



A LALR(1) Parser and Lexical Analyzer
Generator for JavaScript written in
JavaScript

Version 0.26
User's Manual

February 18, 2008

Copyright © 2007, 2008 by Jan Max Meyer, J.M.K S.F. Software Technologies
<http://www.jmksf.com> jssc@jmksf.com



Table of Contents

1.Introducing JS/CC.....	3
Welcome to JS/CC!.....	3
The intention behind JS/CC.....	4
2.A quick start example.....	5
Installation and Start-Up.....	5
Implementing the example.....	7
Compiling the example.....	9
3.The Grammar Definition Language.....	10
General.....	10
The Terminal Declaration Part.....	10
Regular Expressions.....	11
Ambiguous Regular Expressions.....	12
Associativity and Precedence.....	12
Whitespaces.....	12
The Grammar Definition Part.....	12
Accessing right-hand side items in semantic action.....	13
Value types.....	14
Resolving conflicts.....	14
4.Implementing a scripting language.....	17
The idea.....	17
Defining the grammar.....	19
Executing the script.....	21
The complete program.....	21
5.Debugging parsers.....	29
General.....	29
driver.js.....	29
driver_rhino.js.....	30
driver_web.js.....	30
6.Internal documentation.....	31
General.....	31
Folder overview.....	31
Root folder.....	31
Source folder.....	32
Document folder.....	34
Samples folder.....	34
Enhancing JS/CC itself.....	35
Building JS/CC.....	35
Code style.....	35
Developer's documentation.....	36
License Agreements.....	36
Credits.....	38

1.Introducing JS/CC

Welcome to JS/CC!

JS/CC is the first available parser development system for JavaScript and other ECMAScript-derivates so far, and has been developed both with the intention of building a productive compiler development system and creating an easy-to-use learning environment for students and people interested in how parse table generation is done and how bottom-up parsing generally works.

JS/CC unions both: A regular expression-based lexical analyzer generator matching individual tokens from the input character stream and a LALR(1) parse table generator, computing the parse tables for a given context-free grammar specification. Such a context-free grammar is defined in a Backus-Naur-Form-based meta language, and allows the insertion of individual semantic code to be evaluated on a rule's reduction.

JS/CC itself has been entirely written in JavaScript, and is compatible with Microsoft's JScript and JScript.NET, so it can be executed using four different ways: As slightly limited platform-independent, browser-based JavaScript embedded on a Website (jscc.html), as Windows Script Host Application (jscc.js), as a compiled JScript.NET executable (jscc.exe), or even as interpreted Mozilla Rhino script on a console; The current JS/CC distribution provides all possibilities of execution; However, for productive execution, it is recommended to use the command-line versions of JS/CC, because only these versions are capable of assembling a complete compiler from a JS/CC parser specification, which is then stored into a .js-JavaScript source file.

To use JS/CC and for understanding its internals and behavior, certain knowledge on context-free grammars, bottom-up parsing techniques and compiler construction theories in general is assumed. To get a fundamental introduction to compiler design and the different techniques and appendages on this topic, I warmly suggest to the book *Compiler Design in C*, written by Allen I. Holub¹.

Maybe in future, the JS/CC user's manual itself (or I'll write a system-independent one) will provide a detailed introduction on the very huge topic of compiler construction, but for now, I limit this manual only on the usage of JS/CC itself, and its internals.

If you plan to add new features or write enhancements to JS/CC or its documentation, please read the information below in the particular section about JS/CC's internals.

Now, enjoy using JS/CC!

¹ Allen I. Holub, **Compiler Design in C** [Prentice-Hall Software Series], ISBN 978-0131550452

The intention behind JS/CC

JS/CC is an open source project released under the terms and conditions of the Artistic License.

When I started coding JS/CC, it was just planned as a quick-and-dirty fun project of implementing a LALR(1) parser generator in JavaScript for educational purposes, but it rapidly grew up because I added tons of more, useful features through the time, making it a really productive and useable system for compiler writers and people who are just interested in how a parser generator works. The best of all within this software is, that JS/CC is capable to compile a complete, working parser for any language from a grammar specification with embedded, semantic code segments only in a normal web-browser like Mozilla Firefox.

JS/CC was entirely written by me, Jan Max Meyer. I'm a software developer and hobbyist programmer from Germany and was born in 1985. The art of compiler design is a study on my own, because it very much interests me and makes a lot of fun. This project is one result of this study. I never studied on computer sciences yet, but plan this somewhere in the future.

In my professional life, I work as application software developer at a local business software company. As one of my hobbies, I also run my own, tiny software company – this is J.M.K S.F. Software Technologies – from home, where I'm in writing shareware programs and individual solutions around the area of software development tools, business software and web-solutions. JS/CC is a non-profit based product of this business.

But for now, I wish you much fun and success on using JS/CC. I hope that this software will be useful in your business or just helps you on understanding how bottom-up parsing works a little more clearly.

A thing I would also deeply appreciate is, if you have the fun and engagement on developing JS/CC further, adding new features or just make it more popular.

Best regards

Jan Max Meyer

PS: If you really like JS/CC and if you want to give me a smart pleasure for my efforts in developing this software, I would be very happy if you spend some money to the WDCCS – Whale and Dolphin Conservation Society – who fights against the senseless killing and capturing of marine animals, like whales and dolphins, which is unfortunately the case in some countries of the world today.

We must keep this world for us and our children, and stop this senseless killing of endangered animals. And in my opinion, every second counts!

To support the life of whales and dolphins in our oceans, please visit <http://www.wdcs.org>.

Thank you very much.

