Intensional Context-Free Grammar

by

Richard Little
M.A., University of Northern BC, 2000
B.Sc., University of Northern BC, 1996
B.Sc., University of Western Ontario, 1994

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Supervisory Committee

Intensional Context-Free Grammar

by

Richard Little
M.A., University of Northern BC, 2000
B.Sc., University of Northern BC, 1996
B.Sc., University of Western Ontario, 1994

Supervisory Committee

Dr. William Wadge, Department of Computer Science
Supervisor

Dr. Alex Thomo, Department of Computer Science
Departmental Member

Dr. Bruce Kapron, Department of Computer Science
Departmental Member

Dr. Hélène Cazes, Department of French
Outside Member

Abstract

Supervisory Committee
Dr. William Wadge, Department of Computer Science
Supervisor
Dr. Alex Thomo, Department of Computer Science
Departmental Member
Dr. Bruce Kapron, Department of Computer Science
Departmental Member
Dr. Hélène Cazes, Department of French
Outside Member

The purpose of this dissertation is to develop a new generative grammar, based on the principles of intensional logic. More specifically, the goal is to create a psychologically real grammar model for use in natural language processing. The new grammar consists of a set of context-free rewrite rules tagged with intensional versions.

Most generative grammars, such as transformational grammar, lexical functional-grammar and head-driven phrase structure grammar, extend traditional context-free grammars with a mechanism for dealing with contextual information, such as subcategorization of words and agreement between different phrasal elements. In these grammars there is not enough separation between the utterances of a language and the context in which they are uttered. Their models of language seem to assume that context is in some way encapsulated in the words of the language instead of the other way around.

In intensional logic, the truth of a statement is considered in the context in which it is uttered, unlike traditional predicate logic in which truth is assigned in a vacuum, regardless of when or where it may have been stated. To date, the application of the principles of intensionality to natural languages has been confined to semantic theory.

We remedy this by applying the ideas of intensional logic to syntactic context, resulting in intensional context-free grammar.

Our grammar takes full advantage of the simplicity and elegance of context-free grammars while accounting for information that is beyond the sentence itself, in a realistic way. Sentence derivation is entirely encapsulated in the context of its utterance. In fact, for any particular context, the entire language of the grammar is encapsulated in that context. This is evidenced by our proof that the language of an intensional grammar is a set of context-free languages, indexed by context.

To further support our claims we design and implement a small fragment of English using the grammar. The English grammar is capable of generating both passive and active sentences that include a subject, verb and up to two optional objects. Furthermore, we have implemented a partial French to English translation system that uses a single language dimension to initiate a translation. This allows us to include multiple languages in one grammar, unlike other systems which must separate the grammars of each language. This result has led this author to believe that we have created a grammar that is a viable candidate for a true Universal Grammar, far exceeding our initial goals.

Table of Contents

List of Figures

Acknowledgments

The creation of this dissertation was a long journey and along the way I received much help, support, encouragement, patience and cooperation from those around me. I would to thank everyone and anyone who falls into this category. The following few I would like to single out for specific contributions.

I would like to begin with my supervisor Dr. Bill Wadge. Bill has spent many hours over the last few years with me, reading, meeting, discussing, rereading, meeting again, and discussing some more and on and on. He has always been thorough and tough but at the end of the day when he gave me the thumbs up I knew it was good.

I would also like to thank the rest of my committee, Dr. Alex Thomo, Dr. Bruce Kapron, and Dr. Hélène Cazes for their time and efforts. I would particularly like to thank Alex who in his role as Graduate Advisor to the department has kept me on track for the last year or so in a number of ways.

On that note, I also have to give a big thanks to Wendy Beggs, Graduate Secretary to the department. I have a feeling that Wendy has spent more administrative hours on me over my time here than on all the other grad students combined. Thanks Wendy and go Leafs go!

My final thanks go to my wonderful family; my wife Tracy Bulman and my children Jora and Oscar. Jora and Oscar, you were both born while I was working on my PhD and although they were some of the toughest years of my life they were also the best years of my life because of you. Tracy, what can I say? It's finally over! You don't know how much I appreciate everything you have done for our family to allow me to get to here. I love you!

1 Introduction

We propose *Intensional Context-Free Grammar* (ICFG), which consists of traditional context-free rewrite rules that are guided by an implicit, multi-dimensional *version space*. This gets us a psychologically realistic, multi-versioned generative grammar which can account for long-distance dependencies within a sentence, simply and elegantly. This is possible by deriving sentences within the context that governs them instead of having that context parceled up and woven into the derivation itself. To illuminate these ideas further we need to first acquaint ourselves with a few ideas. We begin in this chapter with an introduction to formal generative grammars and a discussion of the merits of context-free grammar as a model for natural languages. We follow this with a look at three different grammars as extensions to the context-free model. We then propose our alternative to these grammars before revisiting the claims we make above. We end this chapter with a preview of the rest of the dissertation.

1.1 Formal Generative Grammars

The idea of *grammar* has been around for centuries in one form or another. It was not until the middle of the last century, at the confluence of advances in mathematical logic and the invention of the computer, that the idea of *formal generative grammar* took hold. In 1956, Noam Chomsky published an article called *Three models for the description of language* [1] which he expanded upon in 1957 in the seminal book *Syntactic Structures* [2]. In [2], he articulates the idea that some part of language learning in humans is innate. That is, although each person's native language is learned in early childhood, some part of the structure of language must already exist. He believed that humans were born with

some finite grammar, called the *Universal Grammar*, preprogrammed into their brains that could be used to generate the infinite number of sentences that a given language contains. For Chomsky, this explained the ease with which humans acquire the highly complex ability to speak and comprehend a language.

In *Syntactic Structures*, Chomsky explores some simple grammar types as possible English grammar models before introducing us to his proposed alternative *Transformational Grammar*. The three models that he describes are *simple lists*, *finite state grammars* (to become *regular grammars*) and *phrase structure grammars* (to become *context-free grammars*). This sparks a huge paradigm shift in syntactic theory, producing the subfield of formal generative grammar. Within this field there are differing opinions, which all grew out of the models proposed by Chomsky, with proponents for and against each.

In the formal grammar literature it is universally accepted that natural languages, viewed as sets of strings (or *stringsets*), are infinite and thus any formalization must be finite but capable of generating infinitely many strings. For me this is not so clear cut. How can a language be infinite? No one could ever speak infinitely many sentences. Furthermore, any sentence that becomes too long would be incomprehensible, and for a language to be infinite it would have to include many very long sentences. The human mind can only process so much and comprehend it due to short-term memory limitations. So, are sentences of this type then grammatical?

The motivation for this belief lies in the nature of the proposed grammars themselves. To be able to generate infinitely many sentences, while avoiding simple lists, you need a mechanism that is recursive, which the proposed grammars all provide. But, if

you do not allow for infinitely many strings then you need a stopping point and it is hard to provide one, for the limit of comprehension is hard to measure and likely different for different people. What generative grammarians do is assume an idealistic version of language, one in which grammatical versus ungrammatical is known in all cases and arbitrarily long strings are allowed. Chomsky himself notes this in [3, p. 3] when he writes,

> Linguistic theory is concerned primarily with an ideal speaker-listener, in a completely homogeneous speech-community, who knows its language perfectly and is unaffected by such grammatically irrelevant conditions as memory limitations, distractions, shifts of attention and interest, and errors in applying his knowledge of the language in actual performance.

It seems that these arguments are unnecessary. That is, it is enough to note that due to feature agreement, subcategorization and the large size of a natural language corpus we can justify the use of some formal grammar that is more complex than the three simple models Chomsky discusses. Clearly, there is other evidence to suggest that we do not just memorize sentences in the form of some big list. For instance, how we so easily incorporate new words into our language and construct new sentences with them.

Furthermore, in the case of regular grammars and context-free grammars, it is not a simple matter to account for feature agreement, like subject-verb agreement, without going slightly beyond the capability of the grammar. It is even more difficult to account for agreements that occur over longer distances like those in (1), for example, where there are three pronouns all dependent on the subject noun for their form.

(1) *John went to his school and he did his homework.*

In the next section we look at context-free grammars in more detail to illustrate these last two points.

1.2 Why Not Context-Free Grammars?

Formally, a *context-free grammar* (*CFG*) is a four-tuple, $G = (N, T, S, P)$, where T is a finite set of atomic symbols called *terminals*, N is a finite set of rewrite symbols called *nonterminals*, S is the unique nonterminal start symbol, and P is a finite set of *rewrite rules* (or *production rules*) of the form $A \rightarrow \alpha$, where A is a nonterminal and α is a string of terminals and nonterminals. In terms of a natural language grammar, T is the lexicon of words, N is the set of symbols representing the phrasal categories, S is the symbol representing a sentence, and P is the set of constituent structure rules which represent the recursive grouping of words into well formed, lexical phrases.

For example, consider the context-free grammar given in (2).

(2) $S \rightarrow NP\ VP$
$NP \rightarrow (D)\ A^*\ N\ PP^*$
$VP \rightarrow V\ (NP)\ (PP)$
$PP \rightarrow P\ NP$
$D \rightarrow a|the|some$
$A \rightarrow big|brown|old$
$N \rightarrow birds|fleas|dog|hunter$
$V \rightarrow attacks|ate|watched$
$P \rightarrow for|beside|with$

Here, NP stands for *noun phrase*, VP for *verb phrase*, PP for *preposition phrase*, D for *determiner*, A for *adjective*, N for *noun*, V for *verb* and P for *preposition*. For economy of expression I have taken a few liberties here that are common practice in the literature. The parentheses, like in $VP \rightarrow V\ (NP)\ (PP)$, indicate optional symbols. This allows us to represent multiple rules in one[1]. The asterisk, as in $NP \rightarrow (D)\ A^*\ N\ PP^*$, indicates zero or more instances of the symbol preceding it. Thus, a noun can be preceded by any finite number of adjectives and followed by any finite number of preposition phrases. Finally,

[1] Thus, this rule includes the rules $VP \rightarrow V$, $VP \rightarrow V\ NP$, $VP \rightarrow V\ PP$, and $VP \rightarrow V\ NP\ PP$

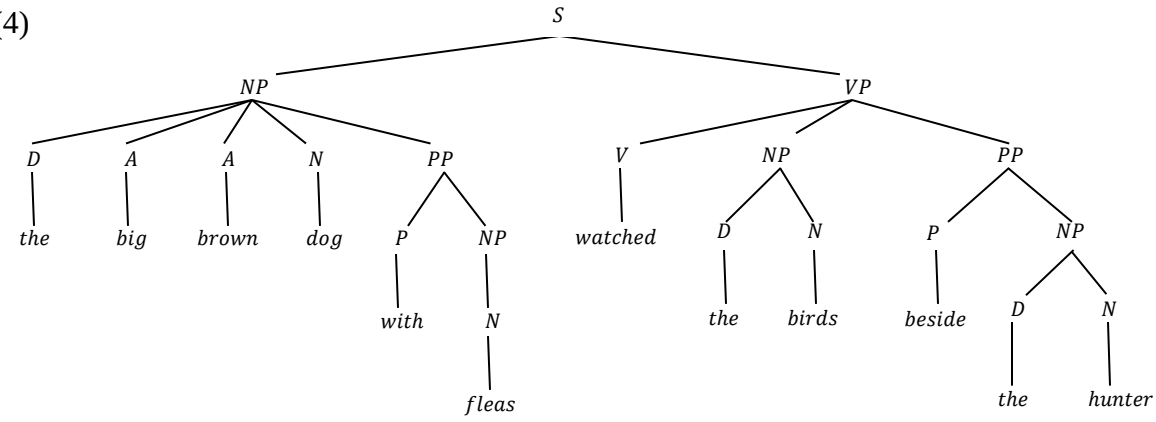
the vertical line is shorthand for 'or'. That is, rule $D \rightarrow a|the|some$, states that an instance of the nonterminal symbol D can be rewritten as the terminal a or the or $some$.

Now, what does a grammar like this do well? In fact, it can generate infinitely many grammatical sentences in a rather economic fashion. For instance, consider the relatively large sentence given in (3).

(3) $The\ big\ brown\ dog\ with\ fleas\ watched\ the\ birds\ beside\ the\ hunter.$

The derivation of this sentence can be summarized through a *tree diagram* (also called a *derivation tree* or *parse tree*). One tree diagram for (3) is given (4).

(4)



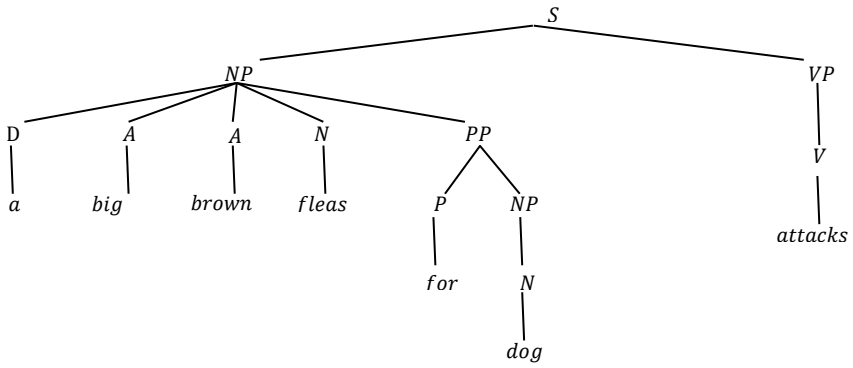
However, as stated above, there has been some dispute over the viability of context-free grammars as natural language models. Chomsky's original intuitions were that CFGs could not deal with word dependencies within a sentence, like feature agreement. He further pointed to dependencies outside of a sentence like those in active-passive sentence pairs as being a problem for CFGs. This sparked a debate that has lasted for decades, which for the most part is just a series of odd examples and counter-examples showing what CFGs can and cannot do. In the long run, all these examples are of families of sentences that are very awkward and are rarely, if ever, used in practice, whether they are grammatical or not.

It seems that Chomsky's original intuitions are in fact partially correct in that even if it can be shown that CFGs can deal with agreements, it is not in an easy or efficient manner. Consider for instance semantics, subcategorization of words and local agreements like that between a noun and its determiner or a verb and its subject noun phrase. Using our example grammar in (2) we can generate sentence (5), which violates English in all these areas.

(5) $*A\ big\ brown\ fleas\ for\ dog\ attacks.$ [2]

Note that, structurally, this sentence is fine as is evident from the associated tree diagram in (6).

(6)



There are a number of things wrong with this particular sentence, let us look at each of them in turn. First, there is a violation of determiner-noun agreement between a and $fleas$. The plural subject noun $fleas$ expects the definite article the, the indefinite $some$ or no determiner at all. Also, there is a violation of the subject-verb agreement in that the verb $attacks$ is dependent on the number of the subject $fleas$, which means it should be $attack$.

[2] The asterisk at the beginning of the sentence indicates that it is ungrammatical in the given language.

Furthermore, $attacks$ is a transitive verb, meaning it expects an object noun phrase to come after it. This is a subcategorization issue, as certain verbs are intransitive, having no object, while others are transitive. Finally, there is a problem with the choice of preposition phrase in the subject noun phrase. Based on the subject noun itself there is an expectation as to the type of preposition and noun in the preposition phrase. Again, this is dependent on the subcategorization of both the noun and the preposition and on their semantics.

You can get around these issues, even within the context-free framework, by subcategorizing your symbols and adding more rules. For example, to deal with subject-verb agreement we could replace the nonterminals NP and VP with the new nonterminals NP_{sg}, NP_{pl}, VP_{sg}, and VP_{pl}, for singular and plural noun phrases and verb phrases. This would result in the new grammar given in (7).

(7) $S \rightarrow NP_{sg}\ VP_{sg}$
$S \rightarrow NP_{pl}\ VP_{pl}$
$NP_{sg} \rightarrow (D)\ A^*\ N_{sg}\ PP^*$
$NP_{pl} \rightarrow (D)\ A^*\ N_{pl}\ PP^*$
$VP_{sg} \rightarrow V_{sg}\ \left(NP_{sg}\right)\ (PP)$
$VP_{sg} \rightarrow V_{sg}\ \left(NP_{pl}\right)\ (PP)$
$VP_{pl} \rightarrow V_{pl}\ \left(NP_{sg}\right)\ (PP)$
$VP_{pl} \rightarrow V_{pl}\ \left(NP_{pl}\right)\ (PP)$
$PP \rightarrow P\ NP_{sg}$
$PP \rightarrow P\ NP_{pl}$
$D \rightarrow a \mid the \mid some$
$A \rightarrow big \mid brown \mid old$
$N_{sg} \rightarrow dog \mid hunter$
$N_{pl} \rightarrow birds \mid fleas$
$V_{sg} \rightarrow attacks \mid ate \mid watched$
$V_{pl} \rightarrow ate \mid watched$
$P \rightarrow for \mid beside \mid with$

Next, we would need to introduce new subcategories for D for determiner-noun agreement, and P for the proper placement of prepositions in the noun phrase, and we would need to further subcategorize the verb categories to account for transitive and intransitive verbs. As you can see this leads to a huge proliferation of rules and your grammar can quickly become unwieldy and this is just the tip of the iceberg. Similar things would need to be done to account for noun subtypes and noun case. Furthermore, we would still need to find similar ways to deal with long-distance dependencies like those in sentence (1) above, and on and on.

Aside from the exponential growth of the grammar there is another problem with the context-free strategy. By subdividing the rules in this way, we are introducing a lot of redundancy. By repeating rules that do the same thing we lose the ability to generalize across categories. Although the lexical categories can be subcategorized based on differences inherent in word subgroups, there are some properties of words that cut across whole categories. So, optimally we would like to account for both category generalities and subcategory differences and this is what cannot be done by a context-free grammar alone. At the same time, context-free grammars are useful, simple and efficient thus leading many to explore ways of augmenting CFGs in ways that take advantage of these properties, while accounting for its deficiencies. In the next section we look at three such grammars.

1.3 Extensions of Context-Free Grammars

Many formal grammars have been developed to account for the complexity of natural languages by extending CFGs in some way. To retain the simplicity of context-free grammars while creating a more efficient natural language description we need to allow

for some context. Here, we are going to look at three well known grammars that extend CFGs each in their own way.

The first is *Transformational Grammar*, which adds to a context-free grammar a system of transformation rules that allow for the manipulation of the structure of the context-free sentences generated by the CFG rules. The second is *Lexical-Functional Grammar*, which adds to the CFG rules an external structure called an *attribute-value matrix*. This structure contains all the subcategory and semantic information of the sentence and is linked to the derivation tree via a series of functions. The third grammar, called *Head-Driven Phrase Structure Grammar*, actually alters the CFG itself by replacing the simple nonterminal symbols with complex symbols in the form of *feature structures*. In effect, they take the attribute-value matrix of Lexical-Functional Grammar, deconstruct it and insert it into the context-free grammar.

Unfortunately, all three of these grammars are quite large and have many intricate details that I would very much like to avoid. My goal here is to give you enough of an idea of how they work so that I can make clear why our grammar is preferable, without overwhelming you with too much information.

1.3.1 Transformational Grammar

Transformational Grammar (TG) is the alternative to CFG originally proposed by Chomsky in [1] and [2]. Since then it has gone through many revisions, most notably in [3] and [4], and is now referred to as *Government and Binding Theory*. In Government and Binding Theory (GB) a sentence has four forms, its *d-structure*, *s-structure*, *phonetic form* and *logical form*, plus a series of maps between them called *transformations*. The *d-*structure (for *deep structure*) is generated by context-free phrase structure rules and

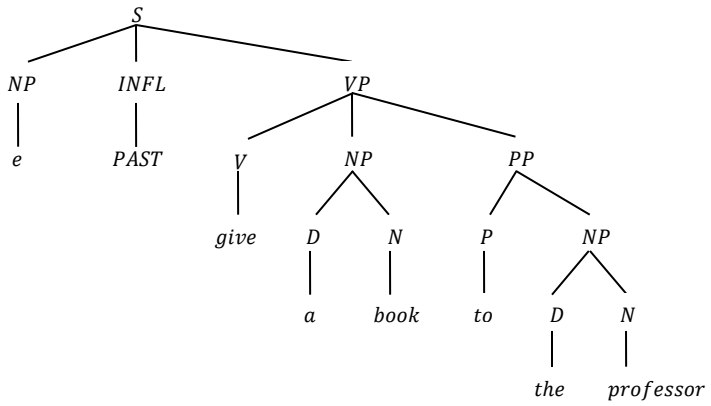
includes nonlexical information like placeholders for the various forms of a word, empty categories to mark movement and co-indexing to relate objects that are dependent on each other. The transformations alter the *d*-structure in ways that cannot be accounted for by CFG rules alone, producing the *s*-structure. The *s*-structure (for *surface structure*) provides the final underlying form of the sentence, which is then used to produce the phonetic interpretation (phonetic form) and the semantic interpretation (logical form[3]), respectively.

To illustrate how Government and Binding Theory works, we will look at an example of the passive sentence construction that incorporates all the elements that we need to look at in GB theory. Consider the passive sentence given in (8) below.

(8) *A book was given to the professor.*

The above is the phonetic form of the sentence whose *d*-structure is given in (9)[4].

(9)



The production rules needed to generate this *d*-structure are given in (10).

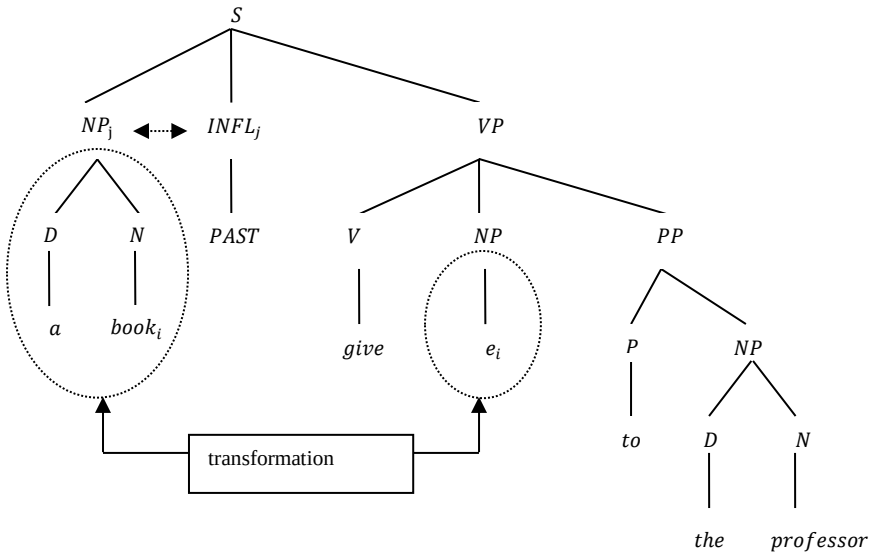
(10) $S \rightarrow NP\ INFL\ VP$
$NP \rightarrow D\ N \mid e$
$VP \rightarrow V\ NP\ PP$
$PP \rightarrow P\ NP$

[3] I will refrain from describing how each grammar deals with semantics as I have not added a semantic component to my own grammar as of yet.
[4] Note, this is the same *d*-structure as the active form of the sentence.

$INFL \rightarrow PAST$
$D \rightarrow a \mid the$
$N \rightarrow book \mid professor$
$V \rightarrow give$
$P \rightarrow to$

After the application of the transformations, trace insertion and coindexing the resulting *s*-structure is given in (11).

(11)



Here, we see the movement of the active object to the subject position in the passive via a transformation, coindexed by the subscript i. Also, the subscript j in the subject NP position and the $INFL$ position enforces subject-verb agreement. Finally, from (11) we construct the phonetic form given in (8) via lexical insertion rules, where we resolve the coindexed elements and the empty categories e_i (called a *trace*) and $PAST$.

Although Transformational Grammar is currently the model of choice for a majority of linguists, it has seen its share of opposition throughout the years. In [5], Peters and Ritchie show that, due to the transformations, every recursively enumerable language is generated by some context-free based transformational grammar, and vice-

versa. The implication here is that once a speaker acquires the grammar for a language they could then use that grammar to generate a nonrecursive language. Furthermore, Fodor, Bever, and Garrett [6] argue that although there is experimental data consistent with the claim that language, as a psychological model, is structural[5], there is no evidence to support the claim that the correct structural model is transformational. It is these two results which prompted Joan Bresnan to propose her *Realistic Transformational Grammar* [7], which later evolved into the Lexical-Functional Grammar.

1.3.2 Lexical-Functional Grammar

Lexical-Functional Grammar (LFG) was developed in the 1970s out of the work of two people, Joan Bresnan and Ronald M. Kaplan [8]. As stated above, LFG is a context-free grammar augmented by an external attribute-value matrix which houses all the subcategorization and agreement information that is accounted for by the transformations in TG. There are three important structures in LFG, *c*-structure, *f*-structure, and θ-structure. The *c*-structure is a derivation tree resulting from the CFG rules, the *f*-structure is the corresponding attribute-value matrix and the θ-structure contains information about the thematic roles (agent, theme, etc.) of the arguments of the predicate[6]. Furthermore, there exist *f*-descriptions, which are mappings between the *c*-structure and the *f*-structure that are used to pass information back and forth between the two.

To illustrate how these structures interact we go back to the example passive sentence (8) used above. For consistency sake we use the same production rules as those used above for TG[7] given here as (12). There is one major difference in that we add the *f*-

[5] As opposed to behavioural.
[6] Again, I will not go into any depth about the semantics.
[7] In reality, LFG uses some different category heads such as IP instead of S

descriptions to the nonterminals as tags, which carry information about the relationship between each node associated with the nonterminal and its place in the *f*-structure.

(12) $S \rightarrow \underset{(\uparrow \text{SUBJ})=\downarrow}{NP} \quad \underset{\uparrow=\downarrow}{INFL} \quad \underset{\uparrow=\downarrow}{VP}$

$NP \rightarrow \underset{\uparrow=\downarrow}{D} \quad \underset{\uparrow=\downarrow}{N}$

$VP \rightarrow \underset{\uparrow=\downarrow}{V} \quad \underset{(\uparrow \text{OBJ})=\downarrow}{NP} \quad \underset{(\uparrow \text{OBL}_{\text{goal}})=\downarrow}{PP}$

$VP \rightarrow \underset{\uparrow=\downarrow}{V} \quad \underset{(\uparrow \text{OBL}_{\text{goal}})=\downarrow}{PP}$

$PP \rightarrow \underset{\uparrow=\downarrow}{P} \quad \underset{(\uparrow \text{OBJ})=\downarrow}{NP}$

The arrows in the *f*-descriptions denote movement of information. Thus, ↑=↓ denotes that contextual information for that node comes from its mother node and is passed to its daughter nodes. The notation (↑SUBJ)=↓ denotes an insertion point for information from the *f*-structure.

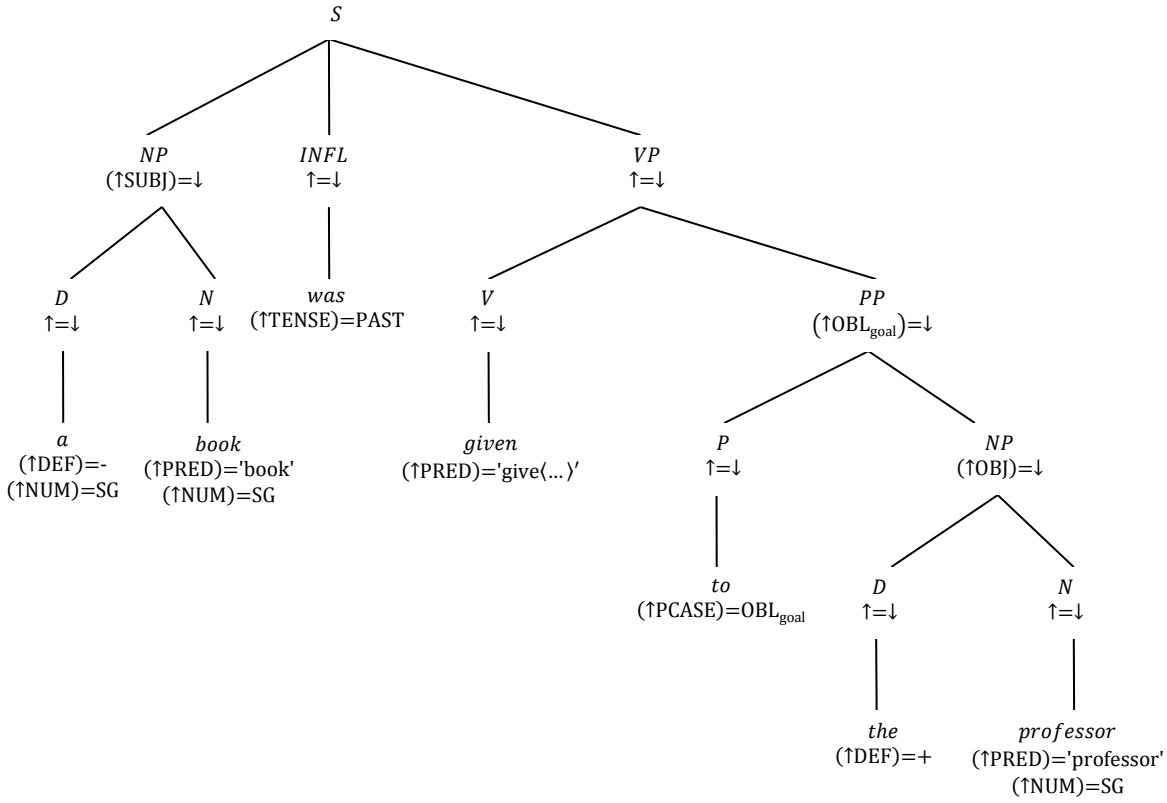
In LFG, the lexicon plays a bigger role in sentence generation. To accomplish this, entries in the lexicon appear not only with their category label but with *f*-description tags as well. These tags carry information about where and how the word can appear in the *c*-structure. In our example grammar the lexicon is as follows,

(13)

was	*INFL*	(↑TENSE) = PAST
a	*D*	(↑DEF) = -
		(↑NUM) = SG
the	*D*	(↑DEF) = +
book	*N*	(↑PRED) = 'book'
		(↑NUM) = SG
professor	*N*	(↑PRED) = 'professor'
		(↑NUM) = SG
given	*V*	(↑PRED) = 'give⟨∅ (↑SUBJ) (↑OBL_{Goal})⟩'
to	*P*	(↑PCASE) = OBL_{Goal}

This gives the *c*-structure;

(14)



The associated *f*-structure is a nested matrix of grammatical functions (attributes) and their values (possibly other *f*-structures). For this example, the *f*-structure is as follows,

(15)

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{DEF} & - \\ \text{PRED} & \text{'book'} \end{bmatrix} \\ \text{TENSE} & \text{PAST} \\ \text{PRED} & \text{'give}\langle \emptyset \quad (\uparrow\text{SUBJ}) \quad (\uparrow\text{OBL}_{\text{goal}}\ \text{OBJ}) \rangle\text{'} \\ \text{OBJ} & \\ \text{OBL}_{\text{goal}} & \begin{bmatrix} \text{PCASE} & \text{OBL}_{\text{goal}} \\ \text{OBJ} & \begin{bmatrix} \text{DEF} & + \\ \text{PRED} & \text{'professor'} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Note that, like in TG, for a passive sentence there is an indication of NP movement, although here it is shown in the *f*-structure and not the *c*-structure.

In LFG the work of the transformations in TG is done with the *f*-structure, the *f*-descriptions and stronger lexicalization. All the contextual information of a sentence is in the *f*-structure and the lexical entries. During derivation, pieces of that information are passed between the *f*-structure and the nodes within the *c*-structure. Although this strategy removes redundancy from the rules, it leads to redundancy in the *c*- and *f*-structures, as there is contextual information being repeated in both as well as over multiple nodes. In Head-Driven Phrase Structure Grammar, this idea is taken a step further.

1.3.3 Head-Driven Phrase Structure Grammar

Head-Driven Phrase Structure Grammar (HPSG) [9] evolved directly from Generalized Phrase Structure Grammar [10], while borrowing from other grammars including LFG and GB. At its simplest, HPSG takes the idea of LFG a step further by embedding the attribute-value matrix directly into the derivation, augmenting the context-free grammar with nonterminals that are themselves feature structures. So, a nonterminal such as V would now be denoted by a feature structure like the following.

(16)

$$\begin{bmatrix} word & \\ \text{HEAD} & verb \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \text{NP} \\ \text{COMPS} & \text{NP} \end{bmatrix} \end{bmatrix}$$

In particular, this complex symbol represents a verb that takes a noun phrase specifier and a noun phrase complement. In other words, it is the complex symbol representing a transitive verb.

Furthermore, the lexical entries are given in terms of feature structures as well[8]. Lexical entries consist of an ordered pair, the first value being the phonological form of the word and the second value a feature structure. For example, the lexical entry for *likes* is given in (17).

(17)

$$\langle likes, \begin{bmatrix} word & \\ \text{HEAD} & verb \\ \text{VAL} & \begin{bmatrix} \text{SPR} & \text{NP} \\ \text{COMPS} & \text{NP} \end{bmatrix} \end{bmatrix} \rangle$$

Finally, there are a number of general rules that constrain the way in which the feature structures can unify into a parse tree. There are two types of rules, grammar rules and lexical rules. These rules resemble the rewrite rules of more traditional phrase structure grammars but they use feature structures instead of simple nonterminals. For example, the Head-Specifier Rule, given as (18), expresses information about which features are shared between a phrasal node and its lexical head (denoted by **H**).

(18)

$$\begin{bmatrix} \text{phrase} & \\ \text{SPR} & \langle \ \ \rangle \end{bmatrix} \rightarrow \boxed{1}\ \mathbf{H} \begin{bmatrix} \text{SPR} & \langle \boxed{1} \rangle \\ \text{COMPS} & \langle \ \ \rangle \end{bmatrix}$$

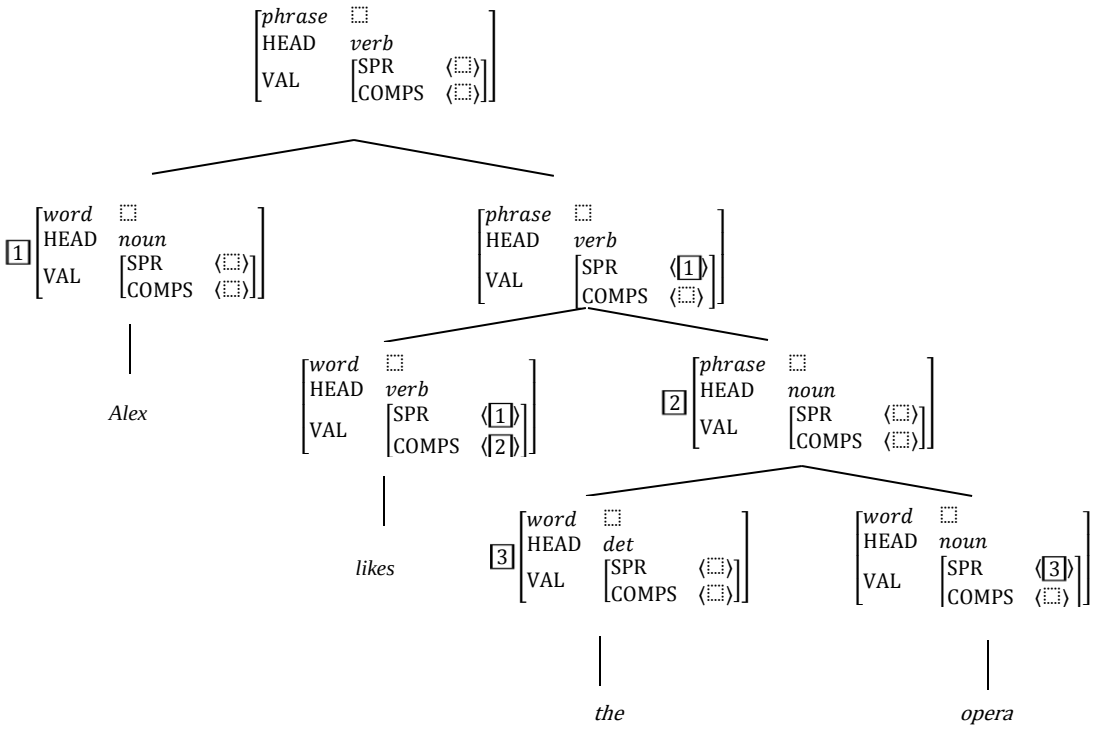
This rule states that a phrase can consist of a head (**H**) preceded by its specifier where the head and specifier have some features in agreement (denoted by the symbol $\boxed{1}$ appearing both before the head and in the specifier list).

