University of North Dakota
UND Scholarly Commons

Theses and Dissertations | Theses, Dissertations, and Senior Projects

January 2013

A Structured PC-PATR Grammar Editor And Application To Bahasa Indonesia

Trafton Fletcher Hardison

Follow this and additional works at: https://commons.und.edu/theses

Recommended Citation
Hardison, Trafton Fletcher, "A Structured PC-PATR Grammar Editor And Application To Bahasa Indonesia" (2013). *Theses and Dissertations*. 1430.
https://commons.und.edu/theses/1430

This Thesis is brought to you for free and open access by the Theses, Dissertations, and Senior Projects at UND Scholarly Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UND Scholarly Commons. For more information, please contact zeineb.yousif@library.und.edu.

A STRUCTURED PC-PATR GRAMMAR EDITOR AND APPLICATION TO BAHASA INDONESIA

By

Trafton Fletcher Hardison
Bachelor of Arts, Moody Bible Institute, 2009

A Thesis
Submitted to the Graduate Faculty

of the

University of North Dakota

In partial fulfillment of the requirements

for the degree of

Master of Arts

Grand Forks, North Dakota
August
2013

Copyright 2013 Trafton Fletcher Hardison

This thesis, submitted by Trafton Fletcher Hardison in partial fulfillment of the requirements for the Degree of Master of Arts from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done, and is hereby approved.

_____

H. Andrew Black

_____

Cheryl A. Black

_____

Kathryn L. Hansen

This thesis is being submitted by the appointed advisory committee as having met all of the requirements of the Graduate School of the University of North Dakota and is hereby approved.

_____

Wayne Swisher

Dean of the Graduate School

_____

Date

Title	A structured PC-PATR grammar editor and application to Bahasa Indonesia
Department	Linguistics
Degree	Master of Arts

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work, or in his absence, by the Chairperson of the department or the dean of the Graduate School. It is understood that any copying and publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my thesis.

Trafton Fletcher Hardison

3 June 2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

1	first person
2	second person
3	third person
AV	actor voice
EXCL	exclusive
FUT	future
NEG	negation, negative
PASS	passive
PL	plural
SG	singular
TR	transitive

ACKNOWLEDGEMENTS

I wish to thank my advisory committee for their advice and comments during the writing and revision of this thesis. I especially wish to thank H. Andrew Black for his help during the development of PC-PATR-EDITOR, his swift and helpful comments during the writing of the thesis, and his patience with my ignorance of the finer points of XLingPaper, which is a fine way to write a linguistics paper. I would like to thank Cheryl A. Black for her instruction in syntactic theory during my time at North Dakota and I would like to thank Kathryn L. Hansen for agreeing to step in at the last minute to serve on my committee. Thank you also to J. Albert Bickford for his willingness to help organize the comittee and input throughout my graduate studies.

I would like to thank Michael R. Gramley for his expertise and help with Windows Presentation Foundation, the C# programming language, and the Model View ViewModel design pattern. He was invaluable at various points in the project when I needed help figuring out a design or hunting down some glitch in the code.

I would also like to thank my wife, Amy. Without her patience, love, and support this project would never have been completed.

ABSTRACT

This thesis describes the development of a structured editor for grammar files for the PC-PATR syntactic parsing program and an analysis tool for its output called PC-PATR-EDITOR. It describes the difficulties with authoring plain text grammar files and reading the parser's output. The editor solves these problems by providing an editing environment where users do not have to remember the file's syntax, where users can only input content valid at the point of editing, and by processing the parser's output and displaying it in ways familiar to most linguists. The paper contains three case studies that illustrate PC-PATR-EDITOR's usefulness by implementing a grammar fragment for Bahasa Indonesia. It offers critiques and suggestions for future improvements for the editor and analysis tool.

CHAPTER 1

PLACE OF PC-PATR IN THE SYNTACTIC PARSER WORLD

PC-PATR is a unification based syntactic parser built by Stephen McConnel (McConnel 2006) based on the PATR-II formalism described by Shieber (2003). It is a very powerful tool capable of encoding and testing grammars from a variety of theoretical perspectives. PC-PATR is currently a console program that takes plain text files as input. This leads to a number of problems. Users are prone to making errors when encoding their grammars and the parser output is difficult to understand. PC-PATR is capable of producing parse "bushes" when a parse fails and informing the user what failed and where. Unfortunately, this output is difficult to decipher due to the quantity and format of text produced. This renders it inaccessible to linguists without a desire to learn how to read its outputs and renders it tedious to use for those who do learn it.

The PATR-II formalism[1] uses feature structures at each node in the parse tree along with one string combining operation to encode grammars. Features are name-value pairs with either complex or atomic values. Complex values are feature structures. Feature structures may contain zero or more features which may have atomic or complex (feature structure) values. An atomic value is simply a value associated with a feature that contains no child features. In Figure 1, *number* and *person* are features with atomic values[2] as they have no child features. In contrast, *subject* and *agreement* are complex features as they have feature structures for values.

[1] According to Shieber, this formalism is recursively enumerable or a type-0 language on the Chomskyan hierarchy (Shieber 2003).

[2] Hereafter feature values will be called feature(s) unless unclear from context.

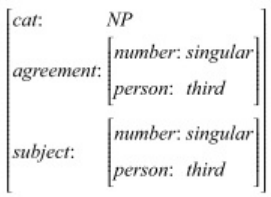

Figure 1. Example feature structure (Shieber 2003:13)

Elements within a feature structure can be selected by means of *feature paths*, which are feature names listed in a given order and surrounded by angle brackets. For example, the path <*agreement number*> would have the value of *singular* in Figure 1.

PATR-II combines feature structures via unification. Unification is an operation where two feature structures can be combined to form a more specific feature structure if and only if both input structures have either the same or no specified value for each feature. Thus if a structure specified that its *number* feature was singular, then it could only combine with other structures whose *number* feature was either singular or unspecified.

The linear order of elements in a sentence is specified in the grammar via context free phrase structure rules. These rules state that the given non-terminal element (on the left hand side) will be the concatenation of the elements on the right hand side in the order they appear. These rules also contain constraints on unification. These state how features from the feature structures associated with the elements of the phrase structure rule must unify.

1.1 Comparison of PC-PATR and other parsers and formalisms

There are many parsers and formalisms that have been developed since computers began to see use in linguistics. PC-PATR's formalism (PATR-II) is not tied to a particular theory of grammar or language. Thus it is useful to linguists from a wide range of perspectives. Unlike PC-PATR other parsers are tied to particular languages or theories of syntax. This section will compare PC-PATR and the PATR-II formalism with a recent formalism, eXtensible MetaGrammar, and three other parsers, the Stanford Parser, Leopar, and the Link Parser.

1.1.1 eXtensible MetaGrammar

XMG (eXtensible MetaGrammar) is a recently developed formalism for developing grammars from multiple theoretical perspectives. According to the XMG Documentation Wiki, XMG strives to generate “a fully redundant strongly lexicalised grammar" from "a compact representation of grammatical information" (Parmentier 2005). XMG specifies syntactic structures with tree fragments instead of rules. The formalism allows users to control how these fragments combine via well-formedness conditions and unification operations on the trees. It currently allows for multiple levels of representation (called *dimensions* within the framework), including syntactic structure *syn*, semantic features *sem*, and an interface between these two *dyn*.

An XMG grammar can be thought of as a tuple of <principles, types, properties, features, classes, value statements>. Principles state well-formedness conditions on tree structures, such as each tree only having one finite verb in the matrix clause. Types define what values are allowed to be associated with a given feature or property. Properties are realizations of types that can be used in principles used to control how tree fragments combine. Features are also realizations of types but are linked to tree nodes. Value statements ask the XMG compiler to compute the full structure of a class (discussed below) by combing all its component fragments.

Classes are XMG’s mechanism for defining tree fragments. They declare what nodes the fragment will contain, their dominance and precedence relationships, and any feature values and unifications. They also declare any relationships on the semantic level and how these *syn* and *sem* levels interface through the *dyn*. Classes can inherit from other classes and define which of their elements are available to combine with elements in other structures.

As a system XMG is both a formalism and a tool set. The tool set consists of a compiler, a virtual machine (VM), and set of post processing modules. These elements work together to expand the fragments described in the text-based input, flesh out the full set of trees, and run any post processing needed to ensure well-formedness or convert the output into

a given format (XML, etc.). The post-processing modules are where different dimensions are processed and were other dimensions can be added.

XMG is not a parser, but another framework for developing grammars. It is targeted at grammar formalisms that use trees templates to specify how elements combine to form larger structures. XMG is a formalism for defining tree fragments; in contrast, PATR-II is a formalism for defining phrase structure rules and constraints on their unification. Both formalisms provide a declarative, order independent language for defining grammars. Unlike XMG, the PC-PATR implementation augments the PATR-II formalism by allowing authors to violate this order independence with overwrite operations known as priority union operations. Priority union operations allow a feature value to be overwritten at a given point during parsing. Both formalisms operate using unification of compatible units and provide feature structures. XMG provides an explicit type system, that is not present in PATR-II. This provides a way to check for errors and typos that is missing from PATR-II and its PC-PATR implementation. Unlike PATR-II, XMG must be compiled, using XMG's tool set, before it can be used in a parser or other natural language processing software.

1.1.2 The Stanford parser

The Stanford Parser is a probabilistic context free parser with implementations in English, Arabic, Chinese, and German (Stanford NLPG 2013a). It requires a training corpus of hand marked up trees to learn how to parse and to build its part of speech tag inventory (Stanford NLPG 2013b). This limits its usefulness for testing grammatical hypotheses until users have an extensive understanding of the language under study. It is a context free parser without feature structures. Thus it must account for grammatical phenomenon like subcategorization by using different part of speech tags. This causes a dramatic increase in the number of rules required to build a complete grammar and misses significant generalizations about the structure of the language under study. In contrast, PC-PATR allows for feature constraints; thus structural similarities can be captured that the Stanford Parser will miss.

Unlike PC-PATR, The Stanford Parser builds its parser based on probability. Thus it can return a specified number of parses ranked by probability for a given input. PC-PATR cannot not do this. Another striking difference is that the Stanford Parser “does not attempt to determine grammaticality” and any notion of grammaticality is relative to the corpus it was trained on (Stanford NLPG 2013b). In contrast, PC-PATR allows the user to specify whether or not to return failures and partial parses and judges grammaticality relative to the grammar file supplied by the user. Also PC-PATR’s underlying formalism is more powerful. Because PATR-II is recursively enumerable it should be able to parse any language; in contrast, context free grammars may have trouble with languages such as Swiss German which is often regarded as being "weakly non-context-free" (Shieber 1985:337).

1.1.3 Leopar

Leopar is a unification based syntactic parser based on the Interaction Grammar (IG) formalism (Guillaume 2012a). Like PATR-II, Interaction Grammars rely on unifying feature structures to build parses, but they differ as IG uses tree fragment templates to represent constituent structure, rather than rewrite rules. IG templates contain "Polarity Features" that specify if a node needs a certain feature to be complete (negative polarity) or if it offers a feature that can complete other features (positive polarity) (Guillaume & Perrier 2010). Like PATR-II, this formalism allows features to be specified as requiring a value, even though they do not currently have one. PATR-II expresses this with the feature value *ANY* and IG with the ~ character. Unlike PC-PATR, Leopar is tied to IG and cannot parse with grammars from other theories. The grammars are XML based using the eXtensible Meta-Grammar formalism (see Crabbé et al. (2008))[3] and the lexicon files are written using the Lexcomp format (see Guillaume (2012b)). The parser is not theory-neutral even though its inputs are.

[3] The XML website lists a number of tools including a structured editor.

1.1.4 The Link Parser

The Link Parser (Temperly et al. 2012) is a parsing tool based on Link Grammar. Like Leopar, it is tied to a particular grammatical theory, but not language. This theory views words in a sentence as containing a number of link points that must be satisfied for a well-formed sentence. Each link point may accept multiple types of links and each links to a preceding or following element. For example, Sleator & Temperly (1991) describe the common noun *cat* with two link points: a preceding determiner and either a preceding link where it is the object of a verb or a following link where it is the subject as shown in Figure 2.

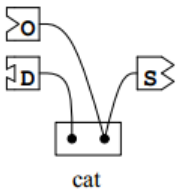

Figure 2. Linking for *cat* from Sleator & Temperly (1991)

Link Grammars are lexically oriented and handle structure building by means of three well-formedness conditions: linking lines may not cross, each link for each word must be linked to another word, and each word in a sentence is linked (Sleator & Temperly 1991). This grammar does not have concepts of constituent structure or grammatical function (Temperly 1999). This is very different from PATR-II as structure building operations are the responsibility of rules and lexical entries can occupy terminal symbols if their category features unify. It could be argued that this formalism is unification based as links must be compatible and must be satisfied in order for a sentence to be complete. However, it is certainly not unification based in the same sense as PATR-II. Because of its tie to Link Grammar, this parser is not theory neutral and so is limited to linguists willing to work within that theory.

1.2 Issues with PC-PATR

PC-PATR is a powerful tool based on a powerful formalism. Human interaction with the parser currently has a few issues because grammars are unstructured, plain text files. Consider Figure 3. This is a simple rule and its associated unification constraints. First, the keyword *rule* identifies it as a rule. The words in brackets serve as an identifier that associates it with nodes in the parser's output where it is used. Then comes a context free phrase structure rule that states that S dominates NP and VP and that NP precedes VP; it is optionally delimited by a colon. Finally, the last two lines are unification constraints that require that the *head* features of S and VP unify and that the *head* feature of NP and the *subject* feature of VP unify. The optional '.' at the end of the last line signifies the end of the rule.

```
rule {sentence formation}
S -> NP VP:
<S head> = <VP head>
<VP head subject> = <NP head>.
```

Figure 3. Example plain text rule[4]

Each character in the above example is a chance for error. If the author accidentally wrote the third line as <*S hed*> = <*VP head*>, misspelling *head*, then PC-PATR will interpret *hed* and *head* as different features. If NP were typed as "N P" then PC-PATR would understand that S should be composed of the nodes N, P, and VP. PC-PATR does not require (or allow) a master list of features to be declared in the grammar file. They come into the grammar as they are used and there is no way to keep track of the structure of the feature system apart from manually extracting it. Thus if the feature path <*agr number*> were inverted at some point in the file, then the user would have declared that *number* has a child *agr* in addition to not selecting the desired features.

[4] This file is from SIL International (2005)

Unlike some parsers mentioned above, PC-PATR is capable of returning partial parses with information about where a given parse failed. This information points out where changes need to be made or what elements are not working as the author expected. The output is either XML or plain text. The XML, while human readable, is difficult to understand. For example, I ran the parser on the sample data available from SIL International (2005) `shown in Appendix A. The output from parsing with these files was 344 lines long and contained two failed parses. The sentence *uther sleep* fails because of the number disagreement between the singular noun *uther* and the plural verb *sleep*. Figure 4 (excerpted from line 87 of the parser output) shows the node and feature markup from the parser output indicating this failure. The attribute on the line 87 “fail="true"" indicates that something in this node failed and the content of the *number* feature “FAIL singular / plural" on line 97 indicates that *number* had conflicting values.

00000087 <Node cat="S" rule="sentence formation" id="s0_1._1" fail="true">
00000088 <Fs> 00000089 <F name="cat"><Str>S</Str></F>
00000090 <F name="head">
00000091 <Fs>
00000092 <F name="subject">
00000093 <Fs>
00000094 <F name="agreement">
00000095 <Fs>
00000096 <F name="person"><Str>third</Str></F>
00000097 <F name="number"><Str>FAIL singular / plural</Str></F>
00000098 <F name="gender"><Str>masculine</Str></F>
00000099 </Fs>
00000100 </F>
00000101 </Fs>
00000102 </F>

00000103 </Fs>

00000104 </F>

00000105 </Fs>

Figure 4. Excerpt from parser XML output for shieber1.sen

The plain text output for the same sentence is shown in Figure 5. While this is much easier to understand, it is still a lot of text to process. The feature structures are not arranged in the same format as the parse tree so the reader must make a mental shift in order to keep track of what feature structure corresponds to what node. Whether the output is XML or plain text, the user still has a lot of information to sort through to find the point of failure.

```
uther sleep
**** Not able to parse this sentence ****
**** Turning off unification ****

1:
    S_1-
   ___|____
 NP_2   VP_3
 uther    |
         V_4
        sleep

S_1 (failed):
[ cat:   S
  head:    [ subject:  [ agreement:  [ person:third
                                     number:FAIL singular / plural
                                     gender:masculine ] ] ] ]

NP_2:
[ cat:   NP
  lex:   uther
  head:    [ agreement:  [ person:third
                         number:singular
                         gender:masculine ] ] ]

VP_3:
[ cat:   VP
  head:    [ subject:  [ agreement:  [ number:plural ] ]
             form:  finite ] ]

V_4:
[ cat:   V
  lex:   sleep
  head:    [ subject:  [ agreement:  [ number:plural ] ]
             form:  finite ] ]
```

Figure 5. Excerpt from plain text parser output for shieber1.sen

Reading the output is not the only problem facing users. PC-PATR provides no mechanism for enabling or disabling rules. Thus for users to turn off a rule they either have to comment it out, or remove it from the grammar. If the rule is removed, then it may be lost, or retyped incorrectly when it is reentered. If the user chooses to create another version of the grammar without the unwanted rule, then some kind of discipline must be developed to track which file is the most advanced, the most correct, or contains a solution to a given problem. Supposing a solution is incompatible with some other element of the grammar, how will the user integrate those solutions? This applies not only to rules, but also to unification constraints. In my experience, unification constraints and feature passing are where most of the tweaking takes place while developing PC-PATR grammars.

When debugging a grammar or focusing on a particular construction, the user may want to disable other rules and templates not relevant to the object of study as this increases the parser's speed. In this case, the same problems of data loss can occur. The user may forget to remove the comments from a rule or forget its exact form if the rule was removed. Thus a mechanism is needed to prevent the loss of data while editing and debugging grammars.

CHAPTER 2

HOW PC-PATR-EDITOR SOLVES THE PLAIN TEXT PROBLEM

In order to address the issues outlined in section 1.2, I have created a structured editor called PC-PATR-EDITOR. This program is written in the C# programming language and uses Windows Presentation Foundation for the user interface. As such, it is currently tied to the Windows platform. This program reads and writes data to an XML format defined in a document type definition created by H. Andrew Black included in Appendix B. As a structured editor it removes the need for the user to know the syntax of the PC-PATR input files, by constraining what the user is allowed to insert at a given point. It also helps keep the grammar consistent by maintaining a hierarchical feature system and lists of terminal and non-terminal symbols used in the grammar.

2.1 Solutions to input errors

A number of the errors discussed in section 1.2 come from user input errors. The use of a structured editor prevents many of them. Figure 6 is a screen shot of PC-PATR-EDITOR's page for editing a rule and its unification constraints. The image contains circled numbers as indexes which were added for reference. This rule is the sentence formation rule from Figure 3. At Index 1, the user can input a name for the rule (this appeared inside curly braces in the first line of Figure 3). Index 2 is over a button that lets the user set the ID tag that will be used to refer to this rule within the grammar.[1]

[1] Currently the editor does not suggest an id. I recommend developing a prefix that helps identify what kind of element the ID refers to. For example, *rule* for rules, *ft* for feature templates, *ct* for constraint templates, *cf* for collection features, *f* for features, and *s* for symbols.

