

JJCRS Howto

Kristoffer Rose
IBM Thomas J. Watson Research Center
<http://www.research.ibm.com/people/k/krisrose>

Contents

1	Quick Start Guide	1
2	Gentle Introduction	2
2.1	Syntax	2
2.2	Generating terms	2
2.3	Including pattern meta-variables	3
2.4	The stuff between	5
2.5	Making it run	5
3	Manual	5
3.1	Preamble	5
3.2	Structural productions	5
3.3	Lexical productions	7

Abstract

JJCRS is a frontend to the JavaCC [2] system intended to build parsers that construct CRS terms for use with CRSX higher order term rewrite engines [1]. This HOWTO explains how to use JJCRS.

The latest version of this document¹ can be found at <http://sf.net/projects/crsx> along with some examples of use and all source code, all distributed under the terms of the Common Public License version 1.0.

1 Quick Start Guide

Get the latest *crsxV.zip* file (this guide assumes $V = 4$) from <http://sourceforge.net/projects/crsx>. Unzip it to obtain the *crsx* directory and change into that directory. Run the command

```
java -cp crsx.jar: net.sf.crsx.jjcrs.JJCRS
```

substituting your proper Java 5 execution command, and check that the following is output:

¹This is revision 1.14 of April 1, 2008.

Usage: JJCRS input-url [output-file]

Now run the following command (all on one line):

```
java -cp crsx.jar: net.sf.crsx.jjcrs.JJCRS
  samples/calc/Calc.jjcrs
  samples/src/net/sf/crsx/samples/calc/Calc.jj
```

This creates the JavaCC file *samples/src/net/sf/crsx/samples/calc/Calc.jj* which on my system is processed by JavaCC by changing to the *samples/src/net/sf/crsx/samples/calc* directory and executing

```
javacc Calc.jj
```

(consult your JavaCC documentation for the precise details). Now we have a *samples/src/net/sf/crsx/samples/calc/Calc.java* file, which we can compile in the usual manner. Once this is done and the class is part of your normal class path then try it with the small included test CRSX script (all on one line)

```
java -jar crsx.jar
  script=samples/calc/test.script
```

which will output

```
Set[2,a,Compute[Plus[2,Times[2,a]],Done]]
simplifies to
Compute[Times[2,Compute[Plus[1,2],Done]],Done]
```

The referenced script file has the details in the usual CRSX style:

```
# Simple test of calculation rewrite.

# Use custom syntax.
parser net.sf.crsx.samples.calc.Calc

# Read term.
term set a = 2; 2+2a
echo term
```

```

echo simplifies to

# Normalize with simple rewrite system.
crs file = samples/Calc/Calc.crs
normalize
echo term

```

(We give the full details of this execution below.)

2 Gentle Introduction

JJCRS is about parsing languages and representing them as higher-order terms in the CRS formalism. This is achieved by writing productions annotated with instructions for how the parsed fragments are combined into terms.

2.1 Syntax

Here is a simple example of a grammar for a simple calculator language with dynamically created registers, written in JJCRS form (but not yet generating any terms):

```

S ::= (E | "set" <var> "=" E) (";" S)? .
E ::= A ("+" E | "-" E)? .
A ::= F (A | "*" A | "/" A)? .
F ::= <int> | "(" S ")" | <var> .
<int> ::= ["0"-"9"]+ .
<var> ::= ["a"-"z"] ["a"-"z", "0"-"9"]* .

```

This shows the following conventions:

- Productions use ::= and finish with . (period).
- Productions come in two flavors: *structural* productions define non-terminals named with simple identifiers, and *lexical* productions define terminals (or tokens) with names enclosed in <>s.
- Structural productions are similar to JavaCC grammars: The content of a structural production is built from unions (with |) of concatenated units that include non-terminals, terminals (in <>), literals (in ""), and groups (in ()) with the usual optional trailing ?, *, or +).
- Lexical productions are similar to JavaCC token definitions: The content of a lexical production is a regular expression using “character classes” (in [] with an optional ?, *, or +) and related things.

2.2 Generating terms

Here is a sample expression in the above syntax:

```
set a = 2; 2+2a
```

Our goal is to generate a CRSX terms for this language that captures the binding relationships, specifically that the a variable declared in the first part is bound in the second part. One such representation would use the following CRSX term to represent the calculation:

```
Set[2, a.Compute[Plus[2, Times[2, a]], Done]]
```

This involves two things:

1. Bind the appropriate constructors to the productions.
2. Identify where variables are created and bound.

Here are the structural productions with the appropriate annotations:

```

S ::= {Compute} E (";" S | {Done})
    | {Set} "set" <var>^x "=" E
    (";" S[x] | {Done}) .

E ::= A# ( {Plus}  {{A#}} "+" E
    | {Minus} {{A#}} "-" E
    | {{A#}} ) .

A ::= F# ( {Times} {{F#}} A
    | {Times} {{F#}} "*" A
    | {Divide} {{F#}} "/" A
    | {{F#}} ) .

F ::= <int>$ | "(" S ")" | <var>! .