Now, let us look at how HPSG uses these ideas to build a parse tree, given in (20), for the following sentence.

(19) *Alex likes the opera.*

[8] The terminals are still words, as in the other grammars.

(20)



HPSG solves some of the redundancy of LFG by removing the *f*-structure but there is still contextual redundancy within the derivation tree. It is still understood that the information is being passed around the derivation tree, although here it is in the form of unification of feature structures. We propose a grammar that has an external structure, like that of LFG, but with no redundancy and no context within the derivation.

1.4 Intensional Context-Free Grammar

With Intensional Context-Free Grammar we improve upon the ideas of these other grammars by extending CFGs with a feature structure we call the *version* (or *context*) *space*. Our use of version space derives from the *intensional logic* idea of context [11]. In intensional logic the truth of a logical statement is dependent on the context in which it is

uttered. That context is typically implied by the *possible world* in which you are immersed. For example, the truth of the sentence

(21) $The\ temperature\ is\ eighteen\ degrees$

is dependent on a context which consists of many parameters including the time of day, the city, country, year, the system of measurement (Celsius, Fahrenheit, Kelvin), etc. Most of the values associated with these parameters are implied. So, when uttered, it is implied that the current time and place is meant. Furthermore, if spoken outside to a friend, it is implied that it is the air temperature in Celsius (if in Canada). On the other hand, if in a chemistry lab, it may be referring to the temperature of a chemical in Kelvin. In this way we define a statement as an *intensional* statement which has multiple meanings, called the *extensions* of the statement.

The original motivation for intensional logic was to account for meaning (semantics) in natural language. But in fact, this idea of context can be extended to include structure (syntax) as well, meaning that syntactic derivation can be guided by this same implicit context space. The production rules in ICFG offer two ways of interacting with the context space, which extend the grammar beyond context-freeness. Firstly, we allow for different versions of our production rules, each coinciding with a context in which the rule applies. This is done with a system of rule labels we call *version tags*, denoted V, where V is a version expression (defined formally in Chapter 3) that represents the context in which you would choose the associated rule. Thus, we have *tagged production rules* denoted by $\underset{V}{\rightarrow}$. For example, in ICFG the two versions of the NP rule,

(22) $NP \rightarrow D\ N$

$NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$

correspond to four of the NP rules given in (7); a definite noun phrase (singular and plural), an indefinite, singular noun phrase, and an indefinite, plural noun phrase, respectively. The version tags are compared to the current context during derivation, all of which are points in the version space, and the formal structure of the version space provides us with a means of selecting the most appropriate tagged rule, even if no tag matches the current context exactly.

The second way a production rule interacts with the version space is through the use of *context modifiers*, denoted M, where M is a full version expression or a single context parameter with no values[9]. As the name suggests, context modifiers are used to change the current context temporarily. When a modifier is encountered during derivation, we continue on under the modified context until the relevant part of the derivation is completed. For example, the rule

(23) $VP \rightarrow V\ NP$[obj]

could be used in the derivation of *the dogs chase the cat*. This rule allows for the plural agreement of the subject noun phrase and the verb while forcing no such agreement on the object noun phrase. This particular type of context modifier is called the *drill-down* operator because it allows us to focus on one portion of the current context[10].

The use of version tags and context modifiers provide us with two benefits; (i) the current context is not explicitly tied up in the derivation. The context tags are used to guide our derivation based on the current context without the derivation having explicit knowledge of that context. Thus, there is no need to carry current context, or portions thereof, from node to node in the corresponding parse tree. (ii) We can introduce new

[9] The difference will be explained in Chapter 3.
[10] We explore the context modifiers more deeply in Chapter 3.

dimension values into the version space at any time. The context modifiers let us access the current context when needed, meaning there is no limit to what parameters can be assigned a value; features, thematic roles, language, dialect, discourse, etc. This leads us to believe that ICFG would be a perfect candidate for Chomsky's Universal Grammar [2], (briefly described at the top of page 2 in this dissertation.)

1.5 Conclusion

One component of generative linguistics I only mentioned briefly in the opening is the psycholinguistic view of the goals of a grammar. You can see from Section 1.3 that there is some overlap where these three grammars are concerned. I would contend further that there is one aspect of all three of these grammars that is the same. For each, the context is part of the derivation and thus part of the sentence. This implies a mental view that sees the context generated from within the sentence. We see this through their handling of long-distance dependencies. That is, in constituents that are separated by long distances in the parse tree which are dependent on each other in some way, for example pronouns that are dependent on the subject noun phrase.

In transformational grammar, long distance dependencies are dealt with through a series of local transformations that leave behind co-indexed traces. In LFG, the grammatical functions between the AVM and the parse tree, as well as the function tags in the rewrite rules, are used to pass context around from feature structure to parse tree and from node to node. In HPSG, the feature structures carrying the local contexts *are* the complex rewrite symbols and different constraints govern how these structures interact, again by passing information from node to node. In Intensional Context-Free Grammar

the sentence derivation is driven by the context space in which you are immersed. Meaning and context come first then the sentence representing the reality.

The remainder of this dissertation is dedicated to expanding on the ideas introduced here as well as introducing some other related topics. We begin in Chapter 2 with an introduction to intensional logic, the paradigm underlying the intensional context-free grammar presented in this dissertation. Included in this chapter is a description of intensional logic and its roots in semantic analysis. Furthermore, we see how this intensional foundation leads to applications of the concept in programming languages, software versioning, webpage design, and more.

In Chapter 3 the ideas from the previous two chapters come together as we formally introduce intensional context-free grammar. Here we describe the structure of the production rules and their variation on traditional context-free rules; the version tag. We also look at the context modifier operations and the best-fit algorithm used to select the appropriate version of a given tag in the version space.

Chapters 4 and 5 contain an exploration of the mathematical properties of ICFG. In Chapter 4 we prove that the language of an ICFG is an indexed set of context-free languages. We also define the fixpoint semantics of ICFG in Chapter 5. We follow this with a proof of the equivalence between the stringset language produced by the fixpoint semantics and the derivation language produced by a direct application of the rewrite rules.

Chapter 6 presents an application of ICFG, in the area of natural language processing. The chapter begins with an introduction to basic linguistics. We then present a small grammar in the style of ICFG and discuss how the grammar works. Furthermore,

we expand the grammar with a few more complex structures (ex. passive sentences) and see how ICFG can be used to handle them.

Chapter 7 describes the implementation of the grammars developed in Chapter 6 in the macro language developed by Dr. William Wadge called MMP. Here we give a brief introduction into MMP (full details of which appear in the appendix) and its use for this dissertation. We follow this with a presentation of the NLP grammar created by the author and the results obtained from it. We close this chapter with an exploration of the use of ICFG in machine translation.

Chapter 8 is the conclusion of the dissertation, in which we present our final remarks on the topic. Included is an analysis of the benefits of our grammar and a discussion of other areas to which we could apply ICFG. We conclude with a discussion of some of the future work that could be generated off of this topic.

2 Intensional Logic and Applications

In Chapter 1 we looked at three formal generative grammars, TG, LFG, and HPSG. We saw that none of them represent a realistic mental model for natural language processing. In Chapter 3 we introduce what we believe to be a better alternative to these grammars. Before we can do that we must look at the evolution of the major tool we will be using, *intensionality*. Thus, in this chapter we introduce the idea of intensionality which comes from intensional logic, presented in the first section. In the rest of the sections we look at one particular evolutionary stream of intensional logic, the application of intensionality to computer programming in the form of intensional programming languages, intensional software versioning and intensional markup languages.

2.1 Intensional Logic

In intensional logic the truth assignment of a sentence is dependent on the context or *possible world* in which it is stated. Typically, this context is not stated explicitly but is implied by the world in which the statement is uttered. A statement itself is defined as the *intension* whereas the interpretation of that statement, in a given context, is defined as the *extension*. In this section, we look at the contributions of four people to the evolution of intensional logic; Rudolph Carnap, Saul Kripke, Dana Scott and Richard Montague. Note that these ideas did not begin with these four particular people. The importance of their work is in the formalization of these ideas in a semantic sense as opposed to a strictly syntactic approach. In fact, the notions of intension, extension and possible worlds had been explored in modal logic in the past, but it is not my goal here to present a complete

history of modal logic. On the other hand, I do want to present some background knowledge of modal logic as it is the foundation on which intensional logic is built.

In traditional propositional logic propositional symbols represent sentences. For example, we can let P be the sentence $Bill\ is\ right$. Such a proposition is said to be either true (1) or false (0) independent of context. Then, more complex sentences (also called *well-formed formulas* or *wffs*) are formed by combining atomic propositions and logical operators, typically *negation* ($\neg$), *conjunction* ($\wedge$), *disjunction* ($\vee$), *conditional* ($\rightarrow$), and *biconditional* ($\leftrightarrow$). Complex sentences are truth-valued dependent on the truth-values of the atomic propositions and the properties of the logical operators. Thus, for example, if P is as above and Q is $it\ is\ raining$, then $(P \wedge Q)$ is true if and only if Bill is right and it is raining.

Predicate logic extends propositional logic by representing sentences with predicates that may contain variables representing possible entities in the sentence that can be quantified. Quantification is represented by additional logical operators called quantifiers. Typically the quantifiers of predicate logic are the *universal* ($\forall$) and the *existential* ($\exists$). For example, if we let $P(x)$ represent $x\ is\ right$, then we can form the new sentence $\forall xP(x)$ meaning $Every\ x\ is\ right$. So, if our universe of discourse is the set of all Bills, then $\forall xP(x)$ can be interpreted as $Every\ Bill\ is\ right$.

Although the dominant form of logic throughout the history of the study of formal logic, as well as being very powerful and important in the furtherance of mathematical and philosophical thought, neither propositional nor predicate logic are expressive enough to represent fully all natural language sentences. In the same way that quantifiers

and variables are used to extend the propositional logic to account for more complex sentences, modal logic introduces operators that express modality in sentences.

Modals are words that qualify a sentence like $always$ in the sentence $Bill\ is\ always\ right$. This sentence expresses something beyond the present as in $Bill\ is\ right$. It says that in all the time that Bill has existed he has always been and will always be right. Modal operators work in much the same way as quantifiers but instead of ranging over entities (like nouns), they range over contexts (possible worlds). The most common modal operators are *necessity* (□) and *possibility* (◊). Thus, $Bill\ is\ always\ right$ can be represented by $\Box P$, interpreted as $necessarily\ Bill\ is\ right$.

In modal logic, truth-value assignment must also be extended to include the modal operators. It is understood then, that $\Box P$ is true if and only if P is true in all contexts and $\Diamond P$ is true if and only if P is true in at least one context. The nature of the contexts over which the modals range and how they affect truth assignments are major parts of the discussion of the next four sections and contribute directly to the evolution of intensional logic.

2.1.1 Rudolph Carnap

In *Meaning and Necessity* [11], Rudolf Carnap looks to assign meaning to sentences based on implicit contextual information. This work is the culmination of work done by Carnap from 1934 to 1955. In particular, he wants to provide a semantic analysis of logical truth in modal logic to remedy some of the problems posed by modal logic up to that point. Modal logic had been around since the time of Aristotle where he proposed the

ideas of necessity and possibility, but had not been as well studied as conventional logic in which truth-values are assigned absolutely.

One of the major stumbling blocks for modal logic was its apparent violation of the *law of substitution*. Carnap uses the following example to describe the problem. Given the true statements

(1) $The\ number\ of\ planets\ =\ 9$[11]
(2) $9\ is\ necessarily\ greater\ than\ 7$

we can produce the false statement

(3) $The\ number\ of\ planets\ is\ necessarily\ greater\ than\ 7$

by the substitution of identicals.

Carnap saw that the reason for the breakdown lies in the nature of the statements themselves. That is, the truth of (1) is *contingent*; it did not have to be true, whereas the truth of (2) is *logical*. Thus, in producing sentence (3) we are mixing statement types. The problem pointed out by Carnap is that the truth of (1) is extensional, in that it is true in our world but it did not have to be that way (and in fact is not now, see footnote 11 below). The intension, *the number of planets* has possibly different extensions in different worlds whereas the intension *9* has the extension *9* in all worlds.

Formally, Carnap proposes that the intension of a statement is the proposition expressed by it while the extension is its truth-value in some world of reference, called a *state-description*. The idea of state-description evolves into possible worlds with Kripke and Scott but for Carnap a state-description is a class of sentences which contains, for every atomic sentence, either this sentence or its negation. He also defines a series of terms which distinguish between these two concepts. For instance, he defines *truth* to be

[11] Note that this example comes from [12] which was written before Pluto was down-graded from a planet to a dwarf planet.

extensional truth and *L-truth* to be intensional truth and similarly, *equivalence* and *L-equivalence,* etc.

Thus, for example, the sentence

(4) $Dion\ Phaneuf\ is\ the\ captain\ of\ the\ Toronto\ Maple\ Leafs$

is true because the person named Dion Phaneuf does happen to be the captain of the hockey team named the Toronto Maple Leafs at this time. On the other hand it is not *L*-true, since we can imagine a world in which the captain is someone else. In fact, ten years ago the captain was Mats Sundin and thus that would be some such world. The intension expresses something beyond the extension. $The\ captain\ of\ the\ Toronto\ Maple\ Leafs$ entails many properties at different times (he plays for the team, wears a C on his jersey, leads the team, etc.), which $Dion\ Phaneuf$ does not.

With these new definitions Carnap then proposes a new law of substitution in which there is extensional substitution and intensional substitution (called *L-substitution*). Thus, you may *L*-substitute one expression for another if and only if they are *L*-equivalent. This prevents examples like in (1)-(3) since the intension of $the\ number\ of\ planets$ is not *L*-equivalent to the intension of 9, even if they are equivalent in the extensional sense for some world (in particular, the world in which Carnap lived). It turns out that Carnap's completeness proof fails because some of his notions are incorrect (particularly in his failure to connect state-descriptions to possible worlds semantics [12]) but it lays the groundwork for Kripke's possible world semantics.

2.1.2 Saul Kripke

In [13], [14], and [15], Saul Kripke gives the first formal account of modal logic. In fact, he presents soundness and completeness results for a number of modal logics over the

course of these three papers. For Kripke a model for a modal logic is a triple (G, K, R) where K is a nonempty set of possible worlds, $G \in K$ is the actual world and R is an *accessibility relation* on K. Kripke shows that by altering the properties (reflexive, transitive, etc.) of the relation R, you actually change the model to account for different modal logics. A model for a well-formed formula A of a modal logic is a binary function $\Phi(P, H)$ where P is a variable ranging over the subformulae of A and H is a variable ranging over possible worlds such that $\Phi(P, H)$ is true or false. Given this definition then, for Kripke $\Phi(\Box P, H)$ is true if $\Phi(P, H')$ is true for all $H' \in K$ such that HRH'. That is, P is necessary in world H if and only if it is true in all possible worlds accessible from H.

In Kripke's model not much is said about the nature of the possible worlds themselves. For him there is no set structure to these worlds, just a theoretic representation of the complex properties that make up a possible world and a way to access that world. On the other hand, he does spend some time on the nature of the individuals that populate these worlds. Formally, the set of individuals that the atomic variables range over are consistent from world to world. That is, no matter what different properties the worlds may have and how those properties affect the individuals in those worlds, the individuals are the same. In fact, this idea is embraced by all four authors discussed in this section and is explored thoroughly by Kripke in his philosophical text *Naming and Necessity* [16]. The main point is that a name given to an entity in some world is the same in all worlds whether the properties of that entity change, even if that entity does not in fact exist in that world. Consider for example the sentence,

(5) $Dion\ Phaneuf\ was\ not\ the\ captain\ of\ the\ Toronto\ Maple\ Leafs\ in\ 1962.$

This sentence is true and to be able to make this truth assignment we need to reference an individual who was not alive in the stated possible world of 1962.

Of course, the work done by Kripke is groundbreaking in modal logic semantics, so much so that many refer to this type of possible world semantics as Kripke semantics. Unlike Carnap's semantics, Kripke's semantics for modal logic are correct but in both cases they are confined to modal logic. For us, it is not the modal logic semantics that is of importance but the formalization of intension and extension from Carnap and possible worlds from Kripke and how these ideas influence both Scott and Montague, eventually leading to the work presented in this dissertation.

2.1.3 Dana Scott

In *Advice on Modal Logic* [17], Dana Scott lays out a semantics for modal logic which borrows from and improves upon the ideas of Carnap and Kripke. Scott refines the formalization of intensions and extensions with a correct and concise notation and terminology and provides a new conception of what the possible worlds should look like. Although he does not call it thus, Scott in fact develops the first fully formed intensional logic that is actually quite accessible.

Scott's biggest innovation, the one that is particularly relevant to the development of intensional programming, is his view of the structure of the possible worlds. For Scott, instead of complex sets with undefined structure and an accessibility relation, possible worlds are represented by coordinate points, which he calls *points of reference*, in an index set where each coordinate can vary independently of the others. This allows us to state explicitly the factors that are relevant to the interpretation of the logic's semantics. It also allows the logic to be much more flexible by putting most of the work of interpreting

different modal operators in the index set. So, the possible worlds is a fixed set I of indices where, for each $i \in I$, we can incorporate into i as many coordinates as necessary to account for the relevant properties of the possible worlds for that logic. For example, it could be that $i = (w, t, p, ...)$, where w is a world, t a time, p a spatial coordinate, etc.

Another innovation of Scott is his division of the types of objects or *individuals* in the logic. He proposes three sets of individuals, $A_i \subseteq D \subseteq \mathrm{V}$, where the A_i are a family of domains of *actual individuals*, one for each $i \in I$, D is the domain of all *possible individuals* and V the domain of *virtual individuals*. The distinction between these sets is as follows; suppose we let I represent time points, then the A_i would be the set of people alive at time i, whereas D is the set of all people alive or dead (and possibly yet to be born). The set V of virtual individuals would denote ideal individuals, that is, abstract entities like *the King of France*. Essentially, you can think of the set D as the extensional individuals and V the intensional.

For Scott the intension of a proposition φ, denoted $\|\varphi\|$, is a function from points of reference in set I into the set of truth values, $\{1,0\}$. The extension of φ at i, denoted $\|\varphi\|_i$, is the truth-value of φ at some particular $i \in I$. Furthermore, given a term τ, Scott makes a distinction between *individual concepts*, denoted by $\|\tau\|$, and individuals, $\|\tau\|_i$, which is the value of the individual concept at some $i \in I$. That is, the intension of an individual and its extension in some world. You can then define the semantics in these terms, for instance, necessity is defined by $\|\Box\varphi\|_i = 1$ if and only if $\|\varphi\|_j = 1$ for all $j \in I$ and equality is defined by $\|\tau = \sigma\|_i = 1$ if and only if either $\|\tau\|_i = \|\sigma\|_i$ or neither are defined.

Let us consider a previous example to illustrate some of the above ideas. Let φ be sentence (4), restated here as (6),

(6) *Dion Phaneuf is the captain of the Toronto Maple Leafs*.

Furthermore, let τ be the term *Dion Phaneuf*, σ the term *the captain of the Toronto Maple Leafs*, and I the set of all NHL seasons. Then $\|\tau\|_{2013} = \|\sigma\|_{2013} =$ Dion Phaneuf, whereas $\|\tau\|_{1962} \neq \|\sigma\|_{1962}$ since $\|\sigma\|_{1962} =$ George Armstrong and not Dion Phaneuf. Consequently, $\|\varphi\|_{2013} = 1$ but $\|\varphi\|_{1962} = 0$ and thus, in this case, $\|\Box\varphi\|_i = 0$.

Scott's is a fully formed intensional logic which directly leads to the ideas of intensional programming, intensional versioning and eventually intensional context-free grammars. At around the same time Richard Montague was developing an intensional logic of his own, although with different motivations. It turns out that both their logics are quite similar[12] but in most of the intensional programming literature the focus is on Scott's contributions. I am going to briefly outline some of Montague's work too because his motivations more closely resemble my own, that is, in developing a model for natural language processing.

2.1.4 Richard Montague

Montague's intensional logic resembles that of Scott's in overall ideology but they are different in the details. Montague's logic is much less accessible due to notational differences and the ad hoc nature of its creation. As his development straddles that of Kripke and Scott he borrows from both their work as well as Carnap's. For example, Montague originally foresaw possible worlds much the way Kripke did but after *Advice* [17] he began to use an index set, although with only two coordinates; possible world and

[12] In fact, in a letter from Montague to Scott, Montague outlines some of these similarities [16, p. 173].

time. Of course the major difference is in their motivations for creating the logic. In some ways Montague had a more limited scope while in others it was much more ambitious as his goal was to develop a formal logic of natural language.

In a series of papers from 1960 to 1970, collected in *Formal Philosophy* [18], Richard Montague develops his intensional logic for the semantical analysis of natural languages. In the first two papers, *Logical Necessity, Physical Necessity, Ethics, and Quantifiers* and *'That'*, Montague explores a semantics for modal logic in much the same way that Kripke did. He suggests that a general modal operator is in order to cover the different ideas of logical necessity, physical necessity and ethical necessity. He also combines them with quantifiers, something that had not been done at the time. Furthermore, he cites Kripke's completeness theorem as applying to his general modal operator.

In the next two papers, *Pragmatics* and *Pragmatics and Intensional Logic*, he adopts Scott's idea of possible worlds as points of reference. For Montague, pragmatics is the key to developing a formal logic of natural language as opposed to syntax and semantics alone. Pragmatics is defined by Montague as meaning in context, that is, intensional semantics, whereas traditional semantics is meaning without context, or extensional semantics.

These ideas culminate with *The Proper Treatment of Quantification in Ordinary English*, where Montague presents his fully formed intensional logic for a fragment of English. He does this by first defining a syntax for the fragment of English in the style of category grammar. In category grammar you define a small number of categories containing the words of your grammar, grouped by word type, and a number of rules for

combining categories. For example, in Montague's notation, $B_{IV} = \{run, walk, talk, \ldots\}$ is the category of intransitive verbs, $B_{t/t} = \{necessarily\}$ is the category of sentence modifying adverbs, etc. He then defines his intensional logic syntax and semantics.

The intensional logic syntax defines categories by type; type e the category of entity expressions (those with entities as their extensions), type t the category of truth value expressions (those with truth values as their extensions), and a notational object s, the *sense* of an object of one of the other types. The syntax also provides ways of combining these types of categories into new categories; if a and b are types then $\langle a, b \rangle$ is a type and $\langle s, a \rangle$ is a type. There is a one-to-one correspondence between these categories of types and the categories of the English syntax, given above, by a translation map. Thus, given the semantics of the intensional logic we can define the semantics for the English grammar. In general, rules that are of the form category A/B combine with category B to give category A and the intension of categories A/B and B give the extension of category A.

Much of the work in intensional programming is based on Scott's intensional logic, from Lucid[13] to ICFG. But the importance of Montague's ideas, to this dissertation, should not be overlooked because it is his intensional logic, although not the one used, that inspired the present author to investigate the use of intensional logic in natural language syntax. Similar to Richard Montague, we want to create a formal grammar using the foundations of intensional logic to facilitate natural language generation and comprehension. Unlike Montague, we use a fully syntactic approach where we have a context-free grammar informed by a possible world context space in which the context is

[13] Introduced in Section 2.2.1.

syntactic rather than semantic. That is, the values of the indices indicate information about the syntactic nature of the possible world. This is not to say that semantics could not be a part of the properties of the possible worlds, it is just not necessary for our purposes[14].

2.2 Intensional Programming

The Intensional Programming paradigm has its roots in the Lucid[15] programming language [19] and all of its successors and extensions. Originally envisioned as a language for manipulating infinite data structures via iteration as a formal description of computation, Lucid eventually evolved into a dataflow language with infinite data streams (pLucid [19], LUSTRE [20], Ferd Lucid [19, pp. 217-222] and ILucid [19, pp. 223-227]) and then finally into an intensional programming language (Field Lucid [21], Indexical Lucid [22], Granular Lucid [23], Plane Lucid [24], [25], TLucid [26], Tensor Lucid [27], and TransLucid [28]). Along the way, the main ideas of intensional logic; intension, extension and possible worlds, have been applied to Lucid and have evolved within the paradigm. I will avoid a detailed explanation of all the flavours of Lucid here, for that there are other sources, for example [29]. The goal in this section, as in the last, is to focus on the application of the ideas of intension, extension and possible worlds and to explore how they have evolved and informed the work covered later in this dissertation.

[14] On the other hand, as stated in the future work section of the conclusion, it would definitely be fruitful, possible and eventually necessary to incorporate semantics into the system.

[15] We use the term Lucid to cover the entire family of Lucid programs, typically the first Lucid is referred to as Original Lucid to avoid confusion.

2.2.1 Lucid

In the early versions of Lucid (pre-intensional), a variable is an infinite sequence (or data stream) which takes on different values at different time points. For example, variable $X = \langle x_0, x_1, \ldots, x_i, \ldots \rangle$ has value x_i at time i. The temporal operators, first, next, and fby, are defined to reference the values of the variable at different times. Thus, whenever $X = \langle x_0, x_1, \ldots, x_i, \ldots \rangle$ and $Y = \langle y_0, y_1, \ldots, y_i, \ldots \rangle$, then

(7) first $X = \langle x_0, x_0, \ldots, x_0, \ldots \rangle$,

(8) next $X = \langle x_1, x_2, \ldots, x_{i+1}, \ldots \rangle$,

(9) X fby $Y = \langle x_0, y_0, y_1, \ldots, y_i, \ldots \rangle$.

Multidimensional versions of Lucid are also defined, which either add arbitrarily more time dimensions, with the new notation $X_{t_0 t_1 t_2 \ldots}$ for the value of X where the i^{th} time dimension has value t_{i-1} for $i \geq 1$, or maintain one time dimension but add arbitrarily more space dimensions, with the notation $X_t^{s_0 s_1 s_2 \cdots}$ for the value of X at time t where the i^{th} space dimension has value s_{i-1} for $i \geq 1$. New operators are added with similar functionality as above but manipulate the new dimensions.

The standard implementation of all Lucid flavours is a demand-driven technique called *eduction*. To *educe* the value of a variable at some time you make a request in the form of a (*variable, tag*) pair. From there, one of three things will occur; the requested value already exists in the Lucid cache called the *warehouse*, the evaluation is dependent on other (*variable, tag*) pairs, also requested, or the value is calculated.

With their 1987 paper *Intensional Programming* [21], Antony Faustini and William Wadge re-envisioned Lucid as an intensional programming language via the intensional logic of Scott and Montague. In intensional versions of Lucid, expressions are

intensions mapping tags (possible worlds) to extensions as values. So, a variable X is now an intension with values as extensions at possible worlds i. As before, in single dimensional Lucid, that world is a time point but it can also be multidimensional as in Scott's points of reference.

The basic functionality of pre-intensional Lucid is maintained, but interpreting Lucid as intensional provides some major benefits. The intensional versions solve an ongoing problem of the early versions of Lucid by providing a means of equating the denotational and operational semantics [30]. Furthermore, it allows Lucid to evolve in ways which may not have been foreseen in the earlier versions. This evolution mostly comes about through the nature of the possible worlds themselves. Originally, a possible world is a point of reference with values as time and/or space dimensions accessed through indexing. With the development of intensional software versioning, coordinate values in a point of reference become dimension-value pairs and an algebra is defined on the set of all possible worlds, which allows for a more flexible interaction between intensions and extensions.

2.2.2 Software Versioning

One particularly important extension of the intensional programming group is the development of version control tools in software development. In 1993, John Plaice and William Wadge [31] applied intensional contexts to software versioning by viewing the different possible versions of software components as possible worlds. Thus, they created a software versioning system that used an arbitrarily-dimensioned, *uniform version space* shared by the entire system, in which a complete software system is formed by taking the most relevant version of each component.

To do this they develop a *version algebra* partially ordered by an operation called *refinement*, denoted by $\leq$. Thus, $V \leq W$, read as W *refines* V, means that version W of a particular component is an extension (or improvement or a direct alteration) of version V. The simplest version of any component is called the *vanilla* version and is denoted by ϵ. The system also allows you to *join* versions, denoted +, defined to be the least upper bound induced by the refinement relation. That is, $V_1 + V_2$ is the least upper bound of versions V_1 and V_2 if and only if for all V such that $V_1 \leq V$ and $V_2 \leq V$, then $V_1 + V_2 \leq V$.

When constructing the complete version of a software system, you need a way of selecting the appropriate component versions. In the possible worlds versioning system of Plaice and Wadge this is called the *variant substructure principle* (later to become the *best-fit algorithm*), with which the most relevant version of each component is selected, based on the refinement relation. Thus, the intension of a software system is the family of all versions of the components, while the extensions are the particular versions that are assembled based on the variant substructure principle. In this way you avoid duplication of components while ensuring that a relevant version of each component exists, to allow for the assembly of the whole.

2.2.3 Web Authoring

The version control system devised by Plaice and Wadge is an important system in the intensional programming world. One of the first uses of the version space phenomenon (outside of software versioning) comes in the form of web authoring languages. The successive web authoring programs Intensional Hypertext Markup Language (IHTML) [32], IHTML2 [33], Imperative Scripting Language (ISE) [34], Intensional Markup Language (IML) [35], and the æther [36] are created, which all extend regular HTML

with intensions. This is done by revising the version space system and best-fit algorithm to account for dimension labels and the use of contexts as values themselves.

2.2.3.1 IHTML

IHTML allows authors to define a whole indexed family of HTML variants using a single source file. The intension is the family of HTML pages while each individual page serves as an extension. So, authors can provide multiple sources for the same page where each source is labeled with a different version. The version space is partially ordered using the updated versioning system. The version dimensions can be attributed to any of the markup elements of traditional HTML.

In the software versioning system, dimension labels are implied, they are not part of the system itself. In IHTML [32], Taner Yildirim proposes the use of explicit dimension identifiers separated from the value of that dimension with the ':' symbol. For example, we can have the following version of some web page:

(10) `platform:Mac+lang:French+cuisine:chinese`

which is the Macintosh version of a French page on Chinese cuisine.

Another new and important idea in IHTML is the *transversion* link. Transversion links are links that are used to change the context of the current version, allowing for the first time travel across possible worlds. Transversion links are regular links with a `vmod` tag that provides a version expression representing the requested version. As a consequence, a different version of the target page is linked to. In this way you can select the appropriate versions of pages via the current context or you can select the context of the version you are looking for. This modification of the current context is relative. That is, only the values of the dimensions listed will change, the rest will remain as they are.

For example, suppose your current context is `language:English+background:blue` and we have a link of the form

(11) `<a href=”page1” vmod=”language:French”>`.

This link will be interpreted as a request for the `langage:French+background:blue` version of page 1, which now becomes the current context.

2.2.3.2 IHTML2

With IHTML2 [33], Gordon Brown improves upon IHTML in a number of ways, much of them to do with implementation on the server-side and overall efficiency of the system. What is of relevance to us is the addition of dimensions as values, the corresponding change in the best-fit algorithm and the addition of the `version` tag.

An IHTML2 site has a version space in which its pages exist. A version space here is a set of dimensions with a set of indices, including ϵ. A version is a point in this dimensional space. An omitted dimension is implied to be the vanilla version of that dimension.

IHTML2 also introduces the difference between a *version* and a *version modifier*. A version is referenced absolutely using the `version` tag to allow for links to some page(s) unrelated to the current version. A version modifier retains its meaning from IHTML. That is, it is used to change particular dimensions relative to the current version using the `vmod` tag. In both IHTML and IHTML2, after being chosen by the best-fit algorithm, the requested IHTML page is translated into a regular HTML page and loaded into a browser for viewing.

2.2.3.3 ISE

For part of his Master's thesis [34], Paul Swoboda took the next step in intensional web authoring and created a versioned scripting language called ISE, in the spirit of Perl but using versioning for all of its identifiers, including files, variables, and functions. In doing so he also extends the version space by allowing dimension identifiers to be nested arbitrarily deep. Thus, allowing the value of a dimension to be a version expression itself.

Consider the following example version tag given in (12).

(12) `subj:<pred:cat+num:pl+def:no>`

This is the version tag for a subject noun phrase representing the indefinite, plural form of `cat`. Here the dimension `subj` has value `pred:cat+num:pl+def:no` which is itself a point in the version space with three dimensions.

One result of dimension nesting is that dimensions now distribute over joins in the version algebra. Thus, (12) is equivalent to

(13) `subj:pred:cat+subj:num:pl+subj:def:no`.

As a result of both dimension nesting and the distributive property, the new version algebra also allows a single dimensionless value in a version tag, called the *base value*. Consider the alternate version tag for the same object given as (14).

(14) `subj:<cat+num:pl+def:no>`

Here, `cat` is the base value for `subj`, allowing for version tag `cat+num:pl+def:no` in the algebra.

2.2.3.4 IML

In [35], Wadge adds Intensional Markup Language (IML) to ISE as a front end to get back to the simplicity of markup while still maintaining the power of imperative

programming. IML uses troff macros to extend HTML and is translated into ISE which in turn is translated into HTML readable by your browser. Wadge and monica schraefel [37] further enhance IML by implementing a user model to create an adaptive hypermedia system. This system selects the appropriate pages for a user based on their context in terms of a user profile. They also create an adaptable hypermedia by allowing users to determine a version explicitly, something not typically done in the intensional paradigm.

2.2.3.5 The Æther

In 2003, in his PhD thesis [36], Paul Swoboda created *the æther*. The æther is the natural culmination of the evolution of the context space, making it an active participant in the environment it encapsulates. Formally, the æther is a context space in combination with a number of registered participants. When the context space is modified it broadcasts the context changes to all the participants. In this way, a web community can communicate and interact through the æther with the æther as a dynamic participant.

There have been a couple of reasonably successful implementations of the æther in web authoring. Swoboda, John Plaice, and Ammar Alammar developed a collaborative intensional web page presenting paintings kept at the Louvre [38]. The page allows users to view what other users are viewing while maintaining their own view, each with their own contexts. Blanca Mancilla [39] developed a server, the Anita Conti Mapping Server (ACMS), which has an intensional web page containing an intensional map, each with its own context. With the web page and map, browsers collaborate through an æther allowing multiple users to change both the page and map.

2.2.4 Extending Lucid

With the development of the version space and its application to web authoring, much of the focus in the intensional programming paradigm switches to the context itself. We saw in the previous section how this evolved through intensional web page design, culminating in the distributed web communities of the æther. Simultaneously, these ideas feed back into the intensional programming stream and provide new extensions of Lucid.

The first application of the version space to Lucid is in Plane Lucid, in which Weichang Du and Wadge [24], [25] developed an extension of Lucid that uses the best-fit algorithm to apply the most relevant definition of a variable before evaluating that variable. They developed Plane Lucid as the language for an intensional 3D spreadsheet program. The spreadsheet is in fact represented by an intensional variable S that has as its extensions values in three dimensions, two spatial and one temporal, representing cells in the spreadsheet.