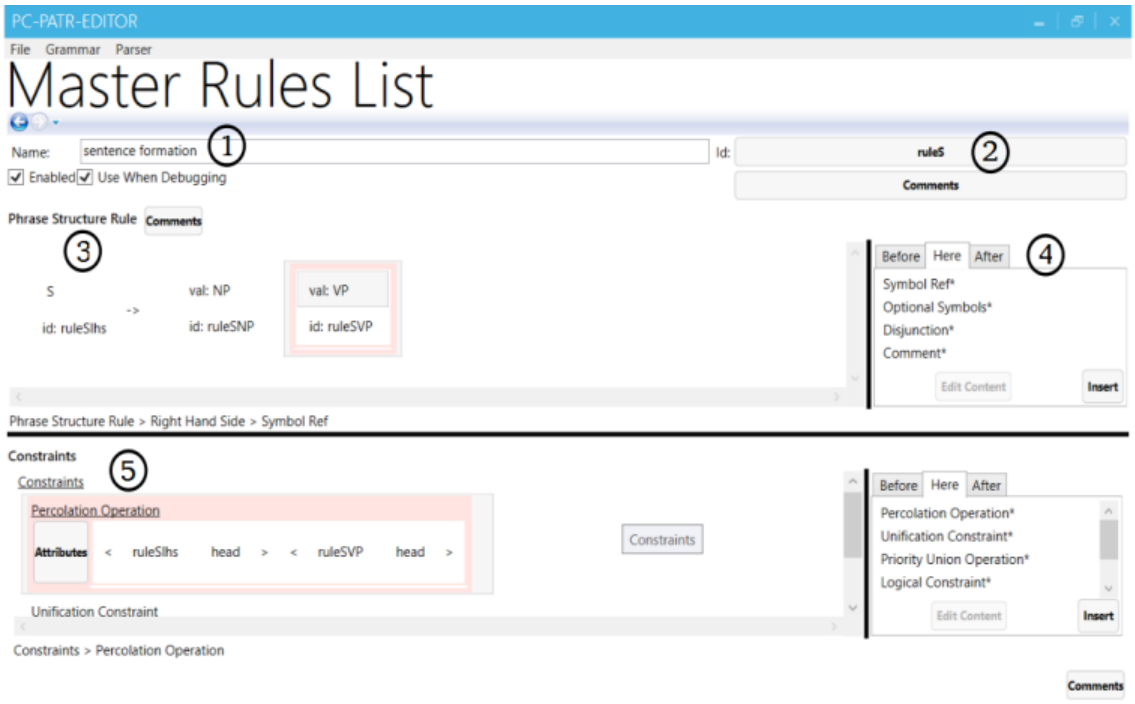

Figure 6. Screen shot of PC-PATR-EDITOR editing a rule

Indexes 3 and 4 show the primary editor control that is found throughout the program. This control is composed of two parts: the content pane (Index 3) and three option lists (Index 4). These lists show the user what can be inserted at the selected point as well as before and after it. Each element in the content pane is either a container block or a content block that has a link to its parent and its children (if any). The parent of a given block has a rule that determines what elements it may contain. When the user selects a block in the content pane, its parent looks up what may be inserted at that point and displays the options in the tab titled "Here" in options lists (Index 4). The parent also determines if the user can insert anything before or after the selected element and displays those options in the "Before" and "After" tabs. If an element is allowed to repeat (indicated by an '*' after its name in the options list), then it will be appended after the selected element; If not, then the selected element will be replaced. The control also displays the ancestry of the currently selected block below the content pane to help orient the user (just above Index 5).

This system removes the onus from the user to know and respect PC-PATR's grammar file syntax. The user simply chooses from a list of available elements and the editor handles making sure that the final output works with PC-PATR. For example, Index 3 shows a simple phrase structure rule where the user has selected the last element. The "Here" list (Index 4) shows that he or she can insert a SymbolRef (terminal or non-terminal symbol), a set of optional symbols, or a disjunction. The user does not have to know that PC-PATR requires that optional symbols be surrounded by parenthesis or that disjuncts in disjunctions be separated by a forward slash. Thus the potential for these syntax errors has been removed.

Content blocks have an edit method that is invoked by clicking on their content or by a button located in the bottom right of the "Here" list (not shown). This causes the appropriate list or input to be displayed so that the block's content is constrained to what is allowed at that point by PC-PATR. Figure 7 shows the result of clicking on a SymbolRef's symbol value. A list of all terminal and non-terminal symbols defined in the grammar is displayed. The user clicks to make a selection and clicks the OK button to change the content of the block or clicks Cancel to leave the block the way it was. Note that the user cannot accidentally add a new symbol or mistype the desired symbol name. This removes the chance for errors by simple typing mistakes.

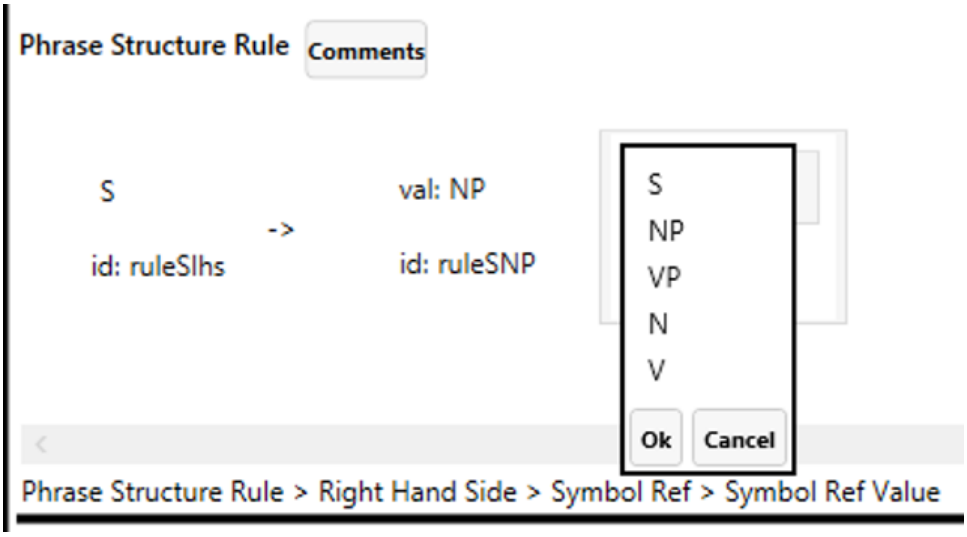

Figure 7. Inserting a symbol on the right hand side of a phrase structure rule

2.2 Solutions to content errors

Other types of errors can be caused by the user making changes to the content of a rule or template or by not remembering the name of a symbol or the structure of the feature system. If a change affects an element that is referenced in another place, but that change is not percolated throughout the grammar by the user, then the grammar will not behave as expected and simply be incorrect. The editor helps prevent content errors by providing the user with a list of options to choose from when selecting what symbol or feature to insert. This takes the load of remembering which symbols are terminal or non-terminal or the structure of the feature system off the user's mind and helps ensure a consistent grammar.

Figure 8 shows what happens when the user clicks on the left hand side of a phrase structure rule. This list displays only the symbols the user has defined as non-terminal symbols. This prevents the mistakes above and prevents a user from accidentally creating a non-terminal category from a terminal one without meaning to do so.

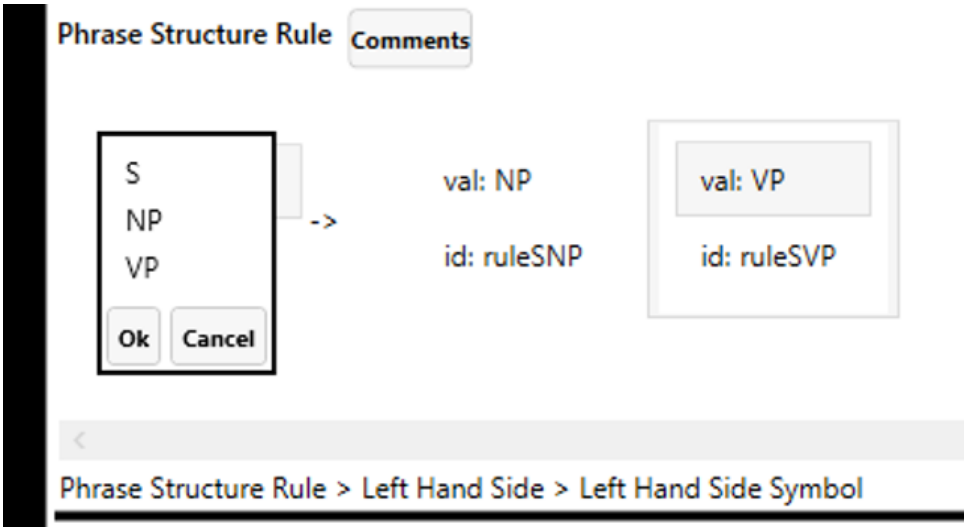

Figure 8. Selecting the left hand symbol in a phrase structure rule

In Figure 6, the content pane at Index 3 shows the elements in the phrase structure rule. These elements are displayed in a similar manner to how the rule would be written in the plain text file, but with one major difference. Each element has a unique ID tag. The editor treats symbols in phrase structure rules as elements occupying slots called SymbolRefs.

Nodes in the phrase structure rule are referenced in unification constraints by their SymbolRef's ID tag. Thus if the user changes what symbol occupies a slot, the constraint will automatically reference the correct symbol when the grammar is executed. For example, suppose the user realized that there is some other phrase, such as an IP, between the S and VP nodes. So he or she changes the phrase structure rule from *S -> NP VP* to *S -> NP IP* but the *head* feature still needs to percolate from IP to S. Because nodes are referenced by their SymbolRef's id, the references to IP in the unification constraints have not been broken. If the user chooses to delete an element of the phrase structure rule, a warning is displayed that this could break any references to this item and the user is asked if he or she would like to continue. This helps prevent references between elements of the grammar from being broken while providing an easy way for the user to edit content.

Index 5 in Figure 6 shows the list of unification constraints associated with this rule. This is the same editing control as for phrase structure rules. Figure 9 shows the first constraint from this rule which consists of two feature paths (shown inside angle brackets) and the result of clicking on the first element of the second feature path. The first element of a feature path is a reference to a node in the phrase structure rule. When this block is clicked, the editor displays a list of the SymbolRefs in the phrase structure rule including the left hand element. Thus users cannot accidentally reference a symbol that is not found in the phrase structure rule.

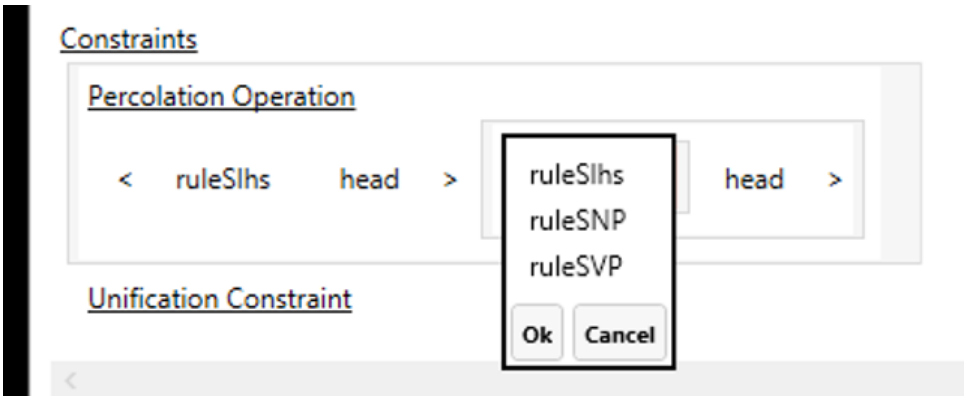

Figure 9. Selecting a symbol in a unification constraint

Feature paths (and other elements that reference features) can only include features that have been defined in the feature system for the grammar being edited. When the user clicks on a block that contains a reference to a feature, the editor presents a hierarchical list of features that are available to insert for the selected block. For the first element of a feature path, any feature is available. The end of a path only allows features that are children of the first feature, PC-PATR grammar primitives (cat, lex, etc.), or special features known as "Collection Features."[2] The editor enforces this requirement by presenting only these elements to the user.

PC-PATR-EDITOR also helps prevent content errors by requiring that features be defined in a hierarchical feature system before being available for use. Figure 10 shows the feature system page in the editor. The system is displayed in a tree to reflect its hierarchical nature (Index 1). Each feature has a name (Index 2) and an ID (Index 3). Like SymbolRefs, features are referred to by their ID tag in the grammar. Thus the name of the feature can be freely changed without having to change all of its usages in rules and templates. The feature system offers the user a way to create references to features within the system. This prevents the user from having to define the same feature in multiple places if it is needed as the child of multiple features. Suppose a user wished to state that the *subject* feature of a verb could have all the same features as the *head* feature. This would be accomplished by inserting a reference to the *head* feature as a child of *subject*. This prevents duplication and further chances for error.

[2] Collection features are "not your garden variety morpho-syntactic features but rather a way to do things in PC-PATR" (H. Andrew Black, personal communication, 27 Jan 2012). They provide a mechanism for passing groups of features around syntax trees and using them in constraints. They will be discussed later in this section.

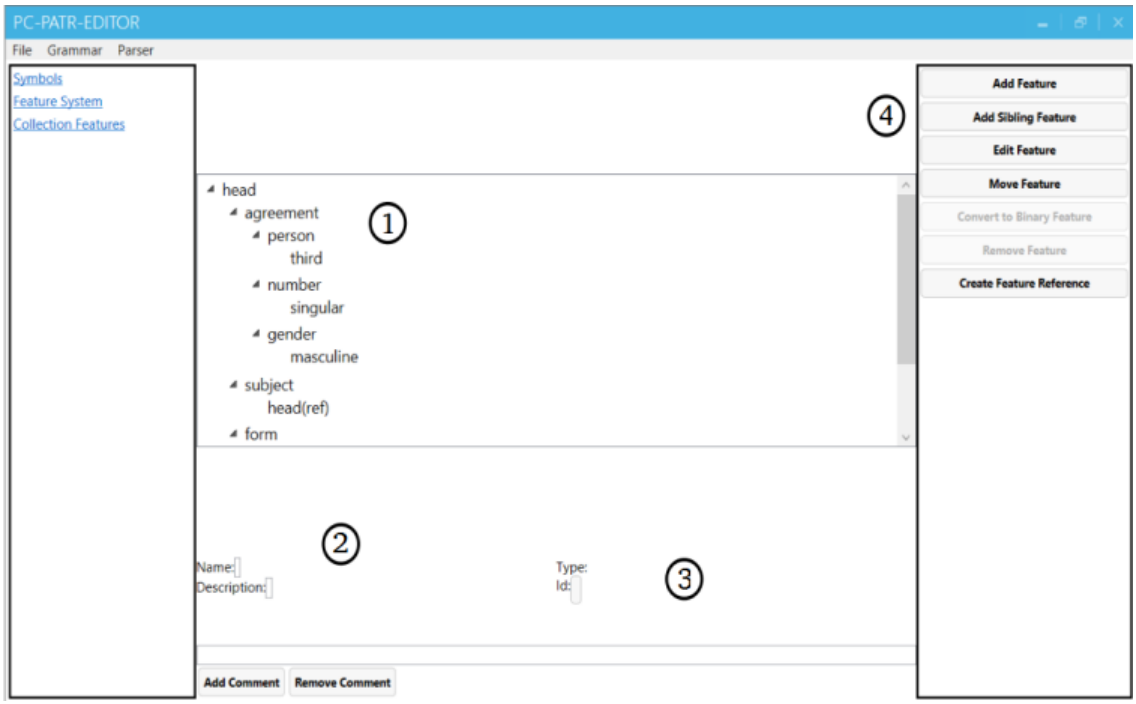

Figure 10. PC-PATR-EDITOR feature system edit page

Collection features are a special kind of feature that is used like a pipeline for passing groups of feature around syntax trees and for making them available in unification constraints. Like morpho-syntactic features, collection features must be defined before they are available for use. They are defined like symbols. Figure 11 shows the editor's page for defining collection features. The editor displays a list of all collection features defined in the grammar (Index 1) and a pane for editing the selected collection feature (Index 2). Like other features, collection features are referenced by their ID tag. Users can define an ID by clicking on the button to the right of Index 2 and edit the name and description of collection features using the input fields to the right and below Index 2. Users can also add and remove comments.

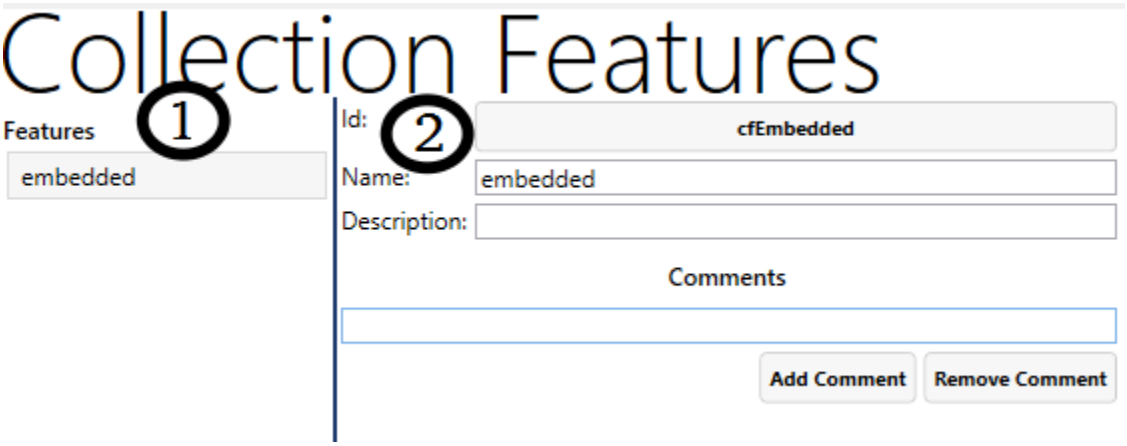

Figure 11. PC-PATR-EDITOR collection feature edit page

The editor requires users to define lists of terminal and non-terminal symbols before they can be used in the grammar. Figure 12 shows part of PC-PATR-EDITOR's page for defining symbols with the symbol edit dialog open. This page shows the list of non-terminal symbols defined in the grammar (Index 1) and the list of available editing actions (Index 2). In this figure the user has clicked Edit Non Terminal Symbol. This caused the symbol edit dialog to be displayed (Index 3). This dialog allows users to edit the name and ID of the selected symbol, as well as add an optional description and any comments. The list of terminal symbols works the same way.

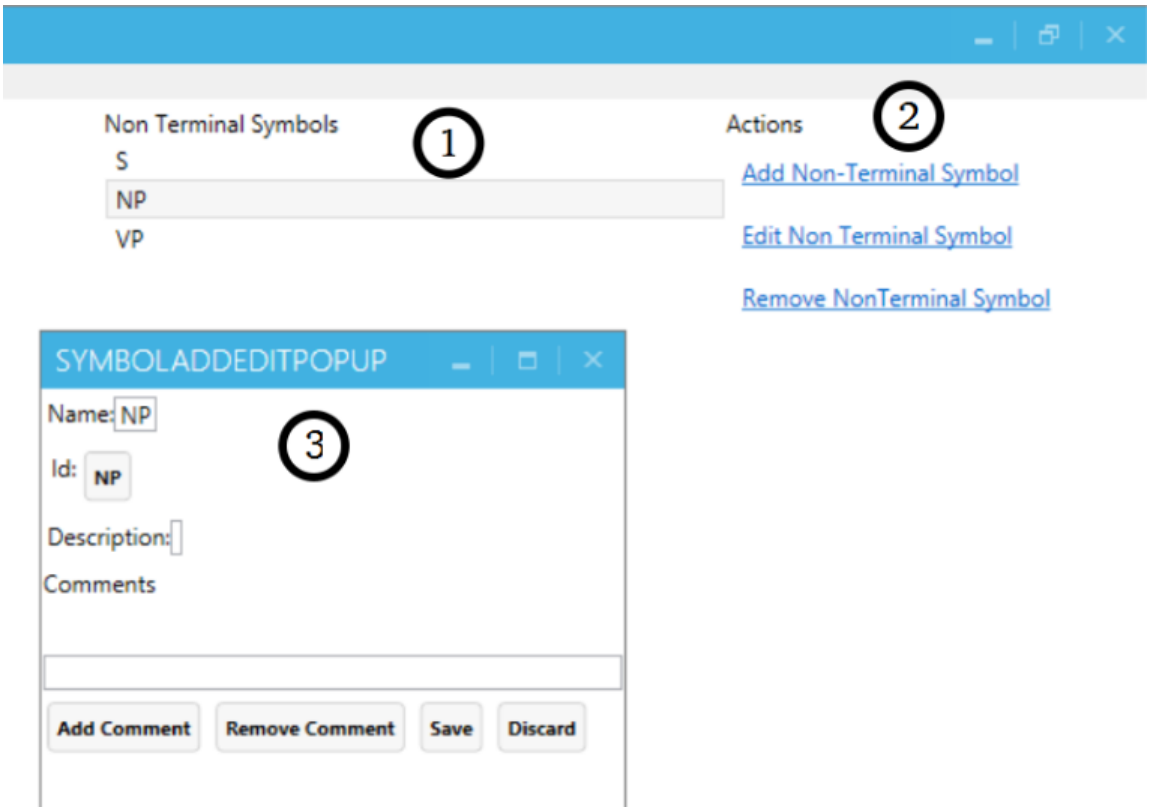

Figure 12. Excerpt from PC-PATR-EDITOR's symbol's edit page

PC-PATR-Editor helps users create coherent grammars by requiring users to define what symbols and features can appear in the grammar before they are available for use. This removes the chance for errors due to simple typing mistakes. It also helps users remember the structure of the feature system and what symbols have been defined. The editor also provides a structured environment for developing grammars. This lets the user focus on the content of the grammar rather than the syntax of the files required by PC-PATR.

CHAPTER 3

HOW PC-PATR-EDITOR SOLVES THE DEBUGGING PROBLEM

Users of PC-PATR face two main types of difficulties when debugging and editing a grammar: the complexity of the parser output and making edits without data loss. PC-PATR can return a lot of valuable information when a parse fails. This information is encoded in either a verbose plain text or XML format. In either case, this output contains a lot of text that must be comprehended before the user can make informed decisions to revise the grammar. In response to a failure or unexpected result, a user might want to try running the grammar without certain elements to see how that affects the output. As noted in section 1.2, this requires removing rules, commenting out rules, and/or creating multiple versions of the grammar file. This may result in data loss and a higher mental load on authors who have to keep track of what has been done where.