```

These additional notations have been used:

- A CRS *constant constructor prefix*, {C}, generates a C-construction that will take everything that follows it as parameters (up to the end of the enclosing group). So the very first line leads to a Compute construct with either the result of E and S as parameters or the result of E and Done.
- Similarly, if a terminal is suffixed with \$, like <int>\$, then the actual token value is used as a *variable constructor prefix*, generating a construction that takes everything following as parameters (up to the end of the enclosing group as before – for <int>\$ that just happens to be right away as the group ends immediately after).
- If a terminal is suffixed with ^ and a variable name, such as <var>^x in the second line, then the terminal is expected to *define a new variable* where the provided name (x in our case) serves as a local alias.

- If a terminal is suffixed with ! then the terminal should be bound somewhere in the context. The <var>! in the last line will match the occurrences of a in our example and thus make the connection.
- Otherwise unmarked non-terminals are just recursively inserted, so, for example, the effect of the "(" S ")" choice in the last line propagates to be the effect of the F non-terminal in that case.
- If a non-terminal is suffixed with a variable alias in brackets, like S[x] in the third line of our example, then that means that the actual variable is “bound” in and thus can occur in subterms generated by the non-terminal. This is how we record, for example, that the a defined in our small example term is, in fact, bound in the 2+2a part.
- A non-terminal with a # suffix means that the subterm constructed by the underlying production is *buffered* and must be included *once* explicitly later in the production. So the A# first in the E-production means that an A should be parsed but the result is saved for later. (Notice that “once” means that the buffered subterm is *used* once, so the term can be used in several alternatives.)
- Double-braced complete terms, such as the {{A#}} after {Plus} in the in E-production, is a way to explicitly generate a subterm for insertion at the specified point. In the example the effect is that the subterm saved just before is inserted inside the Plus-construction as the first parameter. Here is another version of the E-production, which is completely equivalent, but achieves the necessary construction by buffering and constructing everything explicitly:

```
 ::= A# ("+" E# {{ Plus[A#,E#] }}
      | "-" E# {{ Minus[A#,E#] }}
      | {{ A# }} ) .
```

Also notice that we had to refactor the definition a little bit: we replaced the original S-production

```
S ::= (E | "set" <var> "=" E) (";" S)?.
```

with what is effectively

```
S ::= E (";" S)?
      | "set" <var> "=" E (";" S)?.
```

This is because the two occurrences of S have different binding contexts and so cannot be parsed the same

way. Also notice that optionality is rare when generating terms: even for the “there is nothing” case something usually has to be generated to make sure that the enclosing constructor has a consistent number of arguments. Specifically, we have replaced (";" S)? with (";" S | {Done}).

2.3 Including pattern meta-variables

One good reason for wanting to parse programs into higher-order terms such as provided by CRSX is that it is then possible to *rewrite* those programs when they are in term form. We might, for example, wish to express that calculations of the form $c + cx$ can be replaced by $c(1 + x)$. One way to express this is to use the CRSX-translated form and write a CRSX rewrite rule like

```
PushC: Plus[#c,Times[#c,#x]]
       → Times[#c,Plus[1,#x]]
```

which will rewrite our sample term

```
Set[2,a.Compute[Plus[2,Times[2,a]],Done]]
→ Set[2,a.Compute[Times[2,Plus[1,a]],Done]]
```

However, it is doubly nice if one can then express the rewrite rules *in the native syntax* of programs rather than generic CRSX syntax shown. The CRSX engine supports this through the generic parser interface. If appropriately set up then the following rewrite rule is equivalent to the previous one:

```
PushC: <code> #c+#c**x </code>
       → <code> #c*(1+#x) </code>
```

where the HTML <code>...</code> wrapper identifies the text to be parsed (with CRSX terms and rules written in HTML, tags only show up as appropriate formatting, of course). This is achieved by the CRS rules parser invoking our custom parser for the embedded code fragment (similarly to the way the JJCRS engine invokes the CRSX parser for embedded term construction).

To achieve the above affect, we have to extend the syntax of our small calculator with CRSX “meta-variables” in “meta-applications”, the construct used to build patterns in rules.