The next step was to let dimensions themselves become first-class values. Originally, this was done in Tensor Lucid [27], which allowed declared dimensions to be used as values. With Multidimensional Lucid [40] though, this idea was simplified by letting any ground value become a dimension. This freed up the Lucid system by allowing the creation of dimensions on the fly. On the other hand, this caused implementation problems with the eductive algorithm due to the possibly unbounded memory use of the warehouse.

The current version of Lucid is called TransLucid [28]. With TransLucid they have again altered the version space, now to allow for contexts as first-class values (an idea that first appeared in [41] . This is a combination of the ideas of dimensions as

values in Multidimensional Lucid and dimension nesting from ISE. They have also developed a new implementation mechanism called *lazy eduction* to remedy the memory issues mentioned above. With lazy eduction requests for values are built up one dimension at a time from the warehouse beginning with an empty tag.

2.3 Conclusion

The one area that has seen little use of intensionality is in natural language processing and generative grammars. Considering the original motivation of Richard Montague for intensional logic, natural language semantics, this seems remiss. Not to say that there has been no work in this area. In 1994, Senhua Tao [42] proposed an Indexical Attribute Grammar (IAG) which is an attribute grammar defined over an indexical context space. His purpose was to give an alternative method for describing the semantics of programming languages. In IAG, attributes are intensions over a three dimensional (tree, multitime, and identifier) context space. Thus a (tree, multitime, identifier) point constitutes a possible world. Obviously, this is a very specific grammar with a narrow use of context and would not be useful for natural language generation.

There have also been two examples of sentence generation using the intensional paradigm. In both cases, the generator is built on ISE and thus is web specific. Both use a small grammar and lexicon and construct French sentences by an informal method. In Wadge & Wadge [43], they use an intensional web site to generate French sentences by giving a user a table of options for different lexical categories and grammatical functions. For example, you can select a subject, verb, and two objects. For the nouns you have the choice of nominal or pronominal, singular or plural and an optional adjective. Your verb can be positive or negative and can have an adverb. But, sentence construction here is not

via generative grammar methods, that is, there are no CFG-like rules. In actuality, their program selects the correct sentence from a list of possibilities based on context as opposed to constructing it. The choice of sentence just becomes another option in the overall page construction of the web site.

In Li [44], French sentences are also generated, with more choices and grammatical functions, using IML. French e-Flash Card (FFC) is the resulting system; it is composed of a French knowledge base, a logic engine and a presentation layer. FFC allows users to produce sentences dynamically. It also allows the user to select a verb to be conjugated. But, again it is not a true generative grammar. So, the idea of a generative grammar based on intensions is relatively unexplored and this is precisely what we propose to do with this dissertation.

3 Intensional Context-Free Grammar

Here we present our generative grammar, *Intensional Context-Free Grammar* (ICFG). Like the grammars presented in Chapter 1, ICFG consists of context-free phrase structure rules that work in coordination with an external structure containing information about the syntactic context; subcategorization, agreement, etc. That external structure for us is the version space that has developed within the intensional programming paradigm presented in Chapter 2. Unlike those grammars, the external context structure is not tied up in the derivations generated by the phrase structure rules. For us, derivation instead is encapsulated within the version space implicitly. That is, the version space is seen as a whole, encompassing and driving everything that occurs within a derivation without having to explicitly become a part of the derivation.

We accomplish this by attaching version tags to our production rules. The version tags contain version expressions, which represent points (or possible worlds) in the version space. During derivation there exists a current context, also a point in the version space, representing the current possible world. Thus, at any given point in a derivation, the appropriate production rule is chosen based on the best-fit algorithm between the version tags and the current context, using the refinement relation. In this way, tagged production rules are intensions whose extensions are the terminal strings generated by those rules in some particular context. Furthermore, a grammar is an intension whose extensions are the languages generated in some particular context.

We now proceed with the formal introduction of intensional context-free grammar. We begin by introducing the version space and summarizing the results of the previous chapter on the version algebra. The version space used here is that of Paul

Swoboda's ISE program with nested dimensions and contexts as values. We then present the grammar formally; this includes a formal description of the context operators that coincide with the `version` and `vmod` operators of ISE. We also introduce a new operator, the `drill-down` operator, which allows us to strip away dimension tags in order to focus on the full versioned value of some particular dimension in the current context.

3.1 The Version Space

To discuss the version space we need to implement some consistent notational conventions. There are times when we look at version expressions in the abstract, with variable representations, and there are times when we look at actual version expressions with identifiers. In general, to distinguish between the two we italicize variable expressions and use a plain font style with no italics for actual expressions.

As we have stated before, we use version expressions to represent points in the version space. It is these version expressions that act as the tags in our production rules. At the ground level we have a dimension identifier and a value identifier as a pair separated by a colon (`:`). The identifiers are simple strings with no spaces and are usually descriptive. For example, we may have a dimension associated with language that has two possible values; `language:French` or `language:English`.

There is a further distinction when discussing variable expressions. In some cases we represent an expression with atomic variables, where there is a one-to-one correspondence between variable and possible identifier. In these cases we use lowercase letters, with or without subscripts, from the beginning of the alphabet to represent dimensions and lowercase letters, with or without subscripts, from the end of the alphabet

to represent values. For example, $d{:}\,v$ could be used as a representation of an expression with a single *dimension:value* pair. In this example d could represent the dimension `language` and v could represent the value `French`.

The version algebra also allows for nested dimensions grounded by a value. Thus for example, $d{:}\,e{:}\,v$ represents a version expression with dimension d that has value $e{:}\,v$, which itself represents a version expression with dimension e and value v. For instance, the expression `oblique:object:dog` can be represented by $d{:}\,e{:}\,v$.

In other cases we use meta-variables to represent portions of a version expression, where there is not necessarily a one-to-one correspondence with identifiers in an actual expression. For example, there are times when we need to denote a full, nested dimension without being explicit about how many identifiers exist in the label. For those occasions we use uppercase letters from the beginning of the alphabet. So, our example expression above, `oblique:object:dog`, can be equally denoted by $D{:}\,v$, where D denotes the entire dimension label, `oblique:object`.

Finally, we use uppercase letters from the end of the alphabet to represent full version expressions. Thus, `oblique:object:dog` can also be denoted by V or, $d{:}\,V$ where V represents version `object:dog`.

To summarize, there are four different variable types to consider when discussing points in the version space, each represented by different ranges of letters from the alphabet, uppercase or lowercase and with or without subscripts. These are summarized below.

Atomic dimensions: $d, e, f, d_i, e_i, f_i, \ldots$

Atomic values: $u, v, w, u_i, v_i, w_i, \ldots$

Nested dimensions: $D, E, F, D_i, E_i, F_i, \ldots$

Version expressions: $U, V, W, U_i, V_i, W_i, \ldots$

3.1.1 Properties of the Version Space

The version space has a join operation, denoted $+$, which allows us to join two versions together to make a new version, provided the two versions are compatible. So, the expression $d{:}\,v \;+\; e{:}\,f{:}\,w$ represents a version expression with two dimensions, d and $e{:}\,f$, with values v and w, respectively. For example, this could represent the version expression, `language:English+oblique:object:dog`.

The nesting of dimension labels in conjunction with the join operation allows a version to have a single unlabeled value, called the *base value*. Dimension labels distribute over $+$ as follows:

(1) $d_1{:}\langle v_1 \;+\; d_2{:}\,V_2 \;+\; \cdots\rangle \;=\; d_1{:}\,v_1 \;+\; d_1{:}\,d_2{:}\,V_2 \;+\; \cdots$,

where d_i are atomic dimension labels, v_1 is the base value and V_2 is a version expression. In (1), d_1 has a *full version value* of $\langle v_1 \;+\; d_2{:}\,V_2 \;+\; \cdots\rangle$ and a base value of v_1. Note that a dimension does not have to have a base value. For example, in the expression $d_1{:}\langle d_2{:}\,v_1 \;+\; d_3{:}\,V_2 \;+\; \cdots\rangle$, dimension d_1 has no base value although $d_1{:}\,d_2$ does[16].

We can also see here the two canonical forms that a version expression can take. On the left hand side of (1) we have the *factored canonical form* of the expression and on the right the *non-factored canonical form*. Both forms are important to the version space for different reasons. The factored form allows us to find the full version value of specific dimensions while the non-factored form allows us to find the base values.

[16] Namely, v_1.

In summary, the syntax for version expressions has the following form,

(2) $V = \epsilon \mid v \mid D{:}V \mid V + V$
$D = d \mid D{:}d$
$d = \text{dim\_id}$
$v = \text{val\_id}$

where ϵ is the empty or vanilla version. There are some restrictions on the terms of a version expression. A version expression may have only one base value per dimension. Furthermore, dimensions cannot have more than one value. That is, a non-factored version expression containing a subexpression of the form $D{:}v + D{:}w$ cannot exist so long as $v \neq w$. Thus, the expression `language:French+language:English` cannot exist in the version space. All dimensions that are not explicitly stated in a version expression are understood to have vanilla as their value.

The version space is the set of all possible version expressions, partially ordered by the *refinement relation*, denoted by $\leq$. For versions V and W, $V \leq W$ is read as V *refines* to W and means that W is a more specific version than V. Furthermore, we say $V \leq W$ *maximally*, or $V \leq_{\max} W$, if for any version $U \neq V$ where $U \leq W$, then V does not refine to U. The join operator is the least upper bound induced by refinement. Thus, $U + V$ is the least upper bound of U and V if and only if for all W, such that $U \leq W$ and $V \leq W$, then $U + V \leq W$.

Explicitly, refinement between version expressions in non-factored canonical form is defined as

(3) $D_1{:}v_1 + D_2{:}v_2 + \cdots \leq E_1{:}w_1 + E_2{:}w_2 + \cdots$

if and only if for each D_i either $v_i = \epsilon$ or $D_i{:}v_i = E_j{:}w_j$ for some j.

The vanilla version has no dimensions or values and refines to any version. The refinement relation is transitive and the join operation is idempotent, commutative, and associative, all summarized below:

(4) $D{:}\,\epsilon = \epsilon$

(5) $V + \epsilon = V$

(6) $V + V = V$

(7) $V + W = W + V$

(8) $U + \langle V + W \rangle = \langle U + V \rangle + W$

(9) $\epsilon \leq V$

(10) $V \leq V + W$

(11) If $V \leq U$ and $U \leq W$, then $V \leq W$

(12) If $V \leq V'$ and $W \leq W'$, then $V + W \leq V' + W'$

To make clear the objects we are discussing later in this chapter and beyond, we must introduce some notation on versions and dimensions.

Definition 3.1 Let $\boldsymbol{\mathcal{V}}$ be the version space and $\boldsymbol{\mathcal{D}}$ be the set of all possible dimensions. Furthermore, let $D \in \boldsymbol{\mathcal{D}}$ and let $V \in \boldsymbol{\mathcal{V}}$, then

i. $\mathcal{D}(V) = \{D | D{:}\, v \leq V \; and \; v \neq \epsilon\}$ is the set of all non-vanilla dimensions in version V,

ii. $\nu(V) = \begin{cases} v & if \; v \leq V \\ \epsilon & otherwise \end{cases}$ is the base value of version V,

iii. $\nu(D, V) = \begin{cases} v & if \; D{:}\, v \leq V \\ \epsilon & otherwise \end{cases}$ is the base value of dimension D in version V, and

iv. $\Gamma(D, V) = \begin{cases} U & if \; D{:}\, U \leq V \\ \epsilon & otherwise \end{cases}$ is the full version value of dimension D in version V, where V is in factored canonical form.

Example 3.2 Consider the version expression $V = d_1{:}\langle v_1 \ + \ d_2{:}v_2 \ + d_2{:}d_3{:}v_3\rangle$. Here, the set of non-vanilla dimensions is $\mathcal{D}(V) = \{d_1, d_1{:}d_2, d_1{:}d_2{:}d_3\}$, the base value of the version is $\nu(V) = \ \epsilon$, the base value of dimension d_1 is $\nu(d_1, V) = v_1$ and the base value of $d_1{:}d_2$ is $\nu(d_1{:}d_2, V) = v_2$, but the base value of dimension d_2 in V is $\nu(d_2, V) = \epsilon$. Furthermore, the full versioned value of dimension d_1 is $\Gamma(d_1, V) = \langle v_1 \ + \ d_2{:}v_2 \ + \ d_2{:}d_3{:}v_3\rangle$.

Note that although dimension d_2 has no base value in version V, it does have one in the full versioned value of dimension d_1. That is, $\nu\big(d_2, \Gamma(d_1, V)\big) = \nu(d_1{:}d_2, V) = v_2$.

3.2 Production Rules

Definition 3.3 *Intensional Context-Free Grammar* (ICFG) is a four-tuple, $G = (N, T, S, P)$, where N is a finite set of nonterminal symbols, T a finite set of terminal symbols, $S \in N$ is the start symbol, and P is a finite set of intensional production rules of the form,

(13) $A \underset{V}{\rightarrow} \gamma$

where A is a nonterminal, V is a version expression and γ is a string of terminals and nonterminals with or without associated context modifiers.

The production rules in ICFG offer two innovations; version tags and context modifiers. The version tags represent the possible worlds in which the production rules may be applied to form their extensions. The context modifiers allow the grammar to access other possible worlds from the actual world, which we call the *current context* and denote C.

If a version tag is empty then the version corresponds to the vanilla context and can be dropped from the rule. Tagged rules are used to determine the appropriate rule to apply at any given point in a derivation. Thus, when deriving nonterminal A we check the current context against the version tags of all the versions of the A rule and use the best-fit algorithm to select the most appropriate one.

3.2.1 The Best-Fit Algorithm

The current context exists as a point in the version space. The context tags associated with our production rules also represent points in the version space. Using the refinement relation we compare all the versions of a production rule to each other and to the current context and select the best-fit version of the rule. The best-fit version of a rule is a version that refines to the current context maximally. In the case where there is more than one maximally best-fit version then the choice proceeds nondeterministically.

To illustrate how the best-fit algorithm works, let us consider the following example in which we have defined five versions of the production rule for a nonterminal A.

(14) $A \rightarrow \gamma_1$
$A \xrightarrow[\langle \text{ln:en} \rangle]{} \gamma_2$
$A \xrightarrow[\langle \text{ln:fr} \rangle]{} \gamma_3$
$A \xrightarrow[\langle \text{ln:en+br:ie} \rangle]{} \gamma_4$
$A \xrightarrow[\langle \text{ln:en+br:ff} \rangle]{} \gamma_5$

Here, A is an arbitrary nonterminal of some grammar and each γ_i is some string of terminals and nonterminals. For the purposes of this example we have included real version expressions. This example could represent a choice of webpage content based on

a user's preferences as is done in IHTML[17]. Dimension `ln` is language, with a choice of French (`fr`) or English (`en`), dimension `br` stands for browser, either Internet Explorer (`ie`) or Firefox (`ff`), and `pl` is platform, which can be `pc` or `mac`.

Suppose that the current context is $C =$ `<ln:en+br:ie+pl:pc>` and that there is a call to the production rule for nonterminal A in a derivation. To determine the best-fit rule, we observe the relationship between the five context tags and the current context, as points in the version space, illustrated below with refinement relations denoted by a line.

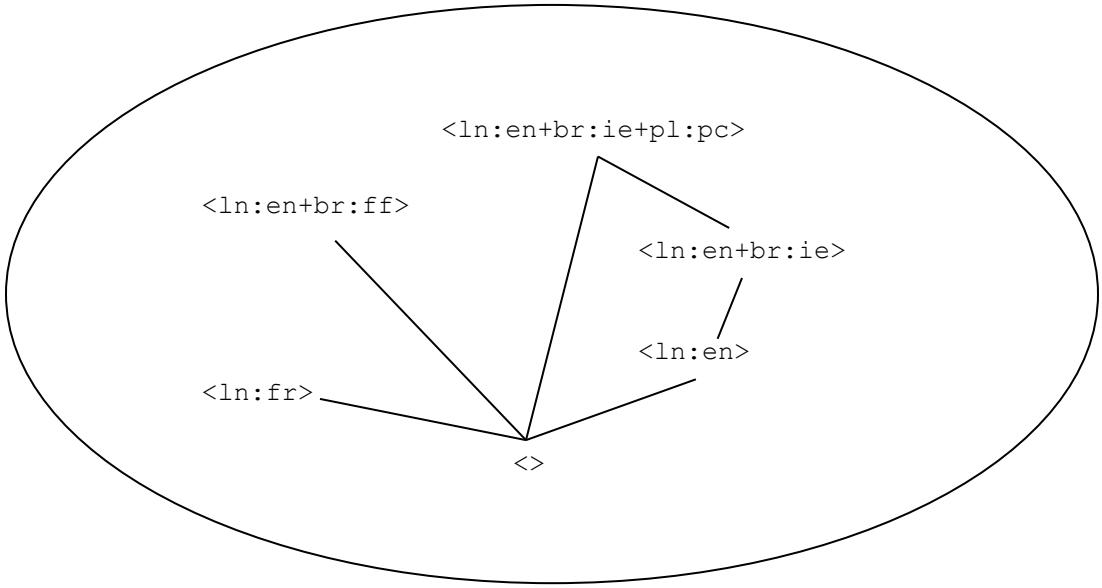


Figure 1 - Context space

[17] IHTML is discussed in Section 2.2.3.

Next, we remove any context points that do not refine to the current context.

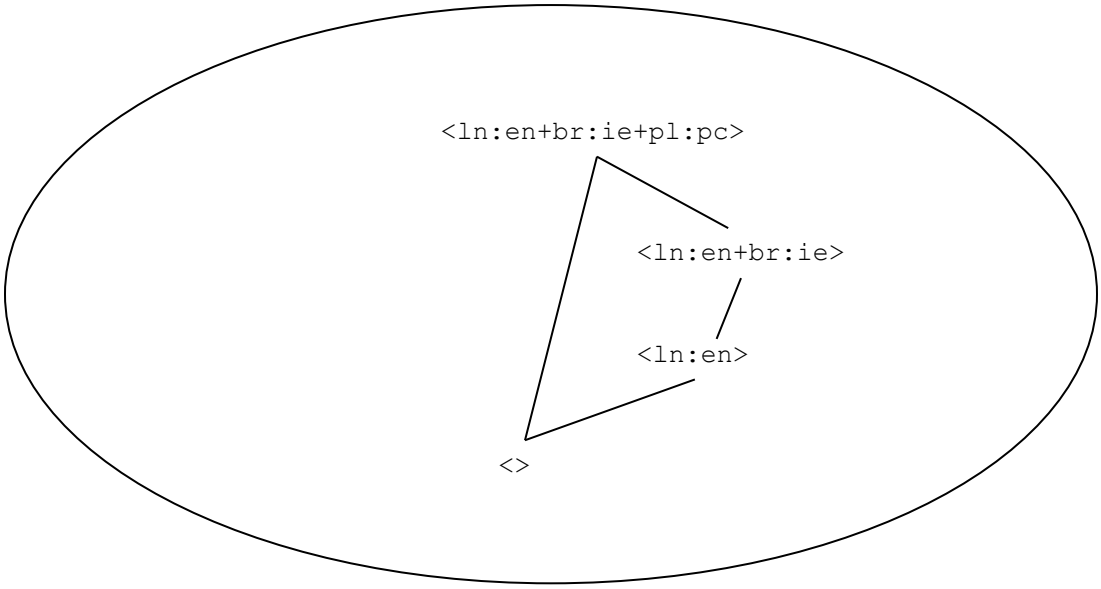


Figure 2 - Context space after refinement

Of the points that are left, if any, we choose the context points that refine to the current context maximally, which in this example is `<ln:en+br:ie>`.

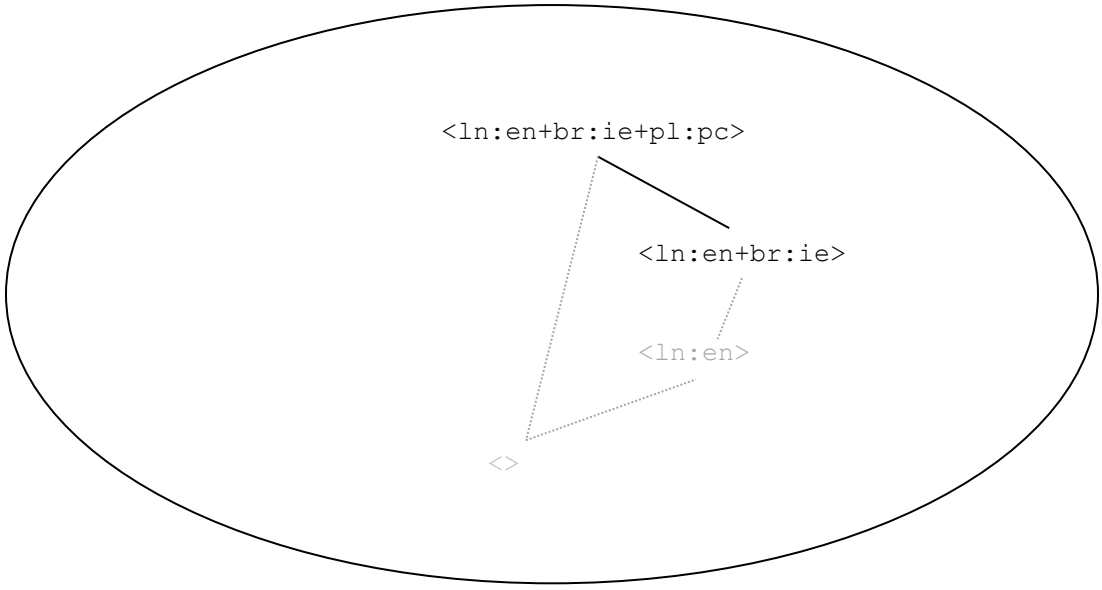


Figure 3 - Context space after maximal refinement

Thus, the derivation would proceed by replacing nonterminal A with string γ_4.

3.2.2 Context Modifiers

There are two types of context modifiers, *absolute* and *relative*, corresponding to the `version` and `vmod` identifiers of ISE, respectively. In ICFG we use angle brackets, ⟨ ⟩, to indicate an absolute context modifier and square brackets, [ ], to indicate a relative context modifier. Furthermore, there are two types of relative context modifiers, the traditional `vmod` operator and the `drill-down` operator. Thus, rules that have context modifiers in them are of the following three forms,

(15) $A \underset{V}{\rightarrow} \alpha B\langle W\rangle\beta$

(16) $A \underset{V}{\rightarrow} \alpha B[W]\beta$

(17) $A \underset{V}{\rightarrow} \alpha B[D]\beta$

where A and B are nonterminals, α and β are strings of terminals and nonterminals with or without context modifiers, V and W are version expressions and D is a single, ungrounded dimension (possibly nested) used for a `drill-down` operation.

The modifiers affect the context locally in a derivation. That is, the change applies to the subtree rooted at the nonterminal to the modifier's immediate left. In the case of rule (15), $\langle W\rangle$ is an absolute context modifier, this means that the current context becomes W before the application of the appropriate B rule under the new context.

In the case of rule (16), $[W]$ is one type of relative context modifier containing the dimension-value pairs to be modified in the current context. For every dimension $D \in \mathcal{D}(W)$, the base version value, $\nu(D, C)$, in the current context, C, is changed to the value $\nu(D, W)$. Recall that dimensions not explicitly stated in a version expression have ϵ as their value, thus if the current context does not have a value for the dimension to be

modified then $D\colon \nu(D, W)$ is added directly to the current context via the join operation. All other dimensions maintain their current values.

With rule (17) we have the second relative modifier, in which the current context becomes $\Gamma(D, C)$, the full version value of dimension D. In both cases of the relative context modifier it is used to change some part of the current context before the application of the appropriate B rule.

To illustrate the use of these modifiers, consider the following simple grammar fragment that does a translation from French to English[18].

(18) $S \rightarrow L\langle \text{ln: fr} + \text{subj: } \langle \text{chien} + \text{nm: sg} \rangle \rangle$
$L \xrightarrow[\langle \text{ln:fr} \rangle]{} T[\text{ln: en} + \text{subj: dog}]$
$T \xrightarrow[\langle \text{ln:en} \rangle]{} NP[\text{subj}]\ VP$
$NP \xrightarrow[\langle \text{dog} \rangle]{} the\ dog$

Suppose the initial current context for this grammar is vanilla, that is $C = \epsilon$.[19] The S rule uses the absolute context modifier, attached to the nonterminal L, to change the current context to `<ln:fr+subj:<chien+nm:sg>>`. The L rule then uses a relative modifier to change the `ln` dimension in the current context to `en` and the base value of the `subj` dimension to `dog`. Finally, the T rule uses the drill-down modifier to change the current context to `<dog+nm:sg>`, which ensures the selection of the `<dog>` version of the NP rule.

[18] This is a prelude to the kind of thing we will do in Chapters 6 and 7.
[19] For this example C could be anything to start as the current context is changed right away.

3.2.3 Derivation in ICFG

Definition 3.4 For an intensional context-free grammar, $G = (N, T, S, P)$, current context C, nonterminal A, and strings of terminals and nonterminals with or without context modifiers α, β, and γ, the *derives relation*, denoted $\underset{G,C}{\Rightarrow}$, is defined as follows,[20]

(19) $\alpha A \beta \underset{G,C}{\Rightarrow} \alpha\gamma\beta$ if $A \underset{V}{\rightarrow} \gamma$ is a production rule in G and $V \leq_{max} C$,

(20) $\alpha A\langle W\rangle \beta \underset{G,W}{\Rightarrow} \alpha\gamma\beta$ if $A \underset{V}{\rightarrow} \gamma$ is a production rule in G and $V \leq_{max} W$,

(21) $\alpha A[W]\beta \underset{G,C\circ W}{\Longrightarrow} \alpha\gamma\beta$ if $A \underset{V}{\rightarrow} \gamma$ is a production rule in G and $V \leq_{max} C\circ W$.

(22) $\alpha A[D]\beta \underset{G,\Gamma(D,C)}{\Longrightarrow} \alpha\gamma\beta$ if $A \underset{V}{\rightarrow} \gamma$ is a production rule in G and $V \leq_{max} \Gamma(D, C)$.

Some new notation has been introduced in rule (21) that should be explained. The notation $C\circ W$ denotes the context expression that results from expression C after modifying it by expression W.

To be clear, the context modifiers are not part of the grammar, meaning that they are not part of the derivation even though it may look like they are. These are just notational conventions for the grammar, or metasymbols, indicating the look of the current context when it has been modified. Once the context is modified that symbol is no longer necessary in the actual derivation. You can think of the pairs of nonterminals and context modifiers as intensional nonterminals, that is the nonterminal and the possible world that it is derived from.

Definition 3.5 The *extended derivation relation*, denoted $\underset{G,C}{\Rightarrow}^*$, as $\alpha \underset{G,C}{\Rightarrow}^* \gamma$ if and only if α derives γ, in grammar G under context C, in zero or more steps, where α and γ are strings of terminals and nonterminals with or without context modifiers.

---

[20] The G can be dropped when the grammar is understood.

You may have noticed that the context may change throughout a derivation upon encountering context modifiers. We denote this by changing the second symbol under the derives symbol, as shown in (20) to (22). The context returns to a previous state once the derivation rooted at a modified nonterminal is complete. With the extended derivation we only need to indicate the starting current context, with the understanding that it may have changed within the derivation being represented.

Definition 3.6 The *extensional language generated by G in context C*, denoted $L_C(G)$, is defined as $L_C(G) = \left\{w \in T^* \mid S \underset{G,C}{\Longrightarrow}^* w\right\}$.

Definition 3.7 The *intensional language generated by G*, denoted $L(G)$, is defined as the indexed set of context, extensional language pairs,

$$L(G) = \{\langle C, L_C(G)\rangle\}_{C \in \boldsymbol{\mathcal{V}}}$$

where $\boldsymbol{\mathcal{V}}$ is the version space.

Thus, the intensional language generated by an intensional grammar is actually an indexed set of extensional languages, indexed by the context in which the extension is generated.

3.2.4 An Example

Let us look at an example of a full grammar with multiversioned rules that makes use of a context modifier. We also look at a couple of derivations and see how the notation we introduced above works in practice.

Consider the grammar $G = (\{S, A, B\}, \{a, b\}, S, P)$ with the following set of production rules.

(23) $S \underset{\langle\ \rangle}{\longrightarrow} AB$

$A \underset{\langle \mathrm{d{:}1} \rangle}{\longrightarrow} a$

$$A \underset{\langle\ \rangle}{\longrightarrow} aA[\mathrm{d}]$$
$$B \underset{\langle \mathrm{d}{:}1\rangle}{\longrightarrow} b$$
$$B \underset{\langle\ \rangle}{\longrightarrow} bB[\mathrm{d}]$$

Here we have one version of an S rule and two versions each of an A rule and a B rule, each with a vanilla version. We have denoted vanilla by the empty version tag, $\langle\ \ \rangle$, but as you may recall we can drop empty version tags from our rules as they are still understood to be vanilla.

For each consistent current context of the form $C = \langle \underbrace{\mathrm{d}{:}\cdots\mathrm{d}}_{k\geq 1}{:}\, 1 + V\rangle$, where V is any version expression, the language generated by G in context C is $L_C(G) = \{a^k b^k\}$. For any other context C, the language is empty, or $L_C(G) = \emptyset$.

A derivation of the string $aaabbb$ is given as follows,

(24) $$S \underset{\langle \mathrm{d}{:}\mathrm{d}{:}\mathrm{d}{:}1\rangle}{\Longrightarrow} AB \underset{\langle \mathrm{d}{:}\mathrm{d}{:}\mathrm{d}{:}1\rangle}{\Longrightarrow} aA[d]B \underset{\langle \mathrm{d}{:}\mathrm{d}{:}1\rangle}{\Longrightarrow} aaA[d]B \underset{\langle \mathrm{d}{:}1\rangle}{\Longrightarrow} aaaB$$
$$\underset{\langle \mathrm{d}{:}\mathrm{d}{:}\mathrm{d}{:}1\rangle}{\Longrightarrow} aaabB[d] \underset{\langle \mathrm{d}{:}\mathrm{d}{:}1\rangle}{\Longrightarrow} aaabbB[d] \underset{\langle \mathrm{d}{:}1\rangle}{\Longrightarrow} aaabbb.$$

You can see that the grammar makes use of the drill-down modifier to strip off $n-1$ atomic dimension identifiers in order to produce n copies of a terminal, for each terminal in $\{a, b\}$. This is not exactly the slickest way to do this but it illustrates the use of a context modifier.

3.3 Conclusion

We have created a new formal grammar called Intensional Context-Free Grammar that uses tagged (intensional) rewrite rules to derive sets of strings under specific contexts. Consider again the example ICFG given in (23), in effect, this grammar generates all the strings of the form $a^n b^n$ for any $n \geq 1$, but it does so by generating infinitely many one

string extensional language sets, each derived under a different context. This is not the same as the context-free language $L(G) = \{a^n b^n | n \geq 1\}$, generated by the context-free grammar $G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow ab\})$. This begs the question, what is the relationship between context-free languages and intensional context-free languages? We tackle this question in the next chapter.

4 Is ICFG Context-Free?

We concluded the last chapter with a question; what is the relationship between ICFGs and CFGs? The question we really need to consider first is; is there a relationship between ICFG and CFG? Or, more to the point, can we justify the CFG in ICFG? Of course, the answer to the third question is yes. The answer to the second question is also yes, but not in the way you might think. So, we are back to the first question; what is the relationship between ICFG and CFG?

It turns out that we cannot compare CFG and ICFG directly. That is, ICFG does not really fit into Chomsky's hierarchy. The fact is context-free languages are sets of terminal strings whereas intensional context-free languages are indexed sets of languages which are themselves sets of terminal strings. So, we cannot say things like CFL $\subset$ ICFL. But, this is not all bad; we can compare context-free languages to individual extensional languages generated by an intensional context-free grammar. It is our goal in this chapter to show that the extensional languages are in fact context-free, justifying our use of the term context-free in intensional context-free grammar.

Consider the intensional grammar $G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ with the following set of production rules.

(1) $S \rightarrow ABC$
$A \underset{\langle \mathrm{d}:1 \rangle}{\longrightarrow} a$
$A \rightarrow aA[\mathrm{d}]$
$B \underset{\langle \mathrm{d}:1 \rangle}{\longrightarrow} b$
$B \rightarrow bB[\mathrm{d}]$
$C \underset{\langle \mathrm{d}:1 \rangle}{\longrightarrow} c$
$C \rightarrow cC[\mathrm{d}]$

In much the same way as grammar (23) of Chapter 3, this grammar generates the strings $a^n b^n c^n$ for any $n \geq 1$. This example is significant in that the language $\{a^n b^n c^n | n \geq 1\}$ is known to be non-context-free and although this grammar does not generate that language it does generate all the same strings as that language. Somehow ICFG is doing something more than any one context-free grammar can do. So, even though we cannot exactly fit our grammar into Chomsky's hierarchy, we do get an idea of its place in say a parallel universe (or some other possible world.)

4.1 CFLs Can be Derived by an ICFG

In this section we show that for every context-free language there exists an intensional context-free grammar that can generate the exact context-free language as an extensional language[21] in any context. This is done by simply adding the vanilla version tag to every production rule in the context-free grammar.

Theorem 4.1 Let $G = (N, T, S, P)$ be a context-free grammar. Then, there exists an ICFG $G' = (N, T, S, P')$, such that for any current context C, $L_C(G') = L(G)$.

Proof. Note that for any CFG G, the set of all production rules in G can be written with version tags without changing the language generated by G. We simply tag each rule with the vanilla version. That is, for each CFG $G = (N, T, S, P)$ we create ICFG $G' = (N, T, S, P')$ where for each rule $A \to \gamma \in P$ we create rule $A \underset{\langle\ \rangle}{\longrightarrow} \gamma \in P'$. Because all the rules are vanilla, no matter what the current context is the choice of production rule is exactly as it would be with the original context-free grammar. Thus, $S \underset{G}{\Rightarrow}^* w$ if and only if $S \underset{G',C}{\Longrightarrow}^* w$, for any C.■

[21] Definition 3.6.

With this in mind we can actually generate the language $\{a^n b^n | n \geq 1\}$ discussed in Chapter 3. For example, the CFG $G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow ab\})$ can be used to create ICFG $G' = \left(\{S\}, \{a, b\}, S, \left\{S \underset{\langle\ \rangle}{\longrightarrow} aSb, S \underset{\langle\ \rangle}{\longrightarrow} ab\right\}\right)$. Thus, the intensional language of G' is $L(G') = \{\langle C, L(G)\rangle\}_{C \in \boldsymbol{V}}$, for any C. Essentially, context-free languages are constants in intensional context-free languages when viewing intensional languages as functions from contexts to extensions.

We now proceed to a much more complex and interesting idea; that the extensional languages generated by any intensional grammar are in fact context-free. We present this result in three stages; first we devise the algorithm for generating a context-free grammar given an intensional context-free grammar and a current context. We then show that the algorithm terminates. Finally, we prove that the set of strings generated by the intensional grammar, under the given context, are the same as those generated by the context-free grammar constructed by the algorithm.

4.2 Generating Context-Free Grammars

The algorithm given below, Definition 4.2, takes an intensional context-free grammar G and some current context C and constructs the corresponding context-free grammar G_C. Informally, the algorithm uses the current context and constructs nonterminals and production rules for the context-free grammar based on the nonterminal-context pairs from the original intensional grammar. That is, we derive the extensional nonterminals explicitly from the intensional nonterminals for each context. The notation of the extensional nonterminals incorporates the context in which they are devised.