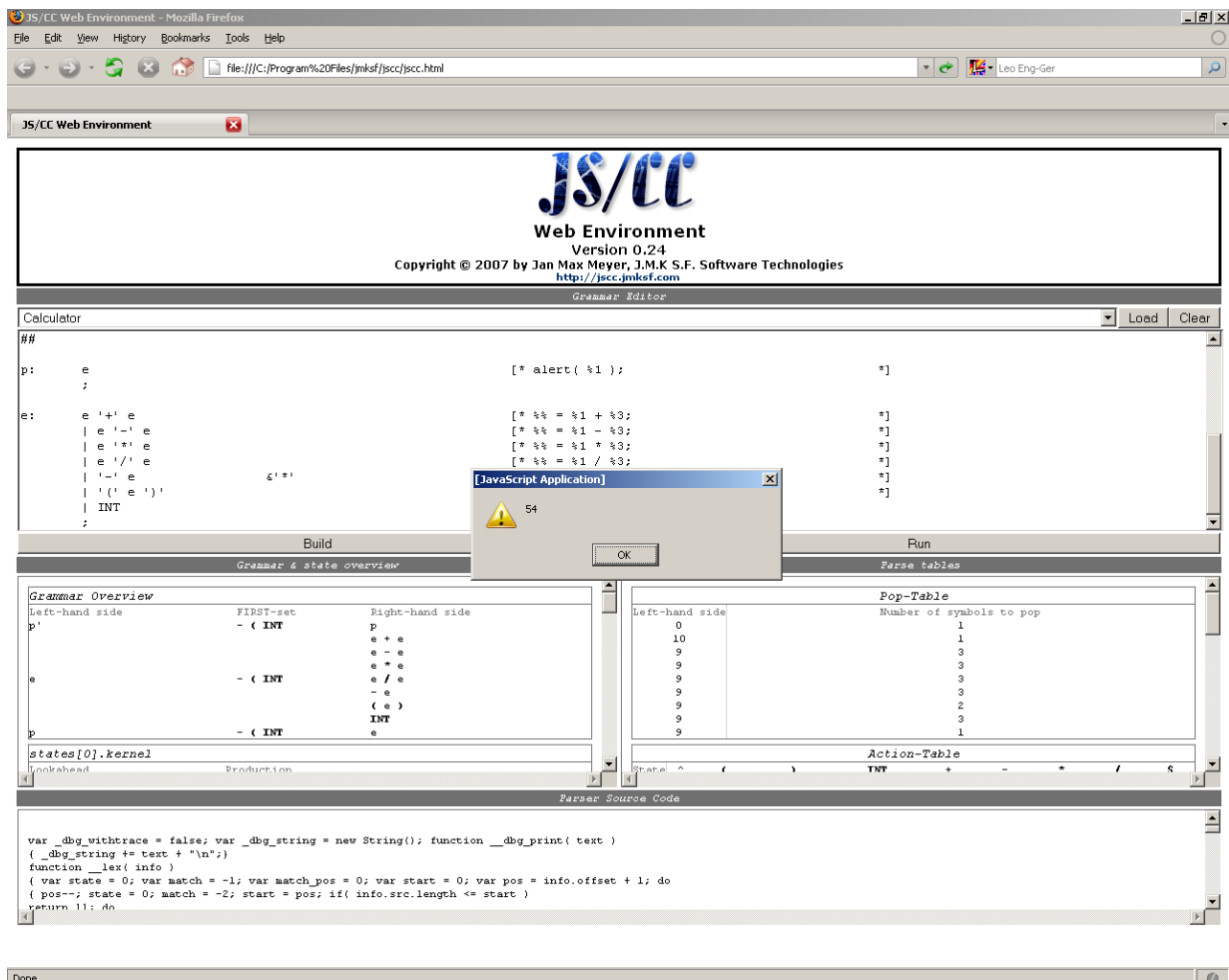
2.A quick start example

Installation and Start-Up

This section should be a quick start tutorial and how-to guide about how JS/CC is used.

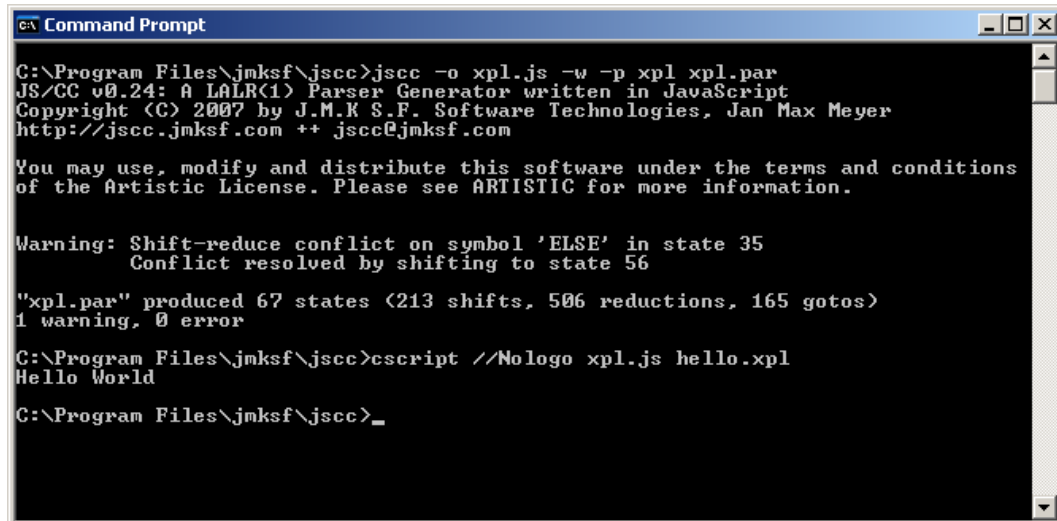
To install JS/CC, simply unzip the download package to any directory of your choice – Then the software is ready to run. The Windows Setup-Edition of JS/CC installs this folder structure to the directory you select, and adds some start menu entries.

As mentioned earlier, you can decide between three execution modes of JS/CC. For education and to get more familiar with JS/CC's grammar definition language, it is recommended to first use the JS/CC web environment, which is provided with the file *jssc.html*. The web-interface is like a mini-development environment for JS/CC, which lets you directly edit your grammar and enables a one-click table creation and a parser test facility. You can edit, build and run you complete parser using the web-interface. The samples-dropdown box provides some pre-defined example grammars which can directly be loaded and modified.



Screenshot 1: JS/CC executed using the web-based version within a browser window

If you would like to use the command-line versions of JS/CC, you can decide between *jscc.js*, which is a JScript to be executed as Windows Script Host application, and the stand-alone version, *jscc.exe*, which was compiled using the Microsoft JScript.NET compiler. For the Windows Script Host version, it is strongly recommended to use *cscript* as interpreter, not *wscript*; Using *wscript*, you would have to click lots of message boxes ;).



```
C:\Program Files\jmkfs\jscc>jscc -o xpl.js -w -p xpl xpl.par
JS/CC v0.24: A LALR(1) Parser Generator written in JavaScript
Copyright (C) 2007 by J.M.K S.F. Software Technologies, Jan Max Meyer
http://jscc.jmkfs.com ++ jscc@jmkfs.com

You may use, modify and distribute this software under the terms and conditions
of the Artistic License. Please see ARTISTIC for more information.

Warning: Shift-reduce conflict on symbol 'ELSE' in state 35
        Conflict resolved by shifting to state 56

"xpl.par" produced 67 states (213 shifts, 506 reductions, 165 gotos)
1 warning, 0 error

C:\Program Files\jmkfs\jscc>cscript //Nologo xpl.js hello.xpl
Hello World

C:\Program Files\jmkfs\jscc>_
```

Screenshot 2: JS/CC executed from the command-line

Implementing the example

In parser generators like yacc, a simple parser for mathematical expressions is the best example to become familiar with the syntax and usage, so we do this even here! The parser definition language itself is described below in a more detailed fashion.

Store the following grammar definition into a text file of your choice; For parser source files, the .PAR file extension could be used. The file is named "calc.par" in this case.

```
/~      Expression calculator written in JS/CC

      This is a WSH-script implementation; For JScript.NET or even JavaScript
      on the web, use the according functions to interact with the user.      ~/

/~      Tokens to be ignored (e.g. whitespaces, comments)      ~/
!      ' |\t'
;

/~      Grammar tokens      ~/
'\+'
'\-'
'\*'
 '/'
'\('
'\)'
'[0-9]+'      INT      [* %match = parseInt( %match ); *]
'[0-9]+\.[0-9]*|[0-9]*\.[0-9]+'      FLOAT      [* %match = parseFloat( %match ); *]
;

##

/~      The non-terminal "p" is the entry symbol, because it is the first one!      ~/
program:  expr      [* WScript.Echo( %1 ); *]
;

/~      Don't confuse with the tokens: Here, we use the unescaped values because these
      are not interpreted as regular expressions at this position!      ~/
expr:     expr '+' term      [* %% = %1 + %3; *]
      | expr '-' term      [* %% = %1 - %3; *]
      | term      /* Default semantic action fits here! */
;

term:     term '*' factor      [* %% = %1 * %3; *]
      | term '/' factor      [* %% = %1 / %3; *]
      | factor      /* Default semantic action fits here! */
;

factor:   '(' expr ')'      [* %% = %2; *]
      | INT      /* Default semantic action fits here! */
      | FLOAT      /* Default semantic action fits here! */
;

