ISSN 2043-0167

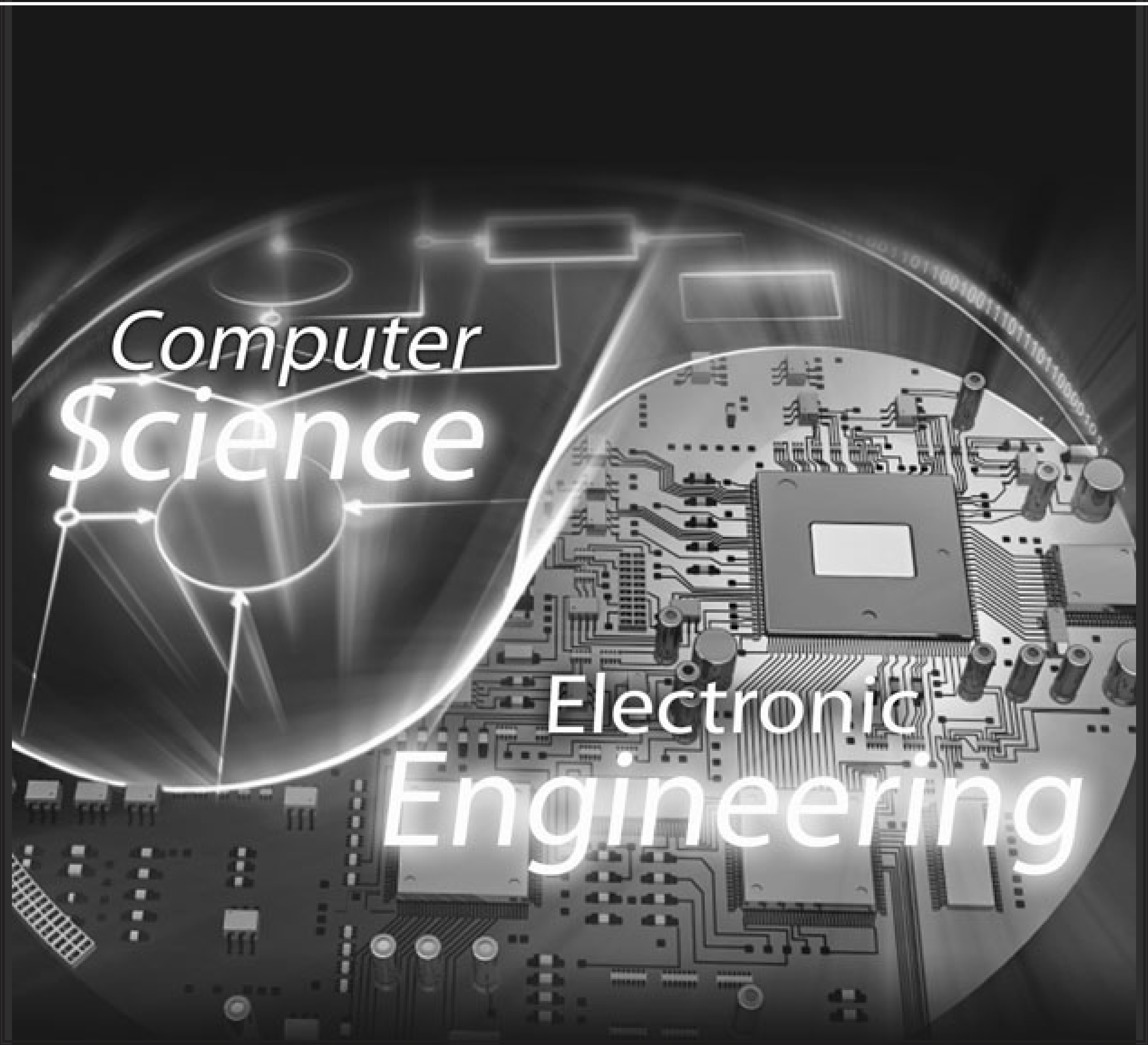
DyLan: Parser for Dynamic Syntax

Arash Eshghi, Matthew Purver and Julian Hough

Queen Mary
University of London

EECSRR-11-05 October 2011

School of Electronic Engineering
and Computer Science



DyLan[*]: Parser for Dynamic Syntax

Arash Eshghi, Matthew Purver and Julian Hough
School of Electronic Engineering and Computer Science
Queen Mary University of London
Mile End Road, London, E1 4NS

June 30, 2011

Contents

[*]DyLan is short for **Dy**namics of **Lan**guage. Available from `http://dylan.sourceforge.net/`.

1 How to read this document

This document describes some of the details of the prototype implementation of the Dynamic Syntax (DS) grammar formalism, *DyLan*. As such, it should be read in conjunction with the documentation (Javadoc) that comes with the implementation, and various papers/books that describe the Dynamic Syntax framework itself (Kempson et al., 2001; Cann et al., 2005; Purver et al., 2006). This document is intended to be a bridge between DS theory and implementation; so here, we give only a short introduction to DS describing the basic concepts, data-structures and procedures involved, with blue hyper-links in square brackets (e.g. [2]) to other parts of this document which specify where they are in the implementation and how they have been implemented. But even there, for the most part, we only give an overview. Detail is provided only when the code and the Java Documentation are judged as needing further elaboration. In the introduction to DS, there will also be references to various parts of the DS literature which elaborate on the concept under discussion. For those who are unfamiliar with DS, we suggest that you read at least this introduction as otherwise the implementation section (3, the second half of the document) will not make sense at all. And of course, if you are already familiar with DS, you can skip the introduction altogether.

2 Introduction

The DyLan parser has been implemented in Java, following the formal details of Dynamic Syntax as per Kempson et al. (2001); Cann et al. (2005). This includes a parser [3] and generator [3.3] for English, which take as input a set of computational actions [7], a lexicon [8.1] and a set of lexical actions [7]; these are specified separately in text files in the IF..THEN..ELSE [7.2] procedural meta-language of DS, allowing any pre-written grammar to be loaded. Widening or changing its coverage, i.e. extending the system with new analyses of various linguistic phenomena, thus does not involve modification or extension of the Java program itself, but only the lexicon and action specifications. The current coverage includes a small lexicon, but a broad range of structures: complementation, relative clauses, adjuncts, tense, pronominal and ellipsis construal, all in interaction with quantification.

More recently, DyLan has been integrated into the Jindigo dialogue system (Schlangen and Skantze, 2009) to provide its *Interpreter* module. See section 2.3, for more theoretical detail on how this has been achieved.

2.1 Dynamic Syntax

Dynamic Syntax (Kempson et al., 2001; Cann et al., 2005) is a parsing-directed grammar formalism, which models the incremental, online processing of linguistic input. Unlike many other formalisms, DS models the incremental building up of *interpretations* without presupposing or indeed recognising an independent level of syntactic processing. Thus, the output for any given string of words, parsed incrementally, token by token, is a semantic tree [4] representing its predicate-argument structure, as in Figure 1.

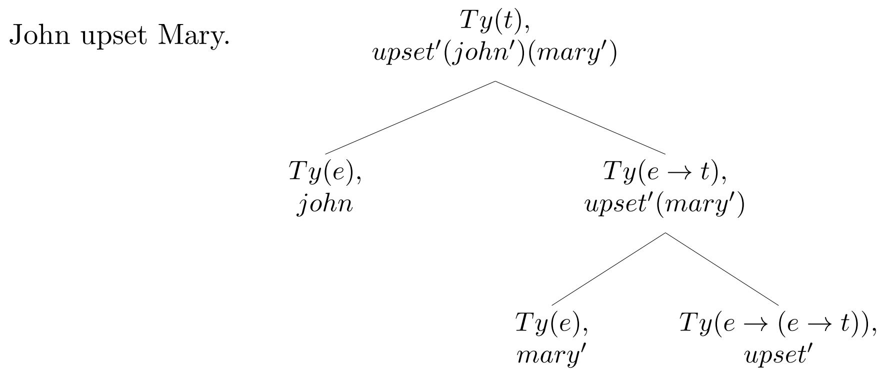

John upset Mary.
$Ty(t)$, $upset'(john')(mary')$
$Ty(e)$, $john$
$Ty(e \to t)$, $upset'(mary')$
$Ty(e)$, $mary'$
$Ty(e \to (e \to t))$, $upset'$

Figure 1: A simple DS tree for *"John upset Mary"*

Grammaticality is defined as parsability, that is, the successful incremental construction of such tree-structure logical forms, using all the information given by the words so far in the sequence, in a left to right, time linear manner.

The logical formulae [6] are lambda terms of the epsilon calculus (Meyer-Viol, 1995), and more recently Record Types of Type Theory with Records (TTR, see Cooper (2005), but also see below section 2.2 for how TTR has been integrated with DS).

2.1.1 Parsing

The central tree-growth process of the model is defined in terms of conditional *actions* [7] whereby such structures are built up. These take the form both of general structure-building principles (*computational actions*), independent of any particular natural language, and of specific actions induced by parsing particular lexical items (*lexical actions*). The core of the formal language is the modal tree logic LOFT (Blackburn and Meyer-Viol, 1994), which defines modal operators $\langle\downarrow\rangle$ and $\langle\uparrow\rangle$, which are interpreted as indicating daughter and mother relations, respectively, with two subcases $\langle\downarrow_0\rangle$, and $\langle\downarrow_1\rangle$ distinguishing daughters decorated with argument or functor formulae, and two additional operators $\langle L\rangle$ and $\langle L^{-1}\rangle$ to license paired linked trees (these are used for the interpretation of relative clauses and adjuncts among other things, see (Cann et al., 2005)). These operators can be grouped together to specify different modalities [4.2], i.e. paths to other locations/nodes on the tree, relative to the current node. For example $\langle\uparrow_1\downarrow_0\rangle Ty(e)$ says that the argument sister of the current node is of type e. The actions defined using this language are conditional transition functions between semantic trees, monotonically extending the input tree and node decorations.