In CRSX, all symbols including a # are meta-variables, and this will work for calculations, too, as hinted at above, as # is not otherwise allowed. In addition, we can use brackets for the meta-application arguments as this does not create conflicts in the syntax. (In some richer language it can be quite hard to find unused syntax for meta-applications – a good trick is to *require* the meta-application argument parentheses.)

```

/** Toy calculator language. */
#class net.sf.crsx.samples.calc.Calc : S

/* Grammar. */
S ::= {Compute} E (";" S | {Done}) | {Set} "set" <var>^x "=" E (";" S[x] | {Done}) | MS.
E ::= A# ( {Plus} {{A#}} "+" E | {Minus} {{A#}} "-" E | {{A#}} ) .
A ::= F# ( {Times} {{F#}} A | {Times} {{F#}} "*" A | {Divide} {{F#}} "/" A | {{F#}} ) .
F ::= <int>$ | "(" S ")" | <var>! | MF .

/* Tokens. */
<int> ::= ["0"-"9"]+.
<var> ::= ["a"-"z"] ["a"-"z","0"-"9"]*.

/* Special pattern support. */
MS ::= <metaS>@ ("[" E ("," E)* "]" )? .
<metaS> ::= "##" <var> .
MF ::= <metaF>@ ("[" E ("," E)* "]" )? .
<metaF> ::= "#" <var> .

#skip ::= " " | "\r" | "\n" | "\t" .

```

Figure 1: Compilable JJCRS calculation parser.

Here is the F production from before with supporting productions to allow meta-applications:

```

F ::= <int>$ | "(" S ")" | <var>! | MF .

MF ::= <metaF>@ ("[" S ("," S)* "]" )?
<metaF> ::= "#" <var> .

```

We have permitted one more form of expression to the F-production, defined by the M production, which illustrates the following:

- A terminal with a @ appended is interpreted as a *meta-application opener*, which takes all following subterms (in the group, like \$ discussed previously) as arguments – here it is all the S subterms.

Now we can write rules like

```

PushC: <code> #c+#c#x </code>
       → <code> #c*(1+#x) </code>

```

from before.

However, this only permits meta-variables for expressions, not whole sessions. We should make sure that our calculation language allows patterns to occur everywhere needed for the rules we need to express – this has to be checked in each case, of course, and often meta-applications have to be permitted in several different places in the syntax.²

²However, it should be noted that we here exploit that the CRSX formalism is unsorted so there is no check that a given meta-application pattern actually matches something of the same syntactic category.

In our case we would also like rules that observe the bindings, for example, we might decide that all bindings of, say, the constant 2, should be inlined:

```

Inline2: <code> set x = 2; ##e[x] </code>
        → <code> ##e[2] </code>

```

To make this other kind of meta-variable (with double #) work we need to also augment the S production as follows:

```

S ::= ... | MS .

MS ::= <metaS>@ ("[" S ("," S)* "]" )?
<metaS> ::= "##" <var> .

```

The important thing to remember is that only aspects of the syntax that are *explicitly represented in the terms* can be matched against. So, for example, since our term representation parses both 2a and 2*(a) into Times[2,a], we intentionally have no way of distinguishing those terms.

Another consequence is that we have no way of writing patterns that identify constants.³ If instead the F production was written

```

F ::= Int <int>$ | ...

```

then it is possible to create a variant pattern syntax that will only match Int-constructions.⁴

³Actually, we do: CRSX allows magic patterns that recognize constructors with a name matching a particular regular expression but that is cheating.

⁴However, do *not* use this trick to mark variables as that breaks when the variable is substituted by something else keeping the now bad marker.

2.4 The stuff between

Finally, we need to make sure the lexical specification is complete, also for the stuff between the tokens. To this end a special skip production is needed:

```
#skip ::= " " | "\r" | "\n" | "\t" .
```

The right hand side is the same as for a lexical production.

2.5 Making it run

Now all we have to do is to wrap our specification up as a JJCRS file to be compiled.

Figure 1 shows the completed example, ready to pass to JJCRS. The generated file should then be processed by JavaCC and the Java source file compiled normally. Once the new class is available we can execute the script in the quick start guide.

3 Manual

The full grammar for a JJCRS specification is shown in Figure 2.⁵ This section explains the details of how a grammar is interpreted.

3.1 Preamble

The top-level production is *Grammar*.

