

## GDE – Handout 3

We are continuing to use GramKit, but now move on to look at ProFIT, an extension to Prolog, written by Gregor Erbach, who now works in Berlin (Erbach 1995). Grammars using ProFIT can be embedded in GramKit. We will be concentrating on an HPSG-like implementation, done by Graham Wilcock, and based on the grammar developed in Sag & Wasow (1999). This differs in a number of details from the ‘standard’ version of HPSG. The grammar covered in this handout has a very small coverage; a subsequent lab session and handout will deal with a broader-coverage grammar.

---

1

---

This handout again assumes that, as recommended, you have previously set up a directory called GDE, to contain various subdirectories. If you didn’t set up such a directory, or used another name, you will need to adapt the instructions in this section as appropriate.

As before, start by creating an appropriate directory:

```
cd GDE
mkdir GramKit2
```

Then copy all the necessary files into it:

```
cd GramKit2
cp /home/rhum/paul/GFL/GramKit2/* .
```

GramKit2 now includes a Readme file, and a sample session in `example`. Loading a ProFIT grammar causes a lot of other files to be automatically created and loaded as well. So GramKit2 will contain a lot more files in due course; see the end of Section 5 below on how to cope with this.

---

2

---

Now load a grammar:

```
sicstus -l load
```

(Remember, that’s an ‘ell’, not a ‘one’.) Get into parse mode and try out the parser:

```
| ?- parse.
|: Kim walked
```

The output should consist of a tree and a semantics (on which more shortly). To see more detailed output, get out of parse mode, and then:

```

| ?- show(signs).

yes
| ?- parse.
|: Kim walks

```

You can change the ‘Default settings’ in `load.pl` to get different kinds of output, but I don’t suggest you try this yet. Some other sentences to parse might be *Kim loves Sandy* and *Kim gave Sandy Pat*.

The structures output are not very easy to read at first, but are not so bad once you get used to them. When `show(signs).` is in operation, part of the representation of *Kim walked* is:

```

syn!head!form!fin&
      mod!none&
val!comps![]&
      spr![]&

```

Here, `!` separates attribute from value, and `&` shows conjoining. Depth of indentation in the structures equates to hierarchical position; e.g. `head` and `val` are both values of `syn`. `[]` shows an empty list. The `comps` and `spr` features together handle subcategorization, and `val` stands for ‘valency’. It is `spr` that contains the subject on its list. (There’s a bit more on this treatment of subcategorization later.) In more conventional feature-structure terms, the above would be shown as:

$$\left[ \begin{array}{c} \text{SYN} \\ \left[ \begin{array}{l} \text{HEAD} \\ \text{VAL} \end{array} \right] \left[ \begin{array}{l} \left[ \begin{array}{cc} \text{FORM} & \text{FIN} \\ \text{MOD} & \text{NONE} \end{array} \right] \\ \left[ \begin{array}{cc} \text{COMPS} & \langle \rangle \\ \text{SPR} & \langle \rangle \end{array} \right] \end{array} \right] \end{array} \right]$$

Sorts (types) are not shown in ProFIT structures, as they are redundant, being predictable from other information. ProFIT does not print uninstantiated features, i.e. those with variables that are not linked to anything.

The semantic representation printed is the value of the `sem!restr` feature of the string. For *Kim walks*, the semantics is:

```

[name!Kim&
  named!B&
  reln!name
  ,
  walker!B&
  reln!walk
]

```

This is an example of Minimal Recursion Semantics. There is a series of attribute–value pairs, but there are no deeply-embedded parts of the structure, rather the elements of the representation are separated by a comma, are implicitly connected by conjoining (with ‘and’), and are related to each other by the use of variables (indices). In this case, the structure denotes a relation (**reln**) of walking, and the argument of this is a **walker**, indexed **B**. This index links up to an entity named **Kim**. For more on this way of representing meaning, see Sag & Wasow (1999, ch. 5) and Copestake et al. (1995).

There is a means of tracing the action of the parser, which can be a help in debugging. To activate this, first get out of parse mode if necessary, then type:

```
| ?- trace(active).
```

However, you may find that this gives you too much information to handle easily! Type `trace(inactive).` to turn tracing off.

---

3

---

Before examining the grammar responsible for the above structures, it is necessary to look at ProFIT, concentrating on what it adds to standard Prolog. Our examples are taken from `firstgram.fit`, which is the grammar you’ve been using. I suggest that you approach ProFIT on a need-to-know basis: only try to understand a particular part of it if you really have to know about it so you can make use of it. There’s not much point in trying to understand the whole system.

Sorts or types are declared by using the following notation:

```
Supersort > [Subsort(1), . . . . , Subsort(n)].
```

For instance:

```
lxm > [const_lxm, infl_lxm].
```

The supersort here is `lxm` (‘lexeme’) and the subsorts are `const_lxm` ‘constant (invariable) lexeme’ and `infl_lxm` ‘inflectable lexeme’. The most general sort is `top`, and all other sorts must be subsorts of `top`.