The node decorations are *labels* [5] of different kinds, carrying different types of information. In other words, each node is a set of labels. Labels can be true or false at any particular node; in most cases the truth of a label at a node is its presence on that node. One notable exception is that of modal labels, which usually check the presence of a label on another node on the tree, e.g. $\langle\uparrow_1\downarrow_0\rangle Ty(e)$ holds if $Ty(e)$ is present on the sister node of the current node.

The concept of requirement (also a kind of label, [**??**]) is central to the parsing process, $?X$ representing the imposition of a goal to establish X, for any label X. Requirements may thus take the form $?Ty(t)$, $?Ty(e \to t)$, $?\langle\downarrow_1\rangle Ty(e \to t)$, $?\exists x Fo(x)$, $?\exists x Tn(x)$, etc. All requirements that are introduced have to be satisfied during the construction process.

For example, the first action in parsing a sentence is a general computational action [**??**], termed INTRODUCTION[1], which develops the standard initial Axiom tree, with only one root node (here, as in all such partial tree-structures, there is a pointer, $\diamond$, indicating the node under development):

$?Ty(t), Tn(0), \diamond$
(i.e. a basic requirement to construct a propositional formula), to
$?Ty(t)$, $Tn(0)$, $?\langle\downarrow_0\rangle Ty(e)$, $?\langle\downarrow_1\rangle Ty(e \to t)$, $\diamond$
thereby inducing the sub-goals of constructing a type e argument (0) node and a type $e \to t$ predicate (1) node, by which a predicate-argument formula can eventually be derived. In the lexicon, words are are associated with lexical actions (see below 8.1 for how the lexicon is structured) in a similar style, each a sequence of tree-update actions in an IF..THEN..ELSE format [7.2], employing the explicitly procedural *atomic actions* [7.1], *make*, *go*, *put*, *merge* and others. A simple lexical action for a proper name *John* is as follows:

	IF	$?Ty(e)$	If there's a requirement for a Type e formula
	THEN	`put`$(Ty(e))$	Label the node as Type e
John		`put`$(Fo(John')$	Label the node with semantic content/formula $John'$
		`put`$(\langle\downarrow\rangle \perp$	Bottom restriction: this is a leaf node.
	ELSE	ABORT	

A subsequent general computational action (THINNING) then removes the now satisfied type requirement. A more complex lexical action for a transitive verb *dislike* takes the following form, first making a new predicate node of type $e \to (e \to t)$, and then an argument node with a

[1]In more recent versions of DS, namely in the treatment of English auxiliaries in Cann (2010), the INTRODUCTION and PREDICTION actions are dispensed with for cross-linguistic generality, and also due to a unified treatment of passives, where the syntactic subject is fixed by the main verb itself to the logical object position.

requirement for type e (to be satisfied by parsing the object):

	IF	$?Ty(e \to t)$	If there's a requirement for a Type $e \to t$ formula
	THEN	$\mathtt{make}(\langle\downarrow_1\rangle); \mathtt{go}(\langle\downarrow_1\rangle)$	Make the functor (predicate) daughter (down 1) and go there
		$\mathtt{put}(Ty(e \to (e \to t)))$	Label the node as Type $e \to (e \to t)$
		$\mathtt{put}(Fo(\lambda x \lambda y.Dislike'(x)(y))$	Label the node with semantic content/formula $\lambda x \lambda y.Dislike'(x)(y)$
dislike		$\mathtt{put}(\langle\downarrow\rangle \perp)$	Bottom restriction: this is a leaf node.
		$\mathtt{go}(\langle\uparrow_1\rangle)$	Go back up to the mother node
		$\mathtt{make}(\langle\downarrow_0\rangle); \mathtt{go}(\langle\downarrow_0\rangle); \mathtt{put}(?Ty(e))$	Make the argument daughter, go there and label it as requiring a type e formula - which the object of the verb will provide at a later point in the parse.
	ELSE	ABORT	

So an IF-THEN-ELSE action is composed of a sequence of labels [5], and two separate sequences of atomic actions [7.1], one comprising the THEN block, and the other the ELSE block. The execution of such an action proceeds by first *checking* [5] the labels in order; if all succeed (i.e. if all are true at the node where the pointer is), the THEN block is executed; otherwise the ELSE block of actions fire.

This format of lexical specification is general: all lexical items are defined as providing such actions, the concept of lexical content being essentially procedural. These obligatory lexical actions, together with optional computational actions (also in the same format), induce a sequence of partial trees in a monotonic growth relation as each word is consumed in turn.

We now turn to an algorithmic description of the parsing process.

2.1.2 The parsing process [3]

Given a sequence of words $(w_1, w_2, ..., w_n)$, the parser starts from the *axiom* tree T_0 (a requirement to construct a complete tree of type t), and applies the corresponding lexical actions $(a_1, a_2, \ldots, a_n)$, optionally interspersing general computational actions (which can apply whenever their preconditions are met). More precisely: we define the *parser state* [3] at step i as a set of partial trees S_i. Beginning with the singleton axiom state $S_0 = \{T_0\}$, for each word w_i:

1. Apply all lexical actions a_i corresponding to w_i to each partial tree in S_{i-1}. For each application that succeeds (i.e. the tree satisfies the action preconditions), add resulting (partial) tree to S_i.
2. *Adjust* [3.1.1] the parse state: For each tree in S_i, apply all possible sequences of computational actions and add the result to S_i.

If at any stage the state S_i is empty, the parse has failed and the string is deemed ungrammatical. If the final state S_n contains a complete tree (all requirements satisfied), the string is grammatical and its root node will provide the full sentence semantics; partial trees provide only partial semantic specifications.[2]

[2]Note that only a subset of possible computational actions can apply to any given tree; together with a set of heuristics on possible application order, and the merging of identical trees produced by different sequences, this helps reduce complexity.

2.1.3 Graph representations

Sato (2010) shows how this procedure can be modelled as a *directed acyclic graph*, rooted at T_0, with individual partial trees as nodes, connected by edges representing single actions. While Sato uses this to model the search process, we exploit it (in a slightly modified form) to represent the linguistic *context* available during the parse – important in DS for ellipsis and pronominal construal (Details are given in (Cann et al., 2007; Gargett et al., 2009), but also see the next section 2.1.4.).

We can also take a coarser-grained view via a graph which we term the *state* graph; here, nodes are states S_i and edges the sets of action sequences connecting them. This subsumes the tree graph, with state nodes containing possibly many tree-graph nodes; and here, nodes have multiple outgoing edges only when multiple word hypotheses are present. This corresponds directly to the input word graph (often called a word *lattice*) available from a speech recognizer, allowing close integration in a dialogue system – see below. We also see this as a suitable structure with which to begin to model phenomena such as hesitation and self-repair: as edges are linear action sequences, intended to correspond to the time-linear psycholinguistic processing steps involved, such phenomena may be analysed as building further edges from suitable departure points earlier in the graph.[3]

2.1.4 Parsing in Context [**3.2**]

So far we have been considering parsing without any notion of context in place. This is, of course, essential if we are to account for context-dependent phenomena in dialogue or monologue, such as anaphora and different kinds of ellipsis, and their resolution. Here, we will not go into the detail of how exactly such constructions are analysed in DS (for which, see Cann et al. (2007)). Nevertheless, in order to later present how contextual parsing has been implemented we introduce the core mechanisms here.

There are generally three basic mechanisms in DS which enable the resolution of anaphora and ellipses from context:

1. *Substitution*:Copying/substitution of a formula from some other prior tree in context, into the tree under construction. This is used to deal with anaphora and strict VP-Ellipsis readings.

2. *Rerunning of Actions*:Rerunning a sequence of actions used before to build some prior tree in context, but this time with the current tree under construction as input. This is used to get sloppy readings for VP-Ellipsis, and to resolve some forms of Bare Argument Ellipsis (see Purver et al. (2006) for more theoretical detail)

3. *Direct Extension* Extending a tree in context directly: used to deal with Split Utterances and adjuncts but also Clarification Ellipsis and Sluicing.

Thus, we need to store both trees at every step of a parse, and the actions that were used to build them up, so that these may subsequently be used to recover the meaning of anaphoric and elliptical expressions. To achieve this, the notion of a parse state described above will be revised in favour of one in which parse states will have tuples in them rather than simply trees. We define a *parser tuple* [3] as a triple, $\langle T, W, A\rangle$, where T is a (possibly partial) semantic tree, W is the string of words so far parsed and A the sequence of actions (computational and lexical) used to construct T from W. The initial parse state S_0 contains only a single triple, in which T is the initial Axiom

[3] There are similarities to chart parsing here: the tree graph edges spanning a state graph edge could be seen as corresponding to chart edges spanning a substring, with the tree nodes in the state S_i as the agenda. However, the lack of a notion of syntactic constituency means no direct equivalent for the active/passive edge distinction; a detailed comparison is still to be carried out.

tree and W and A are both empty: $S_0 = \{\langle Axiom, \emptyset, \emptyset\rangle\}$. Accordingly we revise the parse process described above:

Given a sequence of words $(w_1, w_2, ..., w_n)$, the parser starts from the empty state $S_0 = \{\langle Axiom, \emptyset, \emptyset\rangle\}$, and applies the corresponding lexical actions $(a_1, a_2, \ldots, a_n)$, optionally interspersing general computational actions (which can apply whenever their preconditions are met). More precisely: we define the *parser state* [3] at step i as a set of parser tuples of the form $\langle T_i, W_i, A_i\rangle$. Beginning with $S_0 = \{\langle Axiom, \emptyset, \emptyset\rangle\}$, for each word w_i:

1. Apply all lexical actions a_i corresponding to w_i to each parser tuple $\langle T, W, A\rangle$ in S_{i-1}. If the application is successful (i.e. the input tree T satisfies the action preconditions), we will get an output tree T_o. Construct the new parser tuple $\langle T_o, W + w_i, A + a_i\rangle$ and add it to S_i.
2. *Adjust* [3.1.1] the parse state: For each tree in S_i, apply all possible sequences of computational actions in the same manner and add the result to S_i.

If at any stage the state S_i is empty, the parse has failed and the string is deemed ungrammatical.