For each of the new extensional nonterminals we create the corresponding production rules headed by the given nonterminal. This is done by replacing each intensional nonterminal on the right hand side of the original production rule with the corresponding extensional nonterminal in the given context. During this process we may come across context modifiers. When we do, we use the modifier to create new contexts and new corresponding extensional nonterminals. We repeat the process for any new contexts that we encounter, stopping when there are no more new contexts.

Definition 4.2 Given an intensional context-free grammar $G = (N, T, S, P)$ and current context C, we derive the context-free grammar $G_C = (N_C, T, S.C, P_C)$ by the following steps:

1. Let there be two sets of contexts, for new contexts, $\boldsymbol{K}_{new}$, and old contexts, $\boldsymbol{K}_{old}$, respectively. We use these to keep track of any new contexts we encounter via the context modifiers. Let $\boldsymbol{K}_{new} = \{C\}$ and $\boldsymbol{K}_{old} = \emptyset$ to start.
2. For some context $K \in \boldsymbol{K}_{new}$ and for each $A \in N$ and production $A \underset{V}{\rightarrow} \gamma \in P$, such that $V \leq_{max} K$, we put $A.K \in N_C$ and $A.K \rightarrow \gamma^K \in P_C$. For each $X \in (N \cup T)$ in the string γ there is a corresponding X^K in the string γ^K, such that
    a. if $X = a \in T$, then let $X^K = a$ in the (corresponding position in the) string γ^K,
    b. if $X = B \in N$, then let $X^K = B.K$ in the string γ^K,
    c. if $X = B\langle W \rangle$ for some $B \in N$, then let $X^K = B.W$ in the string γ^K. Also, if W is not already in $\boldsymbol{K}_{new} \cup \boldsymbol{K}_{old}$, we add W to $\boldsymbol{K}_{new}$,
    d. if $X = B[W]$ for some $B \in N$, then let $X^K = B.(K \circ W)$ in the string γ^K. Also, if $K \circ W$ is not already in $\boldsymbol{K}_{new} \cup \boldsymbol{K}_{old}$, we add it to $\boldsymbol{K}_{new}$, and

e. if $X = B[D]$ for some $B \in N$, then let $X^K = B.\Gamma(D, K)$ in the string γ^K.

Also, if $\Gamma(D, K)$ is not already in $\boldsymbol{K}_{new} \cup \boldsymbol{K}_{old}$, we add it to $\boldsymbol{K}_{new}$.

3. Remove K from $\boldsymbol{K}_{new}$ and add it to $\boldsymbol{K}_{old}$.

4. If $\boldsymbol{K}_{new} = \emptyset$, stop. Otherwise, go back to step 2.

Before proving that the algorithm halts, let us consider an example to illustrate its use. We apply the algorithm to the ICFG given above in (1).

Example 4.3 Let $G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ be an intensional context-free grammar with production rules,

(2) $S \to ABC$
$A \underset{\langle \mathrm{d}:1\rangle}{\longrightarrow} a$
$A \to aA[\mathrm{d}]$
$B \underset{\langle \mathrm{d}:1\rangle}{\longrightarrow} b$
$B \to bB[\mathrm{d}]$
$C \underset{\langle \mathrm{d}:1\rangle}{\longrightarrow} c$
$C \to cC[\mathrm{d}]$

and let $C = \langle \mathrm{d}: \mathrm{d}: 1\rangle$. We construct the context-free grammar $G_C = (N_C, \{a, b, c\}, S.C, P_C)$ as follows:

1. Let $\boldsymbol{K}_{new} = \{\langle \mathrm{d}: \mathrm{d}: 1\rangle\}$ and $\boldsymbol{K}_{old} = \emptyset$.

2. Let $K = \langle \mathrm{d}: \mathrm{d}: 1\rangle$, then there are four production rules that apply maximally;

   $S \to ABC$, $A \to aA[\mathrm{d}]$, $B \to bB[\mathrm{d}]$, and $C \to cC[\mathrm{d}]$. We add $S.\langle \mathrm{d}: \mathrm{d}: 1\rangle$,

   $A.\langle \mathrm{d}: \mathrm{d}: 1\rangle$, $B.\langle \mathrm{d}: \mathrm{d}: 1\rangle$, and $C.\langle \mathrm{d}: \mathrm{d}: 1\rangle$ to N_C. Furthermore, we add

   $S.\langle \mathrm{d}: \mathrm{d}: 1\rangle \to A.\langle \mathrm{d}: \mathrm{d}: 1\rangle B.\langle \mathrm{d}: \mathrm{d}: 1\rangle C.\langle \mathrm{d}: \mathrm{d}: 1\rangle$,
   $A.\langle \mathrm{d}: \mathrm{d}: 1\rangle \to aA.\langle \mathrm{d}: 1\rangle$,
   $B.\langle \mathrm{d}: \mathrm{d}: 1\rangle \to bB.\langle \mathrm{d}: 1\rangle$,
   $C.\langle \mathrm{d}: \mathrm{d}: 1\rangle \to cC.\langle \mathrm{d}: 1\rangle$,

   to P_C and $\langle \mathrm{d}: 1\rangle$ to $\boldsymbol{K}_{new}$.

3. $\boldsymbol{K}_{new} = \{\langle \mathrm{d}: 1\rangle\}$ and $\boldsymbol{K}_{old} = \{\langle \mathrm{d}: \mathrm{d}: 1\rangle\}$.

4. $\boldsymbol{K}_{new} \neq \emptyset$ so go back to step 2.

2. Let $K = \langle \mathrm{d}: 1 \rangle$, then there are four production rules that apply maximally;

   $S \to ABC$, $A \underset{\langle \mathrm{d}:1 \rangle}{\longrightarrow} a$, $B \underset{\langle \mathrm{d}:1 \rangle}{\longrightarrow} b$, and $C \underset{\langle \mathrm{d}:1 \rangle}{\longrightarrow} c$. We add $S.\langle \mathrm{d}: 1 \rangle$, $A.\langle \mathrm{d}: 1 \rangle$, $B.\langle \mathrm{d}: 1 \rangle$, and $C.\langle \mathrm{d}: 1 \rangle$ to N_C. Furthermore, we add

   $S.\langle \mathrm{d}: 1 \rangle \to A.\langle \mathrm{d}: 1 \rangle B.\langle \mathrm{d}: 1 \rangle C.\langle \mathrm{d}: 1 \rangle$[22]
   $A.\langle \mathrm{d}: 1 \rangle \to a$,
   $B.\langle \mathrm{d}: 1 \rangle \to b$,
   $C.\langle \mathrm{d}: 1 \rangle \to c$,

   to P_C.

3. $\boldsymbol{K}_{new} = \emptyset$ and $\boldsymbol{K}_{old} = \{\langle \mathrm{d}: \mathrm{d}: 1 \rangle, \langle \mathrm{d}: 1 \rangle\}$.

4. Done.

The algorithm depends on one thing to stop, that there are finitely many new contexts generated from C by the context modifiers in G. Of course, we want to show that the algorithm does indeed have a stopping point no matter what the input grammar. To accomplish this we need to consider how many contexts (or version expressions) can be generated from C given some finite number of absolute, relative and drill-down modifiers.

4.3 There Are Finitely Many Derivable Contexts

Note that the number of modifiers in some given intensional context-free grammar is finite since there are finitely many production rules each with finitely many nonterminals on the right hand side of a production. Consider the nature of any one context, be it the starting context, C, or some modified context. Contexts are represented by version

[22] Note that this rule is useless as $S.\langle \mathrm{d}: 1 \rangle$ is not the start symbol and never appears on the right-hand side of a production rule.

expressions. The non-vanilla terms of a version expression, in its nonfactored canonical form, consist of either a single value (the base value) or a string of dimension labels, separated by colons, terminating in a value. That is, they are of the form v or $d_1 : d_2 : \cdots : d_n : v$. There are finitely many terms and for each term there are finitely many nested dimension labels. So, given some ICFG G and current context C, there is an upper bound on the number of dimension labels in any single term, i.e. there is a maximum depth to the nesting.

Consider the current context C and all the context modifiers in some grammar, G. For each let the depth of the version expression representing it be the maximum depth of all the terms in the expression. Then, let the depth of the grammar be the maximum depth of all these version expressions. We contend that the depth of any derivable context from C (via the algorithm) does not exceed the depth of the grammar and thus there are a finite number of derivable contexts.

Definition 4.4 Given a single-term version expression of the form $d_1 : d_2 : \cdots : d_n : v$, we define the *depth* of the expression to be the number of atomic dimension labels plus one. We denote this,

$$depth(d_1 : d_2 : \cdots : d_n : v) = n + 1.$$

If there are no dimension labels, i.e. the version is a base value, then the depth is one. That is, $depth(v) = 1$. Furthermore, the depth of the vanilla version is zero, or $depth(\epsilon) = 0$.

Definition 4.5 Given any version expression, V, we define $depth(V)$ to be the depth of the largest term of V in its nonfactored canonical form. Thus, if $V = d_1 : d_2 : \cdots : d_n : v + e_1 : e_2 : \cdots : e_m : u$ and $n \geq m$, then $depth(V) = n + 1$.

Definition 4.6 Let G be an intensional context-free grammar and C the current context. We define $depth(G)$ to be equal to the depth of the current context or context modifier with the maximum depth. That is, if G has context modifiers $M_1, M_2, \ldots, M_n$, and only these, then

$$depth(G) = max\{depth(C), depth(M_1), depth(M_2), \ldots, depth(M_n)\}.$$

Theorem 4.7 Given an ICFG G with depth N and current context C, then for any context modifier M in G, the depth of C modified by M is bound by N.

Proof. Let G be an ICFG such that $depth(G) = N$, and let the current context be C. First, consider the absolute modifier, $M = \langle W \rangle$ for some version expression W. Recall that the new current context in this case is just W and thus it must be that $depth(W) \leq depth(G) = N$ by Definition 4.6.

Now consider the relative modifier $M = [W]$, where W is a version expression. For every term $D{:}\,v$ in $C{\circ}W$, either $D{:}\,v$ is a term in W or it is a term in C. Thus, $depth(C{\circ}W) = max\{depth(C), depth(W)\} \leq depth(G) = N$.

Finally, if M is a drill-down modifier, then $M = [D]$ where D is a (possibly nested) dimension label and the depth of the modified context is actually strictly smaller than the depth of C since drill-downs remove dimension labels. That is, $depth\big(\Gamma(D, C)\big) < depth(\mathrm{C}) \leq N$.■

In Theorem 4.7 we have shown that the depth of any derived context is bound by the depth of the grammar. We now use this knowledge to show that there are finitely many of these derived contexts. We do this by showing that there are only finitely many possible version expressions bounded by the depth of the grammar generated from the dimensions and values of the grammar.

Theorem 4.8 Given an ICFG G with depth N and current context C, there are finitely many derivable contexts from C in G.

Proof. Let G be an ICFG with depth N and let C be the current context. From C and all the modifiers in G, let $n \geq 0$ be the total number of distinct atomic dimension labels and $m \geq 0$ the total number of distinct atomic values. Using N, n, and m we count the number of possible version expressions of depth N that can be generated from this collection of all atomic dimensions and values.

We start by counting the total number of possible individual terms. Recall that a term consists of zero or more atomic dimensions grounded by a base value and in this case bounded by the depth of the grammar. So, there are m possible terms of depth one (the base values), $n \cdot m$ possible terms of depth two, $n \cdot n \cdot m = n^2 \cdot m$ possible terms of depth three, and so on up to $n^{N-1} \cdot m$ possible terms of depth N. Thus, there are a total of $M = (1 + n + n^2 + \cdots + n^{N-1}) \cdot m$ possible terms.

Next, we count the total number of expressions bounded by depth N that can be built from these M terms. Recall that the version space is idempotent, meaning that for any version expression V (including one that is a single term), $V + V = V$. What this means is that given that there are only M possible terms, any version expression built from these terms will have at most M terms in it. If an expression had more than M terms than two of those terms would be the same and thus by idempotence one could be removed. That is, given a version expression V with M terms and given term W, then $V + W = V$.

So, there are M choose 0, denoted $\binom{M}{0} = \frac{M!}{(M-0)!0!} = 1$, version expressions with 0 terms (the vanilla version), $\binom{M}{1} = M$ version expressions with 1 term, and so on to $\binom{M}{M} = 1$ version expression with M terms. Thus, there are a total of $\binom{M}{0} + \binom{M}{1} + \cdots + \binom{M}{M} = 2^M$ possible version expressions from the atomic dimensions and values of this grammar. Clearly, this is a finite number and it includes all the derivable contexts from C in G since it includes all the contexts that can be generated by the atomics in G and any derivable context in G will only include dimensions and values from G.■

Theorem 4.8 guarantees that the algorithm in Definition 4.2 halts, which is our goal in this section. Given that we have an algorithm for generating context-free grammars from an intensional context-free grammar and that we know it halts, we need to now look at the languages that these grammars generate and show that they are the same.

4.4 The Extensions of an ICFG are Context-Free

Theorem 4.9 Let $G = (N, T, S, P)$ be an intensional context-free grammar and let C be some current context. Furthermore, let $G_C = (N_C, T, S.C, P_C)$ be the context-free grammar derived from G and C via the algorithm given in Definition 4.2. Then, the extensional language of grammar G at context C is equal to the language of grammar G_C. That is, $L_C(G) = L(G_C)$.

Proof. We proceed by first showing that $L_C(G) \subseteq L(G_C)$. Let $w \in L_C(G)$, thus, $S \underset{G,C}{\Rightarrow}^* w$ in some finite number of steps. Ultimately, we want to show that $S \underset{G_C}{\Rightarrow}^* w$ and thus $w \in L(G_C)$. But in fact, we show the stronger result that any string derived from a

nonterminal in G, under some context K, derivable from C via context modifiers, is derivable from the corresponding nonterminal in the grammar G_C. That is, if $A \underset{G,K}{\Rightarrow}^* w$, then $A.K \underset{G_C}{\Rightarrow}^* w$.

The proof is done inductively on the number of steps in the derivation $A \underset{G,K}{\Rightarrow}^* w$. Suppose that A derives w in one step. Then, $A \underset{G,K}{\Rightarrow} w$ and there exists a production $A \underset{V}{\rightarrow} w \in P$ such that $V \leq_{max} K$. By the algorithm, there exists a corresponding rule $A.K \rightarrow w \in P_C$ since K is C or derivable from C via the context modifiers of G. Of course, this implies that $A.K \underset{G_C}{\Rightarrow} w$.

Now, let it be true for derivations of up to k steps and suppose $A \underset{G,K}{\Rightarrow}^* w$ in $k+1$ steps. Then, there exists a production $A \underset{V}{\rightarrow} \gamma \in P$ where $V \leq_{max} K$ such that $A \underset{G,K}{\Rightarrow} \gamma$ and $\gamma \underset{G,K'}{\Rightarrow}^* w$ in k steps, where K' is K or K modified. Thus, by the algorithm there exists a production $A.K \rightarrow \gamma^K \in P_C$ and $A.K \underset{G_C}{\Rightarrow} \gamma^K$.

Let $\gamma = X_1 \cdots X_n$ where each $X_i \in (N \cup T)$ and let $w = w_1 \cdots w_n$ such that $X_i \underset{G,K_i}{\Rightarrow}^* w_i$ in k or less steps, where each context K_i is K or K modified. We have seen by the above algorithm that for each X_i in γ there is the corresponding X^{K_i} in γ^K that is in one of two forms,

1. if $X^{K_i} = a \in T$, then $a \underset{G_C}{\Rightarrow}^* a$,
2. if $X^{K_i} = B.K_i$, where $B \in N$, then $B.K_i \underset{G_C}{\Rightarrow}^* w_i$ by induction.

This implies that $\gamma^K \underset{G_C}{\Rightarrow}^* w$ and thus $A_K \underset{G_C}{\Rightarrow}^* w$. Therefore, $L_C(G) \subseteq L(G_C)$.

Similarly, to show that $L(G_C) \subseteq L_C(G)$ we show that if $A.K \underset{G_C}{\Rightarrow}^* w$, where $A.K \in N_C$, then $A \underset{G,K}{\Rightarrow}^* w$. Note that since $A.K \in N_C$, by the construction of G_C, K is derivable from C by some combination of context modifiers in G. Again we proceed by induction on the length of the derivation $A.K \underset{G_C}{\Rightarrow}^* w$. If the derivation is just one step, that is $A.K \underset{G_C}{\Rightarrow} w$, then there exists the production $A.K \to w \in P_C$. By the algorithm then there also exists the production $A \underset{V}{\to} w \in P$, such that $V \leq_{max} K$. This implies that $A \underset{G,K}{\Rightarrow} w$.

Assume that it is true for derivations of k steps or less. That is, if $A.K \underset{G_C}{\Rightarrow}^* w$ in k or less steps, then $A \underset{G,K}{\Rightarrow}^* w$. Now let $A.K \underset{G_C}{\Rightarrow}^* w$ in $k+1$ steps, then $A.K \underset{G_C}{\Rightarrow} \gamma^K$, such that $\gamma^K = X^{K_1} \cdots X^{K_n}$ where $X^{K_i} \in (N_C \cup T)$, and $\gamma^K \underset{G_C}{\Rightarrow}^* w$ in k steps. The first step in the derivation, $A.K \underset{G_C}{\Rightarrow} \gamma^K$, implies that there exists a rule $A.K \to \gamma^K \in P_C$. This rule was generated by the algorithm from the rule $A \underset{V}{\to} \gamma \in P$ where $V \leq_{max} K$ and thus $A \underset{G,K}{\Rightarrow} \gamma$.

Let $\gamma = X_1 \cdots X_n$, which is possible because there is a one-to-one correspondence from symbols in G to symbols in G_C in the algorithm. Thus, X_i in γ is used to generate X^{K_i} in γ^K. Furthermore, let $w = w_1 \cdots w_n$ such that $X^{K_i} \underset{G_C}{\Rightarrow}^* w_i$ in k or less steps, for each $i = 1, \ldots, n$. There are three possibilities for each X^{K_i};

1. if $X^{K_i} = a \in T$, then $X_i = a$ and $X_i \underset{G,K}{\Rightarrow}^* w_i = a$,
2. if $X^{K_i} = B.K$ for some $B.K \in N_C$, then $X_i = B \in N$ and $B \underset{G,K}{\Rightarrow}^* w_i$ by induction,
3. if $X^{K_i} = B.K'$ for some $B.K' \in N_C$ where $K' \neq K$, then by the algorithm a modifier was applied to K resulting in K'. The modifier was either absolute or relative, either way, $X_i \underset{G,K'}{\Rightarrow}^* w_i$ by the induction hypothesis.

Thus, $\gamma \underset{G,K_i}{\Longrightarrow}^{*} w$ where K_i is K or K' as described above and $A \underset{G,K}{\Longrightarrow}^{*} w$. Therefore, $L(G_C) \subseteq L_C(G)$ and we have shown that $L_C(G) = L(G_C)$.■

4.5 Conclusion

This result tells us that an intensional context-free grammar is an indexed set of context-free grammars, indexed by context. This goes beyond our original expectations in a very interesting way. Here we have an intensional grammar whose extensions are context-free grammars. Thus, any one intensional grammar encapsulates the power of many context-free grammars. This justifies our use of the term context-free in our intensional grammar, which is actually dependent on context. We have shown that although the intensions are dependent on context, the definition of intensionality, the extensions are not. In the next chapter we explore our other important mathematical result; the denotational semantics.

5 The Semantics of ICFG

An important aspect of proposing a new grammar is an analysis of the meaning, or *semantics*, of the grammar. Not to be confused with the semantics of a language, as in its use in linguistics, the semantics of a grammar in computer science is an analysis of the expected outcome of the grammar seen as a programming language. This can be done in a number of ways. Typically, both an operational and denotational semantic analysis is proposed with the goal of equating the two.

To define the operational semantics we need to define an implementation independent interpreter of the grammar. This interpreter would generate the set of sentences produced by the grammar which in turn would represent the meaning of the grammar. For rewrite grammars like ICFG the operational semantics is simply an application of the rewrite rules through derivation, starting from S, producing the language of the grammar. This we have already done in Chapter 3.

The denotational semantics defines the meaning of a grammar to be the least fixpoint of a transformation associated with the input-output relation of the grammar. A common way to achieve a denotational semantics of a context-free grammar is to create a transformation from the nonterminal symbols of the grammar to sets of strings with set concatenation and set union. For example, suppose we want to define the denotational semantics of the context-free grammar $G = (N, T, S, P)$, where P is given in (1).

(1) $S \to AB$
$A \to a \mid aA$
$B \to b \mid bB$

Our goal is to define a semantic map, $\Lambda^G: N \to \boldsymbol{\mathcal{P}}(T^*)$, from the nonterminals of the grammar to our semantic domain, sets of terminal strings. To do so we first define the

semantic equations that any maps of this type satisfy. Thus for any map, $\Lambda: N \to \boldsymbol{\mathcal{P}}(T^*)$, the following equations hold,

$$\Lambda[\![\gamma]\!] = \Lambda[\![X_1]\!] \cdot \cdots \cdot \Lambda[\![X_n]\!]$$

where $\gamma = X_1 \cdots X_n$ and each $X_i \in (N \cup T)$, and

$$\Lambda[\![X]\!] = \begin{cases} \{X\} & \text{if } X \in T, \\ \Lambda(X) & \text{if } X \in N. \end{cases}$$

Above we have introduced new notation common to denotational semantics. Here, each Λ not only represents a semantic map but also a denotation for a family of sets, one for each nonterminal in a given grammar. We use the notation $\Lambda(X)$ to index a particular set in the family, the one associated with the nonterminal represented by X. So, terminals are mapped to singleton sets containing the terminal and nonterminals are mapped to the set in the family of sets that the nonterminal indexes.

Given these equations we use the production rules of the grammar to define a function $\Phi^G: \big(N \to \boldsymbol{\mathcal{P}}(T^*)\big) \to \big(N \to \boldsymbol{\mathcal{P}}(T^*)\big)$, which is used to construct the particular map $\Lambda^G: N \to \boldsymbol{\mathcal{P}}(T^*)$. For example, for the first production rule of the grammar given in (1) we define $\Big(\Phi^G(\Lambda)\Big)[\![S]\!] = \Lambda[\![A]\!] \cdot \Lambda[\![B]\!]$. This rule says that the set indexed by $[\![S]\!]$ in a family of sets $\Phi^G(\Lambda)$ is equal to the set indexed by $[\![A]\!]$ in Λ, concatenated with the set indexed by $[\![B]\!]$ in Λ. Note that the dot, $(\cdot)$, denotes element-wise set concatenation, thus for any two sets X and Y, the set concatenation is defined as $X \cdot Y = \{xy | x \in X, y \in Y\}$.

When the grammar G is understood, we drop the superscript G in Φ^G. Furthermore, we shorthand the notation $\big(\Phi(\Lambda)\big)[\![S]\!]$ with $\Phi(\Lambda)[\![S]\!]$, which is less cluttered. So, the full definition of Φ for grammar (1) is given in (2).

(2) $\Phi(\Lambda)[\![S]\!] = \Lambda[\![A]\!] \cdot \Lambda[\![B]\!] = \Lambda(A) \cdot \Lambda(B)$

$\Phi(\Lambda)[\![A]\!] = \Lambda[\![a]\!] \cup \Lambda[\![a]\!] \cdot \Lambda[\![A]\!] = \{a\} \cup \{a\} \cdot \Lambda(A)$
$\Phi(\Lambda)[\![B]\!] = \Lambda[\![b]\!] \cup \Lambda[\![b]\!] \cdot \Lambda[\![B]\!] = \{b\} \cup \{b\} \cdot \Lambda(B)$

Finally, we define the map $\Lambda^G: N \to \boldsymbol{\mathcal{P}}(T^*)$ to be the least fixpoint of the map Φ, as follows,

(3) $\Lambda_0(S) = \emptyset$,
$\Lambda_{i+1}(S) = \Phi(\Lambda_i)[\![S]\!]$,
$\Lambda_0(A) = \emptyset$,
$\Lambda_{i+1}(A) = \Phi(\Lambda_i)[\![A]\!]$,
$\Lambda_0(B) = \emptyset$
$\Lambda_{i+1}(B) = \Phi(\Lambda_i)[\![B]\!]$.

where $\Lambda^G = \bigsqcup\{\Lambda_j\}_{j\geq 0}$, the least upper bound of the chain $\{\Lambda_j\}_{j\geq 0}$ defined in (3). In this way we start with a number of empty sets, one for each nonterminal in the grammar, and build them all up with successive applications of the function, with the goal being the complete set of strings that each nonterminal generates.

This semantics works because of three important properties. First, the powerset of terminal strings is a lattice with the empty set as the bottom and set union as least upper bound. Second, the map Φ is monotonic, meaning the sets never lose elements from one iteration to the next. Third, Φ is continuous, meaning least upper bounds are preserved, ensuring that a fixpoint exists for the map. That is, the sets are tending towards their least upper bounds.

Before moving on to our denotational semantics we need to discuss notation. As you can see from the example above, we incorporate three distinct domains in the semantics; grammars, sets and families of sets. In particular, we often need to distinguish between operations on sets and operations on families of sets. Order on sets is denoted as usual, with $\subset$, $\subseteq$, $\supset$, and $\supseteq$. The corresponding notation for families of sets is $\sqsubset$, $\sqsubseteq$, $\sqsupset$, and $\sqsupseteq$, where, for example, for any two families of sets, Λ_1 and Λ_2, then $\Lambda_1 \sqsubseteq \Lambda_2$

denotes that $\Lambda_1(X) \subseteq \Lambda_2(X)$ for each corresponding pair of indexed sets. Also, for the least upper bound of an ω-chain of sets, $\{X_i\}_{i\geq 0}$, we use $\bigvee\{X_i\}_{i\geq 0}$, whereas, for the least upper bound of a chain of families of sets, $\{\Lambda_i\}_{i\geq 0}$, we use $\coprod\{\Lambda_i\}_{i\geq 0}$, such that for any set X in the family of sets $\coprod\{\Lambda_i\}_{i\geq 0}$, we say that $\coprod\{\Lambda_i\}_{i\geq 0}(X) = \bigvee\{\Lambda_i(X)\}_{i\geq 0}$.

For the denotational semantics of ICFG we use a similar process as the one used above for CFGs, but of course we need to take into account context. So, instead of a denotation set for each nonterminal we have a number of denotations for each nonterminal in the form of context-nonterminal pairs. That is, for every context C and nonterminal A, we build the set $\Lambda^G(A)_C$ of strings generated by A in context C. Furthermore, with our notation we have to account for nonterminals with associated modifiers. This is done at the ground level as shown in the following definition.

Definition 5.1 Let $G = (N, T, P, S)$ be an ICFG and $\boldsymbol{\mathcal{V}}$ the version space. Then, for any map $\Lambda: N \to \boldsymbol{\mathcal{P}}(T^*)$, $C \in \boldsymbol{\mathcal{V}}$, and string $\gamma = X_1 \cdots X_n$, where each X_i is a terminal or nonterminal, possibly modified, the following hold;

$$\Lambda[\![\gamma]\!]_C = \Lambda[\![X_1]\!]_C \cdot \cdots \cdot \Lambda[\![X_n]\!]_C$$

and,

$$\Lambda[\![X]\!]_C = \begin{cases} \{a\} & \text{if } X = a, \\ \Lambda(A)_C & \text{if } X = A, \\ \Lambda(A)_W & \text{if } X = A\langle W\rangle, \\ \Lambda(A)_{C\circ W} & \text{if } X = A[W], \\ \Lambda(A)_{\Gamma(D,C)} & \text{if } X = A[D]. \end{cases}$$

As with context-free denotations, we use the production rules to build our sets using the map Φ on indexed families of sets. The difference now is that we again have to account for context by only taking the union of those sets who refine to the current context maximally.

Definition 5.2 Let $G = (N, T, P, S)$ be an ICFG and $\boldsymbol{\mathcal{V}}$ the version space. For each context $C \in \boldsymbol{\mathcal{V}}$ and each nonterminal $A \in N$, such that $A \underset{V_1}{\rightarrow} \gamma_1, \cdots, A \underset{V_m}{\rightarrow} \gamma_m \in P$ and $V_i \leq_{max} C$ for each $i = 1, \ldots, m$, we define the function $\Phi^G : \big(N \rightarrow \boldsymbol{\mathcal{P}}(T^*)\big) \rightarrow \big(N \rightarrow \boldsymbol{\mathcal{P}}(T^*)\big)$ by,

$$\Phi^G(\Lambda)[\![A]\!]_C = \Lambda[\![\gamma_1]\!]_C \cup \cdots \cup \Lambda[\![\gamma_m]\!]_C.$$

Thus, nonterminals are mapped to sets of terminal strings, derivable in the given context, by taking the union of the various maximal versions of the nonterminal.

Before we prove that our map is monotonic and continuous we need to look at two results from ordered set theory. The first result is that the powerset of any set is a lattice, with intersection and union as meet and join, respectively.

Theorem 5.3 [45] For any set X, the ordered set $\langle \boldsymbol{\mathcal{P}}(X); \subseteq \rangle$, where $\boldsymbol{\mathcal{P}}(X) = \{S \subseteq X\}$, is a complete lattice in which *join* and *meet* are as follows:

$$\bigvee\{A_i\}_{i \in I} = \bigcup\{A_i\}_{i \in I},$$

$$\bigwedge\{A_i\}_{i \in I} = \bigcap\{A_i\}_{i \in I}.$$

We also want to show that the concatenation of the least upper bounds of some finite number of ω-chains is equal to the least upper bound of the ω-chain of the concatenations. That is, concatenation is continuous.

Lemma 5.4 Consider the n ω-chains $\{X_{1,j}\}_{j \geq 0}, \ldots, \{X_{n,j}\}_{j \geq 0}$ where each $X_{i,j} \in \boldsymbol{\mathcal{P}}(T^*)$, then

$$\bigvee\{X_{1,j}\}_{j \geq 0} \cdot \cdots \cdot \bigvee\{X_{n,j}\}_{j \geq 0} = \bigvee\{X_{1,j} \cdot \cdots \cdot X_{n,j}\}_{j \geq 0}.$$

Proof. Consider the n ω-chains $\{X_{1,j}\}_{j \geq 0}, \ldots, \{X_{n,j}\}_{j \geq 0}$ where each $X_{i,j} \in \boldsymbol{\mathcal{P}}(T^*)$. Let $x \in \bigvee\{X_{1,j}\}_{j \geq 0} \cdot \cdots \cdot \bigvee\{X_{n,j}\}_{j \geq 0}$, thus $x = x_1 \cdots x_n$ where each $x_i \in \bigvee\{X_{i,j}\}_{j \geq 0}$ for $i = 1, \ldots, n$. So, $x_i \in X_{i,j_i}$ for some $j_i \geq 0$. That is, if x_i is in the least upper bound of a

chain then it is in some set in the chain since the least upper bound is defined by set union. Furthermore, for some x_k where $k = 1, \dots, n$, $x_k \in X_{k,j_k}$ and $j_k \geq j_i$ for all for $i = 1, \dots, n$. This implies that $x_1 \cdots x_n \in X_{1,j_k} \cdot \cdots \cdot X_{n,j_k}$ and thus an element of $\bigvee\{X_{1,j} \cdot \cdots \cdot X_{n,j}\}_{j\geq 0}$. Therefore, $\bigvee\{X_{1,j}\}_{j\geq 0} \cdot \cdots \cdot \bigvee\{X_{n,j}\}_{j\geq 0} \subseteq \bigvee\{X_{1,j} \cdot \cdots \cdot X_{n,j}\}_{j\geq 0}$.

To show that $\bigvee\{X_{1,j} \cdot \cdots \cdot X_{n,j}\}_{j\geq 0} \subseteq \bigvee\{X_{1,j}\}_{j\geq 0} \cdot \cdots \cdot \bigvee\{X_{n,j}\}_{j\geq 0}$ we need only note that $\bigvee\{X_{1,j}\}_{j\geq 0} \cdot \cdots \cdot \bigvee\{X_{n,j}\}_{j\geq 0}$ is an upper bound for the chain $\{X_{1,j} \cdot \cdots \cdot X_{n,j}\}_{j\geq 0}$ and that $\bigvee\{X_{1,j} \cdot \cdots \cdot X_{n,j}\}_{j\geq 0}$ is the least upper bound.■

Now we show that our function is monotonic, meaning that it is order preserving.

Theorem 5.5 The map Φ defined in 5.2 is monotonic. That is, for every pair Λ_1 and Λ_2, where $\Lambda_1 \sqsubseteq \Lambda_2$, then $\Phi(\Lambda_1) \sqsubseteq \Phi(\Lambda_2)$.

Proof. Let $\Lambda_1 \sqsubseteq \Lambda_2$ and consider any nonterminal $A \in N$ and context $C \in \boldsymbol{\mathcal{V}}$. Recall that $\Lambda_1 \sqsubseteq \Lambda_2$ means that $\Lambda_1[\![A]\!]_C \subseteq \Lambda_2[\![A]\!]_C$ for all $A \in N$ and $C \in \boldsymbol{\mathcal{V}}$. Thus, we want to show that $\Phi(\Lambda_1)[\![A]\!]_C \subseteq \Phi(\Lambda_2)[\![A]\!]_C$. Let $w \in \Phi(\Lambda_1)[\![A]\!]_C$, then there exists a $A \underset{V}{\rightarrow} \gamma \in P$ where $\gamma = X_1 \cdots X_n$ and $V \leq_{max} C$ such that $w \in \Lambda_1[\![X_1]\!]_C \cdot \cdots \cdot \Lambda_1[\![X_n]\!]_C$.

By Definition 5.1, for each $k = 1, \dots, n$ if X_k is a terminal, then $\Lambda_1[\![X_k]\!]_C = \Lambda_2[\![X_k]\!]_C$. Otherwise, X_k is a nonterminal or nonterminal with a modifier. In either case, $\Lambda_1[\![X_k]\!]_C \subseteq \Lambda_2[\![X_k]\!]_C$ since $\Lambda_1 \sqsubseteq \Lambda_2$. Therefore,

$$w \in \Lambda_1[\![X_1]\!]_C \cdot \cdots \cdot \Lambda_1[\![X_n]\!]_C \subseteq \Lambda_2[\![X_1]\!]_C \cdot \cdots \cdot \Lambda_2[\![X_n]\!]_C \subseteq \Phi(\Lambda_2)[\![A]\!]_C$$

and thus $\Phi(\Lambda_1)[\![A]\!]_C \subseteq \Phi(\Lambda_2)[\![A]\!]_C$.■

To show that our semantics has a fixpoint we need our map to be both monotonic and continuous. We showed that our map is monotonic in Theorem 5.4, thus ensuring that our partial maps are tending toward something. The following proof on continuity is

a step towards showing that that something is the least upper bound of the chain of mappings.