1. A *Grammar* has $\langle \text{COMMENT} \rangle$ s of two kinds: *documentation* comments start with `/**` and are inserted in the generated JavaCC file on the next emitted unit. Other comments are ignored.
2. There must be one `#class` directive to provide the fully qualified $\langle \text{NAME} \rangle$ for the JavaCC-generated class and the *NameList* of top-level parseable non-terminals. Optionally one can specify the desired JavaCC options as well as imports for the JavaCC-generated class and additional embedded declarations.
3. An `#include` directive inserts the contents of the indicated resource, parsing it as productions unless *JavaCC* is specified (then it is inserted directly in the JavaCC file).
4. A simple $\langle \text{EMBEDDED} \rangle$ text at the production level is simply echoed out to JavaCC, for insertion of advanced JavaCC productions.

⁵The figure is typeset with the *jjcrs.sty* L^AT_EX package included in the release, if you like the style.

3.2 Structural productions

Structural productions define non-terminals.

1. Marking the non-terminal with $\$$ is equivalent to inserting a global constructor $\{\langle \text{NAME} \rangle\}$ for the entire right hand side.
2. Choice, concatenation, and the Kleene operators `?*`, all work as usual in BNF-based formalisms.
3. When a literal $\langle \text{STRING} \rangle$ is used in a structural production then the parser must match the literal. If the literal is not followed by a *Use* then it has no effect on the generated term (see below for alternatives). A following *Use* (items 8 to 12) will operate on the literal itself.
4. When a $\langle \text{TERMINAL} \rangle$ is used in a structural production then the parser must match it. If the terminal is not followed by any markers then it has no further effect and the parsed text of the terminal is discarded (see item below for alternatives). A following *Use* (items 8 to 12) will operate on the actual terminal token.
5. A non-terminal $\langle \text{NAME} \rangle$ requires that the parser finds a subterm for the corresponding structural production. If variables have been defined (with \sim , item 8) previously in the same production, and they should be bound for the subterm, then they must be listed in the variable *NameList* as this is what enables the use of the `!`-annotation for the corresponding variable occurrences. If the declaration of the non-terminal requires parameters then the same number of parameters should be passed using the $\backslash \langle \text{EMBEDDED} \rangle$ syntax (item 7). Finally, if not followed by a *Use* then the effect of parsing the non-terminal is appended to the currently constructed term. A following *Use* (items 8 to 12) will operate on the subterm, or, if it needs a constant, require the subterm to be a single constructor and use the root constructor symbol as the constant value.
6. Directly using an inline “braced” symbol in productions creates constant term structure without parsing anything. If the braced symbol is not followed by a *Use* then the braced text is used as a constructor name with the following production fragment providing the construction arguments (up to the end of the current production or group). A following *Use* (items 8 to 12) will operate on the braced literal itself.
7. An $\langle \text{EMBEDDED} \rangle$ token should contain a complete CRSX term, which is inserted. The term

```

Grammar ::= <COMMENT>* ClassDirective ((COMMENT) | Include | Structural | Lexical | Skip | <EMBEDDED>)* .

ClassDirective ::= "#class" <NAME> ":" NameList
  ("options" Opt ("," Opt)*)? ("imports" NameList)? ("with" <EMBEDDED>)? .
NameList ::= <NAME> ("," <NAME>)* .
Opt ::= <NAME> "=" (<NAME> | <NUMBER> | <STRING>) .

Include ::= "#include" ("JavaCC")? <URL> .

Structural ::= <NAME> ("$"?) " ::= " Choice " .
Choice ::= (Primary)* ("|" (Primary)*)* .
Primary ::= (Literal | Terminal | Nonterminal | Inline | <EMBEDDED> | "(" Choice ")") ("?"|"*"|"+"?)? .
Literal ::= <STRING> Use .
Terminal ::= <TERMINAL> Use .
Nonterminal ::= <NAME> Use ("[" (NameList)? "]"?) ("\\\" <EMBEDDED>)* .
Inline ::= <BRACED> Use .
Use ::= ("^" <NAME> | ("#" <NUMBER>)? ("!"|"!")? ("$" | "@")? (":" <BRACED>)* .

Lexical ::= <TERMINAL> (States)? " ::= " LexicalChoice (<EMBEDDED>)? " .
States ::= "(" ("*" | <NAME> ("," <NAME>)*)? (":" <NAME>)? ")" .
LexicalChoice ::= (LexicalPrimary)* ("|" (LexicalPrimary)*)* .
LexicalPrimary ::= (<STRING> | <UNICHAR> | <TERMINAL> | CharLiteral | "(" LexicalChoice ")") ("?"|"*"|"+"?)? .
CharLiteral ::= ("~")? "[" <CHAR> ("-" <CHAR>)? ("," <CHAR> ("-" <CHAR>))* "]" .

Skip ::= "#skip" (States)? " ::= " LexicalChoice (<EMBEDDED>)? " .

<COMMENT> ::= "/"* (~["*"] | ("*")+ ~["*","/"])* "*" / .

<EMBEDDED> ::= "{{" (~["}"] | "}" ~["}"])* "}" | "{" (~["}"] | "}")* ~["}"]* "}" |
  "{ {" (~["}"] | "}")* ~["}"]* "}" | "[[" (~["]"] | "]" ~["]"])* "]" |
  "%{" (~["%"] | ("%")+ ~["}"])* "%}" .

<STRING> ::= "\"\" (~["\n", "\\", "\""] | "\\\" ~["\n"])* "\"\" | "\"\" (~["\n", "\\", "\""] | "\\\" ~["\n"])* "\"\" .
<TERMINAL> ::= "<" ("#"?) ((LETTER) | <DIGIT> | "_" )+ ">" .
<BRACED> ::= "{" (~["{", "}", "_"])+ "}" .

<CHAR> ::= <STRING> | <LETTER> | <DIGIT> | <UNICHAR> .

<NUMBER> ::= <DIGIT>+ .
<NAME> ::= <LETTER> (["_", "." ]* ((LETTER) | <DIGIT>))* .
<#DIGIT> ::= [0-9] .
<#LETTER> ::= [A-Z, a-z] .
<#UNICHAR> ::= "#x" [0-9, A-F, a-f]+ .

#skip ::= ["_", "\t", "\n", "\r"]* | "/" ~["\n"]* .