Context can now be defined in these terms. At any point in the parsing process, the context C for a particular partial tree T in the set S_i can be taken to consist of:

1. a set of triples $P' = \{..., \langle T_i, W_i, A_i\rangle, ...\}$ resulting from the previous utterance(s); and

2. the triple $\langle T, W, A\rangle$ itself.

Discourse-initially, the set P' will be empty, and the context will therefore be identical to the standard initial parser state, the singleton set P_0 containing only a single triple $\langle T_0, \emptyset, \emptyset\rangle$ (where T_0 is the basic Axiom $= \{?Ty(t), \diamond\}$, and the word and action sequences are empty). As words are consumed from the input string and the corresponding actions produce multiple possible partial trees, together with their corresponding word and action sequences, the parser state set will expand to contain multiple triples; note that the context C available to any tree will still be restricted to its current triple (as P' is empty at this point). Once parsing is complete, we use the final set P_1 to define the new starting state (and context) for the next sentence as $P_1 \cup P_0$ (i.e. P_1 with the addition of the triple containing the basic axiom). This is slightly different in the implementation for which see [3.2].

Note that here we have not covered how this context is in fact used to resolve ellipsis (for which see Purver et al. (2006)). Suffice to say that this is achieved by two special computational actions that use the context for tree update. These are SUBSTITUTION and REGENERATION. The former uses context to copy semantic content/formulae from, into the tree under construction and the latter develops the current tree using sequences of action stored in context. These are discussed in a bit more detail in the implementation section below [5.2].

2.1.5 Generation

With the base formalism set out in a parsing perspective, we can define a generation system reflecting production that applies the very same parsing mechanism, as we shall see, leading to tight coordination between parsing and production. Our point of departure is Otsuka and Purver (2003); Purver and Otsuka (2003), which gives an initial method of context-independent tactical generation in which an output string is produced according to an input semantic tree, the goal tree. The generator incrementally produces a set of corresponding output strings and their associated partial trees (again, on a left-to-right, word-by-word basis) by following standard parsing routines and using the goal tree as a subsumption check. At each stage, partial strings and trees are tentatively extended using some word/action pair from the lexicon; only those candidates which

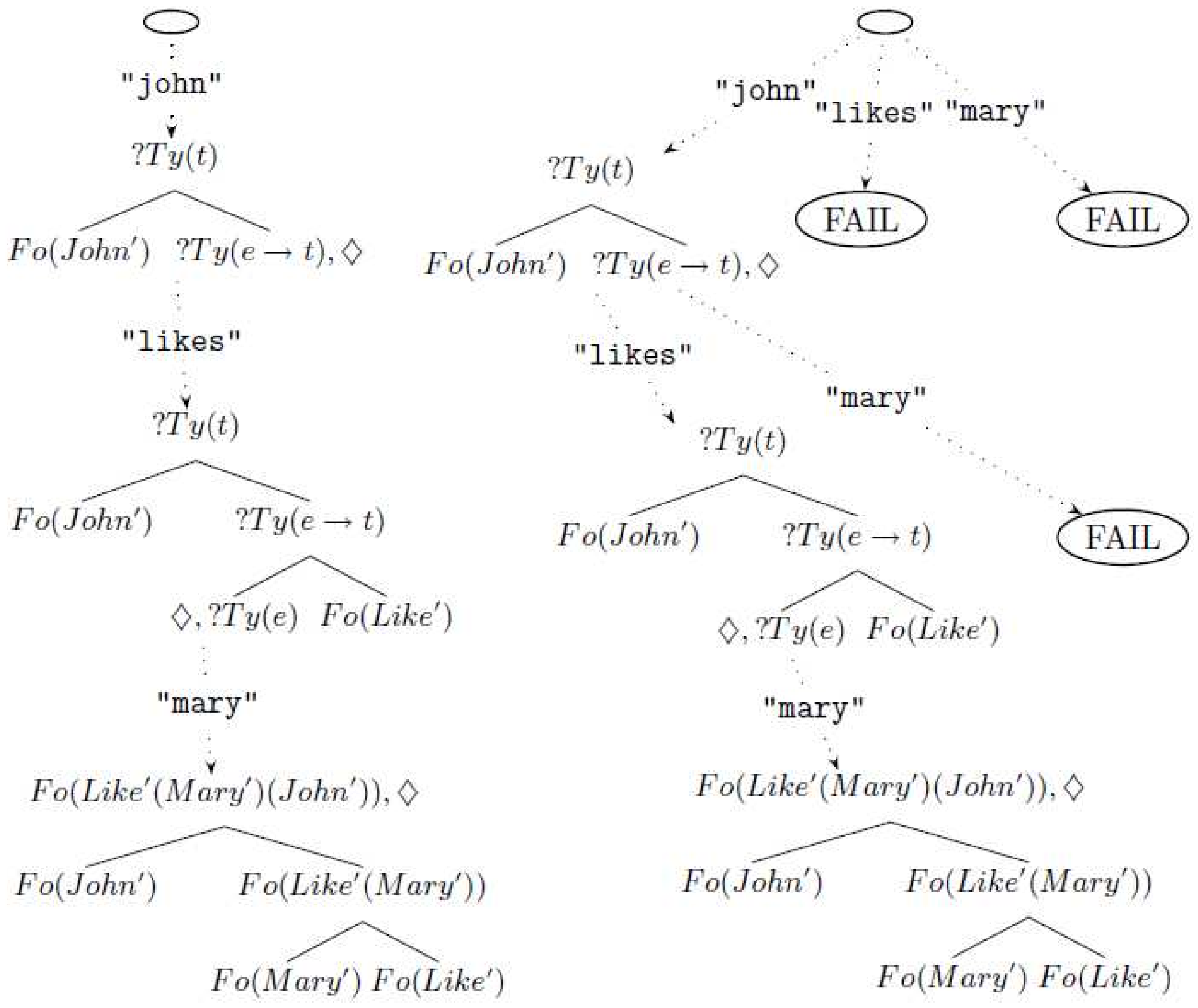


Figure 2: Parsing (left) and Generating (right) of John likes Mary

subsume the goal tree are kept, and the process succeeds when a complete tree identical to the goal tree is produced (see Figure 2). Generation and parsing thus use the same tree representations and tree-building actions throughout.

We can proceed to the definition of a generator state. A generator state G is a pair (T_g, X) of a goal tree T_g and a set X of pairs (S, P), where S is a candidate partial string and P is the associated parser state (a set of $\langle T, W, A\rangle$ triples). Discourse-initially, the set X will contain only one pair, of an empty candidate string and the standard initial parser state, $(\emptyset, P_0)$. As generation progresses, multiple pairs are produced as candidiate partial strings S are considered, each with their own associated parser state P. In generation, the context $\mathcal{C}$ for any partial tree T in a state P is defined exactly as for parsing: the set of triples $P' = \{..., \langle T_i, W_i, A_i\rangle, ...\}$; and the current triple $\langle T, W, A\rangle$. Once generation is complete, the state P_1 paired with the chosen string S_1 is taken to form the new context for the next sentence $P_1 \cup P_0$ (just as with parsing), hand-in-hand with the new initial generator state $X_1 = (\emptyset, P_1 \cup P_0)$. Note here the close relationship between the parsing and generation processes. They share the same basic component of their state (a parser state P, a set of tree/word-sequence/action-sequence triples the generator state merely adds to this (partial) candidate strings and a goal tree), and they share the same representation of context. In addition, as both processes are strictly incremental, there is no requirement that their initial states be empty or contain only complete trees they can in theory start from any parser or generator state.

2.2 Integrating Type-Theory with Records (TTR)

More recent work in DS has started to explore the use of TTR to extend the formalism, replacing the atomic semantic type and FOL formula node labels with more complex *record types* [6], and thus providing a more structured semantic representation. This also allows tighter and more straightforward integration into incremental dialogue systems that work with concept frames, as the mapping between a concept frame and a TTR record type is straightforward (DyLan has been integrated into the Jindigo dialgue system (Schlangen and Skantze, 2009), for which see below section 2.3).

Purver et al. (2010) provide a sketch of one way to integrate TTR in DS and explain how it can be used to incorporate pragmatic information such as participant reference and illocutionary force. As shown in Figure 3(b) above, we use a slightly different variant here: node record types are sequences of typed labels (e.g. $[x : e]$ for a label x of type e), with semantic content expressed by use of *manifest* types (e.g. $[x_{=john} : e]$ where $john$ is a singleton subtype of e).

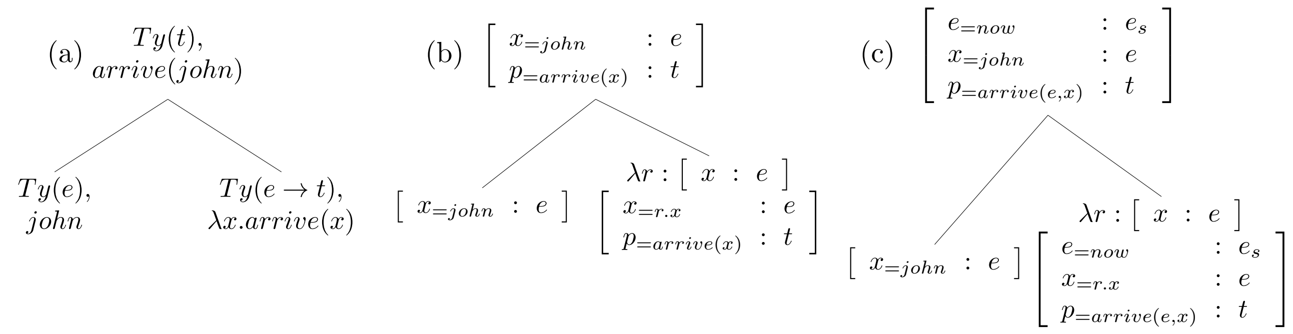

(a)
$Ty(t),$ $arrive(john)$
$Ty(e),$ $john$
$Ty(e \rightarrow t),$ $\lambda x.arrive(x)$
(b)
$\begin{bmatrix} x_{=john} & : & e \\ p_{=arrive(x)} & : & t \end{bmatrix}$
$\begin{bmatrix} x_{=john} & : & e \end{bmatrix}$
$\lambda r : \begin{bmatrix} x & : & e \end{bmatrix} \begin{bmatrix} x_{=r.x} & : & e \\ p_{=arrive(x)} & : & t \end{bmatrix}$
(c)
$\begin{bmatrix} e_{=now} & : & e_s \\ x_{=john} & : & e \\ p_{=arrive(e,x)} & : & t \end{bmatrix}$
$\begin{bmatrix} x_{=john} & : & e \end{bmatrix}$
$\lambda r : \begin{bmatrix} x & : & e \end{bmatrix} \begin{bmatrix} e_{=now} & : & e_s \\ x_{=r.x} & : & e \\ p_{=arrive(e,x)} & : & t \end{bmatrix}$

