

SMT Parser: Tutorial and Reference Manual

Sandiway Fong

University of Arizona

sandiway@arizona.edu

sandiway.arizona.edu

September 2024, updated April 2026

The tutorial section covers installation and how to modify LEX.¹

The reference manual lists the LEX facts that must be defined for the parser to operate.

Contents

1. Installation and Run with first examples (1.1 macOS, 1.2 Windows, 1.3 Linux).
Llega un tren / Dos trenes llegan / John ate a sandwich / They are flying planes
2. Tutorial: Background (LEX).
3. Tutorial: A worked example in English.
Birds that fly instinctively swim /
4. Tutorial: Another worked example in Spanish.
5. Reference Manual
 - 5.1 Basic Notation.
 - 5.2 Initial Workspace (WS).
 - 5.3 word2heads/3 (module parse).
 - 5.4 Verbs θ /3 (module θ -theory), vstem/4 (module LEX).
 - 5.5 Nouns (module LEX) θ Rnoun/1, θ R/1, nR/1.
 - 5.6 Determiners (module parse).
 - 5.7 Adjectives (module LEX) adj/2, aR/2.
 - 5.8 Adverbs (module LEX) adv/2, advWord/2.
 - 5.9 Prepositions (module LEX)
 - 5.10 EXT
 - 5.11 Re-ordering linear operations

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 invaluable Spanish LEX contributions.

Brief Glossary

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

C: *complementizer* head, e.g. C/C_{rel}/C_Q for declarative/relative/interrogative C.

EA: *external argument*, e.g. the θ -relevant item selected by v_{θ} . or a_{θ} .

EXT: language-particular module for externalization (of a **SO**).

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

INFL: *inflectional head*, usually written as INFL _{ϕ} . **MS** finds a θ -relevant item valuing ϕ .

LEX: *lexicon*. Contains heads that enter the initial **WS**.

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

Merge: *set Merge* of items from a **WS**. Resulting set is stored back into the **WS**.

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

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

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

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

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

R: root, e.g. verbal root *eat*. Written with a subscript θ if it selects for an **IA**.

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

SMT: *Strong Minimalist Thesis*.

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

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

θ -configuration: a **SO** containing arguments and predicate, e.g. for transitive *eat*, a configuration {EA, {v_{eat: θ :TNS}, {eat _{θ} , IA}}}} with arguments EA and 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 highest **v**.

v: “little v” verbal head, e.g. v_{eat: θ :pst} selects for verbal root *eat*, introduces a θ -relevant item, an external argument, and has TNS value pst (*past*).

WS: *workspace* for Merge. Heads from LEX populate an initial WS.

1. Installation and Run

The *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 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.

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

```
$ sudo port install swi-prolog      (MacPorts)
$ brew install swi-prolog           (Homebrew)
(Note: $ 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. Assuming your `PATH` environment variable has been set correctly during installation, the 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 directories to your `PATH` variable for convenience.

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

Prolog interfaces with Python via the Janus library. As there are possibly (and perhaps, likely to be) multiple Pythons 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)
```

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 reaches stability under development, more and more of the source code will become publicly available.

```

?- 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, and 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
$ /opt/local/bin/python3.13 -m ensurepip --upgrade

```

(MacPorts, in this case)

Then install `nltk` using `pip`:

```

$ /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()

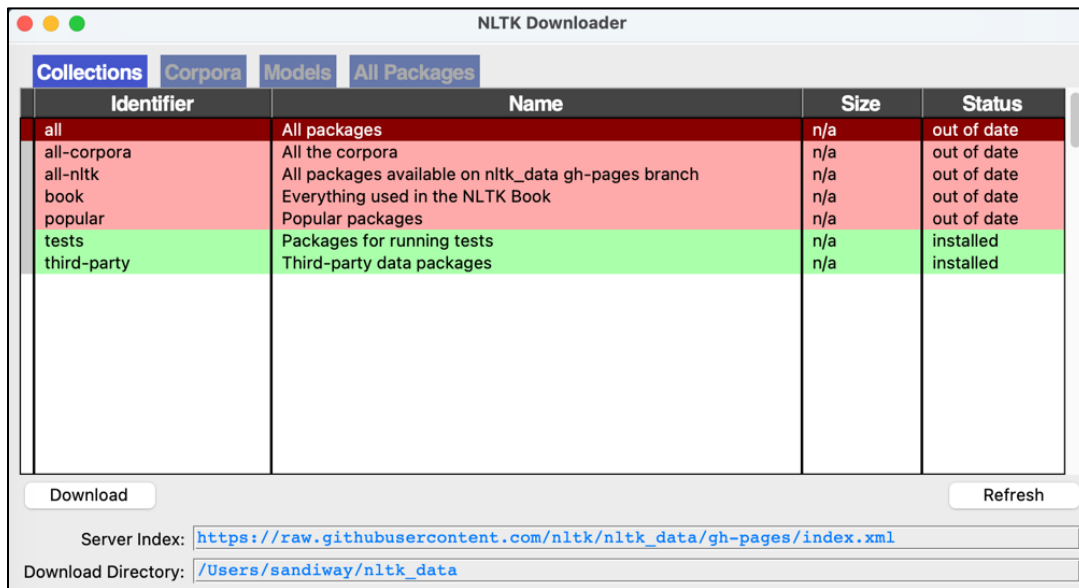
```

```

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 (`~` is shorthand for your home directory), and can be shared across multiple Python installations. 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. 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 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. **false** after ; means no other solutions have been found. Or just press the <return> key to stop the search.)

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

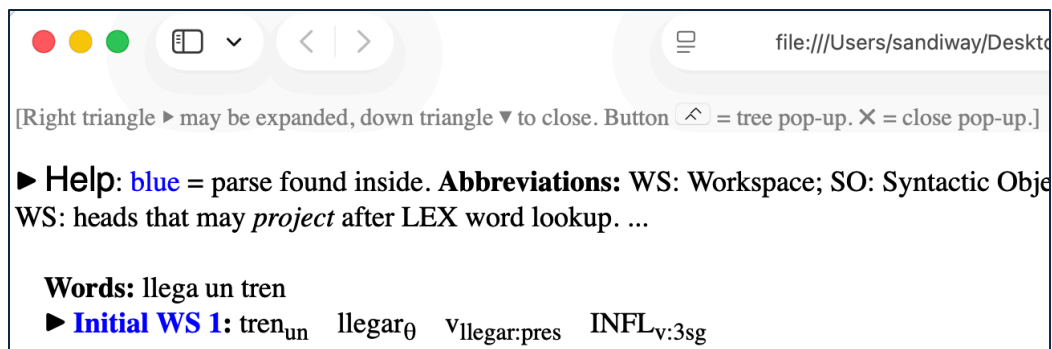
Spellout: llegan dos trenes

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) the Syntactic Object (named above as `SO`, a variable), and (iii) the list of words resulting from EXT of `SO` (named using variable `L`). In this 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, it should automatically pop up the 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:

```
file:///Users/sandaway/Desktop/

[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; WS: heads that may project after LEX word lookup. ...

