VISL CG-3

People who know something about all this:

Tino Didriksen <tino@didriksen.cc>

Eckhard Bick <eckhard.bick@mail.dk>

URL: http://beta.visl.sdu.dk/cg3.html

Table of Contents

Request For Comments

Nothing in this document is set in stone. If you have ideas, needs, or disagree with the changes or additions outlined in this document, please let us know.

Naming

I have called this version of CG ”VISL CG-3” since it is made by VISL and is a general update of the grammar format and parser features that warrants a version bump. VISL CG-3 is fully backwards compatible with CG-2 and VISLCG, and contains switches to enforce their respective behavior in certain areas.

Character Encoding

By default, grammar file, input stream, and output stream are considered to be in ISO-8859-1 (Latin 1) encoding as this covers most the grammars and tool chains currently written. Internally and through switches there is full support for Unicode (UTF-16) and any other encoding known by the unicode library. I have chosen the IBM International Components for Unicode library[1] as it provides strong cross-platform unicode support with built-in Perl compatible regular expressions.

Grammar Distribution Protection

Compiled grammars have very little reverse-engineering protection. Even if you write out the compiled grammar to a plain text file (in itself a trivial task), you will lose all comments and set operations. A neat file such as

```
# Spelling sets
LIST RET = <R-Rare> <M-max> <F-max> <L-max> ;
LIST RET-LIST = <E-TYPE:list> ;
SET RET-ALL = RET OR RET-LIST ;
LIST RET-MAX = <W-max> <S-max> <M-max> <F-max> ;
SET RET-SUBMAX = RET-MAX - (<M-max>) ;
```

would after preprocessing, compilation, and subsequent decompilation look more like

```
LIST _GEN_AE3FC = <R-Rare> <M-max> <F-max> <L-max> <E-TYPE:list>
LIST _GEN_0BA6F = <W-max> <S-max> <F-max>
```

which on the whole doesn't convey nearly as much useful information.

Certain switches can furthermore enable hashing of match-only tags (tags to be added via mapping must be in plain text), and forced obscuring of all named entities (sets, offsets), plus stripping all knowledge of exact line numbers. The hashing can be reversed by checking which input tags hash to which known values, and tags are usually so short that even brute-force is an option.

1 http://www.ibm.com/software/globalization/icu/ and http://icu.sf.net/

Sections

CG-2 has three seperate grammar sections: `SETS`, `MAPPINGS`, and `CONSTRAINTS`. VISLCG added to these with the `CORRECTIONS` section. Each of these can only contain certain definitions, such as `LIST`, `MAP`, or `SELECT`. As I understand it, this was due to the original CG parser being written in a language that needed such a format. In any case, I did not see the logic or usability in such a strict format. VISL CG-3 has a single section header `SECTION`, which can contain any of the set or rule definitions. Sections can also be given a name for easier identification and anchor behavior, but that is optional. The older section headings are still valid and will work as expected, though.

By allowing any set or rule definition anywhere you could write a grammar such as:

```
DELIMITERS = "<$.>"
LIST ThisIsASet = "<sometag>" "<othertag>"

SECTION SafeSection
LIST ThisIsAlsoASet = atag btag ctag
SET Hubba =  ThisIsASet - (ctag)
SELECT ThisIsASet IF (-1 (dtag))
LIST AnotherSet =  "<youknowthedrill>"
MAP (@bingo) TARGET AnotherSet
```

Notice that the first `LIST ThisIsASet` is outside a section. This is because sets are considered global regardless of where they are declared and can as such be declared anywhere, even before the `DELIMITERS` declaration should you so desire[2]. A side effect of this is that set names must be unique across the entire grammar, but as this is also the behavior of CG-2 and VISLCG that should not be a surprise nor problem.

Rules are applied in the order they are declared. In the above example that would execute `SELECT` first and then the `MAP` rule.

2 In fact, DELIMITERS and PREFERRED-TARGETS are also global. Feel free to put them in the middle or end of the file.

New Rules

Firstly, the CG-2 optional seperation keywords IF and TARGET are completely ignored by VISL CG-3, as is the line terminating semi-colon, so only use them for readability. The definitions are given in the following format:

```
KEYWORD <required_element> [optional_element]
```

DELIMIT

```
[wordform] DELIMIT <target> [contextual_tests]
```

This will work as an on-the-fly sentence (disambiguation window) delimiter. When a reading matches a `DELIMIT` rule's context it will cut off all subsequent cohorts in the current window immediately restart disambiguating with the new window size.