Figure 3: A simple DS tree for *"john arrives"*: (a) original DS, (b) DS+TTR, (c) event-based

We further adopt an event-based semantics along Davidsonian lines (Davidson, 1980). As shown in Figure 3(c), we include an event term (of type e_s) in the representation: this allows tense and aspect to be expressed (although Figure 3(c) shows only a simplified version using the current time *now*). It also permits a straightforward analysis of optional adjuncts as extensions of an existing semantic representation; extensions which predicate over the event term already in the representation. Adding fields to a record type results in a more fully specified record type which is still a subtype of the original:

$$\begin{bmatrix} e_{=now} & : & e_s \\ x_{=john} & : & e \\ p_{=arrive(e,x)} & : & t \end{bmatrix} \mapsto \begin{bmatrix} e_{=now} & : & e_s \\ x_{=john} & : & e \\ p_{=arrive(e,x)} & : & t \\ p'_{=today(e)} & : & t \end{bmatrix}$$

"john arrives" $\mapsto$ *"john arrives today"*

Figure 4: Optional adjuncts as leading to TTR subtypes

2.3 Dialogue System

DyLan has now been integrated into a working dialogue system by implementation as an `Interpreter` module in the Java-based incremental dialogue framework Jindigo (Skantze and Hjalmarsson, 2010). Jindigo follows Schlangen and Skantze (2009)'s abstract architecture specification and is specifically designed to handle units smaller than fully sentential utterances; one of its specific implementations is a travel agent system, and our module integrates semantic interpretation into this.

As set out by Schlangen and Skantze (2009)'s specification, our `Interpreter`'s essential components are a *left buffer* (LB), *processor* and *right buffer* (RB). *Incremental units* (IUs) of various types are posted from the RB of one module to the LB of another; for our module, the LB-IUs are ASR word hypotheses, and after processing, domain-level concept frames are posted as RB-IUs for further processing by a downstream dialogue manager. The input IUs are provided as updates to a word lattice, and new edges are passed to the DyLan parser which produces a state graph as described above in 2.1.2 and 2.1.3: new nodes are new possible parse states, with new edges the sets of DS actions which have created them. These state nodes are then used to create Jindigo domain concept frames by matching against the TTR record types available (see below), and these are posted to the RB as updates to the state graph (*lattice updates* in Jindigo's terminology).

Crucial in Schlangen and Skantze (2009)'s model is the notion of *commitment*: IUs are hypotheses which can be revoked at any time until they are *committed* by the module which produces them. Our module hypothesizes both parse states and associated domain concepts (although only the latter are outputs); these are committed when their originating word hypotheses are committed (by ASR) and a type-complete subtree is available; other strategies are possible and are being investigated.

2.3.1 Mapping TTR record types to domain concepts incrementally

Our `Interpreter` module matches TTR record types to domain concept frames via a simple XML matching specification; TTR fields map to particular concepts in the domain depending on their semantic type (e.g. *go* events map to `Trip` concepts, and the entity of manifest type *paris* maps to the `City[paris]` concept). As the tree and parse state graphs are maintained, incremental sub-sentential extensions can produce TTR subtypes and lead to enrichment of the associated domain concept.

$$
\text{User: I want to go to Paris} \ldots \quad
\left[\begin{array}{lcl}
e_{=\epsilon,e17,PresentState} & : & e_s \\
e1_{=\epsilon,e19,FutureAccomp} & : & e_s \\
x1_{=Paris} & : & e \\
p2_{=to(e1,x1)} & : & t \\
x_{=speaker} & : & e \\
p1_{=go(e1,x)} & : & t \\
p{*}_{=want(e,x,p1)} & : & t
\end{array}\right]
\quad Trip(to: City[paris])
$$

$$
\text{User: } \ldots \text{from London} \quad
\left[\begin{array}{lcl}
e_{=\epsilon,e17,PresentState} & : & e_s \\
e1_{=\epsilon,e19,FutureAccomp} & : & e_s \\
x1_{=Paris} & : & e \\
p2_{=to(e1,x1)} & : & t \\
x2_{=London} & : & e \\
p3_{=from(e1,x2)} & : & t \\
x_{=speaker} & : & e \\
p1_{=go(e1,x)} & : & t \\
p{*}_{=want(e,x,p1)} & : & t
\end{array}\right]
\quad \begin{array}{c} Trip(from: City[london], \\ to: City[paris]) \end{array}
$$

Figure 5: Incremental construction of a TTR record type over two turns

Figure 5 illustrates this process for a user continuation; the initial user utterance is parsed to produce a TTR record type, with a corresponding domain concept – a valid incremental unit to post in the RB. The subsequent user continuation *"from London"* extends the parser state graph, producing a new TTR subtype (in this case via the DS apparatus of an adjoining *linked* tree (Cann et al., 2005)), and a more fully specified concept (with a further argument slot filled) as output.

System behaviour between these two user contributions will depend on the committed status of the input, and perhaps some independent prosody-based judgement of whether a turn is finished (Skantze and Schlangen, 2009). An uncommitted input might be responded to with a backchannel (Yngve, 1970); commitment might lead to the system beginning processing and starting to respond more substantively. However, in either case, the maintenance of the parse state graph allows the user continuation to be treated as extending a parse tree, subtyping the TTR record type, and finally mapping to a fully satisfied domain concept frame that can be committed.

3 Parsing and generation: The `ds` package

This package provides the top level API for DS parsing and generation. It implements the parsing and generation algorithms described above in section 2.1.2, namely, context-free parsing/generation, and parsing/generation in context. Recall that for these, we needed slightly different notions of parser tuple and parser state and this is reflected in the corresponding classes:

- The `SimpleParser` and `SimpleGenerator` classes, which implement context-free parsing/generation, use a `ParserState` object: a set of `ParserTuple` objects.
- The `ContextParser` and `ContextGenerator` classes, which implement parsing/generation in context, use a `ContextParserState` object: a set of `ContextParserTuple` objects.

The `ParserState` and `ContextParserState` classes are self-explanatory given the code and the JavaDoc, so we do not go into them in any more detail here. Suffice it to say that `ContextParserState` subclasses `ParserState`, the only difference between them being in the kind of ParserTuple they have as their members: while `ParserTuples` populate a `ParserState`, `ContextParserTuples` populate a `ContextParserState`.

The parsing and generation algorithms themselves are implemented in the `Parser` and `Generator` classes, which `ContextParser` and `ContextGenerator` then subclass.

3.1 The `Parser` class

This class implements the parsing algorithm described above in section 2.1.2. It has access to a lexicon (a set of words associated with lexical actions, often read from a text file) and a grammar (the set of computational actions, also often read from file). Parsing, i.e. the top-level `parseWords` method, proceeds by taking a list of words and parsing them one by one in order, interspersing all sequences of computational actions. The result is a new `ParserState`, containing a set of `ParserTuples`, representing all possible interpretations of the given string of words so far. The interspersing of computational actions is done in the `adjust` method which we describe summarily below.

This class is abstract, leaving the `execAction` method unimplemented. This method takes a `ParserTuple`, an action (lexical or computational), and a word (empty/null if the previous argument is a computational action) as argument, and returns a new `ParserTuple` which is the result of applying that action to the `ParserTuple` passed as argument. This method is left abstract so that it can be implemented differently in contextual and context-free parsing.

As noted, the subclasses of the Parser class are `ContextParser` and `SimpleParser`. In `ContextParser`, the `execAction` method returns a new `ContextParserTuple`, whereas in `SimpleParser`, a simple new `ParserTuple` is returned.

3.1.1 Adjusting the state

Given a `ParserState` S, the `adjust` method applies all sequences of computational actions to every `ParserTuple` in S. Some of these will fail as the actions' preconditions will often not be met. But every time this is successful the resulting `ParserTuple` is added to S. The method is called after each word is parsed.

The computational complexity of this method is worse than exponential: roughly speaking, for n computational actions, the complexity is $\mathcal{O}(n!)$. However, at the moment, this method keeps track of which action/tree combinations have already been tried and which ones have succeeded, so that they won't be tried again. Moreover, if a resulting tree is already contained in the state, it will not be added. This reduces the complexity a great deal, but state adjustment still remains a computationally expensive process. In the future, we aim to develop a set of parsing heuristics, possibly learned from annotated data, in order to do scored parsing whereby not all actions have equal probability of being applied, i.e. they will be prioritised. This will help reduce the complexity.

3.2 Parsing in Context: `ContextParser`

The `ContextParser` class subclasses Parser implementing the `execAction` method, such that instead of simply `ParserTuple`s, it takes a `ContextParserTuple` as argument, and returns a tuple of the same type. This class uses a `ContextParserState` object to keep track of the parse state, so that the parse state is populated with tuples of type `ContextParserTuple`, rather than just a `ParserTuple`.

Now recall from the introduction (2.1.4) that parsing in context involved transitions, given by both lexical and computation actions, between parse states which contain tuples of the form $\langle T, W, A\rangle$, where T is a partial tree, W is the sequence of words parsed so far, and A the sequence of actions used to get to $\langle T, W, A\rangle$. The `ContextParserTuple` class represents just such a tuple. As noted, objects of this class are members of a `ContextParserState` object, used by the `ContextParser` to store the parse state when parsing in context.