```

```

/~      This is the parser entry point; Because this entry point could be very individual,
        the compiler programmer has to decide which way he wants to read the source, parse
        it and report the errors, if there are any.                                ~/

[ *
var error_offsets = new Array();
var error_lookaheads = new Array();
var error_count = 0;

WScript.Echo( "Please enter an expression:" );
var str = new String( WScript.StdIn.ReadLine() );

/*
        The "##PREFIX##" is a wildcard, where an optional, unique name will be inserted by
        JS/CC when the parser is constructed. This enables the possibility to use multiple
        different parsers in one project or script.
*/
if( ( error_count = __##PREFIX##parse( str,
        error_offsets, error_lookaheads ) ) > 0 )
{
    for( i = 0; i < error_count; i++ )
        WScript.Echo( "Parse error near \""
                        + str.substr( error_offsets[i] ) +
                        "\", expecting \"" +
                        error_lookaheads[i].join() +
                        "\"" );
}
*]

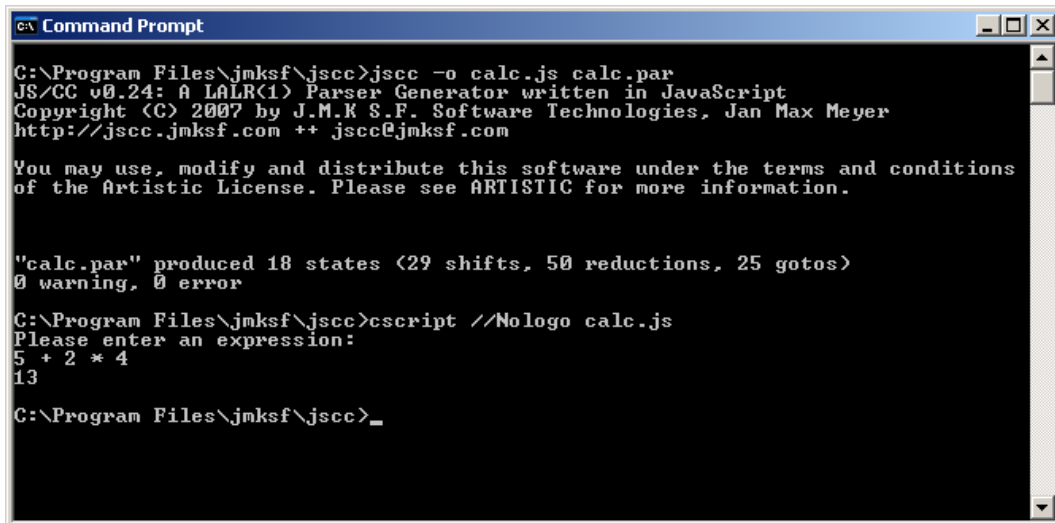
```


Compiling the example

To compile a working parser from the example described above, run JS/CC as follows from a Windows prompt:

```
jscc -o calc.js calc.par
```

That's all! The result, calc.js, can now simply be executed using cscript.



```
C:\Program Files\jmkfs\jscc>jscc -o calc.js calc.par
JS/CC v0.24: A LALR(1) Parser Generator written in JavaScript
Copyright (C) 2007 by J.M.K S.F. Software Technologies, Jan Max Meyer
http://jscc.jmkfs.com ++ jscc@jmkfs.com

You may use, modify and distribute this software under the terms and conditions
of the Artistic License. Please see ARTISTIC for more information.

"calc.par" produced 18 states (29 shifts, 50 reductions, 25 gotos)
0 warning, 0 error

C:\Program Files\jmkfs\jscc>cscript //Nologo calc.js
Please enter an expression:
5 + 2 * 4
13

C:\Program Files\jmkfs\jscc>_
```

Screenshot 3 - Compiling and executing the example

With a few changes on the grammar's semantic, a port to JavaScript for Web or JScript.NET (which allows to compile into a stand-alone executable) can also be simply performed.

3.The Grammar Definition Language

General

As you can get from the expression calculator example above, the grammar definition language of JS/CC consists of two parts: The token definition part, and the grammar part, which defines the language's grammar using a Backus–Naur–Form–styled meta language. The two parts are separated by a **##** symbol, which could be seen as a "separating line" between both parts.

The Terminal Declaration Part

Tokens, or terminal symbols, are defined in the first part of each grammar definition file. These terminals are defined using a regular expression meta language, but it's also possible to give each terminal symbol an individual name and a individual code segment to be executed when this token is recognized by the generated lexical analyzer (e.g. to cut the leading and trailing quotation marks if a string is recognized from the token's attribute).

The general syntax to define tokens is

regular-expression	label	code
.		
.		
.		
;		

where *label* and *code* is optional.

The regular expression is specified in the ways described below, using a single- or double-quoted string.

A token's label is defined as a single word identifier, e.g. "FLOAT" or "INTEGER_NUMBER". Not allowed are separated words. If no label is specified, JS/CC uses the regular-expression definition itself as label, but without taking escape-characters, so the regular expression '\+' will result in the label '+', as in the above example. If '+' itself is specified as regular expression, a parse error will occur because the plus-character is the symbol for a positive closure in regular expressions.

A semantic code action is defined by enclosing the desired JavaScript code segment with a [***** and *****] symbol. If more than one code segment is specified in a row, all segments are summarized to one segment to be attached to the terminal symbol. To simply access things like the matched pattern, the offset where the pattern starts or the source string in this individual code segments, the wildcards **%match**, **%offset** and **%source** should be used. These wildcards are later substituted by the particular variable names in the resulting lexical analyzer.

Because JS/CC does also allow to pass precedence and associativity information to tokens or token groups, each block of token definitions is closed by a semicolon (;). Because of that, the semicolon is set behind the last token definition also in the above example, even if we don't use any precedence information here.

Regular Expressions

Because it was impossible to build a properly working lexical analyzer using JavaScript's build-in RegExp-object, JS/CC features its own implementation of regular expression processing. This is also the reason that not all the features of the JavaScript RegExp-object are provided, like back-references and pre-defined character classes.

The symbols and operators to be used within JS/CC's own regular expression language are summarized in the following table.

They form a minimal implementation of a regular expression engine.

<i>Language Element</i>	<i>Description</i>
Character	One character specifies exactly that character; If a regular-expression operator like + or * should be used, it must be escaped via \.
\ascii-code	One character, defined via ASCII-code, e.g. "\220" matches the Ü-umlaut of the extended ASCII-table.
\character	Escaped character. Must be used when a character of the meta-language itself should be matched, e.g. "\\ ".
.	Any character (character class matching all available characters).
[...]	Character-class; If a beginning circumflex (^) is given, the character-class is negated. Character ranges can be specified using a dash. For example "[A-Za-z]" specifies all capital and lower-case alphabet letters.
(...)	Sub-expression.
	Or-operator; Allows to specify different expressions at one level.
*	Kleene-closure operator (none or many), to be specified behind a character, character-class or sub-expression.
+	Positive-closure operator (one or many), to be specified behind a character, character-class or sub-expression.
?	Optional-closure operator (one or none), to be specified behind a character, character-class or sub-expression.