ANCHOR, JUMP, EXECUTE

```
ANCHOR <anchor_name>
SECTION [anchor_name]
[wordform] JUMP <anchor_name> <target> [contextual_tests]
[wordform] EXECUTE <anchor_name> <anchor_name> <target> [contextual_tests]
```

These rules will allow you to mark named anchors and jump to them based on a context. In this manner you can skip or repeat certain rules. `JUMP` will jump to a location in the grammar and run rules from there till the end (or another `JUMP` which sends it to a different location), while `EXECUTE` will run rules in between the two provided `ANCHOR`s and then return to normal.

MINVALUE, MAXVALUE

```
MINVALUE <identifier> <integer>
MAXVALUE <identifier> <integer>
```

These are used to define the range of numerical tag matching (explained in next section). The identifier here is e.g. the `F` in `<F=15>`. You do not need to define these, nor do you need to define both at the same time; it will help if you know the range, but dynamic matching can be done even if you don't.

DEFINEGLOBAL, SETGLOBAL, REMGLOBAL

```
DEFINEGLOBAL <variable_name> <value> [value] [value] ... [value]
[wordform] SETGLOBAL <variable_name> <value> <target> [contextual_tests]
[wordform] REMGLOBAL <variable_name> <target> [contextual_tests]
```

Global variables complement and are very similar to Metadata, but are set and unset on-the-fly via the grammar as opposed to input information. First you define a uniquely-named global variable and all the values it must be possible for it to be, after which you can set and unset the variable to these values based on contextual tests. You can also test the value of a global variable in contextual tests via `GLOBAL:variable_name:value` or test if it is set at all with `GLOBAL:variable_name`. This is useful for sentences where you want to save certain states for later, such as which gender the main actor of a paragraph is.

Set Operations

Set Operation Comparison					
CG-2		***VISLCG***		***VISL CG-3***	
Union	OR	Union	OR	Union	OR
Difference	-			Difference	-
Cartesian Product	+			Cartesian Product	+
		Fail-fast	-	Fail-fast	^
		Intersection	+	*(see the Tags section)*	

A few example set declarations:

```
LIST ASET = A B C
LIST DSET = D E F
SET UNION = ASET OR DSET
SET DIFFERENCE = ASET - C
SET PRODUCT = ASET + DSET
SET FAILFAST = ASET ^ C
```

In CG-2 these are processed as:

```
UNION = A B C D E F
DIFFERENCE = A B
PRODUCT = (A D) (A E) (A F) (B D) (B E) (B F) (C D) (C E) (C F)
```

In VISLCG these are processed as:

```
UNION = A B C D E F
DIFFERENCE = A B but will never match the set if the reading contains C.
PRODUCT = (A D) (D A) (A E) (E A) ... (C F) (F C)
```

In VISL CG-3 these are processed as:

```
UNION = A B C D E F
DIFFERENCE = A B
PRODUCT = (A D) (A E) (A F) (B D) (B E) (B F) (C D) (C E) (C F)
FAILFAST = A B but will never match the set if the reading contains C.
```

Tags

First some example tags as we know them from CG-2 and VISLCG:

```
"<wordform>"
"baseform"
<W-max>
ADV
@error
(<civ> N)
```

Now some example tags as they may look in VISL CG-3:

```
"<Wordform>"i
"base?o*"w
"^[Bb]ase.*"r
<W>65>
(<F>=15> <F<=30>)
!ADV
^<dem>
(N <civ>)
```

Tag Order

Starting with the latter, `(N <civ>)`, as this merely signifies that tags with multiple parts do not have to match in-order; `(N <civ>)` is the same as `(<civ> N)`. This is different from previous versions of CG, but I deemed it unncecessary to spend extra time checking the tag order when hash lookups can verify the existance so fast. A side effect of this means there is no intersection set operator, nor need of one.