There is a slight difference between how the notion of context was formulated theoretically above, and how it has been implemented. Recall that the context $\mathcal{C}$ for any tuple, $\langle T, W, A\rangle$, during the parse process was defined to consist of:

1. a set of triples $P' = \{..., \langle T_i, W_i, A_i\rangle, ...\}$ resulting from the previous utterance(s); and
2. the triple $\langle T, W, A\rangle$ itself.

Generally when moving on from parsing one string of words to another (including when there's a speaker switch, and the parser needs to turn itself into the corre), the parse state is reset initially (via the `addAxiom` method) to include only the axiom tuple, i.e. $\langle Axiom, \emptyset, \emptyset\rangle$. This new Axiom tuple, together with each new tuple produced from it during the parse of the new string of words is linked, or has a reference to, a 'previous' `ContextParserTuple`, providing its context. This means that each new tuple, during the course of the new parse, will have exactly one old tuple as its context. But P' above is a set containing possibly more than one tuple. This is accommodated by initially having as many copies of the axiom tuple, as there are old tuples in P'. For example, if two trees (tuples) resulted from the previous string of words (corresponding to two different interpretations of that string), then when resetting the state for the new parse, we'd have two copies of the Axiom tuple, each linked to one of the tuples in P'. This also means that any subsequent tuple produced from any one of these Axiom tuples, will be linked to the same tuple in P' and thus have the same context.

3.3 Generation

As noted in the introduction, generation in DS is parasitic on parsing, using exactly the same mechanisms for tree update. This is reflected in the corresponding class *Generator* which relies on a Parser object to test a word hypothesis at each step of the generation process. The Generator class is abstract leaving the choice of a Parser object down to its subclasses, namely, `SimpleGenerator` and `ContextGenerator`. While the former relies on a `SimpleParser` for tree update, the latter relies on a `ContextParser`. Furthermore, this means that context-dependent generation is achieved through context-dependent parsing.

The generation algorithm (as described in section 2.1.5) is implemented in Generator's `generate` method which takes the goal tree to be realised as argument. This proceeds one word at a time (calling the `generateWord` method), each time extending the current `GenerationState`. This continues as long as a word hypothesis from the lexicon can be found whose parsing would result in a tree that subsumes the goal tree. If not, the resulting `GenerationState` will be empty and generation has failed.

The `GeneratorTuple` and `GenerationState` classes correspond directly to what was described in the introduction above (2.1.5), and given the Java Documentation and the code, we do not go into them here.

4 Trees: The `ds.tree` package

This package implements the DS semantic Tree in addition to the language of the Logic of Finite Trees (LOFT (Blackburn and Meyer-Viol, 1994)), which enables the interpretation of modal labels as used in the specification of lexical and computational actions [7].

4.1 The `Tree` Class

Rather than the usual way of defining trees recursively, this implementation uses a map from node addresses [4.2] to nodes as the underlying data structure, for closer correspondence to DS. Each node on the tree is a set of labels.

Each tree has exactly one pointer which is just a node address. This is the address of the node that contains the DS pointer, marking the node currently under development in the parse. Parsing, i.e. the top-level `parseWords` method, proceeds by taking a list of words and parsing them one by one in order, interspersing all sequences of computational actions.

In addition to the usual get and set methods and moving about on the tree, this class provides the tree manipulation operations that correspond to atomic actions [7.1] in DS. These are:

- *make*: this takes a single modal operator [4.2.3] as argument and adds the node pointed to by the operator to the tree. The modal operator has to be one of $\langle\downarrow_0\rangle$ or $\langle\downarrow_1\rangle$, corresponding to the argument and functor daughters of the pointed node respectively.

- *go*: this takes a modality [4.2] as argument and sets the pointer to the node reached when the path specified by that modality is traversed.

- *put*: takes a label [5] as argument and adds it to the pointed node
- *merge*: takes a modality as [4.2] argument and adds the contents of the node specified by that modality to the pointed node, removing the node specified by the modality.

4.1.1 Tree *Subsumption*

As outlined in the introduction, at each step of the generation process (after running an action, whether computational or lexical), the output tree is checked against the goal tree for subsumption. If the produced tree subsumes the goal tree, we produce the word whose associated lexical action we just ran and then carry on extending the tree until it matches the goal tree exactly.

The Tree class provides the `subsumes` method for this, which takes another Tree as argument. This calls the `subsumes` method provided by the Node class, for all the nodes on the tree. The subsumes method in the Node class in turn calls the subsumes method provided by each label for all the labels on the node. A Node subsumes another if for each label on the node, there's some other label on the other node which is subsumed. The Tree subsumption check succeeds if every node subsumes some other node on the other tree.

4.2 Node Addresses and Modalities

4.2.1 Node addresses

Each node on a tree has an address relative to the root node. The root node has the address '0'. As trees are binary in DS, every other node on the tree is reachable from the root node, via a sequence of steps, either going down-left to the argument daughter (0) or down-right to the functor daughter (1). If node A has address X, then the address of the left (argument) daughter of A is given by adding a 0 at the end of X. The right (functor) daughter's address is given by adding a 1.

We also have paired *Linked-Trees* in DS, used to model relative clauses and island constraints among other things (for more on which see almost any DS paper/book, e.g. Cann et al. (2005), chapter 3). A linked tree is parasitic on the main/matrix tree, and is connected to it at a certain node on the main tree. The address of the root of a linked tree attached at the node address X on the main tree, is given by adding a 'L' at the end of X.

In DS we also have the notion of an *unfixed* or sometimes *locally unfixed* node, i.e. a node whose place on the tree (its address) is not yet fixed at a certain point in the parse, but only constrained to be e.g. somewhere below some other node. This is used to model clitic clustering in some languages and left dislocation in English, among other things (we have not covered these in the introduction above, see any DS paper/book for more detail). The address of a node that is unfixed but constrained to be below node A with address X is given by concatenating the Kleene Star (*) at the of X. For locally unfixed nodes concatenate a 'U'.

A node address therefore is a string of 0s, 1s, *s, Us and Ls, relative the root of a main/matrix tree in DS.

4.2.2 The `NodeAddress` class

In addition to storing a node address as a string and, as such, playing the role of the key in the map that underlies our Trees, this class provides a set of methods which enable the interpretation of tree modal operators and modalities in LOFT. A method, `to`, determines, given a tree, whether a node address can be reached from another, by traversing the path specified by a given modality. Another, `go`, returns the address of the node that is reached when a given modality is traversed.

The class also contains methods for the expansion of underspecified modal operators (those with the Kleene Star, see any DS paper/book).

4.2.3 Modal Operators and Modalities

The `BasicModalOperator` class stores a single modal operator in LOFT. This is the combination of a direction, up or down, with a path, one of either 0, 1, L, * or U, e.g. $\langle\downarrow_0\rangle$ is a single modal operator which 'points to' the argument (0) daughter of the current node.

These can then be grouped together, in the `Modality` class, to form a single modality, e.g. $\langle\uparrow_0\downarrow_1\rangle$ is modality composed of exactly two modal operators which together, point to the node that is reached by going up to the mother node and then down to the functor (1) daughter (i.e. to the sister node).

Note that a modality isn't really meaningful on its own. It needs to be combined with some label to form a modal label. For example, $\langle\uparrow_0\downarrow_1\rangle Ty(e)$, a modal label, means that $Ty(e)$ holds, or is present, on the sister node.

These classes also provide regular expressions which are used to read/parse modalities used in action specifications. The regular expressions specify what it is for a sequence of characters to specify a modality, i.e. they are used to parse/construct modalities from strings.

5 Labels: The `ds.label` package

This package implements all DS labels, as used in action specifications. All the classes here subclass the `Label` class.

Recall that the most central characteristic of all labels is that they can be checked for truth or falsity at the pointed node on a tree. The Label class therefore includes the `check` method which takes a tree and a context (a `ParserTuple`, see above) as argument and returns either true or false. This method should always specify what it is for a label to be true. By default, in the Label class, the truth of a label is simply defined, in the `check` method, as its presence on the pointed node. But specific labels often override this method (e.g. a Modal Label needs to check truth of a label on another node, and not the pointed one).

Any particular subclass of `Label` almost always defines its functor or predicate . For example a type label in DS is associated with the 'ty' functor, a formula label with 'fo'. Each label type then also defines a regular expression, often in terms of this functor, which is used in the constructor of the class to parse/construct a label from a string (as read from an action specification). For a table of all labels and associated functors see below 6.

Any particular subclass of `Label` should also implement its own `getMetas` method, which returns the set of all the *meta-variables* contained within it (for an explanation of what meta-variables are and their relation to labels and actions see below 7.2.1).

Subclasses of `Label` must also implement the `instantiate` method. See below 7.2.1 for an explanation of what this method is supposed to do. We can't go into it here as we haven't yet introduced meta-variables.

Most of these labels and their associated classes are self-explanatory given the DS literature and the Java Documentation. So we do not go into them in any detail. Some of these might be less clear, so below we provide an overview of what they are.

5.1 Embedded Labels: the `EmbeddedLabel` and `EmbeddedLabelGroup`

Embedded labels, i.e. subclasses of the `EmbeddedLabel` class, are labels which embed some other label. One example is Requirement which is given by combining its functor, '?', with another label, representing a goal to establish that other label on the node on which it appears. Other embedded labels are: `ExistentialLabel`, `NegatedLabel`.

`EmbeddedLabelGroup` on the other hand, is the super class of all labels that embed more than one other label. One example is `ModalLabel`, which embeds a group of labels, e.g. $\langle\downarrow_0\rangle(Ty(e), Fo(John'))$, which checks for the truth of $Ty(e)$ and $Fo(John')$, on the argument daughter node of the pointed node.

Other subclasses of `EmbeddedLabelGroup` are: `ExistentialLabelConjunction` and `ContextualTreeLabel`.

