

SMT Parser: Tutorial and Reference Manual

Sandiway Fong

University of Arizona

sandiway@arizona.edu

sandiway.arizona.edu

September 2024, updated April-May 2026. Draft August 2026.

[Incomplete: NOT FOR GENERAL DISTRIBUTION.]

The tutorial section covers installation, how to run the provided examples, and on how to modify **LEX** for the *LING 696A* course version of the SMT Parser, henceforth referred to as the **696A SMT Parser**. This version comes in macOS, Windows 11 and Linux flavors and is freely available for download and play.¹ Caveat: it is a limited teaching vehicle and not designed to be a general-purpose parser. The reference manual lists the **LEX** facts that must be defined for the parser to operate correctly for a particular language.

Brief History:

2024: SMT Parser using spec-INFL_φ. (*No longer available.*) First version of the manual. (*Deprecated.*)

2024-2025: NSF sub-project to add Spanish. Collaboration with PI Hilton Alers-Valentin (U. of Puerto Rico at Mayaguez).

2026: The **696A SMT Parser**. spec-INFL_φ eliminated with respect to Merge. Language-particular Edge of INFL_φ spellout defined in **EXT**. Revised manual (*still not quite complete*).

2027: A few examples from languages other than English and Spanish to illustrate points of theory, e.g. serial verbs.

Late 2027: Anticipated addition of the acquisition module for **EXT**. (*Perhaps an overly optimistic timeframe.*)

(The research version of the SMT Parser handles more examples, more languages, and contains a more advanced theory but currently has restricted distribution limited to collaborators. It is not freely available for obvious reasons, e.g. unclean development source code, instability due to implementation bugs, incomplete testing and outright mistakes in my tinkering with theory, etc. However, examples from the research version (unavailable in the 696A SMT Parser) may show up in papers and talks. If interested in discussing theoretic aspects or particular languages, please email me.)

Note: the linguistic theory behind the SMT Parser and how it works is documented elsewhere.

- See my BFSU talk slides for an overview.
URL: sandiway.arizona.edu/bfsu2026/bfsu2026.pdf
- There is a paper in preparation (based on the BFSU talk).
- See also extensive LING 696A course notes.

¹ The author gratefully acknowledges the work of UA graduate students *Alex Tubens* (Ph.D linguistics) and *Gabriela Quinones* (M.S. Human Language Technology) for their Spanish LEX contributions.

Contents

1. Installation and Run

- 1.1 macOS. *Llega un tren / Dos trenes llegan.*
- 1.2 Windows 1. *John ate a sandwich.*
- 1.3 Linux. *They are flying planes* (structurally ambiguous).

2. Tutorial: Background (LEX).

3. Tutorial: A worked example in English.

Birds that fly swim / Birds that fly instinctively swim

4. Tutorial: Another worked example in Spanish.

5. Reference Manual

- 5.1 Some Basic Notation.
- 5.2 The Initial Workspace (WS_0).
- 5.3 word2heads/3 (module parse).
- 5.4 Verbs θ /3 (module θ -theory), vStem/4, vRoot/3 (module LEX).
- 5.5 Verbs: passive, progressive, perfective (module LEX)
- 5.6 Nouns (module LEX) θ Rnoun/1, θ R/1 , nR/1.
- 5.7 Determiners (module parse).
- 5.7 Adjectives (module LEX) adj/2, aR/2.
- 5.9 Adverbs (module LEX) adv/2, advWord/2.
- 5.10 Prepositions (module LEX)
- 5.11 EXT
- 5.12 Re-ordering linear operations

6. Computation Graphs

- 6.1 Four styles shown on the course webpage
- 6.2 How to interpret the diagrams

[Note: diagrams used in this manual and on the webpage have been generated by the research version of the parser and are currently unavailable with the 696A SMT Parser.]

7. List of Examples Tested

- 7.1 English
- 7.2 Spanish

Brief Glossary

Important: this document does not describe the implemented linguistic theory (*described elsewhere*). Nevertheless, this alphabetical list of terms may be helpful in understanding the workings of the parser.

appl: *applicative verbal head*, written as appl_{θ} as it selects for a (usually dative) **IA**. **appl** is a head of the *propositional domain*. See discussion of *gustar* (like in Spanish).

aux verb: *auxiliary verb*, a non- θ -selecting head in the verb phrase. These are **v**'s that extend the core verb phrase, i.e. the θ -configuration, prior to transition to the *clausal domain*.

C: *complementizer head*, e.g. $C/C_{\text{rel}}/C_Q$ for declarative/relative/interrogative **C**, respectively. The head of the *clausal domain*.

Clausal domain, the domain of **C** and **INFL** heads.

EA: *external argument*, e.g. the θ -relevant item selected by v_{θ} . (Or a_{θ} , but not used in the 696A SMT parser).

EXT: language-particular module for externalization (of a **SO**). (A parameterized theory of **EXT** is under development.)

FormCopy (FC): an operation that is automatically triggered at the interface with **EXT**. In the case of c-commanding identical inscriptions within the same Phase, only the higher occurrence is pronounced. See M-gap in (Chomsky 2021). The *SMT Parser* indicates unpronounced occurrences in **SO**s using gray coloring.

FormSet (FS): *form set*, a **SBO**, a set-forming operations that selects for 2 (or more) **WS** items that are *coherent* in some sense, e.g. all are θ -relevant items, predicate heads or θ -configurations. Resulting set is stored into the (next) **WS**, and is accessible for future **Merge** or **FS** operations (as is the case with all new **WS** items).

fut: future tense (**TNS**), a feature of highest **v** (*not used in the English implementation*).

Head: a bundle of features from a syntactic "atom" of **LEX**, i.e. the smallest unit that can enter the **WS** in this theory. (Sub-atomic heads are not subject to syntactic **SBO**s, we assume they are formed within **LEX**.)

I-Language: core unconscious *Knowledge of Language* that is part of our heritage as humans and inaccessible to conscious thought (or introspection). The SMT Parser implements a theory of **I-Language**.

IA: *internal argument*, e.g. the θ -relevant item selected by R_{θ} . or appl_{θ} .

INFL: *inflectional head*, written as INFL_{φ} in this theory. The subscript φ stands for φ -features, e.g. Person, Number and grammatical Gender. INFL_{φ} triggers **MS** and finds a θ -relevant item valuing φ . INFL_{φ} is a head of the *clausal domain*.

Initial WS: the state of the **WS** before **Merge**, but after **heads** from **LEX** have been inserted.

LEX: *lexicon*. Contains heads that enter the initial **WS**. Currently, for simplicity there are separate **LEX** files for English and Spanish.

Linear operation: an operation that applies to a sequence (not to structure). In Chomsky's terms, these are *structure-independent* operations, not part of **I-Language**.

main verb: a verb selecting for at least one θ -relevant item.

Markovian assumption: **SBO**s operate on items from the current **WS** only, cannot reach back to items from earlier stages of the **WS**. Moreover, there are no **SBO** records kept in the **WS**, e.g. we cannot tell which **SBO** operation created the higher occurrence of **X** in $\{\text{X}, \{\text{X}, \text{Y}\}\}$. See M-gap in (Chomsky 2021).

Merge: a **SBO**, *binary set Merge* of two items X and Y from a **WS**. Resulting set {X, Y} is stored into the (next) **WS**, call it WS'. X and Y are *accessible* items from the **WS** prior to WS'. For *External Merge*, both X and Y are distinct WS items. For *Internal Merge*, only X is a WS item and Y is a sub-term of X found by **Minimal Search**.

MS: *Minimal Search*, c-command domain search for the closest relevant item, e.g. θ or *wh*.

NP: *noun phrase*, a **SO** headed by a nominal.

ϕ -features: person, number (and gender), e.g. 3sg-m for 3rd person singular masculine.

pastp: past participle **EXT** feature on **R**, e.g. pastp *comer* (*eat*) = *comido*.

pres: present tense (**TNS**), a feature of highest **v**.

presp: present participle **EXT** feature on **R**, e.g. presp *eat* = *eating*.

Propositional domain, see *θ -configuration*, for verb phrases the domain of (potential) θ -selecting **R**, **v** and **appl** heads.

pst: past tense (**TNS**), a feature of highest **v**.

Recursion: a **SBO** is *recursive* if its output can serve as input to **SBOs**. **Merge** and **FormSet** are recursive **SBOs**. Recursion requires a **WS**.

R: root, e.g. verbal root *dance* or *eat* _{θ} . Written with a θ -subscript if it selects for an **IA**.

SBO: (primitive) *structure building operation*, **Merge** or **FormSet** (**FS**) in this theory.

SO: (set-based) syntactic object, e.g. {**C**, {**INFL**_{v:3pl}, {**v**_{llegar:pres}, {**llegar** _{θ} , **trenes**_{dos}}}}}}

SMT: *Strong Minimalist Thesis*.

Structure: *syntactic structure (or object)*, a set object formed by **Merge(s)**.

Spellout: a linear process, e.g. the sequence ϕ -**TNS**-**R** 3sg *pst eat* maps to (irregular) *ate*.

θ : *theta*; if a subscript, e.g. on **llegar** _{θ} (*arrive*), head selects for a θ -relevant item. See **R**.

θ -configuration: a **SO** containing (**WS** obligatory) arguments and required predicate heads, e.g. for transitive *eat*, a configuration {**EA**, {**v**_{eat: θ :TNS}, {**eat** _{θ} , **IA**}}}}

θ -relevant item: a NP (or clause), a **SO** headed by a nominal (or **C**) that may bear a θ -role.

TNS: tense, e.g. pres/pst/fut, a feature of (structurally) highest **v**.

v: “little v” verbal head that may select for a θ -relevant item and hosts **TNS**. E.g. **v**_{eat: θ :pst} indicates that this head selects for verbal root *eat*, introduces a θ -relevant item (**EA**), and has **TNS** value *pst* (*past*). Generally, only the highest **v** in a verb phrase can bear **TNS**.

WS: *workspace* for recursive **SBOs**, **Merge** and **FormSet**. *Recursion* requires a **WS**. **Heads** from **LEX** populate an initial **WS**, called WS₀. Results from **SBOs** are placed into the (next) **WS**, e.g. WS₁, WS₂, etc. In this serial implementation, each **SBO** results in a transition between **WSs**. Earlier **WSs** are not accessible to **SBOs**, e.g. at WS_n, items from WS_{n-1} are not accessible, i.e. the *Markovian assumption* applies. (However, in principle non-interfering **SBOs** may apply at the same time and update the **WS** in parallel.)

1. Installation and Run

The *696A SMT Parser* requires *SWI Prolog* (swi-prolog.org) and *Python* (python.org), both freely available, to be present. Step by step instructions are provided below for macOS, Windows 11 and Ubuntu Linux.

As the parser is currently undergoing heavy development in terms of linguistic theory and computational implementation (and subject to radical change), a *.qlf* (*quick-load-file*, i.e. an intermediate code file) is provided instead of the complete (regrettably untidy) source code.²

1.1 SWI Prolog (macOS)

Both *MacPorts* (macports.org) or *Homebrew* (brew.sh) have been verified to work, but currently, the former seems more up to date, especially with respect to the Janus Python interface.³

(All commands below prefixed with the prompt `$` should be typed at the Terminal.⁴ Do not include the prompt itself.)

Install either MacPorts or Homebrew first. Then install SWI Prolog using:

```
$ sudo port install swi-prolog      (MacPorts)
```

```
$ brew install swi-prolog          (Homebrew)
```

(Reminder: `$` above is the sample Terminal prompt and should not be typed.)

On Apple Silicon, MacPorts' path prefix is `/opt/local` and `/opt/homebrew` is Homebrew's. Binaries are stored in `bin` under the prefix. Assuming your `PATH` environment variable has been set correctly during installation (to prefix + `bin`), the SWI Prolog executable file `swipl` can be found as follows:

```
$ which swipl
```

```
/opt/local/bin/swipl              (MacPorts)
```

```
/opt/homebrew/bin/swipl           (Homebrew)
```

If not, you probably should add these prefix + `bin` directories to your `PATH` variable for convenience.⁵

² The intention to provide full source for educational/academic use. The system is currently spread across the three programming languages, Prolog, Python and Javascript. As the code stabilizes, more components of the source code will become publicly available. EXT will probably be the last component to be released (as it's currently a mess).

³ See the *Janus* System. <https://arxiv.org/pdf/2308.15893>. SWI Prolog includes the Janus library.

⁴ *Terminal* is macOS's command line interface. See <https://support.apple.com/guide/terminal/welcome/mac>.

⁵ In Bash, the command `export PATH=dir:$PATH` will prepend *dir* to the current value of the shell variable `PATH`. The list of directories specified in `PATH` is searched (in order of specification) when a command is typed at the Terminal. Only the first matching command is executed, i.e. this is *Minimal Search* applied to a list of directories. (To see the current value of `PATH`, use the command `echo $PATH` or `printenv PATH`.)

The English parser partially relies on `wordnet.morphy()` from Python `nltk` to do (inflectional) stemming and for words not already present in the hand-built LEX.⁶

Prolog interfaces with Python via the Janus library. As there are possibly (and perhaps, likely to be) multiple Pythons installed on your machine, you should ensure that `nltk` is installed for the Janus-linked version.

First, check your Python and whether `nltk` has been installed (for that version) as follows:

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version X.Y.Z)
(rest of banner deleted)
?- ensure_loaded(library(janus)).
true.
?- py_shell.
% Interactive session; added `.` to Python `sys.path`
Python 3.13.13 (main, Apr 8 2026, 18:23:14)
(rest of banner deleted)
(InteractiveConsole)
>>> import nltk
>>> from nltk.corpus import wordnet as wn
>>> wn.morphy('relies','v')
'rely'
>>> ^D
now exiting InteractiveConsole...
true.
?- ^D
% halt
$
```

(Note: `?-` and `>>>` above are the Prolog and Python prompts, respectively. They should not be typed. `^D` is <Control>-D to exit.)

The version number of Python reported by the `py_shell` Prolog command above is 3.13, excluding the minor version number (13). If the `nltk` import command failed, but you know you already have `nltk` installed, the reason is that `nltk` needs to be installed specifically for the particular Python (here, 3.13) called by Janus. (Be careful, you could have multiple Python 3.13 installations at different locations on your machine.) If you know the PATH to the relevant Python executable, use it at the Terminal.

First, ensure that `pip`, the Python package manager, exists for your Python installation:

```
$ which python3.13
/opt/local/bin/python3.13 (MacPorts, in this case)
$ /opt/local/bin/python3.13 -m ensurepip --upgrade
```

Then install `nltk` using `pip`:

⁶ NLTK is the *Natural Language Toolkit* built on top of Python. See nltk.org. WordNet is a large (hand-built) relational database of English verbs, nouns, adjectives and adverbs organized by synonym sets. See wordnet.princeton.edu.

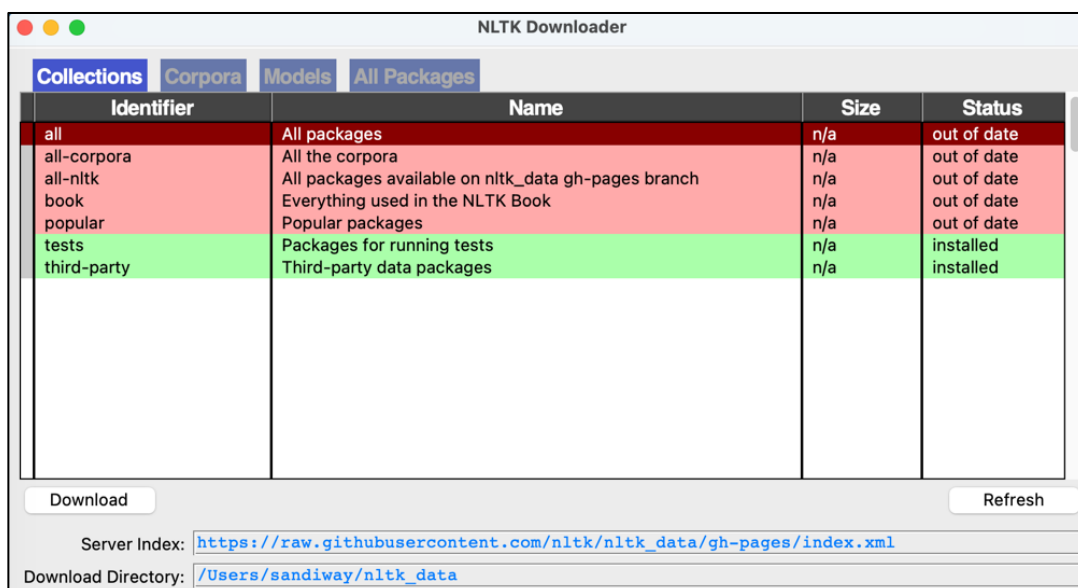
```
$ /opt/local/bin/python3.13 -m pip install nltk
Defaulting to user installation because normal site-packages is not writeable
Collecting nltk
  Downloading nltk-3.9.4-py3-none-any.whl.metadata (3.2 kB)
(much output deleted)
Successfully installed ... nltk-3.9.4 regex-2026.4.4 tqdm-4.67.3
```

Next, make sure relevant nltk corpora, e.g. wordnet, has been installed already. If so, it is a good idea to verify `wordnet.morphy()` as shown earlier.

If wordnet cannot be found, download the corpora from inside Python.

```
$ python
Python 3.12.13 (main, Mar  7 2026, 04:43:23)
>>> import nltk
>>> nltk.download()
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-
pages/index.xml
True
>>> ^D
```

Note version 3.12 (*shown above*) is a different version from 3.13 (*shown earlier*), but this is not a problem. Unlike nltk itself, nltk corpora are stored under `~/nltk_data` on macOS and can be shared across multiple Python installations.⁷ Depending on your Python, you may see a pop-up dialog window (*below*) or a text version (*not shown*) for `nltk.download()`.



If you're having trouble locating the right Python executable on your computer, the nltk package can also be installed directly from inside `py_shell` if the pip package already exists.

⁷ Notation: `~` is shell shorthand for your home directory, e.g. in my case this would be `/Users/sandiway`. Therefore, nltk corpora would be stored somewhere inside `/Users/sandiway/nltk_data`.

Example:

```
$ swipl
?- ensure_loaded(library(janus)).
true.
?- py_shell.
% Interactive session; added `.` to Python `sys.path`
>>> import pip
>>> pip.main(['install','nltk'])
>>> import nltk
>>> nltk.download()
>>> from nltk.corpus import wordnet as wn
>>> wn.morphy('did','v')
'do'
>>> ^D
```

Finally, start the parser. The following example assumes you're in the directory with the SMT Parser files in Terminal and using the Spanish language version. . For example, if you download spanish.zip, first do `unzip spanish.zip`. Then `cd` into directory `spanish`.

(For English language examples, see the Windows and Linux sections below.)

```
$ swipl -s code_Spanish.prolog
% Janus 1.5.2 embeds Python 3.13.13 (main, Apr 8 2026, 18:23:14) ...
WordNet LEX: ?- noMorphy. turns off WordNet LEX, ?- useMorphy. turn back on.
(rest of banner deleted)
?- parse([llega,un,tren],S0,L).
```

```
Words: llega un tren
Initial WS: tren_un  llegar_θ  v_llegar:pres  INFL_v:3sg
INT/EXT: {C, {INFL_v:3sg, {v_llegar:pres, {llegar_θ, tren_un}}}}
Initial Spellout: un tren 3sg-m pres llegar
Spellout: un tren llega
Error: spellout inconsistent with original words!
Initial Spellout: 3sg-m pres llegar un tren
Spellout: llega un tren
Parse found!
S0 = {C, {INFL_v:3sg, {v_llegar:pres, {llegar_θ, tren_un}}}},
L = [llega, un, tren] ;
false.
```

```
?- report.
```

```
REPORT LENGTH: 21
true
```

```
?- ^D
% halt
```

(The disjunctive semicolon (;) is to tell Prolog to search for other solutions. Note, **false** after ; does not mean the results are invalid, it just means no other solutions have been found. If you prefer not to continue the search, just press the <return> key to stop and return to the prompt.)

The -s (script) flag tells swipl to load the file code_Spanish.prolog from the current directory upon startup. Alternatively, load the parser after startup as follows:

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 10.0.2)
(rest of banner deleted)
?- [code_Spanish].
% Interactive session; added `.` to Python `sys.path`
% Janus 1.5.2 embeds Python 3.13.13 (main, Apr 8 2026, 18:23:14) [Clang
21.0.0 (clang-2100.0.123.102)]
WordNet LEX: ?- noMorphy/useMorphy. turns off/on WordNet LEX,
(rest of banner deleted)
?- parse([dos,trenes,llegan],S0,L).

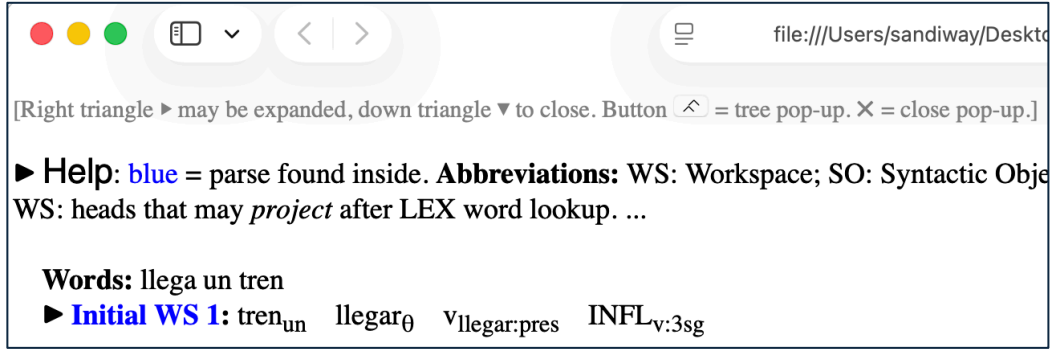
Words: dos trenes llegan
Initial WS: llegar_θ v_llegar:pres INFL_v:2pl trenes_dos
Initial WS: llegar_θ v_llegar:pres INFL_v:3pl trenes_dos
INT/EXT: {C, {INFL_v:3pl, {v_llegar:pres, {llegar_θ, trenes_dos}}}}
Initial Spellout: dos trenes 3pl-m pres llegar
Spellout: dos trenes llegan
Parse found!
S0 = {C, {INFL_v:3pl, {v_llegar:pres, {llegar_θ, trenes_dos}}}},
L = [dos, trenes, llegan] ;

Initial Spellout: 3pl-m pres llegar dos trenes
Spellout: llegan dos trenes
Error: spellout inconsistent with original words!
false.

?- ^D
% halt
```

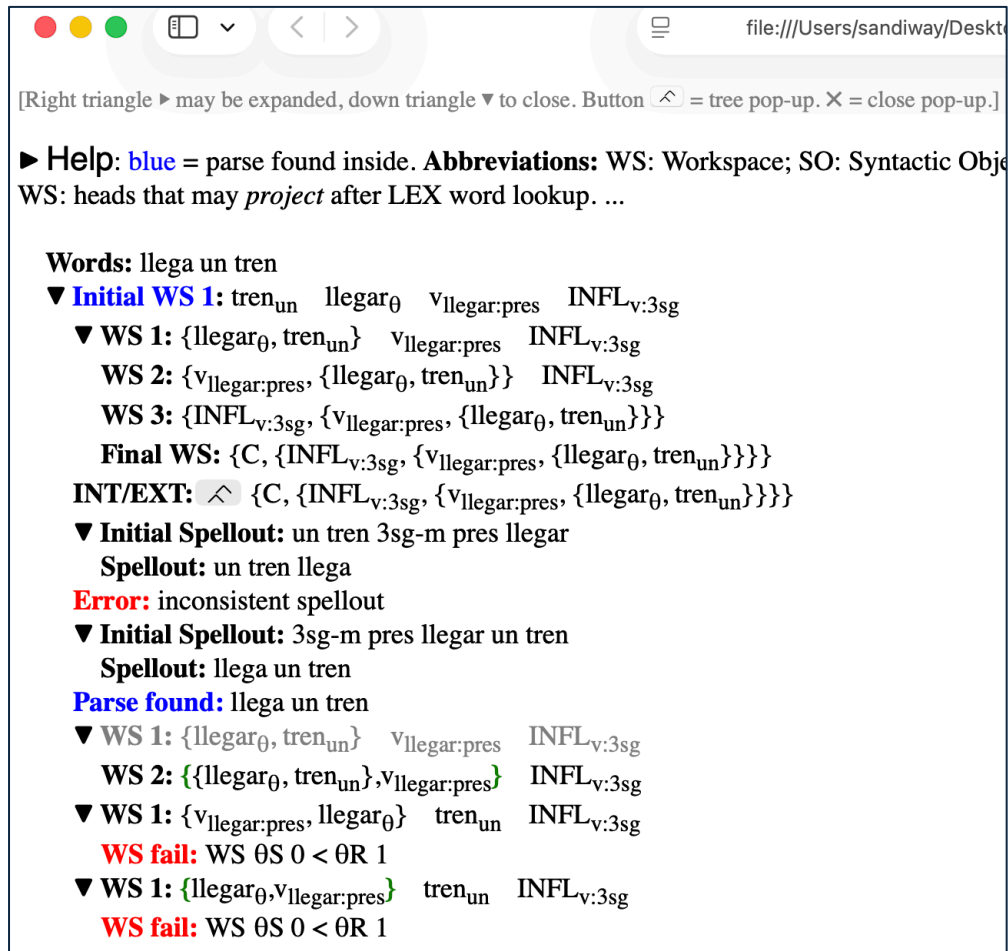
The Prolog command `parse/3` above invokes the SMT Parser. It takes three arguments, (i) a list of words, (ii) a variable **SO** (named above as `S0`, but you could use any variable name you like), and (iii) a variable `L`, the list of words resulting from **EXT** of `S0`. In the 696A version, parses are reported (one-at-a-time) inside Prolog. In the above examples, viz. *llega un tren* (*a train arrives*) and *dos trenes llegan* (*two trains arrive*), a single **I-Language** structure is built (in each case) with **EXT** providing two licit word orderings, but only one can be linearized as the input sentence.