Words: llega un tren
▼ Initial WS 1: trenun llega0 vllega:pres INFLv:3sg
  ▼ WS 1: {llega0, trenun} vllega:pres INFLv:3sg
    WS 2: {vllega:pres, {llega0, trenun}} INFLv:3sg
    WS 3: {INFLv:3sg, {vllega:pres, {llega0, trenun}}}}
    Final WS: {C, {INFLv:3sg, {vllega:pres, {llega0, trenun}}}}
INT/EXT: ⤴ {C, {INFLv:3sg, {vllega:pres, {llega0, trenun}}}}
▼ Initial Spellout: un tren 3sg-m pres llega
  Spellout: un tren llega
Error: inconsistent spellout
▼ Initial Spellout: 3sg-m pres llega un tren
  Spellout: llega un tren
Parse found: llega un tren
▼ WS 1: {llega0, trenun} vllega:pres INFLv:3sg
  WS 2: {{llega0, trenun}, vllega:pres} INFLv:3sg
▼ WS 1: {vllega:pres, llega0} trenun INFLv:3sg
  WS fail: WS 0S 0 < 0R 1
▼ WS 1: {llega0, vllega:pres} trenun INFLv:3sg
  WS fail: WS 0S 0 < 0R 1
```

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

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%`.

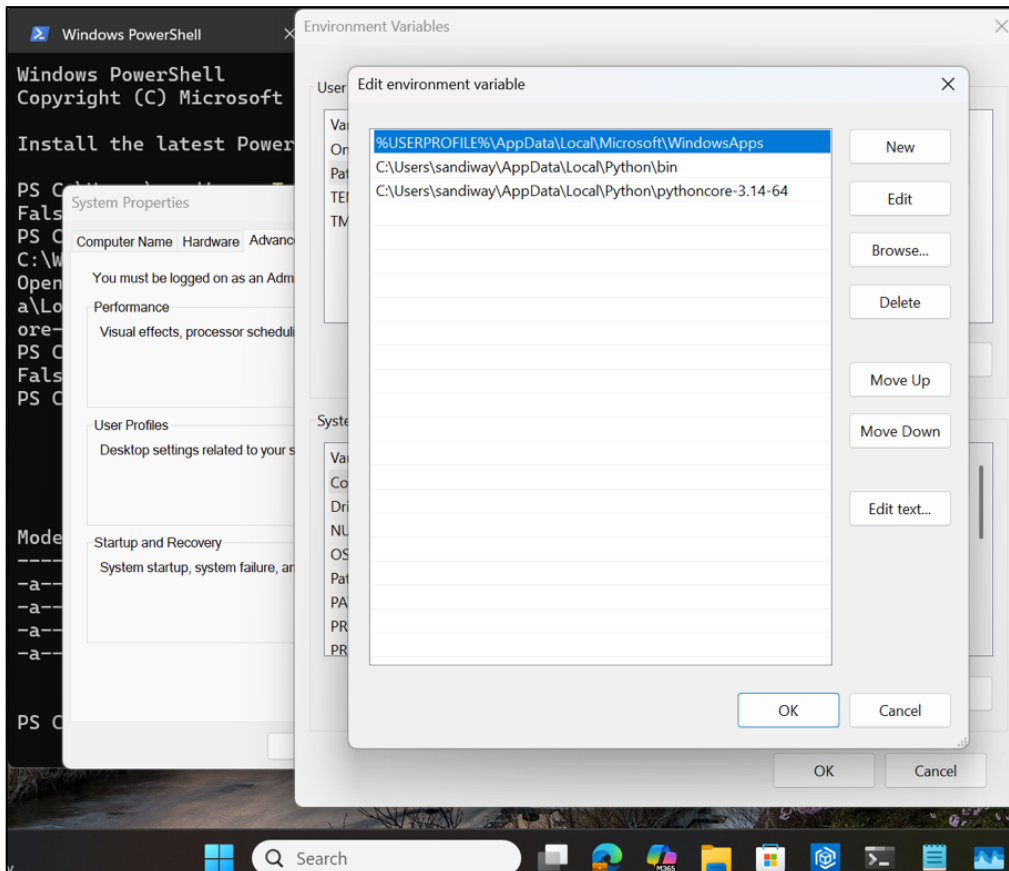
Python from python.org will be installed in `AppData\Local\Python`. Check the specific directory under Python for `python3.dll`. In my case, I have Python 3.14 installed, so the relevant subdirectory is `pythoncore-3.14-64`. (-64 here stands for 64 bits.)

```
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.



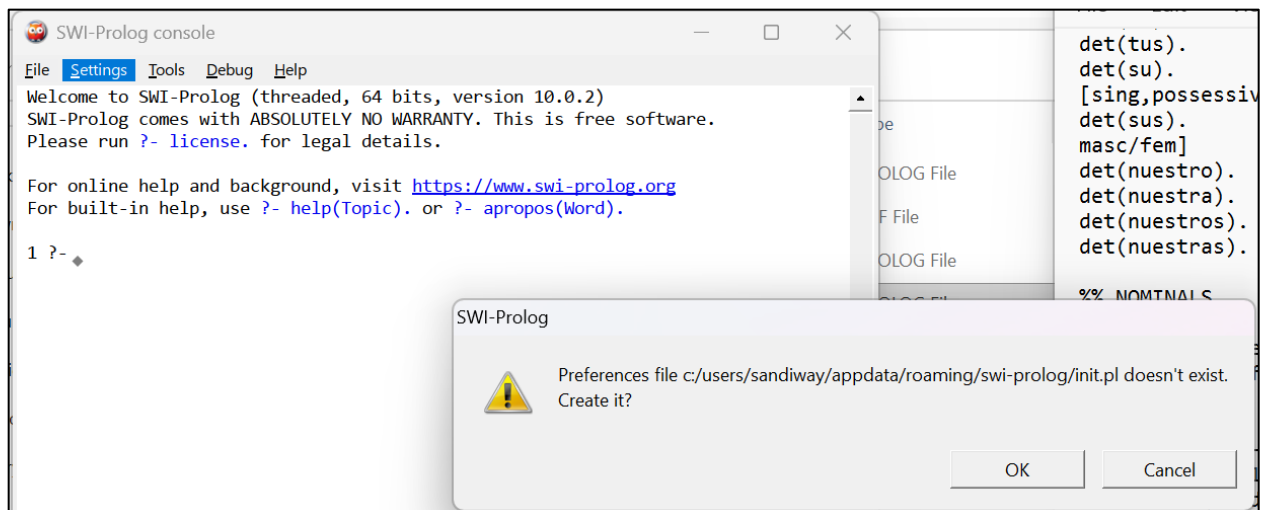
The SMT Parser makes use of the now-standard Unicode utf-8 variable width character encoding format (*Unicode Transformation Format-8*). In macOS and Linux, this is the default, but not on Windows 11 it seems. In Windows PowerShell, check the character set encoding as follows:

```
$ 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

In my case, the directory is under my home directory is the *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.³ 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.)

³ `.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

C:\Users\sandiway\AppData\Roaming\swi-prolog
dir-history
xpce

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

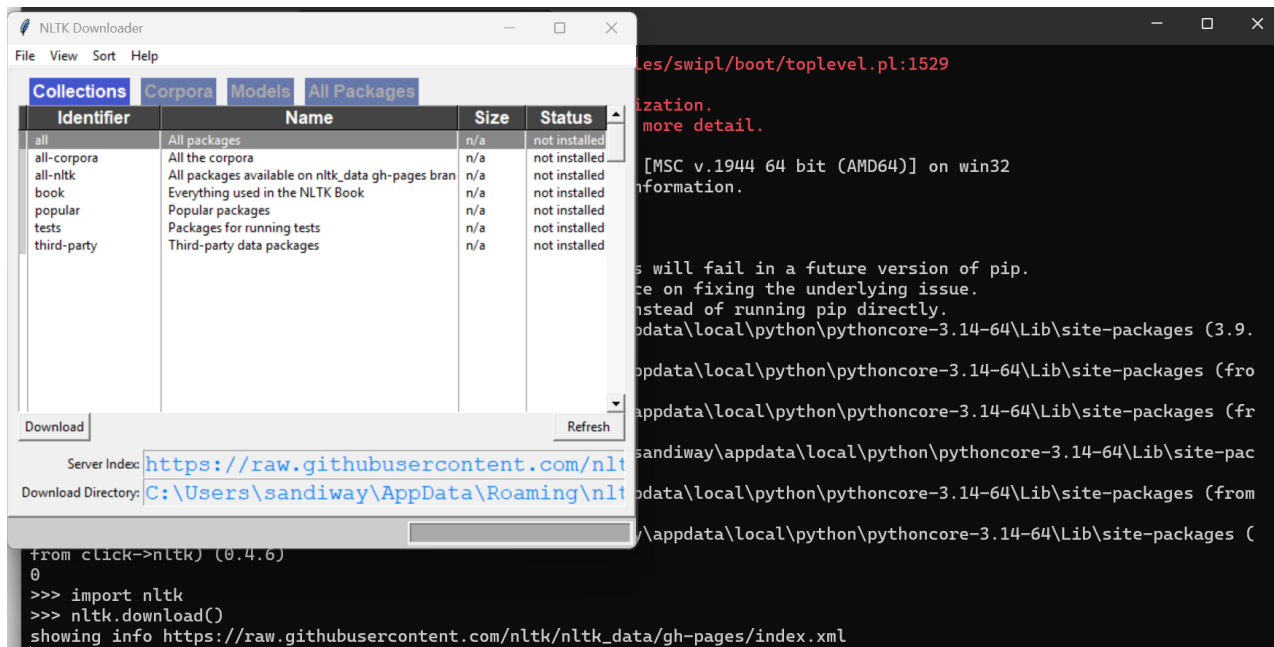
In PowerShell, make sure pip is installed. py is the command for Python.

```
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`.