5.2 Contextual Labels: `ContextualTreeLabel` and `ContextualActionLabel`

Before we can introduce what these labels are we first need to outline the computational actions in which they are used. Recall from the introduction that we postponed discussion of the computational actions that use the context for tree update and enable the resolution of anaphora and ellipses. These were SUBSTITUTION and REGENERATION, which we will now consider summarily:

```
SUBSTITUTION | IF    1: $Ty(X), ?\exists x.Fo(x)$,
             |       2: $\langle T, W, A\rangle \in \mathcal{C}$
             |       3: $\{Ty(X), Fo(Y)\} \in T$
             | THEN  $put(Fo(Y))$
             | ELSE  ABORT
```

The SUBSTITUTION rule[4] says that if the current node is marked as type X (i.e. $Ty(X)$), and it has a requirement for a formula value (i.e. $?\exists x.Fo(x)$), and for some tuple in context $\langle T, W, A\rangle \in \mathcal{C}$, T contains a node of the same type (i.e. X), with some formula decoration Y, then take the formula and decorate the pointed node with it ($put(Fo(Y))$). This rule is used to resolve anaphora and VP-Ellipsis, whereby a formula value is copied from context into the local tree (see Purver et al. (2006) for more detail).

The `ContextualTreeLabel` class provides the means to express lines 2 and 3 of the SUBSTITUTION rule in terms of a single label. As a subclass of `EmbeddedLabelGroup`, `ContextualTreeLabels` embed a group of other labels. In the rule above, the embedded labels are $Ty(X)$ and $Fo(Y)$. Checking/truth of an instance of `ContextualTreeLabel` (the `check` method) is defined as the truth of all the embedded labels, on some node on the context tree, i.e. checking such a label involves searching the trees in context for a node that contains all the embedded labels.

As per the table of label functors 6, `ContexualTreeLabel` has the associated functor, `context_tree`, which is, as noted, used in action specifications to express lines 2 and 3 of the SUBSTITUTION rule. For example, the equivalent of these lines in action specification text files would be: `context_tree(ty(X), fo(Y))`, which is true just in case both ty(X) and fo(Y) can be found on a single node on some tree in context.

[4] This is a simplified version of the original DS SUBSTITUTION rule, in that it does not check to make sure that the formula copied from context is not present locally. For this see (Purver et al., 2006).

We now turn to the second rule/computational action that uses context for tree update:

$$\textsc{Regeneration}\;\left|\;\begin{array}{ll}\textbf{IF} & 1\text{: } Ty(X), ?\exists x.Fo(x),\\ & 2\text{: } \langle T, W, A\rangle \in \mathcal{C},\\ & 3\text{: } \langle a_i, \ldots, a_{i+n}\rangle \sqsubseteq A,\\ & 4\text{: } a_i = \langle \textbf{IF } \phi_1 \text{ , } \textbf{THEN } \phi_2 \text{ , } \textbf{ELSE } \text{ABORT}\rangle,\\ & 5\text{: } ?Ty(X) \in \phi_1,\\ \textbf{THEN} & \texttt{do}(\langle a_i, \ldots, a_{i+n}\rangle)\\ \textbf{ELSE} & \text{ABORT}\end{array}\right.$$

This is the computational action that updates a tree by rerunning a sequence of actions stored in context, on the local tree under construction. In particular, if the pointed node is annotated with some type X (i.e. $Ty(X)$) and there's a requirement for a formula label (i.e. $?\exists x.Fo(x)$), and there's some action sequence in context that is triggered by $?Ty(X)$, then run that action sequence on the local tree (i.e. $\texttt{do}(\langle a_i, \ldots, a_{i+n}\rangle)$).

This action, like all other computational actions is run when the state is *adjusted* (see above), during a parse. It is used to get sloppy readings for VP-Ellipses and to resolve some forms of Bare Argument Ellipsis (for a detailed analysis of how this is achieved see (Purver et al., 2006)).

Analogous to the SUBSTITUTION rule above, in which 2 of the lines were expressed in terms of a single `ContextualTreeLabel`, a `ContextualActionLabel` expresses all lines 2, 3, 4, and 5 of REGENERATION in terms of a single label. The associated functor for `ContextualActionLabel` is **`triggered_by`**, e.g. **`triggered_by(A, ?Ty(X))`** expresses lines 2, 3, 4, 5 in the above rule, where A is a place holder for an action sequence, and ?Ty(X) is what the first action in this action sequence should include as one of its triggers/preconditions. After checking such a label, A will be instantiated to an action sequence, which is run when the parser encounters the atomic action *do*. This always takes just such an action sequence as argument and runs it.

5.3 Quantification and Scope-dependency labels

Some of the labels in the `labels` package are used to establish scope dependencies among the quantifiers (like 'the', 'a', 'every' and 'most') encountered in the course of a parse. We will not go into any detail on how DS handles quantification and how such dependencies are established. For this see Gregoromichelaki (2006); Kempson et al. (2001) for a detailed account. Suffice it to say that the corresponding label classes are: `ScopeStatement`, `ScopeLabel`, `DomLabel`, `DomPlusLabel`, `ScopeSaturationLabel`, and `IndefLabel`. Also see table 6, for what functors each of these have and what their corresponding meanings are in the language of DS.

6 Formulas: the `ds.formula` package

Formulas are passed as argument to a `FormulaLabel`, annotating nodes with semantic content. This package provides classes for the representation and construction of different kinds of formulas. The top level class which all kinds of formula must subclass is `Formula`. This class includes a set of regular expressions for parsing/constructing different kinds of formulas from strings (which are often read from action specification files). See below table 8 for examples of how to write different sorts of formulae in action specifications.

There's a close correspondence between the structure of each sort of semantic formula and the corresponding class and therefore the classes in this package are self-explanatory given the Java documentation and the DS/Epsilon Calculus/TTR literature; we only list these below with a short description of what they are, what operations they support and where in the DS literature they have been discussed:

- The `EpsilonTerm` class: represents epsilon terms, see any DS paper/book, e.g. Cann et al. (2005) section 3.2, on how the content of noun phrases is represented in the epsilon calculus.
- The `CNOrderPair` class: represents CN (common noun) order pairs, for which see any DS paper book on the internal structure of terms in DS, e.g. Cann et al. (2005) section 3.2.
- The `LambdaAbstract` class: represents lambda formulas of the lambda calculus. It provides methods for functional application (beta-reduction), conjunction and variable substitution/renaming. See any DS book/paper.
- The `TTRFormula` class: represents TTR record types. It provides methods for relabelling, intersection, merge of record types in addition to checking whether a record type is a subtype of another. See above section 2.2, or Purver et al. (2009) for a more detailed discussion of how record types can be used for semantic content representations in DS.

7 Actions: the `action` package

As covered in the introduction above, actions in DS drive the parse. They all take a tree as input, extending it or updating it in some way, returning the resulting tree. Accordingly, in the implementation, all actions, i.e. atomic actions and the If-Then-Else action, implement the *Effect* interface, which contains one method: `exec`. This method takes a tree and a context (a `ParserTuple`, possibly empty/null of we're doing context-free parsing, see above) as argument and returns a tree.

Computational and Lexical actions have their own classes, `ComputationalAction` and `LexicalAction`, and these both subclass the *Action* class. The `Action` class simply wraps an If-Then-Else action and gives it a name [5]. In the case of `ComputationalActions` the name will be the DS name of the action, which is read together with the action specification from file, e.g. THINNING, ELIMINATION. In the case of a `LexicalAction` the name will be the word to which the action corresponds.

7.1 Atomic Actions

Some actions, i.e. the classes in the `ds.action.atomic` package, are *atomic* in the sense that they can't be decomposed. As we will see below, each If-Then-Else action is associated with 2 sequences of these, either associated with its THEN or its ELSE clause. Atomic actions are usually basic operations on trees, and the `exec` method in these cases usually calls methods in the `Tree` class, as covered above 4.1. Moreover, atomic actions, very much like methods/functions in any programming language (but in the 'programming language' of DS) usually take arguments of different types. We list all atomic actions below, together with a brief description of what they do (i.e. what the corresponding `exec` method does), what sort of arguments they take and where they're defined in the DS literature. Given this, the corresponding classes should be self-explanatory:

[5]Unlike ever before, in more recent versions of DS a lexical item can be associated with a sequence of more than one If-Then-Else action, run in order. This recent update is due to the integration/reconciliation of the account of auxiliaries in Cann (2010) with existing ellipsis resolution mechanisms, particularly, the re-running of actions from context. For more information, consult the authors of this document as this change has not yet been published

- **Go** takes a modality as argument and moves the pointer to the node that is reached when that modality is traversed. See any DS paper/book.

- **Make** takes a single downward modal operator as argument and makes/creates the corresponding child node. So, `make(\/0)` makes the argument (left) daughter of the pointed node. See any DS paper/book.

- **Put** takes a single label as argument and adds that label to the pointed node. See any DS paper/book.

- **Merge** takes a modality as [4.2] argument and adds the contents (i.e. a set of labels) of the node specified by that modality to the pointed node, removing the node specified by the modality.

- **Do** takes an action sequence as argument and runs it. See above 5.2, but also see Purver et al. (2006) for more detail.

- **BetaReduce** takes no arguments. If the argument and functor daughters of the pointed node both contain formulae, the function expressed by the formula on the functor daughter (a lambda term) is applied to the formula on the argument daughter, and the result put on the pointed node. This is essentially how semantic content accumulates in the root node of the tree, i.e. by a series of such applications. See any DS book/paper for more detail.

- **Freshput** takes a variable type as argument (i.e. one of 'entity', 'event', or 'prop' and decorates the pointed node with a formula that is just a fresh (unused elsewhere on the tree) variable of the specified type. See Cann (2010) for more detail.

- **GoFirst** takes a label as argument and moves the pointer to the first node that contains that label higher up in the tree. See Gregoromichelaki (2006) for more detail.

- **Conjoin** takes a formula as argument and if the pointed node contains a formula label (i.e. is decorated with a formula), decorates the pointed node with the logical conjunction of the two formulas. This action is used to define the computational action, LINK-EVALUATION, for which see any DS book/paper.

