conditional-compilation variable reference, the comment will be retained. Anything else and it will be ignored. So for instance:

//@set @fourteen = 14

var fourteen = /\*@fourteen @\*/;

alert(/\*@\_jscript\_version @\*/);

will reduce to:

/\*@set@fourteen=14@\*/var fourteen=/\*@fourteen@\*/;alert(/\*@\_jscript\_version@\*/)

The above code will obviously not work in non-IE browsers, but it is just a simple example to illustrate how conditional-compilation comments *within expressions* can only contain references to variables in order to be retained in the output.

*However*:

var isMSIE = /\*@cc\_on!@\*/0;

Will have the comment ignored and simply become:

var isMSIE=0

because the comment contains an operator and not a variable reference. Remember the general rule: use conditional-compilation comments at a statement level and AjaxMin should always retain them.

# Variable Renaming With and Eval Statements – Not!

Renaming of local variables and functions really only works if the code is completely known at parsing time. Unfortunately that isn’t always the case with JavaScript. There are two statements in particular that determine the scope chain at runtime: with and eval.

The with statement takes an expression that evaluates at runtime to an object that is placed at the head of the scope chain. For example:

var foo = 10;

with(window)

{

alert(foo);

}

With this code, we have no idea at parse time whether “foo” is referring to the local variable or to a property on the window object.

The eval statement is even more destructive to the variable-renaming algorithm. It takes an expression that evaluates to a string that is then parsed and executed as more JavaScript code. There is no way to programmatically determine at parse time if the string will declare more variables or functions, reference existing variables or functions, or whatever.

The eval statement marks its containing scope as “unknown.” Scopes that are unknown at parse time may have significant errors introduced if they depend on existing variables and functions. The –EVALS switch modifies this behavior to reduce the minification of the algorithm, but remove the risk of the eval statement. If –EVALS:MAKESAFE is specified, no “unknown” scopes will participate in the variable renaming process. This ensures that the code that is run in an eval statement will continue to work in minified code, just as it would in the original sources. If the developer knows that the eval statement will only need variables or functions defined in the containing scope and not in any of the parents, the –EVALS:IMMEDIATE switch can be used. The containing scope will not participate in the local-renaming process, but parent scopes will continue to do so.

In the with statement example above, if we were to change the variable “foo” to “a” and the window object doesn’t contain a “foo” property, we’ve broken the link to the local variable. And if we also change the parameter to the alert function, we could be breaking the link to a foo property on the window object. Every variable and function referenced within a with-scope must remain the same name because we just don’t know.

If you must use eval and with statements, try to isolate them into small scopes under the global level that are tightly coded without long variable or function names. In general, these statements are frowned upon by most JavaScript developers anyway, and the use of them will throw a level 4 warning in AjaxMin.

# Analyzing Your Script

Because JavaScript is an interpreted language, and a type-less one at that, developers do not get the advantage of compile-time checks available to developers in other languages. It is a good idea to always run JavaScript sources through a lint-style checker before deployment. Douglas Crockford maintains a very good jslint application at his website (<http://www.crockford.com/jslint>).

Microsoft Ajax Minifier has a modicum of linting-style capabilities, but they will be increased over time. Using the -ANALYZE option on the command line will change the default warning level to show all warnings (which can be overridden with the –WARN option), and will spew a ton of analytical information for all the scopes in your code. It will list all the global object (variables and functions) defined by your code, then each function scope in order of its appearance in the code. Within the function scopes, each variable and function defined or referenced is listed, along with its status – argument, local, outer, global, etc. If local-renaming is turned on, it also lists what name that field has been renamed to, which is a good aid for debugging. If a function scope is marked as “unknown,” it will be noted in the analytical output, as will a status of “unreachable.”

Using the –ANALYZE option for Microsoft Ajax Minifier, you can find JavaScript coding problems before they cause actual bugs. In JavaScript, you can use variables without defining them – the runtime assumes they are properties on the global object, which is the window object within a browser. This is bad for performance reasons – instead of binding to a local variable when assigning to a variable that isn’t defined, an expando property is created on the window object. It is a very good idea to look at the analysis and ensure that all variables your code is referencing are defined within the proper scope.

function func(p,m)

{

for(n = 2; n < m; ++n)

{

p \*= n;

}

return p;

}

In the code snippet above, the window object will get an expando property named “n” created. Microsoft Ajax Minifier will output a warning for all global variables referenced but not defined. The developer should look at all these warnings and fix instances like the one above, where a local variable definition is called for.