Theorem 5.6 The map $\Phi: \left(N \to \boldsymbol{\mathcal{P}}(T^*)\right) \to \left(N \to \boldsymbol{\mathcal{P}}(T^*)\right)$ defined in 5.2 is continuous. That is,

$$\Phi\left(\coprod\{\Lambda_j\}_{j\geq 0}\right) = \coprod\{\Phi(\Lambda_j)\}_{j\geq 0}.$$

Proof. Let $A \in N$ and $C \in \boldsymbol{\mathcal{V}}$, then for $A \underset{V_1}{\to} \gamma_1, \cdots, A \underset{V_m}{\to} \gamma_m \in P$ where each $V_i \leq_{max} C$ and $\gamma_i = X_{i,1} \cdots X_{i,n_i}$ for $i = 1, \ldots, m$,

$\Phi\left(\coprod\{\Lambda_j\}_{j\geq 0}\right)[\![A]\!]_C = \cup_{i=1}^{m}\left(\coprod\{\Lambda_j\}_{j\geq 0}[\![\gamma_i]\!]_C\right)$ (by definition 5.2)

$= \cup_{i=1}^{m}\left(\coprod\{\Lambda_j\}_{j\geq 0}[\![X_{i,1}]\!]_C \cdot \cdots \cdot \coprod\{\Lambda_j\}_{j\geq 0}[\![X_{i,n_i}]\!]_C\right)$ (by definition 5.1)

$= \cup_{i=1}^{m}\left(\vee\left\{\Lambda_j[\![X_{i,1}]\!]_C\right\}_{j\geq 0} \cdot \cdots \cdot \vee\left\{\Lambda_j[\![X_{i,n_i}]\!]_C\right\}_{j\geq 0}\right)$ (by definition of $\coprod$)

$= \cup_{i=1}^{m}\left(\vee\left\{\Lambda_j[\![X_{i,1}]\!]_C \cdot \cdots \cdot \Lambda_j[\![X_{i,n_i}]\!]_C\right\}_{j\geq 0}\right)$ (by Lemma 5.4)

$= \cup_{i=1}^{m}\left(\vee\{\Lambda_j[\![\gamma_i]\!]_C\}_{j\geq 0}\right)$ (by definition 5.1)

$= \vee\{\cup_{i=1}^{m}\Lambda_j[\![\gamma_i]\!]_C\}_{j\geq 0}$ (by Theorem 5.3)

$= \vee\{\Phi(\Lambda_j)[\![A]\!]_C\}_{j\geq 0}$ (by definition 5.2)

$= \coprod\{\Phi(\Lambda_j)\}_{j\geq 0}[\![A]\!]_C$ (by definition of $\coprod$)

Therefore, $\Phi\left(\coprod\{\Lambda_j\}_{j\geq 0}\right) = \coprod\{\Phi(\Lambda_j)\}_{j\geq 0}$.■

We now borrow a result from order theory which states that if you have a monotonic, continuous self-map on a CPO, then that map has a least fixpoint which is equal to the least upper bound of the chain resulting from iterative applications of that map.

Theorem 5.7 (CPO Fixpoint Theorem 1) [45] Let P be a CPO, let Φ be an order-preserving self-map on P and define $\Lambda^G = \bigsqcup\{\Lambda_j\}_{j\geq 0}$ where $\Lambda_0 = \emptyset$ and $\Lambda_{j+1} = \Phi(\Lambda_j)$, for $j \geq 0$.

I. If Λ^G is a fixpoint of Φ, then Λ^G is the least fixpoint of Φ.

II. If Φ is continuous, then Λ^G is a fixpoint of Φ.

Given this result and those proven above we can show that our map has a least fixpoint for the set representing each nonterminal, which is defined to be the least upper bound of the chain of maps starting with the empty set.

Theorem 5.8 Given an intensional context-free grammar $G = (N, T, P, S)$, let $\Phi(\Lambda)[\![A]\!]_C$ be the map defined in 5.2, such that $\Lambda_0[\![A]\!]_C = \emptyset$ and $\Lambda_{j+1}[\![A]\!]_C = \Phi(\Lambda_j)[\![A]\!]_C$ for each $A \in N$ and $C \in \boldsymbol{\mathcal{V}}$. Furthermore, let Λ^G be the least fixed point of Φ, then $\Lambda^G[\![A]\!]_C = \bigvee\{\Lambda_j[\![A]\!]_C\}_{j\geq 0}$.

Proof. By Theorem 5.3, $\boldsymbol{\mathcal{P}}(T^*)$ is a CPO. By Theorem 5.5, Φ is monotonic and by Theorem 5.6, Φ is continuous. Therefore, by Theorem 5.7, Λ^G is the least fixpoint of Φ and $\Lambda^G[\![A]\!]_C = \bigvee\{\Lambda_j[\![A]\!]_C\}_{j\geq 0}$ for each $A \in N$ and $C \in \boldsymbol{\mathcal{V}}$. ■

So, for every nonterminal in a given ICFG and every possible context, the least fixpoint of the chain of sets representing the meaning of that nonterminal in the given context is the set of all the strings derivable from that nonterminal in that context. Furthermore, we can define the intensional denotation of each nonterminal A as the indexed set of context-least fixpoint pairs, as in $[\![A]\!] = \{\langle C, \Lambda^G[\![A]\!]_C\rangle\}_{C\in\boldsymbol{\mathcal{V}}}$. In particular, if we construct the denotation of the start nonterminal in this way, we should produce the language of the grammar.

5.1 The Denotational Semantics Equals the Operational Semantics

Definition 5.9 Let the denotation of an intensional context-free grammar $G = (N, T, S, P)$ be defined as

$$[\![G]\!] = \{\langle C, \Lambda^G [\![S]\!]_C \rangle\}_{C \in \boldsymbol{\mathcal{V}}},$$

the indexed set of context-stringset pairs where each stringset is the least fixpoint of the chain of stringsets denoting the start nonterminal in the given context.

All that remains is for us to show that the language of the intensional context-free grammar $G = (N, T, S, P)$, is equal to the denotation of the grammar.

Theorem 5.10 Let $G = (N, T, S, P)$ be an intensional context-free grammar. Let Φ: $\big(N \to \boldsymbol{\mathcal{P}}(T^*)\big) \to \big(N \to \boldsymbol{\mathcal{P}}(T^*)\big)$ be defined as in Definition 5.2. For each $A \in N$ and $C \in \boldsymbol{\mathcal{V}}$, let $\Lambda_0 [\![A]\!]_C = \emptyset$ and $\Lambda_{j+1} [\![A]\!]_C = \Phi(\Lambda_j) [\![A]\!]_C$ for $j \geq 1$. Let Λ^G be the least fixpoint of Φ as defined in Theorem 5.8. Then, the denotation of G is equal to the language of G. That is,

$$[\![G]\!] = L(G).$$

Proof. We want to show that $[\![G]\!] = \{\langle C, \Lambda^G [\![S]\!]_C \rangle\}_{C \in \boldsymbol{\mathcal{V}}} = \{\langle C, L_C(G) \rangle\}_{C \in \boldsymbol{\mathcal{V}}} = L(G)$. More specifically, we want to show that for any $C \in \boldsymbol{\mathcal{V}}$, $\Lambda^G [\![S]\!]_C = L_C(G)$ or $\bigvee \{\Lambda_j [\![S]\!]_C\}_{j \geq 0} = \left\{ w \in T^* \mid S \underset{G,C}{\Rightarrow}{}^* w \right\}$. We will in fact show the stronger result that for any $C \in \boldsymbol{\mathcal{V}}$ and $A \in N$, it is true that $\bigvee \{\Lambda_j [\![A]\!]_C\}_{j \geq 0} = \left\{ w \in T^* \mid A \underset{G,C}{\Rightarrow}{}^* w \right\}$ which implies the general result we are after.

First, we show that $\bigvee \{\Lambda_j [\![A]\!]_C\}_{j \geq 0} \subseteq \left\{ w \in T^* \mid A \underset{G,C}{\Rightarrow}{}^* w \right\}$. Let $w \in \bigvee \{\Lambda_j [\![A]\!]_C\}_{j \geq 0}$, thus $w \in \Lambda_k [\![A]\!]_C$ for some $k \geq 1$. We use induction on k to show that $A \underset{G,C}{\Rightarrow}{}^* w$. Suppose $k = 1$, then $w \in \Lambda_1 [\![A]\!]_C = \Phi(\Lambda_0) [\![A]\!]_C$ and hence it must be the case that there exists a

rule $A \underset{V}{\rightarrow} w \in P$ where $V \leq_{max} C$ because $\Phi(\Lambda_0)$ applied to any nonterminal is the empty set. But, by the definition of derivation in ICFG, this implies that $A \underset{G,C}{\Rightarrow} w$ in one step.

Now, suppose that $w \in \Lambda_k[\![A]\!]_C$ implies that $A \underset{G,C}{\Rightarrow}^* w$ for $k \geq 1$ and let $w \in \Lambda_{k+1}[\![A]\!]_C = \Phi(\Lambda_k)[\![A]\!]_C$. This implies that $w \in \Lambda_k[\![X_1]\!]_C \cdot \cdots \cdot \Lambda_k[\![X_n]\!]_C$ for some $A \underset{V}{\rightarrow} X_1 \cdots X_n \in P$ such that $V \leq_{max} C$. Let $w = w_1 \cdots w_n$ such that each $w_i \in \Lambda_k[\![X_i]\!]_C$. If $X_i = b \in T$, then $X_i \underset{G,C}{\Rightarrow} w_i$ in one step. If $X_i = B \in N$, then by induction $X_i \underset{G,C}{\Rightarrow}^* w_i$. If $X_i = B\langle W \rangle$, where $B \in N$ and W is a modifier, then $\Lambda_k[\![X_i]\!]_C = \Lambda_k(B)_W$ by definition 5.1. But $\Lambda_k(B)_W$ also equals $\Lambda_k[\![B]\!]_W$ by the same definition and thus $B \underset{G,W}{\Rightarrow}^* w_i$ by induction. Similarly, if $X_i = B[W]$, then $B \underset{G,C \circ W}{\Longrightarrow}^* w_i$ and if $X_i = B[D]$, then $B \underset{G,\Gamma(D,C)}{\Longrightarrow}^* w_i$, by induction. Therefore, $A \underset{G,C}{\Rightarrow}^* w$ and thus $\bigvee\{\Lambda_j[\![A]\!]_C\}_{j \geq 0} \subseteq \{w \in T^* \mid A \underset{G,C}{\Rightarrow}^* w\}$.

Next, we want to show that $\{w \in T^* \mid A \underset{G,C}{\Rightarrow}^* w\} \subseteq \Lambda^G[\![A]\!]_C$, for all A and C. Here we want to use induction to show that whenever $A \underset{G,C}{\Rightarrow}^* w$ then $w \in \Lambda^G[\![A]\!]_C$, but this time we do so on the length of the derivation. Suppose that $A \underset{G,C}{\Rightarrow} w$ in one step. Then, there exists a rule $A \underset{V}{\rightarrow} w \in P$ such that $V \leq_{max} C$ by the definition of derivation in ICFG. Thus $\{w\} \subseteq \Phi(\Lambda^G)[\![A]\!]_C = \Lambda^G[\![A]\!]_C$ since Λ^G is the least fixpoint of Φ.

For the sake of induction, assume that for any A and C, if $A \underset{G,C}{\Rightarrow}^* w$ in $k \geq 1$ or less steps then $w \in \Lambda^G[\![A]\!]_C$. Now, let $A \underset{G,C}{\Rightarrow}^* w$ in $k+1$ steps. Thus, there exists a rule $A \underset{V}{\rightarrow} \gamma \in P$ with $V \leq_{max} C$ such that $A \underset{G,C}{\Rightarrow} \gamma$ in one step and $\gamma \underset{G,C'}{\Longrightarrow}^* w$ in k steps, where

context C' is either C or C modified. Let $\gamma = X_1 \cdots X_n$, where for each X_i either $X_i = Y_i \in (N \cup T)$, or $X_i = Y_i\langle M\rangle$ or $X_i = Y_i[M]$ such that $Y_i \in N$. Furthermore, let $w = w_1 \cdots w_n$ such that $Y_i \underset{G,C_i}{\Longrightarrow}^* w_i$ in k or less steps where context C_i is either C or C modified, for each $i = 1, \ldots, n$. But, since Λ^G is the least fixpoint of Φ then

$$\Lambda^G[\![A]\!]_C = \Phi(\Lambda^G)[\![A]\!]_C \supseteq \Lambda^G[\![Y_1]\!]_{C_1} \cdot \cdots \cdot \Lambda^G[\![Y_n]\!]_{C_n}$$

and each $w_i \in \Lambda^G[\![Y_i]\!]_{C_i}$ by induction. Thus $w \in \Lambda^G[\![Y_1]\!]_{C_1} \cdot \cdots \cdot \Lambda^G[\![Y_n]\!]_{C_n} \subseteq \Lambda^G[\![A]\!]_C$ and therefore $\left\{w \in T^* \mid A \underset{G,C}{\Longrightarrow}^* w\right\} \subseteq \Lambda^G[\![A]\!]_C$.■

By defining our denotations using the production rules we have closely related the construction of the denotations to the corresponding derivation steps. This leads us to a very nice denotational semantics that fits well with our operational semantics, as is evident by the proof above. Before concluding this chapter we look at an example in which we construct the denotations for the nonterminals of an ICFG, to better understand the process we have developed here.

Example 5.11 Consider the intensional grammar G defined in (4).

(4) $S \to NP[\text{subj}]\ VP$,
$NP \to D\ N$,
$NP \xrightarrow[\text{<def:no+num:pl>}]{} N$,
$VP \to V\ NP[\text{obj}]$,
$D \xrightarrow[\text{<def:yes>}]{} the$,
$D \xrightarrow[\text{<def:no+num:sg>}]{} a$,
$N \xrightarrow[\text{<num:sg>}]{} dog \mid cat$,
$N \xrightarrow[\text{<num:pl>}]{} dogs \mid cats$,
$V \to chased$.

Defining Φ as above we get the equations in (5),

(5) $\Phi(\Lambda)[\![S]\!]_C = \Lambda[\![NP]\!]_{\Gamma(\text{subj},C)} \cdot \Lambda[\![VP]\!]_C$

$$\Phi(\Lambda)[\![NP]\!]_C = \begin{cases} \Lambda[\![N]\!]_C & \text{if } \langle \text{def: no} + \text{num: pl} \rangle \leq C, \\ \Lambda[\![D]\!]_C \cdot \Lambda[\![N]\!]_C & \text{otherwise.} \end{cases}$$

$$\Phi(\Lambda)[\![VP]\!]_C = \Lambda[\![V]\!]_C \cdot \Lambda[\![NP]\!]_{\Gamma(\text{obj},C)}$$

$$\Phi(\Lambda)[\![D]\!]_C = \begin{cases} \{the\} & \text{if } \langle \text{def: yes} \rangle \leq C, \\ \{a\} & \text{if } \langle \text{def: no} + \text{num: sg} \rangle \leq C, \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\Phi(\Lambda)[\![N]\!]_C = \begin{cases} \{dog, cat\} & \text{if } \langle \text{num: sg} \rangle \leq C, \\ \{dogs, cats\} & \text{if } \langle \text{num: pl} \rangle \leq C, \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\Phi(\Lambda)[\![V]\!]_C = \{chased\}$$

For the sake of denotation construction via the least fixpoint Λ^G, we let

$$\Lambda_0[\![S]\!]_C = \Lambda_0[\![NP]\!]_C = \Lambda_0[\![VP]\!]_C = \Lambda_0[\![D]\!]_C = \Lambda_0[\![N]\!]_C = \Lambda_0[\![V]\!]_C = \emptyset$$

for all $C \in \boldsymbol{\mathcal{V}}$. Then,

$$\Lambda_1[\![S]\!]_C = \Phi(\Lambda_0)[\![S]\!]_C = \Lambda_0[\![NP]\!]_{\Gamma(\text{subj},C)} \cdot \Lambda_0[\![VP]\!]_C = \emptyset.$$

Likewise, $\Lambda_1[\![NP]\!]_C = \Lambda_1[\![VP]\!]_C = \emptyset$. Furthermore, $\Lambda_1[\![D]\!]_C = \Phi(\Lambda)[\![D]\!]_C$, $\Lambda_1[\![N]\!]_C = \Phi(\Lambda)[\![N]\!]_C$, and $\Lambda_1[\![V]\!]_C = \Phi(\Lambda)[\![V]\!]_C$ as defined above. In fact, $\Lambda^G[\![D]\!]_C = \Lambda_1[\![D]\!]_C$, $\Lambda^G[\![N]\!]_C = \Lambda_1[\![N]\!]_C$, and $\Lambda^G[\![V]\!]_C = \Lambda_1[\![V]\!]_C$.

At the next iteration $\Lambda_2[\![S]\!]_C = \Lambda_2[\![VP]\!]_C = \emptyset$, but

$$\Lambda_2[\![NP]\!]_C = \Phi(\Lambda_1)[\![NP]\!]_C = \begin{cases} \Lambda_1[\![N]\!]_C & \text{if } \langle \text{def: no} + \text{num: pl} \rangle \leq C, \\ \Lambda_1[\![D]\!]_C \cdot \Lambda[\![N]\!]_C & \text{otherwise.} \end{cases}$$

So,

$$\Lambda_2[\![NP]\!]_C = \begin{cases} \{dogs, cats\} & \text{if } \langle \text{def: no} + \text{num: pl} \rangle \leq C, \\ \{the\ dog, the\ cat\} & \text{if } \langle \text{def: yes} + \text{num: sg} \rangle \leq C, \\ \{the\ dogs, the\ cats\} & \text{if } \langle \text{def: yes} + \text{num: pl} \rangle \leq C, \\ \{a\ dog, a\ cat\} & \text{if } \langle \text{def: no} + \text{num: sg} \rangle \leq C, \\ \emptyset & \text{otherwise.} \end{cases}$$

Here, we have reached the least fixpoint for the nonterminal NP and so $\Lambda^G[\![NP]\!]_C = \Lambda_2[\![NP]\!]_C$.

Continuing in this fashion we find that

$$\Lambda^G[\![VP]\!]_C = \Lambda_3[\![VP]\!]_C$$

$$= \begin{cases} \{chased\ dogs, chased\ cats\} & \text{if } \langle \text{obj: } \langle \text{def: no} + \text{num: pl} \rangle \rangle \leq C, \\ \{chased\ the\ dog, chased\ the\ cat\} & \text{if } \langle \text{obj: } \langle \text{def: yes} + \text{num: sg} \rangle \rangle \leq C, \\ \{chased\ the\ dogs, chased\ the\ cats\} & \text{if } \langle \text{obj: } \langle \text{def: yes} + \text{num: pl} \rangle \rangle \leq C, \\ \{chased\ a\ dog, chased\ a\ cat\} & \text{if } \langle \text{obj: } \langle \text{def: no} + \text{num: sg} \rangle \rangle \leq C, \\ \emptyset & \text{otherwise.} \end{cases}$$

Also,

$$\Lambda^G[\![S]\!]_C = \Lambda_4[\![S]\!]_C$$

$$= \begin{cases} \left\{ \begin{array}{l} dogs\ chased\ dogs, \\ dogs\ chased\ cats, \\ cats\ chased\ dogs, \\ cats\ chased\ cats \end{array} \right\} & \text{if } \langle \text{subj: } \langle \text{def: no} + \text{num: pl} \rangle \rangle + \langle \text{obj: } \langle \text{def: no} + \text{num: pl} \rangle \rangle \leq C, \\ \left\{ \begin{array}{l} dogs\ chased\ the\ dog, \\ dogs\ chased\ the\ cat, \\ cats\ chased\ the\ dog, \\ cats\ chased\ the\ cat \end{array} \right\} & \text{if } \langle \text{subj: } \langle \text{def: no} + \text{num: pl} \rangle \rangle + \langle \text{obj: } \langle \text{def: yes} + \text{num: sg} \rangle \rangle \leq C, \\ \vdots & \vdots, \\ \left\{ \begin{array}{l} a\ dog\ chased\ the\ dogs, \\ a\ dog\ chased\ the\ cats, \\ a\ cat\ chased\ the\ dogs, \\ a\ cat\ chased\ the\ cats \end{array} \right\} & \text{if } \langle \text{subj: } \langle \text{def: no} + \text{num: sg} \rangle \rangle + \langle \text{obj: } \langle \text{def: yes} + \text{num: pl} \rangle \rangle \leq C, \\ \left\{ \begin{array}{l} a\ dog\ chased\ a\ dog, \\ a\ dog\ chased\ a\ cat, \\ a\ cat\ chased\ a\ dog, \\ a\ cat\ chased\ a\ cat \end{array} \right\} & \text{if } \langle \text{subj: } \langle \text{def: no} + \text{num: sg} \rangle \rangle + \langle \text{obj: } \langle \text{def: no} + \text{num: sg} \rangle \rangle \leq C, \\ \emptyset & \text{otherwise.} \end{cases}$$

5.2 Conclusion

We have shown that a set of strings generated by a grammar is equal to the set of strings generated by the least fixpoint of a map defined on the nonterminals of the grammar for any context. This provides us with proof that our grammar does as we intend it to do, which we have illustrated in Example 5.11. To further this example, we have developed an intensional context-free grammar that generates a small but illuminating fragment of

English. We use this grammar as an example of an application of ICFG in the field of Natural Language Processing. The grammar is introduced in the next chapter, Chapter 6, and its implementation is discussed in Chapter 7.

6 An Application of ICFG

Natural language processing is a discipline in computer science in which we attempt to simulate natural languages through the use of computer programming techniques. One way of accomplishing this is to develop natural language grammars and apply them directly to sentence construction and parsing. To illustrate the usefulness and correctness of our grammar we will do just that.

In this chapter we develop an example intensional context-free grammar to be used to generate a small but illustrative English grammar. This grammar will be used to generate a complete and correct set of English sentences using the context space to account for the various subcategorization and agreement issues discussed throughout the chapter. To do this we begin with a brief introduction of linguistics and natural language syntax. Throughout the rest of the chapter we build our example grammar from the ground up, starting with a context-free grammar and adding tagged rules to deal with various issues as we encounter them.

6.1 Natural Language Syntax

Linguistics is the study of human languages. Broadly, this covers various aspects of human languages; historical, social, philosophical, psychological, etc. A central component of linguistic study is *linguistic competence*, that is, the ability of the users of a language to produce and understand a vast number of utterances of that language. In linguistics, the mental system which allows us this linguistic competence is called the *grammar* of the language. It is important to note that the use of the term grammar here differs from the more common use of grammar in relation to languages. This other use of grammar is what linguists refer to as *prescriptive grammar*, which is the type of grammar

taught in elementary school. Prescriptive grammar is a set of rules that constrain the use of a language. This includes things like '*i* before *e* except after *c*'. The term grammar that *we* refer to, known as *descriptive grammar*, is a system for describing a language and its use.

A descriptive grammar can be further broken down into smaller components, which represent a telescoping view of the structures of a language. These components are *phonetics*, the articulation and perception of speech sounds; *phonology*, the patterning of speech sounds; *morphology*, word formation; *syntax*, sentence formation; *semantics*, word and sentence meaning; and *pragmatics*, meaning in the larger context. A fundamental claim of linguistics is that all languages have a grammar, in fact, multiple grammars, that satisfy the components introduced above.

The central component of a grammar is the syntax, in which we define a system of rules and categories that underlie sentence formation. Words can be grouped into *syntactic categories*, classified by their meaning, how they can be altered and where they can occur in a sentence[23]. There are two types of syntactic categories, *lexical* and *non-lexical*. The lexical categories are those that hold the words with substance, that is, those that have meanings that are simpler to define. They are *nouns* (N), *verbs* (V), *adjectives* (A), *prepositions* (P), and *adverbs* (Adv). The non-lexical categories are *determiners* (D), *auxiliary verbs* (Aux), *conjunctions* (Con), *qualifiers* ($Qual$) and *degree words* (Deg). These are words that are typically harder to define and are used to enhance the words in the lexical categories.

[23] Note that syntactic categories are not mutually exclusive. There exist words which can appear in more than one category.

For example, nouns are words that typically name entities, like individuals ($Rich$, $king$) or objects ($book$, $crown$). In English, nouns can be inflected for number, that is, affixed with an s to create the plural form ($kings$, $books$)[24]. Moreover, nouns often appear with a determiner such as the or a. On the other hand, verbs designate actions (run, $jump$), can be inflected for tense (ran, $jumped$) and may appear with auxiliary verbs ($may\ run$, $will\ jump$).

In contemporary linguistics, sentences are seen as being constructed using *phrase structure rules*[25]. With phrase structure rules, words are grouped together into structural units called *phrase structures* (or *constituent structures*), which in turn are grouped together to make larger phrase structures and so on, until a sentence is formed. Similarly, sentences are parsed by being broken down into successively smaller structural units using the same rules. Phrase structure rules are typically denoted with the arrow symbol (→) accompanied by a single symbol on the left hand side and a string of symbols on the right hand side.

It turns out that sentence construction is a little more complicated than this. There are other factors taken into account when grouping words together into phrases. Universal to formal grammars of English is that a sentence has a *subject*, in the form of the top-most NP, a verb V, which is the head of the top-most VP and an optional *object*, the complement NP of the top-most VP[26]. Furthermore, words can be *subcategorized* based on information about a words complement options. For example, some verbs require a NP complement as with the verb $devour$, shown in (1) and (2).

(1) $The\ boy\ devoured\ the\ sandwich.$

[24] Of course not all nouns in English are pluralized by adding an *s*, for example children, mice, fish, etc.
[25] This is known as immediate constituent analysis.
[26] Whether or not an object is present depends on the type of verb, as discussed below.

(2) $*The\ boy\ devoured.$[27]

Such verbs are called *transitive verbs* while those that do not take complements are called *intransitive verbs*.

Another factor that needs to be accounted for is *agreement*. Different word and phrase types have certain features as a group. In some cases there must be agreement between the features of one or more other words or phrases. For example, in English, there is agreement between the form of the verb and the number and person of the subject. This informs sentences like (3) and (4).

(3) $The\ dog\ chases\ the\ cat.$
(4) $The\ dogs\ chase\ the\ cat.$

In (3) the affix s on the verb $chase$ is in agreement with the third person, singular form of the subject noun dog, while in (4) the verb lacks the s because the subject is third person, plural.

There are many more examples of similar issues that arise in English and we will see some of them while we develop our example grammar in the next section. It is these very issues that lead us to incorporate the context space presented in the previous chapters into our set of context-free grammar rules.

6.2 Natural Language Processing with ICFG

In this section we build an intensional grammar for a small fragment of English. By the time the grammar is fully realized it will be capable of generating English sentences that include noun-determiner agreement, subject-verb agreement, transitive and intransitive verbs, prepositions and both the active and passive forms of sentences with transitive

[27] The symbol * denotes a sentence that is deemed to be ungrammatical.

verbs. This is accomplished by starting with a simple context-free grammar or an intensional context-free grammar with empty version tags only. We then proceed to add each of these features and see how the version space and version tags can be used to accommodate them.

Consider the intensional context-free grammar with vanilla versions given as (5).

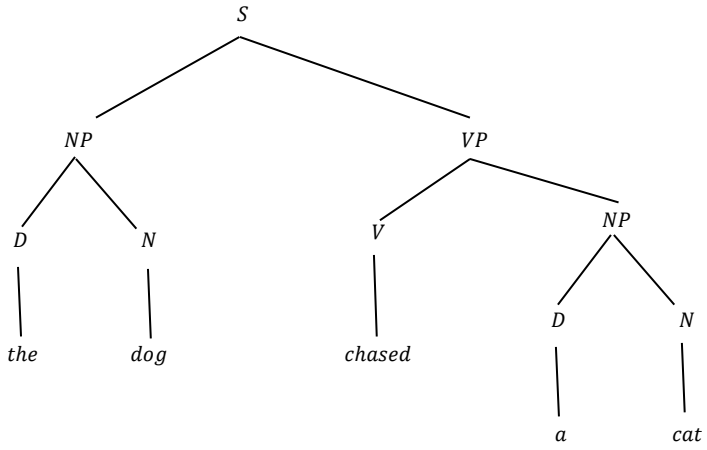
(5) $S \rightarrow NP\ VP$
$NP \rightarrow D\ N$
$VP \rightarrow V\ NP$
$D \rightarrow the|a$
$N \rightarrow dog|cat$
$V \rightarrow chased$

As shown in Chapter 4, this grammar is equivalent to the corresponding context-free grammar. This particular grammar generates the sixteen sentences given in (6) and nothing more.

(6) $The\ dog\ chased\ the\ dog.$
$The\ dog\ chased\ the\ cat.$
$The\ dog\ chased\ a\ dog.$
$The\ dog\ chased\ a\ cat.$
$A\ dog\ chased\ the\ dog.$
$A\ dog\ chased\ the\ cat.$
$A\ dog\ chased\ a\ dog.$
$A\ dog\ chased\ a\ cat.$
$The\ cat\ chased\ the\ dog.$
$The\ catchased\ the\ cat.$
$The\ cat\ chased\ a\ dog.$
$The\ cat\ chased\ a\ cat.$
$A\ cat\ chased\ the\ dog.$
$A\ cat\ chased\ the\ cat.$
$A\ cat\ chased\ a\ dog.$
$A\ cat\ chased\ a\ cat.$

Each of these sixteen sentences is a grammatically correct English sentence generated by the grammar. For instance, consider the derivation tree in (7).

(7)



Suppose now that we want to add the plural forms of the two nouns to our grammar. Thus, we have the context-free grammar given in (8).

(8) $S \rightarrow NP\ VP$
$NP \rightarrow D\ N$
$VP \rightarrow V\ NP$
$D \rightarrow the|a$
$N \rightarrow dog|dogs|cat|cats$
$V \rightarrow chased$

This grammar can also be used to generate the sixteen sentences in (6) as well as forty-eight more, but among those are sentences like (9).

(9) $*The\ dog\ chased\ a\ cats.$

This sentence contains a violation of the determiner-noun agreement. Specifically, you cannot use the indefinite article a in conjunction with the plural form of a noun.

To remedy this situation, we cannot just add the rule $NP \rightarrow N$ since that will just introduce more inappropriate sentences to our language, for instance,

(10) $*Dog\ chased\ cat.$

It is here that we need to start making use of the version tags. Throughout the rest of this chapter, when introducing new dimensions to the version space, we use the following notation $dimension\_name{:}\{set\_of\_possible\_values\}$.

6.2.1 Determiner-Noun Agreement

To ensure determiner-noun agreement we make use of two version dimensions each of which can take on one of two possible values. One dimension is `num` for number of a noun and it has a value range of `sg` or `pl` for singular or plural, respectively. Thus, with our new notation, `num:{sg,pl}`. The other new dimension is `def:{yes,no}` which expresses whether or not the noun is definite or indefinite. With these dimensions at our disposal grammar (5) becomes the ICFG given in (11).

(11) $S \rightarrow NP\ VP$
$NP \rightarrow D\ N$
$NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$
$VP \rightarrow V\ NP$
$D \rightarrow the$
$D \xrightarrow[\langle \text{def:no+num:sg} \rangle]{} a$
$N \xrightarrow[\langle \text{num:sg} \rangle]{} dog|cat$
$N \xrightarrow[\langle \text{num:pl} \rangle]{} dogs|cats$
$V \rightarrow chased$

In grammar (11) we have added the rule $NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$, which will only be chosen when the noun is indefinite and plural. Thus, sentence (10) could never be generated. Furthermore, we have added the tag `<def:no+num:sg>` to the $D \rightarrow a$ rule of (8), which states that the choice of determiner is a only when the noun is singular and indefinite. In all other cases the proper choice of determiner is the. This may seem to suggest that when the current context contains the expression `<def:no+num:pl>` that the determiner is to be the but this is not the case for in this situation the

$NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$ rule would be chosen and thus no determiner would be selected. Finally, we have also divided the N rules into two types, singular and plural, to ensure the proper agreement.

The grammar in (11), as it stands, poses a new problem. We cannot have current context specifications of the `def` and `num` dimensions for both the subject and object nouns simultaneously. For, if one noun were singular while the other were plural, that would lead to an inconsistent version of the form `<num:sg+num:pl>`, which violates one of the axioms of the version space. So, we add new dimension identifiers `subj` and `obj` that take as values versions of the form introduced above, thus `{obj,subj}:{num:{sg,pl}+def:{yes,no}}`.

As our lexicon grows it will become important to distinguish from the various examples of lexical entries. In the case of the lexical category N (and later V) we add a base value corresponding to the words root form. That is, `{obj,subj}:{`N`+num:{sg,pl}+def:{yes,no}}`, where N is a root noun. In the case of the other lexical categories the choice of word will depend on other features of the context space, as in the case of determiners described above. Finally, to avoid duplicate rules for subject and object nouns we will make use of the drill-down operator, giving us the grammar given in (12).

(12) $S \rightarrow NP[\text{subj}]\ VP$
$NP \rightarrow D\ N$
$NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$
$VP \rightarrow V\ NP[\text{obj}]$
$D \rightarrow the$
$D \xrightarrow[\langle \text{def:no+num:sg} \rangle]{} a$
$N \xrightarrow[\langle \text{dog} \rangle]{} dog$
$N \xrightarrow[\langle \text{dog+num:pl} \rangle]{} dogs$

$N \underset{\langle \text{cat} \rangle}{\longrightarrow} cat$
$N \underset{\langle \text{cat+num:pl} \rangle}{\longrightarrow} cats$
$V \rightarrow chased$

Before providing an example derivation of a sentence generated by grammar (12), we will introduce another new notation convention to be adopted throughout the rest of the dissertation. The representation of the current context at any given time in a derivation can become quite unruly and unreadable in the traditional notation as illustrated in (13).

(13) $C = \langle \text{subj}: \langle \text{dog} + \text{num}: \text{pl} + \text{def}: \text{no} \rangle + \text{obj}: \langle \text{cat} + \text{num}: \text{sg} + \text{def}: \text{no} \rangle \rangle$

Instead, we borrow the AVM notation used in many other grammars to represent the same context, shown in (14).

(14)

$$C = \begin{bmatrix} \text{subj}: & \begin{bmatrix} & \text{dog} \\ \text{num}: & \text{pl} \\ \text{def}: & \text{no} \end{bmatrix} \\ \text{obj}: & \begin{bmatrix} & \text{cat} \\ \text{num}: & \text{sg} \\ \text{def}: & \text{no} \end{bmatrix} \end{bmatrix}$$

Thus, given the current context (14), we have derivation (16) for sentence (15).

(15) $Dogs\ chased\ a\ cat.$

(16) $S \underset{C}{\Rightarrow} NP[\text{subj}]\ VP \underset{\langle \text{dog+num:pl+def:no} \rangle}{\Longrightarrow} N\ VP \underset{\langle \text{dog+num:pl+def:no} \rangle}{\Longrightarrow} dogs\ VP$

$\underset{C}{\Rightarrow} dogs\ V\ NP[\text{obj}] \underset{C}{\Rightarrow} dogs\ chased\ N[\text{obj}] \underset{\langle \text{cat+num:sg+def:no} \rangle}{\Longrightarrow} dogs\ chased\ D\ N$

$\underset{\langle \text{cat+num:sg+def:no} \rangle}{\Longrightarrow} dogs\ chased\ a\ N \underset{\langle \text{cat+num:sg+def:no} \rangle}{\Longrightarrow} dogs\ chased\ a\ cat$

So far we have one verb with one form. Suppose we want to add other forms of the verb $chase$. We need to introduce new dimensions to the context space for the verb. For now we will add a base value to the entire context, being the root verb, and a

`tense:{past,pres,fut}` dimension for the tense of the verb and thus the sentence. Furthermore, we need a new category for *auxiliary verbs*, called Aux, to account for the future tense of the verb $chase$ which is $will\ chase$. But, before we do any of this we need to consider agreement between the subject noun and verb on number.

6.2.2 Subject-Verb Agreement

In addition to tense, the form of the verb is affected by the number of the subject noun. For example, if the subject is singular and the tense is present then the form of the verb $chase$ is $chases$. Because the form of the verb is dependent on the form of the noun we do not need a new dimension for agreement, we use the preexisting dimension, `subj:num`. So, our grammar is now given as in (17).