For a table of atomic actions, their corresponding functors and examples see the table below 7.

7.2 The If-Then-Else Action: the `IfThenElse` class

It should by now be clear that in DS, grammatical and lexical rules are all formulated in terms of conditional actions in an If-Then-Else format. As outlined in various places above, an If-Then-Else action has the following three components:

1. **The IF block** is an ordered sequence of `Labels`, also called preconditions or triggers of the action, which are `check`ed whenever an action is executed.

2. **The THEN block** An ordered sequence of atomic actions or `Effects`, or composite macros, that are run in order just in case all the labels in the IF block are true at the pointed node.

3. **The ELSE block** An ordered sequence of atomic actions, or composite macros, that are run just in case one of the labels in the IF block is false, i.e. when the preconditions aren't met.

Before we can move on to describe in more detail how the `exec` method in the `IfThenElse` class is implemented we need to say more about *Meta-variables* in DS, and their implementation.

7.2.1 Meta-variables in DS: the `ds.action.meta` package and the `MetaElement` class

As we have seen, labels, or better to say the predicates/functors associated with them, often take arguments. For example the label functor/predicate Ty, takes a lambda calculus functional/combinatorial type as argument, e.g. $Ty(e \rightarrow t)$. However, these arguments are not necessarily constants, or actual values: they can also contain variables. So for instance $Ty(X)$ is a perfectly valid precondition for an action. In the implementation, we call these *Rule Meta-variables*, as distinct from *Formula Meta-Variables*. The latter provide content underspecification and are used to parse both anaphora and auxiliaries (for more detail see any DS book/paper). By convention - and this is essential in action specifications - Meta-variables are always named using capital letters.

The meaning of rule meta-variables is tacitly existential, i.e. a label like $Fo(X)$ is true just in case there is a possible instantiation for X, such that $Fo(X)$ is present on the pointed node. So for instance, checking $Fo(X)$ on a node that has the label $Fo(John')$ returns true, and instantiates X to $John'$. Thereafter (i.e. in the rest of the If-Then-Else action in which the meta-variable occurs), X will be assumed to have this value, i.e. once X has been instantiated in this way, checking $Fo(X)$ is the same as checking $Fo(John')$.

The `action.meta` package provides classes for the different types of meta-variable, based on the different types of label that can take them as argument, or equivalently, based on the type of value that the meta-variable can be assigned. Examples are types, formulas, node addresses, etc. The name of a class in this package always starts with `Meta`, followed by the kind of value that the corresponding meta-variable can take, e.g. `MetaType`, `MetaFormula`, `MetaNodeAddress`, etc. Note that each Meta- is a subclass of the class that implements the value that the Meta- can take. For instance `MetaType` is a subclass of `Type`, i.e. every `MetaType` is a `Type`, just as in $Ty(X)$, the meta-variable X is a type.

Each of these classes embed/wrap a `MetaElement` of the same type, so for instance, a MetaType embeds a `MetaElement<Type>`. A `MetaElement<T>` is basically a pairing between a name (the name of the variable, e.g. X) and a value/object of type/class T.

There is sometimes more than one possible instantiation for a meta-variable when a label is checked. For instance, consider the label: $< X > Ty(e)$, instructing us to find some modality X at which $Ty(e)$ holds: there may be more than one such modality on any single tree. However, in checking a label in an If-Then-Else action at each step, we need to find an instantiation for the meta-variable(s) involved at that step/label, that not only make that label true, but also all the subsequent labels in the IF clause (as that meta-variable might be involved in those subsequent labels). In other words, the goal is not just to find meta-variable instantiations which make the current label true, but a set of instantiations for all the meta-variables involved in the IF block, such that all the labels turn out to be true. There may be no such set, in which case the action fails, or there may be one or more. This means that we have search space, and that we can use backtracking, for a depth-first search of all possible instantiations of the meta-variables. E.g. if we choose a value v for X at label i of the IF block, and X is involved at label $i + k$, in case value v fails to make label $i + k$ true, we have to go back to step i to find another value for X.

To accommodate backtracking, for any single `MetaElement`, the class provides the means to keep track of which values have been tried and succeeded, i.e. every MetaElement keeps track of its own backtracking history.

7.2.2 WARNING: the `equals` method sets meta-variable value

As noted elsewhere, checking labels often involves checking their presence on a node, i.e. we check to see if there's a label on the node in question (often the pointed node), that is *equal* to the label we are checking. If a label involves a meta-variable, e.g. $Ty(X)$ and the pointed node has a $Ty(e)$ on it, then when checking whether $Ty(X) = Ty(e)$, the `equals` method in the `TypeLabel` class would in turn call the `equals` method for the meta-variable X. This will always succeed and instantiate X to e, unless e is already in the X's backtracking history (i.e. it has already been tried)

The point here is: for all the classes in `action.meta`, checking equality, if successful, instantiates the meta-variable, i.e. checking a label involving a meta-variable will *unify* that label, if possible, with the label to which it is compared. In the above example, $Ty(X)$ will be unified with $Ty(e)$, instantiating X to e.

7.2.3 A potential source of confusion: the `instantiate` method

All meta-variable classes and labels must implement the `instantiate` method. It might seem that this method will assign a value to the meta-variable. This isn't so. Instead, the method returns the actual value assigned to the meta-variable if there is one; otherwise nothing is done and the meta-variable itself is returned unassigned. For example, instantiating the meta-variable $X = e$ (a `MetaType`) would result in a `Type` object representing type e.

All labels (subclasses of `Label`) also implement the `instantiate` method. In the case of labels, the method usually instantiates any embedded meta-variables as above and wraps the returned value in a label of the same kind. For instance, the label $Ty(X = e)$, which has the embedded meta-variable X with the value e, would instantiate to $Ty(e)$. The instantiate method is used when a label is put on a node, i.e. we never put a label containing a meta-variable on a node, we always instantiate it first.

7.2.4 The `exec` method in `IfThenElse` class and the `Backtracker`

The `IfThenElse` class is paired with the helper `Backtracker` class which enables backtracking, i.e. each `IfThenElse` action is associated with its own `Backtracker` object. This class keeps a record of all the meta-variables involved in the IfThenElse action, a record (map) of where (at which IF step) each meta-variable is first used in the action (so that we know which step to go back to in order to try other values for the meta-variable in case a previous value for it failed to make some subsequent label true), and the IF step (an index, pointing to a label) that is being currently checked. The latter is initially set to be 0, i.e. the first label in the IF block of labels. Using these resources the Backtracker class provides the `canBacktrack` method, which returns true if we can backtrack to a point where a previously instantiated meta-variable can be given a new value which also succeeds. This then sets the current IF step to the IF step where this happened, uninstantiates all the meta-variables introduced after this step so that they can get new values too.

Given the Backtracker, the `exec` method in the `IfThenElse` class proceeds by checking the labels one by one. If one of these fails, it attempts to backtrack to a previous step, using the `canBacktrack` method of `Backtracker`. It does this until either all the labels are successfully checked (i.e. a set of assignments can be found for all the meta-variables involved in the labels which make them all true), in which case the sequence of atomic actions associated with the THEN block are executed in order; or all backtracking attempts fail (i.e. the `canBacktrack` method returns false), meaning that we have reached a label that cannot be made true, having tried all possible assignments to the meta-variables so far seen, in which case the atomic actions associated

Label Class	**Functor**	**Example**	**Meaning in DS**
TypeLabel	`ty`	`ty(e>t)`	$Ty(e \rightarrow t)$
FormulaLabel	`fo`	`fo(john)`	$Fo(john')$
Requirement	`?`	`?ty(e>(e>t))`	$?Ty(e \rightarrow (e \rightarrow t))$
ExistentialLabelConjunction	`Ex.`	`Ex.fo(x)`	$\exists x.Fo(x)$
AddressLabel	`tn`	`tn(0)`	$Tn(0)$
ContextualTreeLabel	`context_tree`	`context_tree(fo(X), ty(Y))`	$\langle T, W, A\rangle \in \mathcal{C}$ $\{Fo(X), Ty(Y)\} \in T$
ContextualActionLabel	`triggered_by`	`triggered_by(A, ty(e))`	$\langle T, W, A\rangle \in \mathcal{C}$, $\langle a_i, \ldots, a_{i+n}\rangle \sqsubseteq A$, $a_i = \langle$**IF** ϕ_1 **THEN** ϕ_2 **ELSE** ABORT$\rangle$, $Ty(e) \in \phi_1$
BottomLabel	`!`	`!`	$\perp$
FeatureLabel	`+`	`+Q`	$+Q$
NegatedLabel	¬	`¬ty(e)`	$\neg Ty(e)$
ModalLabel	`<...>`	`< /\1\/0 >fo(john)`	$\langle\uparrow_1\downarrow_0\rangle Fo(John')$
SubsumptionLabel	`subsumes`	`subsumes(< /\1\/0 >)`	$subsumes(\langle\uparrow_1\downarrow_0\rangle)$
DOMLabel	`DOM`	`DOM(a)`	$DOM(a)$
DOMPlusLabel	`DOM+`	`DOM+(a)`	$DOM^+(a)$
ScopeLabel	`Sc`	`Sc(a)`	$Sc(a)$
ScopeStatement	`Scope`	`Scope(a<b<c)`	$Scope(a < b < c)$
IndefLabel	`Indef`	`Indef(+)`	$Indef(+)$

Figure 6: Guide to Action Specification: Table of all DS labels and their associated functors

with the ELSE block will be executed in order (this is often just the atomic action `Abort`, which returns a null tree, and the IfThenElse action has failed).

8 Guilde to Action Specification

The (meta-)language of Dynamic Syntax in which all computational and lexical actions are specified (different labels, modalities, atomic actions, logical symbols etc.) involves, like any other formalism, a set of symbols that can either be produced manually with pen and paper or using type-setting software like Latex. Since we need our parser to be able to load these actions from file we have chosen a set of equivalent text-based symbols, or character combinations that correspond to the DS vocabulary. These are symbols/functors associated with different labels, atomic actions and formulas. Below you will find tables which give the mappings between DS symbols (as you would find in any DS paper/book) and their associated text-based equivalents. Table 6 specifies text-based equivalents of all DS labels and their associated arguments, and examples. Table 7, on the other hand, tells you about text-based equivalents of all DS atomic actions and the kinds of argument that they can take, and table 8 gives you examples of how to specify formulas in text files.