Table 1: JS/CC own regular expression meta language

Even to allow case-insensitive keywords within grammar definitions, a terminal symbol definition can be specified using single-quoted ('...') and double-quoted ("...") strings. A single-quoted string means that a terminal symbol is matched case-sensitive, a double-quoted one matches a terminal in any case order. For example, the terminal symbol definition "PRINT" will match for *Print*, *print*, *PrINT* and *PRINT*, while the definition 'PRINT' will only match for *PRINT* itself.

From these regular expression definitions, JS/CC constructs a deterministic finite automata which acts as lexer in the resulting parser.

Ambiguous Regular Expressions

If there are ambiguous regular expressions (where several expressions match the same string) within the terminal definition part, the expressions defined upper in the terminal definition part will take higher match precedence than the lower defined terminals.. It is recommended to define tokens with a higher specialization level as the first, and tokens with a lower level as the last in your token definition part.

Associativity and Precedence

Tokens can be grouped by precedence levels and associativity. This features allows to write faster and even smaller grammars, by resolving grammar conflicts by weighting terminal symbols.

A group without a group specifier will set no associativity and a precedence level of zero to all terminal symbols in this group (as in the first example).

Else, if a group begins with the symbol < for left-associativity, > for right-associativity and ^ for non-associativity, all terminal symbols within this group are set to the according associativity and precedence level. The precedence level is incremented each time a new group of these three types is opened, so groups that are defined at the bottom of the token definition part take the highest precedence.

The precedence information as associativity is used to resolve conflicts in ambiguous grammars by modifying the parse table's natural content; How this works in practice and what ways and possibilities there are is descibed below in the section dialing with grammar conflicts and their handling.

Whitespaces

A special type of terminal symbol is introduced by the exclamtion-mark (!) symbol: The whitespace symbols!

In this definition, there is only a regular expression possible; A label or code part is prohibited. As whitespace-tokens, terminals that should always be ignored can be specified, e.g. blanks, tabs or comments.

The Grammar Definition Part

The second – and most important – part of a JS/CC parser definition is the grammar definition part. In this part, below the **##** symbol, the definition of the context-free grammar to be parsed by the generated parser is described. This is done by using a Backus-Naur-Form variant meta

language, by defining non-terminals and their productions.

The general syntax for a non-terminal and its productions is

```
non-terminal      : production1 semantic-code
                  | production2 semantic-code
                  .
                  .
                  .
                  | productionn semantic-code
                  ;
```

The *non-terminal* defines a single-word identifier and acts as the left-hand side for the related productions attached to this symbol.

The *production* defines a sequence of zero or multiple terminals and non-terminals, defining the different syntax rules. To specify terminal symbols, it is possible to call them via their (unescaped) regular expression (as described above!) via a string-value or by their label, which must not (but can) be specified as a string value. In the example of the quick start part, both methods are used: FLOAT is called by its label, the token '+', for better readability, is called by its generated name which came from the regular expression.

The semantic-code part behind the productions is optional, and defines an individual semantic code which is executed right before the according production is reduced to its left hand side. Same as in the token definition part, semantic code is enclosed by [* and *] and will be catenated to one code segment which is associated with the according productions when multiple semantic code segments are specified in a row. Read more about this in the section below.

Note that in the above syntax scheme the number of productions is completely variable. At least, one right-hand side must be given to an according left-hand side, although this can be an epsilon production.

Each production is separated by a vertical "pipe" bar (|) from the others, and a non-terminal definition must always be closed by a semicolon. Else, JS/CC can not distinguish if the next symbol belongs to the right-hand side it currently parses or if it reads a new non-terminal definition.

Non-terminal symbols can be called on a right-hand side before they are defined, so the way the rules are defined is arbitrary. Integrative checks on the grammar are done by JS/CC before the parse tables will be constructed.

Accessing right-hand side items in semantic action

Within each semantic action attached to a production, as described above, the values of the right-hand side symbols can be accessed via wildcards, which are replaced by the particular variables and offsets later in the resulting parser.

The %n wildcards are used to address every individual token starting from the left of the right-hand side.

The %% wildcard relates to the value of the left-hand side which is pushed to the parser's value stack right after the right-hand side symbols are popped.

So by passing a value to the %% wildcard causes the inheritance of values from the current right-hand side to another call of the according non-terminal on a right-hand side within the parse tree. The values on the current right-hand side will be discarded when the reduction process occurs, and the value attached to %% is pushed instead, so it can be used elsewhere.

As example, in the given production

```
expr:      expr '+' term      [ * %% = %1 + %3; * ]
```

of the above expression calculator, the return values of the *expr* and the *term* on the right-hand side, which are addressed via %1 and %3 (%2 addresses the value of the '+'-terminal!) are added together, and the result is stored to the left hand-side (the *expr* on the far left in this case). Thus, you have full control over all individual tokens within each production.

If no individual semantic action is given to a right-hand side, the default action

```
%% = %1;
```

is used.

Semantic action between the symbols of a right-hand side is not allowed, only behind them.

Value types

Because JS/CC was designed for the use with JavaScript, or any other typeless scripting language, it is not necessary to define – as in yacc – a special union structure to hold the values on the value stack. Both build-in primary types like String or Number objects as well as user-defined objects [each function in JavaScript is internally represented by an object] can be pushed to and popped off the value stack.

So don't confuse with the values; You have to know what value stands on which position.

Resolving conflicts

To automatically resolve shift-reduce or reduce-reduce conflicts at parse table generation, JS/CC features, by default, two mechanisms.

When a shift-reduce conflict occurs, JS/CC constructs the parse tables in favor of the shift, so the parse tree will grow right derivative.

When a reduce-reduce conflict occurs, JS/CC resolves the problem by reducing the production which was defined first, so productions which were defined above in the grammar will be reduced in favor when this conflict comes up.

As shortly described in the chapter about terminal symbol definition, JS/CC features the possibility of manipulating the natural parse table generated by the LALR(1) table construction algorithm by weighting terminal symbols with a precedence level and an associativity. This information is used within shift-reduce conflicts to better decide if a shift or a reduce operation should be inserted. If a shift-reduce conflict comes up, the precedence level and associativity information is compared with the according production's precedence level, because every production will, by default, get the same precedence level as its rightmost terminal symbol.

To explain this behavior, let's look at the following example. It defines a calculator same as the first example, but with lesser effort in writing the grammar; Here, we have only two non-terminal

symbols instead of four, but we implement the same operator precedence behavior as in the original one.

```

/~ Tokens to be ignored (e.g. whitespaces, comments) ~/
!      ' |\t';

/~      Left-associative tokens, lowest precedence      ~/
<      '\+'
      '\-';

/~      Left-associative tokens, highest precedence     ~/
<      '\*'
      '/';

/~      Non-associative tokens                          ~/
      '\('
      '\)'
      '[0-9]+'          INT          [* %match = parseInt( %match );   *]
      '[0-9]+\.[0-9]*|[0-9]*\.[0-9]+'  FLOAT      [* %match = parseFloat( %match ); *]
      ;

##

program:  expr          [* WScript.Echo( %1 );   *]
        ;

expr:     expr '+' expr  [* %% = %1 + %3; *]
        | expr '-' expr  [* %% = %1 - %3; *]
        | expr '*' expr  [* %% = %1 * %3; *]
        | expr '/' expr  [* %% = %1 / %3; *]
        | '(' expr ')'   [* %% = %2; *]
        | INT
        | FLOAT
        ;

```

Sometimes, it will also be necessary to give a production another precedence level than the one of the rightmost terminal. For example, if we want to add an unary minus operator to the grammar above, the production adopts the precedence level of the minus-symbol by default. But this minus-operator was configured for its use in a binary subtraction, not in an unary subtraction. By simply adding a new rule for unary minus to the grammar, most simple expressions will return the right result (e.g. "-2+3"), but in expressions like "4/-4*5", the result will simply be wrong, because, through our precedence rules for multiplication, the generated parser parses "4/(-(4*5))" instead of "(4/(-4))*5".