The command `report` produces a detailed summary for the parsing called `output.html` in the same directory. On macOS, the SMT Parser should automatically pop up the contents of this file in a browser:



The Workspace can be explored by clicking on the triangles (►) and seeing which derivations leads to successful parses and which are dead-ends.

Example:



On the SMT Parser webpage, this is how examples are displayed.

(Note: there are also network diagrams for the computational space, currently available for the examples on the website. See Section 7 for details.)

1.2 SWI Prolog (Windows 11)

Use the self-installing executable for Microsoft Windows 64-bit editions from swi-prolog.org. (You will need *Administrator Privileges* on your machine.)

The Janus interface to Python has specific requirements on Windows:

Calling Python from Prolog is supported out of the box, provided the folder holding `python3.dll` is in the search `%PATH%`.

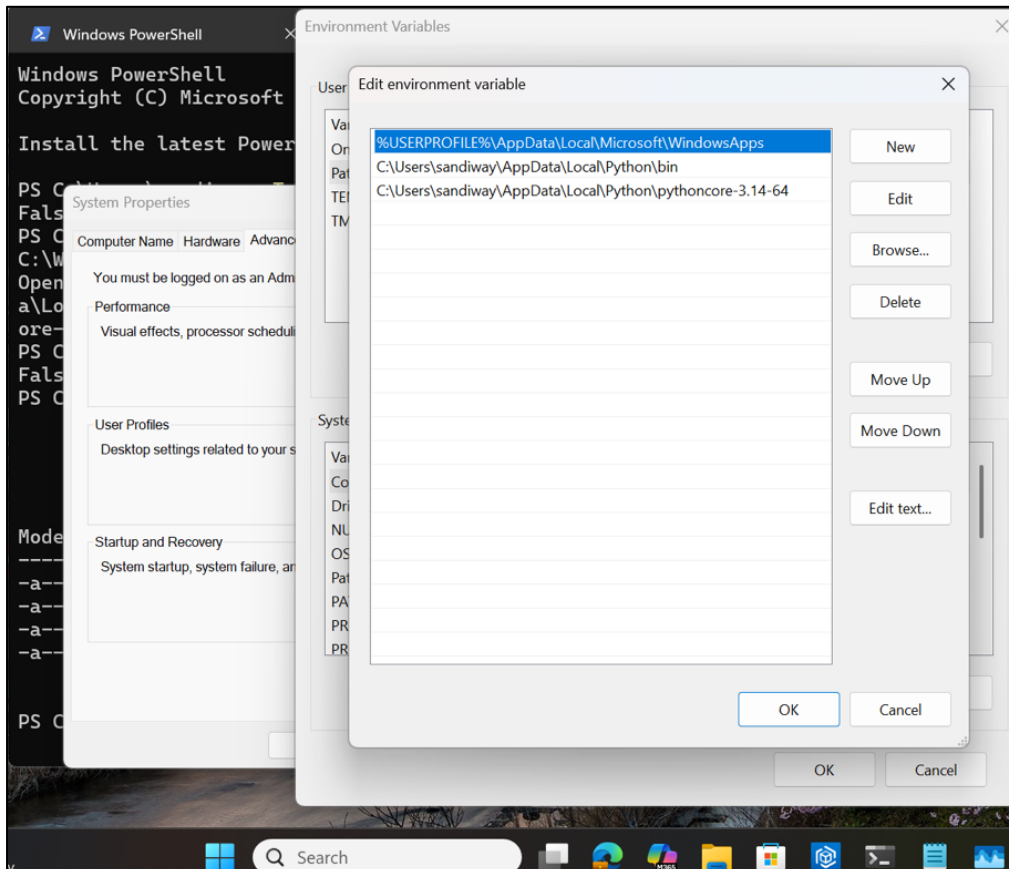
Python from `python.org` will be installed in `AppData\Local\Python`. Check the specific directory under Python for `python3.dll`. In my case, I have Python 3.14 installed, so the relevant subdirectory is `pythoncore-3.14-64`. (-64 here stands for 64 bits.)

```
PS C:\Users\sandiway> ls C:\Users\sandiway\AppData\Local\Python\pythoncore-3.14-64\p*

Directory: C:\Users\sandiway\AppData\Local\Python\pythoncore-3.14-64

Mode                LastWriteTime         Length Name
----                -
-a-----         3/26/2026   6:54 PM          106328 python.exe
-a-----         3/26/2026   6:54 PM           74072 python3.dll
-a-----         3/26/2026   6:54 PM        6753112 python314.dll
-a-----         3/26/2026   6:54 PM          104792 pythonw.exe
```

Then, in Settings, edit the environment variable `PATH` to point to this directory.



The SMT Parser makes use of the now-standard Unicode utf-8 variable-width character encoding format (*Unicode Transformation Format-8*). In macOS and Linux, this is the default, but not on Windows 11 (it seems). In Windows PowerShell, check the character set encoding as follows:⁸

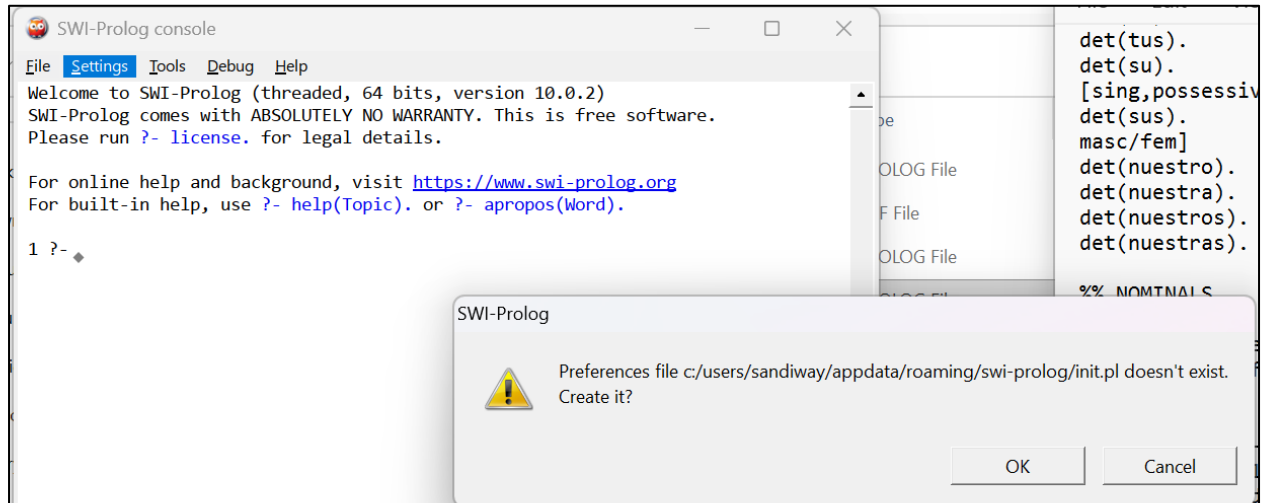
```
$ swipl
?- current_prolog_flag(encoding,X).
X = utf8.
?- ^D
% halt
```

If utf-8 is *not* reported, create a Prolog initialization file to set initialize it to utf-8 whenever SWI Prolog starts up. (Otherwise, there will be file reading errors when the parser is loaded.) To discover the obscure directory for storing the initialization file, note the SWI Prolog documentation states:

After the system initialisation, the system consults (see [consult/1](#)) the user's *init* file. This file is searched using [absolute_file_name/3](#) using the path alias (see [file_search_path/2](#)) `app_config`. This is a directory named `swi-prolog` below the OS default name for placing application configuration data

⁸ On Windows, the equivalent of Terminal on macOS for command line use is either PowerShell or the historical Command shell. PowerShell is recommended for Windows 11. In this document, I assume you are using PowerShell. See <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands>.

In my case, the directory is under my home directory is the *nearly-impossible-to-remember* `appdata/roaming/swi-prolog`. The SWI Prolog Settings drop-down menu contains a `User init` file entry. Selecting it produces a pop-up window.



Notepad can be used to create this `init.pl` file.⁹ It should contain the `set_prolog_flag/2` line shown below. (Note: the `:-` prefix must be included as this is not a fact to be asserted; instead, it is a command that is executed every time the Prolog interpreter starts up.)

⁹ `.pl` is the default Prolog file extension. Unfortunately, `.pl` is overloaded as it is also the Perl file extension. On my computer, since I also program in Perl, I use `.prolog`, which SWI Prolog accepts. Although I have not tested this, I suspect `init.pl` must be named `init.pl` and not `init.prolog`.

```
init.pl
File Edit View
:- set_prolog_flag(encoding, utf8).

Ln 2, Col 1 | 36 characters | Plain text | 100% | Windows (CF) | UTF-8

PS C:\Users\sandiway\appdata\Roaming\swi-prolog> notepad init.pl
PS C:\Users\sandiway\appdata\Roaming\swi-prolog>
```

In PowerShell, make sure pip is installed. py is the command for Python.

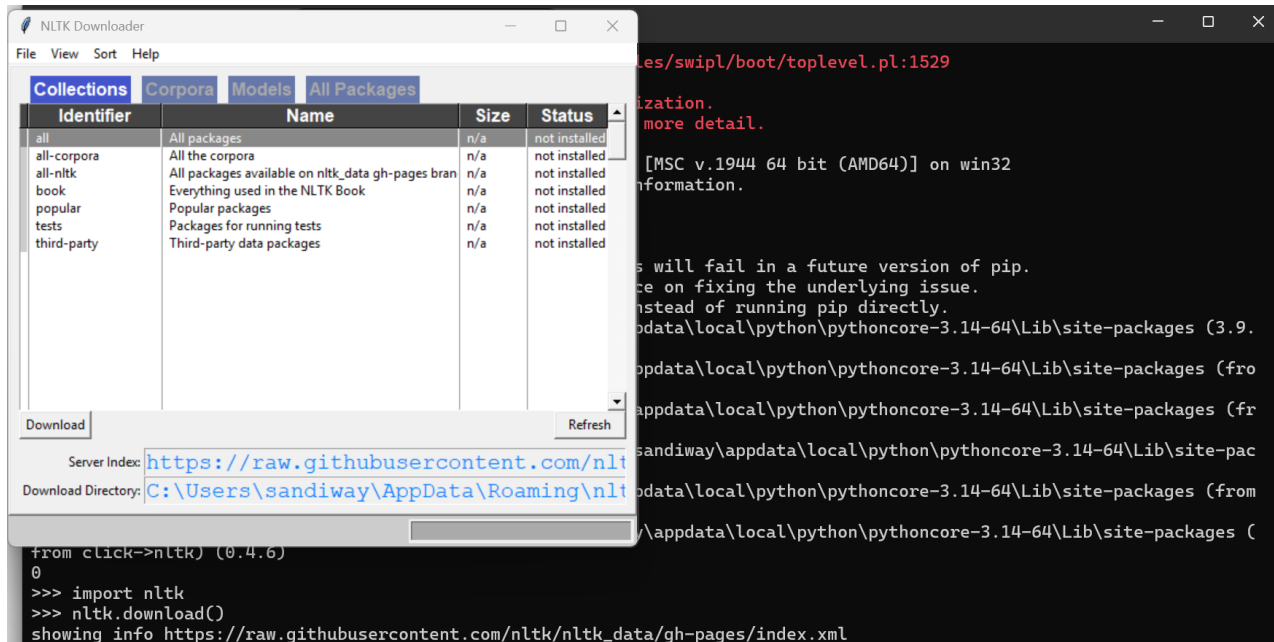
```
py -m ensurepip --upgrade
```

For installing nltk, you can follow the macOS installation documentation above with the appropriate command name substitutions. For example:

```
py -m pip install nltk
```

In the case of Windows, the wordnet, nltk corpora installed via `nltk.download()` will be stored in directory `~\AppData\Roaming\nltk_data`.¹⁰

¹⁰ Notation: the shortcut `~` refers to your home directory. In my case this is `C:\Users\sandiway`.



(Note: type <Control>-Z <Return> to exit Python in Windows.)

In PowerShell, first ensure you are in the directory with the SMT Parser files. For example, if you download english.zip, do

Expand-Archive -Path english.zip -DestinationPath .¹¹

Then cd into directory english. Then you can start the parser as follows:

```

swipl -s code_English.prolog
% Janus 1.5.2 embeds Python 3.14.3 (tags/v3.14.3:323c59a, Feb 3 2026,
16:04:56) ...
WordNet LEX: ?- noMorphy. turns off WordNet LEX, ?- useMorphy. turn back on.
(parts of welcome banner omitted)
Welcome to SWI-Prolog (threaded, 64 bits, version 10.0.2)
?- parse(['John',ate,a,sandwich],S0,L).

```

```

Words: John ate a sandwich
Initial WS: sandwich_a eat_0 v_eat:0:pst INFL_v John
INT/EXT: {C, {INFL_v, {John, {v_eat:0:pst, {eat_0, sandwich_a}}}}}
Initial Spellout: John 3sg pst eat a sandwich
Spellout: John ate a sandwich
Parse found!
S0 = {C, {INFL_v, {John, {v_eat:0:pst, {eat_0, sandwich_a}}}}},
L = [John, ate, a, sandwich] ;

```

```

INT/EXT: {C, {INFL_v, {sandwich_a, {v_eat:0:pst, {eat_0, John}}}}}
Initial Spellout: a sandwich 3sg pst eat John
Spellout: a sandwich ate John
Error: spellout inconsistent with original words!

```

¹¹ Expand-Archive is a PowerShell cmdlet.

false.

2 ?- report.

REPORT LENGTH: 32

true.

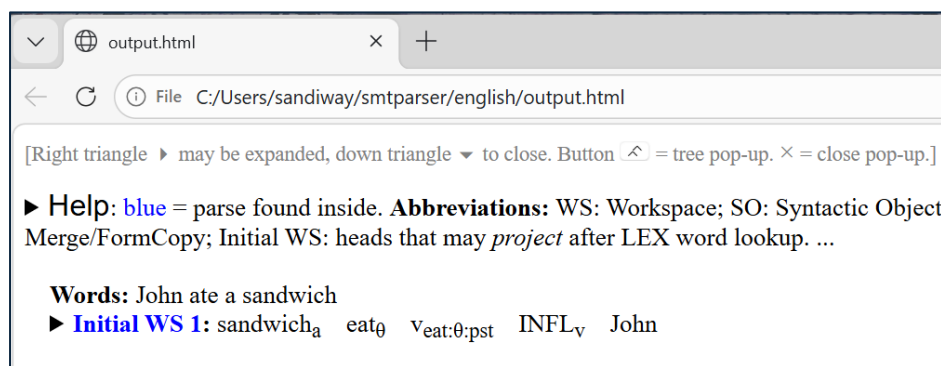
(Note that the input word *John* ('John') must be quoted because Prolog variables generally begin with an uppercase letter. Without quotes, John would be a variable. Atoms normally do not require quotes in Prolog, cf. Python.)

The -s (script) flag tells swipl to load the file `code_English.prolog` from the current directory upon startup.

The Prolog command `parse/3` above invokes the SMT Parser. It takes three arguments, (i) a list of words, (ii) the variable `S0`, denoting the **SO** recovered, and (iii) the variable `L` representing the list of words resulting from **EXT** of `S0`.

Parses are reported (one-at-a-time) inside Prolog. (The disjunctive semicolon (;) tells Prolog to search for other parses. Alternatively, press the <return> key to stop.) In the above example, one **I-Language** structure is built with **EXT** providing two licit word orderings, but only one is compatible with the input sentence.

The command `report` produces a detailed summary for the parsing called `output.html` in the same directory. The file should automatically appear in the browser.¹²



The Workspace can be explored by clicking on the triangles (►) and seeing which derivations leads to successful parses and which are dead-ends.

Example:

¹² In Windows 11, the browser defaults to Microsoft Edge.

```

output.html
File C:/Users/sandiway/smtparser/english/output.html

[Right triangle ▶ may be expanded, down triangle ▼ to close. Button ⌵ = tree pop-up. ✕ = close pop-up.]

► Help: blue = parse found inside. Abbreviations: WS: Workspace; SO: Syntactic Object; Merge/FormCopy; Initial WS: heads that may project after LEX word lookup. ...

Words: John ate a sandwich
▼ Initial WS 1: sandwicha eatθ veat:θ:pst INFLv John
  ▼ WS 1: {eatθ, sandwicha} veat:θ:pst INFLv John
    WS 2: {veat:θ:pst, {eatθ, sandwicha}} INFLv John
    WS 3: {John, {veat:θ:pst, {eatθ, sandwicha}}} INFLv
    WS 4: {INFLv, {John, {veat:θ:pst, {eatθ, sandwicha}}}}
    Final WS: {C, {INFLv, {John, {veat:θ:pst, {eatθ, sandwicha}}}}}
  INT/EXT: ⌵ {C, {INFLv, {John, {veat:θ:pst, {eatθ, sandwicha}}}}}
  ▼ Initial Spellout: John 3sg pst eat a sandwich
    Spellout: John ate a sandwich
  Parse found: John ate a sandwich
  ▼ WS 1: {eatθ, sandwicha} veat:θ:pst INFLv John
    WS 2: {{eatθ, sandwicha}, veat:θ:pst} INFLv John
    WS fail: WS θS 0 < θR 1
  ▼ WS 1: {eatθ, John} sandwicha veat:θ:pst INFLv
    WS 2: {veat:θ:pst, {eatθ, John}} sandwicha INFLv
    WS 3: {sandwicha, {veat:θ:pst, {eatθ, John}}} INFLv
    WS 4: {INFLv, {sandwicha, {veat:θ:pst, {eatθ, John}}}}
    Final WS: {C, {INFLv, {sandwicha, {veat:θ:pst, {eatθ, John}}}}}
  INT/EXT: ⌵ {C, {INFLv, {sandwicha, {veat:θ:pst, {eatθ, John}}}}}
  ▼ Initial Spellout: a sandwich 3sg pst eat John
    Spellout: a sandwich ate John
  Error: inconsistent spellout
  ▼ WS 1: {eatθ, John} sandwicha veat:θ:pst INFLv
    WS 2: {{eatθ, John}, veat:θ:pst} sandwicha INFLv
    WS fail: WS θS 0 < θR 1
  ▼ WS 1: {veat:θ:pst, eatθ} sandwicha INFLv John
    WS fail: WS θS 1 < θR 2
  ▼ WS 1: {eatθ, veat:θ:pst} sandwicha INFLv John
    WS fail: WS θS 0 < θR 2

```

1.3 SWI Prolog (Linux)

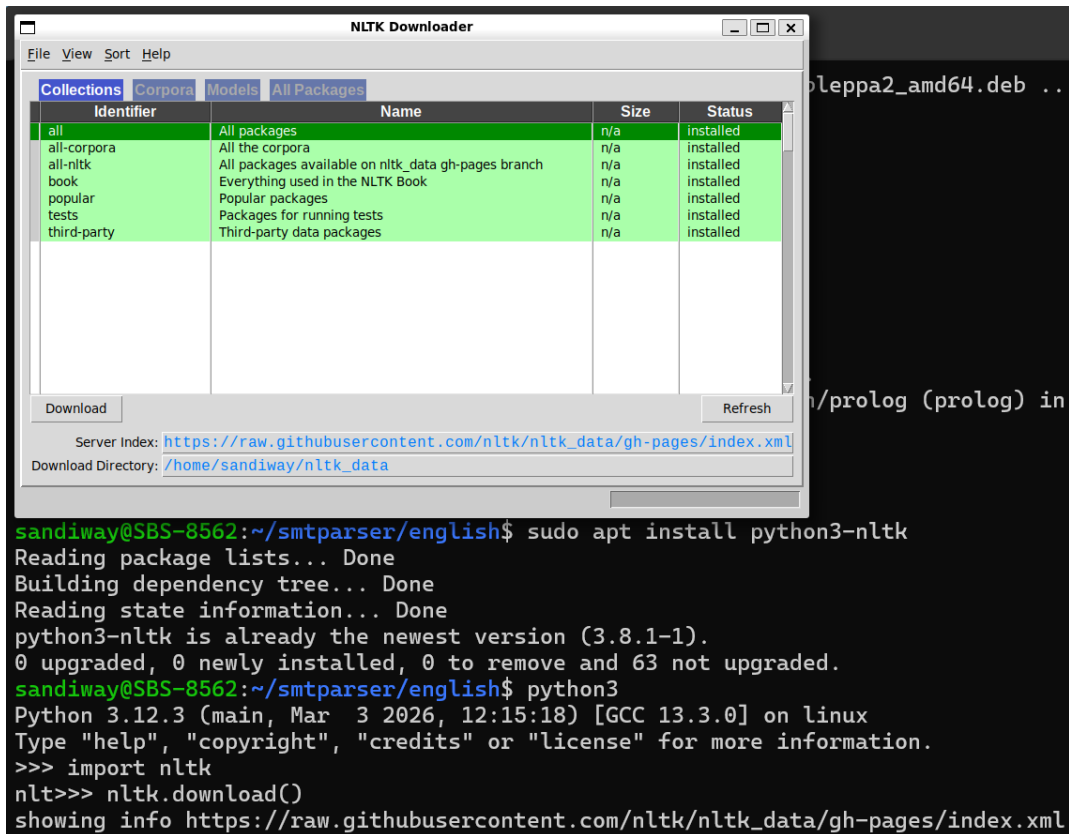
In Linux, virtual environments for specific projects rather than system-wide Python packages installs are now *de rigueur*, i.e. direct pip use is blocked/deprecated.

```
$ lsb_release -a
Ubuntu 24.04.4 LTS
```

Ubuntu 24.04 LTS supports (and relies on) Python 3.12.3. Use `apt install` to load common Python packages. The standard command to invoke (pre-installed) Python is `python3`.

Example:

```
$ sudo apt install python3-nltk
$ python3
>>> import nltk
>>> nltk.download()
```



The SWI Prolog documentation gives the following Ubuntu PPAs (*Personal Package Archives*) instructions.

```
$ sudo apt-add-repository ppa:swi-prolog/stable
$ sudo apt-get install swi-prolog
```

In the Terminal, make sure you are in the directory with the SMT Parser files. For example, if you download english.zip, first do `unzip english.zip`.¹³ Then `cd` into directory english.

Then you can start the parser as follows:

```
$ swipl -s code_English.prolog
% Janus 1.5.2 embeds Python 3.12.3 (main, Mar 3 2026, 12:15:18) [GCC 13.3.0]
Manual & tutorial: https://sandiway.arizona.edu Sandiway Fong (2024-2026)
Run: ?- parse(List,S0,L). (; for more answers)
      List = [..words] (input), S0,L = syntactic object, [..] spellout (output)
```

¹³ If unzip is not installed, do `sudo apt install unzip` first.

(parts of welcome banner omitted)

Welcome to SWI-Prolog (threaded, 64 bits, version 10.0.2)
?- parse([they,are,flying,planes],S0,L).

Words: they are flying planes

Initial WS: planes flying_θ v_pred:pres INFL_v:pl they

INT/EXT: {C, {INFL_v:pl, {v_pred:pres, {flying_θ, {planes, they}}}}}

EXT: {planes,[and],they} 3pl pres be flying

Spellout: {planes,[and],they} are flying

Blocked: inconsistent with input!

EXT: there 3pl pres be flying {planes,[and],they}

Spellout: there are flying {planes,[and],they}

Blocked: inconsistent with input!

INT/EXT: {C_Q_aux(pl,pres,be), {INFL_v:pl, {v_pred:pres, {flying_θ, {planes, they}}}}}

EXT: pl pres be → {planes,[and],they} 3pl pres be flying

Spellout: 3pl pres be {planes,[and],they} flying

Spellout: are {planes,[and],they} flying

Blocked: inconsistent with input!

EXT: pl pres be → there 3pl pres be flying {planes,[and],they}

Spellout: 3pl pres be there flying {planes,[and],they}

Spellout: are there flying {planes,[and],they}

Blocked: inconsistent with input!

Initial WS: planes flying_θ be_θ v_be:θ:pres INFL_v:pl they

INT/EXT: {C, {INFL_v:pl, {they, {v_be:θ:pres, {be_θ, {planes, {flying_θ, planes}}}}}}}

EXT: they 3pl pres be flying planes

Spellout: they are flying planes

Parse found!

S0 = {C, {INFL_v:pl, {they, {v_be:θ:pres, {be_θ, {planes, {flying_θ, planes}}}}}}},

L = [they,are,flying,planes] ;

INT/EXT: {C, {INFL_v:pl, {planes, {v_be:θ:pres, {be_θ, {they, {flying_θ, they}}}}}}}

Error: Case: object "they" not ACC

INT/EXT: {C, {INFL_v:pl, {{they, {flying_θ, they}}, {v_be:θ:pres, {be_θ, planes}}}}}

EXT: flying they 3pl pres be planes

Spellout: flying they are planes

Blocked: inconsistent with input!