The parser in its current state includes different versions of lexical and computational action specifications for the English language. These can be found under the `resources` folder, which

DS action	**Class**	**Argument type**	**Text Example**	**DS Meaning**
go	Go	a modality	`go(< /\1\/0 >)`	`proper(NAME,PERSON,CLASS)`$go(\langle$
make	Make	a downward modal operator	`make(\/1)`	$make(\downarrow_1)$
do	Do	an action sequence	`do(A)`	$do(A)$
merge	Merge	a modality	`merge(< /\1\/* >)`	$merge(\langle\uparrow_1\downarrow_*\rangle)$
put	Put	a label	`put(?ty(e))`	$put(?Ty(e))$
freshput	Freshput	a meta-variable and a variable type	`freshput(X, event)`	$freshput(X, event)$
$beta - reduce$	BetaReduce	none	`beta-reduce`	$beta - reduce$
conjoin	Conjoin	a formula	`conjoin(love(john, mary))`	$conjoin(love(John', Mary'))$

Figure 7: Guide to Action Specification: table of all DS atomic actions, their argument types and examples

Formula Type	**Text-based functor/pattern**	**Example in DS**	**Text-based equivalent**
Lambda Abstract	`^`	$\lambda x.\lambda y.adore(x, y)$	`X^Y^adore(X,Y)`
Epsilon Term	`eps`	$\epsilon, x, man(x)$	`eps,x,man(x)`
Iota Term	`iota`	$\iota, x, man(x)$	`iota,x,man(x)`
Tau Term	`tau`	$\tau, x, man(x)$	`tau,x,man(x)`
CN Ordered Pair	no functor	$x, man(x)$	`x,man(x)`
TTR Record Type	`[label1:type1|label2:type2|...]`	$\begin{bmatrix} x1 & : & john \\ p & : & man(x1) \end{bmatrix}$	`[x1:john|p:man(x1)]`

Figure 8: Guide to Action Specification: table of formulas, associated text-based functors and examples

is itself included right under the main project folder. Each of these corresponds to DS action specifications at various stages of its development. Generally the more recent any of these is, the more linguistic phenomena it covers. Each folder included under `resources` will contain the following text files, described below:

- `computational-actions.txt`: This contains the set of computational actions in the If-Then-Else format.
- `lexical-actions.txt`: This contains the set of lexical actions, also in the If-Then-Else format. Each of these corresponds to a syntactic category (e.g. transitive verb, proper noun, determiner etc.). So each action here could be associated with many of the words in the lexicon.
- `lexicon.txt`: This is the lexicon, one line per word. See below for a more detailed description.
- `lexical-macros.txt`: This contains the set of lexical macros. As noted above, these are named sequences of atomic actions, which can be called by a lexical action.

8.1 Structure of the lexicon and lexical templates

In DS, each word is associated with a lexical action. This is however an inefficient form of representation, as many words fall under the same syntactic category (i.e. their associated lexical actions effectively do the same thing) but differ only e.g. in associated predicates, tense (past, present, infinitive etc.) and person (third person or not), if the syntactic category in question is a verb. Lexical specification in the `lexical-actions.txt` file therefore takes the form of templates. A template takes arguments or place-holders which can be used elsewhere in the body of the action. As an example consider the following lexical action template for proper names in the text-based format as you would find in action specification files:

A Lexical Template

```
proper(NAME,PERSON,CLASS)
IF     ?ty(e)
THEN   put(ty(e))
       put(fo(NAME))
       put(class(CLASS))
       put(person(PERSON))
       put(!)
ELSE   abort
```

Note in the above how the place-holders, NAME, PERSON and CLASS are being used in the action. These templates will be instantiated to fully fledged form, here with NAME, PERSON and CLASS, replaced with actual values read from the lexicon. This is done in the very beginning when the parser is loaded.

As noted, there's one line per word in the lexicon. Each line is a tab separated set of values, with the first value always being a word, the second the syntactic category - corresponding to the name of the lexical template to be used for that word (e.g. 'proper' in the example above). These values are then followed on the same line by the appropriate set of parameters/values for the place-holders in the template. So for instance the following line in the lexicon,

```
john          proper          john          s3          male
```

would instantiate the above lexical template as:

```
IF     ?ty(e)
THEN   put(ty(e))
       put(fo(john))
       put(class(male))
       put(person(s3))
       put(!)
ELSE   abort
```

8.2 Lexical Macros

It is often convenient to group a sequence of atomic actions together to form a single macro. This is both for ease of reading of action specifications, and has psychological/cognitive justification (see Cann (2010)). In lexical action specifications, macros, if defined, can be called/run, just like atomic actions. Macros are loaded together with the lexical and computational actions, but from a separate file, namely, `lexical-macros.txt`, as noted above. Lexical macros can take arguments in a similar fashion. See any of the `lexical-macro.txt` files for examples.

8.3 The `Grammar` and `Lexicon` classes

These are the classes that read all the action templates from file, instantiate, and store them. This is done in the very beginning when the parser is initialised. A `Grammar` object stores all the computational actions. It is itself an ordered set of `ComputationalAction` objects. The `Lexicon` class instantiates all lexical templates from the lexicon file, and stores the actions as a map from words to a set of `LexicalAction` objects (a word may correspond to more than one lexical action, e.g. 'run' which is both a noun and a verb). The classes are not complicated and should be self-explanatory given the Java Documentation so we do not go into them here.

8.4 Some important pointers in action specification

- Put at least one empty line in between every action specification
- An IfThenElse action can be embedded inside another. If you're using this, indentation is crucial. There are examples in the resources already included. Follow those.
- Avoid extra spaces in general, particularly between the arguments of a lexical templates, e.g. avoid `proper(NAME, PERSON, CLASS)`, instead use `proper(NAME,PERSON,CLASS)`.
- Arguments of lexical templates should all be capitalised as above, e.g. `CLASS`, not `Class`
- For rule meta-variables, always use upper case letters starting from X. But this can be modified in the code.
- For formula meta-variables, always use upper case letters starting from U.
- For lambda formulae, always use upper case letters for the bound variables.
- For existentially quantified variables, always use the lower case letter 'x' ONLY.

References

Blackburn, P. and W. Meyer-Viol (1994). Linguistics, logic and finite trees. *Logic Journal of the Interest Group of Pure and Applied Logics 2*(1), 3–29.

Cann, R. (2010). *The Dynamics of Lexical Interfaces*, Chapter Towards an account of the English Auxiliary System: building interpretations incrementally.

Cann, R., T. Kaplan, and R. Kempson (2005). Data at the grammar-pragmatics interface: the case of resumptive pronouns in English. *Lingua 115*(11), 1475–1665. Special Issue: On the Nature of Linguistic Data.

Cann, R., R. Kempson, and L. Marten (2005). *The Dynamics of Language.* Oxford: Elsevier.

Cann, R., R. Kempson, and M. Purver (2007). Context and well-formedness: the dynamics of ellipsis. *Research on Language and Computation 5*(3), 333–358.

Cooper, R. (2005). Records and record types in semantic theory. *Journal of Logic and Computation 15*(2), 99–112.

Davidson, D. (1980). *Essays on Actions and Events.* Oxford, UK: Clarendon Press.

Gargett, A., E. Gregoromichelaki, R. Kempson, M. Purver, and Y. Sato (2009). Grammar resources for modelling dialogue dynamically. *Cognitive Neurodynamics 3*(4), 347–363.

Gregoromichelaki, E. (2006). *Conditionals: A Dynamic Syntax Account.* Ph. D. thesis, King's College London.

Kempson, R., W. Meyer-Viol, and D. Gabbay (2001). *Dynamic Syntax: The Flow of Language Understanding.* Blackwell.

Meyer-Viol, W. (1995). *Instantial Logic.* Ph. D. thesis, University of Utrecht.

Otsuka, M. and M. Purver (2003). Incremental generation by incremental parsing. In *Proceedings of the 6th CLUK Colloquium.*

Purver, M., R. Cann, and R. Kempson (2006). Grammars as parsers: Meeting the dialogue challenge. *Research on Language and Computation 4*(2-3), 289–326.

Purver, M., E. Gregoromichelaki, W. Meyer-Viol, and R. Cann (2010). Splitting the 'I's and crossing the 'You's: Context, speech acts and grammar. In *Aspects of Semantics and Pragmatics of Dialogue. SemDial 2010, 14th Workshop on the Semantics and Pragmatics of Dialogue.*

Purver, M., C. Howes, E. Gregoromichelaki, and P. G. T. Healey (2009). Split utterances in dialogue: a corpus study. In *Proceedings of the 10th Annual SIGDIAL Meeting on Discourse and Dialogue (SIGDIAL 2009 Conference).*

Purver, M. and M. Otsuka (2003). Incremental generation by incremental parsing: Tactical generation in Dynamic Syntax. In *Proceedings of the 9th European Workshop in Natural Language Generation (ENLG).*

Sato, Y. (2010). Local ambiguity, search strategies and parsing in Dynamic Syntax. In E. Gregoromichelaki, R. Kempson, and C. Howes (Eds.), *The Dynamics of Lexical Interfaces.* CSLI. to appear.

Schlangen, D. and G. Skantze (2009). A general, abstract model of incremental dialogue processing. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*.

Skantze, G. and A. Hjalmarsson (2010). Towards incremental speech generation in dialogue systems. In *Proceedings of the SIGDIAL 2010 Conference*.

Skantze, G. and D. Schlangen (2009). Incremental dialogue processing in a micro-domain. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*.

Yngve, V. H. (1970). On getting a word in edgewise. In *Papers from the 6th regional meeting of the Chicago Linguistic Society*.