(17) $S \rightarrow NP[\text{subj}]\ VP$
$S \xrightarrow[\langle \text{tense:fut} \rangle]{} NP[\text{subj}]\ Aux\ VP$
$NP \rightarrow D\ N$
$NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$
$VP \rightarrow V\ NP[\text{obj}]$
$Aux \rightarrow will$
$D \rightarrow the$
$D \xrightarrow[\langle \text{def:no+num:sg} \rangle]{} a$
$N \xrightarrow[\langle \text{dog} \rangle]{} dog$
$N \xrightarrow[\langle \text{dog+num:pl} \rangle]{} dogs$
$N \xrightarrow[\langle \text{cat} \rangle]{} cat$
$N \xrightarrow[\langle \text{cat+num:pl} \rangle]{} cats$
$V \xrightarrow[\langle \text{chase} \rangle]{} chase$
$V \xrightarrow[\langle \text{chase+tense:past} \rangle]{} chased$
$V \xrightarrow[\langle \text{chase+tense:pres+subj:num:sg} \rangle]{} chases$

With this new grammar we can produce the same derivation as (16) with the current context given in (18).

(18)

$$C = \begin{bmatrix} & \text{chase} & \\ \text{tense:} & \text{past} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{no} \end{bmatrix} \\ \text{obj:} & \begin{bmatrix} & \text{cat} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{no} \end{bmatrix} \end{bmatrix}$$

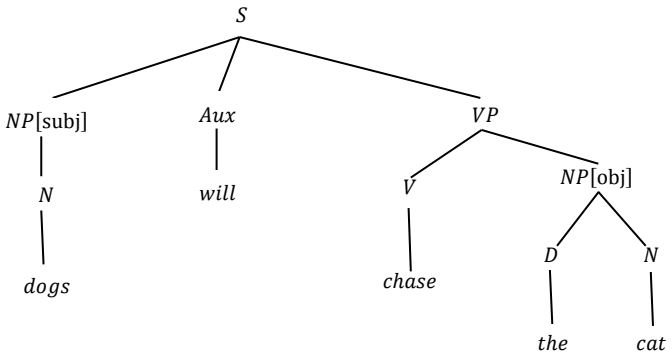
On the other hand, if the current context is changed to (19),

(19)

$$C = \begin{bmatrix} & \text{chase} & \\ \text{tense:} & \text{fut} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{no} \end{bmatrix} \\ \text{obj:} & \begin{bmatrix} & \text{cat} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{yes} \end{bmatrix} \end{bmatrix}$$

we get,

(20)



Next we need to consider more verbs and in fact, different types of verbs. So far we have one verb, *chase*, with three forms. The verb *chase* is an example of a *transitive verb*, that is, a verb that takes an object noun as an argument. There are other types of verbs in English called *intransitive verbs*. Those that do not, and cannot, have an object noun associated with them. Our grammar needs to distinguish between these two types of verbs for a number of reasons. First and foremost, because we do not want to allow a

sentence with a transitive verb that has no direct object (*$*dogs\ chase$*) nor one with an intransitive verb with an direct object (*$*the\ dog\ ran\ the\ cat$*).

6.2.3 Transitive vs. Intransitive Verbs

We want to add an intransitive verb, the root verb run, to the grammar and its corresponding forms, $runs$ and ran. To do so, we introduce a new dimension, `trans:{yes,no}`, that simply signifies if the verb is transitive or not. So, the entries for $chase$ will now include the version value `trans:yes` and the new entries for run will include `trans:no`. Thus, we replace all the versions of the V rule in grammar (17) with those given in (21).

(21) $V \xrightarrow[\langle \text{chase+trans:yes} \rangle]{} chase$
$V \xrightarrow[\langle \text{chase+tense:past+trans:yes} \rangle]{} chased$
$V \xrightarrow[\langle \text{chase+tense:pres+subj:num:sg+trans:yes} \rangle]{} chases$
$V \xrightarrow[\langle \text{run+trans:no} \rangle]{} run$
$V \xrightarrow[\langle \text{run+tense:past+trans:no} \rangle]{} ran$
$V \xrightarrow[\langle \text{run+tense:pres+subj:num:sg+trans:no} \rangle]{} runs$

Furthermore, we need a new version of the VP rule of the form,

(22) $VP \xrightarrow[\langle \text{trans:no} \rangle]{} V$.

By adding the `trans` dimension to each verb entry in the lexicon we do not need to add a `trans:yes` tag to our first VP rule, letting it be a default rule. In this way, any call with a consistent, current context containing `trans:no` will correctly use the $VP \xrightarrow[\langle \text{trans:no} \rangle]{} V$ rule. To be sure, consider the following four contexts:

(23)

a. $C = \begin{bmatrix} & \text{chase} & \\ \text{tense:} & \text{past} & \\ \text{trans:} & \text{yes} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{yes} \end{bmatrix} & \\ \text{obj:} & \begin{bmatrix} & \text{cat} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{no} \end{bmatrix} & \end{bmatrix}$

b. $C = \begin{bmatrix} & \text{run} & \\ \text{tense:} & \text{past} & \\ \text{trans:} & \text{no} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{yes} \end{bmatrix} & \end{bmatrix}$

c. $C = \begin{bmatrix} & \text{chase} & \\ \text{tense:} & \text{past} & \\ \text{trans:} & \text{no} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{yes} \end{bmatrix} & \\ \text{obj:} & \begin{bmatrix} & \text{cat} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{no} \end{bmatrix} & \end{bmatrix}$

d. $C = \begin{bmatrix} & \text{run} & \\ \text{tense:} & \text{past} & \\ \text{trans:} & \text{yes} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{yes} \end{bmatrix} & \end{bmatrix}$

Here, (23)a and (23)b correctly produce the sentences (24)a and (24)b, respectively.

(24) a. *The dogs chased a cat.*
b. *The dogs ran.*

But, although the derivation for (23)c will select $VP \xrightarrow[\langle \text{trans:no} \rangle]{} V$ and the derivation for (23)d will select $VP \rightarrow V\ NP[\text{obj}]$, neither will derive the incorrect sentences they seem to be heading toward because there are no corresponding versions of the verbs given in the lexicon.

This may seem the less efficient way of doing things, (and in fact we eventually add the `trans` dimension to both the rules and lexical entries later,) but there is a good reason for it. Some verbs can actually occur in both categories, that is, they are both

transitive and intransitive. Consider for example the verb eat in its transitive form ((25)a) and intransitive form ((25)b).

(25) a. $The\ dogs\ eat\ a\ cat.$
b. $The\ dogs\ eat.$

Because of the nature of the version space and the best-fit algorithm, applying the `trans` dimension at the lexical level allows us to use only one entry for each form of the verb. This is one of those areas in which our grammar is more efficient. In most grammars they need two entries, one for each type, but by leaving out the `trans` dimension for these types of verbs in our grammar, we can apply either VP rule when those verbs appear in the current context.

Why does this work? To illustrate the answer to this question we need to add the following rules for the verb eat to our grammar.

(26) $V \xrightarrow[\langle \text{eat} \rangle]{} eat$
$V \xrightarrow[\langle \text{eat+tense:past} \rangle]{} ate$
$V \xrightarrow[\langle \text{eat+tense:pres+subj:num:sg} \rangle]{} eats$

Now, let us derive the sentence generated by the current context given in (27), in (28).

(27)

$$C = \begin{bmatrix} & \text{eat} & \\ \text{tense:} & \text{past} & \\ \text{trans:} & \text{yes} & \\ \text{subj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{pl} \\ \text{def:} & \text{yes} \end{bmatrix} & \\ \text{obj:} & \begin{bmatrix} & \text{cat} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{no} \end{bmatrix} & \end{bmatrix}$$

(28) $S \underset{C}{\Rightarrow} NP[\text{subj}]\ VP \xrightarrow[\langle \text{dog+num:pl+def:yes} \rangle]{}{}^{*}\ the\ dog\ VP \underset{C}{\Rightarrow} the\ dog\ V\ NP[\text{obj}]$

$\underset{C}{\Rightarrow} the\ dog\ ate\ NP[\text{obj}] \xrightarrow[\langle \text{cat+num:sg+def:no} \rangle]{}{}^{*}\ the\ dog\ ate\ a\ cat$

At the point in the above derivation where we make the step,

$$the\ dog\ V\ NP[\text{obj}] \underset{C}{\Rightarrow} the\ dog\ ate\ NP[\text{obj}]$$

we choose the verb ate because it is the best fit for the current context C, since `<eat+tense:past>` refines to C maximally. A similar situation occurs when the `trans` dimension is set to `no`, deriving the sentence,

(29) $The\ dog\ ate.$

Before we move on let us take stock of our grammar. In summary, the set of possible dimension-value pairs in our current context at any given time are as follows:

(30)

$$\begin{cases} & \{chase, run, eat\} \\ tense: & \{past, pres, fut\} \\ trans: & \{yes, no\} \\ subj: & \begin{cases} & \{dog, cat\} \\ num: & \{sg, pl\} \\ def: & \{yes, no\} \end{cases} \\ obj: & \begin{cases} & \{dog, cat\} \\ num: & \{sg, pl\} \\ def: & \{yes, no\} \end{cases} \end{cases}$$

As it stands, we have two transitive verbs ($chase, eat$), with which we can create $3 \cdot 2^6$ (3 tense × 2 subj × 2 num × 2 def × 2 obj × 2 num × 2 def) correct sentences, giving us exactly $2 \cdot 3 \cdot 2^6 = 384$ transitive sentences. We also have two intransitive verbs (run, eat), which give us exactly $2 \cdot 3 \cdot 2^3 = 48$ intransitive sentences, for a grand total of exactly 432 sentences.

This, in itself, illustrates that the grammar works on a practical level and even shows one of the benefits of our grammar in the use of the verb eat, as described in this section. Now, we extend it further by adding prepositional phrases and then finally we

explore how the grammar handles something a little more complex with the active versus passive sentence forms.

6.2.4 Prepositions

We add the noun *house* and the prepositions *in* and *to* to the grammar in order to generate sentences like (31) and (32).

(31) *The dog chased the cat in the house*.
(32) *The dog ran to the house*.

To do this we need to also add two new verb phrase rules, one for transitive and one for intransitive verbs with a prepositional phrase, and a preposition phrase rule that takes an object noun phrase. Furthermore, we need a new dimension in the version space called `obl`, for oblique object, that has an `obj` value plus another new dimension `pcase:{loc,goal}`, describing the type of preposition. Essentially, the `pcase:loc` dimension corresponds to the preposition *in* and `pcase:goal` to the preposition *to*. In summary, we add the following rules to our current grammar:

(33) $VP \xrightarrow[\langle \text{trans:no+obl} \rangle]{} V\ PP[\text{obl}]$
$VP \xrightarrow[\langle \text{trans:yes+obl} \rangle]{} V\ NP[\text{obj}]\ PP[\text{obl}]$
$PP \rightarrow P\ NP[\text{obj}]$
$N \xrightarrow[\langle \text{house} \rangle]{} house$
$N \xrightarrow[\langle \text{house+num:pl} \rangle]{} houses$
$P \xrightarrow[\langle \text{pcase:loc} \rangle]{} in$
$P \xrightarrow[\langle \text{pcase:goal} \rangle]{} to$

In both of the new versions of the VP rule we have introduced a new notation, for example, the version tag `<trans:yes+obl>` states that this is the appropriate version of the VP rule when the verb is transitive and there is an oblique object (regardless of its corresponding value.) Note also that we have included a drill-down operation, attached to

the PP in both VP rules. Because both the object noun and the oblique noun are objects we need to distinguish between them in the context space, much like we did with the subject and object nouns in Section 6.2.1. Thus, we can now generate sentences (31) and (32), which have a transitive verb with an oblique object and an intransitive verb with an oblique, respectively.

6.3 Active-Passive Sentences

Transitive sentences have a pair of forms that correspond to one another called the *active* and *passive* forms. In the active form of a sentence the subject is the initiator of the action described by the verb, as in sentence (34), where the dog is the chaser of the cat.

(34) $The\ dog\ chased\ the\ cat.$

In the passive form(s) of the sentence the subject is the recipient of the action ((35) and (36)) and the object is optional, that is it may be dropped ((36)).

(35) $The\ cat\ was\ chased\ by\ the\ dog.$
(36) $The\ cat\ was\ chased.$

Thus, the dog is still the chaser and the cat the chased but in (34) the dog is the subject while in (35) and (36) the cat is the subject. In general, the object noun of the active sentence becomes the subject noun of the passive, the active subject optionally becomes the passive object, complementing the preposition by and the form of the verb changes to its *passive participle* form.

In developing formal grammars for the purposes of computational linguistics, it is common to tackle the active-passive relationship early on. It is a well-studied idea that is simple enough to understand but more challenging to implement than it would first appear. Consider for example sentence (36) above. You cannot simple apply a $VP \rightarrow V$

rule, regardless of the type of grammar, because $chase$ is a transitive verb even though it appears to be intransitive. Verbs maintain their subcategorization properties from the active sentence while taking on a different form. The grammar needs to somehow keep the two forms related while being able to construct both. In Transformational Grammar they generate the passive from the active through transformations. In Lexical-Functional Grammar they map semantic roles to the subject, verb and object, through the argument structure (*a*-structure). For us, we simply introduce a new version dimension, `voice:{act,pass}`, corresponding to the active and passive sentence forms, respectively.

Along with the new dimension we add a number of new lexical entries and rules to our grammar, and we modify a few other existing rules to accommodate the new version dimension. We introduce each of these additions in succession within this section and then summarize the complete grammar in the conclusion.

First, we need to add the preposition by. For now, we see by as a special case preposition with no `pcase`. Instead, we will only use by when the `voice` is `pass`. Thus, its lexical entry is

(37) $P \xrightarrow[\langle \text{voice:pass} \rangle]{} by$.

We also need new entries for the Aux rule, to account for the different forms of $will$ being added by the passive. These are as follows[28];

(38) $Aux \xrightarrow[\langle \text{tense:pres+voice:pass+subj:num:pl} \rangle]{} are\ being$
$Aux \xrightarrow[\langle \text{tense:pres+voice:pass} \rangle]{} is\ being$
$Aux \xrightarrow[\langle \text{tense:fut+voice:pass} \rangle]{} will\ be$
$Aux \xrightarrow[\langle \text{tense:past+voice:pass} \rangle]{} was$

[28] For simplicity sake we are combining entries, like $are\ being$, into one rule.

$$Aux \xrightarrow[\langle \text{tense:past+voice:pass+subj:num:pl} \rangle]{} were$$

In addition, we modify the version tags of the transitive verb entries to include the `voice` dimension. In the case of $chase$, the form $chased$ now becomes the default and as such an extra version tag is attached to the form $chase$, the previous default. Giving us:

(39) $$V \xrightarrow[\langle \text{chase+trans:yes} \rangle]{} chased$$

$$V \xrightarrow[\langle \text{chase+tense:pres+subj:num:pl+trans:yes+voice:act} \rangle]{} chase^{29}$$

$$\langle \text{chase+tense:fut+trans:yes+voice:act} \rangle$$

$$V \xrightarrow[\langle \text{chase+tense:pres+subj:num:sg+trans:yes+voice:act} \rangle]{} chases$$

Finally, we need the passive form of the verb eat, which will become the default rule due to the fact that it occurs in more situations.

(40) $$V \xrightarrow[\langle \text{eat} \rangle]{} eaten$$

This results in an additional tag being attached to the eat version of the lexical entry.

(41) $$V \xrightarrow[\langle \text{eat+tense:pres+subj:num:pl+voice:act} \rangle]{} eat$$

$$\langle \text{eat+tense:fut+voice:act} \rangle$$

$$V \xrightarrow[\langle \text{eat+tense:past+voice:act} \rangle]{} ate$$

$$V \xrightarrow[\langle \text{eat+tense:pres+subj:num:sg+voice:act} \rangle]{} eats$$

In both cases, the passive form of the verb is the default simply because there are more sentences in which that form occurs. Of course, this is more efficient as we will use less memory to store the lexicon and there are less version tags to consider during application of the best-fit algorithm.

Incorporating passive sentences into our grammar also affects the constituent structure rules. We modify or add version tags in the S rules and the VP rules.

[29] The notation here shows that two versions of the V rule have the same derivation. That is, in either of the contexts listed the nonterminal V rewrites to $chase$.

Furthermore, we add two more versions of the VP rule. For the S rule, we need only add another tag to the $S \xrightarrow[\langle \text{tense:fut} \rangle]{} NP[\text{subj}]\ Aux\ VP$ version, which is the version of S we want to apply when dimension `voice` has value `pass`. This gives us the new version:

(42) $S \xrightarrow[\substack{\langle \text{tense:fut} \rangle \\ \langle \text{voice:pass} \rangle}]{} NP[\text{subj}]\ Aux\ VP$

Note that S rewrites to $NP[\text{subj}]\ Aux\ VP$ when either expression `<tense:fut>` or `<voice:pass>` is in the current context, this includes the expression `<tense:fut+voice:pass>`, which has both.

With the VP rules, we need to account for four types of passive sentences using the `voice` dimension. They are of the following form:

(43) $The\ cat\ was\ chased.$
(44) $The\ cat\ was\ chased\ by\ the\ dog.$
(45) $The\ cat\ was\ chased\ in\ the\ house.$
(46) $The\ cat\ was\ chased\ by\ the\ dog\ in\ the\ house.$

In the case of (43), we modify the intransitive VP rule to include another version tag.

(47) $VP \xrightarrow[\substack{\langle \text{trans:no+voice:act} \rangle \\ \langle \text{trans:yes+voice:pass} \rangle}]{} V$

Thus, rule (47) applies to active sentences with an intransitive verb as well as to passive sentences with a transitive verb but no object.[30]

For sentence (46) we need an entirely new version of the VP rule, one that takes two prepositional phrases. Included in the two PPs there are a direct object and an oblique. The difference between this situation and what we saw in sentence (31) is that the direct object is embedded in the first PP. Thus, the rule looks like:

(48) $VP \xrightarrow[\langle \text{trans:yes+obj+obl+voice:pass} \rangle]{} V\ PP\ PP[\text{obl}]$

[30] Although at this point it is not evident that this rule will apply only when there is no object. With the addition of the next three rules this becomes clear.

Notice that we do not drill-down on the first PP because it is not an oblique. In the case of both PPs an object drill-down is applied at the daughter NP node by the $PP \rightarrow P\ NP$[obj] rule.

Sentences (44) and (45) pose an interesting problem. In both cases we want to apply a $VP \rightarrow V\ PP$ rule, but in each case there is a different type of prepositional phrase. In (44) the preposition phrase takes the subject from the active form (now the direct object) and in (45) it takes the oblique object from the active. In our grammar this leads to two versions of the $VP \rightarrow V\ PP$ rule. The version we need to account for (45) already exists in the grammar, we only need to add a new version tag to cover the passive case.

(49) $VP \xrightarrow[\substack{\langle \text{trans:yes+obl+voice:pass} \rangle}]{\langle \text{trans:no+obl+voice:act} \rangle} V\ PP$[obl]

For sentence (44) we need a new version of the rule in which there is no drill-down operation attached to the PP, for the same reasons presented above for rule (48).

(50) $VP \xrightarrow[\langle \text{trans:yes+obj+voice:pass} \rangle]{} V\ PP$

To illustrate a number of these changes, we derive sentence (46). Consider the following current context:

(51)

$$C = \begin{bmatrix} & \text{chase} & & \\ \text{tense:} & \text{past} & & \\ \text{trans:} & \text{yes} & & \\ \text{voice} & \text{pass} & & \\ \text{subj:} & \begin{bmatrix} & \text{cat} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{yes} \end{bmatrix} & & \\ \text{obj:} & \begin{bmatrix} & \text{dog} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{yes} \end{bmatrix} & & \\ \text{obl:} & \begin{bmatrix} \text{pcase:} & \text{loc} & \\ \text{obj:} & \begin{bmatrix} & \text{house} \\ \text{num:} & \text{sg} \\ \text{def:} & \text{yes} \end{bmatrix} \end{bmatrix} & & \end{bmatrix}$$

The derivation proceeds as follows:

(52) $S \underset{C}{\Rightarrow} NP[\text{subj}]\ Aux\ VP$

$\underset{\langle \text{cat+num:sg+def:yes} \rangle}{\Longrightarrow}{}^{*}\ the\ cat\ Aux\ VP \underset{C}{\Rightarrow}{}^{*}\ the\ cat\ was\ V\ PP\ PP[\text{obl}]$

$\underset{C}{\Rightarrow}{}^{*}\ the\ cat\ was\ chased\ by\ NP[\text{obj}]\ PP[\text{obl}]$

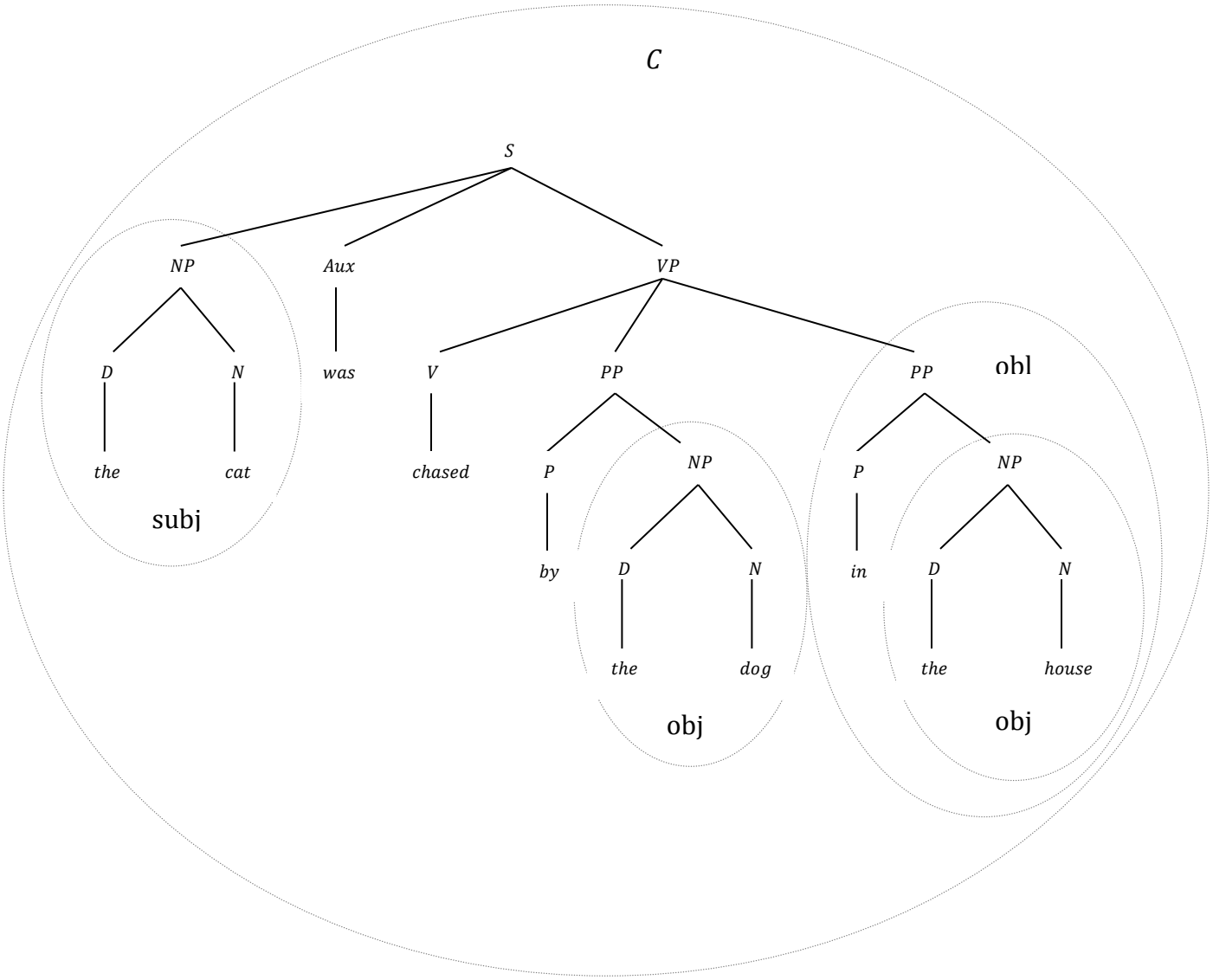
$\underset{\langle \text{dog+num:sg+def:yes} \rangle}{\Longrightarrow}{}^{*}\ the\ cat\ was\ chased\ by\ the\ dog\ PP[\text{obl}]$

$\underset{\langle \text{pcase:loc+obj:}\langle \text{house+num:sg+def:yes} \rangle \rangle}{\Longrightarrow}{}^{*}\ the\ cat\ was\ chased\ by\ the\ dog\ in\ NP[\text{obj}]$

$\underset{\langle \text{house+num:sg+def:yes} \rangle}{\Longrightarrow}{}^{*}\ the\ cat\ was\ chased\ by\ the\ dog\ in\ the\ house.$

Finally, we look at the corresponding derivation tree. We add a visualization of the context space to better appreciate our derivation model.

(53)



6.4 Conclusion

In summary, we have presented an intensional context free grammar $G = (N, T, S, P)$, where N is the set of nine non-terminals,

$N = \{S, NP, VP, PP, D, N, V, P, Aux\}$,

T is the set of twenty-seven terminals,

$$T = \{the, a, dog, dogs, cat, cats, house, houses, chase, chases, chased, run, runs, ran, eat, eats, ate, eaten, in, to, by, will, are\ being, is\ being, will\ be, was, were\},$$

and P is the set of eleven tagged phrase structure rules, (54), and twenty-seven lexical rules, (55).

(54) $S \rightarrow NP[\text{subj}]\ VP$

$S \xrightarrow[\substack{\langle \text{tense:fut} \rangle \\ \langle \text{voice:pass} \rangle}]{} NP[\text{subj}]\ Aux\ VP$

$NP \rightarrow D\ N$

$NP \xrightarrow[\langle \text{def:no+num:pl} \rangle]{} N$

$VP \xrightarrow[\langle \text{trans:yes+obj+voice:act} \rangle]{} V\ NP[\text{obj}]$

$VP \xrightarrow[\substack{\langle \text{trans:no+voice:act} \rangle \\ \langle \text{trans:yes+voice:pass} \rangle}]{} V$

$VP \xrightarrow[\langle \text{trans:yes+obj+obl+voice:act} \rangle]{} V\ NP[\text{obj}]\ PP[\text{obl}]$

$VP \xrightarrow[\substack{\langle \text{trans:no+obl+voice:act} \rangle \\ \langle \text{trans:yes+obl+voice:pass} \rangle}]{} V\ PP[\text{obl}]$

$VP \xrightarrow[\langle \text{trans:yes+obj+voice:pass} \rangle]{} V\ PP$

$VP \xrightarrow[\langle \text{trans:yes+obj+obl+voice:pass} \rangle]{} V\ PP\ PP[\text{obl}]$

$PP \rightarrow P\ NP[\text{obj}]$

(55) $Aux \rightarrow will$

$Aux \xrightarrow[\langle \text{tense:pres+voice:pass+subj:num:pl} \rangle]{} are\ being$

$Aux \xrightarrow[\langle \text{tense:pres+voice:pass} \rangle]{} is\ being$

$Aux \xrightarrow[\langle \text{tense:fut+voice:pass} \rangle]{} will\ be$

$Aux \xrightarrow[\langle \text{tense:past+voice:pass} \rangle]{} was$

$Aux \xrightarrow[\langle \text{tense:past+voice:pass+subj:num:pl} \rangle]{} were$

$D \rightarrow the$

$D \xrightarrow[\langle \text{def:no+num:sg} \rangle]{} a$

$N \xrightarrow[\langle \text{dog} \rangle]{} dog$

$N \xrightarrow[\langle \text{dog+num:pl} \rangle]{} dogs$

$N \xrightarrow[\langle \text{cat} \rangle]{} cat$

$N \xrightarrow[\langle \text{cat+num:pl} \rangle]{} cats$

$N \xrightarrow[\langle \text{house} \rangle]{} house$

$N \xrightarrow[\langle \text{house+num:pl} \rangle]{} houses$

$V \xrightarrow[\langle \text{chase+trans:yes} \rangle]{} chased$

$$V \xrightarrow[\substack{\langle \text{chase+tense:pres+subj:num:pl+trans:yes+voice:act} \rangle \\ \langle \text{chase+tense:fut+trans:yes+voice:act} \rangle}]{} chase$$
$$V \xrightarrow[\langle \text{chase+tense:pres+subj:num:sg+trans:yes+voice:act} \rangle]{} chases$$
$$V \xrightarrow[\langle \text{run+trans:no} \rangle]{} run$$
$$V \xrightarrow[\langle \text{run+tense:past+trans:no} \rangle]{} ran$$
$$V \xrightarrow[\langle \text{run+tense:pres+subj:num:sg+trans:no} \rangle]{} runs$$
$$V \xrightarrow[\langle \text{eat} \rangle]{} eaten$$
$$V \xrightarrow[\langle \text{eat+tense:pres+subj:num:pl+voice:act} \rangle]{} eat$$
$$V \xrightarrow[\langle \text{eat+tense:fut+voice:act} \rangle]{} ate$$
$$V \xrightarrow[\langle \text{eat+tense:pres+subj:num:sg+voice:act} \rangle]{} eats$$
$$P \xrightarrow[\langle \text{pcase:loc} \rangle]{} in$$
$$P \xrightarrow[\langle \text{pcase:goal} \rangle]{} to$$
$$P \xrightarrow[\langle \text{voice:pass} \rangle]{} by$$

Also, we finish with the following set of possible current contexts:

(56)

$$\left\{\begin{array}{ll} & \{chase, run, eat\} \\ tense: & \{past, pres, fut\} \\ trans: & \{yes, no\} \\ voice: & \{act, pass\} \\ subj: & \left\{\begin{array}{ll} & \{dog, cat, house\} \\ num: & \{sg, pl\} \\ def: & \{yes, no\} \end{array}\right\} \\ obj: & \left\{\begin{array}{ll} & \{dog, cat, house\} \\ num: & \{sg, pl\} \\ def: & \{yes, no\} \end{array}\right\} \\ obl: & \left\{\begin{array}{ll} pcase: & \{loc, goal\} \\ obj: & \left\{\begin{array}{ll} & \{dog, cat, house\} \\ num: & \{sg, pl\} \\ def: & \{yes, no\} \end{array}\right\} \end{array}\right\} \end{array}\right\}$$

With this relatively small grammar, we can generate exactly 46,800 valid, grammatically correct sentences and nothing more. To justify this claim, let us consider the three verbs separately, starting with the simplest, *run*. To generate sentences with the verb *run*, the `trans` dimension must be set to `no` and the `voice` dimension to `act`. We

have seen above that with any other setting of those two dimensions the grammar correctly fails to generate a sentence. Thus, the current context for run is of the form in (57)a when there is no oblique object and of the form in (57)b when there is.

(57)

a. $$C = \begin{bmatrix} & \text{run} \\ \text{tense:} & \{\text{past,pres,fut}\} \\ \text{trans:} & \text{no} \\ \text{voice} & \text{act} \\ \text{subj:} & \left\{\begin{matrix} & \{\text{dog,cat,house}\} \\ \text{num:} & \{\text{sg,pl}\} \\ \text{def:} & \{\text{yes,no}\} \end{matrix}\right\} \end{bmatrix}$$

b. $$C = \begin{bmatrix} & \text{run} \\ \text{tense:} & \{\text{past,pres,fut}\} \\ \text{trans:} & \text{no} \\ \text{voice} & \text{act} \\ \text{subj:} & \left\{\begin{matrix} & \{\text{dog,cat,house}\} \\ \text{num:} & \{\text{sg,pl}\} \\ \text{def:} & \{\text{yes,no}\} \end{matrix}\right\} \\ \text{obl:} & \left\{\begin{matrix} \text{pcase:} & \{\text{loc,goal}\} \\ \text{obj:} & \left\{\begin{matrix} & \{\text{dog,cat,house}\} \\ \text{num:} & \{\text{sg,pl}\} \\ \text{def:} & \{\text{yes,no}\} \end{matrix}\right\} \end{matrix}\right\} \end{bmatrix}$$

All told, these contexts can generate

$$3 \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{subj}} + 3 \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{subj}} \cdot \underbrace{\left(2 \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{obj}}\right)}_{\text{obl}} = 3^2 \cdot 2^2 + 3^3 \cdot 2^5 = 36 + 864 = 900$$

sentences with the verb run.

The verb $chase$ presents a rather more complex situation in that we have both active and passive sentences, with or without an oblique object. Furthermore, in some passive cases there is no direct object although there still may be an oblique object. This leads to the following four possible forms of the current context for $chase$.

(58)

a. $$C = \begin{bmatrix} & \text{chase} \\ \text{tense:} & \{\text{past, pres, fut}\} \\ \text{trans:} & \text{yes} \\ \text{voice} & \text{pass} \\ \text{subj:} & \left\{\begin{matrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{matrix}\right\} \end{bmatrix}$$

b. $$C = \begin{bmatrix} & \text{chase} \\ \text{tense:} & \{\text{past, pres, fut}\} \\ \text{trans:} & \text{yes} \\ \text{voice} & \text{pass} \\ \text{subj:} & \left\{\begin{matrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{matrix}\right\} \\ \text{obl:} & \left\{\begin{matrix} \text{pcase:} & \{\text{loc, goal}\} \\ \text{obj:} & \left\{\begin{matrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{matrix}\right\} \end{matrix}\right\} \end{bmatrix}$$

c. $$C = \begin{bmatrix} & \text{chase} \\ \text{tense:} & \{\text{past, pres, fut}\} \\ \text{trans:} & \text{yes} \\ \text{voice} & \{\text{act, pass}\} \\ \text{subj:} & \left\{\begin{matrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{matrix}\right\} \\ \text{obj:} & \left\{\begin{matrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{matrix}\right\} \end{bmatrix}$$

d. $$C = \begin{bmatrix} & \text{chase} \\ \text{tense:} & \{\text{past, pres, fut}\} \\ \text{trans:} & \text{yes} \\ \text{voice} & \{\text{act, pass}\} \\ \text{subj:} & \begin{Bmatrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{Bmatrix} \\ \text{obj:} & \begin{Bmatrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{Bmatrix} \\ \text{obl:} & \begin{Bmatrix} \text{pcase:} & \{\text{loc, goal}\} \\ \text{obj:} & \begin{Bmatrix} & \{\text{dog, cat, house}\} \\ \text{num:} & \{\text{sg, pl}\} \\ \text{def:} & \{\text{yes, no}\} \end{Bmatrix} \end{Bmatrix} \end{bmatrix}$$

Together (58)a and (58)b generate the same number of sentences as (57)a and (57)b, which is 900. The set of current contexts given by (58)c generates,

$$3 \cdot 2 \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{subj}} \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{obj}} = 3^3 \cdot 2^5 = 864$$

active and passive sentences for $chase$ with a direct object but no oblique. Finally, (58)d can generate,

$$3 \cdot 2 \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{subj}} \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{obj}} \cdot \underbrace{\left(2 \cdot \underbrace{(3 \cdot 2 \cdot 2)}_{\text{obj}}\right)}_{\text{obl}} = 3^4 \cdot 2^8 = 81 \cdot 256 = 20{,}736$$

active and passive sentences for $chase$, with both a direct and an oblique object.

In total, that is $900 + 900 + 864 + 20{,}736 = 23{,}400$ sentences involving the verbs $chase$ or run. That leaves the verb eat, which is both transitive and intransitive. Thus, we double the above number giving us $46{,}800$ sentences.