INT/EXT: {C, {INFL_v:pl, {{planes, {flying_θ, planes}}, {v_be:θ:pres, {be_θ, they}}}}}

Error: Case: object "they" not ACC

Initial WS: planes fly_θ:presp v_fly:θ v_v:prog:pres INFL_v:pl they

INT/EXT: {C, {INFL_v:pl, {v_v:prog:pres, {they, {v_fly:θ, {fly_θ:presp, planes}}}}}}}

EXT: they 3pl pres be ing fly planes

Spellout: they are flying planes

Parse found!

S0 = {C, {INFL_v:pl, {v_v:prog:pres, {they, {v_fly:θ, {fly_θ:presp, planes}}}}}}},

L = [they,are,flying,planes] ;

EXT: there 3pl pres be ing they fly planes

Error: Morpheme "ing" unattached!
Blocked: inconsistent with input!
INT/EXT: {C_Q_aux(pl,pres,be), {INFL_v:pl, {v_v:prog:pres, {they, {v_fly:θ, {fly_θ:presp, planes}}}}}}}
EXT: pl pres be → they 3pl pres be ing fly planes
Spellout: 3pl pres be they ing fly planes
Spellout: are they flying planes
Blocked: inconsistent with input!
EXT: pl pres be → there 3pl pres be ing they fly planes
Spellout: 3pl pres be there ing they fly planes
Error: Morpheme "ing" unattached!
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v:pl, {v_v:prog:pres, {planes, {v_fly:θ, {fly_θ:presp, they}}}}}}}
Error: Case: object "they" not ACC
Error: Case: object "they" not ACC
INT/EXT: {C_Q_aux(pl,pres,be), {INFL_v:pl, {v_v:prog:pres, {planes, {v_fly:θ, {fly_θ:presp, they}}}}}}}
Error: Case: object "they" not ACC
Error: Case: object "they" not ACC
INT/EXT: {C, {INFL_v:pl, {v_v:prog:pres, {{planes, they}, {v_fly:θ, {fly_θ:presp, {planes, they}}}}}}}
EXT: {planes,[and],they} 3pl pres be ing fly {planes,[and],they}
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
EXT: there 3pl pres be ing {planes,[and],they} fly {planes,[and],they}
Error: Morpheme "ing" unattached!
Blocked: inconsistent with input!
INT/EXT: {C_Q_aux(pl,pres,be), {INFL_v:pl, {v_v:prog:pres, {{planes, they}, {v_fly:θ, {fly_θ:presp, {planes, they}}}}}}}
EXT: pl pres be → {planes,[and],they} 3pl pres be ing fly {planes,[and],they}
Spellout: 3pl pres be {planes,[and],they} ing fly {planes,[and],they}
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
EXT: pl pres be → there 3pl pres be ing {planes,[and],they} fly {planes,[and],they}
Spellout: 3pl pres be there ing {planes,[and],they} fly {planes,[and],they}
Error: Morpheme "ing" unattached!
Blocked: inconsistent with input!
false.

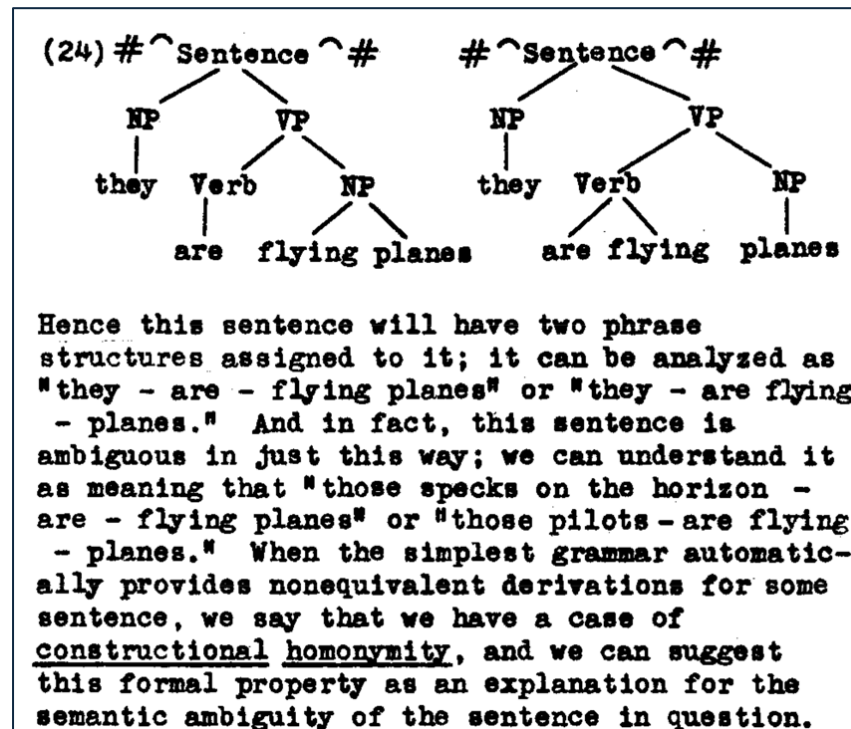
?- report.

The -s (script) flag tells swipl to load the file code_English.prolog from the current directory upon startup.

The Prolog command parse/3 above invokes the SMT Parser. It takes three arguments, (i) a list of words, (ii) the **SO**, named above as S0, a Prolog variable, and (iii) the list of words spelled out, named using Prolog variable L.

Parses are reported (one-at-a-time) inside Prolog. (The disjunctive semicolon (;) tells Prolog to search for other parses. Alternatively, press the <return> key to stop.) In the above example,

taken from (Chomsky 1956) – *see below*, two different **I-Language** structures compatible with the input sentence are built, capturing the intended structural ambiguity.



The command `report` produces a detailed summary for the parsing called `output.html` in the same directory. The file should automatically appear in the browser.¹⁴

 /home/sandiway/smtpars... +
   file:///home/sandiway/smtparser/english/output.html
 [Right triangle  may be expanded, down triangle  to close. Button  = tree pop-up. X = close pop-up.]
Terminal: [run](#). **Derivation Graphs:** (left-to-right) [compact](#) / [full](#); (top-down) [full](#) / [compact](#).
 ▶ Help: [blue](#) = parse inside. **Abbr.:** WS: Workspace; Initial WS: initial heads for Merge after LEX lookup.
Words: they are flying planes
 ▶ **Initial WS 1:** planes flying_θ v_{pred:pres} INFL_{v:pl} they
 ▶ **Initial WS 2:** planes flying_θ be_θ v_{be:θ:pres} INFL_{v:pl} they
 ▶ **Initial WS 3:** planes fly_{θ:presp} v_{fly:θ} v_{v:proq:pres} INFL_{v:pl} they

This Workspace can be explored by clicking on the triangles (►) and seeing which derivations leads to successful parses, i.e. converged on a single **SO**, and which are dead-ends (nonconvergent).

¹⁴ On Ubuntu, the default browser seems to be Firefox. (I see a ton of non-fatal errors messages before the window pops up. As they're non-fatal, I haven't tried to eliminate them. If you know why or how to make them go away, email me.)

There are two initial WSs that lead to parses (highlighted in **blue**) and one that does not (in **black**).

Example (*opening Initial WS 2 only*):

/home/sandiway/smtpars
+

← → ↺
file:///home/sandiway/smtparser/english/output.html

[Right triangle may be expanded, down triangle to close. Button = tree pop-up. = close pop-up.]

Terminal: [run](#). **Derivation Graphs:** (left-to-right) [compact](#) / [full](#); (top-down) [full](#) / [compact](#).

► Help: **blue** = parse inside. **Abbr.:** WS: Workspace; Initial WS: initial heads for Merge after LEX lookup.

Words: they are flying planes

► **Initial WS 1:** planes flying_θ v_{pred:pres} INFL_{v:pl} they

▼ **Initial WS 2:** planes flying_θ be_θ v_{be:θ:pres} INFL_{v:pl} they

► **WS 1:** {flying_θ, planes} be_θ v_{be:θ:pres} INFL_{v:pl} they

► **WS 1:** {flying_θ, they} planes be_θ v_{be:θ:pres} INFL_{v:pl}

► **WS 1:** {be_θ, {planes, {flying_θ, planes}}}} v_{be:θ:pres} INFL_{v:pl} they

INT/EXT: {C, {INFL_{v:pl}, {they, {v_{be:θ:pres}, {be_θ, {planes, {flying_θ, planes}}}}}}}}

EXT: they 3pl pres be flying planes

Spellout: they are flying planes

Parse found: they 3pl pres be flying planes

► **WS 1:** {be_θ, {planes, {flying_θ, planes}}}} v_{be:θ:pres} INFL_{v:pl} they

► **WS 1:** {be_θ, {they, {flying_θ, they}}}} planes v_{be:θ:pres} INFL_{v:pl}

INT/EXT: {C, {INFL_{v:pl}, {planes, {v_{be:θ:pres}, {be_θ, {they, {flying_θ, they}}}}}}}}

Error: Case: object "they" not ACC

Note that the potential parse *planes are flying they* is rejected at EXT as pronominal *they* is incompatible with Accusative (ACC) Case.

Example (*opening Initial WS 3 only*):

```

/home/sandiway/smtpars...
file:///home/sandiway/smtparser/english/output.html

[Right triangle may be expanded, down triangle to close. Button = tree pop-up. X = close pop-up.]

Terminal: run. Derivation Graphs: (left-to-right) compact / full; (top-down) full / compact.

► Help: blue = parse inside. Abbr.: WS: Workspace; Initial WS: initial heads for Merge after LEX lookup.

Words: they are flying planes
► Initial WS 1: planes flyingθ vpred:pres INFLv:pl they
► Initial WS 2: planes flyingθ beθ vbe:θ:pres INFLv:pl they
▼ Initial WS 3: planes flyθ:presp vfly:θ vv:prog:pres INFLv:pl they
  ► WS 1: {flyθ:presp, planes} vfly:θ vv:prog:pres INFLv:pl they
  INT/EXT: {C, {INFLv:pl, {vv:prog:pres, {they, {vfly:θ, {flyθ:presp, planes}}}}}}
  EXT: they 3pl pres be ing fly planes
  Spellout: they are flying planes
  Parse found: they 3pl pres be ing fly planes
  EXT: there 3pl pres be ing they fly planes
  Blocked: inconsistent with input!
  Final WS: {CQ:aux(pl,pres,be), {INFLv:pl, {vv:prog:pres, {they, {vfly:θ, {flyθ:presp, planes}}}}}}
  INT/EXT: {CQ:aux(pl,pres,be), {INFLv:pl, {vv:prog:pres, {they, {vfly:θ, {flyθ:presp, planes}}}}}}
  EXT: pl pres be → they 3pl pres be ing fly planes
  Spellout: 3pl pres be they ing fly planes
  Spellout: are they flying planes
  Blocked: inconsistent with input!

```

Note that the potential parse *are they flying planes?* is rejected at EXT as the word order does not match the input sentence.

2. Tutorial: Background

Three examples of parsing, one in Spanish and two in English, have been given in the previous section. **I-Language** structures are built bottom-up in the **Workspace** (WS) via **Merge**, starting with a selection of lexical and functional heads. That initial set of heads is introduced through parser input triggering the addition of a cascade of heads into the **WS**. Some heads will have θ -theoretic properties, e.g. a verbal root or applicative selecting for an internal argument, a v selecting for an external argument, prompting the construction of a θ -configuration. Some “heads” do not project structure in the theory implemented, e.g. a determiner or nonfinite *to* in English, and *a* in Spanish; instead, these are analyzed as the spellout of features from a nearby head noun.

The hand-built English lexicon is in file `lex.prolog`, and `lex_Spanish.prolog` contains the Spanish lexicon. In this document, we will call the active lexicon **LEX**. **LEX** basically determines the available heads (and their properties) for a given language. (Currently, language-particular **EXT** properties are also stored in **LEX** – this is subject to change in a later release.)

Currently, the hand-built English **LEX** is supplemented by wordnet. If a word does not exist in **LEX**, wordnet is consulted for a default entry when it comes open class words, e.g. if it is a verb, by default, it is considered to be transitive. If it is a noun (or adjective), it’s assumed to be a

singular common noun (or a single argument-taking predicate), respectively. The Spanish lexicon is entirely hand-built (and does not use wordnet).

Assuming you're in the same directory as the file, to load **LEX** at any time, e.g. when you modify the file, type the following at the Prolog prompt (**?-**):

```
?- [lex].
```

or

```
?- [lex_Spanish].
```

After editing LEX, you should always reload it and see if there are errors, but you do not need to reload the entire parser via `code_English.prolog` or `code_Spanish.prolog`.

At startup, loading `code_Language` will automatically load the appropriate **LEX**.

3. Tutorial: A Worked Example in English

Let's add words to **LEX** to handle the following structurally ambiguous sentence from (Berwick & Chomsky, 2016):

(1) Birds that fly instinctively swim

In Terminal, `cd` to the parser directory, start `swipl` and load in the parser (using the shorthand `[file]` notation, where *file* is the name of the code without the file extension, typically, `.pl` or `.prolog`).

```
$ swipl
?- [code_English].
```

Alternatively, specify the full filename (including the file extension) using the `-s` command line option as follows:

```
$ swipl -s code_English.prolog
?-
```

(Note: `code_English.prolog` above should be `code_Spanish.prolog` in the case of Spanish.)

If we try our example sentence:

```
?- parse([birds,that,fly,instinctively,swim],S0,L).
```

Error: 'birds' not found! Edit LEX or useMorphy. or add{Noun/Tr/Adj} etc.
false.

the parser reports an error.

(Note: the LING 696A file `lex.prolog` has been artificially gimped for this exercise.)

Consult the Reference Manual section for an explanation of each of the queries used in the worked example.

If you are unfamiliar with Prolog syntax or operation of the interpreter, a brief review is available in the SWI-Prolog quickstart section (www.swi-prolog.org/pldoc/man?section=quickstart).

Adding a noun: θ R/1, num/2 and word2heads/3

First, make sure you are using a (pure) text editor on your platform. On macOS and Linux, the built-in command `nano lex.prolog` can be used to edit the file `lex.prolog`.¹⁵ On Windows 11, similarly Microsoft Edit is available by default, use `edit lex.prolog`.

There are three predicates θ R/1, num/2 and word2heads/3 that we need to make sure are working correctly.

Nouns are θ -relevant heads. **LEX** has a table θ R/1 (predicate name θ R – θ -relevant – with arity 1) listing these heads. For example, we might expect *bird* (noun) to be present.¹⁶

```
?-  $\theta$ R(bird).  
false.
```

But `false` means *bird* is not present in this list. To add the missing θ -relevant word *bird*, and its plural form *birds*, edit **LEX** and add the entry:

```
 $\theta$ R(bird).
```

[Currently, the entry exists but is commented out using the Prolog comment character `%`.]

(Note: new entries should be placed next to other θ R/1 entries otherwise Prolog will emit a *discontiguous predicate* warning. This is a programming style check, not an error.)

Reload LEX and test:

```
?- [lex].  
true.  
  
?-  $\theta$ R(bird).  
true.
```

¹⁵ Nano is a bare-bones text editor that is included with both macOS and Ubuntu. See nano-editor.org. Full disclosure: I am an Aquamacs fan on macOS, see aquamacs.org.

¹⁶ You may have problems typing Greek characters such as θ . If so, maybe easiest to simply copy-paste from this manual. On macOS, I switch between various *Input Source* keyboard layouts using the Globe key, e.g. between English, Pinyin Chinese, Greek and Hiragana. (θ = u and ϕ = f.) Windows 11 has a similar facility and a legacy ALT code system that I don't know much about. (ALT + 237 = ϕ seems to work, but somehow 952 doesn't work and I need to use charmap for θ .)

Next, test the NUM plural (pl) feature value.¹⁷ Because of its regular morphology, we only need to define the singular form of the common noun. (A rule kicks in to add the default suffix -s.)

```
?- num(bird,pl).  
false.
```

```
?- num(birds,pl).  
true.
```

The Reference section for num/2 describes how to enter an irregular plural form (ahead of regular morphology).

An *Initial Workspace* (WS) will be populated (for **Merge**) by calling word2heads/3 for each word in the input. Test it. Type semicolon (;) if it pauses to obtain other possible answers. If useMorphy is active, we obtain two answers:

```
?- word2heads(birds,Hd,As).  
Hd = [birds],  
As = [] ;  
Hd = [bird_θ, v_bird:θ:pres, INFL_v:3sg],  
As = [] ;  
false.
```

Note, it allows *bird* to be a transitive verb as well. For the inflected form *birds*, it returns ϕ -feature 3sg and TNS feature present (pres). This is because WordNet LEX is active in the background by default. Prolog calls Python using the equivalent of the following code:

```
>>> from nltk.corpus import wordnet as wn  
>>> wn.morphy('birds','v')  
'bird'
```

This means, according to WordNet, *bird* is a verb too. **LEX** will assign verbs transitive status by default. See vStem/3 below.

We can turn off WordNet **LEX** using noMorphy. Then *birds* is only a common noun.

```
?- vStem(birds,V,Suffix).  
V = bird,  
Suffix = s.
```

```
?- noMorphy.  
true.
```

```
?- word2heads(birds,Hd,As).  
Hd = [birds],  
As = [] ;  
false.
```

¹⁷ Be aware that the Spanish lexicon uses a different predicate, num/3. The third argument will be the singular form, e.g. num(aves,pl,ave).

From here on we proceed assuming WordNet LEX has been turned off.

Note: due to rich inflectional morphology, for Spanish, we use vStem/4 instead of English vStem/3. Example: *anduve* (*walk*) has properties:

```
?- vStem(anduve,Root,TNS,AGR).  
Root = andar,  
TNS = past,  
AGR = 1sg.
```

Fly

The verb *fly* is unergative in worked example (1) above. However, LEX has *fly* listed as transitive only as there are two θ -roles listed, one is the complement of *fly*, indicated by verbal root *fly θ* , the other is associated with *v*, indicated by *vfly: θ :pres*. Note there are two entries for the form *fly* as it can be untensed or have TNS feature *pres*.

```
?- word2heads(fly,Hd,As).  
Hd = [fly_ $\theta$ , v_fly: $\theta$ :pres, INFL_v],  
As = [] ;  
Hd = [fly_ $\theta$ , v_fly: $\theta$ , INFL_v],  
As = [] ;  
false.
```

To modify the argument-taking properties of *fly*, examine the table vR/3 in LEX, which defines the θ -properties *v* ("little *v*") and *R* (Root) for each verb.

```
vR(fly, v:fly: $\theta$ ,  $\theta$ ).           % they are flying planes
```

We can look for and replicate the vR/3 entry associated with an unergative verb like *dance*. The empty set symbol ($\emptyset$) indicates that the verb root (*R*) does not have a complement.

```
vR(dance, v:dance: $\theta$ ,  $\emptyset$ ).      % Mary dances  
vR(fly, v:fly: $\theta$ ,  $\emptyset$ ).        % birds fly
```

Reload LEX and test. Notice we have 4 forms now, two transitive, two unergative.

```
?- word2heads(fly,Hd,As).  
Hd = [fly_ $\theta$ , v_fly: $\theta$ :pres, INFL_v],  
As = [] ;  
Hd = [fly_ $\emptyset$ , v_fly: $\theta$ :pres, INFL_v],  
As = [] ;  
Hd = [fly_ $\theta$ , v_fly: $\theta$ :nonf],  
As = [] ;  
Hd = [fly_ $\emptyset$ , v_fly: $\theta$ :nonf],  
As = [] ;  
false.
```

Note the WordNet LEX also lists *fly* as a noun. We will generally have additional entries if noMorphy/ $\emptyset$ is not turned on.

Let's test *birds fly*, using `parse/3`. The variable `S0` will hold the value of the convergent syntactic object.

```
?- parse([birds,fly],S0,L).
```

Words: birds fly

Initial WS: fly_θ v_fly:θ:pres INFL_v birds

INT/EXT: {C, {INFL_v, {birds, {v_fly:θ:pres, {fly_θ, birds}}}}}

EXT: birds 3pl pres fly birds

Blocked: EXT inconsistent!

Blocked: inconsistent with input!

Initial WS: fly_θ v_fly:θ:pres INFL_v birds

INT/EXT: {C, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}

EXT: birds 3pl pres fly

Spellout: birds fly

Parse found!

`S0 = {C, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}},`

`L = [birds,fly] ;`

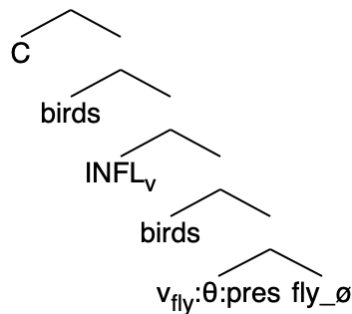
Initial WS: fly_θ v_fly:θ:nonf birds

Initial WS: fly_θ v_fly:θ:nonf birds

false.

Let us briefly examine the parse below produced by an earlier version of the SMT Parser (*no longer available for download*). The root is non-complement-taking `fly_θ`, and the head `v` selects for the root *fly*, second-selects for a θ -relevant **WS** item, and has feature **TNS** with value *pres*, as indicated by head `v_fly:θ:pres`.

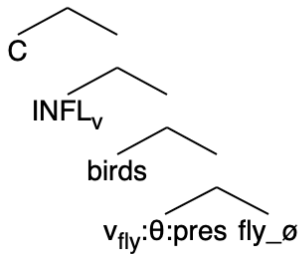
`S0 = {C, {birds, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}}`



FormCopy ensures only the top copy of *birds* at the surface subject position (spec-INFL) is pronounced, the lower copy in the external θ -position associated with *v* is necessary for **INT**.

The 696A SMT Parser does away with the spec-INFL position. Instead, the surface subject is simply spelled out at the left edge of INFL at **EXT**, an option available in the case of English.

`S0 = {C, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}`



INFL triggers **Minimal Search** (MS) for a θ -relevant item in its c-command domain. It finds *birds* in the external argument position. At **EXT**, in the case of English this licenses INFL to spell out *birds* at its left edge. The **SO** computed is simpler (than in the older version).

Let's also check LEX for *swim* as a verb. In this case, using word2heads/3, we see *swim* is already present as an unergative verb.

```

?- word2heads(swim,Hd,As).
Hd = [swim_ø,v_swim:θ:pres,INFL_v],
As = [] ;
Hd = [swim_ø,v_swim:θ:nonf],
As = [] ;
false.

```

This is confirmed by the vR/3 lexical entry:

```
vR(swim, v:swim:θ, ø).           % eagles swim
```

that

In this implementation, relative clauses are introduced by a relative head C_{rel} . If you look in **LEX** for relevant information concerning ' C_{rel} ', you will find a definition for extOnlyWord/2.

```

:- dynamic extOnlyWord/2.

extOnlyWord(who,['C_rel':word(who)]). % ambiguous between noun and relative
pronoun
extOnlyWord(whom,['C_rel':word(whom)]).
extOnlyWord(that,['C_rel':word(that)]).

```

The listing in extOnlyWord/2 means the parser can map an input word *that* directly into the head ' C_{rel} '. Moreover, at **EXT**, the head C_{rel} may be spelled out as *that*. The feature word(*that*) is not used in **Merge**, but will be used by **EXT** in the SMT Parser to match the input word.

Using word2heads/3, introduced earlier, we see:

```

?- word2heads(that,Hs,As).
Hs = [C_rel_that],
As = [] ;

```

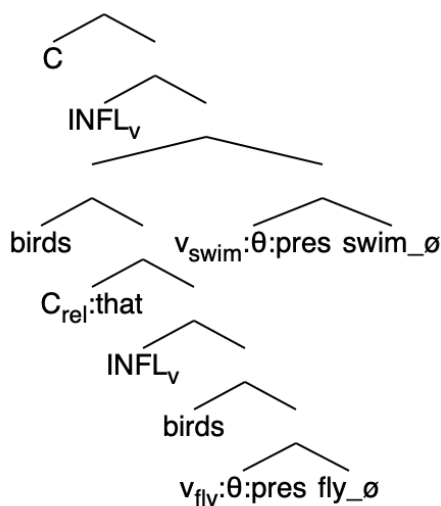
false.