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

In PowerShell, first ensure you are in the directory with the SMT Parser files. 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!
false.

```

```
2 ?- report.
```

```
REPORT LENGTH: 32
true.
```

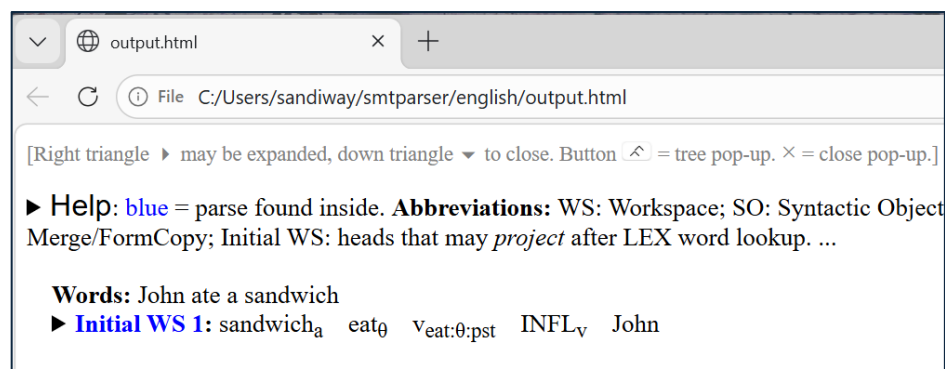
(Note that the input word *John* ('John') must be quoted because Prolog variables begin with an uppercase letter. Without quotes, John would be a variable. Atoms normally do not require quotes in Prolog, cf. Python.)

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 Syntactic Object (named above as `S0`, a variable), and (iii) the list of words resulting from EXT of `S0` (named using 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, one I-Language structure is built 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. The file should automatically appear in the 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:

```

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

```

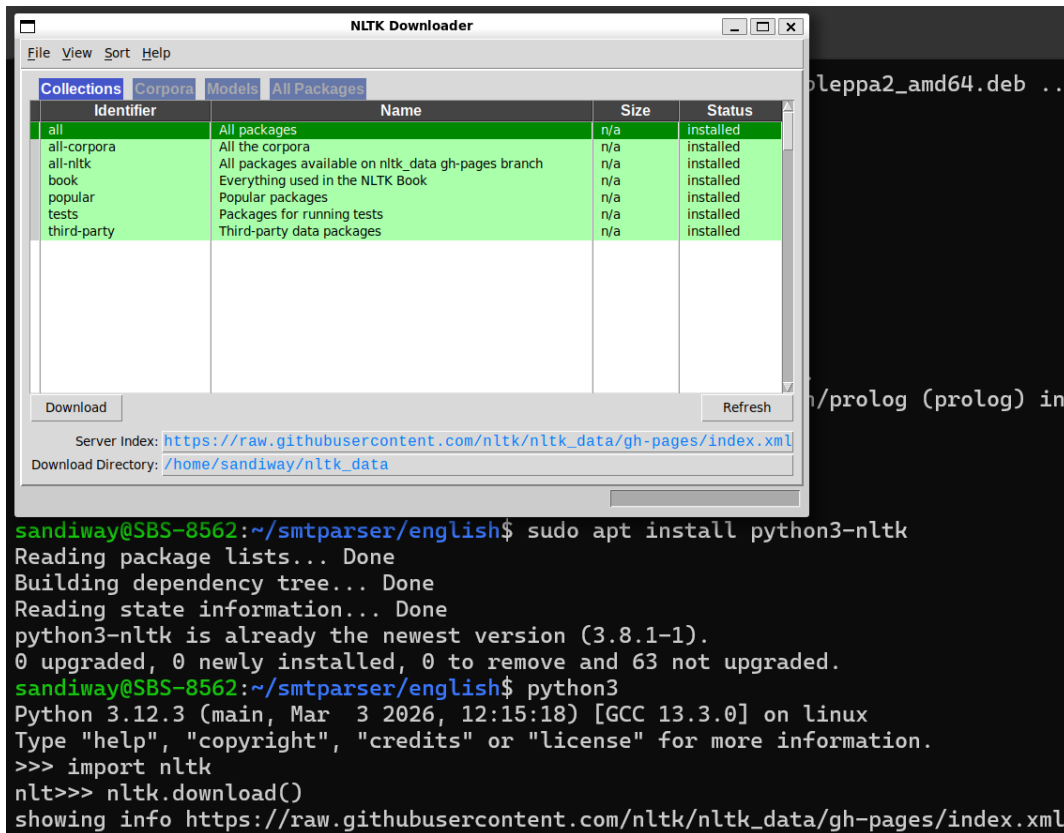
1.3 SWI Prolog (Linux)

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. 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]
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([they,are,flying,planes],S0,L).
```

```
Words: they are flying planes
Initial WS: planes fly_0:presp v_fly:0 INFL_v v_pred INFL_v:2sg they
Initial WS: planes flying_0 v_pred INFL_v:2sg they
Initial WS: planes fly_0:presp v_fly:0 INFL_v be_0 v_be:0:pres
INFL_v:2sg they
```

```

Initial WS: planes flying_0 be_0 v_be:0:pres INFL_v:2sg they
INT/EXT: {C, {INFL_v:2sg, {they, {v_be:0:pres, {be_0, {planes, {flying_0,
planes}}}}}}}}
Initial Spellout: they 3pl pres be flying planes
Spellout: they are flying planes
Parse found!
S0 = {C, {INFL_v:2sg, {they, {v_be:0:pres, {be_0, {planes, {flying_0,
planes}}}}}}},
L = [they, are, flying, planes] ;

INT/EXT: {C, {INFL_v:2sg, {{planes, {flying_0, planes}}, {v_be:0:pres, {be_0,
they}}}}}}
Error: Case: object "they" not ACC
INT/EXT: {C, {INFL_v:2sg, {planes, {v_be:0:pres, {be_0, {they, {flying_0,
they}}}}}}}}
Error: Case: object "they" not ACC
INT/EXT: {C, {INFL_v:2sg, {{they, {flying_0, they}}, {v_be:0:pres, {be_0,
planes}}}}}}
Initial Spellout: flying they they flying they 3pl pres be planes
Spellout: flying they they flying they are planes
Error: spellout inconsistent with original words!
Initial WS: planes v_fly:0 fly_0:presp INFL_v v_v:prog:pres they
INT/EXT: {C, {INFL_v, {v_v:prog:pres, {they, {v_fly:0, {fly_0:presp,
planes}}}}}}}}
Initial Spellout: they 3pl pres be ing fly planes
Spellout: they are flying planes
Parse found!
S0 = {C, {INFL_v, {v_v:prog:pres, {they, {v_fly:0, {fly_0:presp,
planes}}}}}}},
L = [they, are, flying, planes] ;

INT/EXT: {C, {INFL_v, {v_v:prog:pres, {planes, {v_fly:0, {fly_0:presp,
they}}}}}}}}
Error: Case: object "they" not ACC
Initial WS: planes flying_0 v_v:prog:pres they
Initial WS: planes flying_0 v_v:pass:pres they
false.

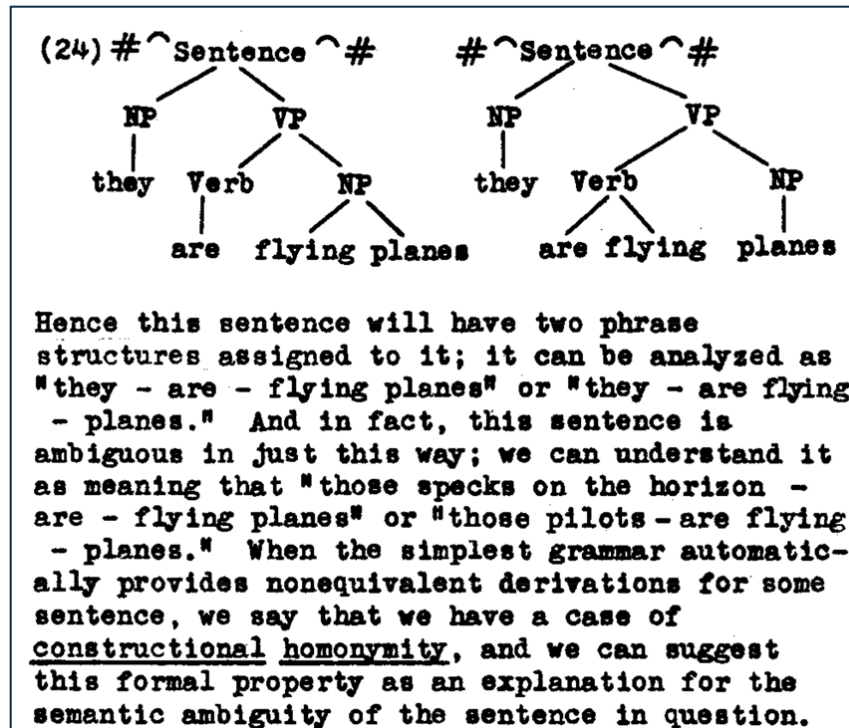
```

?- 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 Syntactic Object (named above as S0, a variable), and (iii) the list of words spelled out (named using 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.

