Developing Sparser

Sparser is …

Checking out from SVN

The code for Sparser is part of a larger collection of code modules for natural language processing code sorted in the BUB SVN repository at

```
svn+ssh://<your name here>@src.dsl.bbn.com/nlp
```

At the moment, checking that out will populate seven toplevel directories, one of which is Sparser, which is all you need.

Legal

BBN has unlimited rights to market, deploy, use and extend Sparser, but nearly all of the code was developed starting in 1989 and continuing through 2005 by David McDonald operating as a sole proprietor with full ownership. To reflect this, every file in the Sparser source code has a header with one copyright line if it has never been revised or extended at BBN and two if it has been.

Here is an example of the header on an 'original' file. The first line is instructions to the Emacs editor; the second is the copyright line. There is a header identifying the name of the file and its symbolic location within the file system, followed by a history log. The copyright line lists the specific years in which the file was being worked on.

```
;;; -*- Mode:LISP; Syntax:Common-Lisp; Package:SPARSER -*-
;;; copyright (c) 1994,1995  David D. McDonald  -- all rights reserved
;;;
;;;     File:  "DM&P"
;;;   Module:  "objects;traces:"
;;;  Version:  February 1995

;; initiated 3/28/94 v2.3, added more 7/13. 8/11 adding traces for
;; the introduction of terms.  Small tweeks ...8/16.  11/17 cases for
;; capitalization. Teeked them 11/23.  Added some for vg-mining 1/3
..1/17
;; 1/23 fixed glitch.  2/13 added traces for RR processing

(in-package :sparser)
```

Whenever one of these original files is modified in other than a trivial way (modifying the case of the code or copying it between file systems with different newline conventions is 'trivial'), we add an additional copyright line for BBN and a hook for SVN repository data. The line reads "extensions copyright (c) <year> BBNT Solutions LLC. All Rights Reserved".

```
;;; -*- Mode:LISP; Syntax:Common-Lisp; Package:(SPARSER LISP) -*-
```

```
;;; copyright (c) 1992-1999  David D. McDonald  -- all rights reserved
;;; extensions copyright (c) 2008 BBNT Solutions LLC. All Rights
Reserved
;;; $Id$
;;;
;;;     File:  "object"
;;;   Module:  "model;core:places:"
;;;  version:  July 2008

;; initiated in 10/12/92 v2.3. Added 'kind of location' 1/17/94.  Added
location-
;; phrase 11/16/95. Added relative-location 11/99. 11/25 Moved in
spatial-
;; orientation (from [words;]). 12/12 fixed choice of category name.
;; 0.1 (7/13/08) Included a subcategory within Define-kind-of-location
so we no
;;      longer have to write kind-specific categories and rules.

(in-package :sparser)
```

If we add a completely new file, the copyright line leaves off the "extensions", since what we are recording is the provenance and status of individual files, not the Sparser system as a whole.

Loading

Sparser is designed to be run on any machine that runs Common Lisp, and to be readily adapted to any sort of file system structure. It is also possible to design custom combinations of grammar rules and models to be pre-loaded in a binary distribution as well as extended with specific modules of dynamically loaded rules that the end-user can extend and modify.

To accommodate this, and still run in an unmodified Common Lisp, Sparser includes a highly parameterized, multi-layer load facility, described in detail in §11.1. If the timing had been different, this facility would have been written using the defsystem capabilities of a Lisp Machine or today using the ASDF defsystem, but as it happened, there was no such capability available on the Apple Macintosh in 1989, and the loader is a completely custom development.

To load Sparser, you either load the Lisp file ~/init/everything.lisp[1] which, as its name suggests, will load all the grammar modules that are defined for the current version (see §x.x). More often, you will want to load a custom script (also a Lisp file, usually in ~/init/scripts/) that specifies a different choice of modules and sets some parameters.

For example, the current focus of development (2008) has been to develop the "Fire" variation of Sparser.[2] It is a variation on the earlier "Domain Modeling and Populating (DM&P) variant, where we try to develop the conceptual model for unknown head nouns and verbs bottom up from the context in which they appear. The specific settings for Fire

[1] From the perspective of the SVN repository this file is at /nlp/Sparser/code/s/init/everything.lisp. Since we rarely need to think about files 'above' the level of the "source" directory "s", we will use the tilde character (~) as an abbreviation for that part of the file path.