We can check **LEX** lookup for the entire sentence using words2heads/4. (We omit description of empty linear actions As (= []) below and the last parameter (= 1) for expository simplicity.) The correct *Initial WS* highlighted below is the one that has both *fly* and *swim* as unergatives with TNS pres. (We currently assume both the relative clause and matrix clause must contain tensed verbs. This should be updated to permit untensed relative clauses as long as nonfinite marker *to* is present in English, e.g. *the desire to fly*.)

```
?- words2heads([birds,that,fly,swim],As,WS,1).
1. WS = [swim_ø,v_swim:θ:pres,INFL_v,fly_ø,v_fly:θ:pres,INFL_v,C_rel_that,birds] ;
2. WS = [swim_ø,v_swim:θ:nonf,fly_ø,v_fly:θ:pres,INFL_v,C_rel_that,birds] ;
3. WS = [swim_ø,v_swim:θ:pres,INFL_v,fly_ø,v_fly:θ:pres,INFL_v,C_rel_that,birds] ;
4. WS = [swim_ø,v_swim:θ:nonf,fly_ø,v_fly:θ:pres,INFL_v,C_rel_that,birds] ;
5. WS = [swim_ø,v_swim:θ:pres,INFL_v,fly_ø,v_fly:θ:nonf,C_rel_that,birds] ;
6. WS = [swim_ø,v_swim:θ:nonf,fly_ø,v_fly:θ:nonf,C_rel_that,birds] ;
7. WS = [swim_ø,v_swim:θ:pres,INFL_v,fly_ø,v_fly:θ:nonf,C_rel_that,birds] ;
8. WS = [swim_ø,v_swim:θ:nonf,fly_ø,v_fly:θ:nonf,C_rel_that,birds] ;
false.
```

The parse generated through Initial WS #3 above is the intended SO:

{C, {INFL_v, {{birds, {C_rel:that, {INFL_v, {birds, {v_fly:θ:pres, fly_ø}}}}, {v_swim:θ:pres, swim_ø}}}}}



For completeness, here is the Terminal output of the parse/3 command:

```
?- parse([birds,that,fly,swim],S0,L).
```

Words: birds that fly swim

Initial WS: swim_ø v_swim:θ:pres INFL_v fly_ø v_fly:θ:pres INFL_v
C_rel_that birds

INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_ø}}}}, {v_fly:θ:pres, {fly_ø, birds}}}}}

EXT: birds that 3pl pres swim 3pl pres fly birds

Blocked: EXT inconsistent!

Blocked: inconsistent with input!

INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_fly:θ:pres, {fly_θ, birds}}}}}}, {v_swim:θ:pres, swim_θ}}}}}
EXT: birds that 3pl pres fly birds 3pl pres swim
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_fly:θ:pres, {fly_θ, birds}}}}}}, {v_swim:θ:pres, swim_θ}}}}}
EXT: birds that 3pl pres fly birds 3pl pres swim
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_θ}}}}}, {v_fly:θ:pres, {fly_θ, birds}}}}}}}
EXT: birds that 3pl pres swim 3pl pres fly birds
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_θ}}}}}, {v_fly:θ:pres, {fly_θ, {birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_θ}}}}}}}}}}}
EXT: birds that 3pl pres swim 3pl pres fly birds that 3pl pres swim
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v, {birds, {v_fly:θ:pres, {fly_θ, {birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_θ}}}}}}}}}}}
EXT: birds 3pl pres fly birds that 3pl pres swim
Blocked: EXT inconsistent!
Blocked: inconsistent with input!
Initial WS: swim_θ v_swim:θ:nonf fly_θ v_fly:θ:pres INFL_v C_rel_that birds
Initial WS: swim_θ v_swim:θ:pres INFL_v fly_θ v_fly:θ:pres INFL_v C_rel_that birds
INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}}, {v_swim:θ:pres, swim_θ}}}}}
EXT: birds that 3pl pres fly 3pl pres swim
Spellout: birds that fly swim
Parse found!
S0 = {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}}, {v_swim:θ:pres, swim_θ}}}},
L = [birds,that,fly,swim] ;

INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_θ}}}}}, {v_fly:θ:pres, fly_θ}}}}}
EXT: birds that 3pl pres swim 3pl pres fly
Spellout: birds that swim fly
Blocked: inconsistent with input!
Initial WS: swim_θ v_swim:θ:nonf fly_θ v_fly:θ:pres INFL_v C_rel_that birds
Initial WS: swim_θ v_swim:θ:pres INFL_v fly_θ v_fly:θ:nonf C_rel_that birds
Initial WS: swim_θ v_swim:θ:nonf fly_θ v_fly:θ:nonf C_rel_that birds
Initial WS: swim_θ v_swim:θ:pres INFL_v fly_θ v_fly:θ:nonf C_rel_that birds
Initial WS: swim_θ v_swim:θ:nonf fly_θ v_fly:θ:nonf C_rel_that birds
false.

The post-parse command `report .` will produce the compact and browser-friendly summary shown below, with ▶ which can be expanded (by clicking). *Initial WS 3* is highlighted in **blue**, as it led to a parse. All other Initial WSs failed.

[Right triangle ▶ may be expanded, down triangle ▼ to close. Button = tree pop-up. ✕ = close pop-up.]

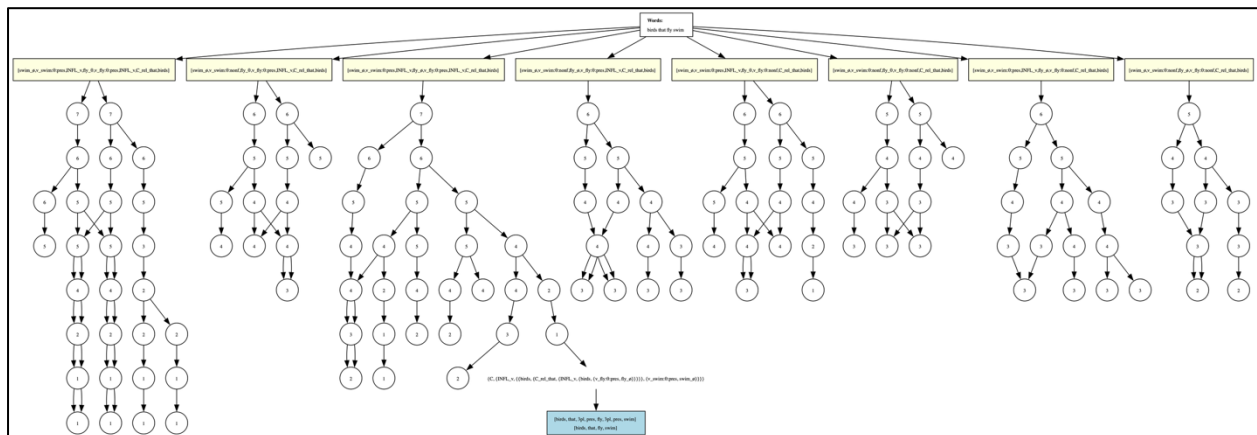
Terminal: [run](#). **Derivation Graphs:** (left-to-right) [compact](#) / [full](#); (top-down) [full](#) / [compact](#).

▶ **Help:** **blue** = parse inside. **Abbr.:** WS: Workspace; Initial WS: initial heads for Merge after LEX lookup.

Words: birds that fly swim

▶ Initial WS 1: swim _∅	v _{swim:0:pres}	INFL _v	fly _∅	v _{fly:0:pres}	INFL _v	C _{rel:that}	birds
▶ Initial WS 2: swim _∅	v _{swim:0:nonf}	fly _∅	v _{fly:0:pres}	INFL _v	C _{rel:that}	birds	
▶ Initial WS 3: swim _∅	v _{swim:0:pres}	INFL _v	fly _∅	v _{fly:0:pres}	INFL _v	C _{rel:that}	birds
▶ Initial WS 4: swim _∅	v _{swim:0:nonf}	fly _∅	v _{fly:0:pres}	INFL _v	C _{rel:that}	birds	
▶ Initial WS 5: swim _∅	v _{swim:0:pres}	INFL _v	fly _∅	v _{fly:0:nonf}	C _{rel:that}	birds	
▶ Initial WS 6: swim _∅	v _{swim:0:nonf}	fly _∅	v _{fly:0:nonf}	C _{rel:that}	birds		
▶ Initial WS 7: swim _∅	v _{swim:0:pres}	INFL _v	fly _∅	v _{fly:0:nonf}	C _{rel:that}	birds	
▶ Initial WS 8: swim _∅	v _{swim:0:nonf}	fly _∅	v _{fly:0:nonf}	C _{rel:that}	birds		

The computational graph below shows the complete search space for this example. Each node is a **WS** and each arrow means a **SBO** fired. Only the third Initial WS (counting from left to right) produces a parse (highlighted in **blue**).



Instinctively

Finally, we need to add the adverb *instinctively* to **LEX**.

```
?- word2heads(instinctively,Hds,As).
false.
```

Adverbs are listed by Prolog predicate `advWord/2`.

```
advWord(carefully,v+_) .           % + permits unkeyed match
advWord(furiously,v+_) .
```

We assume adverbs must seek a verb to modify, in the current theory these are vP selecting adverbs. The second parameter is `v+_`, this means it selects for a phrase headed by `v+_`.

Implementation Detail: Verbs are inserted into an **Initial WS** as a closely connected *cluster of heads*, usually `v` and the verb root, possibly along with `INFL`. These heads are clustered together in the **WS** in the sense that clustered heads may not interfere with heads from other verbs. For instance, suppose transitive *hit* and unaccusative *arrive* occur in the same **WS**. `v` from *hit* selects for an external argument and may not combine with root *arrive*, e.g. to produce **John arrived a train*. The notation `v+_` means that adverbs should combine with but are not keyed to a particular verb cluster. The underscore (`_`) is a Prolog anonymous key (variable) indicating that we permit the adverb to match any vP. [\[Discussion should be moved to a new section.\]](#)

We add:

```
advWord(instinctively, v+_).
```

and reload LEX, checking the entry using `word2heads/2`.

```
?- [lex].
true.
```

```
?- word2heads(instinctively,Hds,As).
Hds = [instinctively_v],
As = [] ;
false.
```

Then we parse the complete sentence as follows:

```
?- parse([birds,that,fly,instinctively,swim],S0,L).
```

```
Words: birds that fly instinctively swim
Initial WS: swim_0 v_swim:0:pres INFL_v instinctively_v fly_0 v_fly:0:pres
INFL_v C_rel_that birds
""
Initial WS: swim_0 v_swim:0:nonf instinctively_v fly_0 v_fly:0:pres INFL_v
C_rel_that birds
Initial WS: swim_0 v_swim:0:pres INFL_v instinctively_v fly_0 v_fly:0:pres
INFL_v C_rel_that birds
INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {instinctively_v,
{birds, {v_swim:0:pres, swim_0}}}}}}, {v_fly:0:pres, fly_0}}}}
EXT: birds that instinctively 3pl pres swim 3pl pres fly
Spellout: birds that instinctively swim fly
Blocked: inconsistent with input!
EXT: birds that 3pl pres swim instinctively 3pl pres fly
Spellout: birds that swim instinctively fly
Blocked: inconsistent with input!
```

INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {instinctively_v, {birds, {v_fly:θ:pres, fly_θ}}}}}}, {v_swim:θ:pres, swim_θ}}}}}
EXT: birds that instinctively 3pl pres fly 3pl pres swim
Spellout: birds that instinctively fly swim
Blocked: inconsistent with input!
EXT: birds that 3pl pres fly instinctively 3pl pres swim
Spellout: birds that fly instinctively swim
Parse found!
S0 = {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {instinctively_v, {birds, {v_fly:θ:pres, fly_θ}}}}}}, {v_swim:θ:pres, swim_θ}}}},
L = [birds,that,fly,instinctively,swim] ;

INT/EXT: {C, {INFL_v, {instinctively_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}}}, {v_swim:θ:pres, swim_θ}}}}}
EXT: birds that 3pl pres fly instinctively 3pl pres swim
Spellout: birds that fly instinctively swim
Parse found!
S0 = {C, {INFL_v, {instinctively_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_fly:θ:pres, fly_θ}}}}}}, {v_swim:θ:pres, swim_θ}}}},
L = [birds,that,fly,instinctively,swim] ;

EXT: birds that 3pl pres fly 3pl pres swim instinctively
Spellout: birds that fly swim instinctively
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v, {instinctively_v, {{birds, {C_rel_that, {INFL_v, {birds, {v_swim:θ:pres, swim_θ}}}}}}, {v_fly:θ:pres, fly_θ}}}}}
EXT: birds that 3pl pres swim instinctively 3pl pres fly
Spellout: birds that swim instinctively fly
Blocked: inconsistent with input!
EXT: birds that 3pl pres swim 3pl pres fly instinctively
Spellout: birds that swim fly instinctively
Blocked: inconsistent with input!
INT/EXT: {C, {INFL_v, {{birds, {C_rel_that, {INFL_v, {instinctively_v, {birds, {v_swim:θ:pres, swim_θ}}}}}}, {v_fly:θ:pres, fly_θ}}}}}
EXT: birds that instinctively 3pl pres swim 3pl pres fly
Spellout: birds that instinctively swim fly
Blocked: inconsistent with input!
EXT: birds that 3pl pres swim instinctively 3pl pres fly
Spellout: birds that swim instinctively fly
Blocked: inconsistent with input!
Initial WS: swim_θ v_swim:θ:nonf instinctively_v fly_θ v_fly:θ:pres INFL_v C_rel_that birds
Initial WS: swim_θ v_swim:θ:pres INFL_v instinctively_v fly_θ v_fly:θ:nonf C_rel_that birds
Initial WS: swim_θ v_swim:θ:nonf instinctively_v fly_θ v_fly:θ:nonf C_rel_that birds
Initial WS: swim_θ v_swim:θ:pres INFL_v instinctively_v fly_θ v_fly:θ:nonf C_rel_that birds
Initial WS: swim_θ v_swim:θ:nonf instinctively_v fly_θ v_fly:θ:nonf C_rel_that birds
false.

The post-parse operation report. produces the following summary:

file:///Users/sandiway/Desktop/coding/smtparser/code/output.ht

[Right triangle ► may be expanded, down triangle ▼ to close. Button ⌵ = tree pop-up. ✕ = close pop-up.]

Terminal: [run](#). **Derivation Graphs:** (left-to-right) [compact](#) / [full](#); (top-down) [full](#) / [compact](#).

► **Help:** [blue](#) = parse inside. **Abbr.:** WS: Workspace; Initial WS: initial heads for Merge after LEX lookup.

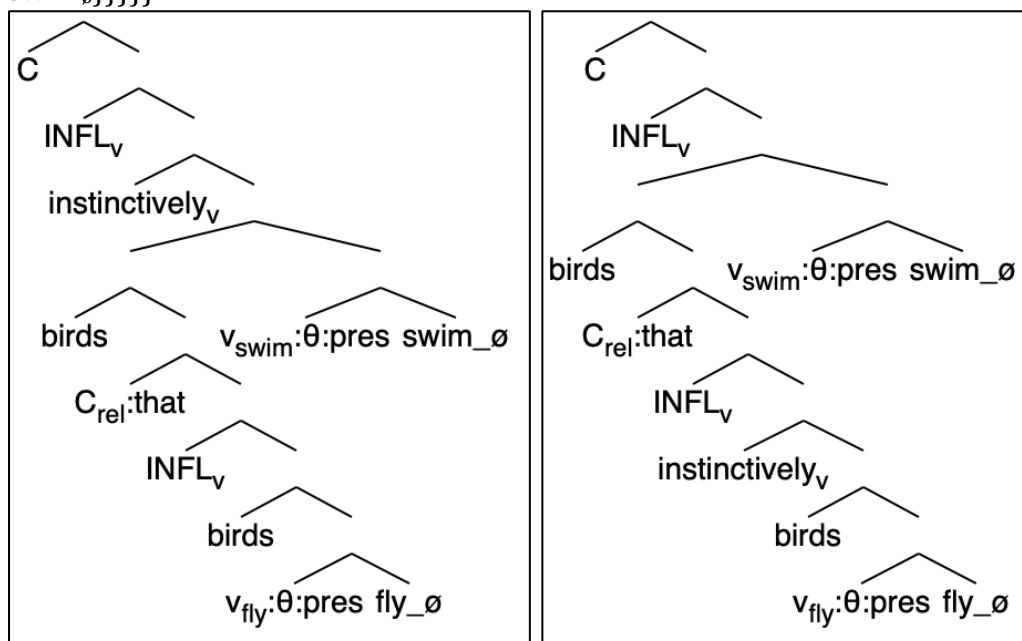
Words: birds that fly instinctively swim

► Initial WS 1:	swim _∅	v _{swim:∅:pres}	INFL _v	instinctively _v	fly _∅	v _{fly:∅:pres}	INFL _v	C _{rel:that}	birds
► Initial WS 2:	swim _∅	v _{swim:∅:nonf}	instinctively _v	fly _∅	v _{fly:∅:pres}	INFL _v	C _{rel:that}	birds	
► Initial WS 3:	swim _∅	v _{swim:∅:pres}	INFL _v	instinctively _v	fly _∅	v _{fly:∅:pres}	INFL _v	C _{rel:that}	birds
► Initial WS 4:	swim _∅	v _{swim:∅:nonf}	instinctively _v	fly _∅	v _{fly:∅:pres}	INFL _v	C _{rel:that}	birds	
► Initial WS 5:	swim _∅	v _{swim:∅:pres}	INFL _v	instinctively _v	fly _∅	v _{fly:∅:nonf}	C _{rel:that}	birds	
► Initial WS 6:	swim _∅	v _{swim:∅:nonf}	instinctively _v	fly _∅	v _{fly:∅:nonf}	C _{rel:that}	birds		
► Initial WS 7:	swim _∅	v _{swim:∅:pres}	INFL _v	instinctively _v	fly _∅	v _{fly:∅:nonf}	C _{rel:that}	birds	
► Initial WS 8:	swim _∅	v _{swim:∅:nonf}	instinctively _v	fly _∅	v _{fly:∅:nonf}	C _{rel:that}	birds		

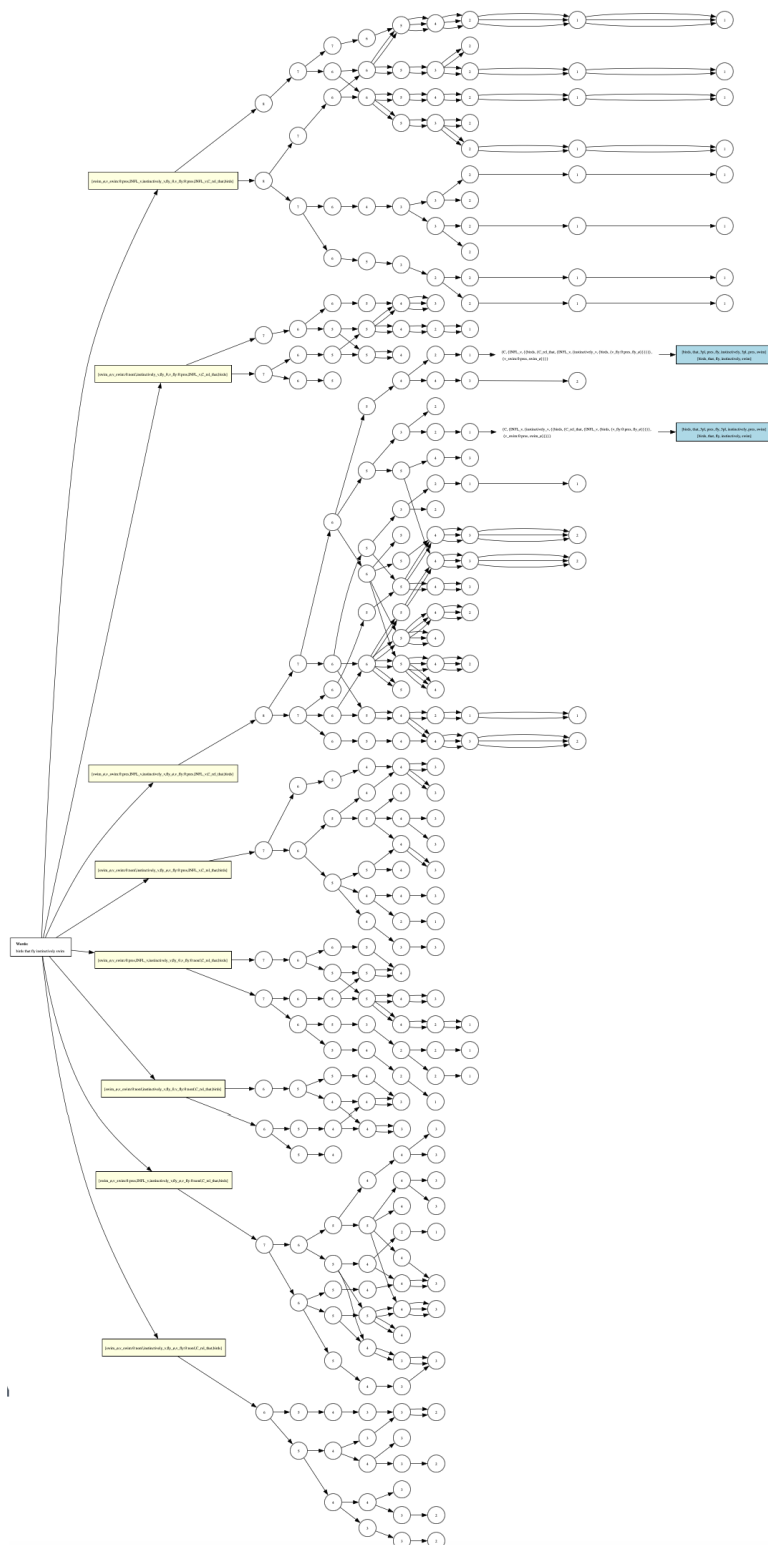
indicating that only *Initial WS 3* converges. In fact, *Initial WS 3* converges twice (due to variable placement of the adverb) and we obtain the following two parses:

{C, {INFL_v, {{birds, {C_{rel:that}, {INFL_v, {instinctively_v, {birds, {v_{fly:∅:pres}, fly_∅}}}}}}, {v_{swim:∅:pres}, swim_∅}}}}}

{C, {INFL_v, {instinctively_v, {{birds, {C_{rel:that}, {INFL_v, {birds, {v_{fly:∅:pres}, fly_∅}}}}}}, {v_{swim:∅:pres}, swim_∅}}}}}



The computational graph below shows the complete search space for this example. Only the third *Initial WS* (counting from top to bottom) converged with two parses (highlighted in blue). Compare with the earlier graph shown for the same sentence without the adverb *instinctively*.



4. Tutorial: A Worked Example in Spanish

Will be on Spanish participles.

See for now Section 5.5 Verbs: passive, progressive, perfective (module **LEX**).

[to come]

[The following orphaned sections should be moved to and integrated with **Section 5. Reference Manual: LEX.**]

Nouns

θ -relevant

Nouns are θ -relevant and selected by heads with *complement* (1st **Merge**) or *specifier* (2nd **Merge**) feature θ .¹⁸ Note that expletive *there* is not considered to be a noun in the 696A SMT Parser as it is not θ -relevant. Instead it is purely a Spellout option for the left edge of INFL available only to English. In fact, expletive *there* does not have a presence in the **WS**.¹⁹ With respect to θ -relevant nouns, we define the table $\theta R/1$. Sample entries:

$\theta R(ball).$
 $\theta R(boat).$
 $\theta R(bombing).$
 $\theta R(car).$
 $\theta R(corridor).$
 $\theta R(city).$

NUM feature

θ -relevant nouns are assumed to be grammatically singular unless otherwise defined. Plurals are defined using $num/2$. To determine whether a noun is plural we enlist WordNet morphy for stemming.

```
morphyStem(Noun,n,Root)
?- morphyStem(ball,n,Root).      Root = ball.
?- morphyStem(balls,n,Root).    Root = ball.
```

Then if $ball + s = balls$, we infer *balls* is plural.

We also have a separate (English-particular) rule implemented that states if a noun ends in -y, the plural form ends in -ies.

¹⁸ We use the traditional terms *complement* and *specifier* for convenience only. They are ill-defined in the current theory, e.g. specifier is related to properties of the head and difficult to define precisely, e.g. the relation *specifier-of* given the logical (?) possibility of stacking multiple specifiers and other cases. There is also no spec-head configuration computation in the theory, so perhaps the term *specifier* can be obsoleted/deprecated. [To be revised.]

¹⁹ Expletive *there* does correlate with indefinite NPs, e.g. for constructions such as *there is a/*the man in the room*. See my *Modern Language Society* January 2026 talk slides for relevant discussion (when other languages come into the picture). Currently, the SMT Parser has no (universal) **I-Language** mechanism to deal with these effects. [Still looking for something.]

```
?- morphyStem(lady,n,Root).      Root = lady.
?- morphyStem(ladies,n,Root).    Root = lady.
```

That is, *lady* - *y* + *ies* = *ladies*.

ϕ -features

$\phi(N, \text{Phi})$ returns ϕ -feature values Phi for a noun N . θ -relevant nouns are assumed to be 3sg unless otherwise defined. A noun N that satisfies $\text{num}(N, \text{pl})$ is 3pl.

Pronouns ϕ -feature values are specified using $\text{per}/2$. These definitions will be accessed by $\phi/2$. Sample entries:

```
per('I', '1sg').
per(we, '1pl').
per(us, '1pl').
per(ourselves, '1pl').
per(you, '2pl').
per(yourselves, '2pl').
per(they, '3pl').
per(them, '3pl').
per(themselves, '3pl').
```

Case

Case is only checked at EXT, not part of Merge syntax. Define restrictions on Case using $\text{caseR}/2$. If not defined here, assume unrestricted. Sample entries.

```
caseR('I', nom).
caseR(me, acc).
caseR(we, nom).
caseR(us, acc).
caseR(they, nom).
caseR(them, acc).
caseR(whom, acc).      % tricky: whom did John see?
```

wh-feature

wh-relevant nouns are defined using $\text{whR}/1$. Sample entries:

```
whR(what).
whR(who).
whR(whom).
```

Other features on nouns

For English relative clause formation, the animacy property seems to select between relative pronouns *who* and *that* for many speakers.