PC-PATR-EDITOR solves these issues by allowing users to enable and disable elements without having to remove them. It provides a debug mode that lets users select a subset of the grammar to run. It also processes the parser's output and displays where failures occur so users do not have to find them manually. Syntax trees are displayed in a manner familiar to most linguists and the feature structure associated with a node is displayed when that node is clicked on. The editor also displays a list of parser settings that help users better debug their grammars and has a few comparison tools.

3.1 Solutions to content loss

Content loss is a problem facing users of PC-PATR when they want to run the grammar without a certain rule or with a different set of constraints. Section 1.2 described how this can lead to data loss and a proliferation of grammar files. PC-PATR-EDITOR provides two control attributes: Enabled and Use When Debugging. Enabled allows for rules, constraints,[1] and feature templates to be enabled or disabled. Use When Debugging allows users to select a subset of the active grammar to be used when the parser is run in debug mode.

Both control attributes can be used to exclude a rule from being included when the parser is run. Use When Debugging is set to false for rules and feature templates by default. It is meant to provide a quick way to run only a subset of the grammar. This provides a mechanism to work out how to handle a particular construction without the interference of the rest of the grammar. This could also be achieved by simply disabling elements, but this way the user can keep the total grammar intact while still working on a small portion. When the user runs the parser in debug mode (see Section 3.2.1), only elements with Use When Debugging set to true (checked) are used by the parser.

Rules by default are created enabled, but not flagged to be used when debugging. These attributes can be easily changed by checking or unchecking the boxes shown in Figure 13. The boxes are located in the top left of the Rule editing page (near Index 1 in Figure 6 in Section 2.1). The rule shown in Figure 13 is enabled and set to be used when debugging.

[1] In this chapter I am using "constraint(s)" to mean all types of constraints possible for rules (percolation operations, unification constraints, logical constraints, priority union constraints, and constraint disjunctions) unless otherwise indicated from context.

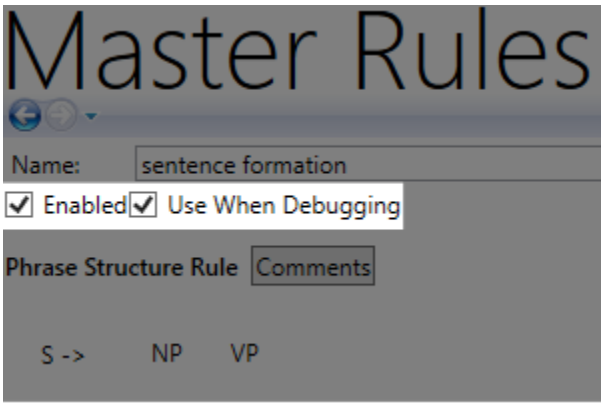

Figure 13. Control attributes for rules

By default, constraints are enabled and flagged to be used when debugging. They will only be included in the grammar if their containing rule is included as well. To modify these attributes for constraints, the user must select the block containing the constraint. This will cause its attributes to be displayed in the Attributes pane of the constraint editing control. Figure 14 shows this; the user has selected the Percolation Operation (pink background) and its Use When Debugging and Enabled attributes are both enabled (bottom right).

Figure 14. Control attributes for constraints.

Feature templates can also be enabled and disabled and marked for use when debugging in the same manner as constraints. Figure 15 shows these attributes for a feature template from a grammar fragment for Bahasa Indonesia (see case studies in Section 4). Like constraints, the user must select the feature template block. This causes the control attributes to be displayed on the right side of the editing pane. In Figure 15 both attributes are set to true and will be included in the grammar whether run normally or in debug mode.

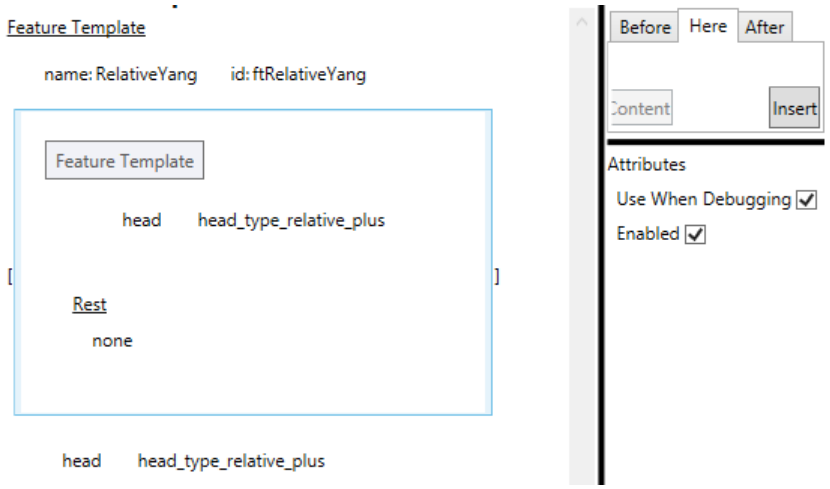

Figure 15. Control attributes for feature templates

PC-PATR-EDITOR also has a page where the control attributes for rules and feature templates can be edited en mass. Figure 16 shows this page. It is divided into two lists displaying rules and feature templates respectively. Each rule and feature template is displayed with its name followed by the Enabled and Use When Debugging attributes as check boxes. Changes are saved when the user clicks Ok or discarded when he or she clicks Cancel.

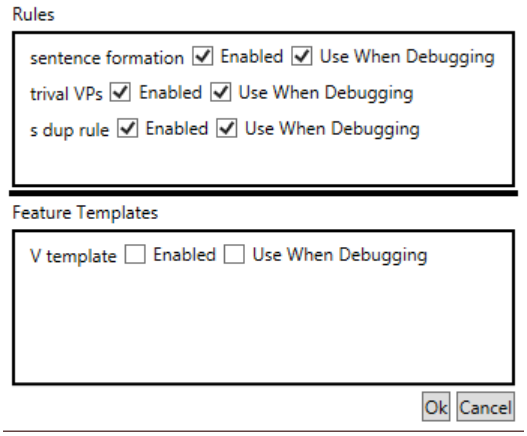

Figure 16. Control attribute mass editing page for rule and feature templates

Users can also select multiple elements to edit at once by holding down the Control key, clicking on them, and then right clicking. This will display a pop up menu shown in Figure 17 where the user has selected two rules from the Rules list. Users can toggle the

Enabled attribute by clicking Toggle Enable/disable and toggle the Use When Debugging attribute by clicking Toggle Use When Debugging. This inverts the value for that attribute on each selected item. If the attribute is unchecked (false) then it will be checked; if it is checked (true), then it will be unchecked.

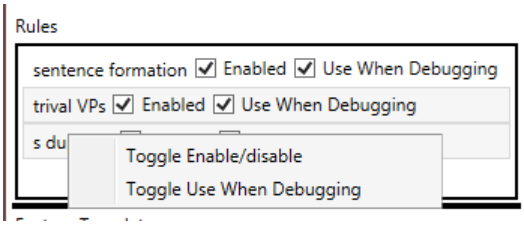

Figure 17. Right click menu for control attribute mass editing page

3.2 Solutions to difficulty reading parser output

PC-PATR's output can be difficult to read, especially when parsing long sentences. As discussed in Section 1.2, the plain text and XML output formats require significant mental energy from users in order to find and evaluate errors (see Figures 4 and 5 in that section). PC-PATR-EDITOR solves these issues by processing the parser's XML output and extracting where failures occurred and what they were. It then displays its analysis in ways that are familiar and easy to understand.

Figure 18 shows the editor's analysis page after being run on the files shown in Appendix A. The controls for the parser and the selected files are shown to the left of Index 1. The analysis pane lists each sentence in the parser output along with whether or not parsing succeeded for it (Index 2). The parse tree for the selected parse is shown using a control developed by H. A. Black used in LingTree (2009b) and the PcPatrBrowser (2009a) (Index 1). The editor displays the feature structure for the parse tree's selected node in a format also developed by H. A. Black (Index 3).

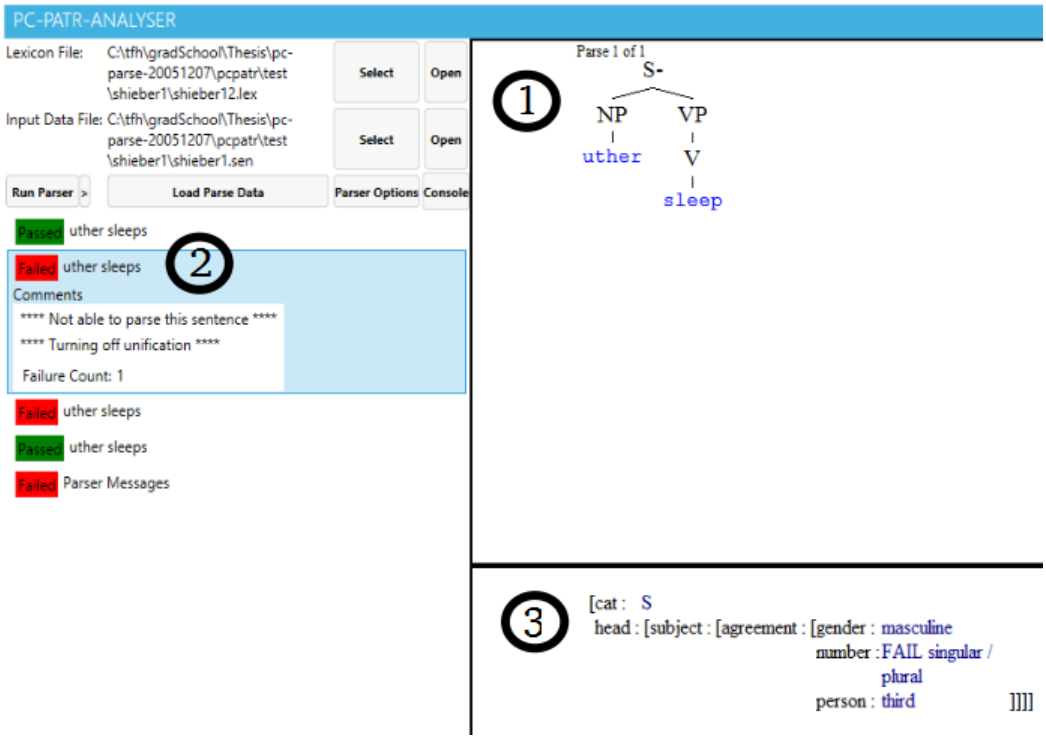

Figure 18. PC-PATR-EDITOR's parse analysis page

3.2.1 Parser modes and options

PC-PATR-EDITOR offers a variety of options for parsing and revising grammars. The main parser controls are shown in Figure 19. The two buttons labeled Select allow the user to choose the lexicon and data files to use when parsing. The Open buttons (to the right) open the selected file in the user's default text editor for quick editing. Along the bottom of this image are different actions available to the user.

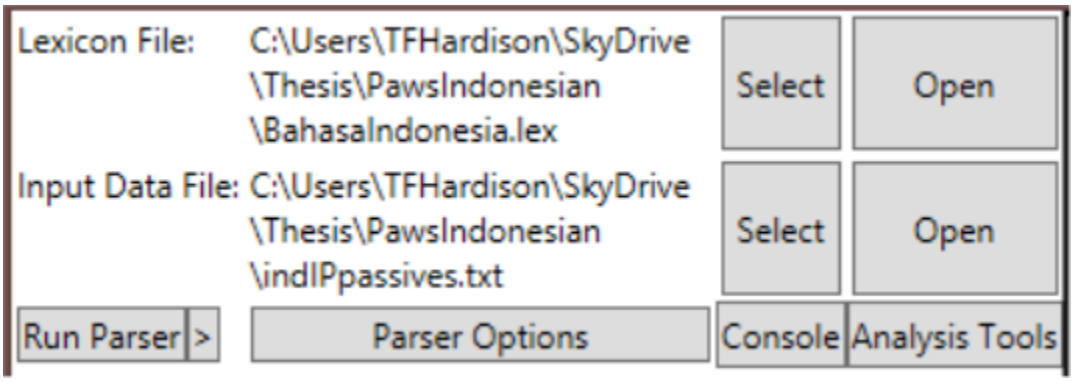

Figure 19. Parser controls

The PC-PATR-EDITOR offers three ways to parse with a grammar: Default, Debug, and Grm mode. The default mode runs the grammar with any rule, constraint, or template marked as enabled. It is executed by clicking the Run Parser button in Figure 19 (bottom left). Thus, if a user has disabled a given rule, it will not be included in the version of the grammar sent to the parser. Debug and Grm modes are accessed by clicking on the ">" button next to Run Parser. This displays the pop up shown in Figure 20. Debug mode runs a version of the grammar with only those rules, constraints, and templates marked to be used when debugging. This lets the user run subsections of the grammar. Grm mode lets the user select a plain text grammar file and use that to parse with the selected lexicon and data file. This mode allows for backwards compatibility with PC-PATR grammars developed using the plain text file format.

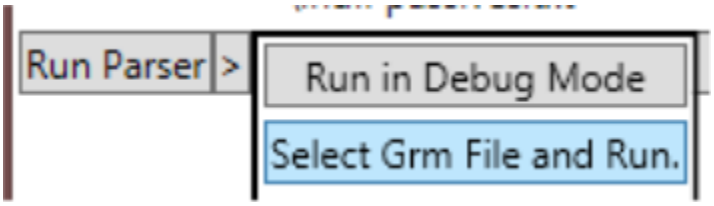

Figure 20. Parser modes

There are two parsing engines available to users in PC-PATR-EDITOR. The default is to use the complied DLL version[2] built into the editor. The second option is to run PC-

[2] Patr100.dll

PATR as a command line tool.[3] The incorporated version is not as stable when working with data that is not well covered by the current grammar and may cause the program to crash if it cannot finish parsing. The command line option avoids this because it runs as a separate process.[4] Also the command line tool's error output is read by PC-PATR-EDITOR. Clicking the Console button (Figure 19 bottom right) displays this output. There are trade-offs to using one version or the other, but the editor offers both as a help in debugging grammars.

PC-PATR-EDITOR has a list of options that can be used when parsing, shown in Figure 21. It is accessed by clicking Parser Options (see Figure 19). The options page is organized by what part of the parser the option pertains to. Figure 21 shows the default options for the parser. For a discussion of what the options are see McConnel (2006).

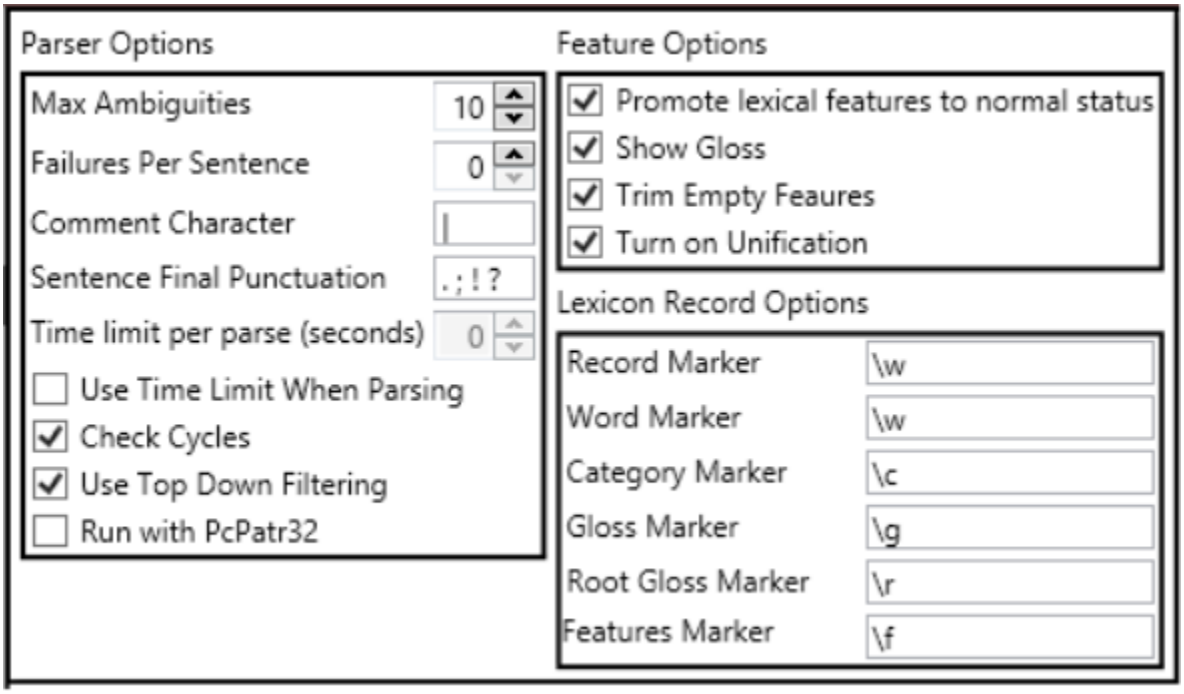

Figure 21. Parser options

[3] pcpatr32.exe

[4] This is advantageous because if pcpatr32.exe crashes, it is in a separate process and will not cause PC-PATR-EDITOR to crash. H. A. Black (personal communication) noted that the crashes could be caused by a bug in the DLL that does not always reset its internal variables when told to by the calling program; this is not a problem for the command line tool because it initializes its variables each time it is executed.

3.2.2 Analysis view

PC-PATR-EDITOR presents its analysis of the parser's output in the analysis pane on the left half of the analysis window (Index 2 in Figure 18). It processes the parser output and returns a status for each sentence and a list of failures (if any) for each parse. The analyzer currently returns three statuses: Passed, Failed, and Passed with ANYs. Passed with ANYs can mean that there were some features that were not assigned a value during the parsing or that unification failed for these features.[5]

When a user selects a sentence, he or she is shown a list of parses for that sentence and the number of failure points in each parse. Figure 22 shows a list of four sentences with their statuses, the input sentence, and the number of failures. In this image, each sentence parsed twice. The second sentence has been selected so its parse list is shown. This sentence contains two parses that each contain a single failure.

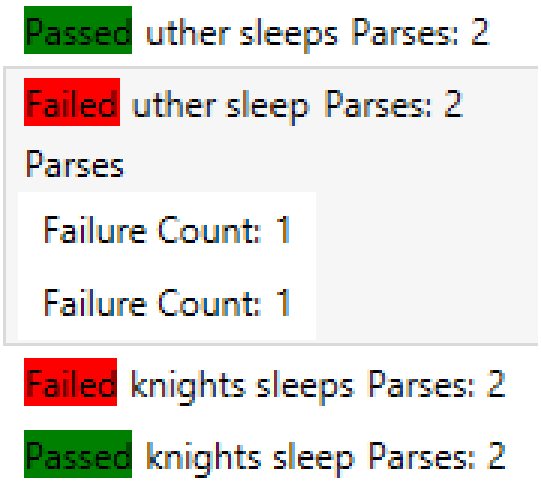

Figure 22. List of parses from shieber1.sen

Figure 23 shows the details for the first parse returned for *uther sleep* in Figure 22. There are two lists for each parse. The first (untitled) is a list of failed nodes. The second

[5] In my experience working on the grammar described in Section 4, a node could be marked as Failed without any failed features. This was probably because of a constraint that required a feature to have a specific feature value. For example, it might be caused by a constraint like <*subcat first comps first*> = *none*. H. A. Black (personal communication) adds that PC-PATR may not always indicate when a logical constraint fails.

(titled "ANY list") displays any nodes with features assigned the value *ANY* by the parser.[6] In this figure, the user has clicked on “Node: S” in the first list. The editor then displayed two additional lists for this node. The first (titled "Failures") contains a list of all features that failed and their value or value conflict. The second (titled "ANYs") is a list of any features in this node with a value of *ANY*. In Figure 23, the editor found that the feature *number* had conflicting values of *singular* and *plural*. It also has an empty ANYs list, so all features have been specified with a value.