JavaScript also allows you to redefine variables that were previously defined within the same scope, which could generate unexpected results and errors. JavaScript only has the global scope and function-level scopes. It does not have block-level scopes as languages like C and C# do. Take, for instance, this code:

function func()

{

var n,a = "outer";

for( n = 0; n < 10; ++n )

{

var a = n;

// do something else

}

alert(a);

}

In C or C#, the alert function would be passed the string “outer,” but in JavaScript it would be passed the number 10. The JavaScript code is actually defining the variable, “a”, *twice* in this code snippet. This is not a runtime error in JavaScript – the second definition simply reuses the existing variable. But it may not generate the output the developer intended. Microsoft Ajax Minifier will display an error for all variables that are defined multiple times within the same scope. Developers should fix these bugs before shipping their code.

Various style warnings may be thrown as well. For instance, Microsoft Ajax Minifier will locate all single-statement blocks that aren’t enclosed in curly braces, and warn the developer that they should add them. This is because developers can sometimes unintentionally introduce bugs in their code by not liberally using curly-braces. Curly-braces also increase readability and maintainability. It is strongly encouraged that developers always enclose their blocks in curly-braces, especially since Microsoft Ajax Minifier will remove them where they aren’t necessary before their code gets deployed. Unlike jslint, Microsoft Ajax Minifier does not force any particular curly-brace coding style – separate-line or same-line – upon the developer (other than to use them).

In the future, Microsoft Ajax Minifier might also alert the developer to statements that don’t end in semi-colons – another possible source of runtime bugs. At the time of this writing, those alerts are not generated.

The format of the analytical output may change in the future to allow for better use by other tools like Excel. At the time of this writing, the output is always made as text to the standard output stream. A function scope will indicate which line the function begins on, then list all the variables and functions referenced within that function. If local-renaming is enabled and the function is known at compile time, what name the fields are renamed to is also listed:

Function AddForecastCallback - starts at line 2233 (renamed to s)

context [argument] (renamed to k)

response [argument] (renamed to g)

cities [local var] (renamed to f)

city [local var] (renamed to i)

info [local var] (renamed to h)

row [local var] (renamed to j)

m\_dataUrl [outer var] (renamed to l)

m\_gatorBinding [outer var] (renamed to a)

m\_itemList [outer var] (renamed to c)

m\_listEl [outer var] (renamed to b)

m\_searchbox [outer var] (renamed to d)

m\_this [outer var] (renamed to e)

AddForecastCallback [outer function] (renamed to s)

AddWeatherCity [outer function] (renamed to t)

ParseWeatherResponse [outer function] (renamed to r)

SetRowBorders [outer function] (renamed to o)

Msn [global var]

url [global var]

document [global object]

escape [global function]

In this particular example, the variable “url” is indicated as a global variable. If we were to check the errors returned, we will find this in the output:

Possible performance problem: Variable 'url' has not been declared

At line 2287, col 7-10: url

This is an example of an undefined variable that gets added as an expando property on the window object. Most-likely the developer intended this reference to be a local variable.

After the scope report, the Undefined Globals report is output if there are any undefined global variables or functions. This report will list every occurrence of a reference to a variable that was not defined within the scope chain and is assumed to be a global defined elsewhere. It is a very good idea to look at each and every one of these instances and ensure that the developer did not accidentally leave out a local variable definition, as it could be a performance bug – or worse.

Undefined Globals:

DOMParser (Constructor) at Line 42, Column 20

infopaneChooserForm (Variable) at Line 4196, Column 15

infopaneChooserForm (Variable) at Line 4229, Column 14

infopaneChooserForm (Variable) at Line 4240, Column 14

infopaneClose (Variable) at Line 4200, Column 15

infopaneClose (Variable) at Line 4233, Column 14

m\_itemList (Variable) at Line 3298, Column 4

ppstatus (Variable) at Line 1505, Column 4

registerNamespace (Function) at Line 1, Column 1

url (Variable) at Line 2287, Column 7

XMLSerializer (Constructor) at Line 50, Column 24

In this example, DOMParser, infopaneChooserform, infopaneClose, registerNamespace, and XMLSerializer are all objects or functions that I know are defined in other modules. However, m\_itemList, ppstatus, and url should be local variables.