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

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

Words: they are flying planes

- **Initial WS 1:** planes fly_θ:presp v_{fly}:θ INFL_v v_{pred} INFL_{v:2sg} they
- **Initial WS 2:** planes flying_θ v_{pred} INFL_{v:2sg} they
- **Initial WS 3:** planes fly_θ:presp v_{fly}:θ INFL_v be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **Initial WS 4:** planes flying_θ be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **Initial WS 5:** planes v_{fly}:θ fly_θ:presp INFL_v v_{v:prog:pres} they
- **Initial WS 6:** planes flying_θ v_{v:prog:pres} they
- **Initial WS 7:** planes flying_θ v_{v:pass:pres} they

(For simplicity, LEX has been modified above to show only the transitive variant of the verb *fly*.)

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

Example (*opening Initial WS 4 only*):

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

► **Help:** blue = parse found inside. **Abbreviations:** WS: Workspace; SO: Syntactic Object; IA
  interface SO post-Merge/FormCopy; Initial WS: heads that may *project* after LEX word loc

Words: they are flying planes

- **Initial WS 1:** planes fly_{θ:presp} v_{fly:θ} INFL_v v_{pred} INFL_{v:2sg} they
- **Initial WS 2:** planes flying_θ v_{pred} INFL_{v:2sg} they
- **Initial WS 3:** planes fly_{θ:presp} v_{fly:θ} INFL_v be_θ v_{be:θ:pres} INFL_{v:2sg} they
- ▼ **Initial WS 4:** planes flying_θ be_θ v_{be:θ:pres} INFL_{v:2sg} they
 - ▼ **WS 1:** {planes, {flying_θ, planes}} be_θ v_{be:θ:pres} INFL_{v:2sg} they
 - WS 2: {be_θ, {planes, {flying_θ, planes}}}} v_{be:θ:pres} INFL_{v:2sg} they
 - WS 3: {v_{be:θ:pres}, {be_θ, {planes, {flying_θ, planes}}}}}} INFL_{v:2sg} they
 - WS 4: {they, {v_{be:θ:pres}, {be_θ, {planes, {flying_θ, planes}}}}}} INFL_{v:2sg}
 - WS 5: {INFL_{v:2sg}, {they, {v_{be:θ:pres}, {be_θ, {planes, {flying_θ, planes}}}}}}}}}
 - Final WS:** {C, {INFL_{v:2sg}, {they, {v_{be:θ:pres}, {be_θ, {planes, {flying_θ, planes}}}}}}}}}

INT/EXT:  {C, {INFL_{v:2sg}, {they, {v_{be:θ:pres}, {be_θ, {planes, {flying_θ, planes}}}}}}}}

► **Initial Spellout:** they 3pl pres be flying planes

Parse found: they are flying planes

- **WS 1:** {planes, {flying_θ, planes}} be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **WS 1:** {planes, {flying_θ, planes}} be_θ v_{be:θ:pres} INFL_{v:2sg} they

INT/EXT:  {C, {INFL_{v:2sg}, {{planes, {flying_θ, planes}}, {v_{be:θ:pres}, {be_θ, they}}}}}}

Error: Case: object "they" not ACC

- **WS 1:** {planes, {flying_θ, planes}} be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **WS 1:** {planes, {flying_θ, planes}} be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **WS 1:** {planes, {flying_θ, planes}} be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **WS 1:** {they, {flying_θ, they}} planes be_θ v_{be:θ:pres} INFL_{v:2sg}

INT/EXT:  {C, {INFL_{v:2sg}, {planes, {v_{be:θ:pres}, {be_θ, {they, {flying_θ, they}}}}}}}}

Error: Case: object "they" not ACC

Example (opening Initial WS 5 only):

/home/sandway/smtpars: x +
 file:///home/sandway/smtparser/english/output.html

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

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

Words: they are flying planes

- **Initial WS 1:** planes fly_θ:presp v_{fly:θ} INFL_v v_{pred} INFL_{v:2sg} they
- **Initial WS 2:** planes flying_θ v_{pred} INFL_{v:2sg} they
- **Initial WS 3:** planes fly_θ:presp v_{fly:θ} INFL_v be_θ v_{be:θ:pres} INFL_{v:2sg} they
- **Initial WS 4:** planes flying_θ be_θ v_{be:θ:pres} INFL_{v:2sg} they
- ▼ **Initial WS 5:** planes v_{fly:θ} fly_θ:presp INFL_v v_{v:prog:pres} they
 - ▼ **WS 1:** {fly_θ:presp, planes} v_{fly:θ} INFL_v v_{v:prog:pres} they
 - WS 2:** {v_{fly:θ}, {fly_θ:presp, planes}} INFL_v v_{v:prog:pres} they
 - WS 3:** {they, {v_{fly:θ}, {fly_θ:presp, planes}}}} INFL_v v_{v:prog:pres}
 - WS 4:** {v_{v:prog:pres}, {they, {v_{fly:θ}, {fly_θ:presp, planes}}}}}} INFL_v
 - WS 5:** {INFL_v, {v_{v:prog:pres}, {they, {v_{fly:θ}, {fly_θ:presp, planes}}}}}}}} INFL_v
 - Final WS:** {C, {INFL_v, {v_{v:prog:pres}, {they, {v_{fly:θ}, {fly_θ:presp, planes}}}}}}}}}
- INT/EXT: {C, {INFL_v, {v_{v:prog:pres}, {they, {v_{fly:θ}, {fly_θ:presp, planes}}}}}}}}
- **Initial Spellout:** they 3pl pres be ing fly planes
- Parse found:** they are flying planes
 - **WS 1:** {fly_θ:presp, planes} v_{fly:θ} INFL_v v_{v:prog:pres} they
 - **WS 1:** {fly_θ:presp, they} planes v_{fly:θ} INFL_v v_{v:prog:pres}
 - INT/EXT: {C, {INFL_v, {v_{v:prog:pres}, {planes, {v_{fly:θ}, {fly_θ:presp, they}}}}}}}}
 - Error:** Case: object "they" not ACC
 - **WS 1:** {fly_θ:presp, they} planes v_{fly:θ} INFL_v v_{v:prog:pres}
 - **WS 1:** {v_{fly:θ}, fly_θ:presp} planes INFL_v v_{v:prog:pres} they
 - **WS 1:** {v_{fly:θ}, fly_θ:presp} planes INFL_v v_{v:prog:pres} they
- **Initial WS 6:** planes flying_θ v_{v:prog:pres} they
- **Initial WS 7:** planes flying_θ v_{v:pass:pres} they

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 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 Workspace. 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 determines the available heads (and their properties) for a given language.

Currently, the hand-built English lexicon 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.

Assuming you’re in the same directory as the file, to load LEX at any time, 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 the words to LEX to handle the following (Berwick & Chomsky, 2016) sentence:

(1) Birds that fly instinctively swim

In Terminal, `cd` to the `smtparser` 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`).

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

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

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

Note: `code_English.prolog` above should be `code_Spanish.prolog` in the case of Spanish. The Reference Manual section explains each of the queries used in the worked example.

If you are unfamiliar with Prolog syntax or the interpreter, consult the SWI-Prolog quickstart section (<https://www.swi-prolog.org/pldoc/man?section=quickstart>) first.

Adding a noun: `θR/1`, `num/2` and `word2heads/3`

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

Nouns are θ -relevant heads. LEX has a table `θR/1` (arity: 1) listing these heads. For example, we might expect *bird* (noun) to be present.

```
?- θR(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:

```
θR(bird).
```

(Note: new entries should be placed next to other `θR/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.  
  
?- θR(bird).  
true.
```

Next, test the `NUM` plural (`pl`) feature value.⁴ 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.
```

⁴ 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)`.