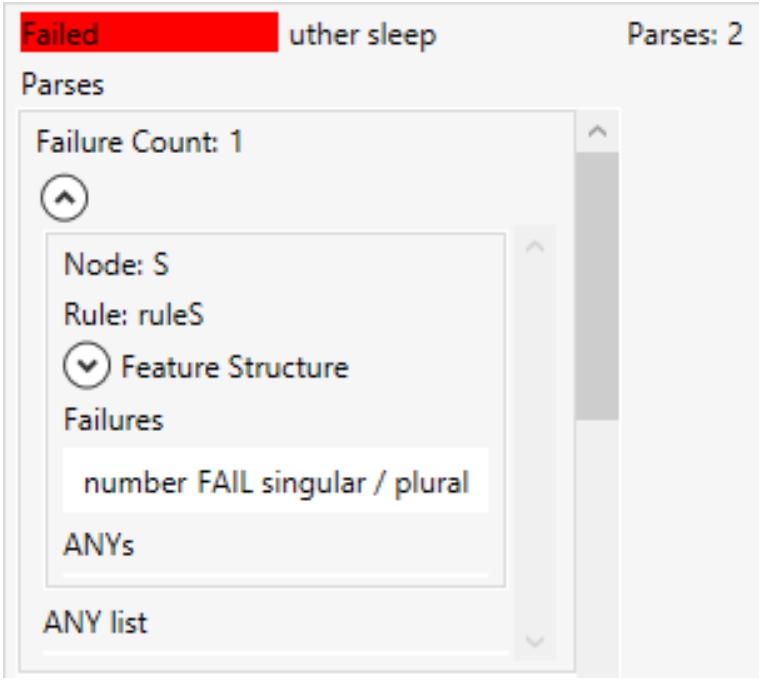

Figure 23. Details of failed parse analysis

3.2.3 Parse tree and feature views

PC-PATR-EDITOR presents a parse tree for the parse selected in the analysis pane. The parse tree is located in the top right side of the analysis window (Index 1 in Figure 18). The parse tree viewer indicates whether a node failed by the presence of a ‘-’ to the right of the node name. It also indicates whether a node appears with the same feature structure

[6] ANY is represented as <Any /> in the parser’s XML output.

in all parses by the presence of a '+' to the right of the node name. In Figure 24, the S is marked with a '-' to indicate that it is a point of failure, and the VP node is marked "+" to indicate that it has the same feature structure in both parses.

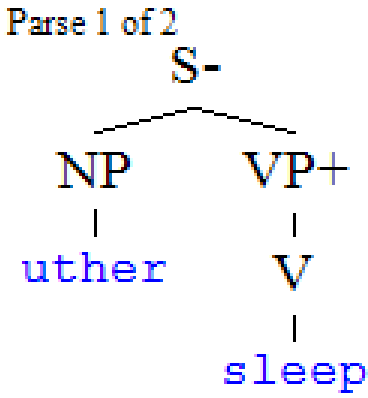

Figure 24. Parse tree for *uther sleep*

This presentation improves on the plain text output by displaying the parse tree in a more familiar format. PC-PATR prints plain text parse trees in ASCII characters[7] rather than the directed graphs common in linguistics textbooks. The parse tree viewer is also a vast improvement over the XML output which does not include a graphical parse tree.

When a user clicks on a node in the parse tree, the feature structure viewer displays the feature structure for that node (Index 3 in Figure 18). This allows users to easily view different feature structures without the mental load of finding them in the plain text output. If the node selected failed, then its analysis is also opened in the analysis pane. This presents the maximum amount of relevant information to the user.

Figure 25 shows the feature structure for the S node in Figure 24. The format of the feature structure displayed is very similar to what is returned in the parser's plain text output. However, the editor still improves on reading the plain text output in two ways. First, the feature structure is displayed when the user clicks on its corresponding node in the parse tree. This connects the parse tree to its component feature structures and lowers the amount

[7] See Figure 5 in Section 1.2 for an example

of context switching for the user. Second, the analysis pane highlights what features failed so the user knows what to look for when examining a feature structure which can be very large (see Figure 47). Furthermore, the user may not need to examine any feature structures to figure out why a parse failed because of the analysis pane.[8]

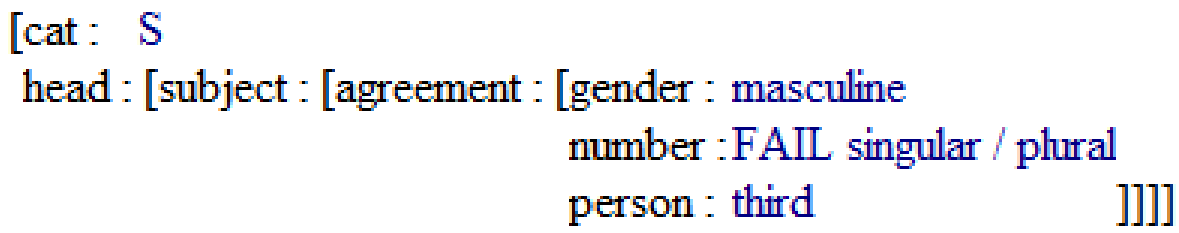

Figure 25. Feature structure view

3.2.4 Additional tools

PC-PATR-EDITOR has three additional analysis tools: a lexicon and input checker, a sentence comparison tool, and a node comparison tool. These tools are designed to help users analyze parse output by providing different views for the data and limiting the amount of failures to sort through. They are accessed through clicking on Analysis Tools (shown bottom right in Figure 19 in Section 3.2.1). This brings up the menu shown in Figure 26.

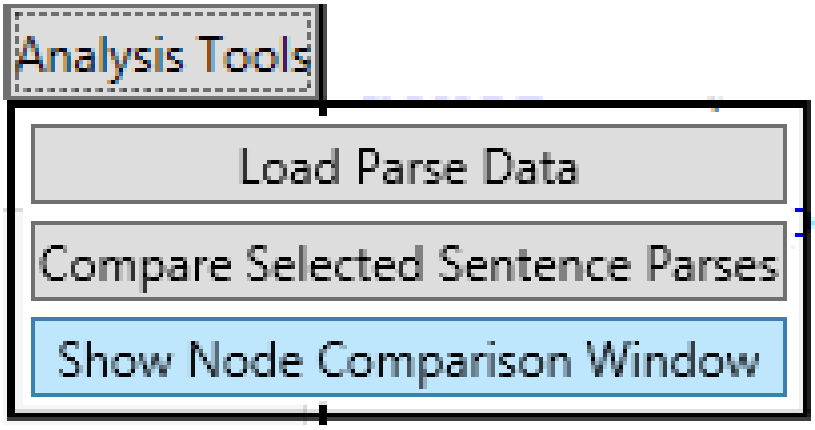

Figure 26. Analysis tools menu

[8] The feature structure is displayed as a web page within an embedded web browser. This means that the feature structure can be saved or printed by right clicking on it.

Load Parse Data allows a user to navigate to and load previous parse data. This data could be a log file generated by PC-PATR-EDITOR or PC-PATR as long as the output was set to XML.[9] Compare Selected Sentence Parses runs the sentence comparison tool described in Section 3.2.4.2 on the sentence currently selected in the analysis view. Show Node Comparison Window opens the node comparison tool described in Section 3.2.4.3. The lexicon and input checker is run automatically when the parser is run.

3.2.4.1 Lexicon and input checking tool

PC-PATR does not perform well when the input data file contains many words that are not in the selected lexicon file. It will not return an analysis for sentences with words missing from the lexicon. To aid users, the editor extracts a list of distinct words in the data file and checks this against the words in the lexicon when the parser is run. If there are words that are not found in the lexicon, then they are displayed in a pop up window shown in Figure 27. This window also asks the user if the parser should continue.

[9] I have found this useful in cases where I had closed the program and wanted to come back to analyzing the same parse data. Clicking Load Parse Data allowed me to get back to doing analysis without having to run the parser. This could be useful if a file took a long time to parse.

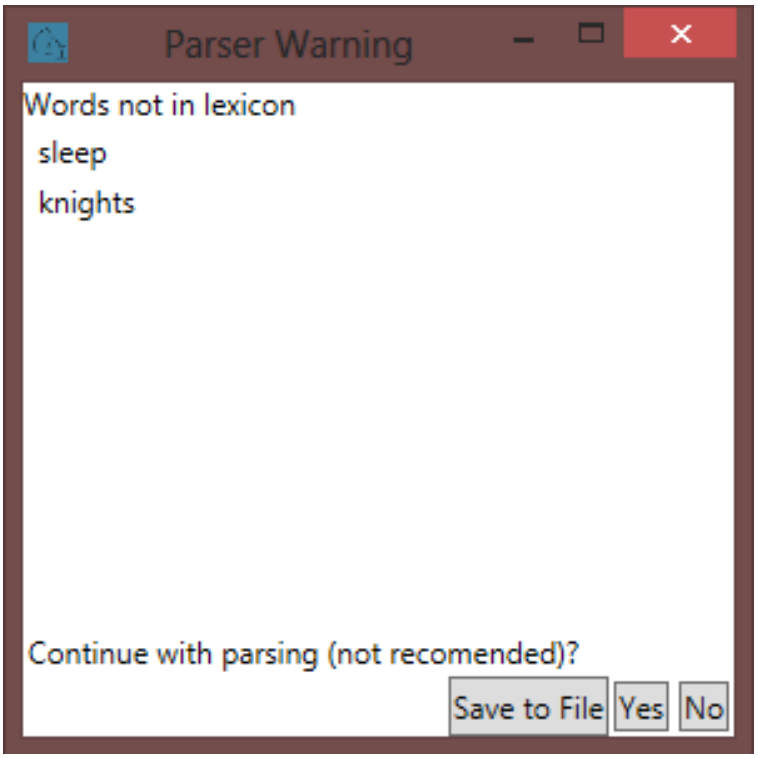

Figure 27. Lexicon checker pop up

The checker can save the list of words not present in the lexicon to a text file. Thus users have a list of what elements need to be added.[10] If the lexicon contains an entry for each word in the input file then this window does not appear. Note that this comparison only matches strings and makes no reference to position in the data file or syntactic category in the lexicon.

3.2.4.2 Sentence comparison tool

There are times when a user might want to see a comparison across all parses for a sentence. This can be useful for seeing how a feature percolates or what led up to or what follows a point of failure. PC-PATR-EDITOR offers a sentence comparison tool that provides this functionality. This tool is accessed by clicking on Compare Selected Sentence Parses in the Analysis Tools menu (see Figure 26 above). Unfortunately this tool is still not

[10] This functionality could be expanded in future versions to save the text file in standard format or a simple list.

working correctly. Ideally it will factor out features common to the parses to highlight their differences. At this point, it is only removing the first common feature if it removes any. It is still included in the editor because it can provide a way for the user to see all the nodes across parses and their features at one time.

The sentence comparison tool compares the feature structures for each node in each parse for the selected sentence. Taking the first parse as the point of reference, the tool compares the nodes in each level of the parse tree. If two nodes have the same structural position and category, then their feature structures are compared. If a feature is present in the first parse and any other parse, then it is factored out of the other parses in the report generated. Unfortunately, this is not working correctly and more often than not leaves nodes in the output that should be removed. This is a matter for future work on the editor.

The result of this analysis is displayed as a web page in the user's web browser. Figure 28 shows the result of running this tool on the sentence *buku in harus kaubaca* 'this book must be read by you' from (8) in Section 4.3. Each parse is listed in its own table with its parse number for a header and vertical lines on each side to delimit it. Each node is listed with its name in bold and the rule that built it in parenthesis next to it.

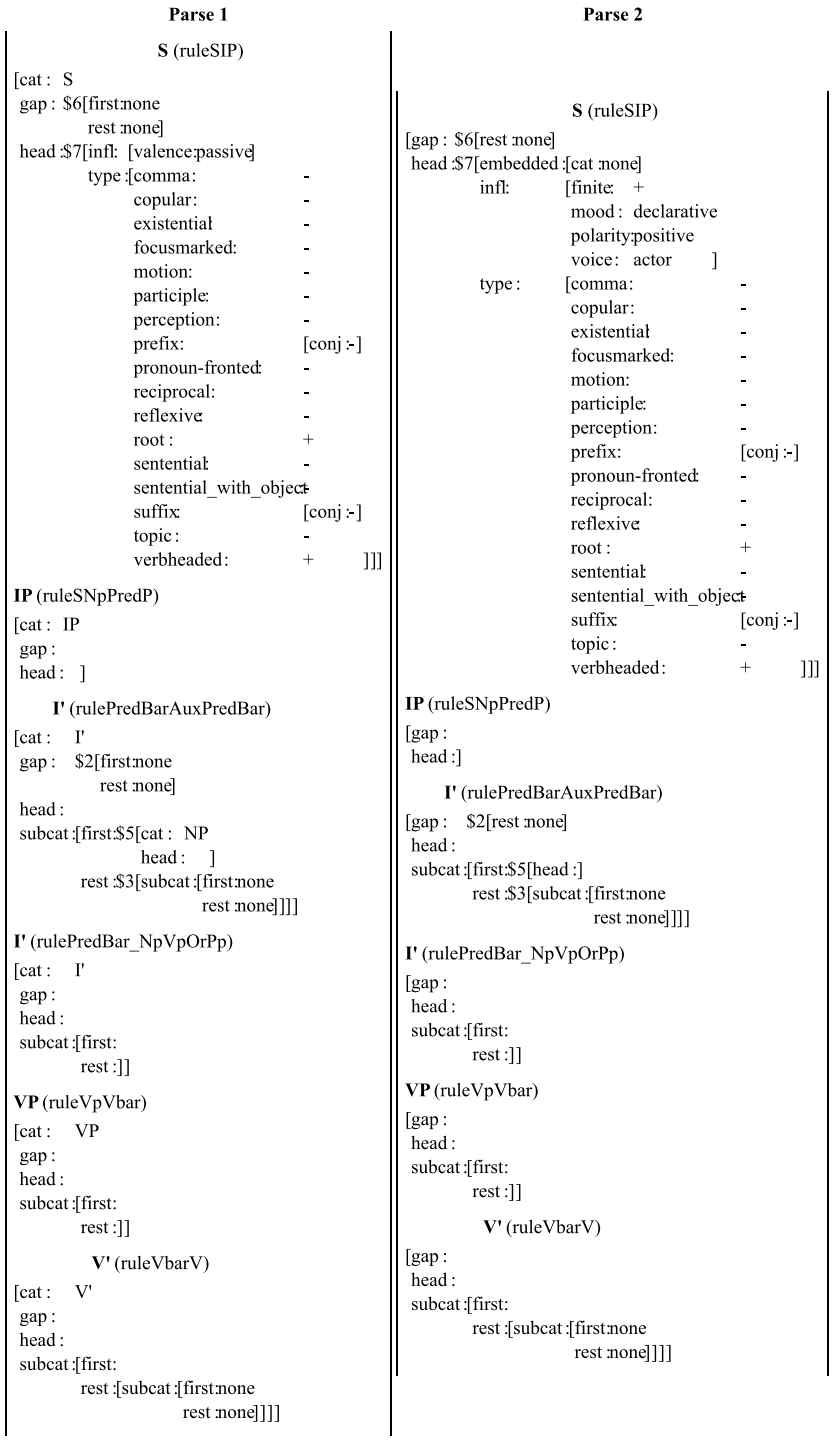

Figure 28. Sentence comparison tool

In Figure 28, the *subcat* feature S node in Parse 2 has the value *[rest : none]*, but *[first : none rest : none]* in Parse 1. The value *first : none* is present in the feature structure

for Parse 2 in the parser output, but has been factored out by the comparison tool. Sadly, it has failed to remove *rest : none* as well. This tool needs work before it will be truly useful, but it was used in the case study described in Section 4.3.

3.2.4.3 Node comparison tool

PC-PATR-EDITOR has a (stable) node comparison tool that allows users to select nodes from the parse tree and add them to a list to compare. Once the user has selected the desired nodes, the feature structures are displayed as a web page in the user's browser. It differs from the sentence comparison tool described in Section 3.2.4.2 because it only displays the nodes selected by the user and does not analyze their feature structures. This tool is accessed by clicking Show Node Comparison Window in the Analysis Tools menu (see Figure 26 above).

Figure 29 shows the Node comparison window with four nodes added for comparison. It is programmed to always be the top window so users can change parses and click on the parse tree viewer without having to select this window again. In this image, the user has added two nodes from two different parses. The name of the node is shown with the parse it was taken from in parenthesis to the right.

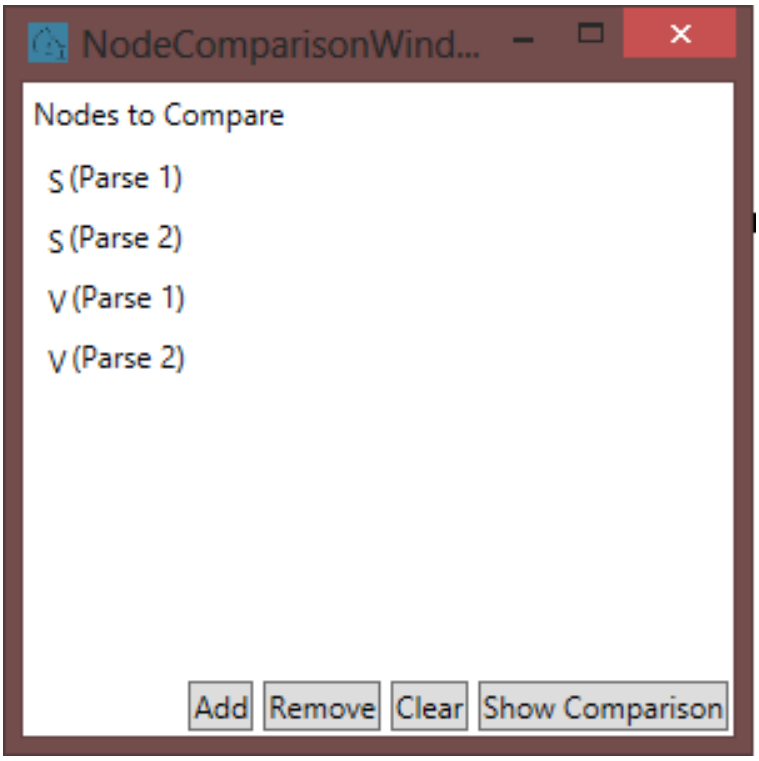

Figure 29. Node comparison window

Nodes are appended to the comparison list by clicking on them in the parse tree and clicking Add on the node comparison window. To add nodes from different parses, the user only needs to select the desired parse in the analysis pane and click on the desired node in the parse tree and click Add. Users can freely switch back and forth between parses, adding nodes. A node can be removed from the comparison by selecting it and clicking the Remove button and the entire list can be emptied by clicking Clear. The comparison is produced by clicking the Show Comparison button.

Figure 30 shows the report generated by clicking Show Comparison for the list in Figure 29. Comparisons are output on one line horizontally, but I have moved the two V nodes to a second line for the sake of this paper. From top to bottom, each node is displayed with the parse it comes from, the name of the rule that built it (if any), and its feature structure. This report shows that the two S nodes were built by different rules, but otherwise have the same features. The two V nodes have "no identifier found" for their Rule Identifier.

This means that there was no rule listed in the parser output, in this case, because they are terminal nodes.

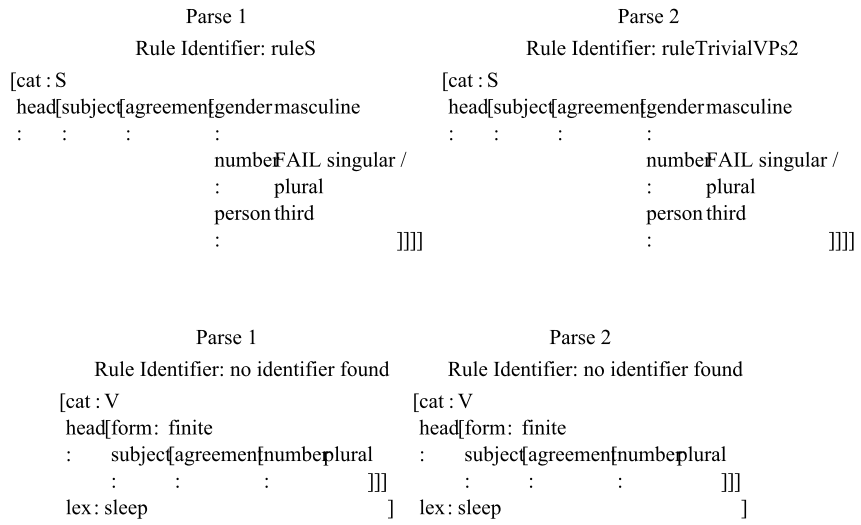

Figure 30. Node comparison report

3.2.5 Summary

PC-PATR-EDITOR solves the problems users face in reading PC-PATR's output by extracting information about failed parses and displaying both failed and passed parses in a form familiar to most linguists. The editor displays each sentence and its analysis of it in a list. For each sentence, the editor allows users to view the parses returned and what nodes (if any) failed along with the identifier of the rule that built them. For each node a quick list of failed features and a full feature structure is available. Parse trees are displayed as directed graphs and users can access a node's feature structure by clicking on it.

The editor also offers additional tools to aid in analysing output. Before the parser is run, the editor checks to see if the words in the input file have a corresponding entry in the lexicon file used. If not, then a list of missing words is displayed. A sentence comparison tool compares nodes in each parse for a given sentence and tries to factor out common features; this tool still needs work. A node comparison tools allows users to select nodes from the parse tree and add them to a list that can be output to the user's web browser

along with their feature structures. These features and tools conspire to solve the issues in understanding the parser's output by making its data accessible quickly and easily.