# Specifying External Globals

Using the analyze feature, Microsoft Ajax Minifier may throw errors for undefined global variables that you know are defined in other modules. These warnings can be eliminated by specifying the known globals on the command-line using the –GLOBAL option. For instance, if you know your code will reference XMLSerializer, Msn, and HelperFunction in your code, but they are defined in another file, you can run Microsoft Ajax Minifier like this:

ajaxmin –analyze inputfile.js –global:XMLSerializer,Msn,HelperFunction

Those specified identifiers will be treated as if they were defined in your modules as global variables, and warning messages will not be generated. This makes it easier to locate the real errors that need to be fixed in your code.

# Maximizing Your Local Variables and Functions

It’s not enough to take any JS file and run Microsoft Ajax Minifier on it. Yes, it should make your downloaded JavaScript file smaller, but there are way of writing your code that will maximize the benefits of minification.

For starters, the developer must recognize that global function and variable names are never renamed. If you code is written in the classic method of defining a series of global functions that call each other and act on global variables, only the local variables within each function will be renamed. A gain will be made, but not as much as could be. The proper coding style is to leave those functions as global those that are truly global, and to next all others as local. This not only will help with your code minification, it will also minimize the possibility of naming collisions between your code and any other JavaScript code that might be added to the web page.

The simplest way of doing this is to define a single global object in your script file as a namespace scope. Within that object, define “global” functions as methods on that object. Any other functions defined will be hidden from other modules as local functions, and therefore avoid naming collisions and be renamed for space gains.

Let’s see an example. Let’s say we are writing a JavaScript file that will provide two methods (“Start” and “Stop”) and a variable, “Status” for the markup or other code to be able to call or reference. There may be any number of other helper functions or variables, but none of them need to be exposed to any other modules:

var MyScope = new function()

{

var my = this;

var requestObject = null;

my.Status = 0;

my.Start = function( url )

{

if ( my.Status == 0 )

{

startRequest( url );

}

else

{

alert("request processing");

}

}

my.Stop = function()

{

if ( my.Status != 0 )

{

cancelRequest();

}

}

function startRequest( url )

{

// kick off the request

my.Status = 1;

}

function cancelRequest()

{

// cancel a pending request

my.Status = 0;

}

};

In this code, MyScope is a global field and will therefore not be renamed, but every other variable and function will be renamed since they are all local to the MyScope object. Properties are not local fields and are therefore also not renamed; so the MyScope object will continue have the property “Status” and the methods “Start” and “Stop.” The variables “my” and “requestObject,” the argument “url,” and the helper functions “startRequest” and “cancelRequest” are all local and will be renamed. They can be as long and semantic as readability requires without having any impact on the client download performance. JavaScript code outside this module (or even within) can call the public methods and properties as MyScope.Status, MyScope.Start or MyScope.Stop. The internal local variables and functions are essentially private and not exposed to any other code outside the MyScope object.

The “my” local field is not necessary, but it’s a very valuable shortcut for these types of namespace objects. The “my” variable will be renamed, most-likely to a single character. If the developer instead used the this literal everywhere, it would not get renamed and would remain at four characters everywhere it is used. AjaxMin will take create a local variable, set it to the this pointer, and use that variable wherever the this pointer is used. It will only do this, however, within the scope directly applicable to the this pointer; it won’t propagate the generated local variable into child scopes because the context of the this pointer in child functions may not be the same.

Namespaces can be nested. At the Msn Portals team, we typically define our namespaces within a first-tier namespace of “Msn.” Our code files typically start off like this:

if (!window.Msn){Msn={};}

Msn.MyNamespace = new function()

{

// namespace object goes here

};

If your code is using the Atlas framework, there is even a helper function for defining the base namespaces: registerNamespace. It will determine if any of the namespace objects passed in to it need to be created or not, and if so, create them as empty objects. If you’re not using Atlas but plan on using multiple levels of namespaces, writing such a global helper function might be a good idea; it greatly simplifies namespace definitions:

registerNamespace("Msn.Portals");

Msn.Portals.MyScope = new function()

{

// namespace object goes here

};

The general coding rule of thumb for maximum minification is: always wrap your code within a namespace, exposing only those functions and variables that truly are global (used by other modules or namespaces). Everything else – all your helper functions and state variables – should be declared as local to your namespace object.

# Cross-Browser Peculiarities

It should come as no surprise to any web developers that different browsers implement the JavaScript language and run-time differently. Most of the time they are pretty much in-sync with only minor variations or differences in object model implementations. However, there are a number of arcane differences that if not watched carefully, can lead to very difficult bugs to diagnose in different browsers. Microsoft Ajax Minifier will attempt to flag those differences as errors so the developer will be encouraged not to engage in paradigms that will cause such problems.