```
?- 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.
```

From here onwards, we proceed assuming the 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: $\theta$ , INFL_v],  
As = [] ;  
Hd = [fly_ $\emptyset$ , v_fly: $\theta$ , INFL_v],  
As = [] ;  
false.
```

The WordNet LEX also lists *fly* as a noun. We will generally have additional entries if *noMorphology* 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_ $\theta$  v_fly: $\theta$ :pres INFL_v birds
```

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

Initial Spellout: birds 3pl pres fly birds

Spellout: birds fly birds

Error: spellout inconsistent with original words!

Initial WS: fly_θ v_fly:θ:pres INFL_v birds

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

Initial Spellout: birds 3pl pres fly

Spellout: birds fly

Parse found!

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

L = [birds, fly] ;

Initial WS: fly_θ v_fly:θ INFL_v birds

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

Initial Spellout: birds 3pl fly birds

Error: Missing TNS

Initial WS: fly_θ v_fly:θ INFL_v birds

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

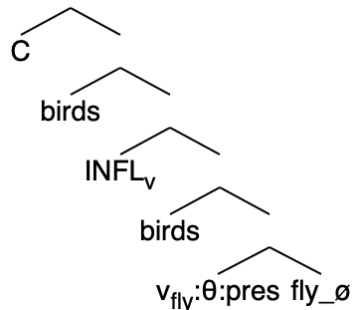
Initial Spellout: birds 3pl fly

Error: Missing TNS

false.

Let's examine the parse below briefly. 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 TNS pres, as indicated by head v_fly:θ:pres. FormCopy ensures only the top copy of *birds* at the surface subject position (spec-INFL) is pronounced, the lower copy in θ-position is necessary for INT.

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



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_θ, v_swim:θ:pres, INFL_v],

As = [] ;

Hd = [swim_θ, v_swim:θ, INFL_v],

As = [] ;

false.

This is confirmed by the vR/3 lexical entry:

vR(swim, v:swim:θ, θ). % eagles swim

that

In this implementation, relative clauses are introduced by a relative pronoun C_{rel} . If you look in LEX for relevant information concerning ‘ C_{rel} ’, you will find a definition for (special word) `spWord/2`:

```
%% for relativization, at parse time
%%  spellout feature word(W)

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

[Note: in LEX, there is both `spWord/1` and `spWord/2`. These should be consolidated in a future update.]

The listing in `spWord/2` means the parser can convert an input word *that* into the head ' C_{rel} '. The feature `word(that)` is not used in Merge. Using `word2heads/3`, we see:

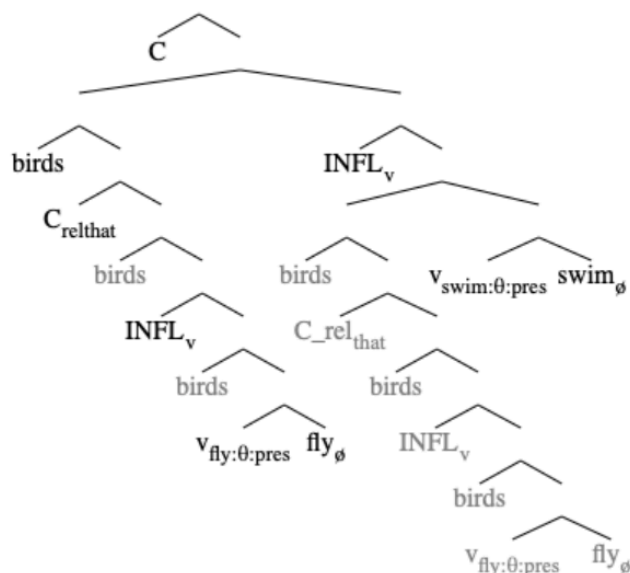
```
?- word2heads(that,Hs,As).
Hs = [C_rel:word(that)],
As = [] ;
false.
```

We can check LEX lookup for the entire sentence using `words2heads/3`. (We omit `As` below for expository brevity.) 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. WS = [swim_ø, v_swim:θ:pres, INFL_v, fly_θ, v_fly:θ:pres, INFL_v, C_rel_that,
birds] ;
2. WS = [swim_ø, v_swim:θ, INFL_v, fly_θ, 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, INFL_v, fly_ø, v_fly:θ:pres, INFL_v, C_rel_that, birds] ;
5. WS = [swim_ø, v_swim:θ:pres, INFL_v, fly_θ, v_fly:θ, INFL_v, C_rel_that, birds] ;
6. WS = [swim_ø, v_swim:θ, INFL_v, fly_θ, v_fly:θ, INFL_v, C_rel_that, birds] ;
7. WS = [swim_ø, v_swim:θ:pres, INFL_v, fly_ø, v_fly:θ, INFL_v, C_rel_that, birds] ;
8. WS = [swim_ø, v_swim:θ, INFL_v, fly_ø, v_fly:θ, INFL_v, C_rel_that, birds] ;
false.
```

The parse given by Initial WS #3 above is:

```
{{birds, {C_relthat, {birds, {INFL_v, {birds, {v_swim:θ:pres, swim_ø}}}}}}, {INFL_v,
{birds, {C_relword(that), {birds, {INFL_v, {birds, {v_swim:θ:pres, swim_ø}}}}}}, {v_fly:θ:pres, fly_ø}}}}
```



[Note: there is a slight discrepancy in rendering between $C_{relthat}$ and $C_{rel_{that}}$. Not critical, but should be fixed.]

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_0 v_swim:0:pres INFL_v fly_0 v_fly:0:pres INFL_v C_rel_that birds

INT/EXT: {C, {birds, {INFL_v, {birds, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}}}

Initial Spellout: birds 3pl pres fly birds that 3pl pres swim

Spellout: birds fly birds that swim

Error: spellout inconsistent with original words!

INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, {fly_0, birds}}}}}}

Initial Spellout: birds that 3pl pres swim 3pl pres fly birds

Spellout: birds that swim fly birds

Error: spellout inconsistent with original words!

INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}

Initial Spellout: birds that 3pl pres swim 3pl pres fly birds that 3pl pres swim

Spellout: birds that swim fly birds that swim

Error: spellout inconsistent with original words!

INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}

Initial Spellout: birds that 3pl pres swim 3pl pres fly birds that 3pl pres swim

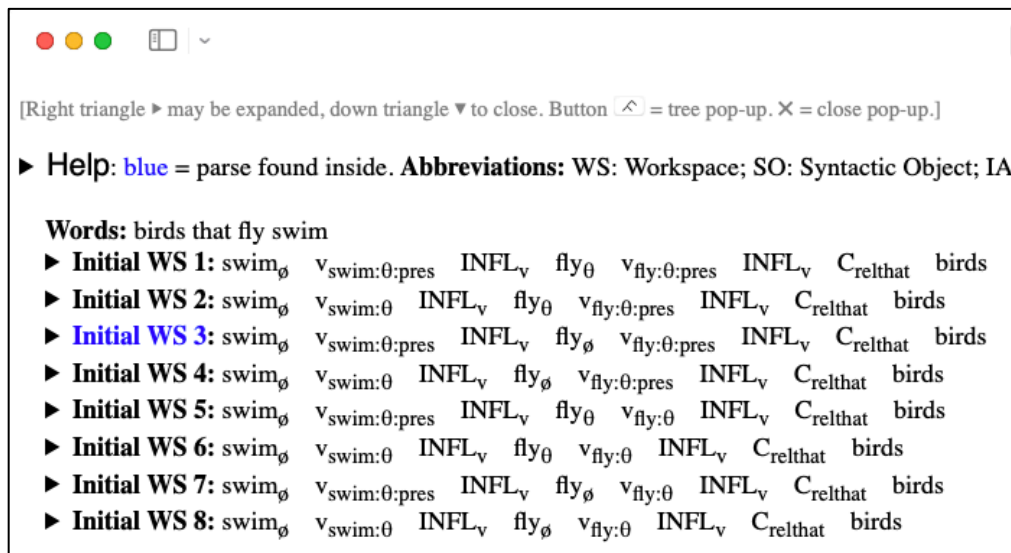
Spellout: birds that swim fly birds that swim

Error: spellout inconsistent with original words!

INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, {fly_0, birds}}}}}}}
Initial Spellout: birds that 3pl pres swim 3pl pres fly birds
Spellout: birds that swim fly birds
Error: spellout inconsistent with original words!
INT/EXT: {C, {birds, {INFL_v, {birds, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}}}}
Initial Spellout: birds 3pl pres fly birds that 3pl pres swim
Spellout: birds fly birds that swim
Error: spellout inconsistent with original words!
INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}}}}
Initial Spellout: birds that 3pl pres swim 3pl pres fly birds that 3pl pres swim
Spellout: birds that swim fly birds that swim
Error: spellout inconsistent with original words!
INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}}}}
Initial Spellout: birds that 3pl pres swim 3pl pres fly birds that 3pl pres swim
Spellout: birds that swim fly birds that swim
Error: spellout inconsistent with original words!
INT/EXT: {C, {birds, {INFL_v, {birds, {v_fly:0:pres, {fly_0, {birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}}}}}}}}
Initial Spellout: birds 3pl pres fly birds that 3pl pres swim
Spellout: birds fly birds that swim
Error: spellout inconsistent with original words!
Initial WS: swim_0 v_swim:0 INFL_v fly_0 v_fly:0:pres INFL_v C_rel_that birds
Initial WS: swim_0 v_swim:0:pres INFL_v fly_0 v_fly:0:pres INFL_v C_rel_that birds
INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_swim:0:pres, swim_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_swim:0:pres:, swim_0:}}}}}}, {v_fly:0:pres, fly_0}}}}}
Initial Spellout: birds that 3pl pres swim 3pl pres fly
Spellout: birds that swim fly
Error: spellout inconsistent with original words!
INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_fly:0:pres, fly_0}}}}}}, {INFL_v, {{birds, {C_rel_that:, {birds, {INFL_v:, {birds, {v_fly:0:pres:, fly_0:}}}}}}, {v_swim:0:pres, swim_0}}}}}
Initial Spellout: birds that 3pl pres fly 3pl pres swim
Spellout: birds that fly swim
Parse found!
S0 = {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_fly:0:pres, fly_0}}}}}}, {INFL_v, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_fly:0:pres, fly_0}}}}}}, {v_swim:0:pres, swim_0}}}}},
L = [birds, that, fly, swim] ;

Initial WS: swim_0 v_swim:0 INFL_v fly_0 v_fly:0:pres INFL_v C_rel_that birds
Initial WS: swim_0 v_swim:0:pres INFL_v fly_0 v_fly:0 INFL_v C_rel_that birds
Initial WS: swim_0 v_swim:0 INFL_v fly_0 v_fly:0 INFL_v C_rel_that birds
Initial WS: swim_0 v_swim:0:pres INFL_v fly_0 v_fly:0 INFL_v C_rel_that birds
Initial WS: swim_0 v_swim:0 INFL_v fly_0 v_fly:0 INFL_v C_rel_that birds
false.

The command `report.` will produce a compact and browser-friendly output, which can be expanded.



Instinctively

Finally, we need to add the adverb *instinctively* to LEX. Adverbs are listed by `advWord/1`.

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

We assume adverbs must seek a verb to modify, in the theory these are vP selecting adverbs. The second parameter is `v+_`, this means it selects for a phrase headed by `v+_`. Verbs are inserted into the Initial WS as a cluster of heads, usually INFL, `v` and the verb root. These heads are clustered together using the `+` notation. Underscore (`_`) here means we permit the adverb to match any vP. (If valued using the key to a cluster (instead of underscore), the selection will keyed too. Keying is clearly incorrect for adverbs.)

We add:

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

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