```

Figure 2: JJCRS grammar.

can contain variables defined with `^` in the same production as well as meta-variables defined with a *Use* buffer marker, `#` (item 12). The full syntax of CRSX terms is described in the separate CRSX manual [1].

8. The *Use* marker `^` followed by a `<NAME>` means *define variable*: `^x` declares that the string token value is a defined variable: (a) when bound in subterms of the production with `[x]` (item 5) then any token inside that subterm marked `!` (item 9) with the same token value refers to the variable; (b) in embedded subterms the variable `x` can be used.
9. The *Use* marker `!` denotes the marked token value as a *variable use*, which means that it must denote a variable bound in the context (8). If marked with `!!`, then the variable may be “free” (so is created if not bound). If not followed by `#` then the variable occurrence is emitted right away, otherwise its is made available through the created meta-application.
10. The *Use* marker `$` denotes that the marked token is a *constructor*. If not followed by `#` then the following production fragment providing the construction arguments (up to the end of the current production or group), otherwise the construction is available through the created meta-application (and will get its parameters from there).
11. The *Use* marker `@` denotes that the marked token is a *meta-variable*. If not followed by `#` then the following production fragment providing the meta-application arguments (up to the end of the current production or group), otherwise the construction is available through the created meta-application (and will get its parameters from there).
12. The *Use* marker `#<NUMBER>` denotes that the marked entity should be *buffered* and made available through a meta-application of the form `<NAME>#<NUMBER>[...]`, which can only be used inside embedded terms (item 7) and parameters (item 5).
13. *Use* additions of the form `:f` change the text made available to the various following markings by invoking the `f` method, which must be defined in the parser class (see with of the class directive) with a declaration like

```
String f (String arg) {...}
```

3.3 Lexical productions

Lexical productions give the syntax of the text of tokens used as terminals.

1. If the name is prefixed by `#` then it is an “internal” token that can only be used in other lexical productions, otherwise it is available for use in structural productions. The right hand side follows JavaCC conventions, specially it allows what JavaCC calls a “complex_regular_expression”: the token text is described as a union of concatenations of strings, other terminals (non-recursively!), single character literals (allowing ranges of characters and using `~` for negation), and groups, with the usual optional occurrence indicators.
2. A “`#skip`” production is similar except it specifies a JavaCC “SKIP” declaration, which describes the notion of space allowed between units in structural productions.
3. The optional extra arguments on the left hand side of lexical productions set the lexical state transition of the token. The names in `<>`s are the states for which the token is active; `<*>` means every state; the default corresponds to `<DEFAULT>`. The name after `:` is the state, if any, that is set after the token has been parsed; the default is to not change the state.
4. An `<EMBEDDED>` text in a lexical production is inserted as a JavaCC “Java lexical action” last in the lexical rule.

References

- [1] Kristoffer Rose. Combinatory reduction systems with extensions. <http://crsx.sourceforge.net>, April 2008.
- [2] Sreeni Viswanadha, Sriram Sankar, et al. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*. Sun, 4.0 edition, January 2006.