PC-PATR-EDITOR helps prevent content loss by allowing rules, constraints, and feature templates to be enabled or disabled. The editor also allows these elements to be flagged with an attribute, Use When Debugging, that specifies whether the element should be included when the parser is run in debug mode. Debug mode allows users to keep the grammar intact and still work with a limited portion while fixing an error or implementing a new construction. These two attributes eliminate the need for users to create multiple versions of the grammar file or comment out and delete elements to work with different versions and subsets of the grammar.

CHAPTER 4

CASE STUDIES IN BAHASA INDONESIA

This chapter contains case studies that describe how PC-PATR-EDITOR was used to develop a grammar fragment for Bahasa Indonesia. This fragment covers basic clauses with active and passive verbs, some relative and embedded clauses, and other constructions. These studies illustrate how elements of the analyzer component of PC-PATR-EDITOR helped discover the reason for failures and redundant parses generated by the grammar. The data used is primarily from Sneddon (1996) with some data taken from Larasati (2012) and examples are from Sneddon (1996) unless otherwise noted. These case studies are from my work implementing Bahasa Indonesia's passive constructions.

I began the grammar by working through the PAWS Starter Kit (Black and Black 2009,2012). The grammar was refined using a X-Bar theory style analysis. Subcategorization templates are used to express required complements and specifiers following a similar implementation by C. Black (1997). Under this analysis most phrase structure rules are binary branching.

Bahasa Indonesia (ISO 369-3 ind) is a Malay language with 22,800,000 speakers in Indonesia and with a total of 23,200,480 speakers worldwide as of 2000 according to the Ethonologue (Lewis 2009). It is an SVO language though other word orders do appear to indicate focus (Sneddon 1996:256-257). Example (1) illustrates normal clause order in a transitive clause; its parse tree is shown in Figure 31. Note that the prefix *meN-* is usually used to mark a transitive verb and that its final consonant undergoes nasal place assimilation.[1]

[1] The example sentences used in these case studies contain pronouns in many cases instead of common noun phrases. This is because they are taken from Sneddon (1996) and that is what he used.

(1) Saya mem-bantu ibu
1SG AV-help mother,
I am helping mother Sneddon (1996:241)

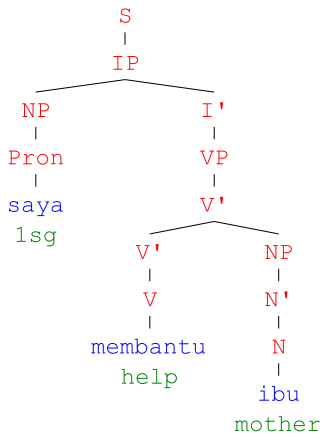

Figure 31. Parse tree for example (1)

In this example, *saya* ‘ 1SG ’ is the subject[2] or agent, *membantu* ‘help’ is a transitive verb, and *ibu* ‘mother’ is the object or beneficiary. In the implemented grammar, the subject is the specifier of the IP node and the object position is an NP dominated by V'. Bahasa Indonesia does not exhibit case marking as illustrated by the word *saya* appearing in the same form in both subject position in (1) and object position in (2). Figure 32 shows the parse tree for (2). Notice that *saya* is in the same structural position as *ibu* in Figure 31 and has not undergone any morphological changes.

(2) Dia men-jemput saya
3SG TR-met 1SG
He met me Sneddon (1996:247)

[2] The notions of subject and object are debated in linguistics literature. However, the status of the terms is not tremendously relevant to illustrating the usefulness of PC-PATR-EDITOR for drafting and revising PC-PATR grammars. Subject could be easily read as agent or actor, and object could be patient or undergoer.

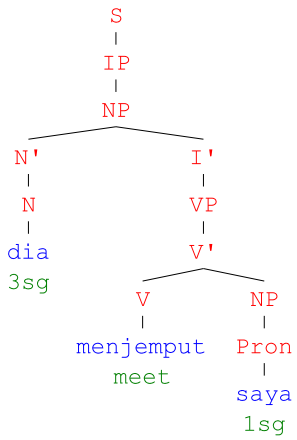

Figure 32. Parse tree for example (2)

In Bahasa Indonesia, the subject position is usually linked with the agent semantic role for transitive clauses. If the speaker wants to place more emphasis on the patient or object, then a passive construction can be used. The language has two main passive constructions called 'passive type 1' and 'passive type 2' by Sneddon (1996) following Dardjowidjojo (1978). The choice between types is based on what form is used to encode the agent in a passive clause. Type 1 is used when the agent is omitted or when the agent is third person (either a noun or pronoun). Type 2 is only used when the agent is explicit and is encoded by a pronoun. If the agent is present and is a third person pronoun then either type can be used.

In type 1 passives the patient takes the subject position occupied by the agent in a transitive clause; the agent (if present) is expressed by an NP or PP placed after the verb; and the verb is marked with the prefix *di-*. Example (3) shows the type 1 passive version of (2).

(3) Saya di-jemput oleh dia
1SG PASS-met by 3 SG
I was met by him Sneddon (1996:248)

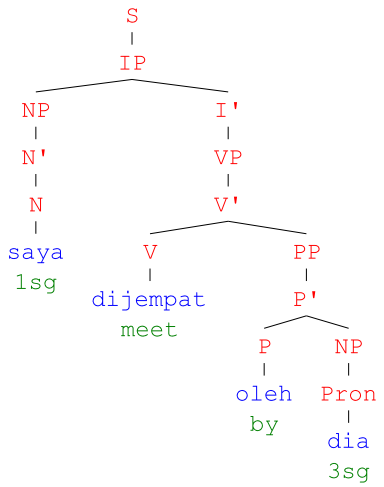

Figure 33. Parse tree for example (3)

Notice how *saya* has been moved to the specifier of IP and *dia* has been moved to a PP dominated by V'.

As noted above, type 2 passives are used with an overt pronominal agent. They are formed by placing the patient in the subject position and by putting the agent in an NP directly before the verb. Note that auxiliaries and other elements are not allowed to come between the agent and the verb. Example (4) shows both the active sentence *kami menjemput dia* and its passive type 2 version.

(4) a. kami men-jemput dia
1PL.EXCL AV-met 3SG
we met him Sneddon (1996:249)

b. dia kami jemput
3SG 1PL.EXCL met
he was met by us Sneddon (1996:249)

Figure 34 shows the parse trees for (4a) and (4b) from left to right respectively.

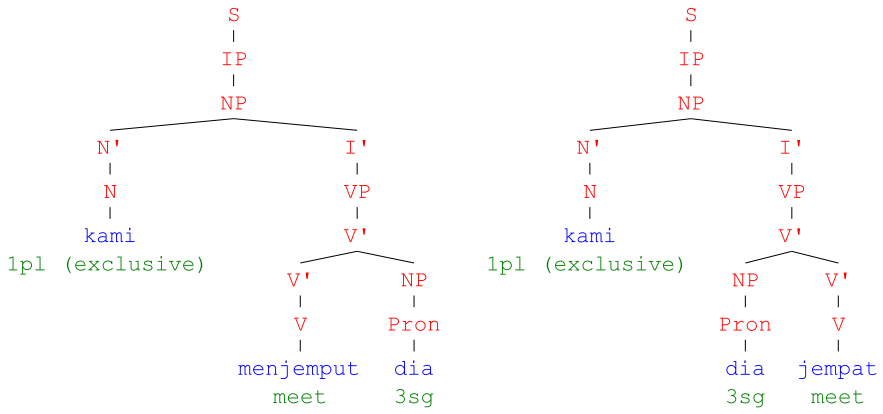

Figure 34. Parse trees for examples (4a) and (4b)

The agent in both examples is encoded by the pronoun *kami*. Notice how it has been moved from the IP's specifier position in the first tree in Figure 34 to an NP node preceding the verb dominated by V' in the second tree.

4.1 Passive type 1

In order to account for type 1 passives, I added the feature template shown in Figure 35.[3] It states that type 1 passives subcategorize for an NP and a PP. I did not need to change any phrase structure rules because this template and the passive phrase structure is similar to those of active transitive verbs.[4]

[3] I wrote most of this case study before I realized that the actor in a passive type 1 construction can be expressed by an NP. So the second element of the subcat disjunction in Figures 35 and 41 was added after the fact for completeness. I went through a similar process to get the complete template working as described in this section.

[4] I designed the subcategorization system to have two lists, one for subjects/specifiers and one for complements. Because the editor does not allow other elements to come between the features subcat and first or rest, I used an embedded subcat list so I could still step through the list of complements. When reading the templates shown in this chapter, it may be helpful to think of the first *subcat : first* and *subcat : rest* as *subcat : specs* and *subcat : comps* respectively.

```
Let PassiveType1 be
[subcat : [ first : [ cat : NP ] ]
            rest : [ subcat : {[ first : [ cat : PP ]
                                 rest : none ]
                               [first : [ cat : NP ]
                                 rest : none ] } ] ]

[head : [ infl : [ valence : passive ]
          agr : [ person : 3rd ] ]
```

Figure 35. Passive type 1 feature template (initial)

I ran the grammar on the example data shown in example (10) in Appendix C. The sentence shown in (5) failed. Figure 36 shows PC-PATR-EDITOR's analysis of this sentence.

(5) saya di-jemput=nya
1SG PASS-met=3SG
I was met by him Sneddon (1996:248)

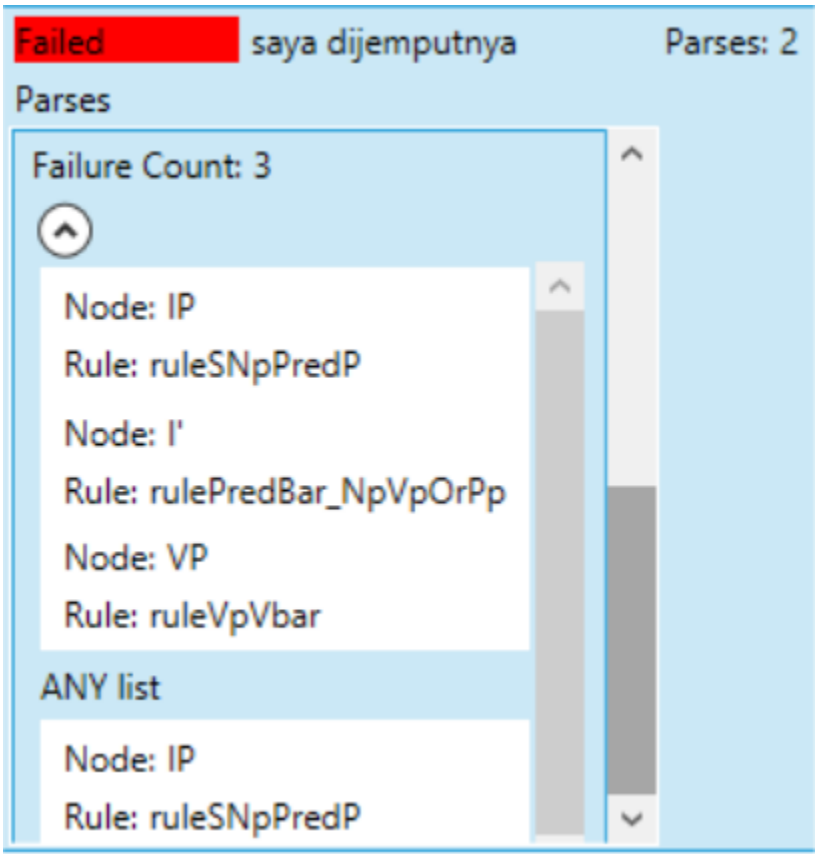

Figure 36. PC-PATR-EDITOR's Analysis of failure of example (5)

In this case, the editor indicated that PC-PATR returned two failing parse attempts. In this image I have selected the second parse which contained three points of failure. According to the analysis, it has three failing nodes: IP, I', VP. The parse tree generated by the editor is shown in Figure 37 with a minus sign next to the node names for IP and VP to indicate that they failed. This is the correct structure, but it contained unification failures.

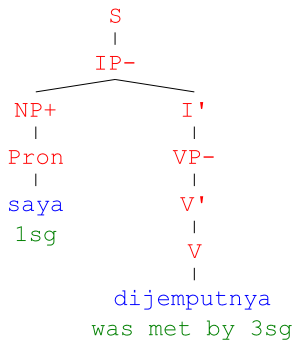

Figure 37. Parse tree for failure of example (5).

Figure 38 shows the details of the VP node failure that I got by clicking on "Node: VP" shown in Figure 36. Notice that this detail view shows the name of the node, the rule that built it and two lists. The first list contains any failed nodes. The second list contains any nodes that parsed with ANY for content.[5] In the VP node, the only failed feature was *first*. The output indicated it was assigned two conflicting features: *none* and a feature structure. By clicking on the VP node in the parse tree, the analyzer displayed its full feature structure (shown in Figure 39). The *subcat rest subcat first* feature had a value of *[first : [cat : PP] rest : none]*. This is the expected value from the PassiveType1 feature template (Figure 35). These facts indicated that the value *none* was introduced at the VP node.

[5] ANY is used in Functional Unification Grammar to indicate a feature "unifies with anything" but must not be present in the final feature structure Shieber (2003:36). In the PC-PATR XML output this is represented as <Any />. In PC-PATR feature notation, this is represented as [ ].

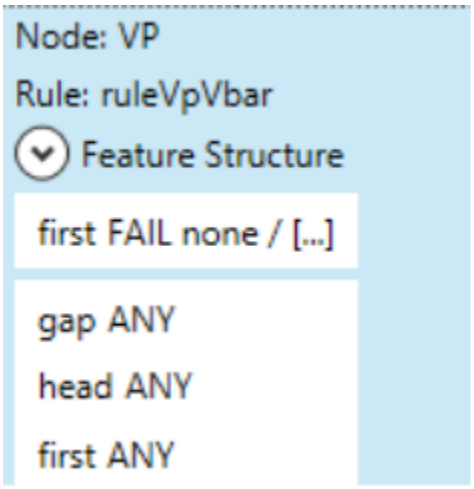

Figure 38. Analysis information for VP node in example (5)

```
[cat :          VP
 gap : ANY
 head : ANY
 subcat :       [first : ANY
                 rest :        [subcat : [first : FAIL none / [...]]]]]
```

Figure 39. Feature structure for VP node in example (5)

The rule that built the VP node is shown in Figure 40. This rule requires that the *subcat rest subcat first* value of V' be *none*. I wrote it this way to ensure that all required complements had been found for a well formed tree. I also wrote the V' rules for PP and NP complements with this unification constraint: *<V'_1 subcat rest subcat first> = <V'_2 subcat rest subcat rest>*. It passes the rest of the child's *subcat rest subcat rest* list up the tree as the *subcat rest subcat first* value for the node it builds.

```
rule {ruleVpVbar} | Vp V'

VP = V'

<VP head> = <V' head>

<VP subcat first> = <V' subcat first>

<VP subcat rest subcat first> = none

<V' subcat rest subcat rest> = none
```

Figure 40. VP rule

If the grammar had worked as expected, complement rules should have assigned *none* to the *subcat rest subcat first* list for the V' child of VP. Because this sentence has no PP, the PP in the <*subcat rest*> list of V' was percolated to the VP rule where it failed to unify.

I changed the template shown to make the PP (and NP) optional by introducing a third disjunct with *none* for *first* and *rest* values shown (highlighted) in Figure 35. When I ran the grammar again, it parsed correctly.

```
Let PassiveType1 be
[subcat : [ first : [ cat : NP ] ]
            rest : [ subcat : {[ first : [ cat : PP ]
                                 rest : none ]
                               [first : [ cat : NP ]
                                 rest : none ]
                               [first : none
                                 rest : none ] } ] ]

[head : [ infl : [ valence : passive ]
          agr : [ person : 3rd ] ]
```

Figure 41. Passive type 1 feature template (final)

For Bahasa Indonesia's passive type 1 construction, the editor was most helpful in highlighting where errors occurred and in providing an easy way to view the feature structures at failures and various other nodes. While it did not hand me the solution, it made it easy to gather enough data to point me in the right direction.

4.2 Passive type 2

Bahasa Indonesia's second passive construction places the patient in the subject position and the agent directly before the verb. This type of construction is used when the agent is explicitly expressed by a pronoun. To implement this construction, I created the rule shown in Figure 42. It allows a V' to be constructed from an NP and a following V'. The second V' is required to be passive and must subcategorize for an NP in its *subcat first* list. Finally, it percolates the *head* and *subcat rest* features to the dominating V'. I assumed that I could use the transitive verb feature template because both type 2 passives and transitive verbs have an agent and a patient in their subcat lists. I also assumed that the V'_2 node in Figure 42 would be unspecified for voice up to this point and would be assigned passive here.

```
rule {ruleVbarNPVbarPassiveAgent} | V' NP V' - passive agent rule

V'_1 = NP V'_2

<V'_1 head> = <V'_2 head>

<V'_2 head infl valence> = passive

<V'_2 subcat first> = <NP>

<V'_1 subcat rest> = <V'_2 subcat rest>
```

Figure 42. Passive type 2 agent NP rule (initial)

I ran the grammar and it did not work as expected. Example (6) (repeated from example (4b)) failed. The rule that builds V' from V in embedding clauses interfered so I disabled it. This removed some failures. Figure 43 shows the parse tree for this sentence. The V' node dominating the NP dominating *kami* indicates that it failed.

(6) dia kami jemput
3SG 1PL.EXCL met
he was met by us Sneddon (1996:249)

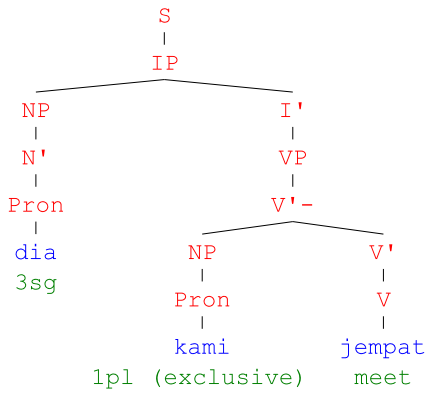

Figure 43. Parse tree for failure of example (6)

By clicking on the V' node in the parse tree viewer, the feature structure in Figure 44 was shown. Notice how the *valence* feature had the value *FAIL passive / active*. This indicated that it at this point *valence* was being assigned values that could not unify. This node was built by the passive agent rule shown in Figure 42 which requires that the V'_2 be *passive*. In this parse the verb *jemput* 'meet' is marked with *valence : active*. When I looked at the feature template for V, I found that it is assigned active voice by default. I created the lexical rule shown in Figure 45 to overwrite the default active value with passive and added it to the relevant lexical entries.

```
[cat :     V'
 head :   [infl : [valence :FAIL passive / active]]
 subcat : [rest : ANY ]                              ]
```

Figure 44. Feature structure for V' node in example (6)

```
Define PassiveType2 as

<out head infl valence> = passive
```

Figure 45. Passive type 2 lexical rule (initial)

Running the grammar returned the analysis shown in Figure 46. It had the status "Passed with ANYs". This means that the grammar could build a parse for the sentence but that some features were not specified with a value. Clicking on the V' node showed the feature *rest* had the value *ANY*.

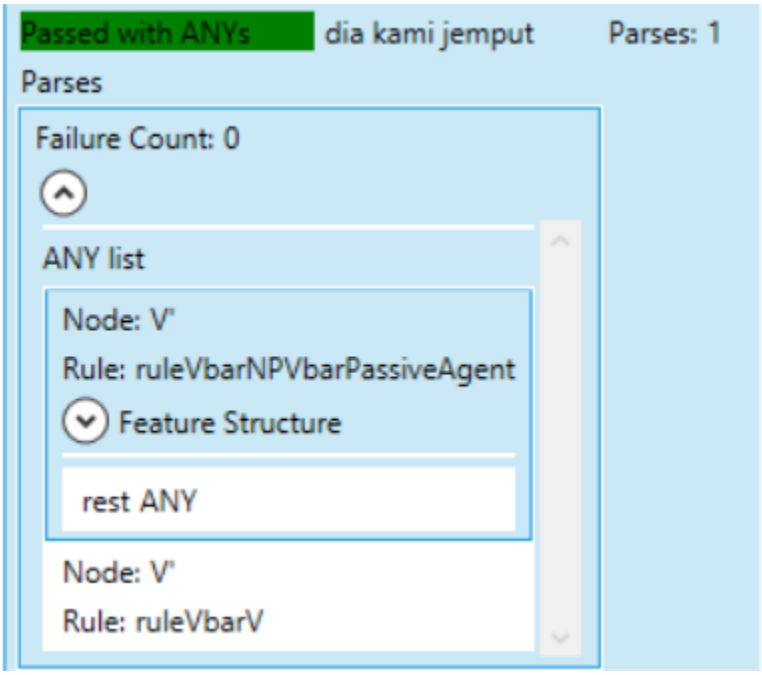

Figure 46. Analysis of example (6) after adding the PassiveType2 lexical rule

The *subcat* feature structure for V' shown in Figure 47 shows that *ANY* is the value for *subcat rest subcat rest*. Notice that *subcat first head* has the index "$7". This coindexes this feature structure with the *head* feature for the NP dominating *dia*. So the *subcat first* is what I expected to find since this feature is used for the subject position (specifier of IP). The presence of *ANY* at subcat rest subcat rest was perplexing, as it should have been assigned *none* by the transitive verb template. This made me wonder if the lexical rule was not passing the subcat from the input active verb to the output passive.