Feature declarations state which features a particular sort has. Their format is:

```
Sort intro [Feature(1): Restriction(1), . . . . ,  
            Feature(n): Restriction(n)].
```

The restrictions state which sort a value of a feature has, and are optional. Example:

```
noun intro [case, ana].
```

Items of sort `noun` have the features `case` and `ana` ‘anaphor’. Another example:

```
gram_cat intro [head:pos, val:val_cat].
```

Items of sort `gram_cat` (‘grammatical category’) have the features `head` of sort `pos` (‘part of speech’) and `val` (‘valency’) of sort `val_cat` (‘valency category’). It is possible to combine

the declaration of subsorts and features:

```
Supersort > [Subsort(1), ..., Subsort(n)]
  intro [Feature(1): Restriction(1), ...,
        Feature(n): Restriction(n)].
```

An example:

```
pos > [verb, adv, adj, conj, prep, nom]
  intro [form, mod].
```

The subsorts of `pos` (‘part of speech’) are as listed (`verb`, etc.); moreover, `pos` has the features `form` and `mod` (‘modifies’). Sorts inherit all the properties of their supersorts.

A special kind of declaration involves a finite domain, e.g.:

```
agr_cat fin_dom [m,f,n] * [sg,pl] * [1,2,3].
```

The sort `agr_cat` has a finite set of atoms as values, one possibility being `m&sg&3` (3rd-person masculine singular). The `*` here means that the subsorts are not mutually exclusive, as an expression may be (say) both `f` and `pl`. A word or phrase can be described by a subset of items from a finite domain, e.g. `3&sg` for a 3rd-person singular expression of whatever gender. But numerals cannot be used on their own: *you* has to be described as `2@agr_cat`, not just `2`. Disjunction may be used, e.g. the German indefinite article *ein* is `3&sg&(m or n)` (i.e. 3rd-person singular, and masculine or neuter gender). Negation can be used also, as in `~(3&sg)` for non-third-singular forms of the English verb. Disjunction can be used outside finite domains too: e.g. to handle both *John continued walking* and *John continued to walk*, one can state that the complement of *continue* has the feature `form!(prp or to)`. However, processing of this kind of disjunction is less efficient.

Templates are used to give names to frequently-used structures. Their format is:

```
Name := Value.
```

For instance:

```
'NP' := <phrase & syn!(head!<noun & val!(spr![] & comps![])).
```

The notation used here (and in other kinds of statement) needs to be attended to. Terms preceded by `<` are sorts, so `<phrase` denotes something of sort `phrase`. As we saw earlier, the `!` separates features from their values: `comps![]` means a feature `comps`, with an empty list as value. The `&` conjoins terms, and the parentheses show the scope of the conjoining. The above states that `'NP'` (in quotes because of its initial capital – otherwise it would be taken as a variable) is an abbreviation for an item of sort `phrase` where the `head` feature has a value of sort `noun` and the `spr` and `comps` features have empty lists as their values. To call a template, place `@` in front of its name, e.g.:

```
tv_lxm := <tv_lxm & @verb_lxm &
  arg_st![_Subj, @'NP'|_].
```

This is itself a template for transitive-verb lexemes. Among other things, it states that the value of `arg_st` is a list whose second member is 'NP', i.e. what the NP template has as its value. We can also see here more examples of the Prolog-like nature of ProFIT: `_Subj` is a place-holder variable, used not to pass values around but just to say 'this is the subject'; and `_` is an anonymous variable, standing for whatever the rest of the value of `arg_st` is.

The grammar is contained in the file `firstgram.fit`. Two important points to be made about this grammar are: (i) it is drastically simplified, especially as far as the lexicon is concerned, in that the only words it handles are proper names and verbs (but the sort hierarchy, rules and principles are not simplified); (ii) you do not need to understand the ins and outs of everything in the file, certainly not at this stage, and it is best to try to get to grips with the parts that you do need to follow (cf. my remarks above about approaching ProFIT on a need-to-know basis – the same applies here). Some examples from the grammar were given in the previous section.

The grammar consists of a number of parts:

- A sort hierarchy with feature declarations; the format of these has been discussed above.
- Abbreviation templates (see above on 'NP').
- HPSG principles, encoded as ProFIT templates.
- Phrase-structure rules.
- A lexicon, making use of a variety of different kinds of statement.

As an example of an HPSG principle, consider the following version of the Head Feature Principle:

```
'HFP1' := (syn!head!HF ---> [syn!head!HF|_]).
```

The `syn!head` feature on a mother node is identical to the `syn!head` feature on the first daughter. `HF` is a variable, required to be the same in both instances, so the value is passed between the two instances. For an example of a PS-rule, see the following, which is one version of the Head-Complement Rule:

```
@'Phon1' & @'HFP1' & @'VALP1' & @'SemP1' & @'Restr1' & (
<phrase & syn!val!comps! []
--->
[ <word & syn!val!comps! [] ]).
```

First comes a row of templates which apply to this rule. The LHS of the rule describes an object of sort `phrase` that has an empty `comps` feature. The RHS describes an object of sort `word` which also has an empty `comps` feature. This creates a phrase from a word such as *walks* and passes features of the word up to the phrase, e.g. by means of the HFP.