To resolve this problem, we need to attach a higher precedence level to the production for unary minus. For this special case, JS/CC features the **&**-directive. The **&**-directive must be specified behind the rule's definition and in front of the semantic code action (if there is any). Behind the **&**-directive, a terminal symbol (both as string or its label) is specified, which precedence level is taken by the production instead of its default value.

So by changing the grammar to

```
expr:      expr '+' expr          [* %% = %1 + %3; *]
          | expr '-' expr        [* %% = %1 - %3; *]
          | expr '*' expr        [* %% = %1 * %3; *]
          | expr '/' expr        [* %% = %1 / %3; *]
          | '(' expr ')'         [* %% = %2; *]
          | '-' expr             &'*'  [* %% = %2 * -1; *]
          | INT
          | FLOAT
          ;
```

we get the right parse tree and result, because our rule with the unary minus has a higher precedence now and reduces instead of shifting in the desired cases.

4.Implementing a scripting language

The idea

As a real, practical example, we will now write a simple, interpreted programming language, called XPL (eXample Programming Language) using JS/CC. XPL is a C-styled script language interpreter, providing simple user input/output operations, loops and conditional execution as well as variables. As variable type, only numbers are supported (integer and floating point, all use the same, internal JavaScript type "Number"). The Definition of user-defined functions is not possible in this scripting language, but it would be a thing of simplicity to add this feature.

For text output, XPL provides a special say-command which allows to output constant texts.

To get familiar with XPL's syntax, look at the following example scripts.

hello.xpl:

```
//This is a simple Hello World script, written in XPL.
say 'Hello World';
```

99-bottles-of-beer.xpl:

```
//The wonderful "99 bottles of beer"-program
bottles = 99;
do
{
    //The output will not be the prettiest, but that is limited
    //by the implementation (you can change it if you want ;)
    write bottles;

    if bottles == 1 say 'bottle of beer on the wall,';
    else say 'bottles of beer on the wall,';

    write bottles;
    if bottles == 1
        say 'bottle of beer';
    else
        say 'bottles of beer';

    say 'Take one down, pass it around,';
    bottles = bottles - 1;

    write bottles;
    if bottles == 0 say 'no more bottles of beer on the wall';
    else if bottles == 1 say 'bottle of beer on the wall';
    else say 'bottles of beer on the wall';

    say '';    //Empty line
}
while bottles > 0;

say 'That''s it!';
```


countdown.xpl:

```
//A rocketry launch countdown ;)
say '--- The final countdown program ---';

do
{
    say 'Enter your starting number (it must be greater or equal 10!):';
    read count;

    if count < 10 say 'The number is lower 10!';
}
while count < 10;

say 'Starting sequence...';
while count >= 0 do
{
    write count;

    //Ignition at 3 loops before lift-off...
    if count == 3 say 'Ignition...';
    else if count == 0 say '...and lift-off!';
    count = count - 1;
}
```

Defining the grammar

The unaugmented grammar definition for the XPL-grammar is the following, and consists of less than 80 lines of grammar and terminal definition code.

```
!      ' |\r|\n|\t|//[^\n]*\n'

      "IF"
      "ELSE"
      "WHILE"
      "DO"
      "SAY"
      "WRITE"
      "READ"
      '{ '
      '}'
      ';'
      '\('
      '\)'
      '='

      '[A-Za-z_][A-Za-z0-9_]*'      Identifier
      '\\'([^\\']|\\'\\')*\\''      String
      '[0-9]+'                      Integer
      '[0-9]+\\. [0-9]*|[0-9]*\\. [0-9]+' Float
      ;

>      '=='
```

```

        '!='
        '<='
        '>='
        '>'
        '<'
        ;

<        '\+'
        '\-'
        ;

<        '/'
        '\*'
        ;

##

Program:          Program Stmt
                  |
                  ;

Stmt_List:        Stmt_List Stmt
                  |
                  ;

Stmt:             IF Expression Stmt
                  | IF Expression Stmt ELSE Stmt
                  | WHILE Expression DO Stmt
                  | DO Stmt WHILE Expression ';'
                  | SAY String ';'
                  | WRITE Expression ';'
                  | READ Identifier ';'
                  | Identifier '=' Expression ';'
                  | '{' Stmt_List '}'
                  | ';'
                  ;

Expression:       Expression '==' Expression
                  | Expression '<' Expression
                  | Expression '>' Expression
                  | Expression '<=' Expression
                  | Expression '>=' Expression
                  | Expression '!=' Expression
                  | Expression '-' Expression
                  | Expression '+' Expression
                  | Expression '*' Expression
                  | Expression '/' Expression
                  | '-' Expression                &'*'
                  | '(' Expression ')'
                  | Integer
                  | Float

```

```
| Identifier  
;
```

Executing the script

It is impossible to interpret an XPL script directly like in the first expression calculator example. It probably works also here, but only in expressions. Statements like **while...do** or **if...else** are not directly interpretable, because they have a body and a conditional part. Because of that, we need to compile the program internally into a structure for a virtual machine. This virtual machine uses an assembly like command language, working with opcodes and operands.

The parser has the purpose to build-up a cascading structure of nodes, which form the program that can be executed. This method avoids the use of dealing with jumps to addresses, but cannot be used to be written to a file.

Note that this compiler will not compile into an assembly code file. The assembly program is constructed directly into the memory as an internal structure. The virtual machine executing this structure will at least exist of one function, which calls itself recursively based on the operation it should execute.

The complete program

This is the complete, augmented program code for the XPL compiler and interpreter, based on the grammar definition from above. A script is executed from a file, which must be attached to the WSH script at startup.

```
/~  
  
    XPL - eXample Programming Language v0.3  
    Written 2007 by J.M.K S.F. Software Technologies, Jan Max Meyer  
  
    The complete source of this program is in the Public Domain.  
  
    This example demonstrates the implementation of XPL, a complete,  
    interpreted scripting language, written in JS/CC.  
  
    XPL provides simple input/output operations and can only handle  
    numeric values.  
  
    Watch out for the *.xpl-files within the example directory, which  
    contain example scripts to be executed using XPL.  
~/  
  
[*  
  
//Structs  
function NODE()  
{  
    var type;  
    var value;  
    var children;
```

```

}

//Defines
var NODE_OP          = 0;
var NODE_VAR         = 1;
var NODE_CONST       = 2;

var OP_NONE          = -1;
var OP_ASSIGN        = 0;
var OP_IF            = 1;
var OP_IF_ELSE       = 2;
var OP_WHILE_DO      = 3;
var OP_DO_WHILE      = 4;
var OP_WRITE         = 5;
var OP_READ          = 6;
var OP_SAY           = 7;

var OP_EQU           = 10;
var OP_NEQ           = 11;
var OP_GRT           = 12;
var OP_LOT           = 13;
var OP_GRE           = 14;
var OP_LOE           = 15;
var OP_ADD           = 16;
var OP_SUB           = 17;
var OP_DIV           = 18;
var OP_MUL           = 19;
var OP_NEG           = 20;

//Management functions
function createNode( type, value, childs )
{
    var n = new NODE();
    n.type = type;
    n.value = value;
    n.children = new Array();

    for( var i = 2; i < arguments.length; i++ )
        n.children.push( arguments[i] );

    return n;
}

//Array to store variable names and values to
var v_names = new Array();
var v_values = new Array();

//Function to store a variable's content to a variables name. If the name does
//not exist already, the variable is automatically created.
function letvar( vname, value )
{