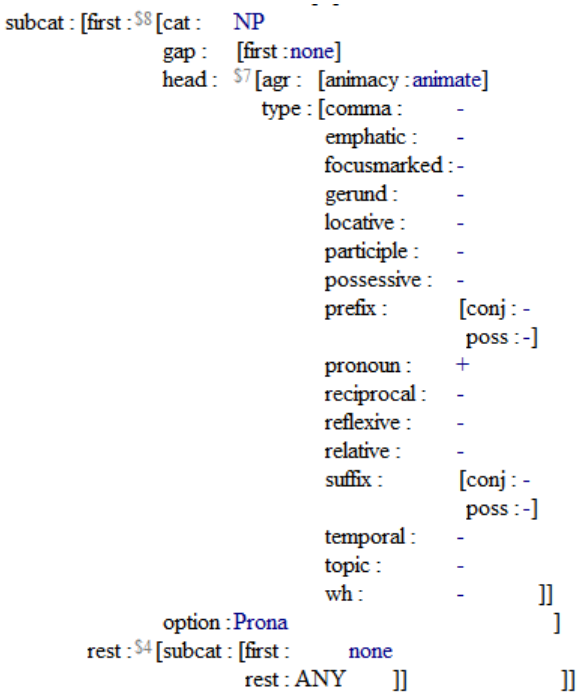

Figure 47. Subcat feature structure for V' in example (6)

I changed the lexical rule to the form shown in Figure 48.

```
Define PassiveType2 as
<out head infl valence> = passive
<out subcat first> = <in subcat first>
<out subcat rest> = <in subcat rest>
```

Figure 48. Passive type 2 lexical rule (final)

Running the grammar at this point returned two failed parses. In the second, the V node has active voice so I figured it was the version of *jemput* without the PassiveType2 lexical rule. The first parse had one point of failure, this time at the VP node. There were no failed features shown at this node. The *subcat* feature still contained *ANY*, but this time at *subcat first*. The VP rule that built this node is shown in Figure 40 in section 4.1. This rule requires that both *subcat rest subcat first* and *subcat rest subcat rest* be *none* to be sure that all complements have been found.

The *subcat rest subcat first* feature at the VP node was *none*, but by checking the *subcat* feature for its child V' node, I found the feature structure shown in Figure 49. The NP at *subcat rest subcat first* would not pass the unification constraints in the VP rule, but this did show that the new version of the PassiveType2 lexical rule worked.

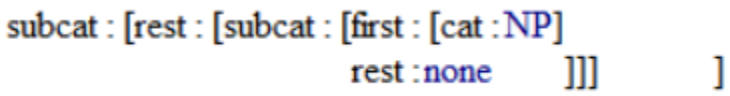

Figure 49. Subcat feature for V' node after modifying PassiveType2 lexical rule

The initial version of the passive agent rule (Figure 42) built a V' from an NP followed by V'. It unified the NP with the *subcat first* feature of V'_2 and percolated *subcat rest* up the parse tree without modification. The VP rule (Figure 40) is designed to require this list to be *none* for all complements. Thus these two rules were passing features in a way that could not unify. To solve this problem, I decided to use the *subcat rest subcat first* feature to hold the agent instead of *subcat first* as in the active transitive verb template in Figure 50.

```
rule {ruleVbarNPVbarPassiveAgent} | V' NP V' - passive agent rule
V'_1 = NP V'_2
<V'_1 head> = <V'_2 head>
<V'_2 head infl valence> = passive
<V'_2 subcat rest subcat first> = <NP>
<V'_1 subcat first> = <V'_2 subcat first>
```

Figure 50. Passive type 2 agent NP rule (final)

By changing the passive agent rule to the form shown in Figure 48, the rule would identify its agent NP with the element at V'_2's *subcat rest subcat first* and percolate its *subcat first* up the tree. Running the grammar with these changes produced the correct

structure shown in Figure 51. The green and red boxes around the NP nodes indicate what feature structure they are coindexed with in Figure 52.

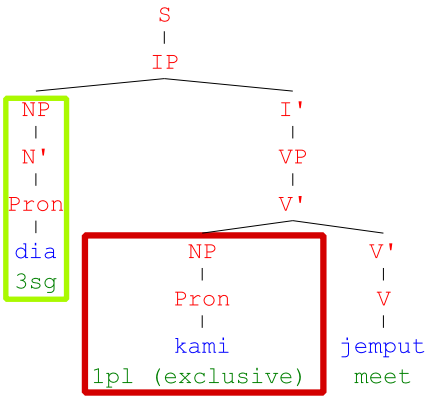

Figure 51. Correct parse structure for example (6)

Figure 52 shows the feature structure for the V' node dominating V. The *head* feature inside *subcat first* (green box) has a index number of "$5" that coindexes it with the NP node dominating *dia* and the *head* feature inside *subcat rest subcat first* has a index of "$1" (red box) that coindexes it with the NP dominating *kami*. This means that the subcat features unified as expected. Thus the correct structure and subcat features were achieved.[6]

[6] These rules are pragmatic since they don't alter the input subcategorization list. I tried to write a lexical rule that did this, but it did not work because the subcat rest has an embedded linked subcat list.

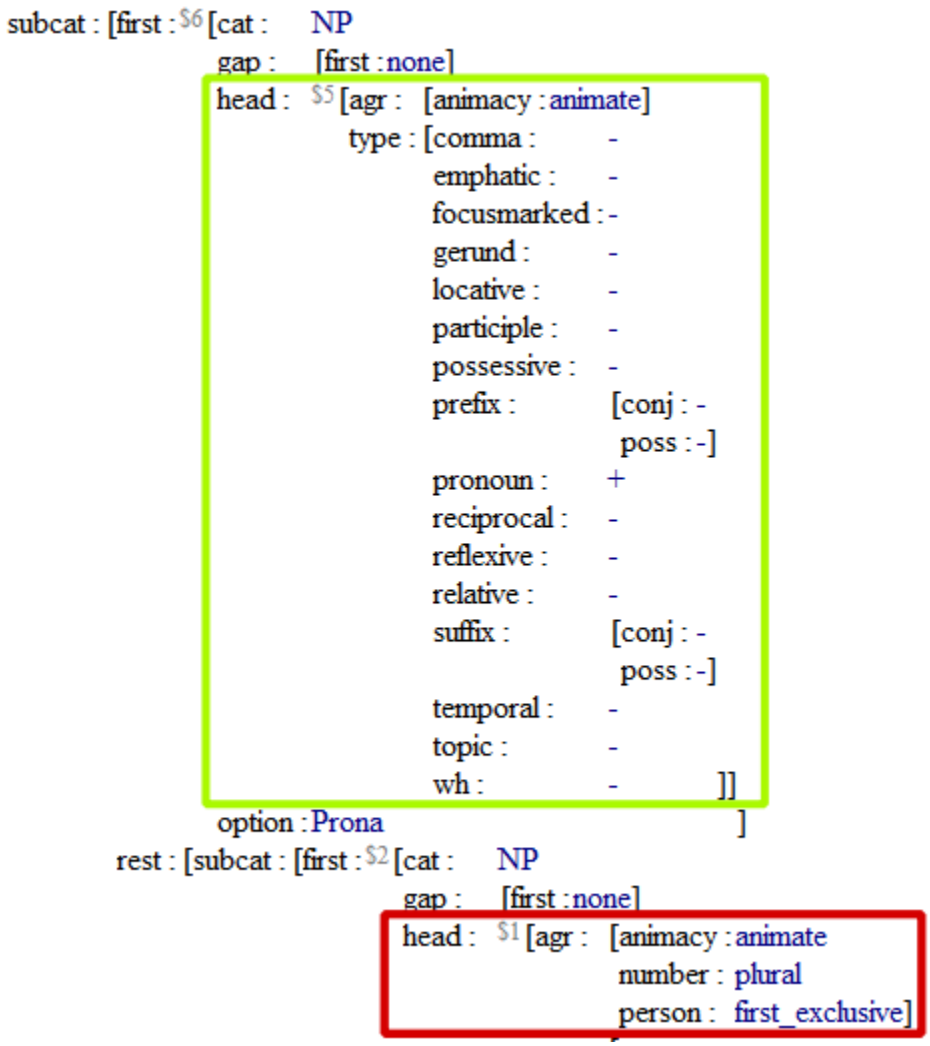

Figure 52. Feature structure for V' node dominating V in successful parse of example (6).

4.3 Passive type 2 with agent expressed by clitics

Bahasa Indonesia has both free and bound pronouns. The table in example (7) (based on Sneddon (1996:165)) shows the bound forms of the pronouns and their corresponding free forms. Note that not all free forms have a corresponding bound form and that *=nya* can be singular or plural, corresponding to *dia* or *mereka*. All other forms are singular.

(7)

	prefixed	suffixed	free
first	ku=	=ku	aku
second	kau=	=mu	engkau, kamu[7]
third		=nya	dia, mereka

The bound pronouns can be used to express possession, the object of an active transitive verb, or the agent of a type 2 passive verb (Sneddon 1996:165-166). It is common for first and second person agent pronouns to appear as proclitics on the verb (Sneddon 1996:249). This is shown in example (8). The patient is *buku ini* 'this book' and the agent is indicated by the bound pronoun *kau*= ' 2SG '.

(8) buku ini harus kau-baca
book this must 2SG=read
You must read this book Sneddon (1996:249)

PC-PATR cannot treat the bound form of the pronouns as a separate word because I am not using any morphological parsing in this project. Consequently, there would not be any explicit element to step through in a subcat list. Therefore, to account for these passives I added the PassiveType2 lexical rule to these entries in the lexicon (Figure 48 in Section 4.2). I also added a new feature template shown in Figure 53. This template is the transitive verb template with the NP at *subcat rest subcat first* (highlighted) set to none because it is satisfied by the clitic.

```
Let AgentAffixedPassiveTransitiveVerb be
[subcat : [first : [cat : NP]
           rest : [subcat : [first : none
                             rest : none]]]]
```

Figure 53. Type 2 passive template with agent clitic (initial)

[7] In Bahasa Indonesia pronouns carry social meaning. Both of these forms have a connotation of a close relationship (see Sneddon (1996:160-165) for discussion)

Initially, when I ran the grammar, there were many parses produced. By looking at the V' nodes in a few of them, I noticed that multiple rules were building them, including one that was only supposed to apply in embedded clauses. This rule allows V' nodes to be built by adding a required complement to the node's *gap* list instead of requiring it to be present in the clause. I disabled this rule and then ran the grammar again.[8]

This time, example (8) passed. The grammar generated the correct structure but with two isomorphic parses. The parse tree for example (8) is shown in Figure 54. The analyzer offers a sentence comparison tool (see Section 3.2.4.2) that compares all parses for the selected sentence and factors out common features. The result is displayed as a web page. Using this tool, I noticed that the features in the nodes dominating the V node shared many features in both parses and that one parse had more features specified than the other. I guessed that the differences were coming from the V node in both parses.

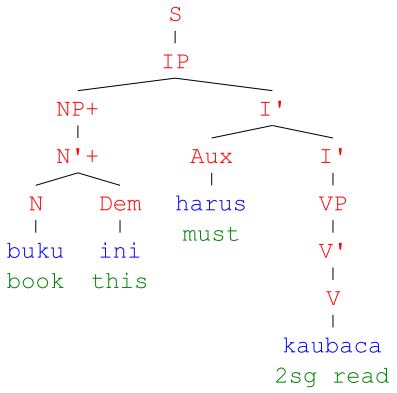

Figure 54. Parse tree for example (8)

The analyzer also provides a node comparison tool (see Section 3.2.4.3) that lets users click on a node in the parse tree and add it to a list to compare. Once the user has selected the desired nodes, he or she can produce a web page that shows the feature structure and the name of the rule used for each feature in the list. Using this tool, I compared the V nodes from both parses; the result is shown in Figure 55.

[8] When I enabled the rule after finishing the grammar for this construction, this rule did not continue to apply in this construction.

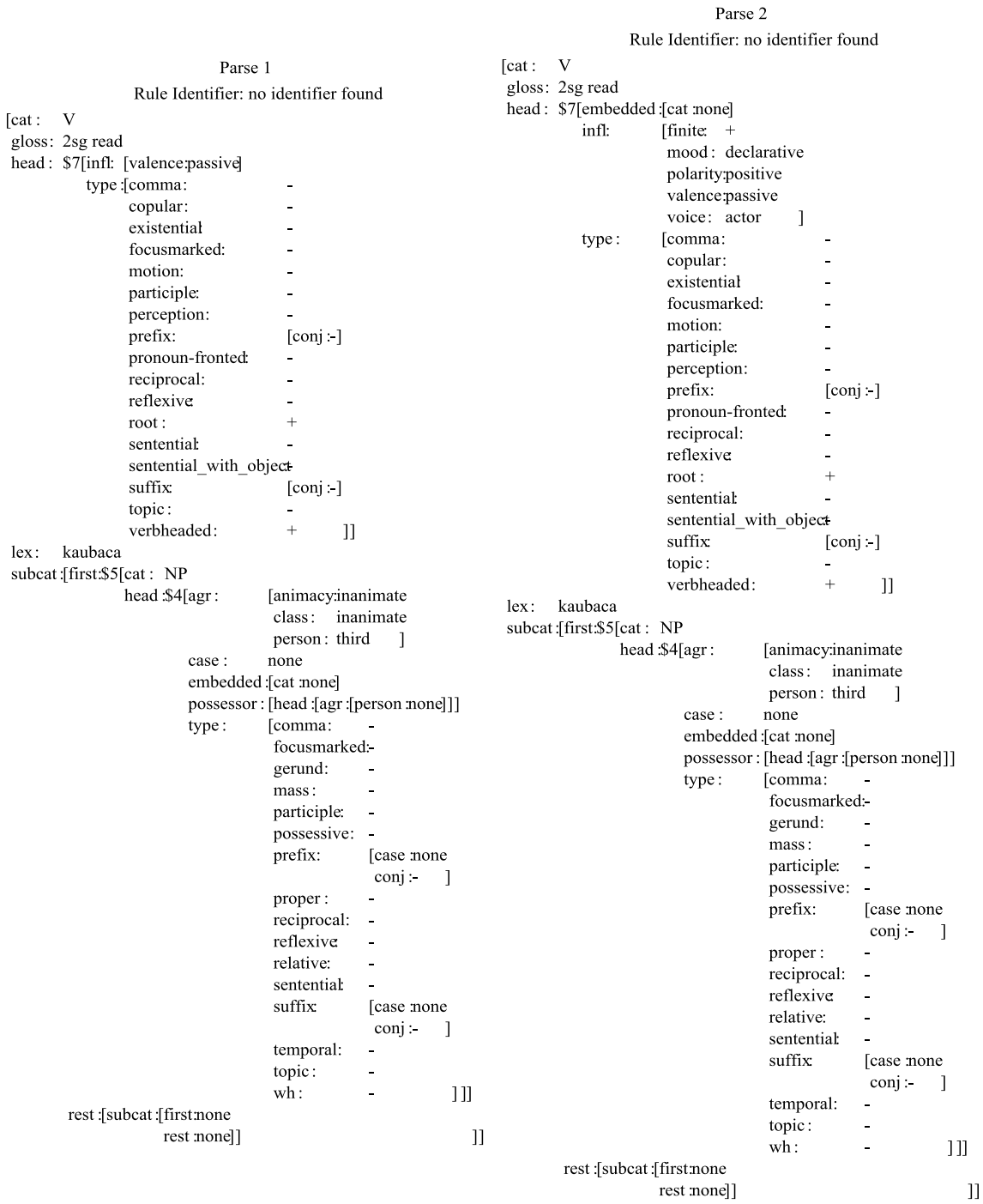

Figure 55. Comparison for V nodes in parses of example (8).

I noticed again that there were many features in one parse but not in another. I was suspicious of the lexical rule because of my experience implementing type 2 passives with unbound pronouns. I moved the passive feature assignment from the lexical rule to the

AgentAffixedPassiveTransitiveVerb feature template as shown in its final form in Figure 56. I ran the grammar and the extra parse was gone.

```
Let AgentAffixedPassiveTransitiveVerb be
[subcat : [first : [cat : NP]
           rest : [subcat : [first : none
                             rest : none]]]]
<head infl valence> = passive
```

Figure 56. Type 2 passive template with agent clitic (final)

4.4 Summary

In my experience using PC-PATR and PC-PATR-EDITOR, the primary difficulties in revising a grammar are eliminating redundant parses and stopping multiple rules from applying to the same input. While the editor cannot develop its own solutions, it provides easy ways to see what is going on when these things happen. The workhorse tools are the parse tree and feature viewers. The other comparison tools are helpful when the problem is of a larger scope or it would be nice to see all the data at one time.

For most of the problems in the case studies, I was able to figure out what was going on by simply clicking on a failed node in the parse tree and seeing what rule built it and what features it had. In the grammar fragment I have two rules that build a V' from a V. One is only supposed to operate in embedded clauses. In Section 4.3 the embedded clause rule was interfering. By clicking on the V' node in the parse tree I was able to see what rule built the node and knew that I should disable this rule while trying to work out type 2 passives with a clitic agent.[9]

The editor also returns three different statuses for a given sentence: Passed, Passed with ANYs, and Failed. These are helpful for knowing how the grammar is working. Passed with ANYs means that the grammar is building the correct structure but that it has some features that are not being assigned. This helps catch false positives or when a grammar is almost

[9] When I enabled the rule again, it did not interfere or produce extra parses.

working. In Section 4.2, I had fixed the failures in the type 2 passive grammar, but the *subcat* feature was still not being assigned as I wanted. It was this status that caused me to look at the feature structures in this parse again.

Unification failures can be difficult to track down as the failing feature may have originated far from the point of failure. Furthermore, it may have been manipulated by the intervening rules. The sentence comparison and node comparison tools allow users to see how features flow from one node to another. These tools also help when it is useful to see data from across the current or multiple parses at one time. In Section 4.3, the node comparison tool helped me see the similarities in the V node across the parses. This led me to suspect the lexical rule I was using to mark these verbs as passive.

CHAPTER 5

CONCLUSION

PC-PATR is a powerful parser implementing the PATR-II formalism. This formalism is unification based, order independent, and uses concatenation as its only string combining operation. It is not tied to a particular language or theory of grammar. PC-PATR has augmented it by adding priority unions (an overwrite operation) and logical constraints for additional power.

The parser is one of many parsers and formalisms available in the linguistics world. This paper reviewed eXtensible MetaGrammar (XMG), the Stanford Parser, Leopar, and the Link Parser. XMG is a formalism for building grammars for lexically oriented theories that use tree fragments to define syntactic structure; it requires its own tool set to compile grammars. The Stanford Parser is a probabilistic context free parser that learns its grammar from a training corpus of marked up parse trees. Leopar is a parser for French based on Interaction Grammar (IG). IG combines tree fragments to build syntactic structures and uses feature structures to control how fragments combine. The Link parser is language independent but is tied to Link Grammar. This theory views words as containing link points that must be satisfied by linking to compatible links in other words for a sentence to be grammatical. Link Grammar has no concept of hierarchical constituent structure.

PC-PATR differs from these parsers in a number of ways. Unlike Leopar, it is not tied to a particular language. PC-PATR uses context free rules to define constituent structure like the Stanford Parser, but is not probabilistic and does not learn a grammar. It differs from Leopar and XMG by defining constituent structure with phrase structure rules instead of tree fragments. Like both XMG and Leopar's IG, PC-PATR is unification based, has

feature structures, and its rules are order independent (unless priority unions are employed). It is language and theory independent unlike Leopar and the Link Parser.

Human interaction with PC-PATR currently faces two main difficulties: the plain text nature of its inputs and the difficulty of reading its output. Grammar files are currently plain text files. This requires users to respect their syntax and offers no method for checking semantic correctness. Though the parser uses feature structures, grammar files do not define a feature system; it is created as it is used. Grammar files do not define a list of terminal or non-terminal symbols so there is no way to ensure that phrase structure rules have the correct structure.