The man/mechanic who I saw

**The man/mechanic that I saw*

The shop that I saw

**The shop who I saw*

Define animate/1. Sample entries:

```
animate(old).
animate(man).
animate(mechanic).
animate(men).
animate(woman).
animate(women).
```

Verbs: introducing the verbal cluster

A cluster of heads is inserted into the Initial Workspace (**WS**) when the parser recognizes a verb.

For example, the copula verb form *is* is analyzed as the pair $\text{INFL}_{V:3sg} - \text{V}_{pred:pres}$. The head INFL here has ϕ -features 3sg and selects for a phrase headed by the categorizer *v*, which, in turn, selects for a phrase headed by a predicate (*pred*), e.g. *criminal*, as in Chomsky's example *the bombing of the cities is criminal*. *v* also has TNS feature *pres* (present). Note the copula has no θ -selection properties. Compare this with a regular verb such as *liked*, which is associated with the cluster $\text{INFL}_{V-V_{like:\theta:pst}} - \text{like}_{\theta}$. The categorizer *v*, here with past TNS (*pst*) and selecting for root *like*, and *like* take a θ -relevant **WS** item at 2nd and 1st **Merge**, respectively.

The first step the parser takes is to stem the inflected verb form (using WordNet morphy), as in (a-i) and (b-i) below. For *is* and *liked*, the suffix is *-s* and *-d*, which are associated with TNS feature *pres* and *pst* (to be attached to *v*), respectively. In (a-iii) and (b-iii), *L* is the list of heads that enter the Initial WS on behalf of *is* and *liked*.

- | | |
|---|---|
| a. | b. |
| i. ?- $\text{vStem}(is, V, S)$. $V=be, S=s$ | ?- $\text{vStem}(liked, V, S)$. $V=like, S=d$ |
| ii. ?- $\text{vEnding}(s, TNS)$. $TNS=pres$ | ?- $\text{vEnding}(d, TNS)$. $TNS=pst$ |
| iii. ?- $\theta(is, L)$. $L=[\text{v}_{pred:pres}, \text{INFL}_{V:3sg}]$ | ?- $\theta(liked, L)$. $L=[\text{like}_{\theta}, \text{v}_{like:\theta:pst}, \text{INFL}_V]$ |

Generally, there may be multiple cluster possibilities for a verb, leading to multiple (distinct) initial WS's. For example, *be* is not only the copula in English, but also a main verb, e.g. *they are aeroplanes*, and the progressive auxiliary *be*, as in *they are eating lunch*. Similarly, a verb like *melt* may participate in various alternations, *the ice melted* / *the sun melted the ice*. We assume **LEX** lookup will return the appropriate cluster possibilities, e.g. as in (a) and (b).

- | | |
|--|---|
| a. | b. |
| ?- $\theta(is, L)$. | ?- $\theta(melted, L)$. |
| $L = [\text{v}_{pred:pres}, \text{INFL}_{V:3sg}]$; | $L = [\text{melt}_{\theta}, \text{v}_{melt:\theta:pst}, \text{INFL}_V]$ |
| $L = [\text{be}_{\theta}, \text{v}_{be:\theta:pres}, \text{INFL}_{V:3sg}]$ | $L = [\text{melt}_{\theta}, \text{v}_{melt:\theta:pst}, \text{INFL}_V]$ |
| $L = [\text{v}_{v:prog:pres}, \text{INFL}_{V:3sg}]$ | |

(Note: melt_{θ} and melt_{θ} signals that *melt* takes no complement ($\emptyset$ = empty set) and a θ -relevant **WS** item as complement, respectively.)

Verbs and LEX

Two distinct systems, **EXT** and **Merge**, are involved when adding a new verb.

- **EXT**: requires stemming (for parsing) and inflection (for spellout)
- **Merge**: requires verbal cluster INFL-v-Root and selectional properties to be defined.

EXT: Stemming

WordNet morphy does general English verb stemming duties when initially presented with a word during parsing. The predicate `morphyStem(Word,C,Root)` is used, with $C = v$ (for verbs). Examples below:

```
?- morphyStem(likes,v,Root).   Root = like.
?- morphyStem(seen,v,Root).   Root = see.
?- morphyStem(sold,v,Root).   Root = sell.
?- morphyStem(saw,v,Root).   Root = saw.
```

In each case above `morphyStem/3` computes the value of the variable `Root`, which then is the key for verb lookup in **LEX** for θ -selection (and other properties).

EXT: Inflection

For spellout after **Merge** has converged, each head H and its features are externalized via `ext( $H,N,Copies,List,OutputFlag$ )`, where *List* is a sequence of words and bound morphemes.²⁰ Content heads such as nouns, adjectives, adverbs and verbs spell out as (named) heads. Functional heads, e.g. C , v and INFL usually will not spell out as words. Instead, they may have features that may spell out as individual words or (bound) morphemes that must be then attached to words at Spellout. For verbs, generally there will be a root head immediately preceded by ϕ and TNS from functional heads INFL and v , respectively. (ϕ on INFL comes from Agree with the surface subject, an operation at **EXT**.) The mapping ϕ +TNS+Root to inflected form, called *Amalgamation*, is defined via `inflectedV( $\Phi,TNS,Root,Word$ )`.

Similarly, `inflectedV( $Suffix,Root,Word$ )` defines rules for regular (concatenative) cases such as *en* + *eat* = *eaten* and *ing* + *eat* = *eating*, as well as *ing* + *like* = *liking*.

```
?- inflectedV(ing,like,Word). Word = liking.
```

Irregular spellout must be listed explicitly in `irregularVspell( $Suffix,Root,Word$ )` as shown below:

```
irregularVspell(en,break,broken).
irregularVspell(ing,see,seeing).

irregularVspell(en,be,been).
```

²⁰ Inputs N and *Copies*, not described here track **FormCopy** heads that will not be pronounced. N gives the local position of the head in the structure and *Copies* contains a list of unpronounced positions. Together, they control whether the head generates output or not. If a copy is recognized, i.e. when N is listed in *Copies*, **EXT** is instructed to not generate anything for the head except with respect to Agree, e.g. in the case of Subject-Verb agreement, even if the argument is not spelled out at the edge of INFL, it can output ϕ -features used for verbal inflection. (Finally, *OutputFlag* controls **EXT** error signaling and debugging, and can be ignored.)

```

irregularVspell(ing,be,being).
irregularVspell(ing,have,having).
irregularVspell(en,have,had).

```

Merge

θ -configurations are built by θ -aware **Merge**, which relies on selectional properties of roots and other heads in the extended verbal projection. These are listed for individual verbs in `vR(Root, LittleV, R_selects)`.

Examples below:

```

vR(arrive, v:arrive,  $\theta$ ).           % a train arrives
vR(be, v:pred, none).              % copula, just a v
vR(be, v:v+:prog, zero).           % PROG -> be ing (keyed)
vR(dance, v:dance: $\theta$ ,  $\emptyset$ ).      % Mary dances
vR(eat, v:eat: $\theta$ ,  $\theta$ ).             % John ate a sandwich
vR(melt, v:melt: $\theta$ ,  $\emptyset$ ).        % ice melted
vR(melt, v:melt: $\theta$ ,  $\theta$ ).          % sun melted the ice
vR(must, v:v+:modal:word(must), zero). % must have (no TNS)
vR(saw, v:saw: $\theta$ ,  $\theta$ ).            % John sawed the rope
vR(see, v:see: $\theta$ ,  $\theta$ ).            % Mary saw John
vR(want, v:want: $\theta$ , 'INFL').        % Mary wants to ...
vR(will, v:v+:modal:word(will), zero). % will/would have (no TNS)
vR(win, v:win: $\theta$ ,  $\emptyset$ ).         % Mary won

```

For example, the entry for unaccusative root *arrive* indicates that it is keyed with the head `v:arrive`. This means the categorizer `v` selects for a phrase headed by root *arrive*. The root *arrive* itself selects for θ (denoting a θ -relevant **WS** item). An INFL head keyed to the categorized `v` by selection is also generated. θ -aware **Merge** can then construct the θ -configuration in (c) below, beginning with heads (a) inserted after stemming the input *arrived*. **IA** represents a θ -relevant item selected from the rest of the **WS**. In (d), INFL_v selects for the `v` phrase built in (c). INFL_v triggers **MS** for a θ -relevant item inside the θ -configuration. There is only one possibility, **IA**. *Internal Merge* produces (e). After the INFL_v phrase has been built, a further addition of declarative complementizer *C* completes the clause and **Merge**'s task is complete.

[a-j below refers to the earlier version of the SMT Parser with spec-INFL in narrow syntax. What follows is out of date and the documentation should be revised!]

- a. INFL_v $v_{\text{arrive:pst}}$ arrive_θ IA_ϕ *C*
- b. INFL_v $v_{\text{arrive:pst}}$ $\{\text{arrive}_{\theta:\text{pst}}, \text{IA}_\phi\}$ *C*
- c. INFL_v $\{v_{\text{arrive:pst}}, \{\text{arrive}_\theta, \text{IA}_\phi\}\}$ *C*
- d. $\{\text{INFL}_v, \{v_{\text{arrive:pst}}, \{\text{arrive}_\theta, \text{IA}_\phi\}\}\}$ *C*
- e. $\{\text{IA}_\phi, \{\text{INFL}_v, \{v_{\text{arrive:pst}}, \{\text{arrive}_\theta, \text{IA}_\phi\}\}\}\}$ *C*
- f. $\{C, \{\text{IA}_\phi, \{\text{INFL}_v, \{v_{\text{arrive:pst}}, \{\text{arrive}_\theta, \text{IA}_\phi\}\}\}\}\}$
- g. $\{C, \{\text{IA}_\phi, \{\text{INFL}_v, \{v_{\text{arrive:pst}}, \{\text{arrive}_\theta, \text{IA}_\phi\}\}\}\}\}$ ($\text{IA}_\phi, \text{IA}_\phi$)
- h. $C \text{ IA}_\phi \text{ INFL}_{v:\phi} v_{\text{arrive:pst}} \text{ arrive}_\theta$
- i. $\text{IA } \phi \text{ pst arrive}$
- j. IA arrived

In (g), **FormCopy** applies, constructing the c-command relation ($IA_\phi, \overline{IA_\phi}$), the 2nd of which will not be externalized. (h) shows the sequence of heads (in English-particular order) transmitted to Spellout at **EXT**. The declarative head C does not spell out. Agree applies between surface subject **IA** and $INFL_v$, valuing ϕ on $INFL_v$, spelling out as morpheme ϕ at (i). At step (j), the mapping from the linear sequence $\phi + \text{pst} + \text{arrive}$ to *arrived* occurs (as part of *Amalgamation*).

The 696A SMT Parser equivalent (for the above sequence of steps) is as follows:

In the case of unergative verb *dance*, the root selects $\emptyset$ (the empty set), i.e. it does not take a complement. The categorizer is listed as $v:\text{dance}:\emptyset$, this means v selects the root *dance* first, and a second **Merge** selects $\emptyset$ (denoting a θ -relevant **WS** item). In the case of verbal form *dances*, θ -aware **Merge** builds the θ -configuration in (c). The rest of the derivation steps follow those above, except for the $\phi + \text{pres} + \text{dance}$ mapping at *Amalgamation*. If **EA** is a plural noun, e.g. *they*, $3\text{pl} + \text{pres} + \text{dance} = \text{dance}$, and $3\text{sg} + \text{pres} + \text{dance} = \text{dances}$ in the case of a singular noun such as *Mary*.

[a-j below refers to the earlier version of the SMT Parser with spec-INFL in narrow syntax. What follows is obsolete and the documentation should be revised!]

- a. $INFL_v \ v_{\text{dance:pres}} \ \text{dance}_\emptyset \ EA_\phi \ C$
- b. $INFL_v \ \{v_{\text{dance:pres}}, \text{dance}_\emptyset\} EA_\phi \ C$
- c. $INFL_v \ \{EA_\phi, \{v_{\text{dance:pres}}, \text{dance}_\emptyset\}\} C$
- d. $\{INFL_v, \{EA_\phi, \{v_{\text{dance:pres}}, \text{dance}_\emptyset\}\}\} C$
- e. $\{EA_\phi, \{INFL_v, \{EA_\phi, \{v_{\text{dance:pres}}, \text{dance}_\emptyset\}\}\}\} C$
- f. $\{C, \{EA_\phi, \{INFL_v, \{EA_\phi, \{v_{\text{dance:pres}}, \text{dance}_\emptyset\}\}\}\}\}$
- g. $\{C, \{EA_\phi, \{INFL_v, \{\overline{EA_\phi}, \{v_{\text{dance:pres}}, \text{dance}_\emptyset\}\}\}\}\} \quad (EA_\phi, \overline{EA_\phi})$
- h. $C \ EA_\phi \ INFL_{v:\phi} \ v_{\text{dance:pres}} \ \text{dance}_\emptyset$
- i. $EA_\phi \ \phi \ \text{pres} \ \text{dance}$
- j. *They dance / Mary dances*

The 696A SMT Parser equivalent is as follows:

5. LEX Reference Manual

This section describes the information the SMT parser demands of **LEX** to operate. **LEX** must supply theory-determined features for heads that project syntactic structure such as nouns, verbs, adjectives, adverbs, prepositions, etc., as well as for items that (in this theory) are not syntactic heads, e.g. case particles, clitics, determiners, DOM, conjunctions, etc. **LEX** also contains the inflectional information needed to drive the **EXT** component of the parser. This information is accessed twice, once to decompose parser input into heads for assembly, and again when syntactic structure is passed to **EXT** to be linearized (and compared against the input). Naturally, much of this information is language particular.

The stages of the SMT parser are as follows:

1. Assuming the input has been tokenized into words, words are looked up in **LEX**, producing a sequence of heads. Ambiguous words are segregated.
2. Linear operations may apply to these heads before they are inserted into the initial Workspace (**Initial WS**). Ambiguity in lookup will result in (multiple) possible initial **WSs**.
3. In an **Initial WS**, **Merge** fires to build sentential structure. If **Merge** converges all heads of the **Initial WS** to a single *syntactic object* (**SO**), that **SO** is passed to the Interface.
4. At the Interface, operations such as phase-sensitive FormCopy and Binding establish additional structural relations.
5. At **EXT**, the **SO** augmented with additional (structural) relations is externalized layer-by-layer through the lens of language-particular **EXT** options, e.g. wrt. word order. The object is transformed from a hierarchical object into a spellout sequence.
6. **Linear operations**, e.g. inflection, may apply to that sequence to form the final spellout form.
7. The final spellout is compared against the parser input. If (substantially) the same, the **SO** is deemed a parse of the input. Multiple **SOs** can be accepted in the case of structural ambiguity, the SMT parser does not incorporate processing preference metrics. Also, **SOs** may have multiple spellouts as linearization need not be unique.

(See BFSU 2026 slides for concrete examples of this process view.)

In the SMT parser, words2heads/4 defines both steps 1 and 2, and produces a series of initial workspaces **Initial WS 1..n**.

5.1 Some Basic Notation

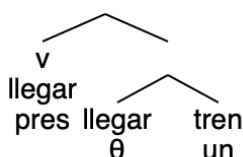
Generally, we may write H_F to indicate a head H has feature F . Multiple features are separated from each other by a colon (:), e.g. $H_{F1:F2}$ means head H has first feature $F1$, second feature $F2$, and so on. This notation with features subscripted is used in SMT Parser Browser output. The following example contains three heads v , $llegar$ (*arrive*) and $tren$ (*train*):

$\{v_{llegar:pres}, \{llegar_{\theta}, tren_{un}\}\}$

In SMT Parser Terminal output (due to the lack of plain text subscripting), $H_{F1:...:Fn}$ will be portrayed as $H_F1...:Fn$, the underscore ($_$) employed here to separate the name of the head from its features. The corresponding Terminal example is:

$\{v_llegar:pres, \{llegar__{\theta}, tren_un\}\}$

Note: $\{\dots\}$ represents a set formed by operation **Merge**. Green curly braces $\{\dots\}$ represents a set formed by operation **FormSet**. Nested sets represent hierarchical structure. Our example above is roughly equivalent to the following tree structure (shown with features stacked below the heads):



However, set members (unlike tree branches) are unordered.

For θ -Merge, the first and 2nd features are decoded. H_{θ} means head H takes a θ -relevant item as its complement. This results in the structure $\{H_{\theta}, \mathbf{IA}\}$, $\mathbf{IA}$ an internal argument. For example, $llegar_{\theta}$ (*arrive*) above is the head and $tren$ (*train*) its $\mathbf{IA}$. Below also, we have an adjectival phrase headed by predicate $long_{\theta}$ and a (verbal) root $like_{\theta}$ with $\mathbf{IA}$ sea_{the} containing a feature *the* (representing the definite article). Similarly, the θ -relevant nominal $corridor_a$ has feature *a* (representing the indefinite article).

$\{long_{\theta}, corridor_a\}$
 $\{like_{\theta}, sea_{the}\}$

$H_{F:\theta}$ means head H first selects for F and takes a θ -relevant item as its argument. For example, (little) v here selects for complement *like* and θ -relevant item $\mathbf{EA}$, an external argument:

$\{\mathbf{EA}, \{v_{like:\theta}, \{like_{\theta}, sea_{the}\}\}\}$

Using conventional terminology, the above structure is a vP headed by transitive $v_{like:\theta}$, a phrase because selectional requirements *like* and θ have been satisfied. Our initial example with *llegar* is also a vP despite not having $\mathbf{EA}$; instead, it is headed by a v which has no θ feature.

5.2 The Initial Workspace (Initial WS)

The *initial Workspace* (**Initial WS**) is populated by the SMT parser iterating over each word in the input to insert the associated heads into the **WS**. Once the **WS** has been initialized, **Merge** will fire to construct θ -configurations preferentially; after which, INFL and an appropriate C head will be added to complete the transition from computed *propositional structures* into the *clausal domain*.

On the staging of **Merge**:

- **Chomsky** (*Minimal Inquiries*, 2000: 132)
 - *Properties of the probe/selector α must be satisfied before new elements of the lexical subarray are accessed to drive further operations.*

Suppose a head has both θ -selection and categorial selection properties, e.g. v may select for R and also a θ -relevant **EA**. Both of these properties must be satisfied by **Merge** before INFL can select for (a phrase headed by) v . Examples:

$\{\text{INFL}_v, \{\text{EA}, \{v_{\theta:R}, \text{RP}\}\}\}$
* $\{\text{INFL}_v, \{v_{\theta:R}, \text{RP}\}\}$ (missing **EA**)
* $\{\text{INFL}_v, \{v_{\theta:R}, \text{EA}\}\}$ (missing **RP**)
* $\{\text{EA}, \{\{\text{INFL}_v, v_{\theta:R}\}, \text{RP}\}\}$ (violates selectional property principle)
* $\{\text{EA}, \{\text{INFL}_v, \{v_{\theta:R}, \text{RP}\}\}\}$ ditto.
* $\{\text{RP}, \{\text{INFL}_v, \{v_{\theta:R}, \text{EA}\}\}\}$ ditto.

In other words, we assume there is only one way to correctly assemble the INFL- v -R structure above, and **Merge** does not (and cannot freely) overgenerate. (See also Section 5.1 for the fixed order of satisfaction of local selectional properties.)

Note INFL is (part of) the *bridge* connecting the propositional and clausal systems.

- **Chomsky**:
 - *INFL is kind of an extension of v , with features **amalgamating** with v , the core element in theta theory, though this formulation also include objects of N, V, A .*

After **Merge** is complete, interface operations, e.g. compute copy relations, will apply to **I-Language** structure. **EXT** will spell out (unordered) **I-Language** structure as a sequence of words while taking in consideration **E-language**-particular factors, including word order variation, whether occurrences should be pronounced or not, affixal morphology and cases of mapping from head features to spellout, e.g. for clitics, determiners and genitive Case realization inside NPs.

5.3 word2heads/3 (module parse)

The WS interface to LEX is given by word2heads/3. Each input word is checked against isWord/1 first. Then chaining word2heads/3 for input words from left to right, applying any Actions between nearest heads, results in an initial WS. Structure-building can then start.²¹

isWord(*Word*) true if *Word* can be found in LEX.

Input: *Word* an (inflected) form

word2heads(*Word*,*Heads*,*Actions*) *Word* maps to WS *Heads* and linear *Actions*.

Input: *Word* an (inflected) form

Output: *Heads* a list of heads to be inserted into the initial WS.

Output: *Actions* (if non-empty) usually call for features to be added to adjacent heads.

For example, word2heads/3 on the English infinitive marker *to* results in.

```
?- word2heads(to,Hs,As).      Hs = [], As = [addFType(tns,to,v)]
```

Hs = [] means nothing is added to the WS. Action addFType/3 means *to* is a marker of non-finite TNS on a following v head.

Similarly, Spanish dative clitic *le* adds nothing to the WS. Action addFType/3 here means *le* ends up a marker on a following appl (applicative) head.

```
?- word2heads(le,Hs,As).      Hs = [], As = [addFType(clitic,le,appl)]
```

By contrast, word2heads/3 on the Spanish verb *llegan* generates no actions, but two possibilities for ϕ -feature agreement, 2pl or 3pl:

```
?- word2heads(llegan,Hs,As).  
Hs = [llegar $\theta$ , vllegar:pres, INFLv:2pl], As = [] ;  
Hs = [llegar $\theta$ , vllegar:pres, INFLv:3pl], As = [] ;  
false.
```

The present tense verb form *llegan* (*arrive*) triggers the generation of three heads, i.e. R = llegar_{θ} , v = $\text{v}_{\text{llegar:pres}}$ and $\text{INFL}_{v:\{2,3\}pl}$. The suffix θ on the root *llegar* indicates it selects for a IA. Next, the suffix *llegar* on v indicates v selects for root *llegar*. **Merge** will make use of this head information to build basically the same θ -configuration for *llegan dos trenes* and *llega un tren* (*two trains arrive/a train arrives*).

```
{vllegar:pres, {llegar $\theta$ , trenun}}  
{vllegar:pres, {llegar $\theta$ , trenesdos}}
```

²¹ The current parser implements this strategy strictly (null hypothesis). However, one could fire structure-building proactively as soon as each head enters the WS to attempt some predictive filtering. We leave this to future work, noting here that no new mechanism (in principle) need be invented as long as θ -theory is maintained.

Following (Chomsky 2021), in this framework there is no need for a separate T head; instead **TNS** (*pres* = present) is a feature of v.²² Finally, the v suffix on INFL indicates it selects for v. Note that word2heads/3 inserts INFL into the **WS** already bearing ϕ -feature 2pl or 3pl. **Merge** constructs the following structures:

```
{INFLv:3sg, {vllegar:pres, {llegarθ, trenun}}}  
{INFLv:3pl, {vllegar:pres, {llegarθ, trenesdos}}}
```

INFL _{ϕ} invokes *Minimal Search* (MS) and locates the **IA**, which INFL _{ϕ} must agree with. In Spanish, the **IA** can (optionally) be pronounced at the left edge of INFL _{ϕ} , giving the alternative word order *dos trenes llegán / un tren llega*.

word2heads/3 for the indefinite article *un*:

```
?- word2heads(un,Hs,As).  
Hs = [], As = [addFType(det,un,type(θRnoun(_,sg,m)))]
```

Again, Hs = [] means nothing is added to **WS**. Action addFType/3 means *un* is a determiner to be added as feature to a θ -relevant noun (θ Rnoun/3) with ϕ -features sg-m (masculine singular).²³

In the case of common nouns such as *trenes/tren* (*trains/train*), word2heads/3 will just generate a simple head for WS insertion (without an action).

```
?- word2heads(trenes,Hs,As).      Hs = [trenes], As = []  
?- word2heads(tren,Hs,As).      Hs = [tren], As = []
```

Given the following θ Rnoun/3 facts for Spanish, article *un* can attach only to masculine singular *tren* (*train*) (not *trenes*).

```
?- θRnoun(trenes,pl,m).  
true.
```

```
?- θRnoun(trenes,pl,f).  
false.
```

```
?- θRnoun(tren,sg,f).  
false.
```

```
121 ?- θRnoun(tren,sg,m).  
true ;  
false.
```

In the case of English, grammatical gender is not checked.

²² In a sentence like *John arrived and will meet the president*, there is a single INFL, but the two verbs have v's with different θ -properties, i.e. one (*meet*) has an **EA** and the other (*arrive*) does not. They also differ in **TNS**. (Chomsky 2021) places **TNS** on v.

²³ The feminine of the article *un* is *una* and the corresponding action is
addFType(det,una,type(θRnoun(_,sg,f))).

5.4 Verbs $\theta/3$ (module θ a), $vStem/4$, $vRoot/3$ (module LEX)

In the case of verbs, $word2heads/3$ (from module $parse$) relies on $\theta/3$ (from module θ -Theory) to produce the list of θ -Merge-relevant *Heads* (drawn from $v-appl-R$) to be inserted. These heads will combine with arguments under **WS Merge** to form a θ -configuration (examples are given in the previous section). Auxiliary verbs do not select θ -relevant items and are separately described in a later section.

$\theta(Word, Heads, Actions)$ *Word* is an (inflected) verb with WS *Heads* and *Actions*.

Input: *Word* an (inflected) form

Output: *Heads* a list of heads to be inserted into the initial **WS**.