```
?- [lex].
true.

?- word2heads(instinctively,L,As).
L = [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 instinctively_v fly C_rel_that birds
Initial WS: swim_0 v_swim:0:pres INFL_v instinctively_v fly C_rel_that birds
Initial WS: swim_0 v_swim:0 INFL_v instinctively_v fly C_rel_that birds
Initial WS: swim 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
...
Initial Spellout: birds that 3pl pres fly instinctively 3pl pres swim
Spellout: birds that fly instinctively swim
Parse found!
S0 = {C, {{birds, {C_rel_that, {birds, {INFL_v, {instinctively_v, {birds,
{v_fly:0:pres, fly_0}}}}}}, {INFL_v, {{birds, {C_rel_that, {birds, {INFL_v,
{instinctively_v, {birds, {v_fly:0:pres, fly_0}}}}}}, {v_swim:0:pres, swim_0}}}},
L = [birds, that, fly, instinctively, swim] ;

INT/EXT: {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_fly:0:pres,
fly_0}}}}}}, {INFL_v, {instinctively_v,
{{birds,{C_rel_that:, {birds,{INFL_v:, {birds,{v_fly:0:pres:, fly_0:}}}}}},
{v_swim:0:pres, swim_0}}}}}}
Initial Spellout: birds that 3pl pres fly instinctively 3pl pres swim
Spellout: birds that fly instinctively swim
Parse found!
S0 = {C, {{birds, {C_rel_that, {birds, {INFL_v, {birds, {v_fly:0:pres, fly_0}}}}}},
{INFL_v, {instinctively_v, {{birds, {C_rel_that, {birds, {INFL_v, {birds,
{v_fly:0:pres, fly_0}}}}}}, {v_swim:0:pres, swim_0}}}}}},
L = [birds, that, fly, instinctively, swim] ;

Initial Spellout: birds that 3pl pres fly 3pl pres swim instinctively
...

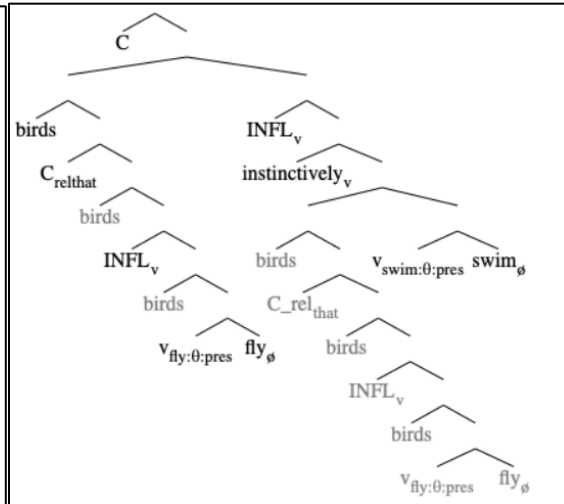
```

We obtain the following two parses:

```

{C, {{birds, {C_relthat, {birds, {INFL_v, {instinctively_v, {birds, {v_fly:0:pres, fly_0}}}}}}}, {INFL_v,
{{birds,{C_relthat:, {birds,{INFL_v:, {instinctively_v:, {birds,{v_fly:0:pres:, fly_0:}}}}}}}}, {v_swim:0:pres,
swim_0}}}}}}
{C, {{birds, {C_relthat, {birds, {INFL_v, {birds, {v_fly:0:pres, fly_0}}}}}}, {INFL_v, {instinctively_v,
{{birds,{C_relthat:, {birds,{INFL_v:, {birds,{v_fly:0:pres:, fly_0:}}}}}}}}, {v_swim:0:pres, swim_0}}}}}}

```



Reference Manual: LEX

Nouns

θ -relevant

Nouns are θ -relevant, and selected by heads with complement or specifier (2nd Merge) feature θ . Define $\theta R/1$. 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 rule implemented that states if a noun ends in *-y*, plural form ends in *-ies*.

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

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

ϕ -features

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

Pronouns ϕ -feature values are specified using $per/2$. 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 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 whR/1. Sample entries:

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

Other features on nouns

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

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* analyzed as the pair $\text{INFL}_{v:3\text{sg}} - \text{V}_{\text{pred:pres}}$. 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 $\text{INFL}_{v-\text{V}_{\text{like}:\theta:\text{pst}}-\text{like}_{\theta}}$. 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 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. ?- vStem(<i>is</i> ,V,S). V=be, S=s | ?- vStem(<i>liked</i> ,V,S). V=like, S=d |
| ii. ?- vEnding(s,TNS). TNS=pres | ?- vEnding(d,TNS). TNS=pst |
| iii. ?- θ (<i>is</i> ,L). L=[vpred:pres, INFL _{v:3sg}] | ?- θ (<i>liked</i> ,L). L=[like θ , vlike: θ :pst, 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. |
| ?- θ (<i>is</i> ,L). | ?- θ (<i>melted</i> ,L). |
| L = [vpred:pres, INFL _{v:3sg}]; | L = [melt $\emptyset$, vmelt: θ :pst, INFL _v] |
| L = [be θ , vbe: θ :pres, INFL _{v:3sg}] | L = [melt θ , vmelt: θ :pst, INFL _v] |
| L = [vv:prog:pres, INFL _{v:3sg}] | |

(Note: *melt $\emptyset$* 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 (parsing) and inflection (for spellout).
- Merge: requires verbal cluster INFL-v-Root and selectional properties to be defined.

EXT: Stemming

Morphy does general verb stemming duties when parsing, morphyStem(Word,C,Root), 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.
```

### 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.<sup>5</sup> 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 attached to words. For verbs, generally there will be a root head immediately preceded by  $\phi$  and TNS from functional heads INFL and v, respectively. ( $\phi$  on INFL comes from Agree with the surface subject.) The mapping  $\phi$ +TNS+Root to inflected form 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).
irregularVspell(ing,be,being).
irregularVspell(ing,have,having).
irregularVspell(en,have,had).
```

## Merge

$\theta$ -configurations are built by  $\theta$ -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, θ). % 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: θ , $\emptyset$). % Mary dances
vR(eat, v:eat: θ , θ). % John ate a sandwich
vR(melt, v:melt: θ , $\emptyset$). % ice melted
vR(melt, v:melt: θ , θ). % sun melted the ice
vR(must, v:v+:modal:word(must), zero). % must have (no TNS)
vR(saw, v:saw: θ , θ). % John sawed the rope
vR(see, v:see: θ , θ). % Mary saw John
vR(want, v:want: θ , 'INFL'). % Mary wants to ...
vR(will, v:v+:modal:word(will), zero). % will/would have (no TNS)
vR(win, v:win: θ , $\emptyset$). % Mary won
```

---

<sup>5</sup> 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  $\phi$ -features used for verbal inflection. (Finally, `OutputFlag` controls EXT error signaling and debugging, and can be ignored.)