```

```

        var i;
        for( i = 0; i < v_names.length; i++ )
            if( v_names[i].toString() == vname.toString() )
                break;

        if( i == v_names.length )
        {
            v_names.push( vname );
            v_values.push( 0 );
        }

        v_values[i] = value;
    }

//Function to get a variable's content over its name
function getvar( vname )
{
    var i;
    for( i = 0; i < v_names.length; i++ )
        if( v_names[i].toString() == vname.toString() )
            return v_values[i];

    return 0;
}

//This is the interpreting function, working on base of the compiled program structure.
function execute( node )
{
    var ret = 0;

    if( !node )
        return 0;

    switch( node.type )
    {
        case NODE_OP:
            switch( node.value )
            {
                case OP_NONE:
                    /* OP_NONE can have childs (a block!) */
                    if( node.children[0] )
                        execute( node.children[0] );
                    if( node.children[1] )
                        ret = execute(
                            node.children[1] );
                    break;
                case OP_ASSIGN:
                    letvar( node.children[0], execute(
                        node.children[1] ) );
                    break;
                case OP_IF:

```

```

        if( execute( node.children[0] ) )
            execute( node.children[1] );

        break;
    case OP_IF_ELSE:
        if( execute( node.children[0] ) )
            execute( node.children[1] );

        else
            execute( node.children[2] );

        break;
    case OP_WHILE_DO:
        while( execute( node.children[0] ) )
            execute( node.children[1] );

        break;
    case OP_DO_WHILE:
        do
            execute( node.children[0] )

        while( execute( node.children[1] ) );

        break;
    case OP_WRITE:
        WScript.Echo( execute( node.children[0] )
            );

        break;
    case OP_READ:
        letvar( node.children[0].toString(),
            WScript.StdIn.ReadLine() );

        break;
    case OP_SAY:
        WScript.Echo( node.children[0] );

        break;
    case OP_EQU:
        ret = execute( node.children[0] ) ==
            execute( node.children[1] );

        break;
    case OP_NEQ:
        ret = execute( node.children[0] ) !=
            execute( node.children[1] );

        break;
    case OP_GRT:
        ret = execute( node.children[0] ) >
            execute( node.children[1] );

        break;
    case OP_LOT:
        ret = execute( node.children[0] ) <
            execute( node.children[1] );

        break;
    case OP_GRE:
        ret = execute( node.children[0] ) >=
            execute( node.children[1] );

        break;
    case OP_LOE:
        ret = execute( node.children[0] ) <=

```



```

                                execute( node.children[1] );
                                break;
                                case OP_ADD:
                                    ret = execute( node.children[0] ) +
                                        execute( node.children[1] );
                                    break;
                                case OP_SUB:
                                    ret = execute( node.children[0] ) -
                                        execute( node.children[1] );
                                    break;
                                case OP_DIV:
                                    ret = execute( node.children[0] ) /
                                        execute( node.children[1] );
                                    break;
                                case OP_MUL:
                                    ret = execute( node.children[0] ) *
                                        execute( node.children[1] );
                                    break;
                                case OP_NEG:
                                    ret = execute( node.children[0] ) * -1;
                                    break;
                                }
                                break;

                                case NODE_VAR:
                                    ret = Number( getvar( node.value ) );
                                    break;

                                case NODE_CONST:
                                    ret = Number( node.value );
                                    break;
                                }
                                return ret;
}

*]

/~ Defining whitespaces and comments ~/
!      ' |\r|\n|\t|//[^\n]*\n'

/~ Keywords (case-insensitive!) and program structure operators ~/
"IF"
"ELSE"
"WHILE"
"DO"
"SAY"
"WRITE"
"READ"
'{'
'}'
';'

```

```

'\('
'\)'
'='
'[A-Za-z_][A-Za-z0-9_]*'      Identifier
'\'([^\']|\\\' )*\''          String      [* %match = %match.substr( 1,
                                   %match.length - 2 );
                                   %match = %match.replace( /'/g,
                                   "\'" );
                                   *]

'[0-9]+'                      Integer
'[0-9]+\.[0-9]*|[0-9]*\.[0-9]+' Float
;

/~ Operators to be used in expressions ~/
>      '=='
      '!='
      '<='
      '>='
      '>'
      '<'
      ;

<      '\+'
      '\-'
      ;

<      '/'
      '\*'
      ;

##

Program:      Program Stmt      [* execute( %2 ); *]
      |
      ;

Stmt_List:    Stmt_List Stmt      [* %% = createNode( NODE_OP,
                                   OP_NONE, %1, %2 ); *]
      |
      ;

Stmt:         IF Expression Stmt      [* %% = createNode( NODE_OP,
                                   OP_IF, %2, %3 ); *]
      | IF Expression Stmt ELSE Stmt      [* %% = createNode( NODE_OP,
                                   OP_IF_ELSE, %2, %3, %5 ); *]
      | WHILE Expression DO Stmt      [* %% = createNode( NODE_OP,
                                   OP_WHILE_DO, %2, %4 ); *]
      | DO Stmt WHILE Expression ';'      [* %% = createNode( NODE_OP, OP_DO_WHILE,
                                   %2, %4 ); *]
      | SAY String ';'      [* %% = createNode( NODE_OP, OP_SAY,
                                   %2 ); *]

```

```

| WRITE Expression ';'          [* %% = createNode( NODE_OP, OP_WRITE,
                                %2 ); *]

| READ Identifier ';'           [* %% = createNode( NODE_OP, OP_READ,
                                %2 ); *]

| Identifier '=' Expression ';' [* %% = createNode( NODE_OP, OP_ASSIGN,
                                %1, %3 ); *]

| '{' Stmt_List '}'           [* %% = %2; *]
| ';'                          [* %% = createNode( NODE_OP, OP_NONE );
                                *]

;

Expression: Expression '==' Expression      [* %% = createNode( NODE_OP, OP_EQU,
                                %1, %3 ); *]

| Expression '<' Expression          [* %% = createNode( NODE_OP, OP_LOT,
                                %1, %3 ); *]

| Expression '>' Expression          [* %% = createNode( NODE_OP, OP_GRT,
                                %1, %3 ); *]

| Expression '<=' Expression        [* %% = createNode( NODE_OP, OP_LOE,
                                %1, %3 ); *]

| Expression '>=' Expression        [* %% = createNode( NODE_OP, OP_GRE,
                                %1, %3 ); *]

| Expression '!=' Expression      [* %% = createNode( NODE_OP, OP_NEQ,
                                %1, %3 ); *]

| Expression '-' Expression       [* %% = createNode( NODE_OP, OP_SUB,
                                %1, %3 ); *]

| Expression '+' Expression       [* %% = createNode( NODE_OP, OP_ADD,
                                %1, %3 ); *]

| Expression '*' Expression       [* %% = createNode( NODE_OP, OP_MUL,
                                %1, %3 ); *]

| Expression '/' Expression       [* %% = createNode( NODE_OP, OP_DIV,
                                %1, %3 ); *]

| '-' Expression &'*'            [* %% = createNode( NODE_OP, OP_NEG,
                                %2 ); *]

| '(' Expression ')'             [* %% = %2; *]
| Integer                       [* %% = createNode( NODE_CONST, %1 ); *]
| Float                         [* %% = createNode( NODE_CONST, %1 ); *]
| Identifier                    [* %% = createNode( NODE_VAR, %1 ); *]

;

[*
//Utility function: Open and read a file
function open_file( file )
{
    var fs = new ActiveXObject( 'Scripting.FileSystemObject' );
    var src = new String();

    if( fs && fs.fileExists( file ) )
    {