Output: *Actions* (if non-empty) usually call for features to be added to adjacent heads.

$\theta/3$ relies on language-particular $vStem/4$ (for inflection decoding), $vRoot/3$ (for θ -properties) and $appl/1$ defined in **LEX**.

$vStem(Word, R, TNS, \Phi)$ *Word* is an (inflected) verb that stems down to root *R*.

Input: *Word* an (inflected) verb form.

Output: *R* the root verb form.

Output: *TNS* tense associated with verb form *Word*, e.g. *pres* or *pst*.

Output: *Phi* ϕ -features associated with verb form *Word*, e.g. *3sg* (English) or *3sg-f* (Spanish).

Examples from English and Spanish **LEX** are given below. Participles, e.g. as in *broken* or *comido* (*eaten*) below, are given special treatment; both *TNS* and ϕ -features have distinguished values *pastp/presp* and *null*, respectively.

```
?- vStem(is,R,TNS,Phi).      R = be,      TNS = pres,  Phi = 3sg
?- vStem(ate,R,TNS,Phi).    R = eat,      TNS = pst,   Phi = null
?- vStem(broken,R,TNS,Phi). R = break,    TNS = pastp, Phi = null
?- vStem(would,R,TNS,Phi).  R = will,     TNS = pst,   Phi = null
?- vStem(to,R,TNS,Phi).     % infinitive marker to is not a verb
false.
```

```
?- vStem(dio,R,TNS,Phi).    R = dar,      TNS = pst,   Phi = 3sg
?- vStem(llegan,R,TNS,Phi). R = llegar,   TNS = pres,  Phi = 2pl ;
                          R = llegar,   TNS = pres,  Phi = 3pl
?- vStem(comido,R,TNS,Phi). R = comer,    TNS = pastp, Phi = null
?- vStem(fue,R,TNS,Phi).   R = ir,       TNS = pst,   Phi = 3sg ;
                          R = ser,      TNS = pst,   Phi = 3sg.
```

Note $vStem/4$ may return multiple values, as in the case of *llegan* (*arrive*) and *fue* above due to ambiguity of form with respect to ϕ -features and root *ir* (*go*) vs. *ser* (*be*), respectively.

To expand the hand-built coverage of English **LEX**, $vStem/4$ will call upon verb entries from WordNet using $morphyStem/3$ to supplement the hand-built entries. (Note $morphyStem/3$ differs from $morphy/3$ in that it bypasses default $noMorph/0$.) It also tries to infer the *TNS* and ϕ -feature from the verb form. (Note the Spanish **LEX** currently uses hand-built entries only.)

The English verb lexicon currently relies on English-particular vStem/3 with suffix encoding below to supply verb forms to language-independent vStem/4 above. (The Spanish **LEX** does not define vStem/3, choosing instead to define vStem/4 with TNS and ϕ -features directly.)

vStem(Word,R,Suffix) *Word* is an (inflected) verb that stems down to root *R* + *Suffix*.

Input: *Word* an (inflected) verb form.

Output: *R* the root verb form.

Output: *Suffix* associated with verb form *Word*, e.g. d, en, '' (the empty atom) or s.

Examples below:

```
?- vStem(eat,Root,Suffix).    Root = eat, Suffix = ''.
?- vStem(eats,Root,Suffix).  Root = eat, Suffix = s.
?- vStem(ate,Root,Suffix).   Root = eat, Suffix = d.
?- vStem(eaten,Root,Suffix). Root = eat, Suffix = en.
?- vStem(eating,Root,Suffix). Root = eat, Suffix = ing.
```

Suffixes generally map to TNS features using English-particular vEnding/2, summarized below.

vEnding(Suffix,TNS) Suffix maps to pst (pastp), pres (presp) or nonf.

Input: *Suffix* one of the following:

1. '' (empty atom), untensed (nonf) or present tense (pres),
2. d/ed, past tense (pst),
3. s (3sg present tense), 1 (1sg present tense) and 2 (2sg present tense), (pres)
4. en, past participle (pastp), and
5. ing, present participle (presp).

Basically, endings d/ed and ''/s are recognized and map to TNS features pst (past) and pres (present), respectively. The empty atom '' alone has the option of no TNS mapping for untensed, e.g. *win* in non-finite *to win*.

(English verb ϕ -feature differences are merely vestigial; suffix s will map to 3sg with respect to INFL $_{\phi}$ Agreement at **EXT**.)

Suffixes are computed from sub-string differences between the Word and Root, not always possible, e.g. as in irregular *sold* = *sell* + *d*, but accomplished automatically for regular cases, e.g. *sees* = *see* + *s*, *liked* = *like* + *d* and *kicked* = *kick* + *ed*. At present, a rule is also provided for Root ending in -y to Word ending in -ies and suffix s, e.g. *flies* from verb root *fly*.

Irregular suffixation cases must be explicitly listed as irregularV/3 entries in **LEX**. Sample entries:

```
irregularV(broken,break,en).
irregularV(broke, break,d).
irregularV(saw, see, d).
irregularV(seen, see, en).
```

If no irregular entry exists and LEX cannot detect the suffix by sub-string computation, but *morphology* indicates it is a valid verb form, LEX guesses that the relevant suffix is both *d* (past tense) and *en* (past participle).

For example, when the verb *land* is not in LEX, but is accessible via WordNet, vStem/3 operates as follows:

```
?- vStem(land, Root,Suffix).    Root = land, Suffix = ''.  
?- vStem(lands, Root,Suffix).   Root = land, Suffix = s.  
?- vStem(landed, Root,Suffix).  Root = land, Suffix = d ; Suffix = en.
```

There is a need to guess is because WordNet *morphology* does not supply any suffixes, even when it recognizes an irregular form. For example, suppose *ate* is not yet listed in irregularV/3, then vStem/3 will return both choices.

```
?- vStem(ate,Root,Suffix).      Root = eat, Suffix = d ; Suffix = en.
```

Once we add an irregularV/3 entry, which takes precedence over guessing, the incorrect answer will be blocked.

```
irregularV(ate,eat,d).
```

```
?- vStem(ate, Root,Suffix).    Root = eat, Suffix = d  
?- vStem(eaten,Root,Suffix).   Root = eat, Suffix = en.
```

Note that no irregular entry is needed for the past participle *eaten* as *eat* + *en* = *eaten*.

vRoot(Root,V,Compl) *Root* has *v* and root θ -properties. Assertion: use_morphology(*v*).

Input: *Root* is the uninflected verb, e.g. *eat*, *dance* or *arrive*.

Output: *V* is a (little) *v* head, e.g. *Veat*: θ , *Vdance*: θ and *Varrive* (respectively, for the above three verbs)

Output: *Compl* indicates the type of complement the root selects for, θ , $\emptyset$ and θ (respectively)

These entries in LEX are listed directly using vR/3 facts. Examples:

```
vR(arrive, v:arrive,   $\theta$ ).      % a train arrives  
vR(dance,  v:dance: $\theta$ ,  $\emptyset$ ).    % Mary dances ( $\emptyset$  = no complement)  
vR(eat,     v:eat: $\theta$ ,     $\theta$ ).     % John ate a sandwich
```

As WordNet *morphology* does not supply θ -theoretic information, if useMorphology/ θ is turned on (or use_morphology(*v*) is asserted), unlisted verbs will be assumed to be transitive, i.e. both *v* and its root selects for θ -relevant items. This is illustrated below in the case of *land*.

```
?- vRoot(land,V,Compl).        % by default, useMorphology/ $\theta$  is off  
false.  
?- assert(use_morphology(v)).   % turn on WordNet just for verbs  
true.  
?- vRoot(land,V,Compl).        V = v_land: $\theta$ , Compl =  $\theta$ .
```

To summarize, verb forms are looked up using θ /3 as the LEX entry point for all languages, with retrieved heads populating the initial WS. θ /3, calls on vStem/4 to come up with a root

(plus TNS and ϕ -feature information). In the case of English, this involves manipulation of suffixal endings (via *vStem/3* and *vEnding/2* as described above, along with some guessing when WordNet is enabled. In the current Spanish LEX, there is no expansion to WordNet, all entries are listed explicitly. For example, *gustar* (*like*) has the following entries:

```
vR(gustar, v:gustar,  $\theta$ ).    % see also appl
appl(gustar).                % takes an applicative  $\theta$ -selecting head
```

That is, *gustar* is a root that maps to three verbal heads, *v* (with no θ -selection), a root selecting for a θ -relevant item, plus an applicative head *appl* (selecting for a θ -argument), as the call to $\theta/3$ below indicates. Inflectional features reside on a separate head *INFL*.

```
?-  $\theta$ (gusta,H,A24).      H = [appl $\theta$ , gustarappl: $\theta$ , vgustar:pres, INFLv:3sg]
?-  $\theta$ (gustan,H,A).        H = [appl $\theta$ , gustarappl: $\theta$ , vgustar:pres, INFLv:2pl] ;
                           H = [appl $\theta$ , gustarappl: $\theta$ , vgustar:pres, INFLv:3pl]
```

In other words, inflected form *gusta* is the 3rd person singular present tense form of root *gustar*, and *gustan* is its 2nd or 3rd person plural past tense form. As *gustar* is a regular Spanish verb of the class *ar*, it makes use of (Spanish-particular) stemming rules to construct the verb forms.

```
class(gustar,ar).
```

```
endings(pres,ar,[o,as,a,amos,an,an]).    % [1sg, 2sg, 3sg, 1pl, 2pl, 3pl]
endings(pst, ar,[é,aste,ó,amos,aron,aron]).
```

Similar entries exist for Spanish *ir* and *er* verb classes. In the case of irregular verbs, the forms must be listed explicitly. For example, the unergative verb *andar* (*walk*) has the following entries:

```
vR(andar, v:andar: $\theta$ ,  $\emptyset$ ).    % walk

irregular(andar,pres,[ando,andas,anda,andamos,andan,andan]).
irregular(andar,pst,[anduve,anduviste,anduvo,anduvimos,anduvieron,anduvieron]).
irregular(andar,fut,[andaré,andarás,andaré,andaremos,andarán,andarán]).
```

The inflected form *anduvimos* (*we walked*) generates the following heads for the initial **WS**.

```
?-  $\theta$ (anduvimos,H,A).  H = [andar $\emptyset$ , vandar: $\theta$ :pst, INFLv:1pl], A = [].
```

²⁴ For simplicity of presentation, action *A* which forces overt expression of a dative clitic is not shown above. For the record, *A* = [last(appl,hasDATclitic(_))] checks to see if there a clitic attached to head *appl* as the last step of the linear process (before Merge begins and structure is built).

5.5 Verbs: passive, progressive, perfective (module LEX)

LEX for θ -selecting verbs are described in the previous section. Auxiliary verbs, e.g. *be* or *ser* (Spanish), do not select for θ -relevant items. In this framework, auxiliary *be* is a *v* that introduces passive or progressive aspect, or functions as a copula to introduce a predicate (e.g. adjectival).

Currently, word2heads/3 reveals 4 entries each for forms *was* (pst) and *is* (pres) (one of which is not an auxiliary, but a main verb, i.e. a θ -selecting verb).

?- word2heads(*was*, Hs, As) .

Hs = [<i>v_{pred:pst}</i> , <i>INFL_v</i>],	As = [] ;
Hs = [<i>be_θ</i> , <i>v_{be:θ:pst}</i> , <i>INFL_v</i>],	As = [] ;
Hs = [<i>v_{v:prog:pst}</i>],	As = [<i>matchLV(v_{v:prog:pst})</i>] ;
Hs = [<i>v_{v:pass:pst}</i>],	As = [<i>matchLV(v_{v:pass:pst})</i>] ;

?- word2heads(*is*, Hs, As) .

Hs = [<i>v_{pred:pres}</i> , <i>INFL_{v:3sg}</i>],	As = [] ;
Hs = [<i>be_θ</i> , <i>v_{be:θ:pres}</i> , <i>INFL_{v:3sg}</i>],	As = [] ;
Hs = [<i>v_{v:prog:pres}</i>],	As = [<i>matchLV(v_{v:prog:pres})</i>] ;
Hs = [<i>v_{v:pass:pres}</i>],	As = [<i>matchLV(v_{v:pass:pres})</i>] ;

(Note: As = Actions, each of which is a head operation on a subsequent head to be carried out prior to insertion into the initial WS (before Merge fires).)

The 4 entries are derived from the following entries in vR/3.

vR(<i>be</i> , <i>v:pred</i> , none).	% <i>copula, just a v</i>
vR(<i>be</i> , <i>v:be:θ</i> , θ).	% <i>main verb be, e.g. they are planes</i>
vR(<i>be</i> , <i>v:v+_:prog</i> , zero).	% <i>PROG → be ing (keyed)</i>
vR(<i>be</i> , <i>v:v+_:pass</i> , zero).	% <i>PASSIVE → be en (keyed)</i>

English spellout of each of these will usually be an inflected form of *be*: an example corresponding to each entry is given below (with the relevant part of the structure non-grayed out).

<i>John is happy</i>	{ <i>INFL_{v:3sg}</i> , { <i>v_{pred:pres}</i> , { <i>happy_θ</i> , <i>John</i> }}}
<i>They are planes</i>	{ <i>INFL_{v:pl}</i> , { <i>they</i> , { <i>v_{be:θ:pres}</i> , { <i>be_θ</i> , <i>planes</i> }}}} }
<i>Mary was eating lunch</i>	{ <i>INFL_v</i> , { <i>v_{v:prog:pst}</i> , { <i>Mary</i> , { <i>v_{eat:θ}</i> , { <i>eat_{θ:presp}</i> , <i>lunch</i> }}}} }
<i>Planes were landed</i>	{ <i>INFL_v</i> , { <i>v_{v:pass:pst}</i> , { <i>v_{land}</i> , { <i>land_{θ:pastp}</i> , <i>planes</i> }}}} }

In the case of progressive and passive *v*, the linear head operation *matchLV/1* action performs the following series of *ad hoc* actions:

- (i) It looks for a following *v*.
- (ii) In English, it also constrains the following verb to have the present and past participle forms for progressive and passive *v*, respectively.
- (iii) In the case of passive *v* only, it also must remove the EA θ -property of the matched (following) *v*.

For example, the parser assumes the verb *land* is transitive as the verb is not in (hand-built) **LEX** but inferred to be a verb through WordNet *morphy*.

```
?-  $\theta$ (landed,Hs,As).
Hs = [land $\theta$ , vland: $\theta$ :pst, INFLv],    As = [] ;
Hs = [land $\theta$ :pastp, vland: $\theta$ , INFLv],    As = []
```

The head $v_{\text{land}:\theta}$ selects for an **EA** but that θ -feature is deleted at the final stage before insertion into the initial **WS** in the case of *planes were landed* above.

Turning to consider the *ser* passive in Spanish (putting aside the reflexive *se* passive for now), as in the sentence *el alimento fue comido* (*the food was eaten*), we assume essentially the same entry for passive *ser* as in English passive *be*.

```
vR(ser, v:ser: $\theta$ ,  $\theta$ ).           % main verb ser
vR(ser, v:v+_:pass, zero).      % passive ser
```

$\theta/3$ reports that the past tense verb form *fue* is ambiguous between *ir* (*go*) and *ser* (*be*); *ser* itself is either a main or the passive verb.

```
?-  $\theta$ (fue,Hs,As).
Hs = [ir $\theta$ , vir: $\theta$ :pst, INFLv:3sg],      As = [] ;
Hs = [ser $\theta$ , vser: $\theta$ :pst, INFLv:3sg],      As = [] ;
Hs = [v:v:pass:pst],                  As = [matchLV(vv:pass:pst)].
```

```
?-  $\theta$ (comido,Hs,As).
Hs = [comer $\theta$ :pastp, vcomer: $\theta$ , INFLv],    As = []
```

The verb form *comido* is the past participle of transitive *comer* (*eat*). Computation proceeds as in the English case, and we obtain the following I-Language structure for *el alimento fue comido* in Spanish. INFL $_{\phi}$ at **EXT** can spell the object at its *el alimento* left edge.

```
{C, {INFLv, {vv:pass:pst, {vcomer, {comer $\theta$ :pastp, alimentoel}}}}}
```

Initial Spellout: el alimento 3sg-m pst ser pastp comer
Spellout: el alimento fue comido

Note the spellout rule for the passive head $v_{v:pass:pst}$ must map to *ser* in Spanish and *be* in its English counterpart *the food was eaten* below.

```
{C, {INFLv, {vv:pass:pst, {veat, {eat $\theta$ :pastp, foodthe}}}}}
```

Initial Spellout: the food 3sg pst be en eat
Spellout: the food was eaten

The $v_{v:pass}$ and $v_{v:prog}$ information is directly coded in **LEX** for each language.

```
passive(be,en).           % be + en + V
progressive(be,ing).      % be + ing + V

passive(ser,pastp).       % ser + pastp + V
progressive(estar,presp). % estar + presp + V
```

Simple rules of linear affixation ensure the sequence 3sg-m + pst + ser combines to spell *fue*, pastp + comer spells *comido*. Similarly, 3sg + pst + be spells *was*, en + eat spells *eaten*.

5.6 Nouns (module LEX)

A brief checklist. For each new noun:

1. Define $\theta R/1$ and check $\theta Rnoun/1$.
2. Check to see `morphyStem/3` (or `morphy/3`) works for stemming the plural form. Define `num/2` for plural form.
3. Check its ϕ -features using $\phi/2$.
4. Decide whether you need to define `caseR/2`, `whR/1`, `nR/2` and `animate/1`.

The parser expects the following predicates to be defined in LEX for nouns.

<code>$\theta Rnoun(Word)$</code>	<code>Word</code> is a θ -relevant form. Examples: <i>birds</i> , <i>bird</i> .
<code>$\theta Rnoun(Word, NUM, GEN)$</code>	<code>NUM</code> (sg, pl) and <code>GEN</code> (m, f) values should be supplied as input. (Note: <code>GEN</code> is ignored for English.)

For all languages, the parser will rely on the following three predicates to retrieve inflectional features. INFL will employ **MS** to identify the closest θ -relevant item and agree with it. (If there is no overt ϕ -feature agreement in the language, $\phi/2$ should be defined to succeed but return some agreed-upon null value for `Phi`.)

<code>num(Word, pl)</code>	True when <code>Word</code> is a plural (pl) noun.
<code>$\phi(Word, Phi)$</code>	<code>Phi</code> (3sg, 3pl) for English, <code>PERNUM-GEN</code> , e.g. 3sg-m, for Spanish.
<code>caseR(Word, CASE)</code>	<code>CASE</code> restriction, e.g. pronouns <i>we</i> (nom) and <i>us</i> (acc) in English.

To expand the hand-built coverage of English **LEX**, the parser accesses nominals through `$\theta Rnoun/3$` , which is also defined for common nouns that may not exist in the hand-built `$\theta R/1$` entries. This expansion is currently delegated to WordNet's *morph* function (defined in Python, and accessed by Prolog via Janus). The commands `useMorph/0` and `noMorph/0` toggle whether this is operational.

For example, *plane* (but not *aeroplane*) is defined in the hand-built `$\theta R/1$` entries. But *aeroplane* is defined in, and is therefore accessible via, WordNet.

```
?-  $\theta R(plane)$ .           % hand-built LEX
true.
?-  $\theta R(aeroplane)$ .      % not present in LEX
false.
?-  $\theta Rnoun(aeroplane)$ . % parser interface predicate
false.

?- useMorph.             % noMorph. to switch it off

?- morphy(aeroplane,n,R). R = aeroplane.
?-  $\theta Rnoun(aeroplane)$ .
true.
?- num(aeroplane,pl).    % singular
false.
```

Routines `morphy/3` and `morphyStem/3` are defined in the English **LEX** to handle noun, verb, adjective and adverb stemming. Note: `θRnoun/1` calls `morphyStem/3`, which (unlike `morphy/3`) bypasses the `noMorphy/1` prohibition.

```
?- morphy(aeroplanes,n,R).      R = aeroplane.
?- morphyStem(aeroplanes,n,R).  R = aeroplane.

?- θRnoun(aeroplanes).
true.
?- num(aeroplanes,pl). % plural
true.
```

`morphyStem/3` directly invokes `wn.morphy()` from `nltk` (in Python) via `Janus` in Prolog.

```
$ python3
>>> from nltk.corpus import wordnet as wn
>>> wn.morphy('aeroplane','n')
'aeroplane'
>>> wn.morphy('aeroplanes','n')
'aeroplane'
```

But, `morphyStem/3` will sometimes fail to work correctly. We cannot infer from identical form and root `R`, as reported below by `wn.morphy()`, that *men* is plural. (However, it stems *women* to *woman* correctly.)

```
?- morphyStem(men,n,R).      R = men.
?- morphyStem(women,n,R).   R = woman.
```

For erroneous cases like *men*, they must be explicitly listed in **LEX** using `num/2` clauses to block *morphy*.

```
num(men,PL) :- !, PL = pl.    % red cut

?- num(men,NUM).      NUM = pl.
?- num(men,sg).      false.
```

Currently, `num/2` is designed to be tested for plurality only, i.e. we will only call `num/2` with 2nd argument set to `pl`. The following (incorrect) queries should never be made.

```
?- num(man,NUM).      false.
?- num(man,sg).      false.
```

Nouns can also project structure and head θ -configurations. In **LEX**, `nR/2` defines whether a noun takes a θ -relevant item as an IA.

```
nR(Word, Compl)      Word is a noun form selecting Compl = θ.
```

For example, deverbal *bombing(s)* has entries:

```
nR(bombing, θ).
nR(bombings, θ).
```

```
θR(bombing).      % Note: no entry for bombings
```

Note that *bombing* is also a θ-relevant head by θRnoun/1.

```
?- θRnoun(bombing).  
true.
```

Although it is not directly listed in **LEX**, the plural form *bombings* is also a θ-relevant head, determined here through *morph* (via *morphStem*/3) and its regular morphology, i.e. *bombing* + *-s* = *bombings*.

```
?- θRnoun(bombings).  
true.
```

```
?- num(bombings,pl).  
true.
```

```
?- morphStem(bombings,n,R).  R = bombing
```

Show structures

Possessives from Case & θ document.

5.7 Determiners

In this framework, determiners do *not* project structure. For θ -theory, we assume **MS** accesses θ -relevant nouns (that head noun phrases) rather than DPs (*Determiner Phrases*), as is sometimes assumed in GB theory.

- Chomsky (p.c.):
 - **Is this External Merge?**
 - We're just ignoring functional elements, stick them in wherever you want. And, of course, you know there's lots of things to say about them, so why does the definite article appear before the noun? In fact, does the definite article even apply to the noun? *Maybe the definite article's a feature of the noun phrase.*
 - **Like in Semitic, for example, it's just distributed among the elements of the noun phrase.**
 - Hebrew: *ha-yeled ha-ze* ('this child')
 - Language-particular: attributive adjectives must agree in definiteness; and predicative adjectives are indicated syntactically, by the lack of an article in conjunction with a definite noun.

Determiners represent the spellout of a bundle of semantic features associated with a head noun, e.g. *the* (*el*) is definite, *a(n)* (or *un/una*) indefinite (in Spanish). With respect to narrow syntax, we treat determiners, e.g. *the* (and *a*), as single features, themselves shorthand for feature bundles, on a noun.

With respect to the parser, head2words/3 will convert a determiner word *D* into a (linear) Action only. No heads will enter the **WS**. That Action is passed on as a future to the parser and will result in *D* being added as a feature to a following noun (defined by θ Rnoun/3) prior to **WS** insertion. Some examples:

```
?- word2heads(the,Hs,As).
    Hs = [], As = [addFType(det,the,type( $\theta$ Rnoun(_,_,_)))]

?- word2heads(el,Hs,As).
    Hs = [], As = [addFType(det,el,type( $\theta$ Rnoun(_,sg,m)))]

?- word2heads(a,Hs,As).
    Hs = [], As = [addFType(det,a,type( $\theta$ Rnoun(_,sg,_)))]

?- word2heads(una,Hs,As).
    Hs = [], As = [addFType(det,una,type( $\theta$ Rnoun(_,sg,f)))]
```

Note the noun may have to satisfy ϕ -feature requirements, e.g. sg-m, via the NUM and GEN parameters to θ Rnoun/3.

The parser expects det/3 to be defined in **LEX**.

`det(Word,NUM,GEN)` *Word* is a determiner inflecting for NUM and GEN.
NUM (sg, pl) and GEN (m, f) values should be supplied or left open (as an anonymous variable =_). (Note: GEN is ignored for English.)

Some sample entries from the Spanish **LEX**:

```
det(un,sg,m).           % a [masc, indef]
det(una,sg,f).          % a [fem, indef]
det(unos,pl,m).         % some [masc, indef]
det(unas,pl,f).         % some [fem, indef]

det(el,sg,m).           % the [masc, def]
det(la,sg,f).           % the [fem, def]

det(dos,pl,_).          % two [pl, def]
det(tres,pl,_).         % three [pl, def]
```

In the English **LEX**, the NUM feature is generally specified, but we leave the gender value open as grammatical gender is not relevant to English. Sample entries:

```
det(a,sg,_).
det(every,sg,_).
det(one,sg,_).
det(two,pl,_).
det(three,pl,_).
det(some,pl,_).
det(the,_,_).
```

TODO: some theory for possessive pronouns.