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  $\theta$  (denoting a  $\theta$ -relevant WS item). An INFL head keyed to the categorized  $v$  by selection is also generated.  $\theta$ -aware Merge can then construct the  $\theta$ -configuration in (c) below, beginning with heads (a) inserted after stemming the input *arrived*. IA represents a  $\theta$ -relevant item selected from the rest of the WS. In (d),  $INFL_v$  selects for the  $v$  phrase built in (c).  $INFL_v$  triggers Search for a  $\theta$ -relevant item inside the  $\theta$ -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.  $INFL_v \ v_{arrive:pst} \ arrive_{\theta} \ IA_{\phi} \ C$
- b.  $INFL_v \ v_{arrive:pst} \ \{arrive_{\theta:pst}, IA_{\phi}\} \ C$
- c.  $INFL_v \ \{v_{arrive:pst}, \{arrive_{\theta}, IA_{\phi}\}\} \ C$
- d.  $\{INFL_v, \{v_{arrive:pst}, \{arrive_{\theta}, IA_{\phi}\}\}\} \ C$
- e.  $\{IA_{\phi}, \{INFL_v, \{v_{arrive:pst}, \{arrive_{\theta}, IA_{\phi}\}\}\}\} \ C$
- f.  $\{C, \{IA_{\phi}, \{INFL_v, \{v_{arrive:pst}, \{arrive_{\theta}, IA_{\phi}\}\}\}\}\}$
- g.  $\{C, \{IA_{\phi}, \{INFL_v, \{v_{arrive:pst}, \{arrive_{\theta}, \cancel{IA_{\phi}}\}\}\}\}\} \quad (IA_{\phi}, \cancel{IA_{\phi}})$
- h.  $C \ IA_{\phi} \ INFL_{v:\phi} \ v_{arrive:pst} \ arrive_{\theta}$
- i.  $IA \ \phi \ pst \ arrive$
- j.  $IA \ arrived$

In (g), FormCopy applies, constructing the c-command relation  $(IA_{\phi}, \cancel{IA_{\phi}})$ , the 2<sup>nd</sup> 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  $\phi$  on  $INFL_v$ , spelling out as morpheme  $\phi$  at (i). At step (j), the mapping from the linear sequence  $\phi + pst + arrive$  to *arrived* happens.

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  $v:dance:\theta$ , this means  $v$  selects the root *dance* first, and a second Merge selects  $\theta$  (denoting a  $\theta$ -relevant WS item). In the case of verbal form *dances*,  $\theta$ -aware Merge builds the  $\theta$ -configuration in (c). The rest of the derivation steps follow those above, except for the  $\phi + pres + dance$  mapping. If EA is a plural noun, e.g. *they*,  $3pl + pres + dance = dance$ , and  $3sg + pres + dance = dances$  in the case of a singular noun such as *Mary*.

- a.  $INFL_v \ v_{dance:pres} \ dance_{\emptyset} \ EA_{\phi} \ C$
- b.  $INFL_v \ \{v_{dance:pres}, dance_{\emptyset}\} \ EA_{\phi} \ C$
- c.  $INFL_v \ \{EA_{\phi}, \{v_{dance:pres}, dance_{\emptyset}\}\} \ C$
- d.  $\{INFL_v, \{EA_{\phi}, \{v_{dance:pres}, dance_{\emptyset}\}\}\} \ C$
- e.  $\{EA_{\phi}, \{INFL_v, \{EA_{\phi}, \{v_{dance:pres}, dance_{\emptyset}\}\}\}\} \ C$
- f.  $\{C, \{EA_{\phi}, \{INFL_v, \{EA_{\phi}, \{v_{dance:pres}, dance_{\emptyset}\}\}\}\}\}$
- g.  $\{C, \{EA_{\phi}, \{INFL_v, \{\cancel{EA_{\phi}}, \{v_{dance:pres}, dance_{\emptyset}\}\}\}\}\} \quad (EA_{\phi}, \cancel{EA_{\phi}})$
- h.  $C \ EA_{\phi} \ INFL_{v:\phi} \ v_{dance:pres} \ dance_{\emptyset}$
- i.  $EA_{\phi} \ \phi \ pres \ dance$

j. *They dance / Mary dances*

## 5. Reference Manual

This section describes the information the parser demands of LEX to operate. We also describe the EXT component of the parser. I-Language structures built by Merge are passed to EXT to be linearized. The parser will compare the possible outputs of EXT to the original input sentence, highlighting correct parses and discarding mismatches.

### 5.1 Basic Notation

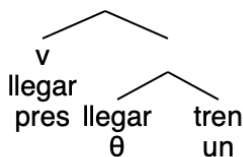
Generally, we may write  $H_F$  to indicate a head  $H$  has feature  $F$ . Multiple features are separated from each other by a colon (:), e.g.  $H_{F1:F2}$  means head  $H$  has first feature  $F1$ , second feature  $F2$ , and so on. This notation with features subscripted is used in Browser output.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

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

In 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  $\theta$ -Merge, the first and 2<sup>nd</sup> features are decoded.  $H_{\theta}$  means head  $H$  takes a  $\theta$ -relevant item as its complement. This results in the structure  $\{H_{\theta}, IA\}$ ,  $IA$  an internal argument. For example,  $llegar_{\theta}$  (*arrive*) above is the head and  $tren$  (*train*) its  $IA$ . Below also, we have an adjectival phrase headed by predicate  $long_{\theta}$  and a (verbal) root  $like_{\theta}$  with  $IA$   $sea_{the}$  containing a feature *the* (representing the definite article). Similarly, the  $\theta$ -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  $\theta$ -relevant item as its argument. For example, (little)  $v$  here selects for complement *like* and  $\theta$ -relevant item  $EA$ , an external argument:

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

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

## 5.2 Initial WS

The initial *Workspace* (WS) is populated by the 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  $\theta$ -configurations; after which, INFL and an appropriate C head will be added to transition computed propositional structures into the clausal domain. 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.<sup>6</sup>

**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  $\phi$ -feature agreement, 2pl or 3pl:

<sup>6</sup> 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  $\theta$ -theory is maintained.

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

The present tense verb form *llegan* (*arrive*) triggers the generation of three heads, i.e.  $R = \text{llegar}_\theta, v = v_{\text{llegar:pres}}$  and  $\text{INFL}_{v:\{2,3\}pl}$ . The suffix  $\theta$  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  $\theta$ -configuration for *llegan dos trenes* and *llega un tren* (*two trains arrive/a train arrives*).

```
{vllegar:pres, {llegarθ, trenun}}
{vllegar:pres, {llegarθ, trenesdos}}
```

Following (Chomsky 2021), in this framework there is no need for a separate T head; instead TNS (*pres* = present) is a feature of  $v$ .<sup>7</sup> Finally, the  $v$  suffix on INFL indicates it selects for  $v$ . Note that word2heads/3 inserts INFL into the WS already bearing  $\phi$ -feature 2pl or 3pl. Merge constructs the following structures:

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

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

word2heads/3 for the indefinite article *un*:

```
?- word2heads(un,Hs,As).
Hs = [], As = [addFType(det,un,type($\theta\text{Rnoun}(_,sg,m)))]$
```

Again,  $Hs = []$  means nothing is added to WS. Action addFType/3 means *un* is a determiner to be added as feature to a  $\theta$ -relevant noun ( $\theta\text{Rnoun}/3$ ) with  $\phi$ -features  $sg-m$  (masculine singular).<sup>8</sup>

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  $\theta\text{Rnoun}/3$  facts for Spanish, article *un* can attach only to masculine singular *tren* (*train*) (not *trenes*).

```
?- $\theta\text{Rnoun}(trenes,pl,m)$.
```

<sup>7</sup> 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  $\theta$ -properties, i.e. one (*meet*) has an EA and the other (*arrive*) does not. They also differ in TNS. (Chomsky 2021) places TNS on  $v$ .