PC-PATR's output contains a lot of helpful information for users as they test grammars. This output is in either a verbose plain text or XML format. In both cases the user has to comprehend a lot of text to find errors. The plain text output does return a graphical parse tree, but that tree does not inform users what rule was used to build each node. The XML output does this, but it provides no graphical parse tree. Finally, feature structures can be quite large, so finding unification failures can be a lengthly process in either format.

To address these issues, I have developed an editor and analyzer called PC-PATR-EDITOR. This tool provides a structured editing environment for grammars and analysis tools for parser output. The grammars are written to an XML format developed by H. A. Black that includes a list of symbols and a feature system definition. The editor views rules and other elements as content within blocks that ensure that they have the correct content. Thus when editing grammar files, users do not have to worry about the syntax errors, but only have to select the desired element from a list and click Insert.

The analysis component of the editor processes the parser's output and highlights where errors have occurred. It provides a nicer display format for parse trees and shows the feature structure for a node in the tree when the user clicks on it. The editor has a sentence comparison tool that compares all parses for a given sentence displaying the result as a web page. It also has a tool that allows users to select specific nodes whose feature structures

they would like to be able to see together. Once the user has selected the desired nodes, their feature structures are displayed side by side for easy comparison. The editor also checks the words in the file to be parsed against the selected lexicon and informs users when words are missing from the lexicon.

PC-PATR-EDITOR was tested by developing a grammar fragment for Bahasa Indonesia. I began by working through the PAWS Starter Kit and used PC-PATR-EDITOR to revise the grammar following an X-Bar style analysis. Three case studies in Section 4 illustrate how different elements of the editor were helpful in this process. These case studies cover Bahasa Indonesia's two main passive constructions and deal primarily with writing rules and templates to make subcategorization features work correctly. Using the tool also highlighted areas where it needs to be revised and could be improved.

I designed the editor thinking about each type of rule, constraint, etc. as a separate piece. The editor presents the grammar as separate pieces to be edited, rather than as a coherent document. Thus to edit a rule, a user must find it in a list, then the rule is displayed in a separate page where edits can be made. I think the editor would be more effective if it were constructed around a document metaphor. This would provide a user experience like editing an academic paper or word document, rather than navigating to parts and pieces to make changes. It would also enable the user to view more of the grammar at one time.

The editor is a structured editor. This means that users are not free to copy, paste, or cut whatever parts they like, but must work with units as a whole. This prevents users from making syntactic errors when editing. This is a different experience from editing a plain text document where users can select and edit whatever pieces they want.

Currently PC-PATR-EDITOR only runs on Microsoft Windows because it was developed using Windows Presentation Foundation. This framework is not included in the Mono Project[1] and so it is not portable to Linux. The Silverlight framework is portable via Moonlight,[2] but I could not use Silverlight because it does not allow access to the XML parsing

[1] http://www.mono-project.com/Main_Page
[2] http://www.mono-project.com/Moonlight

library I used in the data access layer of the application. I did not know this until after I had written most of that layer. The project could be made portable to Linux by rewriting the user interface in Windows Forms which does allow access to the XML library. In Section 5.1 I recommend making some serious changes to the structure of the user interface, so changing the framework used to program it would probably not be all that difficult. Most of the other components not associated with the user interface could be reused.

5.1 Critique of the editor

Gomolka & Humm (2012) set forth the seven principles for what makes a good structured editor for programming languages. Though PATR-II is not a programming language, it does share some features with them. PC-PATR can be thought of as a runtime environment for PATR-II grammars, and grammars do not work or work in unexpected ways if they have too many syntax errors. I have paraphrased their principles below:

1. Focus on content, not syntax;
2. Do not overly restrain the user;
3. Keep presentation compact;
4. Provide a similar user experience to plain text editors;
5. Don't create dependency on a particular tool;
6. Let users customize how content is displayed; and
7. Offer both structured and unstructured editing modes.

Though the above principles are not dogma, I think they provide a helpful rubric to evaluate the editor and make suggestions for future improvements. The editor I have created may pass Principle 1 as users are not required to remember the syntax for grammar files and very little of it is displayed to the user. However, it fails principles 2 through 7. The user is very much restrained (Principle 2), both because the grammar is not displayed as a

document and because common operations like copy, paste, and cut are not possible. The current implementation of the editor's block structure (see Section 2.1) has rather large margins. What only takes up the height of a single line of text in a text editor takes two or three times that much space in PC-PATR-EDITOR. Thus the presentation is not compact (Principle 3).

XML is a great data storage format because it is (marginally) human readable, can be opened in standard text editors, can be easily transformed into different formats via XSLT transformations, and provides a means for checking semantic correctness in addition to syntactic via Schematron (2010). Furthermore parsing XML is not difficult and can be easily done by modern programming languages. But it does remove the ability to edit grammars in other text editors and requires that editors for grammars be able to parse and transform it into something more human friendly. Is the power of XML necessary as the plain text format could be extended and editors could be developed that check the correctness of the files?[3]

Regardless of the data format, PC-PATR-EDITOR would greatly benefit from being changed to use a document metaphor for displaying the grammar and implementing of copy, paste, and cut operations. Users would be able to see the entire grammar easily and moving and duplicating elements would be easier. In terms of user experience, the editor could become more or less structured. If the editor is developed along less structured lines, then an approach where users still define the feature system and terminal and non-terminal symbols would allow for autocomplete, error highlighting, and some level of semantic checking.[4] This approach would require developing a recognizer for PC-PATR's file syntax so errors could be found.

[3] H. A. Black has already written a tool that converts plain text grammar files to XML (2012). This requires parsing the plain text files and converting them to a separate format. Surely a recognizer could be developed as well. An extended version of the PATR-II formalism could be developed where features systems and symbol lists could be defined on lines delimited by the comment character and a special character such as '#'. This would mean that PC-PATR could be run on the file without the extensions causing errors and the double character line marker would provide an easy way to know what the content of those lines contained.

[4] Visual Studio and many other integrated development environments allow users to select lines of text and comment or uncomment all of them at the same time. This would be nice to include if the editor is developed in an unstructured way.

To continue with a structured editing experience, the content blocks need to become smaller and more clear (Principle 3). The method for choosing and inserting elements also needs to be more fluid.[5] The current system only has two kinds of blocks (content and list) and a grammar that defines how these blocks are composed (see Section 2). It requires that users choose what type block they want to insert from a list. This means that users cannot type a grammar but must use the mouse or track pad for most editing tasks. I recommend changing this so that the editor allows users to use the keyboard more.

Gomolka & Humm (2012) describe a structured editor for the Lisp programming language in which blocks are constructed based on the keywords typed by the user. PC-PATR-EDITOR could be developed along these lines. The editor would insert a generic block when the user types a keyword that would be aware of what types it could be based on the keyword entered. As the user types, the editor would evaluate what elements had been added until the block could be assigned a type. Once a type is assigned, then it could be evaluated for errors. Also if a type had required elements, these elements could be inserted automatically once the type is determined.

Table 1 shows an example of how this might work. In this figure, each row represents an action taken by the user and how the editor might understand it; the left column shows the user input and the right shows the possible types assigned by the editor. In the first row the user has typed "<". This keyword is associated with Feature Paths, Subcat Paths, and Category References. The second line does not disambiguate this any further. By the third row, a Category Reference has been ruled out. And by the fourth row, the user has typed enough text that this element must be a Subcat Path. If a block does not have all its required

[5] This may require changing the way the internal language for defining how blocks are allowed to compose works. The current Document Type Definition for PC-PATR grammar files is very detailed. As such it is easy to check that the content of the grammar is semantically correct as well as syntactically. For example, that features in feature paths have parent child relationships and that only non-terminal symbols are used on the left hand side of a phrase structure rule. But it also creates many different data objects and removes the possibility of editing grammars in text editors. For example, there is a representation for disjunctions in the plain text format ("{ a \ b }"). In the XML format there are separate markup tags for disjunctions in subcategorization templates, feature templates, phrase structure rules.

elements, then it could be flagged as invalid and the user forced to deal with it before saving or running the grammar.[6]

Table 1. Example block parsing process.

<	Feature Path, Subcat Path, or Category Ref
<NP	Feature Path, Subcat Path, or Category Ref
<NP head	Feature Path or Subcat Path
<NP head subcat	Subcat Path

This style would still be structured, but allow a more fluid editing experience like a plain text editor. Gomolka & Humm (2012) made an interesting observation about this style of editing. "This means, using a structured editor, the programmer directly works on the syntax tree of the program" and that the source text is not modified "until the user saves the current document or changes to the textual representation" (Gomolka & Humm 2012:88). This layer of separation provides the opportunity for data validation (which PC-PATR-EDITOR currently lacks) and means that invalid data could be saved to a separate file so that the last valid grammar could be preserved.

To convert the editor to use a document metaphor, I recommend a text based representation based on the look of the PATR-II formalism. For rules, constraints, and other elements that are part of the formalism, the representation could be modeled after their appearance in the plain text files. But PATR-II does not have a formalism for defining a feature system or lists of terminal and non-terminal symbols. This could easily be implemented using a format like the one shown in Figure 57 for the feature system and Figure 58 for symbols.

[6] If the user is allowed to save invalid data, then the data access layer of the application will need to be modified to account for elements that are missing data.

```
head {
        agr {
                person { 1, 2, 3 }
                number { singular, plural }
            }
        type {
                passive { +, - }
             }
    }
```

Figure 57. Example feature system textual representation

```
Non-terminal Symbols = [S, NP, VP]

Terminal Symbols = [N, V]
```

Figure 58. Example textual representation for symbols lists

One objection to this representation is that the XML format uses ID tags to refer to symbols and features. These ID tags could be supplied when the grammar was saved. The actual ids are largely irrelevant to the user and are a way to ensure consistency by defining things in one place only. This could be accomplished by requiring that feature and symbol names are defined before use and that the features and symbols used in the grammar appear exactly as defined. If the ID attribute needs to be explicitly available to the user, then it could be accessed when the user selects a feature or symbol through a side menu or detail view.

I think the editor would benefit from having a structured and unstructured editing mode (Principle 7). This would necessitate being able to handle a grammar that had syntax errors. If the textual form could not be parsed into a block structure, then it would be difficult or impossible to save it as XML. Thus the editor would need the ability to save both as plain text and XML. H. A. Black has already written a tool that converts plain text grammar files to XML; this tool could be used to convert the text to block structures via parsing the XML output or could be used as a starting point for the converter.

5.2 Critique of the analysis component

In general, I think the analysis component of PC-PATR-EDITOR performed well during the case studies (Section 4). The user interface provides quick access to the data needed to revise grammars. As noted in Section 3.2.4.2, the sentence comparison tool does not work properly. Hopefully, this could be fixed in a future version. The user interface elements that display the editor's analysis of the parser output could be cleaned up. List captions could be more informative and margins tweaked so things are easier to see.

As noted above, PC-PATR-EDITOR is a Windows only application. The user interface for the analysis page makes use of the data templating functionality built into Windows Presentation Foundation.[7] It is primarily used in displaying the analysis of the parser output in the analysis pane (see screen shots in Section 3.2). If the project were to be ported to Linux, these elements would not work. Data templating is present in Silverlight through the DataTemplateSelector class, but is not present in Windows Forms.

The primary use of data templates is to allow lists of different types of elements to display each type differently. In PC-PATR-EDITOR, I have used this functionality to allow sentences to have one interface, the parses inside of them to have another, and points of failure inside parses to have yet another. This could easily be duplicated by having a series of lists whose content depends on the selected item in the previous list. The user interface could be restructured as shown in Figure 59. The selected sentence's parse list would be displayed in the parse list box (top middle), the selected parse's failed nodes would be displayed in the failed nodes list box (top right), and the selected node's failed features would be displayed in the failed features box (middle right). The selected parse's parse tree would be displayed when the parse was selected and the selected nodes feature structure would be shown when the user clicked on it in the parse tree.

[7] http://msdn.microsoft.com/en-us/library/ms742521.aspx

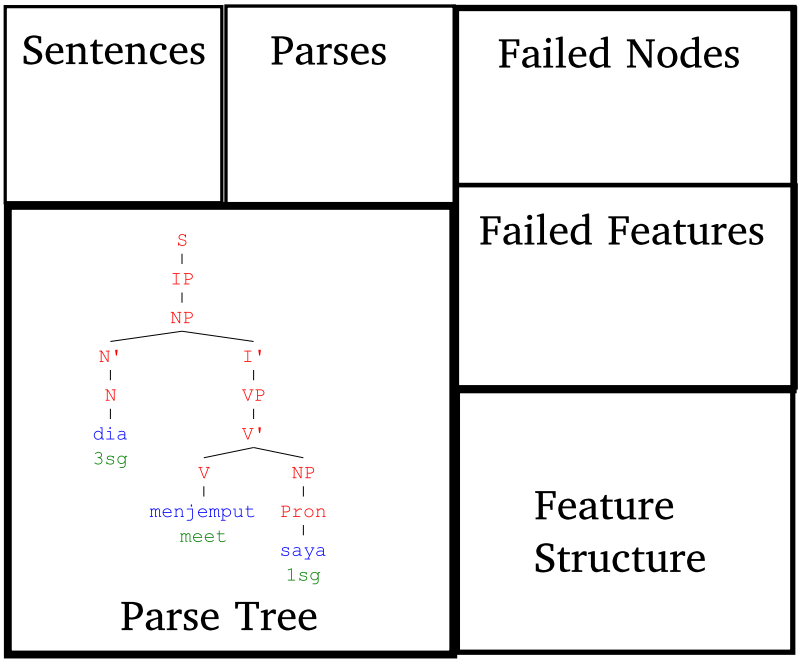

Figure 59. Possible restructuring of analysis page for Windows Forms

PC-PATR-EDITOR provides additional analysis tools described in Section 3.2.4. Currently, these tools are tightly integrated into the editor. Because they operate on the XML returned by the parser, they could be easily made more modular. The editor could provide a tool list where each tool could be given one of five hooks that integrate it into the application described in Table 2. It would be nice if they could be enabled and disabled as well. This would allow users to create custom tools and integrate them into the editor by providing the editor with a path to the tool to be run and the kind of hook to use. Any enabled tools with the Input Files hook would be automatically run when the parser is run; other hooks would cause tools to run when requested.

Table 2. Possible hooks for custom tools

Input Files	Run the tool before parsing with the paths to the file to parse and the lexicon.
Parse Log	Run the tool with the path to the entire parser output file.
Selected Sentence	Run the tool with the path to a file containing the XML representation of the selected sentence.
Selected Parse	Run the tool with a path to a file containing the XML representation of the selected parse.
Selected Node	Run the tool with a path to a file containing the XML representation of the selected node.

For example, the sentence comparison tool (Section 3.2.4.2) operates on the XML analysis returned for the selected sentence. This tool could be its own executable that could be run using the Selected Sentence hook. The lexicon and input checking tools (Section 3.2.4.1) could be run using the Input Files hook.

Finally, the analysis component of PC-PATR-EDITOR could be built as its own application. Users would navigate to a plain text or XML grammar file to use when parsing like they currently do with lexicon and data files. If the file selected were an XML file, then the file would be transformed to the correct format before parsing. This would allow the analysis component to be accessible to people who do not want to use a structured editor.

5.3 Summary

PC-PATR is a powerful tool for linguistic research and testing grammars. It suffers because its input files are plain text and its output formats are difficult to read. Syntax and content errors are easy to make when editing grammars. Content can be lost because the plain text format does not provide a simple mechanism for preventing rules and other elements from being used when parsing as they must be commented out or deleted. PC-PATR can return information about what rules were used to build nodes in a parse and where unification failures occurred. But this information is either encoded in XML or a verbose plain text format and so is hard to read.

I have made a first attempt to develop a structured editor for PC-PATR grammar files called PC-PATR-EDITOR. It stores grammars using XML and requires users to define a feature system and lists of terminal and non-terminal symbols for use in the grammar. Users are only allowed to use features and symbols defined in these lists when editing rules, constraints, and templates and thus prevents many of the input errors that plague the plain text format. The editor only lets users insert elements when they will result in a valid grammar file and thus eliminates the chance for syntax errors.

PC-PATR-EDITOR also provides an analysis component that processes the parser output and highlights errors. It displays parse trees in a form familiar to most linguists and displays a node's feature system when it is clicked on. This makes parser output easier to understand. It also provides a few additional tools to aid in analyzing and viewing the parser's output.

I tested the program by developing a grammar fragment for Bahasa Indonesia using the PAWS Starter Kit to create an initial draft. The grammar was revised using an X-Bar style analysis. This process highlighted some strengths and weaknesses of the program. I have made recommendations for future changes to the editor and analysis component in Sections 5.1 and 5.2 respectively.

PC-PATR-EDITOR is not perfect by any stretch of the imagination, but it does have some good parts. It allows users to edit grammars without having to remember the syntax of PC-PATR grammar files or hunt down syntax errors. This saves users time. Structured editing is a different experience than working with a text editor. The editing component needs some significant revision to make this feel natural and easy to use. The analysis component works well and could be broken out into its own application and modified to accept custom tools.

APPENDICES

APPENDIX A

PC-PATR ON SHEIBER1 TEST DATA

Below is a simple experiment I ran to illustrate the XML output from the command line version of PC-PATR. It is discussed in Section 1.2. I ran the command line implementation of PC-PATR on the Shieber1 test data available from SIL International (2005). The grammar file is shown below, followed by the data used in example (9), and finally the output returned is shown. This output was generated by the following commands.

1. load grammar shieber1.grm
2. load lexicon shieber1.lex
3. set failures on
4. set tree XML
5. file parse shieber.sen tfh.log

I have changed the comment character from the original ';' to '|' for ease of use with PC-PATR-EDITOR.

```
||| -*- Mode: PATR -*-

|||============================================================
|||              Demonstration Grammar One
||| Includes:  subject-verb agreement
|||============================================================

Parameter:  Start symbol is S.

Parameter:  Restrictor is <cat>
```

<head form>.

Parameter: Attribute order is cat lex sense head
   subject
   form agreement
   person number gender
   s np vp v.

|||==
||| Grammar Rules
|||==

Rule {sentence formation}

S -> NP VP:

<S head> = <VP head>
<VP head subject> = <NP head>.

Rule {trivial verb phrases}

VP -> V:

<VP head> = <V head>.

|||==
||| Lexicon
|||==

Lexicon root.

Word uther:

<cat> = NP
<head agreement gender> = masculine
<head agreement person> = third
<head agreement number> = singular.

Word knights:
<cat> = NP
<head agreement gender> = masculine
<head agreement person> = third
<head agreement number> = plural.

Word sleeps:
<cat> = V
<head form> = finite
<head subject agreement person> = third
<head subject agreement number> = singular.

Word sleep:
<cat> = V
<head form> = finite

<head subject agreement number> = plural.

Example (9) shows the shieber1.sen file from SIL International (2005).