5.8 Adjectives (module LEX)

Currently, an adjective always triggers θ -Merge to form a θ -configuration. Optionally, nominalization can be triggered after θ -Merge to create a θ -relevant item for further Merge. The parser interface to adjectives is via `adj/2`.

`adj(Word, Compl)` *Word* is an adjective, `Compl =  $\theta$` indicates it selects for a θ -relevant item.

Input: *Word* an (inflected) adjective form.

Output: `Compl =  $\theta$` if *Word* takes an IA.

Example: `narrow $\theta$` selects for an object as in $\{\textit{narrow}_{\theta}, \textit{corridor}_{the}\}$. This is a θ -configuration. Nominalization is optional as the follow examples demonstrate:

The corridor is narrow $\{\text{INFL}_{\varnothing}, \{\text{be}_{pres}, \{\textit{narrow}_{\theta}, \textit{corridor}_{the}\}\}\}$

I liked the narrow corridor $\{\text{INFL}_{\varnothing}, \{\text{I}, \{\text{v}_{\theta}, \{\text{like}_{\theta}, \{\textit{corridor}_{the}, \{\textit{narrow}_{\theta}, \textit{corridor}_{the}\}\}\}\}\}\}$

Adjectives are explicitly listed in `aR/2`, but the parser interfaces via `adj/2` to permit LEX expansion via *morphology* (see below).

`aR(Word, Compl)` *Word* is an adjective, `Compl =  $\theta$` indicates it selects for a θ -relevant item.

Input: *Word* an (inflected) adjective form

Output: `Compl =  $\theta$` if *Word* takes an IA.

Sample entries:

`aR(colorless,  $\theta$ ).`

`aR(criminal,  $\theta$ ).`

`aR(dark,  $\theta$ ).`

`aR(green,  $\theta$ ).` *% all color names can be adjectives or nouns, right?*

`aR(happy,  $\theta$ ).`

`aR(long,  $\theta$ ).`

`aR(flying,  $\theta$ ).` *% gerunds can be adjectival*

To expand the hand-built coverage of English LEX, *morphology/3* (controlled by *useMorphology/0*) is used to find additional adjectives. For example, *adventurous* is not an adjective in the hand-built LEX.

`?- adj(adventurous, C).`

false.

`?- useMorphology.`

true.

`?- adj(adventurous, C).      C =  $\theta$ .`

For Spanish, adjectives also inflect for number and gender. Currently, all forms are listed explicitly. `aPhi/4` defines these φ -feature components (inefficiently) keyed by the inflected form.

`aR(larga,  $\theta$ ).` *% long, f*

`aR(largo,  $\theta$ ).` *% long, m*

```
aR(lenta,  $\theta$ ).           % slow, f
aR(lento,  $\theta$ ).         % slow, m
```

```
aPhi(larga, '3sg', f).    % long
aPhi(largo, '3sg', m).    % long
aPhi(lenta, '3sg', f).    % slow
aPhi(lento, '3sg', m).    % slow
```

Dyadic adjectives require an additional θ -selecting head.

```
{John, {a, {proud, he}}}
```

5.9 Adverbs (module LEX)

Triggers selection (sMerge).

Example: *carefully*_v seeks a vP to modify in *the mechanic carefully fixed the car*.

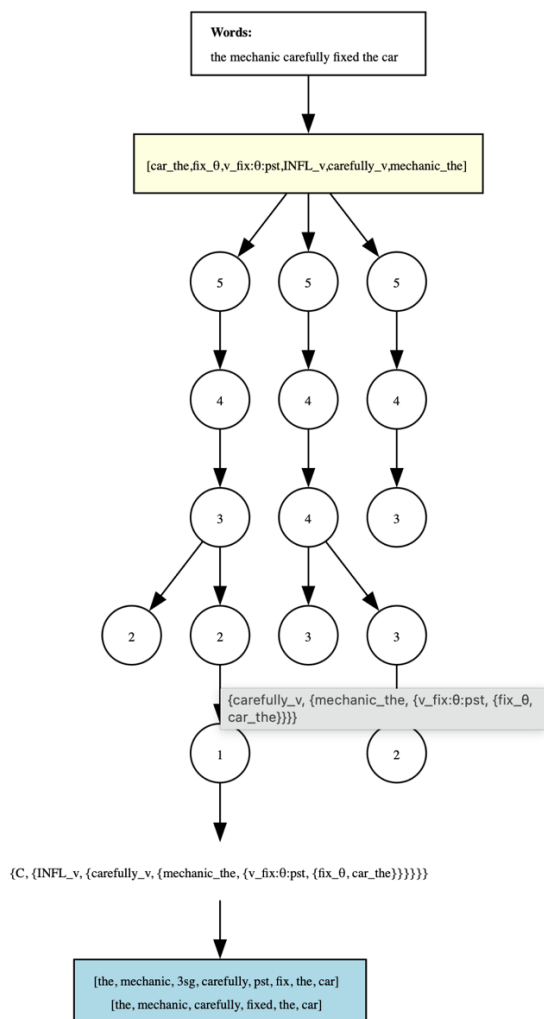
The vP θ -configuration associated with transitive verb *fix* is:

$\{\text{mechanic}_{\text{the}}, \{\text{v}_{\text{fix}:\theta:\text{pst}}, \{\text{fix}_{\theta}, \text{car}_{\text{the}}\}\}\}$

As θ -configuration construction takes precedence over other operations, *carefully* cannot be Merged until this is complete. Next, sMerge of *carefully* takes place (at the first opportunity).

$\{\text{carefully}_v, \{\text{mechanic}_{\text{the}}, \{\text{v}_{\text{fix}:\theta:\text{pst}}, \{\text{fix}_{\theta}, \text{car}_{\text{the}}\}\}\}\}$

The adverb then minimally c-commands *fix*. After *carefully*, only two heads INFL and C remain to be merged to complete the parse. The abbreviated computation graph is shown below.



`adv(Word, Select)` *Word* is an adverb, `Select = v+_` indicates it seeks a vP.

Input: *Word* an adverb form.

Output: Select the head of a phrase that adverb *Word* should modify.

In the hand-built LEX, `advWord/2` lists the entries currently available.

Some sample English and Spanish entries:

```
advWord(carefully,v+_).      % + permits keyed match
advWord(furiously,v+_).
```

```
advWord(cuidadosamente,v+_). % carefully
advWord(instintivamente,v+_). % instinctively
advWord(furiosamente,v+_).   % furiously
advWord(frecuentemente,v+_). % frequently
```

(+) keying of selected *v* is necessary as adverbs are part of the extended verbal projection (which is clustered).

EXPLAIN

To expand the hand-built coverage of English LEX, `morph/3` (controlled by `useMorph/0`) is used to find additional adverbs. For example, *slowly* is not an adverb in the hand-built LEX.

```
?- adv(slowly,S).
false.
```

```
?- useMorph.
true.
```

```
?- adv(slowly,S).      S = v.
```

By default, we assume adverbs imported by *morph* modify vP. For adverbs that are sentential in nature, one should select INFL instead of *v*, but WordNet does not distinguish between the two types.

Note that currently we also do not take advantage of regular morphology. For example, *-ly* adverbs from adjectives, e.g. *carefully* from *careful* and *slowly* from *slow*, are not yet implemented. Similarly, it should be possible to use *-mente* in Spanish, e.g. *lento* (*slow*) maps to *lento* (sg-f), then *lentamente* (*slowly*).

5.10 Prepositions (module LEX)

Triggers θ -Merge and secondary selection (`s2Merge`).

Example: $\text{by}_{\theta:v}$ in *the food was eaten by the man*.

$\text{by}_{\theta:v}$ selects for a θ -relevant object, *the man*, and θ -Merge constructs $\{\text{by}_{\theta:v}, \text{man}_{\text{the}}\}$.

Preposition $\text{by}_{\theta:v}$ selects also for a phrase headed by *v*.

This is secondary selection, applied after θ -Merge above.

The phrase is $\{v_{v:pass:pres}, \{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\}\}$, headed by $v_{v:pass:pres}$.

Head $v_{v:pass:pres}$ suppresses θ -selection (a requirement peculiar to $by_{\theta:v}$).

This is so we can identify the θ -role suppressed by $v_{v:pass:pres}$ with the one selected by $by_{\theta:v}$.

Other prepositions, e.g. *with* or *on* as in *with a telescope* or *on the boat*, do not have this secondary selection requirement on vP. In other words, θ -selection by other prepositions will not identified with the main arguments of the vP.

Example derivation with $by_{\theta:v}$.

Initial WS 3: $man_{the} \quad by_{\theta:v} \quad eat_{\theta:pastp} \quad v_{eat} \quad INFL_v \quad v_{v:pass:pres} \quad food_{the}$

WS 1: $\{eat_{\theta:pastp}, food_{the}\} \quad man_{the} \quad by_{\theta:v} \quad v_{eat} \quad INFL_v \quad v_{v:pass:pres}$

WS 2: $\{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\} \quad man_{the} \quad by_{\theta:v} \quad INFL_v \quad v_{v:pass:pres}$

WS 3: $\{v_{v:pass:pres}, \{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\}\} \quad man_{the} \quad by_{\theta:v} \quad INFL_v$

WS 4: $\{by_{\theta:v}, man_{the}\} \quad \{v_{v:pass:pres}, \{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\}\} \quad INFL_v$

WS 5: $\{\{v_{v:pass:pres}, \{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\}\}, \{by_{\theta:v}, man_{the}\}\} \quad INFL_v$

WS 6: $\{food_{the}, \{INFL_v, \{\{v_{v:pass:pres}, \{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\}\}, \{by_{\theta:v}, man_{the}\}\}\}\}$

$\{C, \{food_{the}, \{INFL_v, \{\{v_{v:pass:pres}, \{v_{eat}, \{eat_{\theta:pastp}, food_{the}\}\}\}, \{by_{\theta:v}, man_{the}\}\}\}\}\}$

the food 3sg pres be en eat by the man

Spellout: the food is eaten by the man

Passivization as main verb θ -role suppression

In the example above, passive $v_{v:pass:pres}$ selects for $v_{eat:\theta}$, which selects for an external argument (θ). Passive $v_{v:pass:pres}$ transforms $v_{eat:\theta}$ into v_{eat} , which then does not project an external argument (θ -role suppression).

5.11 EXT

5.12 Re-ordering linear operations

Φ -feature duplication

%%% AFFIXATION

%%% Cases:

%%% 1. (Usual): $\phi + TNS + vRoot \rightarrow$ inflected verb

%%% 2. $\phi + TNS + vRoot + clitic \rightarrow$ clitic + inflected verb

%%% 3. Do-support: $\phi + TNS [+DO] \neg vRoot \rightarrow$ do-insertion $\rightarrow$ inflected "do"

%%% Note: should be limited to root clauses only

spell_inflectedV/4

spell_inflectedV(Φ i, TNS, Root, Word)

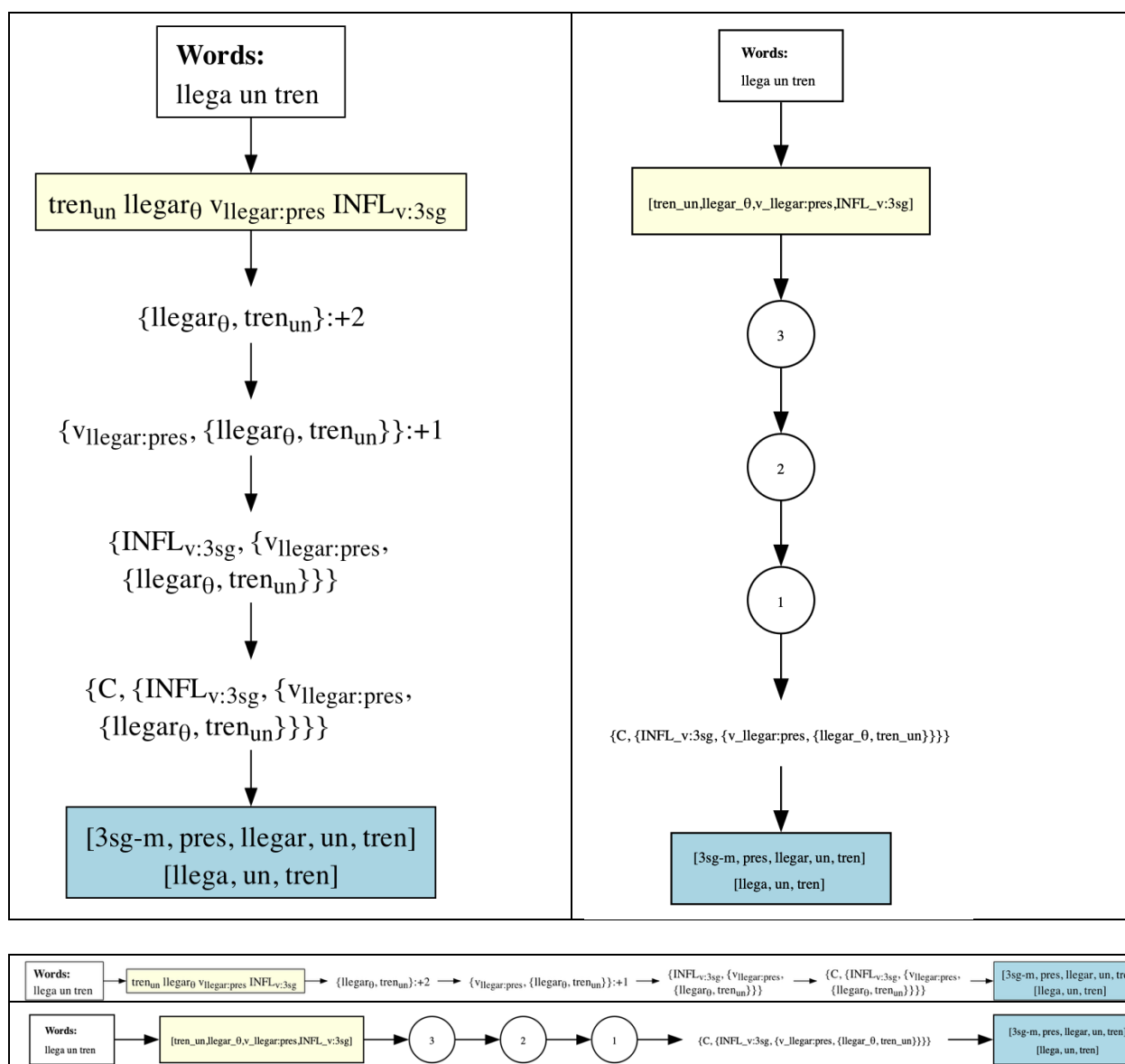
6. Computation Graphs

The post-parse command `report .` generates a simple step-by-step text view of the SMT Parser's search space. This command is available in all versions of the parser.

6.1 Four styles shown on the course webpage

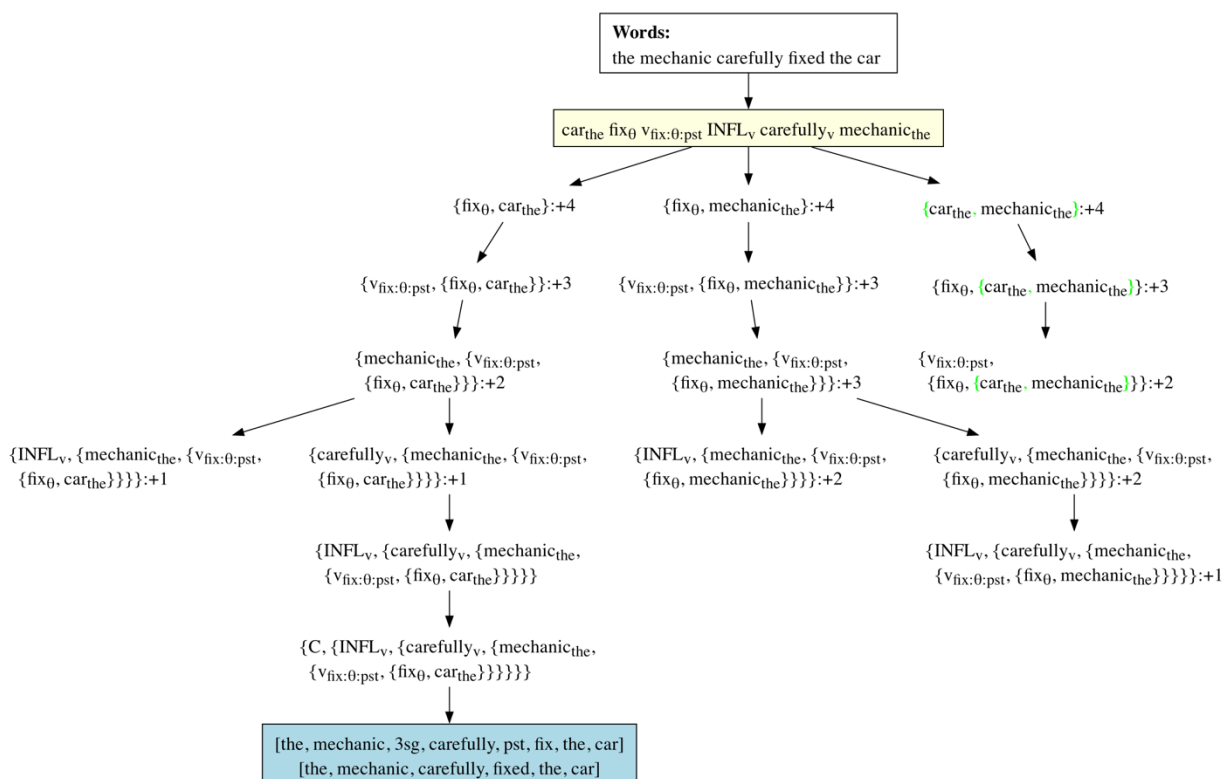
A separate post-parse command `graphall .` is available in the research version. It is used to generate the graphical representations of the computation search space for the SMT Parser.

There are four views available for all parses shown on the course website, two laid out in top-down format and two in left-to-right format. All views contain the same information. For example, here are the view for the Spanish sentence *llega un tren* (a train arrived).



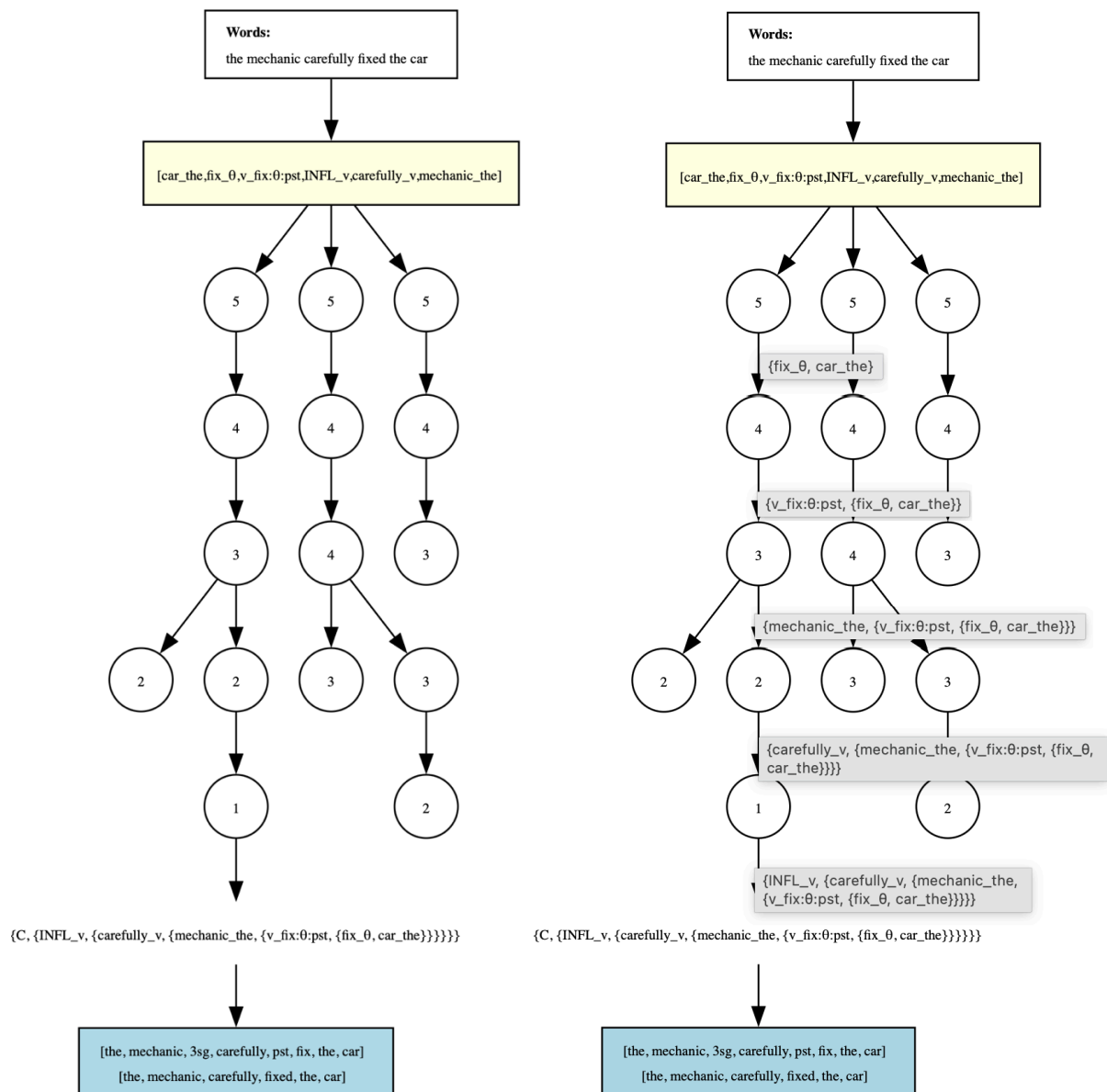
6.2 How to interpret the diagrams

Here is the complete top-down graph for the sentence *the mechanic carefully fixed the car*.



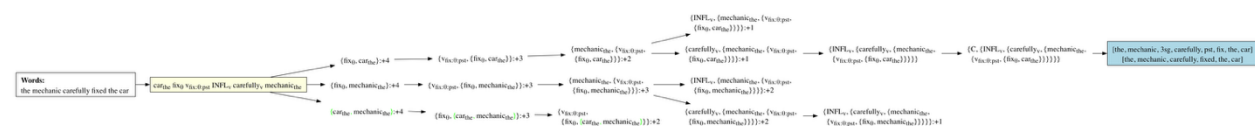
The input words to the parser are given at the top node. Lexical lookup determines the Initial **WS** represented by the rectangle immediately below (highlighted in yellow). Each structure-building step of the parser is represented by an arrow connecting an input **WS** to the next one. For example, the ternary branching indicates that the parser tried three possible **SBO** steps from the Initial **WS**, viz. (i) Merge *fix* and *car*, (ii) Merge *fix* and *mechanic*, and (iii) combine *car* and *mechanic* into a conjunct NP using **FormSet** (distinguished from **Merge** by curly brackets highlighted in green). At each succeeding **WS**, only the most recent structure built is explicitly shown. The $+n$ notation indicates that there are n other items in the **WS**. Hence, once v_{fix} merges with $\{\text{fix}_{\theta}, \text{car}_{\text{the}}\}$, the WS size decreases by 1 (from $1+4$ to $1+3$). The output of **EXT** is shown at the bottom of the computation tree (highlighted in blue).

A compact version of the top-down graph is also available. The corresponding computation space is shown below for our example *the mechanic carefully fixed the car*.

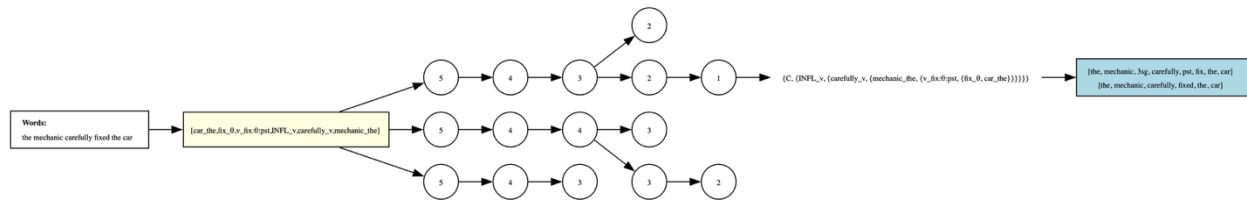


Each node represents a **WS**, with the number showing the number of **WS** items. By moving the pointer over the number, a tooltip showing the most recently built structure appears *below* the **WS** node. Tooltips are given in the diagram on the right for the successful computational thread.

In this computation space, the top-down diagrams are more convenient as the full left-to-right computation graph is very wide.

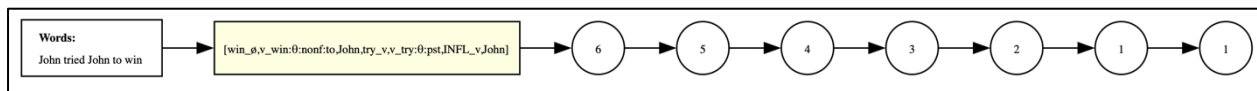


The abbreviated left-to-right graph also supports tooltips but is not much narrower.