```

```

        var f = fs.OpenTextFile( file, 1 );
        if( f )
        {
            src = f.ReadAll();
            f.Close();
        }
    }

    return src;
}

//Main
if( WScript.Arguments.length > 0 )
{
    var str                = open_file( WScript.Arguments(0) );
    var error_cnt          = 0;
    var error_off          = new Array();
    var error_la           = new Array();

    if( ( error_cnt = __##PREFIX##parse( str, error_off, error_la ) ) > 0 )
    {
        for( i = 0; i < error_cnt; i++ )
            WScript.Echo( "Parse error near \""
                + str.substr( error_off[i], 10 ) +
                ( ( str.length > error_off[i] + 10 ) ? "...\" : "\"" ) +
                "\", expecting \"" + error_la[i].join() + "\"" );
    }
}
else
    WScript.Echo( 'usage: xpl.js <filename>' );
*]

```

5.Debugging parsers

General

When JS/CC compiles a grammar, the parse tables are inserted into a so called driver template which then executes the parser based on the content of the parse tables.

This parser drivers offers several methods to debug grammars.

JS/CC comes with two parser driver templates. They are defined in the files `driver.js_` and `driver_web.js_`, where the first one is for console-based parsers compiled for JScript.NET and JScript for Windows Script Host, and the latter one is a platform independent, web-based parser to be executed in a web browser.

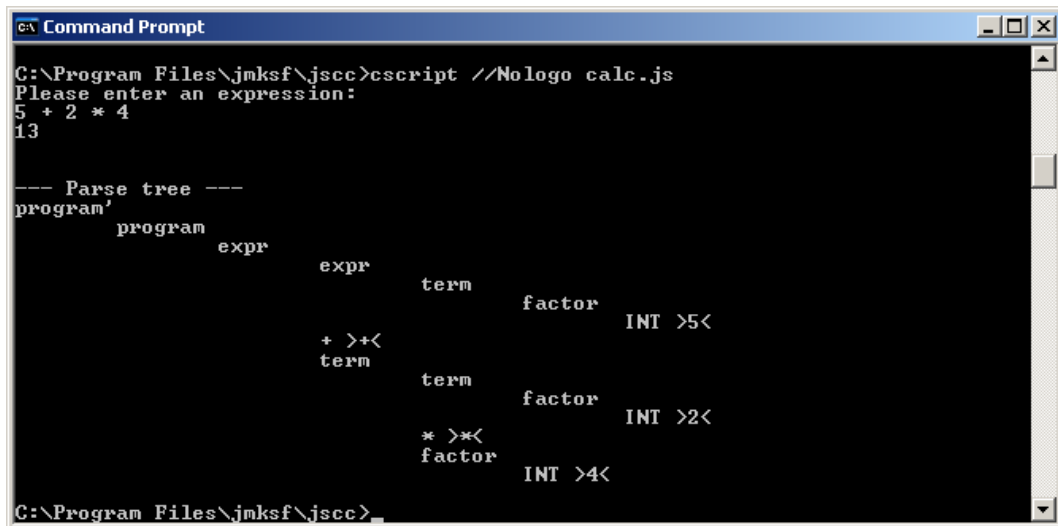
The grammar debugging features of the respective parse table driver are enabled by setting some global variables to true.

driver.js_

The `driver.js_` parser driver for JScript (both JScript.NET and JScript on Windows Script Host) provides the most debugging features. It is capable to trace the parse process – even step by step – and to print a pseudo-graphical parse tree as an ASCII-based image to the console.

The provided debugging features are:

- Trace (`##PREFIX##_dbg_withtrace`)
The parse trace provides information about each parser state which is entered with which input tokens, parse stack and value stack. This feature is very useful to understand the parse process and to detect possible value access errors.
- Step-by-step (`##PREFIX##_dbg_withstepbystep`)
This can only be used in conjunction with the parse trace, and waits that the user presses enter to continue to the next state. Has no effect when trace is switched off.
- Parse tree generation (`##PREFIX##_dbg_withparsetree`)
Generates a parse tree at the end of the parse. This parse tree, which is printed in a pseudo-graphical tree view, represents the structure of the complete parsed input splitted into all the used productions and tokens at the leafes of the tree.



```
C:\Program Files\jmkfs\jsc>escript //Nologo calc.js
Please enter an expression:
5 + 2 * 4
13

--- Parse tree ---
program'
  program
    expr
      expr
        term
          factor
            INT >5<
        + >+<
        term
          term
            factor
              INT >2<
            * >*<
            factor
              INT >4<
```

Screenshot 4: The parse tree visualization feature of the *driver.js_parser*

To switch the debugging features on, set the variables in the brackets to true before invoking parser function. The `##PREFIX##` will be replaced by the optionally defined prefix you set when calling JS/CC.

driver_rhino.js_

This driver features the same possibilities like *driver.js_*, but can be used for parsers running on the the Mozilla Rhino platform.

driver_web.js_

This is the web-based version of the JS/CC default driver, based on *driver.js*. It should be used for browser-based JavaScript applications, generated to a HTML-file. It provides even the parse trace and step-by-step debugging features, but parse tree generation is not provided here.

6.Internal documentation

General

After all, the JS/CC parser development system consists of two parts: The parser generator, which analyzes a grammar, computes the parse table, and builds the code, and the parser driver, which works on the generated parse tables and runs the parser.

The parser driver is used by the parser generator as a template, where the parse tables and some arbitrary values are inserted to special, pre-defined insertion points (the so called "wildcards").

The result of this process is the individual parser, which can be executed.

Note that both parts of the JS/CC system have different copyright restrictions: The parser generator is a copyrighted open source program, for more information, see the Artistic License provided with the program package. The parser driver (template) instead is in the public domain – there is no copyright on it, which enables the use of JS/CC generated parsers both for commercial (even closed-source) and non-commercial applications. This also includes that changes of any kind are allowed on the parser driver, but note, that changing the parser driver without any knowledge on it can make newly generated parsers work wrong.

This internal documentation should be a short introduction for people who are interested in both coding or to simply get more familiar with the parser generator and the parser driver.

Note that with the name JS/CC, the parser generator itself is meant in the most cases.

Folder overview

This chapter gives a short overview about the directories of the JS/CC software package.

Root folder

The root-folder contains the binaries and executables of JS/CC.