<sup>8</sup> The feminine of the article *un* is *una* and the corresponding action is `addFType(det,una,type( $\theta\text{Rnoun}(\_,sg,f))$` ).

**true.**

?-  $\theta$ Rnoun(*trenes*,pl,f).  
**false.**

?-  $\theta$ Rnoun(*tren*,sg,f).  
**false.**

121 ?-  $\theta$ Rnoun(*tren*,sg,m).  
**true ;**  
**false.**

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

## 5.4 Verbs $\theta/3$ (module $\theta$ a), vstem/4 (module LEX)

In the case of verbs, word2heads/3 (from module parse) relies on  $\theta/3$  (from module  $\theta$ -Theory) to produce the list of  $\theta$ -Merge-relevant *Heads* (drawn from v-appl-R) to be inserted. These heads will combine with arguments under WS Merge to form a  $\theta$ -configuration (examples are given in the previous section).

**$\theta(\text{Word}, \text{Heads}, \text{Actions})$**  *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 and vR/3 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*  $\phi$ -features associated with verb form *Word*, e.g. 3sg (English) or 3sg-f (Spanish).

Examples from English and Spanish LEX:

```
?- 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
```

Note vStem/4 may return multiple values, as in the case of *llegan* (*arrive*) above (due to ambiguity of form with respect to  $\phi$ -feature values).

To expand the hand-built coverage of English LEX, vStem/4 will call upon verb entries from WordNet using morphyStem/3. It also tries to infer the TNS and  $\phi$ -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  $\phi$ -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  $\phi$ -feature differences are merely vestigial; suffix s will map to 3sg with respect to INFL 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 done 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. Some entries:

```
% irregularV(Word,Root,Suffix)
irregularV(broken,break,en).
irregularV(broke,break,d).
irregularV(saw,see,d).
irregularV(seen,see,en).
```

When 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, *morphology* on *ate* returns root *eat*, but does not supply a suffix. Suppose *ate* is not yet listed in irregularV/3, then vStem/3 will return both choices.

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

Once we add the following irregularV/3 entry, the incorrect guess is eliminated.

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

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

## 5.5 Nouns (module LEX)

A brief checklist. For each new noun:

1. Define  $\theta R/1$  and check  $\theta Rnoun/1$ .
2. Check to see  $morphStem/3$  (or  $morph/3$ ) works for stemming the plural form. Define  $num/2$  for plural form.
3. Check its  $\phi$ -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.

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

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

|                      |                                                                                 |
|----------------------|---------------------------------------------------------------------------------|
| $num(Word, pl)$      | True when <i>Word</i> is a plural (pl) noun.                                    |
| $\phi(Word, \Phi i)$ | $\Phi i$ (3sg, 3pl) for English, PERNUM-GEN, e.g. 3sg-m, for Spanish.           |
| $caseR(Word, CASE)$  | CASE 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, 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

?- morph(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 2<sup>nd</sup> 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  $\theta$ -configurations. In LEX, `nR/2` defines whether a noun takes a  $\theta$ -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  $\theta$ -relevant head by  $\theta Rnoun/1$ .

`?- θRnoun(bombing).`  
**true.**

Although it is not directly listed in LEX, the plural form *bombings* is also a  $\theta$ -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).`     $R = \text{bombing}$

Show structures

Possessives from Case &  $\theta$  document.

## 5.6 Determiners

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

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 Workspace (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  $\theta$ Rnoun/3) prior to WS insertion. Some examples:

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

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

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

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

Note the noun may have to satisfy  $\phi$ -feature requirements, e.g. sg-m, via the NUM and GEN parameters to  $\theta$ 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.

## 5.7 Adjectives (module LEX)

Currently, an adjective always triggers  $\theta$ -Merge to form a  $\theta$ -configuration. Optionally, nominalization can be triggered after  $\theta$ -Merge to create a  $\theta$ -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  $\theta$ -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  $\{\textit{narrow}_{\theta}, \textit{corridor}_{\textit{the}}\}$ . This is a  $\theta$ -configuration. Nominalization is optional as the follow examples demonstrate:

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

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

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

`aR(Word, Compl)` *Word* is an adjective, `Compl =  $\theta$`  indicates it selects for a  $\theta$ -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, `morphy/3` (controlled by `useMorphy/0`) is used to find additional adjectives. For example, *adventurous* is not an adjective in the hand-built LEX.

`?- adj(adventurous, C).`

**false.**

`?- useMorphy.`

**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  $\varphi$ -feature components (inefficiently) keyed by the inflected form.

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

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

```
aR(lenta, θ). % slow, f
aR(lento, θ). % 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  $\theta$ -selecting head.

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

## 5.8 Adverbs (module LEX)

Triggers selection (sMerge).

Example: *carefully*<sub>v</sub> seeks a vP to modify in *the mechanic carefully fixed the car*.

SHOW PARSE ETC.

`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 *lenta* (sg-f), then *lentamente* (*slowly*).

## 5.9 Prepositions (module LEX)

Triggers  $\theta$ -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  $\theta$ -relevant object, *the man*, and  $\theta$ -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  $\theta$ -Merge above.

The phrase is  $\{V_{v:\text{pass:pres}}, \{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}\}$ , headed by  $V_{v:\text{pass:pres}}$ .

Head  $V_{v:\text{pass:pres}}$  suppresses  $\theta$ -selection (a requirement peculiar to  $\text{by}_{\theta:v}$ ).

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

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,  $\theta$ -selection by other prepositions will not identified with the main arguments of the  $vP$ .

Example derivation with  $\text{by}_{\theta:v}$ .

**Initial WS 3:**  $\text{man}_{\text{the}}$   $\text{by}_{\theta:v}$   $\text{eat}_{\theta:\text{pastp}}$   $V_{\text{eat}}$   $\text{INFL}_v$   $V_{v:\text{pass:pres}}$   $\text{food}_{\text{the}}$

**WS 1:**  $\{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}$   $\text{man}_{\text{the}}$   $\text{by}_{\theta:v}$   $V_{\text{eat}}$   $\text{INFL}_v$   $V_{v:\text{pass:pres}}$

**WS 2:**  $\{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}$   $\text{man}_{\text{the}}$   $\text{by}_{\theta:v}$   $\text{INFL}_v$   $V_{v:\text{pass:pres}}$

**WS 3:**  $\{V_{v:\text{pass:pres}}, \{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}\}$   $\text{man}_{\text{the}}$   $\text{by}_{\theta:v}$   $\text{INFL}_v$

**WS 4:**  $\{\text{by}_{\theta:v}, \text{man}_{\text{the}}\}$   $\{V_{v:\text{pass:pres}}, \{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}\}$   $\text{INFL}_v$

**WS 5:**  $\{\{V_{v:\text{pass:pres}}, \{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}\}, \{\text{by}_{\theta:v}, \text{man}_{\text{the}}\}\}$   $\text{INFL}_v$

**WS 6:**  $\{\text{food}_{\text{the}}, \{\text{INFL}_v, \{\{V_{v:\text{pass:pres}}, \{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}\}, \{\text{by}_{\theta:v}, \text{man}_{\text{the}}\}\}\}\}$

$\{C, \{\text{food}_{\text{the}}, \{\text{INFL}_v, \{\{V_{v:\text{pass:pres}}, \{V_{\text{eat}}, \{\text{eat}_{\theta:\text{pastp}}, \text{food}_{\text{the}}\}\}\}, \{\text{by}_{\theta:v}, \text{man}_{\text{the}}\}\}\}\}\}$

the food 3sg pres be en eat by the man

**Spellout:** the food is eaten by the man

### Passivization as main verb $\theta$ -role suppression

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

## 5.10 EXT

## 5.11 Re-ordering linear operations

$\Phi$ -feature duplication

%%% AFFIXATION

%% Cases:

%% 1. (Usual):  $\phi + \text{TNS} + v\text{Root} \rightarrow$  inflected verb

%% 2.  $\phi$  + TNS + vRoot + clitic -> clitic + inflected verb  
%% 3. Do-support:  $\phi$  + TNS [+ DO]  $\neg$ vRoot -> do-insertion -> inflected "do"  
%% Note: should be limited to root clauses only

spell\_inflectedV/4

spell\_inflectedV(Phi,TNS,Root,Word)

## INDEX