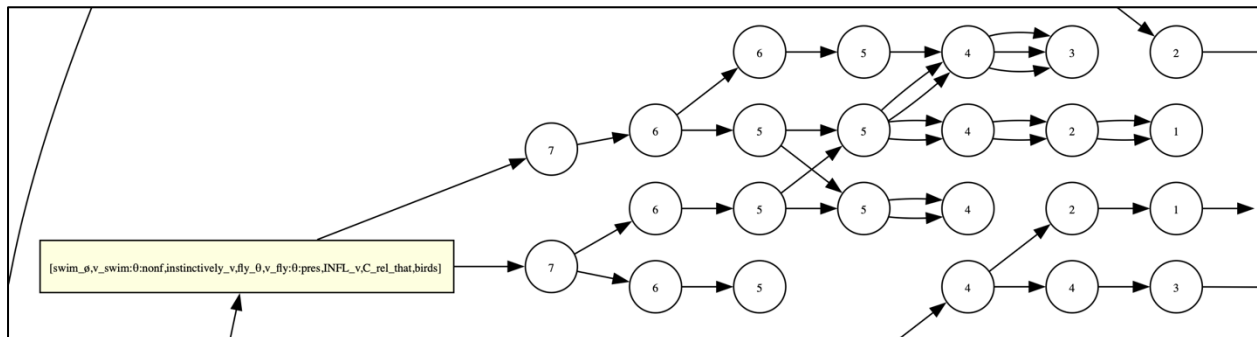


[Todo: what multiple connections mean.]

Computation graphs can be simple. For example, the derivation of the illicit M-gap example **John tried John to win* is entirely deterministic in the SMT Parser, as shown below. The parser has no choice points and ends up not converging.

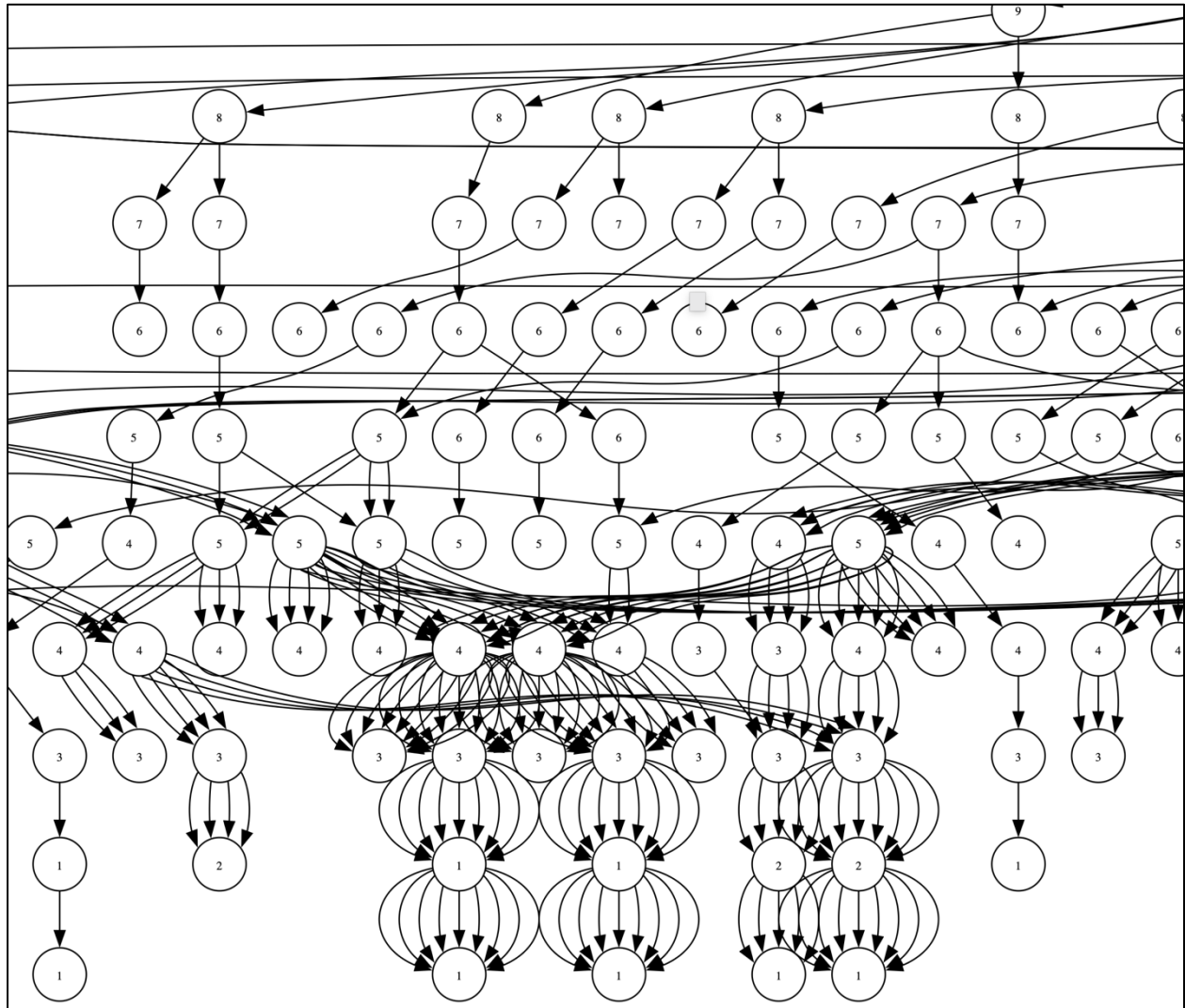


Graphs provide a global view of different threads of computation. It should be emphasized that the SMT Parser has no knowledge of this computational graph. Computational threads are independent and *Markovian*. In particular, it has no knowledge of when a particular **WS** is visited more than once, e.g. up to three times in the graph excerpt below from the parse of *birds that fly instinctively swim*.

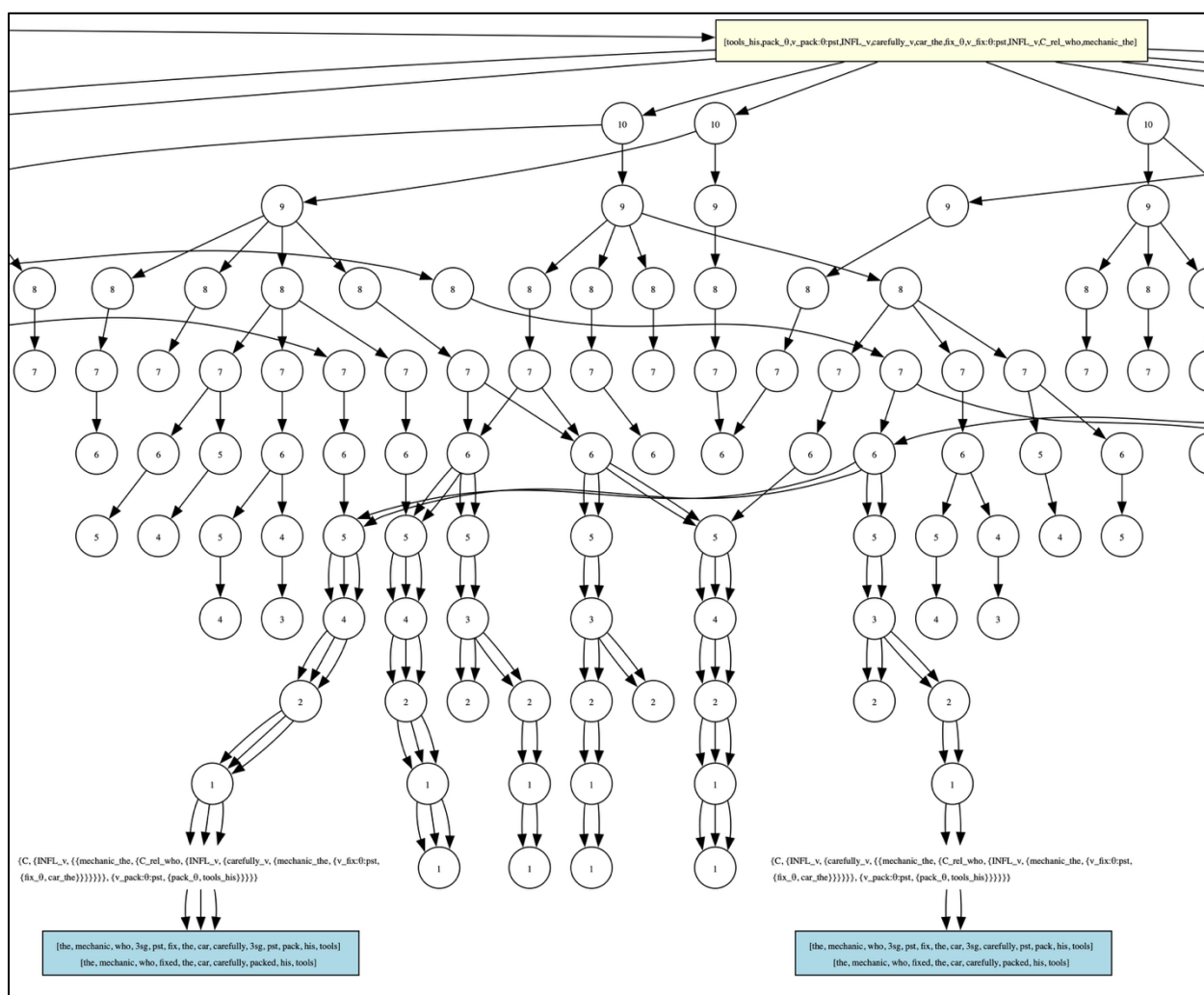


(Note: in fact, all graphing is computed post-parse using the commands `graph/graphall`.)

In some cases, there seems to be considerable redundancy or wasted effort in the computation graph, as the excerpt below from the parse of the structurally ambiguous *the mechanic who fixed the car carefully packed his tools* seems to suggest.



Above, we have some **WSs** being visited up to 9 times. In fact, diving in a bit deeper, when you examine the two successful parses, the graph reveals that when the adverb *carefully* modifies the verb *fix*, the parser computes the corresponding **SO** three times. But when adverb *carefully* modifies the other verb *pack*, it only computes the **SO** twice. The relevant part of the computation graph is shown below.



The question for *computational modeling* is whether these apparent redundancies are true redundancies or whether they are unavoidable. The short answer is that they are not redundant computations. In fact, suppose we contrive to block some of the structure-building pathways (using some stipulated computation rule) such that we only obtain the two intended parses once each. The result will be missed parses for some (grammatical) sentence. In other words, language is productive: as part of its property of *discrete infinity*, the language engine makes full use of the computational freedom afforded to the **SBOs** defined in this parser.²⁵

[Insert the relevant section from the draft *Linguistic Review* submission that demonstrates the phenomenon here?]

²⁵ Another way to think about this issue is whether an *Oracle* (using computer science terminology) exists that always can prune the computation space efficiently without missing any parse. If such an *Oracle* exists, the *Knowledge* that it encodes could be part of **I-Language** (Knowledge of Language). Or perhaps, it could be part of broader *Knowledge* (and not specific to language). Other questions include whether this *Knowledge* can be acquired. Such questions can only be tackled if we can demonstrate such an *Oracle*. Interpreting the SMT and the *Markovian assumption* broadly, we hypothesize that no such (powerful) device should exist in this framework.

7. List of Examples Tested

Each time a significant change is made to the code of the parser, a *core* set of examples is run and compared against the previous run. (These are not the complete set of examples used to develop the parser.)

Currently, there are about 200 basic English examples and 30 Spanish examples (most are grammatical). This list will grow in length and more core languages will be added.

One goal is to add existing GB theory examples from another parser.
Another is to add examples from already completed theoretical work.

7.1 English

1. parse([a,train,arrives],S,L).
2. parse([a,train,arrived],S,L).
3. parse([did,a,train,arrive],S,L).
4. parse([a,train,has,arrived],S,L).
5. parse([has,a,train,arrived],S,L).
6. parse([a,train,arrive],S,L).
7. parse(['Mary',dances],S,L).
8. parse(['Mary',danced],S,L).
9. parse(['Mary',dance],S,L).
10. parse(['John',likes,the,corridor],S,L).
11. parse(['John',liked,the,corridor],S,L).
12. parse(['John',like,the,corridor],S,L).
13. parse(['John',sees,'John'],S,L).
14. parse(['I',like,the,old,corridor],S,L).
15. parse(['I',like,the,old,narrow,corridor],S,L).
16. parse([the,old,corridor,is,narrow],S,L).
17. parse([the,narrow,corridor,is,old],S,L).
18. parse([is,the,narrow,corridor,old],S,L).
19. parse(['I',like,the,old,and,narrow,corridor],S,L).
20. parse(['I',like,the,old,narrow,and,long,corridor],S,L).
21. parse([who,sees,'John'],S,L).
22. parse([who,saw,'John'],S,L).
23. parse([who,see,'John'],S,L).
24. parse([who,does,'Mary',like],S,L).
25. parse([who,does,'Mary',likes],S,L).
26. parse([who,did,'Mary',like],S,L).
27. parse([who,did,'Mary',likes],S,L).
28. parse([who,did,'Mary',liked],S,L).
29. parse([who,'Mary',likes],S,L).
30. parse([who,'Mary',like],S,L).
31. parse([who,did,'Mary',saw],S,L).
32. parse([who,disappeared],S,L).
33. parse([disappeared,who],S,L).
34. parse([there,disappeared,who],S,L).
35. parse([who,there,disappeared],S,L).
36. parse([the,food,was,eaten],S,L).
37. parse([was,the,food,eaten],S,L).
38. parse([did,the,food,was,eaten],S,L).
39. parse([the,food,is,eaten,by,the,man],S,L).

40. parse([is, the, food, eaten, by, the, man], S, L).
41. parse([the, man, is, eaten, the, food], S, L).
42. parse([the, train, is, arrived], S, L).
43. parse([the, food, is, being, eaten], S, L).
44. parse([the, man, would, have, been, eaten, the, food], S, L).
45. parse([the, man, is, eating, the, food], S, L).
46. parse([is, the, man, eating, the, food], S, L).
47. parse([the, food, is, eating, by, the, man], S, L).
48. parse([the, man, has, eaten, the, food], S, L).
49. parse([has, the, man, eaten, the, food], S, L).
50. parse([did, the, man, have, eaten, the, food], S, L).
51. parse([the, man, has, been, eating, the, food], S, L).
52. parse([has, the, man, been, eating, the, food], S, L).
53. parse([the, man, would, have, been, eating, the, food], S, L).
54. parse([would, the, man, have, been, eating, the, food], S, L).
55. assert(use_morphy(v)).
56. assert(ΘR(old)).
57. parse([the, old, man, the, boat], S, L).
58. retract(use_morphy(v)).
59. retract(ΘR(old)).
60. parse(['John', ate, a, sandwich], S, L).
61. parse([did, 'John', eat, a, sandwich], S, L).
62. parse([sandwich, a, ate, 'John'], S, L).
63. parse([ideas, sleep], S, L).
64. parse([ideas, sleep, furiously], S, L).
65. parse([colorless, ideas, sleep], S, L).
66. parse([colorless, ideas, sleep, furiously], S, L).
67. parse([colorless, green, ideas, sleep, furiously], S, L).
68. parse([colorless, green, ideas, sleeps, furiously], S, L).
69. parse([furiously, sleep, ideas, green, colorless], S, L).
70. parse([revolutionary, new, ideas, appear, infrequently], S, L).
71. parse([do, colorless, green, ideas, sleep, furiously], S, L).
72. parse(['John', is, happy], S, L).
73. parse([is, 'John', happy], S, L).
74. parse([does, 'John', happy], S, L).
75. parse(['John', happy], S, L).
76. parse(['John', be, happy], S, L).
77. parse([the, bombing, was, criminal], S, L).
78. parse([the, bombing, were, criminal], S, L).
79. parse([the, bombings, were, criminal], S, L).
80. parse([the, bombings, was, criminal], S, L).
81. parse([the, bombing, of, the, cities, was, criminal], S, L).
82. parse([the, bombing, of, the, city, was, criminal], S, L).
83. parse([the, bombing, of, the, cities, were, criminal], S, L).
84. parse([the, bombing, of, the, city, were, criminal], S, L).
85. parse([the, bombings, of, the, city, were, criminal], S, L).
86. parse([the, bombings, of, the, cities, were, criminal], S, L).
87. parse([the, bombings, of, the, city, was, criminal], S, L).
88. parse([the, bombings, of, the, cities, was, criminal], S, L).
89. parse([the, mechanic, fixed, the, car], S, L).
90. parse([the, mechanic, carefully, fixed, the, car], S, L).
91. parse([the, mechanic, fixed, the, car, carefully], S, L).
92. parse([carefully, ' ', the, mechanic, fixed, the, car], S, L).
93. parse([did, the, mechanic, fix, the, car, carefully], S, L).
94. parse([did, the, mechanic, fixed, the, car, carefully], S, L).

95. parse([the,mechanic,packed,his,tools],S,L).
 96. parse([the,mechanic,carefully,packed,his,tools],S,L).
 97. parse([the,mechanic,packed,his,tools,carefully],S,L).
 98. parse([carefully,' ',the,mechanic,packed,his,tools],S,L).
 99. parse([who,carefully,packed,his,tools],S,L).
 100. parse([who,packed,his,tools,carefully],S,L).
 101. parse([the,mechanic,who,fixed,the,car,packed,his,tools],S,L).
 102. parse([the,mechanic,who,fixed,the,car,carefully,packed,his,tools],
 ,S,L).
 103. parse([the,mechanic,who,carefully,fixed,the,car,packed,his,tools],
 ,S,L).
 104. parse([the,mechanic,who,fixed,the,car,packed,his,tools,carefully],
 ,S,L).
 105. parse([carefully,' ',the,mechanic,who,fixed,the,car,packed,his,to
 ols],S,L).
 106. parse([the,ball,that,arrived,disappeared],S,L).
 107. parse([eagles,fly],S,L).
 108. parse([eagles,instinctively,fly],S,L).
 109. parse([eagles,fly,instinctively],S,L).
 110. parse([instinctively,' ',eagles,fly],S,L).
 111. parse([do,eagles,fly,instinctively],S,L).
 112. parse([eagles,that,instinctively,fly,swim],S,L).
 113. parse([eagles,that,fly,swim,instinctively],S,L).
 114. parse([eagles,that,fly,instinctively,swim],S,L).
 115. parse([eagles,swim],S,L).
 116. parse([eagles,can,swim],S,L).
 117. parse([eagles,that,fly,swim],S,L).
 118. parse([eagles,that,can,fly,swim],S,L).
 119. parse([eagles,that,fly,can,swim],S,L).
 120. parse([can,eagles,swim],S,L).
 121. parse([can,eagles,that,fly,swim],S,L).
 122. parse([instinctively,' ',eagles,that,fly,swim],S,L).
 123. parse([eagles,are,swimming],S,L).
 124. parse([are,eagles,swimming],S,L).
 125. parse([eagles,that,are,flying,swim],S,L).
 126. parse([eagles,that,fly,are,swimming],S,L).
 127. parse([are,eagles,that,fly,swimming],S,L).
 128. parse([are,eagles,that,swimming,fly],S,L).
 129. parse([do,eagles,that,can,fly,swim],S,L).
 130. parse([young,eagles,fly],S,L).
 131. parse([eagles,are,flying],S,L).
 132. parse([young,eagles,are,flying],S,L).
 133. parse([are,young,eagles,flying],S,L).
 134. parse([eagles,young,are,flying],S,L).
 135. parse(['John',saw,'John'],S,L).
 136. parse(['John',tried,to,win],S,L).
 137. parse(['John',tried,'John',to,win],S,L).
 138. parse(['John',wants,to,win],S,L).
 139. parse(['John',arrived],S,L).
 140. parse(['John',met,'Bill'],S,L).
 141. parse(['John',will,meet,'Bill'],S,L).
 142. parse(['John',arrived,and,met,'Bill'],S,L).
 143. parse(['John',arrived,and,meet,'Bill'],S,L).
 144. parse(['John',arrived,and,will,meet,'Bill'],S,L).
 145. parse(['John',arrived,and,will,met,'Bill'],S,L).

```

146. parse([did, 'John', arrive, and, meet, 'Bill'], S, L).
147. parse([did, 'John', arrived, and, meet, 'Bill'], S, L).
148. parse([did, 'John', arrive, and, met, 'Bill'], S, L).
149. parse([did, 'John', arrived, and, met, 'Bill'], S, L).
150. parse([who, arrived, and, met, 'Bill'], S, L).
151. parse([who, arrived, and, will, meet, 'Bill'], S, L).
152. parse([who, arrived, and, will, met, 'Bill'], S, L).
153. parse([who, arrived, and, disappeared], S, L).
154. parse([who, greeted, 'John', and, met, 'Bill'], S, L).
155. parse([who, arrived, and, who, disappeared], S, L). % error
156. parse([a, man, from, the, city, arrived, and, met, 'Bill'], S, L).
157. retract(ΘR(who)).
158. parse([the, mechanic, who, fixed, the, car, arrived, and, met, 'Bill'], S, L
    ).
159. assert(ΘR(who)).
160. parse(['John', and, 'Bill', arrived], S, L).
161. parse(['John', arrived, and, 'Bill', met, 'Mary'], S, L).
162. parse(['John', observed, the, man], S, L).
163. parse([the, man, was, observed], S, L).
164. parse([the, man, was, observed, by, 'John'], S, L).
165. parse([the, man, was, observed, 'John'], S, L).
166. parse(['John', observed, the, man, with, a, telescope], S, L).
167. parse([the, man, with, a, ball, observed, 'John'], S, L).
168. parse([the, man, with, a, ball, observed, a, boy, with, a, telescope], S, L).
169. assert(vR(be, v:be:θ, θ)).
170. assert(vR(fly, v:fly:θ, θ)).
171. parse([they, are, flying, planes], S, L).
172. retract(vR(be, v:be:θ, θ)).
173. retract(vR(fly, v:fly:θ, θ)).
174. parse(['John', is, likely, to, win], S, L).
175. parse([is, 'John', likely, to, win], S, L).
176. parse([there, arrived, a, train], S, L).
177. parse([there, is, arriving, a, train], S, L).
178. parse(['John', swims], S, L).
179. parse([there, 'John', swims], S, L).
180. parse(['John', likes, 'Mary'], S, L).
181. parse([there, 'John', likes, 'Mary'], S, L).
182. parse([they, awarded, several, prizes], S, L).
183. parse([several, prizes, were, awarded], S, L).
184. parse([prizes, were, awarded, by, 'John'], S, L).
185. parse([were, several, prizes, awarded], S, L).
186. parse([there, were, awarded, several, prizes], S, L).
187. parse([there, is, likely, to, arrive, a, man], S, L).
188. parse([there, are, likely, to, be, awarded, several, prizes], S, L).
189. parse([several, prizes, are, likely, to, be, awarded], S, L).
190. parse([several, prizes, are, expected, to, be, awarded], S, L).
191. parse(['John', was, expected, to, award, several, prizes], S, L).
192. parse([we, expected, 'John', to, dance], S, L).
193. parse(['John', was, expected, to, dance], S, L).
194. parse([we, expected, a, train, to, arrive], S, L).
195. parse([we, expect, several, prizes, to, be, awarded], S, L).
196. parse([we, expect, 'John', to, award, several, prizes], S, L).
197. parse([we, expect, to, award, several, prizes], S, L).
198. parse(['John', expects, him, to, be, sad], S, L).
199. parse(['John', expects, to, be, sad], S, L).

```

200. parse(['John', expects, he, to, be, sad], S, L).
 201. parse(['John', expected, 'Bill', to, leave], S, L).
 202. parse(['John', persuaded, 'Bill', to, leave], S, L).

7.2 Spanish

1. parse([un, tren, llega], S, L).
 2. parse([llega, un, tren], S, L).
 3. parse([llegan, dos, trenes], S, L).
 4. parse([dos, trenes, llegan], S, L).
 5. parse([llegaron, dos, trenes], S, L).
 6. parse([dos, trenes, llegaron], S, L).
 7. parse([llegó, dos, trenes], S, L).
 8. parse([dos, trenes, llegó], S, L).
 9. parse([tú, caminas, frecuentemente], S, L).
 10. parse([usted, camina, frecuentemente], S0, L).
 11. parse([las, aves, vuelan], S, L).
 12. parse(['María', comió, comida], S, L).
 13. parse([ella, comió, comida], S, L).
 14. parse([la, mujer, comió, comida], S, L).
 15. parse([el, alimento, fue, comido], S, L).
 16. parse([fue, comido, el, alimento], S, L).
 17. parse([estoy, caminando], S, L).
 18. parse([estamos, estudiando], S, L).
 19. parse(['María', está, comiendo], S, L).
 20. parse(['María', está, comiendo, la, comida], S, L).
 21. parse([me, gusta, el, libro], S, L).
 22. parse([me, gustan, los, libros], S, L).
 23. parse([a, ella, le, gusta, él], S, L).
 24. parse([él, le, gusta, a, ella], S, L).
 25. parse([le, gusta, él, a, ella], S, L).
 26. parse([le, gusta, a, ella, él], S, L).
 27. parse([le, gusta, él], S, L).
 28. parse(['Juan', me, lo, dio], S, L).
 29. parse(['María', es, orgullosa, de, ella], S, L).
 30. parse(['María', está, orgullosa, de, ellos], S, L).
 31. parse(['María', está, orgullosa, de, ellas], S, L).
 32. parse([quién, vió, a, 'Juan'], S, L).