## Ambiguous Try/Catch Variables

It seems that all browsers except IE properly follow the language specs for JavaScript and define the name of the error variable in the catch block within the catch block itself. IE defines the variable within the containing scope. This can lead to ambiguous references that will behave differently in different browsers. For example:

var a, e = 10;

try

{

// force an error

a = foo;

}

catch(e)

{

// error handling

}

alert(e);

In IE, there is only one variable named “e,” and the alert will be the “undefined variable ‘foo’” error. In all other browsers, the error object is scoped to the catch block only, and does not affect the variable defined in the containing scope; therefore the alert shows 10.

This code will throw an “ambiguous catch variable” error in Microsoft Ajax Minifier. If the developer wishes to produce solid code that works the same in all modern browsers, he should make sure the name of the error variable in the catch block is not the same as a variable referenced in the containing scope. The above code should be written as:

var a, e = 10;

try

{

// force an error

a = e / 0;

}

catch(err) // name should not collide with containing scope

{

// error handling

}

alert(e);

## Ambiguous Named Function Expressions

Named function expressions have the same problem. All browsers save IE follow the language spec and allow the name of the function expression to be scoped only to the function expression’s scope, thereby allowing recursive calls to be made. IE defines the name of the expression in the *containing* scope. The same problems can occur:

var foo = 10;

(function foo(cnt){

if (--cnt > 0)

{

foo(cnt)

}

})(10);

alert(foo);

All browsers but IE will properly run this code because the variable foo defined in the outer scope is not the same foo defined within the function expression scope. The inner call will recurse for 10 iterations, and then the alert will show 10. This code will not execute in IE! This is because for IE, there is only one field named foo. When the code is parsed, the function expression creates the field “foo” in the outer scope and assigned it the value of the function expression. Once execution begins, the first var statement sets that field to the value 10. The first time the function expression is run, it tries to recurse by executing the value of “foo” as a function – but it’s no longer a function; it’s the value 10 and an error is produced.

Microsoft Ajax Minifier will detect when a named function expression collides with a variable referenced in the containing scope, and will throw an error so the developer will know that the code will behave differently in different browsers.

# Other Coding Tips

Try to use object and array literals instead of the new operator and the Array and Object constructors. This will save nine or ten bytes per instance:

var o = new Object(); // long way

var o = {}; // short way

var a = new Array(); // long way

var a = []; // short way

var a = new Array(one, two, three); // long way

var a = [one,two,three] // short way

However, Microsoft Ajax Minifier will automatically make these substitutions unless the –NEW:KEEP parameter is specified.

The document and window objects are typically used over and over again within JavaScript code. It’s a good idea to shortcut those object within your namespace, thereby allowing minification to reduce the six or seven characters for each instance down to a single character:

var w = window;

var d = document;

// use w and d in your code instead of window and document

For DOM methods that are used frequently, supply local (and therefore renamable) shortcuts. For example, document.getElementById might be used many times in your code. Create a shortcut function:

function GetElById(id)

{

document.getElementById(id);

}

That function name could be renamed to a single character, thereby distilling document.getElementById(“id”) down to E(“id”). That’s a savings of over twenty characters per instance (not counting the original shortcut function). For frequently-used functions, the savings add up quickly.

# Introduction to CSS Minification

CSS minification as performed by this tool is not as complicated or in-depth as JavaScript minification. The CSS minification mostly consists of removing whitespace and most comments. Comments marked as “important” will always remain in the output, but depending on the settings used, certain comment-based hacks might also remain in the generated output.

Other small changes performed by the tool include reducing RGB values to their smallest possible length. For instance, rgb(255,128,0) would be changed to “#ff8000,” and “#ff9900” would be changed to “#f90.” Depending on the color-names setting, certain colors might also be changed to standard color names if they are shorter, or vice-versa. For instance, if strict (W3C-compliant) color names are to be used, “#ff0000” would get changed to “red.” There are also a larger set of color names that all major browsers recognize, but the use of which would produce invalid code when run through the W3C CSS validation tool. If the developer doesn’t care about generating invalid CSS (that still works in all major browsers), this setting can further reduce minified code.

Ideally a CSS minification tool would be able to analyze the rule set and eliminate redundant rules. This tool does not perform that kind of minification at this time.