Literal String Modifiers

The upper three additions to the feature sheet all display what I refer to as literal string modifiers, and there are three of such: 'i' for case-insensitive, 'w' for wildcard matching, and 'r' for a Perl Compatible Regular Expression (PCRE) match. Internally, wildcard is the same as PCRE as the preprocessing step converts `"base?o*"w` to `"^base.o.*$"r`. Using these modifiers will significantly slow down the matching as a hash lookup will no longer be enough. Certain simpler constructs, such as `"^[bw](ase|ord)form$"r` can be preprocessed into a list of hashable finite possibilities: `"baseform" "bordform" "waseform" "wordform"`. Modifier 'w' and 'r' cannot be combined, but you could combine 'ir' or 'iw' for

case-insensitive PCRE and wildcard matches, respectively.

Numerical Matches

Then there are the numerical matches, e.g. `<W>65>`. This will match tags such as `<W:204>` and `<W=156>` but not `<W:32>`. The second tag, `(<F>=15> <F<=30>)`, matches values `15≥F≥30`. All usual numerical comparisons are valid: = != > < >= <=. These constructs are also slower than simple hash lookups, but are optimized with a partial hash index on the identifier (in these cases `W` and `F`). If you pass minimum and maximum attainable values to the parser, it can preprocess these tags into a finite list such as `<F:15> <F:16> ... <F:29> <F:30>` and `<W:66> <W:67> ... <W:999> <W:1000>`, assuming we set 1000 as max value.

Tag Negation

You can negate a tag by prepending a !, as in `!ADV`. This has the effect of matching if some tag other than `ADV` exists in the reading, but won't match if `ADV` is actually present. This is mostly useful in tags such as `(V TR !ADV)` as `!ADV` alone would match quite a lot.

Fail-Fast Tag

A Fail-Fast tag is the ^ prefix, such as `^<dem>`. This will be checked first of a set and if found will block the set from matching, regardless of whether later independent tags could match. It is mostly useful for sets such as `LIST SetWithFail = (N <bib>) (V TR) ^<dem>`. This set will never match a reading with a `<dem>` tag, even if the reading matches `(V TR)`.

Contextual Tests

You can query the existence of a tag in the previous or next disambiguation windows by using a test such as `(<2C (<div>))` which will pass if a cohort in the 2nd window to the left of the current has `<div>` in all its readings. This is a very powerful and potentially very slow operation.

Input Format and Metadata

Input format is also fully backwards compatible, but much more lenient. Anything that does not look like a cohort or reading is treated as plain text and the parser will output it unmodified inline in as close to the same place as the text was in the input.

Metadata is one of the areas that we have discussed and so far the solution is to treat input text such as `<META Newspaper> ... normal cohorts and plain text here ... </META Newspaper>` as defining a global `META:Newspaper` tag that you can reference in sets and rules. Metadata can be nested and/or interleaved at will.

Overlaps

Global variables, Metadata, and spanning disambiguation windows have considerable overlap in functionality. E.g., it might be tempting to query a previous disambiguation window for a gender, but it would be much lighter to define and track a variable for this purpose. On the other hand, if you need to look ahead there is not much choice but to query following windows, unless some kind soul has provided the necessary information as input metadata.

Complete Keyword List

```
DELIMITERS = <wordform> ...
PREFERRED-TARGETS = <tag> ...

LIST <set_name> = <tag> ...
SET <set_name> = <set_or_tag> [operation] [set_or_tag] ...

SECTION [anchor_name]

ANCHOR <anchor_name>
[wordform] JUMP <anchor_name> <target> [contextual_tests]
[wordform] EXECUTE <anchor_name> <anchor_name> <target> [contextual_tests]

DEFINEGLOBAL <variable_name> <value> ...
[wordform] SETGLOBAL <variable_name> <value> <target> [contextual_tests]
[wordform] REMGLOBAL <variable_name> <target> [contextual_tests]

MINVALUE <identifier> <integer>
MAXVALUE <identifier> <integer>

[wordform] DELIMIT <target> [contextual_tests]

[wordform] ADD <tag> <target> [contextual_tests]
[wordform] MAP <tag> <target> [contextual_tests]
[wordform] REPLACE <replacement> <target> [contextual_tests]

[wordform] SUBSTITUTE <remove> <inserts> <target> [contextual_tests]
[wordform] APPEND <inserts> <target> [contextual_tests]

[wordform] REMOVE <target> [contextual_tests]
[wordform] SELECT <target> [contextual_tests]
[wordform] IFF <target> [contextual_tests]
```