7 Implementation of ICFG

In Chapter 6 we developed an example of an intensional context-free grammar capable of deriving a number of English sentences. The grammar includes examples of sentences with both transitive or intransitive verbs, preposition phrases and both active and passive forms. In this chapter we discuss the implementation of that grammar. For the implementation we use a macro processing language called MMP, developed by Bill Wadge and Paul Svoboda[31], capable of expanding macro calls based on versioned macro definitions, augmented by the version space. As such, it is an obvious choice for our grammar because of its built in facility for dealing with versioning.

In the first section we introduce enough of the macro processing system necessary to facilitate the reader in the understanding of our grammar. In the second section we present the grammar in its MMP form and discuss some of the results. In the third section we outline the idea of using our grammar in a translation system. This includes a working example of one half of the process (English sentence generation from a given French sentence).

7.1 Multipurpose Macro Processor (MMP)

MMP is a macro processing system that takes a text document as input and produces an output text document. The input document consists of text both marked and unmarked by tags. The unmarked text is copied directly to the output document while the marked text is processed producing possibly new text.

[31] No publications on MMP as of yet, just the documentation in the Appendix.

7.1.1 Macro Calls

Macro calls in MMP either have a pair of curly bracketed tags around some text or a single curly bracketed tag. In the opening tag you will find the name of the tag along with a space separated list of zero or more input arguments. For example,

(1) `{memo Tom Dick}I received your proposal.{/memo}`

Here we have a macro called `memo` with two input arguments, `Tom` and `Dick`. Between the two tags we have the text `I received your proposal.`, which is the *body* of the call.

What we see here is just a call to some macro, we do not know at this point what it does with the input arguments or the body of the call. Typically, the MMP processor will replace the tags and body of the call with some new text called the *value* or *result*. For example, the above call may be replaced with the following,

(2) `MEMO`

```
From: Tom
To: Dick

I received your proposal.
```

As we said before, MMP also allows for a single tag with no body, still with optional arguments. For example, the call

(3) `{dept Physics/}`

might expand to

(4) `Physics Department of the University of Victoria`

Furthermore, MMP macro calls can be nested. For example,

(5)
```
{memo Tom Dick}I received a spreadsheet from the {dept
Physics/}.{/memo}
```

could evaluate to

(6)
```
MEMO


From: Tom

To: Dick


I received a spreadsheet from the Physics Department of the
University of Victoria.
```

In the case of nested calls MMP *always* evaluates inner macros first, with one exception, which we will immediately explain.

7.1.2 Quoting

There are times when we want to exert some control over the expansion mechanism. For instance, you may want to delay the evaluation of some nested tags or you may want to have the processor ignore symbols that may be mistaken for macro calls themselves. In MMP we have a special macro for this purpose, the `"` (double quote) symbol. For example,

(7)
```
{"}Greetings from the {dept Physics/}, everyone.{/"}
```

The quote macro returns its body as its result, verbatim, overruling the inner macro evaluation first.

Often, you may want to quote the body of a macro call. The only way to do this is to enclose the body in a quote macro, as in (8).

(8)
```
{memo Tom Dick}{"}I received a spreadsheet from
 the {dept Physics/}.{/"}{/memo}
```

This occurs so frequently that MMP allows quote to be used as a kind of meta-macro, in which the first argument of the quote macro is the name of the macro being defined. In this way, (8) becomes

(9) `{" memo Tom Dick}I received a spreadsheet from the {dept Physics/}.{/"}`

Calls to quote like this are evaluated as a call to the first argument (treated as a macro name) together with the remaining arguments as the arguments of the implied call. The body of the implied call is the quoted body of the original call. Notice that this is a call to the quote macro (not the memo macro) so that the closing tag is `{/"}`, not `{/memo}`.

7.1.3 Macro Definitions

In MMP most of our macros are defined by substitution templates, where each template is a string with spaces for the arguments and body of the calling macro. The simplest macro definitions contain just a string as the substitution template. To define their own template macros users can call a special system macro "`:`" that takes one argument, the name of the defined macro, and a body, the substitution string for the defined macro. For example, consider the macro definition,

(10) `{: uvic}University of Victoria{/:}`

When this macro is evaluated it associates the substitution string `Univerisity of Victoria` to the macro named `uvic`. Thus, all calls to `{uvic/}` henceforth expand to `University of Victoria`.

As before, if you want the template of a macro to contain a nested call meant to be expanded when the macro you are defining is itself called, then you must use the quote macro. So, if you want to define the macro `greeting` with the template,

(11) `Welcome to the Uvic class of {thisyear/}.`

you would have to define the macro as follows,

(12) `{" : greeting}Welcome to the Uvic class of {thisyear/}.{/"}`

This allows you to define the macro `{: this year}2013{/:}`, after the definition of the `greeting` macro and still have the call `{greeting/}` expand to

(13) `Welcome to the Uvic class of 2013.`

7.1.4 Macro Versions

Macro definitions in MMP can be intensional, allowing for macros to have multiple versions. We accomplish this by allowing multiple definitions of the same macro, each having a different tag attached to it. This tag corresponds to the context under which that particular version of the macro is to be expanded. The tag is actually an argument of the definition macro, which follows the macro's name. We use version expressions, defined in previous chapters, to denote context. It is these version expressions that act as the tags in our macro definitions. Consider the following example:

(14)
```
{: ferd <green> <blue>}green and blue{/:}
{: ferd <red>}red{/:}
{: ferd}vanilla{/:}
```

Here, we define the macro `ferd`, which has three versions. If the maximal context under which `ferd` is expanded is `<green>` or `<blue>`, then `ferd` evaluates to `green and blue`. If the context is `<red>`, then `ferd` evaluates to `red`. Under all other possible contexts `ferd` evaluates to `vanilla`. The latter version of `ferd` corresponds to the vanilla version, which in effect is the default version of the macro. In MMP we define the vanilla version by simply omitting the context tag or by using an empty context tag, <>. Note that the vanilla version of a macro is not mandatory.

7.1.5 Context Macros

The MMP interpreter determines the version of the macro to expand under the context in which it is evaluated, the current context. The current context is a background phenomenon that guides macro evaluation, also represented by a version expression. The interpreter's choice of an appropriate macro version is dependent on the version space and best-fit algorithm, ideas developed in Chapters 2 and 3.

There are special system macros for setting and modifying the current context. The `[]` macro allows you to change the current context. The first argument of `[]` is the new current context. This macro can change the current context in one of two ways; you can replace the entire current context with a new context by enclosing the first argument in angle brackets, `< >`, or you can modify particular dimensions of the current context by enclosing the argument in square brackets, `[ ]`.

So, to set the current context to `<green+a:1>` we use the call `{[] <green+a:1>/}`. Then, to modify the value of dimension `a` to `2`, we use the call `{[] [a:2]/}`, changing the current context to `<green+a:2>`. The call `{[] <green+a:1>/}` corresponds to the absolute context modifier, $\langle W \rangle$, and `{[] [a:2]/}` corresponds to the relative modifier, $[W]$, both defined in Section 3.2.

Furthermore, there exists the `!` macro that allows you to change the current context temporarily. Any macro evaluated in the body of the `!` macro will be expanded under the temporary context, defined by the first argument of the `!` macro. For example, the following (along with the `ferd` definition above, (14))

(15)
```
{[] <>/}
   before {ferd/}
   {! <green+a:1>}during {ferd/}{/!}
   after {ferd/}
```

will evaluate to

(16)
```
before vanilla
during green and blue
after vanilla
```

There also exists a system macro, `#`, which returns a full version value of the current context. A call to `#`, without arguments, returns the entire current context. To see the value of some particular dimension in the current context, you enter the name of the dimension in the first argument of the `#` macro. Any other arguments are ignored. So, the following

(17)
```
{[] <a:1+b:2+c:3>/}
{#/}
{# a/}
{# a b/}
```

will produce

(18)
```
<a:1+b:2+c:3>
1
1
```

A combination of the `!` macro and the `#` macro allows us to implement the drill-down operator. For example,

(19)
```
{[] <red+foo:<green+a:1>>/}
before {ferd/}
{! {# foo/}}during {ferd/}{/!}
after {ferd/}
```

will evaluate to

(20)
```
before red
during green and blue
after red
```

7.1.6 Other Relevant System Macros

MMP provides a number of built-in or system macros that are executed for their side effects and/or produce results that cannot be obtained from a template. We have already introduced some of them, for example the definition macro `:` and the quote macro `"`. We use a couple of these system macros in our implementation, thus we introduce them here.

The arguments of a macro call are referred to in its definition by the system macros `1`, `2`, `3`, `4`, …, and its body by the macro `_`. These act as placeholders in our substitution templates, marking the place where the arguments or bodies are to be substituted upon calls to the defined macro. For example, we can define the `uvic` macro as,

(21) `{" : uvic}{1/} and {2/} {_/} University of Victoria.{/"}`

Then, the call

(22) `{uvic Tom Dick}are students at the{/uvic}`

Will expand to

(23) `Tom and Dick are students at the University of Victoria.`

Two more relevant system macros are the `c:` macro, which is equivalent to switch in other languages, and the comma "`,`" macro, used to separate the cases in the `c:` macro. The `c:` macro switches cases based on its first argument. For example, in the following macro definition,

(24)
```
{" : tr}{c: {1/}}
     cat
     {,}chat{/,}
     dog
     {,}chien{/,}
     cow
     {,}vache{/,}
```

```
    default
    {,}hein?{/,}
  {/c:}{/"}
```

macro `c:` evaluates to `chat` if the first argument of macro `tr` is `cat`, `chien` if it is `dog`, `vache` if it is `cow`, and `hein?` in all other cases.

7.2 Implementation of ICFG

We have just seen that MMP is a macro expansion language which outputs strings based on some context space. Furthermore, MMP allows us to nest macro definitions arbitrarily deep while expanding the inner-most tags first. In Chapter 6 we developed a grammar for generating strings (sentences) of English words involving the nouns dog, cat, and $house$, and the verbs $chase$, run and eat. To generate these sentences we apply context-free rewrite rules, nested arbitrarily deep, guided by a context space.

Here it should be evident that the properties of both fit one another and thus we use MMP to implement that grammar. We reproduce the phrase structure rules of Chapter 6 as macro expansion definitions and thus generate the corresponding strings. Let us begin with a look at the lexicon.

7.2.1 lexicon.mmp[32]

Our intensional context-free lexicon from Chapter 6, (55), is reproduced here in the syntax of MMP as (27). Before we see the full lexicon, let us deconstruct and compare one entry. Consider the ICFG lexical entry given in (25).

(25) $V \xrightarrow[\substack{\langle \text{chase+tense:pres+subj:num:pl+trans:yes+voice:act} \rangle \\ \langle \text{chase+tense:fut+trans:yes+voice:act} \rangle}]{} chase$

In MMP, this rule is of the following form:

[32] This is the name of the MMP file containing the lexicon.

(26) `{: V <chase+tense:pres+subj:num:pl+voice:act+trans:yes>`
`<chase+tense:fut+voice:act+trans:yes>}chase {/:}`

You can see here that we are using the `:` macro whose first argument, `V`, is used as the name for a new macro definition. The remaining two arguments are the two context tags under which the new macro is to be expanded. The body of the macro definition, `chase` , becomes the body of the newly defined `V` macro. Thus, when evaluating a call to `{V/}`, the macro expands to `chase` if either `<chase+tense:pres+subj:num:pl+voice:act+trans:yes>` or `<chase+tense:fut+voice:act+trans:yes>` refines to the current context maximally. Of course, this is precisely what we want to happen to this lexical entry when implementing our grammar.

Lexical entries in our grammar do not have nonterminals in their body. Therefore, we do not need macro calls in the body of our lexical macro definitions. For this reason we do not need the `"` macro in our lexical definitions. All in all, we have five lexical macro definitions, each with multiple versions, given as follows:

(27)
```
{: D }the {/:}
{: D <def:no+num:sg>}a {/:}

{: N <dog>}dog {/:}
{: N <dog+num:pl>}dogs {/:}
{: N <cat>}cat {/:}
{: N <cat+num:pl>}cats {/:}
{: N <house>}house {/:}
{: N <house+num:pl>}houses {/:}

{: V <chase+trans:yes>}chased {/:}
{: V <chase+tense:pres+subj:num:pl+voice:act+trans:yes>
<chase+tense:fut+voice:act+trans:yes>}chase {/:}
{: V <chase+tense:pres+subj:num:sg+voice:act+trans:yes>}chases
{/:}
{: V <run+tense:past+trans:no>}ran {/:}
{: V <run+trans:no>}run {/:}
{: V <run+tense:pres+subj:num:sg+trans:no>}runs {/:}
{: V <eat>}eaten {/:}
```

```
{: V <eat+tense:pres+subj:num:sg+voice:act>}eats {/:}
{: V <eat+tense:past+voice:act>}ate {/:}
{: V <eat+tense:pres+subj:num:pl+voice:act>
<eat+tense:fut+voice:act>}eat {/:}

{: Aux}will {/:}
{: Aux <tense:pres+voice:pass+subj:num:pl>}are being {/:}
{: Aux <tense:pres+voice:pass>}is being {/:}
{: Aux <tense:fut+voice:pass>}will be {/:}
{: Aux <tense:past+voice:pass>}was {/:}
{: Aux <tense:past+voice:pass+subj:num:pl>}were {/:}

{: P <pcase:loc>}in {/:}
{: P <pcase:goal>}to {/:}
{: P <voice:pass+obj:~>}by {/:}
```

One final note about the lexical entries in MMP; there is a new notation included in the macro definition `{: P <voice:pass+obj:~>}by {/:}`, specifically the `~`. This symbol represents the current, non-vanilla base value of the given dimension, in this case the `obj` dimension. Thus, if `obj` has a base value (i.e. is not vanilla) and `voice` is `pass` then macro `P` expands to string `by`.

7.2.2 rules.mmp

The phrase structure rules from Chapter 6, (54), are given in MMP as shown in (28):

(28)
```
{" : S}{! {# subj/}}{NP/}{/!}{VP/}{/"}
{" : S <tense:fut> <voice:pass> <tense:fut+voice:pass>}{! {#
subj/}}{NP/}{/!}{Aux/}{VP/}{/"}

{" : NP}{D/}{N/}{/"}
{" : NP <def:no+num:pl>}{N/}{/"}

{" : VP <trans:yes+voice:act+obj:~>}{V/}{! {#
obj/}}{NP/}{/!}{/"}
{" : VP <trans:no+voice:act> <trans:yes+voice:pass>}{V/}{/"}
{" : VP <trans:yes+voice:act+obj:~+obl:pcase:~>}{V/}{! {#
obj/}}{NP/}{/!}{! {# obl/}}{PP/}{/!}{/"}
{" : VP <trans:yes+voice:pass+obj:~>}{V/}{PP/}{/"}
{" : VP <trans:no+voice:act+obl:pcase:~>
<trans:yes+voice:pass+obl:pcase:~>}{V/}{! {#
obl/}}{PP/}{/!}{/"}
{" : VP <trans:yes+voice:pass+obj:~+obl:pcase:~>}{V/}{PP/}{!
{# obl/}}{PP/}{/!}{/"}
```

```
{" : PP }{P/}{! {# obj/}}{NP/}{/!}{/"}
```

Again, let us focus on one definition version to see how we go from an ICFG production rule to an MMP macro definition. Consider the ICFG phrase structure rule given in (29).

(29) $VP \xrightarrow[\langle \text{trans:yes+obj+voice:act} \rangle]{} V\ NP[\text{obj}]$

In MMP, this rule becomes the macro definition:

(30)
```
{" : VP <trans:yes+voice:act+obj:~>}{V/}{! {#
obj/}}{NP/}{/!}{/"}
```

In the case of our non-lexical ICFG rules, there are nonterminals in the body of the rule. This translates to having macro calls in the body of our macro definitions for these types of rules. For this reason we use the `"` macro in definition (30) (as well as the rest of the non-lexical definitions). Thus, upon encountering this particular definition of the `{VP/}` macro, the processor will not expand on the calls to `{V/}`, `{! {# obj/}}`, and `{NP/}` until a call to this macro itself is encountered.

Note furthermore, that rule (29) makes use of the drill-down operator, $NP[\text{obj}]$. In the corresponding macro definition, this is accomplished by the subsequent macro calls `{! {# obj/}}{NP/}{/!}`. Recall, the system macro `!` will temporarily replace the current context with the version expression given in its first argument. This change lasts until the MMP processor encounters the closing tag `{/!}`. In the first argument we also use the system macro `#` which returns the full versioned value of the given dimension, in this case `obj`. Thus, the current context becomes the value of `obj` as long as we are deriving the subtree rooted at the object NP node in our derivation tree, precisely the function of the drill-down operator of ICFG.

Looking closely at (54) of Chapter 6 and (28) above, one will probably note that one rule has been slightly changed. In (54), the second version of the sentence rule is:

(31) $S \xrightarrow[\langle voice{:}pass \rangle]{\langle tense{:}fut \rangle} NP[subj]\ Aux\ VP$

The corresponding macro definition is:

(32)
```
{" : S <tense:fut> <voice:pass> <tense:fut+voice:pass>}{! {#
subj/}}{NP/}{/!}{Aux/}{VP/}{/"}
```

In the macro definition we have included an extra, redundant version of the same rule, tagged by `<tense:fut+voice:pass>`.

This inconsistency arises from a difference in the theory versus the implementation. MMP is in the midst of ongoing development, independent of the ideas presented here. In the grammar, when more than one maximal rule exists during a derivation, the derivation continues nondeterministically, much like any context-free grammar. The MMP processor on the other hand, regards this situation as an error and stops all expansion at that point.

In the example grammar, rule (31) is the only one where this problem arises. In any case where the current context contains the subexpression `<tense:fut+voice:pass>`, we should be able to apply the rule based on the fact that either expression, `<tense:fut>` or `<voice:pass>`, refines to `<tense:fut+voice:pass>` maximally. In MMP though, this situation is viewed as two different rules both refining maximally, so instead of making a choice the processor just stops. We add the `<tense:fut+voice:pass>` version of the rule and avoid this conflict altogether. Going forward, this inconsistency will need to be remedied to fully develop a proper ICFG implementation.

7.2.3 sentence.mmp

Given the set of macro definitions presented above in (27) and (28), we still need a means to generate the strings of our language. Recall that an intensional context-free language is an indexed set of context-language pairs, where the language is the set of strings derived in the given context. We use MMP to generate the strings, via macro expansion, given the context. So, calls to the processor have the following form:

(33)
```
{[]
    <eat+tense:pres+trans:no+voice:act+subj:<dog+num:pl+def:no>>/}
    {S/}
```

What you see here are two macro calls; the first, `[]`, sets the current context, denoted C in our grammar, to `<eat+tense:pres+trans:no+voice:act+subj:<dog+num:pl+def:no>>`. The second macro call, `{S/}`, begins the expansion of the corresponding string (actually, set of strings but, as stated above, MMP is not yet capable of that.)

Let us follow the macro calls given in (33), and compare them to the corresponding derivation. The first call sets the current context, as stated above. The call to the `{S/}` macro expands to

(34) `{! {# subj/}}{NP/}{/!}{VP/}`

in the current context. The call to `{! {# subj/}}` evaluates the `{# subj/}` call first as the inner-most call. The result of this is

(35) `{! <dog+num:pl+def:no>}{NP/}{/!}{VP/}`

which evaluates to

(36) `{NP/}{/!}{VP/}`

with the side effect of changing the current context to `<dog+num:pl+def:no>`.

Next we have

(37) {N/}{/!}{VP/}

followed by

(38) dogs {/!}{VP/}

and then

(39) dogs {VP/}

with the current context being set back to

<eat+tense:pres+trans:no+voice:act+subj:<dog+num:pl+def:no>>

Now, we expand {VP/}, giving us

(40) dogs {V/}

and finally

(41) dogs eat

In the derivation notation of Chapter 3, this expansion looks like the following.

(42) {S/} $\underset{C}{\Rightarrow}$ {! {# subj/}}{NP/}{/!}{VP/}
$\underset{C}{\Rightarrow}$ {! <dog+num:pl+def:no>}{NP/}{/!}{VP/}
$\underset{\text{<dog+num:pl+def:no>}}{\Longrightarrow}$ {NP/}{/!}{VP/}
$\underset{\text{<dog+num:pl+def:no>}}{\Longrightarrow}$ {N/}{/!}{VP/}
$\underset{\text{<dog+num:pl+def:no>}}{\Longrightarrow}$ dogs {/!}{VP/}
$\underset{C}{\Rightarrow}$ dogs {VP/}
$\underset{C}{\Rightarrow}$ dogs {V/}
$\underset{C}{\Rightarrow}$ dogs eat

Compare this to the actual derivation:

(43) $S \underset{C}{\Rightarrow} NP[\text{subj}]\ VP$

$\underset{\text{<dog+num:pl+def:no>}}{\Longrightarrow} N\ VP$

$\underset{\text{<dog+num:pl+def:no>}}{\Longrightarrow} dogs\ VP$

$\underset{C}{\Rightarrow} dogs\ V \underset{C}{\Rightarrow} dogs\ eat$

7.3 Machine Translation

An obvious application of any Natural Language Processing system is to *Machine Translatio*n (*MT*). The basic idea in MT is the use of a computer to take a sentence in some language (the source language) as input and translate it into some other language (the target language) as output. The goal of such a system is to not just translate each word from source to target language, but to maintain the original meaning of the source sentence while taking into account differences in the two languages such as word order, subcategorization, agreement properties, etc. Generally, there are two types of approaches to Machine Translation being employed currently; *direct translation* and *indirect translation*[33].

7.3.1 Direct Machine Translation

Direct translation methods are the oldest form of MT, and they consist of directly translating the source sentence into the target sentence. There are a few different ways to accomplish this. The easiest but least reliable way is a direct word for word translation. Of course, this approach has many problems. Not all languages have words with the same meaning. Furthermore, word order is rarely the same between any two languages.

A more useful direct approach is called the example-based method. In example-based systems, there are bilingual text corpora consisting of a collection of example sentence pairs and their translations. These pairs of sentences are then used as examples to compose other translations, sort of like a template. For example, consider the pair of sentences given in (44).

[33] All the definitions and terminology in this section are taken from *An Introduction to Machine Translation* [46].

(44) $The\ dog\ chased\ the\ cat\ =\ Le\ chien\ a\ chassé\ le\ chat$
$The\ woman\ chased\ the\ child\ =\ La\ femme\ a\ chassé\ l'enfant$

This pair of sentence translations provides us with a few pieces of vital information that can be useful in other translations. For instance,

(45) $X\ chased\ Y = X\ a\ chassé\ Y$
$The\ dog = Le\ chien$
$The\ cat = Le\ chat$
$The\ woman = La\ femme$
$The\ child = l'enfant$

The third, and most common direct approach currently, is the statistical method. Statistical methods of machine translation also use bilingual (or multilingual) text corpora, to provide a knowledge base for translation. In the statistical method though, different statistical models[34] are used to assign probabilities to the possible translation pairs. So, when a sentence is given as input the pair with the highest probability is deemed the likely translation and is given as the output[35].

The direct approaches seem to benefit from being simpler to model, more efficient in terms of processing speed and more robust in that they can process a larger range of sentence types. On the other hand, they can be cumbersome in terms of memory use and are less reliable. That is, they tend to have higher error rates compared to the indirect methods we discuss below.

7.3.2 Indirect Machine Translation

Indirect (or rule-based) translation methods make use of linguistic knowledge, in the form of formal grammars, in order to translate from one sentence to another. The basic idea is that you have two grammars, the source grammar and the target grammar

[34] For example, Hidden Markov Models.
[35] Google Translate is an example of statistical MT.

(corresponding to the source and target sentence.) You use the source grammar to parse a sentence and store the information gained from the parse in some intermediate structure. That information usually includes not only the categorization of the words but also the subcategorization information and possibly semantic information as well. You then generate the target sentence using this structure and the target grammar.

Within the indirect paradigm there are two main types of systems; those that use the *interlingual method* and those that use the *transfer method*. In the interlingual method the intermediate structure, called the *interlingua*, is language independent and is used directly by both the source and target language. So, once the source grammar derives the interlingua, then the target grammar takes the interlingua and generates the target sentence, without change to its contents.

With the transfer method, there are actually two intermediate structures, one for the source sentence and one for the target sentence. So, you parse the sentence with the source grammar to produce the source intermediate structure. You then *transfer* the information from the source structure into a corresponding target structure, not necessarily containing exactly the same information. Finally, you generate the target sentence from the target structure using the target grammar.

It is evident that the indirect approaches are more complex, involving more linguistic knowledge, but when applied are more reliable. Furthermore, within the indirect approach there is a relatively similar trade-off between interlingua and transfer. The ultimate goal is the interlingua approach, where the intermediate structure between any two languages is fully independent of those languages. This trade-off has been

famously visualized in the linguistics world by Bernard Vauqois with the Vauqois triangle, shown in Figure 4 - Vauqois' Triangle.

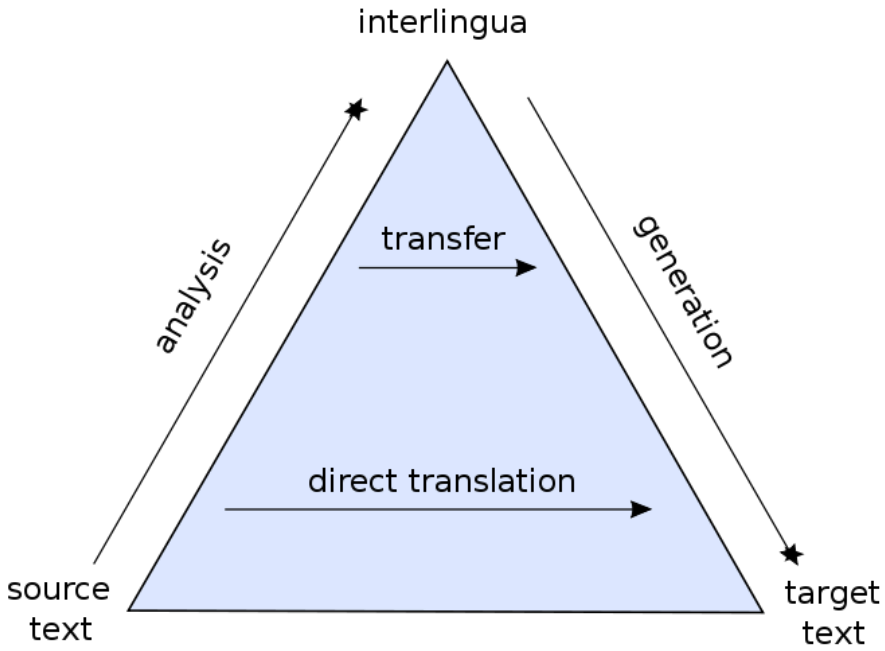


Figure 4 - Vauqois' Triangle

7.3.3 Machine Translation with ICFG

Theoretically, the interlingual method is what computational linguists strive for. A purely interlingual method, with an intermediate structure that is independent from languages, would be much more flexible than any other method described. You could use the same structure to translate between any two languages. This would remove the need to store huge text corpora for multiple languages. It would also remove the need to generate multiple intermediate structures, one for each language in question. To date, the problem with the interlingua has been abstracting away all language dependent information. What happens is you must over specify the interlingua. The more languages your system

incorporates, the more over specified the intermediate structure becomes. So, the transfer method is currently far more popular within the indirect paradigm.

It occurred to this author that ICFG offers a system for Machine Translation that is closer to the interlingual method than the transfer method, but that does have components of both. In the ICFG translation system you would parse the source sentence with the source grammar[36], building the current context for the sentence from the version tags along the way. We would then transfer some components of the current context, but only those that would be necessary to continue. We would not be creating a new structure, just altering the current structure. We would then use the same (possibly modified) current context to generate the target sentence from the target grammar.

Minimally, we would only need to change the language dimension and the root words for the major objects in the sentence (subject-verb-object). In some cases we may need to add a dimension or so, but we would rarely, if at all, need to remove dimensions by taking advantage of the fact that the best-fit algorithm is not concerned with over specification. To illustrate these ideas we have implemented one half of a French-English translation system in our grammar, for a small subset of the English sentences that the grammar can generate.

Currently, the grammar does not have a parser, or French rules, so we have created a small list of French sentences (the active sentences with $chase - dog - cat$ as verb-subject-object) and their context as a version expression. The list is set up as a switch, using the `c:` macro, and simulates a French parser. So, given one of the accepted French sentences, we find it in the list, change the current context and then call the transfer component. The transfer component translates the French words to their English

[36] Actually, the source version of the grammar, but we will talk more about this in the final chapter.

counter-parts in the current context, modifies the new `lg` dimension from `fr` to `en`, then calls our English sentence generator.

The switch has forty-eight sentence-context pairs but here we will present just a few of them to give you an idea of how it works. We have defined a `translate` macro, (46), whose body contains the switch.

(46)
```
{" : translate}{c: "{_/}"}
    {" , "le chien chasse le chat"}
           {[] <chasser+tense:pres+trans:yes+voice:act+lg:fr+
                subj:<chien+num:sg+def:yes+gen:ml>+
                obj:<chat+num:sg+def:yes+gen:ml>>/}
           {contrans/}{S/}{/"}
    {" , "un chien chasse le chat"}
           {[] <chasser +tense:pres+trans:yes+voice:act+lg:fr+
                subj:<chien+num:sg+def:no+gen:ml>+
                obj:<chat+num:sg+def:yes+gen:ml> >/}
           {contrans/}{S/}{/"}
    {" , "le chien chasse un chat"}
           {[] chasser+tense:pres+trans:yes+voice:act+lg:fr+
                <subj:<chien+num:sg+def:yes+gen:ml>+
                obj:<chat+num:sg+def:no+gen:ml> >/}
           {contrans/}{S/}{/"}
    ⋮
    {" , "des chiens chasseront des chats"}
                 {[] chasser+tense:fut+trans:yes+voice:act+lg:fr+
                    <subj:<chien+num:pl+def:no+gen:ml>+
                    obj:<chat+num:pl+def:no+gen:ml> >/}
           {contrans/}{S/}{/"}
    {, default}We cannot translate this sentence{/,}
   {/c:}{/"}
```

We use the `translate` macro to initiate the translation process, by placing a French string in the body of a `translate` call. For example, the call given in (47) will translate the French sentence `le chien chasse le chat` to the English `the dog chases the cat`.

(47) `{translate}le chien chasse le chat{/translate}`

This call initiates the switch, which creates the current context given in (48), using the `[]` macro.

(48) `<chasser+tense:pres+trans:yes+voice:act+lg:fr+subj:<chien+num:sg+def:yes+gen:ml>+obj:<chat+num:sg+def:yes+gen:ml>>`

We then call the `contrans`[37] macro, defined in (49), followed by the `S` macro which starts our generator.

(49)
```
{" :
contrans}{langtrans/}{verbtrans/}{subjtrans/}{objtrans/}{/"}
{" : langtrans <lg:fr>}{[] [lg:en]/}{/"}
{" : subjtrans <subj:chien>}{[] [subj:dog]/}{/"}
{" : subjtrans <subj:chat>}{[] [subj:cat]/}{/"}
{" : objtrans <obj:chien>}{[] [obj:dog]/}{/"}
{" : objtrans <obj:chat>}{[] [obj:cat]/}{/"}
{" : verbtrans <chasser>}{[] [chase]/}{/"}
```

The `contrans` macro is where we do the transfer step, which, as was stated before, is simply a modification of the current context. Note, in the current context associated with the French sentence, we have a new dimension `gen:{ml,fm}`, which is associated with nouns and contains the gender of the noun. This is a property that English proper nouns do not have, and as such we do not need this dimension. But, the best-fit algorithm ignores this dimension when generating the English sentences and thus we do not have to do anything to it.

7.4 Conclusion

Of course, translation between languages is not quite as simple as we have let on here with our example. For instance, going from English to French would introduce the need to add to the context as we transfer. We would have to give the gender dimension a value, but that value would be in the lexicon in the context tag of the corresponding French word, so it would not be hard to get.

[37] For *con*text *trans*lation.

Beyond that there are situations that arise in translation where whole components of the context space would need to be altered. But the bottom line is, it can be done. We have the tools for modifying context without having to create a whole new structure. And in that regard we are moving closer to a true language interlingua structure.

8 Conclusion

At the outset, the intention of this dissertation was to develop a new grammar formalism for use in natural language processing, based on the principles of intensional logic. Our claim was that this grammar, which we called Intensional Context-Free Grammar, was a more realistic model for sentence parsing and generation. I believe that throughout the course of this work we have met and exceeded this goal in a number of ways.

First, we have successfully developed our intensional context-free grammar, incorporating intensional versions in a context-free rewrite system. We have shown through our definition of derivation that, as was claimed, sentence derivation is encapsulated in the context space and not vice-versa. The only time that context is seemingly part of a derivation, it is only as a marker for changes in the current context through context modifiers. Once those changes are instituted the markers disappear from the derivation. In comparison to the three well-known grammar formalisms we surveyed in Chapter 1, this seems a more realistic psychological model of language. Something that is very important to the understating of not only the processes of the brain in language but in all cognitive processes in general. In the psycholinguistic community there is a belief that the discovery of a realistic language model of the brain will lead to realistic overall cognitive models [6].

Based on studies of language use and language errors, it is generally believed in the psycholinguistics community that speech production has four components; conceptualization of the message, forming a plan to convey that message, implementing that plan, and monitoring for errors [46]. It is further believed by a majority of the community that this process proceeds in exactly that order. There are four traditional

models of speech production, three of which apply this linear order. Let's focus on the most recent model to compare it to our grammar.

The Bock and Levelt Model [47] proposes four levels of processing; message, functional, positional and phonological. The message corresponds to conceptualizing the idea you want to convey. It is holistic and instantaneous and produces output which is passed on to functional processing. Functional processing incorporates two jobs, selecting the appropriate lexical content words and assigning the grammatical roles for those words. This provides a tangible framework for the message you want to convey. Positional processing uses the framework provided by functional processing to assign word order and incorporate affixes and inflectional information. Phonological encoding entails the process of preparing the message you have built in the first three stages to be spoken. This includes picking the right phonological components and intonations needed to actually articulate the message verbally.

Focussing on the middle two levels, functional and positional, you can see that these parallel the process that Intensional Context-Free Grammar implements. Once we formulate an idea to convey, we create a current context in the form of a version expression. In this version expression we include the base words and grammatical roles, such as subject, object, etc. Based on this context we then derive the sentence through the re-write rules, producing the proper word order while incorporating the functional words such as determiners and prepositions. Like the Bock and Levelt Model, and unlike other grammars, in ICFG these two components worked serially as we have shown throughout the dissertation.

We also made some other exciting and important observations throughout our researches. One very interesting result from Chapter 4 is that the language of an intensional context-free grammar is actually an indexed set of context-free languages. Thus, with one intensional grammar we encompass many (in fact infinitely many) context-free grammars, each indexed by a point in the version space, which is at the heart of the intensional paradigm. The implications of this go beyond our aspirations in this dissertation, of a viable generative grammar in natural language processing. It is clear that this could lead to a new hierarchy in which we classify intensional languages of every type, not just the intensional context-free languages.

We discovered in Chapter 7 that intensional context-free grammar is a promising candidate for a true interlingual machine translation system. With a very small amount of work we illustrated ICFG's natural capability in this area. By simply taking advantage of the flexibility of the version space with the introduction of one new dimension for language, we can go from one language to another by modifying this dimension.

In Chapter 1 we saw that one of the original motivations of Chomsky's work was that of the Universal Grammar. Chomsky reputed the long standing behaviorist idea that language is entirely learned. Before transformational grammar, it was widely accepted that when born your mind is a completely clean slate and that all learning, including language, is just a collection of learned behaviors based on outside stimulus. Chomsky argued that this could not possibly be based on a number of factors, including some that we mentioned in Chapter 1. This view has become very popular in the linguistic and psycholinguistic communities in the decades since[38].