# Default CSS Minification

As mentioned in the introduction, the CSS minification portion of AjaxMin does not perform as sophisticated algorithms as the JavaScript portion. By default it will:

* Remove all insignificant whitespace.
* Remove all comments.
* Remove all unnecessary semicolon separators.
* Reduce color values.
* Reduce integer representations by removing leading and trailing zeros.
* Remove unit specifiers from numeric zero values.
* Utilize W3C-strict color names to further reduce resulting code.

# CSS Minification Options

There are only a few CSS-specific command-line switches available in the Microsoft Ajax Minifier. Obviously, the –CSS switch to turn on the CSS Parser is one of them.

In order to specify how color names can be used in the resulting output, use the –COLORS switch. If the value of that switch is HEX (-COLORS:HEX), then no color names will be used, and the resulting RGB hexadecimal values will be substituted. If the STRICT value is used (default value), W3C-strict colors will be used if they are smaller than the hexadecimal representation. There are 17 colors recognized as W3C-strict: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white, and yellow. If the MAJOR value is used, the color set is expanded to a well-known set supported by all major browsers. Code that uses these color names, however, will not validate as strict-compliant. The complete list of colors can be found online at a number of sites (there are a couple-hundred names with equivalent RGB values).

By default, all comments are removed. This behavior can be modified by specifying the –COMMENTS switch with one of three values. The default behavior is the NONE values (-COMMENTS:NONE). If all comments are to be preserved, use the ALL value, and if certain comment-based hacks are to be preserved, use the HACKS value.

If all rules need to be terminated with a semicolon, then specify the –TERM switch.

The only thing the –ANALYZE option will do is display a message indicating the amount of reduction the tool made in the sources, and how much further g-zipping on the server would produce.

The –ECHO option can be used to send the input unchanged to the output.

# CSS Comment-Based Hacks

It has been very popular to use comment-based hacks to provide different CSS to different browsers without sniffing the user-agent string. It’s not a recommended practice, but it’s prevalent enough to warrant some attention by this tool. A number of popular hacks are preserved in the resulting code if the CssParser.CommentMode property is set to CssComment.Hacks. These hacks are (affected rules replaced with an ellipses):

* Ignore rule for Internet Explorer for the Mac (discontinue product): comments that end with an escaped asterisk in the comment-terminator sequence continue on until the next unescaped terminator in MacIE. The result is used to hide rules from that browser:  
  /\*(anything or nothing inside)\\*/.../\*(anything or nothing inside)\*/
* A bug in Netscape 4 and Opera 5 allows a particular comment format to hide rules form all browsers except those:  
  /\*/\*//\*/.../\*(anything or nothing inside)\*/
* A bug in Netscape 4 allows rules to be hidden from that browser:  
  /\*/\*/.../\*(anything or nothing inside)\*/
* A bug in IE6 can cause a property/value to be ignored if the property name is followed by at least one whitespace character and a comment before the colon:  
  property /\*(anything or nothing inside)\*/:value
* A bug in IE5.5 can be used to hide a property/value pair from that browser when a comment is between the colon and the value:  
  property:/\* (anything or nothing inside) \*/value
* And finally, if we are preserving comment-based hacks, when the parser encounters an empty comment, it will assume it was put there as a comment-based hack and preserve it:  
  /\*\*/ or /\* \*/

Important comments are always preserved, so one option for preserving comment-based hacks is to use that syntax: /\*! (whatever) \*/. The exclamation point must immediately follow the opening asterisk. It will be removed in the resulting output, but the rest of the comment will remain.

# Merging Localized Resource Files

Complex and global web applications frequently require localized JavaScript and CSS. There are many ways to perform that task. Microsoft Ajax Minifier provides support for merging RESX string resource files directly into your code at build time.

Let’s start off with an example. Let’s say you have a RESX file names “strings.resx” in your project that contains two localizable strings. The first one is called “Greeting” and contains a string that will be written to the DOM using a document.write call. The second one is called “Praise” and is used as the text of an alert box:



To “compile” this RESX file into your JavaScript source code, you supply the path to the RESX file and a global object name in the command line of ajaxmin via the –RES option:

-RES:*global* *path*

The *global* portion is a valid JavaScript identifier that your code expects to be defined containing all the globalized strings in the resource file. It should *not* be an identifier that is actually defined within your source; Microsoft Ajax Minifier will take care of integrating the virtual object into your code. The *path* portion is the path to the RESX file. For instance, if your code expects the strings.resx file to be using a global resource object named “Strings,” you would specify this on the command line of ajaxmin:

-RES:Strings strings.resx

Code your JavaScript source exactly as if that Strings object exists. For instance, your source code file (foo.js) might be something like:

document.write("<h1>" + Strings.Greeting + "</h1>");

alert(Strings.Praise);

So when we minify foo.js specifying the strings.resx file as the resources using the command line:

ajaxmin foo.js –RES:Strings strings.resx

The resulting code will be:

document.write("<h1>Hello!</h1>");alert("Excellent!")