Recall that in this grammar there are two features to handle subcategorization (known as valency features), `comps` and `spr`. It is `spr` that contains the subject on its list. So rather than having a single `subcat` feature, a transitive verb such as *love* has `spr` and `comps` features, and the value of each of these is a list with a single member ('NP'). To see this in more detail, turn `show(signs)` on, and then just get the system to parse the verbs *walks* and *loves*. Sag & Wasow (1999, pp. 83–86) discuss `spr` and `comps`.

The lexicon consists of a list of words, together with lexeme templates, lexical rules and rules to handle inflectional morphology and spelling alternations. From the list of proper names:

```
lex('Kim', @pn_lxm('Kim')).
```

The template for proper noun lexemes tells us what being a proper noun involves:

```
pn_lxm(Name) := <pn_lxm & @noun_lxm &
  syn!(head!agr!(3&sg) &
    val!(spr![] & comps![])) &
  arg_st![] &
  sem!(index!I & restr![reln!name & name!Name & named!I]).
```

Among other things, proper nouns are saturated and are 3rd-person singular, as well as having all the properties supplied by the template for noun lexemes. The following lexical rule converts a lexical entry for a proper noun into an object of sort `word`, and copies all the features over:

```
lex(Name, <word & syn!Syn & arg_st!Arg_St & sem!Sem)
:-
  lex(Name, @pn_lxm(_) & syn!Syn & arg_st!Arg_St & sem!Sem).
```

Note the use of variables such as `Syn` to pass information around. This, then, is the general structure of the lexicon in this implementation: an item is described by the grammar-writer in an abbreviated way, and the grammar expands this skeletal statement into a full-fledged entry for a lexeme or a word. For more on this way of handling the lexicon, see Sag & Wasow (1999, ch. 8): this discusses lexical types and lexical rules.

I suggest you begin your own work with ProFIT by expanding the grammar through adding some lexical information that is just like what is already there – e.g. a new proper name *Mike* and a new verb *runs*. Recall what I have previously said on altering copies of grammars rather than the original, and on commenting on the alterations you make. Your revised grammar has to have a name that ends with `.fit` (except don't call it `english.fit`, as this is the name of the grammar file you'll be using in the next handout!). And remember to change the name of the grammar to be loaded in the `load.pl` file.

Start by adding *Mike* to the proper noun lexeme list, so you can parse *Mike walks*. Next add *run* to the verb lexeme list (by copying and modifying the entry for *walk*). There is

no need to add *runs* anywhere, as the system contains a morphological analyzer that will relate *runs* to *run*. Also, add **run** to the subsorts of **sit** near the start of the file, and add the following feature declaration:

```
run intro [runner].
```

This says that any situation of sort **run** contains a feature **runner**. Make sure the grammar parses *Mike runs*.

When it does, you can adapt the grammar so that it covers subordinate clauses. To have the grammar parse *Kim thinks Pat walks*, you will need to make a number of changes:

1. Add a sort for verbs that subcategorise for subordinate clauses; it might be a subsort of **verb\_lxm**. Choose a suitable label for the sort.
2. Add **think** as a subsort of **sit**, and say which features (semantic roles) **think** introduces (along the lines of **see** and **love**).
3. Add a template for **s(Index)**, along the lines of the existing one for **np(Case, Index)**, but adapting the **head** feature so it describes S not NP.
4. Add a template for your new verb sort; model it on (say) the **tv\_lxm** template, but make sure the second item on the **arg\_st** list is **@s(\_)**. This will call your new **s(Index)** template, and ensure that the second argument of *think* is sentential.
5. Enter *think* in the lexicon (i.e. the verb lexeme list). To do this, copy the entry for *love* and alter it as appropriate. In particular, this means calling the template for your new verb sort, changing the **arg\_st** list as you just did above, and entering the relevant semantic roles (the ones you said that **think** introduces – see above).

Try to parse *Kim thinks Pat walks* and look at the semantics. Then try ill-formed strings such as *Kim thinks walks* and *Kim thinks Pat*.

Loading a ProFIT grammar causes a lot of other files (with suffixes such as **.boo** and **.sig**) to be automatically created and loaded too. There's no real point in keeping these automatically-created files at the end of a session; to get rid of them, use the command:

```
tidy
```

This deletes all these unneeded files. If you ever want to delete just the ones related to a particular grammar, type:

```
tidy X
```

(where X is replaced by whatever your grammar is called, but without the **.fit** suffix: so if the grammar is **gram.fit**, type **tidy gram**). Whatever you do, don't accidentally delete a **.fit** file!

## References

- Copestake, A. et al. (1995), Translation using Minimal Recursion Semantics, *in Proceedings of TMI-95*, Leuven, pp. 15–32.
- Erbach, G. (1995), ProFIT: Prolog with features, inheritance and templates, *in Proceedings of the 7th European ACL*, Dublin, pp. 180–187.
- Sag, I. & Wasow, T. (1999), *Syntactic Theory: a Formal Introduction*, CSLI.