[2] "Free-text Information and Relation Extraction"

are in a script, and to load it you enter (your version of) this instruction into the REPL of an otherwise empty Lisp.[3]

```
(load "~/ws/nlp/Sparser/code/s/init/scripts/fire.lisp")
```

Another specialized version of Sparser under development is

!!!!!!!! full pathnames

Developing Rules and Models

Sparser can be used as a conventional recognition grammar, where one just write simple parsing rules (like the example phrase structure rules below). But it's forte is object-driven information extraction, where the output is a set of newly instantiated objects and the phrase structure rules are only a means to an end.

Crib sheet for phrase structure rule forms

Absolute minimum

```
(def-cfr based-at ("based" "at"))
```

Daughter referent — transparently incorporating prepositions

```
(def-cfr in-date ("in" date)
  :form pp
  :referent (:daughter right-edge))
```

Binding a variable of an already instantiated individual (Chomsky adjunction)

```
(def-cfr date (number date)
  :form np
  :referent (:head right-edge
             :bind (day . left-edge)))
```

Instantiating a category given its head word

```
(def-cfr earliest-arrival-date ("EAD")
  :referent (:instantiate-individual earliest-arrival-date))
```

Instantiating a category and binding one of its variables

```
(def-cfr quantity-of (number "x")
```

[3] Because of some legacy conventions that have yet to be routed out, the Lisp can not be case-sensitive. For example you would use the "alisp" version of Allegro.

```
  :form determiner
  :referent (:instantiate-individual requested-resource
             :with (quantity left-edge)))
```

Context Sensitive Rules

```
(def-csr kind location
  :left-context based-at
  :form np
  :referent (:function recast-kind-as-a-location right-edge))
```

Form Rules

```
(def-form-rule (quantity-of n-bar)
  :form np
  :referent (:function recast-as-resource right-edge))
```

"Exporting"

```
(set-generic-treetop-action category::earliest-arrival-date
                        'export-bindings/recursively)
```

A full example

Suppose we want to identify all the references to particular military units within some document, e.g. "ninth Army", "24th MEU". To do this we need to create a set of parsing rules for Sparser to apply, a representation for military units for these rules to use to store what they have found, a way of indexing the elements of this representation so that every reference to, e.g., the '2nd support command', is taken to the same instance of a military unit rather than creating a new one, and a way of signaling that military units are 'interesting' and should be included in objects that are extracted from the document.

Analysis

The first step in developing a representation and rule set for Sparser is to look closely at the way references to the things of interest are phrased. For things like military units, the conventions of the US military in how units are to be named are comparatively easy to analyze. For the simple cases, the name of a unit is composed of its 'type' – "Army", "Division", etc. – combined with a number.

The next consideration is whether these two parts have already been modeled, in which case we can just reuse them, e.g. ordinal numbers like 24th, or whether it's likely that they will useful in other relationships, in which case we should define them as separate notions now in anticipation of that later use. (Compare one of the ways that we describe something's age: "42 years old". The number and the unit of time are eminently reusable, but this use of "old" is idiosyncratic and stays within the rule system.)

Categories

The different types or kinds of objects that are modeled in Sparser are represented by 'categories' (see §9.1). Based on the simple analysis, we will want one category for the different types of military units, and another category for military units per se, which will incorporate a reference to the unit type along with its distinguishing information. For the moment, let us just look at how we could define the category for 'type-of-military-unit' and how it leads to the automatic creation of parsing rules, and take up the representation of full military units later on.

Categories are created by evaluating a special form as described in detail in §9.1.1. The result is (a) an object of type 'referential-category', and usually (b) one or more context free phrase structure rules (§4).

Defining a category entails specifying the following items. Strictly speaking only the name is required, but useful ones will provide slots for storing information about the category (variables) and information about what rules should be created (realization specifications).

1. Giving it a name.
2. Deciding where it should fit in the taxonomic inheritance lattice of the overall ontology (optional – :specializes).
3. Specifying how individuals that have this category as one of their types will be recorded in the discourse history (optional – :instantiates, see §9.6.).
4. Defining a set of 'variables' (§9.???) and their value restrictions (optional – :binds)
5. If individuals (§9.???) with this category are ever created by rule, they have to be 'interned' so that subsequent instances will all refer to the same object, in which case the :index field has to be filled out (§9.1.3).
6. Specifying how the we might recognize references to the category within a text, i.e. what rules should be written for it (:realization – §9.5).