So what happened here? The steps Microsoft Ajax Minifier goes through to merge resources are as follows:

1. Open the resource file and create a *virtual* global variable named “Strings” (taken from the command line options) with the properties Message and Praise.
2. Process the input source code. Whenever a reference to a property on the global “Strings” object is encountered, the entire property reference is replaced with the string literal from the resource file. If there is no property on the virtual object (and therefore, no corresponding string in the RESX file) to match what is in the JS sources, an empty string is substituted.
3. Final-pass string literal concatenation combines the expression inside the document.write call to a single string literal.

If I then localized strings.resx into Chinese and create a RESX file named “strings-zh.resx”:



I would compile it against the same sources in the same manner:

ajaxmin foo.js –RES:Strings strings-zh.resx

But the resulting code would contain the localized strings. In this example, I saved the output to a file using the ANSI encoding, so the Unicode Chinese characters are properly escaped for the encoding scheme:

document.write("<h1>\u4f60\u597d\uff01</h1>");alert("\u771f\u68d2\uff01")

If the –ECHO option is used on the command line as well (copy un-minified input straight to output), then instead of replacing the property references with the literals, the actual Strings object is inserted at the head of the output, resulting in:

var Strings={Greeting:"Hello!",Praise:"Excellent!"};

document.write("<h1>" + Strings.Greeting + "</h1>");

alert(Strings.Praise);

This can be useful for debugging, but should not be used for production code. The reason is that the Strings object generated in this way will include *all* strings in the RESX file. When the properties used are replaced with literals, only those strings within the RESX file *that are actually used* will get into your resulting JavaScript code. This allows your project to maintain a single RESX for all your script localization needs without having to flood your JS files with unused strings.

For CSS, the scenario is a little different. The command-line switch is the same, although the “global” value of the –RES switch is not used. What you do is place a specially-formatted comment in your CSS right before the property whose value you wish to replace with a localized string. The format of this comment is: /\*[*id*]\*/, where “id” is the name of the string in the RESX file. When a comment like this is encountered, the string with the given name is looked up in the RESX file, and its value replaces the value of the next property encountered.

Let’s look at an example. Say your RESX file contains a string named “BodyFontFamily” so you can change the font of the body element depending on your market’s localization. The CSS property you would write would specify a default value, and be prefaced with the appropriately-formatted comment:

body

{

/\*[BodyFontFamily]\*/

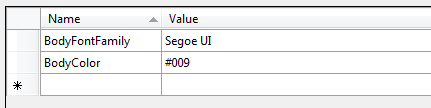
font-family: Arial;

/\*[BodyColor]\*/

color: black;

}

If your RESX file looks like this:



Then the resulting CSS when using the –RES option would be:

body{font-family:Segoe UI;color:#009}

If the RESX file does not contain a string with the name specified in the comment, the next property value is left as-is and not replaced.

# Pretty Print

Sometimes you need to debug your minified code. Trying to debug a single line of JavaScript thirty thousand characters long will drive you crazy – if your debugger can even handle it. Enter the -PRETTY option. It is recommended to use this option in your debug build process. It will perform all the same minification, but instead of outputting all the code on a single line, it will be output it all in an easy-to-read multi-line format. The default indent size is four space characters; to change that size, specify the numeric size of the indent on the –PRETTY switch, separated by a colon. For instance, to use only two spaces per indent level: -PRETTY:2.

For JavaScript, using the –PRETTY option will turn off local variable- and function- renaming unless explicitly turned on by using the –RENAME command-line switch.

When using the DLL API, the Pretty Print option is controlled via the OuputMode and IndentSize properties on the CodeSettings object.

The –PRETTY switch works with both JavaScript and CSS.

# DLL Version of Microsoft Ajax Minifier

A DLL version of the code is also distributed. It does not contain any of the advanced switching or lint-style reporting that the EXE contains, but it does provide access to the abstract syntax tree produced by the JavaScript parser. This can be highly beneficial to projects that wish to do advanced modification or examination of the parsed code tree.