<i>File</i>	<i>Usage/Description</i>
jssc.exe	JS/CC parser generator as JScript.NET executable; Recommended for productive usage.
jssc.html	JS/CC parser generator web interface for interactive parser generation and debugging.
jsccr.js	JS/CC parser generator implementation to be executed on the Mozilla/Rhino ECMAScript interpreter.
driver.js_	JS/CC parser driver template to be used with WSH and JScript.NET programs. It provides powerful debugging possibilities and is used for productive parser generation.

<i>File</i>	<i>Usage/Description</i>
driver_web.js_	JS/CC web parser driver to be used for web-based applications. It features lesser debug possibilities than driver.js_ and cannot be executing with WSH or Jscript.NET.
driver_rhino.js_	JS/CC parser driver for parsers to be executed on Mozilla/Rhino.
ARTISTIC	The license agreement of JS/CC (Artistic License).
CHANGES	Log about the changes and bugfixes in the latest version.
README.RHINO	Some additional information for the JS/CC Rhino port and how it is used.
jssc_logo_small.png	JS/CC Logo used by <i>jssc.html</i> .
jssc_logo.png	A huger version of the JS/CC Logo.

Table 2: root directory overview

Source folder

The source folder (**/src**) contains the whole source code of the JS/CC parser generator, and some additional files which are necessary to build and develop JS/CC.

<i>File</i>	<i>Usage/Description</i>	<i>Used by</i>
jssc.js	JS/CC parser generator entry source; This is the general entry point for all JScript versions of JS/CC.	JScript.NET WSH JScript
jssc_RHINO.js	The JS/CC program entry for the Mozilla/Rhino port version.	Rhino
jssc_global.js	Global declarations and variables. This file contains only declarations, switches and other defineable values used within the whole process of parse table generation.	JScript.NET WSH JScript Web Rhino
jssc_util.js	Utility functions used by several modules.	JScript.NET WSH JScript Web Rhino
jssc_debug.js	Debug and dump functions for JS/CC	JScript.NET

<i>File</i>	<i>Usage/Description</i>	<i>Used by</i>
	debugging.	WSH JScript Web Rhino
jscc_parse.par jscc_parse.js jscc_PARSE_WEB.js jscc_parse_RHINO.js jscc_parse_RHINO_BOOTSTRAP.js	<p>JS/CC is a program which is build by using itself as development tool.</p> <p>JS/CC's own grammar parser has been entirely rewritten in JS/CC, and generates jscc_parse.js during the make process. Changes should only be done to jscc_parse.par.</p> <p>In the early time of JS/CC, a hand-written recursive descent parser was used to parse JS/CC grammars.</p> <p>Be careful when changing this file!</p> <p>(The grayed files are files which are generated by JS/CC itself during the make-process!)</p>	JScript.NET WSH JScript Web Rhino
jscc_integrity.js	Grammar integrity check functions.	JScript.NET WSH JScript Web Rhino
jscc_first.js	FIRST-set computation for parse JS/CC grammars.	JScript.NET WSH JScript Web Rhino
jscc_tabgen.js	Implementation of the LALR(1) parse table generation algorithm.	JScript.NET WSH JScript Web Rhino
jscc_printtab.js	Generation of JavaScript code for the parse and lexing tables.	JScript.NET WSH JScript Web Rhino
jscc_regex.par	Same to JS/CC's internal regular expression	JScript.NET

<i>File</i>	<i>Usage/Description</i>	<i>Used by</i>
jscc_regex.js jscc_REGEX_WEB.js jscc_regex_RHINO.js jscc_regex_RHINO_BOOTSTRAP.js	<p>parser; This has also been rewritten entirely in JS/CC itself. This parser builds a nondeterministic finite automata from a regular expression, which can be converted to a deterministic finite automata later.</p> <p>Changes should only be made to jscc_regex.par.</p> <p>Be careful when changing this file!</p> <p>(The grayed files are files which are generated by JS/CC itself during the make-process!)</p>	WSH JScript Web Rhino
jscc_bitset.js	Bitset functionalities for smarter memory usage at NFA construction.	JScript.NET WSH JScript Web Rhino
jscc_lexdfa.js	Subset-construction and minimization algorithm, builds the lexer tables for the lexical analyzer.	JScript.NET WSH JScript Web Rhino
jscc_lexdbg.js	Debug-functions for the lexical analyzer generation.	JScript.NET WSH JScript Web Rhino
Makefile	Makefile to manage the build process of the console-versions of JS/CC. As Make utility, Microsoft's nmake has been used, delivered with Visual Studio or Platform SDK.	
Makefile.rhino	Makefile for the Mozilla/Rhino port of JS/CC. Please run this on a Unix system, not on Windows.	

Table 3: src directory overview

Document folder

The document folder (**/doc**) contains this user's manual as PDF file.

Samples folder

The samples folder (**/samples**) contains some example files which are compileable using JS/CC. There you can find the examples used within this manual. Please run the Makefile to build the

examples.

Enhancing JS/CC itself

Building JS/CC

To setup a JS/CC development environment, the path to the JScript.NET Compiler must be in the system's PATH environment variable. It is also necessary to have a make-tool available, such as Microsoft nmake or Borland make, which also must be in the PATH.

Build JS/CC using the Makefile in the source directory.

Code style

When I developed JS/CC, I used a compact, C-like code style. JS/CC has also been build up on a structured and procedural way – no object-oriented aspect is used here.

When new code is added or changed within JS/CC, my wish is that this coding style is kept. It is also very important to hold a clean and continous style how the code is written.

Please write

```
if( a == b )
{
    x = 10; /* Comment! */
    y = new String();
    z = parseInt( x, 10 );
}
else
    x = 0;
```

instead of such ugly Kernighan&Ritchie-styled formatting like

```
if (a==b) {
    x=10;y=new String();z=parseInt(x,10);
} else {
    x=0;
}
```

It is really heavily to read and does not look that nice. Please do also watch for the parenthese and whitespace usage. A function/keyword call should be formatted as

```
function( param1, param2, param3 );
```

and not as

```
function(param1,param2,param3);
```

or the opposite

```
function ( param1 , param2 , param3 ) ;
```

It is also necessary to write comments wherever they are required. The function headers should be taken for new, huger functions. Smaller functions must not be furnished with a header, but a note about the author who wrote it and a short description will be useful.

If there where any changes made on JS/CC, please send them to me by mail, so I can put them into

the public distribution package. Version maintenance is also done by me.

Developer's documentation

I tried to comment the most functions as good as I can, so the code itself acts also as developer documentation. Please correct or enhance the comments wherever you think it is necessary.

Thanks a lot!

License Agreements

JS/CC is released under the Artistic License.

Preamble:

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions:

- "Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.
- "Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder.
- "Copyright Holder" is whoever is named in the copyright or copyrights for the package.
- "You" is you, if you're thinking about copying or distributing this Package.
- "Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)
- "Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as ftp.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.

b) use the modified Package only within your corporation or organization.

c) rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.

d) make other distribution arrangements with the Copyright Holder.

4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

a) distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.

b) accompany the distribution with the machine-readable source of the Package with your modifications.

c) accompany any non-standard executables with their corresponding Standard Version executables, giving the non-standard executables non-standard names, and clearly documenting the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.

d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own.

6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package.

7. Subroutines supplied by you and linked into this Package shall not be considered part of this Package.

8. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

9. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Credits

Thanks a lot to Louis P. Santillan for his efforts in porting JS/CC to the Mozilla/Rhino ECMAScript platform.



*A salute to all that real coders in the world
from Jan Max and Fynn!*