Quick tables of differences in the various CG versions

Grammar Keywords		
CG-2	***VISLCG***	***VISL CG-3 Strict***
DELIMITERS	DELIMITERS	DELIMITERS
PREFERRED-TARGETS	PREFERRED-TARGETS	PREFERRED-TARGETS
SETS	**SETS**	
LIST	LIST	LIST
SET	SET	SET
		SECTION
MAPPINGS	**MAPPINGS**	
ADD	ADD	ADD
MAP	MAP	MAP
REPLACE	REPLACE	REPLACE
	CORRECTIONS	
	SUBSTITUTE	SUBSTITUTE
	APPEND	APPEND
CONSTRAINTS	**CONSTRAINTS**	
REMOVE	REMOVE	REMOVE
SELECT	SELECT	SELECT
IFF		IFF
END	END	

Interesting Command Line Parameters		
CG-2	***VISLCG***	***VISL CG-3***
--debug	--debug	--debug
--check-only	--check-only	--check-only
--grammar	--grammar	--grammar
--expand		
--help	--help	--help
--unique-correct		--unique-correct
--input	--input	--input
--output		--output
--messages		--messages
	--no-corrections	--no-corrections
--no-mappings	--no-mappings	--no-mappings
--trace-constraints	--trace-constraints	--trace-constraints
	--trace-mappings	--trace-mappings
	--unsafe	--unsafe
		--encoding
		--disable-tags

Raw Brainstorm Text

* Central/modular engine with a wrapper for piping (e.g., vislcg/bincg), one for distribution generation (e.g., gencg), and one for embedding (e.g., OrdRet).

* Primary emphasis on speed. Target is 5 to 10 times faster than the current implementation.

* Pre-compute all set operations in rules. This takes a lot of memory and increases the startup time, but decreases the overall run time for large inputs.

* Word / Sentence / Document meta information

* Regular Expressions

* Literal string modifiers, a'la "<af>"i for a case-insensitive match, "<a*f>"w for a wildcard match, and "<a[fd]>"r for a regexp match. Mixing wildcard with regexp is not supported, but wildcard+nocase and regexp+nocase is fine.

* Tags are positive or negative. All are positive by default, but can be negated with !tag, which means everything except tag.

* Tags can furthermore be denied a'la ^tag, which means if tag is in the reading it can never match the set.

* The tag (*) will always match all existing readings, meaning you can do (3 (*)) to simply verify that a cohort exists 3 positions to the right. The tag itself cannot have modifiers of any kind, as they would not make sense.

* DELIMITER and PREFERRED-TARGETS can contain any tag.

* PREFERRED-TARGETS has been changed to operate on the last removed reading by pulling a preferred reading back from the dead.

* Tests can refer to previous and subsequent windows with < and > positions.

* Sets can deny other sets with operator ^. ASET ^ BSET is the same as ASET OR (^BSETs-tags). A set with tags (ANT VEC ^BEF) (^AF) will fail-fast, meaning if the reading has AF it can never match the set, even if it has ANT VEC without BEF. Similarly, ASET ^ (tag) is the same as ASET OR (^tag), which has the same behavior as ASET + (^tag) but the latter is not optimized by the parser. Logically, whether you add a denial to each and every tag with + or you check for a single tag denial with OR, is the same end-result; difference to the parser is O(n) vs O(1) though.

* Reference and compare numerical values a'la the <W:65> tags. All tags in the input matching /<[a-z]+:[0-9]+>/i will be checked against tags in the CG matching /<[a-z]+[:=<>][0-9]+>/i. E.g., the CG tags <W=65> and <W>10> will both match the input tag <W:65>.

* Order of application as order in file

* Named offsets, Jumps to named offsets.

* (NOT -2 xyz) risks being true for the first word if it is the only context.

* SUBSTITUTE

* APPEND, now with a target.

* wordform DELIMIT target tests, in a mapping context.

* Lots of caching, such as remembering which constraints/tests a certain cohort passed/failed for later -N references. +N refs could also be cached for later 0 and -N refs. Plus careful tests such as +1C also matches non-careful +1. A test for a single tag also matches tests for any sets which contain the tag.

* Threading: Since sentences must be run through the engine one at a time and completely self-contained, we can utilize multi-CPU machines better by processing multiple sentences in parallel.

* The test (1 (!tag)) is equivalent to (NOT 1 (tag)), but the latter is generally faster.

* A rule "<word>" SELECT SET + (tag) equals SELECT SET + (tag) + ("<word>")