If all you need is to minify input code, you can simply use the supplied Microsoft.Ajax.Utilities.Minifier object. Create an instance of the object, and pass your input code to one of the MinifyJavaScript methods, depending on the language and whether you want to override the default settings. The return value is the minified code string.

public string MinifyJavaScript(string source);

public string MinifyJavaScript(string source, CodeSettings settings);

The various CodeSettings object properties modify the way the output JavaScript is generated:

* **CollapseToLiteral**: convert “new Object()” to “{}” and “new Array()” to “[].” And of course, “new Array(1,2,3,4,5)” becomes “[1,2,3,4,5]” and “new Array(“foo”)” becomes “[“foo”]”. However, “new Array(5)” does not get minified, because that makes an array with five initial slots – it’s not the same as [5].
* **CombineDuplicateLiterals**: combine duplicate literals into local variables. So the code:

function foo(a)

{

a.b = 12345;

a.c = 12345;

a.d = 12345;

a.e = 12345;

}

gets changed to:

function foo(a)

{

var b=12345;

a.b=b;

a.c=b;

a.d=b;

a.e=b

}

The savings are much more dramatic with large, frequently-used strings. Works with numbers, strings, nulls, and this-pointers. The this-pointers only get minified within the current scope, since child functions might have a different meaning for the pointer. It will also only pull out reused variables within function scopes – it won’t create a global variable with the constant, as that may interfere with other global variables. For the maximum minification, wrap all your code within a namespace function scope. Beware, though, that gzipped results may end up being larger when using this option due to the change in entropy of the resulting code.

* **EvalTreatment**: Normally an eval statement can contain anything, including references to local variables and functions. If it is expected to do so, when the tool encounters an eval statement, that scope and all parent scopes cannot take advantage of local variable and function renaming because things could break when the eval is evaluated and the references are looked up. To reduce the amount of resulting minification but make sure that all possible references in evaluated code will hold true, use the MakeAllSafe value. However, sometimes the developer knows that he’s not referencing local variables in his eval (like when only evaluating JSON objects), and this switch can be set to Ignore to make sure you get the maximum reduction in resulting code size. Or alternatively, if the developer knows the code being evaluated will only access local variables and functions in the current scope and nowhere else, the MakeImmediateSafe value can be specified and all parent scopes will still rename their locals. Very dangerous setting; should only be used when you are certain of all possible behavior of evaluated code.
* **IndentSize**: for the multi-line output feature, how many spaces to use when indenting a block (see OutputMode).
* **LocalRenaming**: renaming of locals. There are a couple settings: KeepAll is the default and doesn’t rename variables or functions at all. CrunchAll renames everything it can. In between there is KeepLocalizationVars, which renames everything it can except for variables starting with L\_. Those are left as-is so localization efforts can continue on the minified code.
* **MacSafariQuirks**: There are two quirks that Safari on the Mac (not the PC) needed: throw statements always seem to require a terminating semicolon; and if statements that only contains a single function declaration need to surround that function declaration with curly-braces. Basically, if you want your code to always work in Safari, set this to true. If you don’t care about Safari (for instance, in a corporate environment where the browser your users can use is highly restricted), setting this value to false might save a few bytes.
* **OutputMode**: SingleLine minifies everything to a single line. MultipleLines breaks the minified code into multiple lines for easier reading (won’t drive you insane trying to debug a single line). The only difference between the two outputs is whitespace. (see also: IndentSize).
* **RemoveUnneededCode**: Should be set to true for maximum minification. Removes unreferenced local functions (not global functions, though), unreferenced function parameters, quotes around object literal field names that won’t be confused with reserved words, and it does some interesting things with switch statements. For instance, if the default case is empty (just a break), it removes it altogether. If there is no default case, it also removes other empty case statements. It also removes break statements after return statements (unreachable code).
* **StripDebugStatements**: removes “debugger” statements, any calls into certain namespaces like $Debug, Debug, Web.Debug or Msn.Debug. also strips calls to the WAssert function.

If you wish to perform actions on the generated syntax tree directly, you can use the Microsoft.Ajax.Utilities.JSParser object directly. Create an instance of the object, passing in the source code you wish to parse. Calling the Parse method (passing in the same CodeSettings object) will return an abstract syntax tree node to the block representing your code.