[38] Not entirely without its opponents though.

Many generative grammars that have been proposed have had aspirations of modeling the Universal Grammar. That is, they try to separate the rules of the grammar into those which are language independent (the universal part) and those that are language specific. It is in this area again that we feel the intensional context-free grammar has great potential. I envision a grammar in which the collection of vanilla versions of the rules are the kernel or universal portion of the grammar and that language specific rules are simply tagged by the language dimension. In fact, we can go beyond this as the context space lends itself naturally to having levels of related rules. That is, at the core we have the universal grammar rules, then we have a series of rules shared by many related languages, and on and on until at the periphery we have a collection of rules specific to each language, all easily done in ICFG.

8.1 Future Work

Obviously, the work done in this dissertation spans many disciplines, including computer science, mathematics, linguistics, and psychology. So, it should be no surprise that avenues are open in each for future work. Mathematically, there is work to be done expanding on the ideas of Chapter 4, namely, investigating other applications of intensionality to grammars. For instance, we could develop definitions for intensional regular grammars, intensional context-sensitive grammars and intensional unrestricted grammars and explore the relationship between the languages of each. It is possible that this leads to a hierarchy of intensional languages, corresponding to Chomsky's hierarchy.

Linguistically, there is obviously some work to be done expanding on the current grammar. There are other phrasal categories, sentence types and complex dependencies to account for as well as many more words to add to the lexicon. There can also be work

done on adding a semantic[39] component to the grammar. The semantics would also be represented by dimensions in the version space, thus the overall structure of the grammar would not change. Furthermore, there is much work to be done adding other languages to the grammar. Along with this, more thought and work needs to be done on ICFG as a translation system. My investigation into this in Chapter 7 was very brief. There definitely needs to be more thought and work done in this area, but the potential seems very fruitful.

One avenue that seems very interesting to me for a translation system is the æther [36]. If you will recall, the æther is an environment that is a combination of a context space and a number of participants registered to that context space. The context space does not only receive modifications from a user but can also inform the contexts of other users. Essentially, it is a communication system in which all changes in context filter through the context space. This seems to be exactly what could be used for an interlingual translation system. In this way you can have communication between many users of many languages all filtered through the context space.

Computationally, to implement a full translation system there needs to be work done on a parser. Of course, there are well known parsing techniques out there for context-free grammars that can be explored. The bulk of the work here will be in building a context for the sentence being parsed, particularly for translation. It is in the context expression where the translation will occur. This would include dealing with ambiguity, something that was not dealt with in this dissertation. All of the working grammars that I

[39] In the linguistics sense.

came across in my researches integrate statistical methods into the grammar to deal with ambiguity. I suspect that we would be no exception.

Finally, the choice of MMP for implementation was governed by the fact that it was the only tool available with built-in intensionality capabilities. To be sure, it has been very useful and convenient as a means for illustrating the usefulness of ICFG as an English grammar. But, MMP has its limitations, one of which is very important to the further development of ICFG. In MMP, there cannot be more than one maximal version of a rule. If at any time in a derivation this occurs, then MMP returns an error message and stops. Obviously, this does not correspond to the theoretical definition of derivation in ICFG. There needs to be work done on making a true implementation of ICFG.

Work can be done on MMP itself to change this and it may be in the future. Another possibility that has occurred to me is to develop an intensional interface to a Prolog system. Prolog is a programming language that was developed specifically to implement definite clause grammars, which are context-free grammars with variable binding. As such it may be well suited for implementing intensional context-free grammars. It may be worth developing an intensional Prolog interpreter but this is not something I have actually looked into in any detail.

Bibliography

[1] N. Chomsky, "Three models for the description of language," in *I.R.E. Transactions on Information Theory, September*, 1956.

[2] N. Chomsky, Syntactic Structures, The Hague: Mouton & Co.'s - Gravenhage, 1957.

[3] N. Chomsky, Aspects of the Theory of Syntax, Cambridge, Massachusetts: MIT Press, 1965.

[4] N. Chomsky, Lectures on Government & Binding, Dordrecht: Foris Publications, 1981.

[5] J. P. S. Peters and R. W. Ritchie, "On the Generative Power of Transformational Grammars," *Information Sciences*, vol. 6, pp. 49-83, 1973.

[6] J. A. Fodor, M. F. Garrett and T. G. Bever, The Psychology of Language: An Introduction to Pyscholinguistics and Generative Grammar, New York: McGraw-Hill, 1974.

[7] J. Bresnan, "A realistic transformational grammar," in *Linguistic Theory and Psychological Reality*, M. Halle, J. Bresnan and G. A. Miller, Eds., Cambridge, Massachusettes: MIT Press, 1978, pp. 1-59.

[8] J. Bresnan, Ed., The Mental Representation of Grammatical Relations, Cambridge, Massachuesettes: MIT Press, 1982.

[9] C. Pollard and I. A. Sag, Information-Based Syntax and Semantics, Volume 1: Fundamentals, Stanford: CSLI Publications, 1987.

[10] G. Gazdar, E. Klein, G. Pullum and I. Sag, Generalized Phrase Structure Grammar, Oxford: Basil Blackwell Publisher Ltd, 1985.

[11] R. Carnap, Meaning and Necessity: A Study in Semantics and Modal Logic, 2nd Edition ed., Chicago: University of Chicago Press, 1956.

[12] J. Hintikka, "Carnap's Heritage in Logical Semantics," in *Rudolf Carnap, Logical Empiricist*, J. Hintikka, Ed., Dordrecht, D. Reidel Publishing Company, 1975, pp. 217-242.

[13] S. A. Kripke, "A Completeness Theorem in Modal Logic," *Journal of Symbolic Logic*, vol. 24, pp. 1-14, 1959.

[14] S. A. Kripke, "Semantical Analysis of Modal Logic I. Normal Modal Propositional Calculi," *Mathematical Logic Quarterly*, vol. 9, no. 6, pp. 67-96, 1963.

[15] S. A. Kripke, "Semantical Analysis of Modal Logic II. Non-normal Modal Propositional Calculi," in *The Theory of Models*, Amsterdam, 1965.

[16] S. A. Kripke, Naming and Necessity, Cambridge, Mass.: Harvard University Press, 1980.

[17] D. Scott, "Advice on Modal Logic," in *Philosophical Problems in Logic*, K. Lambert, Ed., Dordrecht, D. Reidel Publishing Company, 1968, pp. 143-173.

[18] R. Montague, Formal Philosophy: Selected Papers of Richard Montague, R. H. Thomason, Ed., New Haven: Yale University Press, 1974.

[19] W. W. Wadge and E. A. Ashcroft, Lucid, The Dataflow Programming Language, London: Academic Press, 1985.

[20] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305-1320, September 1991.

[21] A. A. Faustini and W. W. Wadge, "Intensional Programming," in *The Role of Languages in Problem Solving 2*, Elsevier, North Holland, 1987.

[22] A. A. Faustini and R. Jagannathan, "Indexical Lucid," in *Proceedings of the 4th ISLIP*, Meno Park, 1991.

[23] R. Jagannathan, C. Dodd and I. Agi, "GLU: A High-Level System for Granular Data-Parallel Programming," *Concurrency: Practice and Experience*, vol. 9, no. 1, pp. 63-83, January 1997.

[24] W. Du and W. W. Wadge, "The educitve implementation of a three-dimensional spreadsheet," *Software-Practice and Experience*, vol. 20, no. 11, pp. 1097-1114, 1990.

[25] W. Du and W. W. Wadge, "A 3D spreadsheet based on intensional logic," *IEEE Software*, vol. 7, no. 3, pp. 78-89, 1990.

[26] S. Tao, "TLucid and Intensional Attribute Grammars," in *Proceedings of the 6th ISLIP*, Quebec, 1993.

[27] J. Paquet, Intensional Scientific Programming, PhD Dissertation, Quebec City, Quebec: Laval University, 1999.

[28] G. Ditu, The Programming Language TransLucid, PhD Dissertation, Sydney: University of New South Wales, 2007.

[29] J. Plaice, B. Mancilla and G. Ditu, "From Lucid to TransLucid: Iteration, Dataflow, Intensional and Cartesian Programming," *Mathematics in Computer Science*, vol. 2, no. 1, pp. 37-61, November 2008.

[30] A. A. Faustini, "An Operational Semantics for Pure Dataflow," in *Automata, Languages and Programming*, M. Nielsen and E. M. Schmidt, Eds., 1982, pp. 212-224.

[31] J. Plaice and W. W. Wadge, "A New Approach to Version Control," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 268-276, March 1993.

[32] T. Yildirim, Intensional HTML, Master's thesis, Victoria: University of Victoria, 1997.

[33] G. D. Brown, Intensional HTML 2: A Practical Approach, Master's thesis, Victoria: University of Victoria, 1998.

[34] P. Swoboda, Practical Languages for Intensional Programming, Master's thesis, University of Victoria, 1999.

[35] W. W. Wadge, "Intensional Markup Language," in *Lecture Notes in Computer Science: Distributed Communities on the Web, Third International Workshop, DCW 2000, Quebec City, Canada, Proceedings*, Quebec, June 19-21, 2000.

[36] P. Swoboda, A Formalization and Implementation of Distributed Intensional Programming. PhD Dissertation, Sydney: University of New South Wales, 2003.

[37] W. W. Wadge and m. c. schraefel, "A Complementary Approach for Adaptive and Adaptable Hypermedia: Intensional Hypertext," 2001.

[38] J. Plaice, P. Swoboda and A. Alammar, "Building Intensional Communities Using Shared Context," in *Distributed communities on the Web: Third International Workshop*, Quebec City, 2000.

[39] B. Mancilla, Intensional Infrastructure for Collaborative mapping, PhD Dissertation, Sydney: University of New South Wales, 2004.

[40] J. Plaice, "Multidimensional Lucid," in *Distributed communities on the Web: Third International Workshop*, Quebec City, 2000.

[41] V. S. Alagar, J. Paquet and K. Wan, "Intensional Programming for Agent Communication," in *Declarative Agent Languages and Technologies II*, vol. 3476, J. Leite, A. Omicini, P. Torroni and P. Yolum, Eds., New York, NY: Springer-Verlag, 2005, pp. 239-255.

[42] S. Tao, Indexical Attribute Grammars, PhD Dissertation, Victoria: University of Victoria, 1994.

[43] W. W. Wadge and C. Wadge, "Multidimensional French," in *Proceedings of the Eleventh Annual International Symposium on Language for Intensional Programming*, Palo Alto, California, USA, May 1998.

[44] H. Li, An Intensional Tool Applied in French Language Educational Software, Master's thesis, Victoria: University of Victoria, 2003.

[45] B. A. Davey and H. A. Priestley, Introduction to Lattices and Order, Second Edition, Cambridge: Cambridge University Press, 2002.

[46] T. B. Jay, The Psychology of Language, Upper Saddle River, New Jersey: Prentice Hall, 2003.

[47] K. Bock and W. Levelt, "Language production: Grammatical encoding," in *Handbook of Psycholinguistics*, San Diego, Academic Press, 1994, pp. 945-984.

[48] W. J. Hutchins and H. L. Somers, An Introduction to Machine Translation, San Diego, CA: Academic Press Inc., 1992.

Appendix – MMP Documentation

The following text is a direct copy of an unpublished MMP manual written by Bill Wadge and Rich Little:

MMP is a general purpose Macro Processor in the tradition of TRAC, M6, the troff/groff macro facility and others. It accepts an input text document and produces a corresponding output text document which results from copying ordinary text unchanged while expanding macro calls.

MMP differs from other macro processors in the form that macro calls take: they look like XML elements, using a slightly changed and vastly simpler syntax. Expanding MMP macros can therefore be thought of as implementing user-defined tags.

MMP differs from XML in that tags are marked using curly parentheses (`{` and `}`) rather than angle brackets. Also, attributes are not named; instead, the order in which they appear determines their significance (for which reason they are called *arguments*).

Here is a typical MMP macro call:

```
{memo Tom Dick}I received your proposal.{/memo}
```

This is a call to a macro called `memo`. The first argument (actual parameter) is `Tom` and the second argument is `Dick`. The text between the opening and closing tags (`I received your proposal.`) is called the *body* of the call.

Once the MMP processor has recognized and parsed a macro call, it replaces the entire call its *value* or *result*. Typically (but not always) this value is defined by a template into which the arguments and body are substituted. For example, the above call might produce

```
MEMO
```

```
From: Tom

To: Dick


I received your proposal.
```

Notice that the result has newlines in it – MMP is not line oriented, and newlines play no special role (other than as whitespace – see later). Templates can have embedded newlines as can bodies and even arguments.

Even though the MMP calling format is based on the XML syntax, there is no possibility of confusing the two – angle brackets have no special status at all for MMP. This makes it easy to define macros that produce XML markup. In fact that is one of the major motivations.

As with XML, MMP allows calls consisting of a single tag and no body. For example, the call

```
{dept Physics/}
```

might expand to

```
Physics Department of the University of Victoria
```

Although single tag calls have no bodies, the can still have arguments, as in the example.

Evaluation

MMP macro calls, like XML elements, can be nested. For example,

```
{memo Tom Dick}I received a spreadsheet from
the {dept Physics/}.{/memo}
```

Whenever calls are nested, the question arises, in what order are the calls evaluated. In the example we could expand the call of “dept” first, so that the call to “memo” would have

```
I received a spreadsheet from
 the Physics Department of the University of Victoria
```

as its body. Alternatively, we could first expand the “memo” macro with

```
I received a spreadsheet from the {dept Physics/}.
```

as its body, then expand any occurrence of `dept` in the resulting text.

In this example, the end result will be the same, and this is often the case. Sometimes, however, it will make a difference. MMP avoids any possible ambiguity by *always evaluating inner macros first*, with one exception, which we will immediately explain. Inner-first evaluation happens even if (as is possible) the enclosing macro ignores its body, and therefore ends up discarding the result of the inner evaluation.

Quoting

All macro processors must have some mechanism to force text to be handles verbatim, with macro expansion turned off. This is necessary if text may accidentally contain patterns that MMP would otherwise interpret as (parts of) a macro call. Also, sometimes we need to override the inner-first evaluation rule and delay evaluation until later.

MMP provides a special macro for this purpose. Its name is `"` (a double quote character) and it is called like any other macro; for example

```
{"}Greetings from the {dept Physics/}, everyone.{/"}
```

The quote macro returns its body as its result. However, the body is *not* evaluated first; it is passed on verbatim. The quote macro is the *only* macro with this property. No other built-in macro does this, and it is not possible for users to define their own macros that quote their bodies.

Of course, quoting is not foolproof. If the text we are quoting itself contains the sequence `{/"}`, MMP will be fooled. If this is what we want, we must be ingenious (more later).

It's actually not unusual to want to quote the body of a macro call. As we just said, we cannot define a macro that does this automatically. We simply have to enclose the body in a quote macro:

```
{memo Tom Dick}{"}I received a spreadsheet from
the {dept Physics/}.{/"}{/memo}
```

This can be really cumbersome, so MMP allows quote to be used as a kind of meta-macro:

```
{" memo Tom Dick}I received a spreadsheet from
the {dept Physics/}.{/"}
```

Here quote is being called with three arguments and a body. Calls to quote like this are evaluated as a call to the first argument (treated as a macro name) together with the remaining arguments as the arguments of the implied call. The body of the implied call is the verbatim (quoted) body of the original call. Notice that this is a call to the quote macro (not the memo macro) so that the closing tag is `{/"}`, not `{/memo}` (an easy mistake to make).

Macro definitions

Most macros are defined by substitution templates, each template being a string with 'holes' for the arguments and the bodies. In MMP these holes are special 'system' macros. The macro calls that act as placeholders for the arguments are

```
{1/}  {2/}  {3/}  ...
```

In other words, a call to the macro 1 without arguments or body marks the position where the first argument is to be substituted. Similar calls to macros “2”, “3”, “4” and so on mark substitution points for the second, third, fourth etc arguments. The call `{_/}` to the macro “_” marks where the body is to be substituted.

The definition of an ordinary template macro is therefore simply a string. Users can define their own template macros using a special system macro “:”. This macro is called with one argument and a body. Evaluating such a call to : returns the empty string but has a side effect: it enters the body as the template string for the (macro whose name is) the first argument.

For example, evaluation of the call `{: uvic}University of Victoria{/:}` associates the template string `University of Victoria` with the macro name `uvic`. Evaluation of the call {uvic/} thereafter expands to University of Victoria. The uvic macro can be called with arguments and a body, but they will be ignored, because none of the special system macros just discussed appear in the template for uvic. For example,

```
{uvic Tom Dick}are students at the{/uvic}
```

will also expand to `University of Victoria.`

Suppose instead we wanted the template for uvic to be

```
{1/} and {2/} {_/} University of Victoria
```

so that the last call would expand to

```
Tom and Dick are students at the University of Victoria.
```

We might rush to the keyboard and enter the following call to :

```
{: uvic}{1/} and {2/} {_/} University of Victoria{/:}
```

but this will blow up in our face. As we already explained, apart from " every MMP macro evaluates its body first. This means that the MMP processor will attempt to

evaluate `{1/}` – and fail since there is no complete macro call ready to be evaluated, and hence no first argument. (Nor is there yet a second argument or a body).

Clearly, we want : to take the string `{1/}` verbatim, without expanding it. So we need to quote the body of the call. The direct way to do this is

```
{: uvic}{"}{1/} and {2/} {_/} University of Victoria{/"}{/:}
```

This works fine but is a bit clumsy. The idiomatic way to do this is with a call to " with : and `uvic` as arguments:

```
{" : uvic}{1/} and {2/} {_/} University of Victoria{/"}
```

Quotes are also needed if you want to the template of a macro to contain a nested call meant to be expanded when the macro you are defining is itself called.

For example, suppose you want the macro greeting to have the template

```
Welcome to the Uvic class of {thisyear/}.
```

The call

```
{" : greeting}Welcome to the Uvic class of {thisyear/}.{/"}
```

does the job. If later on you add `{: this year}2007{/:}`, the call `{greeting/}` will expand to

```
Welcome to the Uvic class of 2007.
```

Suppose, however, by mistake you use the call

```
{: greeting}Welcome to the Uvic class of {thisyear/}.{/:}
```

The difference is that MMP will now try to expand the call `{thisyear/}`, and then use the result as part of the template to be associated with greeting. If you haven't already defined the macro thisyear, you'll get an undefined macro error. If you have already defined `thisyear` to be `2007`, everything works fine – at first.

Problems arise, however, in the new year. In January you redefine thisyear to be `2008`; but when `{greeting/}` is called, you get the message from 2007. On the other hand, if you'd used the quoted definition for greeting, it would expand in January to

```
Welcome to the Uvic class of 2008.
```

Parsing the Argument List

As we already indicated, macro calls can have arguments, corresponding to the attributes of XML. MMP arguments, however, are much simpler. The arguments (if any) are listed in the tag following the macro name, separated from it and each other by blanks or other whitespace characters (newlines are treated as whitespace). Arguments are ordered, not named, and can be arbitrary strings. Arguments can optionally be enclosed in double quotes, and these quotes are necessary for arguments that contain whitespace or any of the characters `{`, `}`, or `/`. Quote characters can also appear in arguments, in which case they must be escaped with `\`.

The following tag illustrates these rules (for the sake of precision the example is not indented):

```
{fred @yg4 "tom" "dick and jane" alpha
        beta "long time
no see" "$%^\"*"/}
```

In this example fred is called with seven arguments. The first five arguments are (one per line, no indentation):

```
@yg4
tom
dick and jane
alpha
beta
```

The sixth argument

```
long time

no see
```

has an embedded newline, and the seventh,

```
$%^"*
```

has an embedded quote.

A macro call can have an arbitrary number of arguments, independent of its definition. If a template macro is called with more arguments than are referred to in its template, the extra ones are ignored. If not enough are supplied, an evaluation-time error will result. The only syntax checking done by MMP is to ensure that individual tags are well formed, and that opening and closing tags match (have the same name – are properly nested). There are no MMP DTDs.

Evaluation of tags

We've already seen that the MMP processor evaluates the body of a macro before it evaluates the macro itself. The same is true of the tags that form the macro call, though we have yet to see an example.

Suppose that Tom wants to send a thank-you memo to Brad Steel, Dean of Engineering. Using the memo macro defined above, he can write

```
{memo Tom "Brad Steel"}Thank you for the visit.{/memo}
```

But suppose Tom defines a macro for the Dean of Engineering:

```
{: DE}Brad Steel{/:}
```

Can Tom use this macro for the argument of the memo macro? Yes, he simply writes

```
{memo Tom "{DE/}"}Thank you for the visit.{/memo}
```

The MMP processor will encounter the call to the `DE` macro while parsing the opening tag of the call to memo. When is does, it will suspend the parsing and evaluate

the call, then resume the parsing, consuming initially the characters produced by this macro expansion.

Notice that the quote symbols do *not* disable the expansion of the call to DE. MMP distinguishes between quote symbols and a call to the quote macro. A call to the quote macro will disable expansions in its body. However, quote symbols serve only to prevent embedded whitespace from acting as an argument separator.

If Tom does not enclose the call to DE in quote symbols, as follows

```
{memo Tom {DE/}}Thank you for the visit.{/memo}
```

then expanding the call to DE results in

```
{memo Tom Brad Steel}Thank you for the visit.{/memo}
```

In this call the second argument is Brad and the third argument (which memo ignores) is Steel.

In this case the result is probably not what Tom wants, but sometimes this phenomenon is useful; it allows us invoke a whole series of arguments with a macro call. Suppose that Tom's invite macro generates a series of invitations, one for each argument. Then the call

```
{invite Dick Harry Pam Sally Rajiv/}
```

will generate an invitation to each of Tom's five best friends. If he wants, Tom can define a macro pals by

```
{: pals}Dick Harry Pam Sally Rajiv{/:}
```

and then invite them with

```
{invite {pals/}/}
```

Macros can of course be body- or argument- nested to any depth.

Macro Processing – Call Parsing

It should be clear that with quoting and nesting, the evaluation process is not as simple as it might appear. Therefore we give a somewhat more detailed description of the evaluation algorithm.

As we already explained, MMP (like most macro processors) passes input directly to output until it encounters a macro call. So the real processing starts when MMP finds `{` in its input stream.

When it does, it assumes that the open curly is the first character of an opening or opening/closing tag. It therefore parses the tag, and consumes all the characters that make up the tag (meaning, does not pass them on to the output). The parsing process is simple, because the syntax of tags is simple: an opening curly parenthesis, a name, and then a possibly empty sequence of arguments, each preceded by at least one whitespace character. Names and arguments are strings of characters none of which are whitespace or any of the special characters `{`, `}`, and `/`. Finally, a tag is terminated by either `}` or `/}`.

Names can have `"` embedded but not arguments. However, arguments can be enclosed in `"`s, in which case whitespace characters are taken as part of the arguments. Furthermore, the special characters and `"` can be part of an argument string if they are escaped by `\` (`\` itself can be escaped).

If the tag parser encounters an unescaped `{` (even within a quoted argument), it assumes that it has come across a macro call nested within the tag. It therefore suspends parsing until the call has been evaluated, then resumes parsing at the first character produced by the evaluation. It does *not* evaluate macro calls appearing in these characters. Another way to describe this is that the processor gathers the characters that

make up the tag, evaluating where necessary, then parses these characters without further evaluation.

Once the tag has been evaluated, what happens next depends on whether it is an opening or opening/closing tag (whether it ends in `}` or in `/}`). In the latter case, the processor takes the name and arguments and assembles them into a macro *call* object. These objects wrap up all the data needed to evaluate a call: the name, the arguments, and the body.

In the case of an open/closing tag there is no body, but there is a name and arguments. This call object is not stored, but passed on directly for immediate evaluation (see below). The results of the evaluation are placed at the head of the input stream, and the processing continues.

If the tag is an opening tag, we again assemble the name and arguments into a call object, but we leave a slot open for the body. This *incomplete* call object is placed on the 'incomplete' stack, a stack of incomplete calls whose closing tags have not yet been encountered. We need a stack because macro calls can be arbitrarily body-nested.

Once we stack the incomplete call, we proceed through the body, still evaluating. But we direct the output stream to a separate buffer so as not to mix up data output before we encountered the opening tag (this data will not be part of the body).

Once we have encountered an opening tag, we have to be on the lookout for closing tags, which begin `{/`. These are parsed according to similar rules (they can even have arguments, as explained later). Once we have parsed the closing tag, we assemble the parts (name plus arguments) into an incomplete closing tag. This closing call should

be compatible with the call on the top of the incomplete stack (and will be if the calls were correctly nested).

Once we match opening and closing tags we merge them into a complete call, by combining arguments and adding the contents of the output buffer as the body. Once again, we are ready to evaluate a macro call.

Macro Processing – Evaluation

Macro evaluation is triggered whenever we have a complete call at hand. This can happen immediately, after parsing an open/closing tag, or later, when we finally parse and match a closing tag with a previously parsed and stacked opening tag. In either case we have at hand all the information needed to expand a macro: the name, the arguments, and the body (if there is one).

In either case an object encapsulating all this information is placed on the top of the 'completed' stack and evaluation proceeds. During evaluation the completed call remains on the top of the completed stack and the body, name, and arguments are all available. Of course, if evaluation results in another macro call being parsed, the new call will temporarily cover the older one until the evaluation of the newer one is complete.

The MMP processors first step is to check if the name is that of one of the built-in or system macro. If it is, then that special code is executed. If it is not a built-in macro – if it is user defined – then the processor looks up its definition. This input is copied into the input stream in front of the input currently waiting to be processed. Conceptually, there is a marker placed between the characters of the definition and the rest of the input. Then normal input processing resumes. This step corresponds to entering the body of a subroutine in a conventional procedural language.

The processor refers to the call on top of the competed stack if it encounters any of the special system macros while processing the definition. For example, if it parses and evaluated the call `{3/}` it retrieves the third argument stored in this call and copies these characters into the input stream.

When the processor encounters the marker which indicates the end of a copied definition, it removes the marker but also pops the completed stack. This corresponds to exiting the body of a subroutine in a traditional procedural language.

Useful System Macros

MMP provides a number of built-in or system macros that are executed for their side effects and/or produce results that cannot be obtained from a template. We have already introduced two of them, namely the definition macro `:` and the quote `macro "`. Also, the arguments of a macro call are referred to in its definition by the system macros `1`, `2`, `3`, `4`, …, and its body by the macro `_`.

There are as well more mundane macros for carrying out humble calculations. The plus macro `+` returns the sum of its arguments (which it expects to be numerals). Thus

```
{+ 4 8 3 0/}
```

will evaluate to `15`. The `<` macro determines whether its first argument is less than its second argument, returning 1 if it is and 0 otherwise (1 and 0 play the role of 'true' and 'false' respectively in MMP). In the same way, `=` tests for equality.

Macro Versions

A major innovation of MMP is allowing for macros to have multiple versions. We accomplish this by allowing multiple definitions of the same macro, each having a different tag attached to it. This tag corresponds to the context under which that particular

version of the macro is to be expanded. The tag is actually an argument of the definition macro, which follows the macro's name. Consider the following example:

```
{: ferd <green> <blue>}green and blue{/:}
{: ferd <red>}red{/:}
{: ferd}vanilla{/:}
```

Here, we define the macro `ferd`, which has three versions. If the context under which `ferd` is expanded is `<green>` or `<blue>`, then `ferd` evaluates to `green and blue`. If the context is `<red>`, then `ferd` evaluates to `red`. Under all other possible contexts `ferd` evaluates to `vanilla`. The latter version of `ferd` is called the *vanilla* version, which in effect is the default version of the macro. We define the vanilla version by simply omitting the context tag or by using an empty context tag, `<>`. Note that the vanilla version of a macro is not mandatory.

Context Tags

We use *version expressions* to denote context. It is these version expressions that act as the tags in our macro definitions. At the ground level we have *dimension labels* and *dimension values* that come in pairs separated by a colon, :, e.g. *d*:*v* says that dimension *d* has the value *v*. The version expressions allow for nested dimensions grounded by a value, e.g. *d*:*e*:*v* states that dimension *d* has value represented by the version expression *e*:*v*.

The version expressions are elements of a *version algebra* that has a join operation, denoted +. We use the join operation to combine two versions together to make a new version, provided the two versions are compatible. So, the expression *d*:*v* + *e*:*w* defines the version with two dimensions *d* and *e* with values *v* and *w*, respectively.

The nesting of dimension labels in conjunction with the join operation allows a version to have a single unlabelled value, called the *base value*. Dimension labels distribute over + as follows:

$$d_1{:}\langle v_1 + d_2{:}V_2 + \cdots \rangle = d_1{:}v_1 + d_1{:}d_2{:}V_2 + \cdots,$$

where d_i are dimension labels, v_1 is the base value and V_2 is a version expression.

Context Macros

The MMP interpreter determines the version of the macro to expand under the context in which it is evaluated, or the *current context*. The current context is a background phenomenon that guides macro evaluation, also represented by a version expression. There are special system macros for setting and modifying the current context.

The `[]` (or `!=`) macro allows you to change the current context. The first argument of `[]` is the new current context. This macro can change the current context in one of two ways; you can replace the entire current context with a new context by enclosing the first argument in angle brackets, `< >`, or you can modify particular dimensions of the current context by enclosing the argument in square brackets, `[ ]`.

So, to set the current context to `<green+a:1>` we use the call `{[] <green+a:1>/}`. Then, to modify the value of dimension `a` to `2`, we use the call `{[] [a:2]/}`, changing the current context to `<green+a:2>`.

Furthermore, there exists a `!` macro that allows you to change the current context temporarily. Any macro evaluated in the body of the `!` macro will be expanded under the temporary context, defined by the first argument of the `!` macro. For example, the following (along with the `ferd` definition above)

```
{[] <>/}
before {ferd/}
```

```
{! <green+a:1>}during {ferd/}{/!}
after {ferd/}
```

will evaluate to

```
before vanilla
during green and blue
after vanilla
```

There also exists a system macro, `#`, which returns the value of the current context. A call to `#,` without arguments, returns the entire current context. To see the value of some particular dimension in the current context, you enter the name of the dimension in the first argument of the `#` macro. Any other arguments are ignored, (although it may be useful to allow for a number of particular dimensions to be specified). So, the following

```
{[] <a:1+b:2+c:3>/}
{#/}
{# a/}
{# a b/}
```

will produce

```
<a:1+b:2+c:3>
1
1
```

Version Space

The interpreter's choice of an appropriate macro version is dependent on the *version space* and *best-fit algorithm*, ideas developed by Bill Wadge and John Plaice[40]. Here, Plaice and Wadge introduced version control tools for software development. In their

[40] Plaice, J. and Wadge, W. W. A new approach to version control. *IEEE Transactions on Software Engineering* 19(3):268-276, March 1993.

system, different software components can have multiple versions (like MMP macros). A complete version of a piece of software is formed by taking the most relevant version of each component. The most relevant version of the components is determined by properties of the version space and best-fit algorithm.

The version space is the set of all possible contexts, partially ordered by a *refinement relation*, denoted $\leq$. For versions V and W, $V \leq W$ is read as V *refines* to W and means that W is a more specific version than V. Furthermore, we say $V \leq W$ *maximally* if for any version $U \neq V$ where $U \leq W$, then V does not refine to U.

The refinement relation is transitive, which is to say that if $V \leq U$ and $U \leq W$, then $V \leq W$, where V, U, and W are versions. The empty version, called the *vanilla* version and denoted <>, has no dimensions or values and refines to any version, that is $<> \leq V$ for all versions V. The join operator, +, is the least upper bound induced by $\leq$, meaning $U + V$ is the least upper bound of U and V if and only if for all W such that $U \leq W$ and $V \leq W$, then $U + V \leq W$. The join operation is also idempotent, commutative, associative and has the properties:

$$V \leq V + W,$$
$$\frac{V \leq V' \;\; W \leq W'}{V + W \leq V' + W'}.$$

Points in the version space are the same version expressions described above. In terms of version expressions, the refinement relation is defines as

$$d_1 : V_1 + d_2 : V_2 + \cdots \leq e_1 : W_1 + e_2 : W_2 + \cdots$$

if and only if for each d_i either $V_i = <>$ or $d_i = e_j$ and $V_i \leq W_j$ for some j.

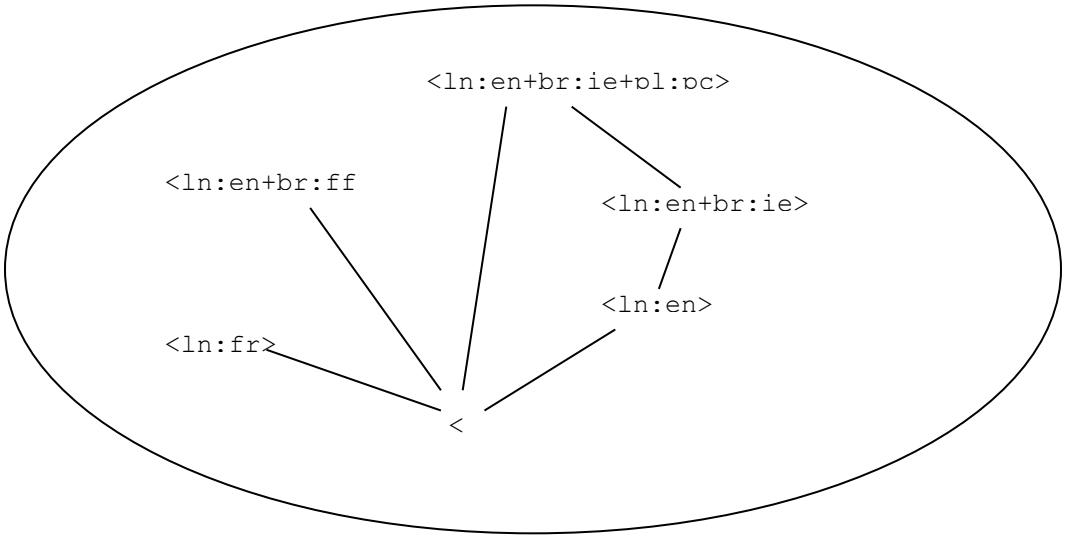
Best-fit Algorithm

The current context exists as a point in the version space. The context tags associated with our macro definitions also represent points in the version space. Using the refinement relation we compare all the versions of a macro definition to each other and to the current context and select the *best-fit* version of the rule. The best-fit version of a rule is a version that refines to the current context maximally. If there is more than one maximal version, then the system produces a failed best-fit error.

To illustrate how the best-fit algorithm works, let us consider the following example, in which we have defined five versions of an image macro to be embedded in a web page. The choice of macro is dependent on the language of the page, the type of browser and the platform of the computer.

```
{: image}<img src=”image1.gif”>{/:}
{: image <ln:en>}<img src=”image2.gif”>{/:}
{: image <ln:fr>}<img src=”image3.gif”>{/:}
{: image <ln:en+br:ie>}<img src=”image4.gif”>{/:}
{: image <ln:en+br:ff>}<img src=”image5.gif”>{/:}
```

Now, suppose that the current context is `<ln:en+br:ie+pl:pc>` and there is a call to the image macro, `{image/}`. To determine the best-fit image, the interpreter views the relationship between the five context tags and the current context as points in the version space (illustrated below with refinement relations denoted by a line).



Next, we throw away any context points that do not refine to the current context. Of the points that are left, if any, we choose the context point the refines to the current context maximally, `<ln:en+br:ie>`. So, in this example the interpreter would produce the output `<img src=”image4.gif”>`.