// create the parser from the source string.

// pass null for the assumed globals array

JSParser parser = new JSParser( source, null );

string minified;

// hook the engine error event

parser.CompilerError += new CompilerErrorHandler(OnCompilerError);

try

{

// parse the input

Block scriptBlock = parser.Parse(settings);

if (scriptBlock != null)

{

// we'll return the minified code

minified = scriptBlock.ToCode();

}

}

catch(JScriptException e)

{

// other error handling

}

There are two analogous methods for minifying CSS:

public string MinifyStyleSheet(string source);

public string MinifyStyleSheet(string source, CssSettings settings);

The CssSettings object contains all the settings switches available for CSS minification. Its properties are:

* **CommentMode** – how to treat comments in the code. The default is CssComment.None, which will strip all comments except “important” comments. CssComment.All will leave all comments, and CssComment.Hacks will only leave certain known comment-based hacks.
* **Severity** – the maximum severity level to present as errors in the output. The default is zero, meaning only syntax-error generating errors. If the developer wishes to see more detail (possible errors, warnings, style suggestions), then the severity threshold can be raised. It is a numeric integer value.
* **TermSemicolons** – forces all rules to be terminated with semicolons if set to true (default is false).
* **ColorNames** – by default, W3C-strict color names will be used if they are shorter than the equivalent RGB values. This is the CssColor.Strict setting. If no color names are to be used, set this value to CssColor.Hex. If a set of colors recognized by all major browser is okay to use (W3C-strict validation is not required), set this value to CssColor.Major.
* **ExpandOutput** – Boolean value indicting whether to output all minified code on a single line or to break it into easy-to-read multiple lines. Default is false.
* **IndentSpaces** – if ExpandOutput is set to true, this property is the number of space characters to use for each indent. The default is 4.

In addition to the Minifier object, developers can use the CssParser object directly. To minify CSS source, create an instance of the CssParser, set the appropriate Settings property values, and call the Parse method:

CssParser parser = new CssParser();

parser.CssError += new EventHandler<CssErrorEventArgs>(OnCssError);

parser.FileContext = sourceFileName;

string crunchedStyles = parser.Parse(source);

# Error Ouput

The output of Microsoft Ajax Minifier conforms to the MSBuild/Visual Studio format for error and warning messages. In order for MSBuild to properly recognize and parse error messages in the stderr output stream, they must conform to this format:

*origin*: [*subcategory*] *category* *code*: *text*

The *origin* section of the message is a required, non-localized indicator of where the error occurred. If the error is with the Microsoft Ajax Minifier tool itself (bad parameter, etc), then the origin will simply be the name of the tool (“ajaxmin.exe”). If the error is within one of the input source files, the file name will be source, followed by the line and column information:

*path*(*line*,*columnstart*-*columnend*)

The *subcategory* is spec’d as an optional localized modifier of the category field. Typically this field is used to indicate the severity of the error thrown.

The *category* field is required, contains one of two possible values, and is not localizable. The value can be either “error” or “warning.”

The *code* is a required non-localized code to further indicate the error that has occurred. It cannot contain any spaces. Code values for Microsoft Ajax Minifier will always begin with the prefixes “AM,” “JS,” or “CSS.” For example, should Microsoft Ajax Minifier encounter any IOException when reading or writing any files, the code will be “AM-IO.” Errors within a JavaScript source file will contain the internal error number, for example, JS1016 indicates an unterminated comment error.

The *text* field is an optional, localized string giving a human-readable description of the error that has occurred. It’s technically optional, but Microsoft Ajax Minifier will always include some description in this field.

With Microsoft Ajax Minifier conforming to this output format, if Microsoft Ajax Minifier is integrated into your build system, errors and warnings generated by Microsoft Ajax Minifier will appear in the proper error and warning panes, and double-clicking on the errors will take you to the appropriate point within the source file.

Some examples of Microsoft Ajax Minifier error strings:

* bar.js(2,5-10): run-time error JS1010: Expected identifier: while
* a.js(3,4-9): coding error JS1206: Did you intend to write an assignment here: a = 4
* a.js(2,5-13): code warning JS1137: 'abstract' is a new reserved word and should not be used as an identifier: abstract
* foo.js(9,56-57): performance warning JS1135: Variable 'i' has not been declared: i
* j.js(7,1-5): improper technique warning JS1267: Always use full statement blocks: else
* ajaxmin.exe: error AM-USAGE: Invalid switch: -9