(9) a. uther sleeps
b. uther sleep
c. knights sleeps
d. knights sleep

Below is the XML output from parsing shieber1.sen shown in example (9). The line numbers are not original to the parser's output and have been added for reference.

```
00000001 <Analysis count="1">
00000002 <Parse>
00000003 <Node cat="S" rule="sentence formation" id="s0_1._1">
00000004 <Fs>
00000005 <F name="cat"><Str>S</Str></F>
00000006 <F name="head">
00000007 <Fs id="s0_1.co2">
00000008 <F name="subject">
00000009 <Fs id="s0_1.co1">
00000010 <F name="agreement">
00000011 <Fs>
00000012 <F name="person"><Str>third</Str></F>
00000013 <F name="number"><Str>singular</Str></F>
00000014 <F name="gender"><Str>masculine</Str></F>
00000015 </Fs>
00000016 </F>
00000017 </Fs>
00000018 </F>
00000019 <F name="form"><Str>finite</Str></F>
```

```
00000020 </Fs>
00000021 </F>
00000022 </Fs>
00000023 <Leaf cat="NP" id="s0_1._2">
00000024 <Fs>
00000025 <F name="cat"><Str>NP</Str></F>
00000026 <F name="lex"><Str>uther</Str></F>
00000027 <F name="head" fVal="s0_1.co1"></F>
00000028 </Fs>
00000029 <Lexfs>
00000030 <F name="cat"><Str>NP</Str></F>
00000031 <F name="lex"><Str>uther</Str></F>
00000032 <F name="head">
00000033 <Fs>
00000034 <F name="agreement">
00000035 <Fs>
00000036 <F name="person"><Str>third</Str></F>
00000037 <F name="number"><Str>singular</Str></F>
00000038 <F name="gender"><Str>masculine</Str></F>
00000039 </Fs>
00000040 </F>
00000041 </Fs>
00000042 </F>
00000043 </Lexfs>
00000044 <Str>uther</Str>
00000045 </Leaf>
00000046 <Node cat="VP" rule="trivial verb phrases" id="s0_1._3">
00000047 <Fs>
00000048 <F name="cat"><Str>VP</Str></F>
00000049 <F name="head" fVal="s0_1.co2"></F>
00000050 </Fs>
00000051 <Leaf cat="V" id="s0_1._4">
00000052 <Fs>
00000053 <F name="cat"><Str>V</Str></F>
00000054 <F name="lex"><Str>sleeps</Str></F>
00000055 <F name="head" fVal="s0_1.co2"></F>
00000056 </Fs>
00000057 <Lexfs>
00000058 <F name="cat"><Str>V</Str></F>
00000059 <F name="lex"><Str>sleeps</Str></F>
00000060 <F name="head">
00000061 <Fs>
00000062 <F name="subject">
00000063 <Fs>
```

```
00000064 <F name="agreement">
00000065 <Fs>
00000066 <F name="person"><Str>third</Str></F>
00000067 <F name="number"><Str>singular</Str></F>
00000068 </Fs>
00000069 </F>
00000070 </Fs>
00000071 </F>
00000072 <F name="form"><Str>finite</Str></F>
00000073 </Fs>
00000074 </F>
00000075 </Lexfs>
00000076 <Str>sleeps</Str>
00000077 </Leaf>
00000078 </Node>
00000079 </Node>
00000080 </Parse>
00000081 </Analysis>
00000082
00000083 **** Not able to parse this sentence ****
00000084 **** Turning off unification ****
00000085 <Analysis count="1" fail="true">
00000086 <Parse>
00000087 <Node cat="S" rule="sentence formation" id="s0_1._1" fail="true">
00000088 <Fs>
00000089 <F name="cat"><Str>S</Str></F>
00000090 <F name="head">
00000091 <Fs>
00000092 <F name="subject">
00000093 <Fs>
00000094 <F name="agreement">
00000095 <Fs>
00000096 <F name="person"><Str>third</Str></F>
00000097 <F name="number"><Str>FAIL singular / plural</Str></F>
00000098 <F name="gender"><Str>masculine</Str></F>
00000099 </Fs>
00000100 </F>
00000101 </Fs>
00000102 </F>
00000103 </Fs>
00000104 </F>
00000105 </Fs>
00000106 <Leaf cat="NP" id="s0_1._2">
00000107 <Fs>
```

00000108 <F name="cat"><Str>NP</Str></F>
00000109 <F name="lex"><Str>uther</Str></F>
00000110 <F name="head">
00000111 <Fs>
00000112 <F name="agreement">
00000113 <Fs>
00000114 <F name="person"><Str>third</Str></F>
00000115 <F name="number"><Str>singular</Str></F>
00000116 <F name="gender"><Str>masculine</Str></F>
00000117 </Fs>
00000118 </F>
00000119 </Fs>
00000120 </F>
00000121 </Fs>
00000122 <Lexfs></Lexfs>
00000123 <Str>uther</Str>
00000124 </Leaf>
00000125 <Node cat="VP" rule="trivial verb phrases" id="s0_1._3">
00000126 <Fs>
00000127 <F name="cat"><Str>VP</Str></F>
00000128 <F name="head">
00000129 <Fs>
00000130 <F name="subject">
00000131 <Fs>
00000132 <F name="agreement">
00000133 <Fs>
00000134 <F name="number"><Str>plural</Str></F>
00000135 </Fs>
00000136 </F>
00000137 </Fs>
00000138 </F>
00000139 <F name="form"><Str>finite</Str></F>
00000140 </Fs>
00000141 </F>
00000142 </Fs>
00000143 <Leaf cat="V" id="s0_1._4">
00000144 <Fs>
00000145 <F name="cat"><Str>V</Str></F>
00000146 <F name="lex"><Str>sleep</Str></F>
00000147 <F name="head">
00000148 <Fs>
00000149 <F name="subject">
00000150 <Fs>
00000151 <F name="agreement">

```
00000152 <Fs>
00000153 <F name="number"><Str>plural</Str></F>
00000154 </Fs>
00000155 </F>
00000156 </Fs>
00000157 </F>
00000158 <F name="form"><Str>finite</Str></F>
00000159 </Fs>
00000160 </F>
00000161 </Fs>
00000162 <Lexfs></Lexfs>
00000163 <Str>sleep</Str>
00000164 </Leaf>
00000165 </Node>
00000166 </Node>
00000167 </Parse>
00000168 </Analysis>
00000169
00000170 **** Not able to parse this sentence ****
00000171 **** Turning off unification ****
00000172 <Analysis count="1" fail="true">
00000173 <Parse>
00000174 <Node cat="S" rule="sentence formation" id="s0_1._1" fail="true">
00000175 <Fs>
00000176 <F name="cat"><Str>S</Str></F>
00000177 <F name="head">
00000178 <Fs>
00000179 <F name="subject">
00000180 <Fs>
00000181 <F name="agreement">
00000182 <Fs>
00000183 <F name="person"><Str>third</Str></F>
00000184 <F name="number"><Str>FAIL plural / singular</Str></F>
00000185 <F name="gender"><Str>masculine</Str></F>
00000186 </Fs>
00000187 </F>
00000188 </Fs>
00000189 </F>
00000190 </Fs>
00000191 </F>
00000192 </Fs>
00000193 <Leaf cat="NP" id="s0_1._2">
00000194 <Fs>
00000195 <F name="cat"><Str>NP</Str></F>
```

```
00000196 <F name="lex"><Str>knights</Str></F>
00000197 <F name="head">
00000198 <Fs>
00000199 <F name="agreement">
00000200 <Fs>
00000201 <F name="person"><Str>third</Str></F>
00000202 <F name="number"><Str>plural</Str></F>
00000203 <F name="gender"><Str>masculine</Str></F>
00000204 </Fs>
00000205 </F>
00000206 </Fs>
00000207 </F>
00000208 </Fs>
00000209 <Lexfs></Lexfs>
00000210 <Str>knights</Str>
00000211 </Leaf>
00000212 <Node cat="VP" rule="trivial verb phrases" id="s0_1._3">
00000213 <Fs>
00000214 <F name="cat"><Str>VP</Str></F>
00000215 <F name="head">
00000216 <Fs>
00000217 <F name="subject">
00000218 <Fs>
00000219 <F name="agreement">
00000220 <Fs>
00000221 <F name="person"><Str>third</Str></F>
00000222 <F name="number"><Str>singular</Str></F>
00000223 </Fs>
00000224 </F>
00000225 </Fs>
00000226 </F>
00000227 <F name="form"><Str>finite</Str></F>
00000228 </Fs>
00000229 </F>
00000230 </Fs>
00000231 <Leaf cat="V" id="s0_1._4">
00000232 <Fs>
00000233 <F name="cat"><Str>V</Str></F>
00000234 <F name="lex"><Str>sleeps</Str></F>
00000235 <F name="head">
00000236 <Fs>
00000237 <F name="subject">
00000238 <Fs>
00000239 <F name="agreement">
```

```
00000240 <Fs>
00000241 <F name="person"><Str>third</Str></F>
00000242 <F name="number"><Str>singular</Str></F>
00000243 </Fs>
00000244 </F>
00000245 </Fs>
00000246 </F>
00000247 <F name="form"><Str>finite</Str></F>
00000248 </Fs>
00000249 </F>
00000250 </Fs>
00000251 <Lexfs></Lexfs>
00000252 <Str>sleeps</Str>
00000253 </Leaf>
00000254 </Node>
00000255 </Node>
00000256 </Parse>
00000257 </Analysis>
00000258
00000259 <Analysis count="1">
00000260 <Parse>
00000261 <Node cat="S" rule="sentence formation" id="s0_1._1">
00000262 <Fs>
00000263 <F name="cat"><Str>S</Str></F>
00000264 <F name="head">
00000265 <Fs id="s0_1.co2">
00000266 <F name="subject">
00000267 <Fs id="s0_1.co1">
00000268 <F name="agreement">
00000269 <Fs>
00000270 <F name="person"><Str>third</Str></F>
00000271 <F name="number"><Str>plural</Str></F>
00000272 <F name="gender"><Str>masculine</Str></F>
00000273 </Fs>
00000274 </F>
00000275 </Fs>
00000276 </F>
00000277 <F name="form"><Str>finite</Str></F>
00000278 </Fs>
00000279 </F>
00000280 </Fs>
00000281 <Leaf cat="NP" id="s0_1._2">
00000282 <Fs>
00000283 <F name="cat"><Str>NP</Str></F>
```

```
00000284 <F name="lex"><Str>knights</Str></F>
00000285 <F name="head" fVal="s0_1.co1"></F>
00000286 </Fs>
00000287 <Lexfs>
00000288 <F name="cat"><Str>NP</Str></F>
00000289 <F name="lex"><Str>knights</Str></F>
00000290 <F name="head">
00000291 <Fs>
00000292 <F name="agreement">
00000293 <Fs>
00000294 <F name="person"><Str>third</Str></F>
00000295 <F name="number"><Str>plural</Str></F>
00000296 <F name="gender"><Str>masculine</Str></F>
00000297 </Fs>
00000298 </F>
00000299 </Fs>
00000300 </F>
00000301 </Lexfs>
00000302 <Str>knights</Str>
00000303 </Leaf>
00000304 <Node cat="VP" rule="trivial verb phrases" id="s0_1._3">
00000305 <Fs>
00000306 <F name="cat"><Str>VP</Str></F>
00000307 <F name="head" fVal="s0_1.co2"></F>
00000308 </Fs>
00000309 <Leaf cat="V" id="s0_1._4">
00000310 <Fs>
00000311 <F name="cat"><Str>V</Str></F>
00000312 <F name="lex"><Str>sleep</Str></F>
00000313 <F name="head" fVal="s0_1.co2"></F>
00000314 </Fs>
00000315 <Lexfs>
00000316 <F name="cat"><Str>V</Str></F>
00000317 <F name="lex"><Str>sleep</Str></F>
00000318 <F name="head">
00000319 <Fs>
00000320 <F name="subject">
00000321 <Fs>
00000322 <F name="agreement">
00000323 <Fs>
00000324 <F name="number"><Str>plural</Str></F>
00000325 </Fs>
00000326 </F>
00000327 </Fs>
```

00000328 </F>
00000329 <F name="form"><Str>finite</Str></F>
00000330 </Fs>
00000331 </F>
00000332 </Lexfs>
00000333 <Str>sleep</Str>
00000334 </Leaf>
00000335 </Node>
00000336 </Node>
00000337 </Parse>
00000338 </Analysis>
00000339
00000340
00000341 File parsing statistics: 4 sentences read
00000342 2 sentences with 0 parses
00000343 2 sentences with 1 parse

00000344 2 of 4 (50.0 %) parsed at least once

APPENDIX B

PC-PATR XML DOCUMENT TYPE DEFINITION

Below is the document type definition developed by H. A. Black used in PC-PATR-EDITOR.

```
<!-- -->
 <!-- ****************** PCPATR.DTD ****************** -->
 <!-- -->
 <!-- A formal specification of the conceptual model of a PC-PATR -->
 <!-- grammar file. -->
 <!-- -->
 <!-- VERSION 0.2.0 -->
 <!-- -->
 <!-- This file is maintained by Andy Black. Send comments or -->
 <!-- suggested refinements by email to: andy_black@sil.org -->
 <!-- -->
 <!-- The revision history is maintained at the end of the file. -->
 <!-- -->

 <!ENTITY % constraintControl "
   enabled (yes|no) 'yes'
   useWhenDebugging (yes | no) 'yes'"
 >
 <!-- enabled indicates whether or not the rule or constraint is currently being used.
   This lets one keep older ideas around and makes it easy to restore them.
   useWhenDebugging indicates whether the item is to be included in a debug run or not.
   By design, for useWhenDebugging most rules are "off" while most constraints are "on"
since this is the normal case.
 -->

 <!-- -->
 <!-- ********************** PC-PATR GRAMMAR ****************** -->
 <!-- -->
```

```
<!ELEMENT patrGrammar (comment*, featureTemplates?, constraintTemplates?, rules,
parameters?, lexicalRules?, masterLists) >

<!-- The top-level element is named "patrGrammar" for "PC-PATR grammar". It consists
of a set of rules and optional
  feature templates, parameters, lexical rules, and constraint templates. There also are a set
of required master lists.
 -->
<!ELEMENT masterLists (nonTerminals, terminals, builtInPrimitives, featureSystem?,
collectionFeatures?) >
<!ELEMENT builtInPrimitives (cat, lex, gloss, rootgloss, none) >
<!-- The built-in primitives are items that PC-PATR always has.
  These should appear in every XML file that conforms to this DTD.
  We suggest using ids of pcpatrGrammarCat, pcpatrGrammarLex, pcpatrGrammarGloss,
pcpatrGrammarRootGloss,
  and pcpatrGrammarNone for these items.
 -->
<!ELEMENT cat EMPTY >
<!ATTLIST cat
  id ID #REQUIRED
>
<!ELEMENT lex EMPTY >
<!ATTLIST lex
  id ID #REQUIRED
>
<!ELEMENT gloss EMPTY >
<!ATTLIST gloss
  id ID #REQUIRED
>
<!ELEMENT rootgloss EMPTY >
<!ATTLIST rootgloss
  id ID #REQUIRED
>
<!ELEMENT none EMPTY >
<!ATTLIST none
  id ID #REQUIRED
>

<!-- -->
<!-- ********************** RULE ************************** -->
<!-- -->
<!ELEMENT rules (rule+) >
<!ELEMENT rule (comment*, identifier?, phraseStructureRule, constraints?) >
<!ATTLIST rule
```

```
  id ID #REQUIRED
  enabled (yes|no) "yes"
  useWhenDebugging (yes | no) "no"
>
<!-- A rule consists of zero or more comment lines,
  an optional rule identifier (used more as a comment than anything else)
  a phrase structure rule, zero or more constraints.

  id is a unique ID for this rule.
  enabled indicates whether it is being used or not.
  useWhenDebugging indicates whether the item is to be included in a debug run or not.
-->

<!ELEMENT phraseStructureRule (leftHandSide, rightHandSide, comment*) >
<!-- A phrase structure rule consists of a left hand side,
  a right hand side, and an optional comment. -->

<!ELEMENT comment (#PCDATA) >
<!-- A comment. -->

<!ELEMENT identifier (#PCDATA) >
<!-- A user-defined identifier for the rule. -->

<!ELEMENT leftHandSide EMPTY >
<!ATTLIST leftHandSide
  id ID #REQUIRED
  symbol IDREF #REQUIRED
>
<!-- A left hand side is always a nonTerminal. -->

<!ELEMENT rightHandSide (symbolRef | optionalSymbols | disjunction | comment)* >
 <!-- A right hand side is a list of terminal and non-terminal symbols, optional symbols
and/or disjunctive symbols.
  optional and disjunctive symbols may be nested to any depth.
-->

<!ELEMENT symbolRef EMPTY >
<!ATTLIST symbolRef
  id ID #REQUIRED
  symbol IDREF #REQUIRED
>
<!-- a symbolRef refers to a terminal or a nonTerminal symbol -->

<!ELEMENT optionalSymbols ((symbolRef | optionalSymbols | disjunction)+) >
```

```
<!ELEMENT disjunction ((symbolRef | optionalSymbols)+, disjunctiveSymbols) >
<!ELEMENT disjunctiveSymbols ((symbolRef | optionalSymbols)+, disjunctiveSymbols?)
>

<!-- -->
<!-- ************** CONSTRAINTS ***************** -->
<!-- -->
<!ELEMENT constraints (percolationOperation | unificationConstraint | priorityUnionOp-
eration | logicalConstraint | constraintDisjunction | comment)* >
<!-- There are four kinds of constraints in this mark-up (PC-PATR only has three, but we
are treating
  percolation operations specially): percolation operations, unification constraints, priority
union
  operations, and logical constraints.
  Any of these may occur in a disjunction of constraints.
-->
<!ELEMENT constraintDisjunction (constraints, constraints+) >
<!ELEMENT percolationOperation ((featurePath | subcatPath), (featurePath | subcatPath),
comment*) >
<!ATTLIST percolationOperation
  %constraintControl;
>
<!-- A percolation operation has two parts:
  the first is a path whose symbol is the left hand side symbol of the phrase structure rule;
  the second is a path whose symbol is one of the right hand side symbols of the phrase
structure rule -->

<!ELEMENT featurePath ((embeddedFeaturePath | catValuePath | glossValuePath | lex-
ValuePath | rootGlossValuePath)?)>
<!ATTLIST featurePath
  node IDREF #REQUIRED
  featureStart IDREF #REQUIRED
  featureEnd IDREF #IMPLIED
>
<!-- a feature path consists of
a reference to a node in a phrase structure rule (to a symbolRef);
a reference to a starting complex or closed feature or subcat or collection feature; and
a reference to an ending complex or closed feature or a feature value or collection feature
or a part of a subcat list (i.e. subcat, first, or rest)
(The intermediate nodes are implicit in the feature system.)
-->
<!ELEMENT embeddedFeaturePath (embeddedFeaturePath?)>
<!ATTLIST embeddedFeaturePath
  featureStart IDREF #REQUIRED
```

```
  featureEnd IDREF #IMPLIED
>
<!-- an embedded feature path consists of
 a reference to a starting complex or closed feature or subcat or collection feature; and
 a reference to an ending complex or closed feature or a feature value or collection feature
or a part of a subcat list (i.e. subcat, first, or rest)
 (The intermediate nodes are implicit in the feature system.)
-->
<!ELEMENT subcatPath EMPTY>
<!ATTLIST subcatPath
  node IDREF #REQUIRED
  featureStart IDREF #IMPLIED
  featureEnd IDREF #IMPLIED
  subcatEnd (noneValue | first |rest) #IMPLIED
  featureAfterStart IDREF #IMPLIED
  featureAfterEnd IDREF #IMPLIED
>
<!-- a subcat path consists of
 a reference to a node in a phrase structure rule (to a symbolRef);
 any preceding feature path items bounded by
  a reference to a starting complex or closed feature; and
  a reference to an ending complex or closed feature or a feature value
  (The intermediate nodes are implicit in the feature system.)
 subcat (output automatically)
 any further refinement of the subcat path (e.g. first or rest)
 any following feature path (featureAfterStart and featureAfterEnd)
-->

<!ELEMENT unificationConstraint ((featurePath | subcatPath), (featurePath | featureVal-
ueRef | categoryValueRef | featureValueDisjunction | subcatPath | lexValuePath | glossVal-
uePath | rootGlossValuePath| noneValue | templateFeatureStructure), comment*) >
<!ATTLIST unificationConstraint
  %constraintControl;
>
<!-- A unification constraint has two parts: the first is a feature path;
  the second is either another feature path or a feature value -->

<!ELEMENT categoryValueRef EMPTY>
<!ATTLIST categoryValueRef
  node IDREF #REQUIRED
>
<!-- a categoryValueRef refers to node in the phrase structure rule -->

<!ELEMENT featureValueDisjunction (featureValueRef, featureValueRef+) >
```

```
<!ELEMENT priorityUnionOperation ((featurePath | subcatPath),(featurePath | featureValueRef | categoryValueRef | featureValueDisjunction | noneValue | templateFeatureStructure), comment*) >
<!ATTLIST priorityUnionOperation
  %constraintControl;
>
<!-- A priority union operation has two parts:
  the first is a feature path whose symbol is the left hand side symbol of the phrase structure rule;
  the second is a feature path whose symbol is one of the right hand side symbols of the phrase structure rule -->

<!ELEMENT logicalConstraint (featurePath, (logicalExpression | logicalConstraintRef), comment*) >
<!ATTLIST logicalConstraint
  %constraintControl;
>
<!-- A logical constraint operation has two parts:
  the first is a feature path whose symbol is one of the symbols of the phrase structure rule;
  the second is a logical expression -->
<!ELEMENT logicalConstraintRef EMPTY >
<!ATTLIST logicalConstraintRef
  constraint IDREF #REQUIRED
>
<!-- a logicalConstraintRef refers to a constraintTemplate -->
<!ELEMENT logicalExpression (unaryOperation | binaryOperation) >
<!ELEMENT unaryOperation (factor) >
<!ATTLIST unaryOperation
  operation (existence | negation) "existence"
>
<!ELEMENT binaryOperation (factor, comment?, factor) >
<!ATTLIST binaryOperation
  operation (and | or | conditional | bicondtional) "and"
>
<!ELEMENT factor (feature | complexFeatureRef | collectionFeatureRef | logicalExpression | catValue | lexValue | glossValue | rootGlossValue) >

<!ELEMENT feature (feature?) >
<!ATTLIST feature
  featureStart IDREF #REQUIRED
  featureEnd IDREF #REQUIRED
  indexedVariable (none | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) "none"
>
```

```
<!-- a feature consists of
a reference to a starting complex or closed feature; and
a reference to an ending complex or closed feature or a feature value
(The intermediate nodes are implicit in the feature system.)
an optional indexedVariable which acts as a variable for the value
  of the feature indicated by the closed feature referred to by featureEnd
-->

<!-- -->
<!-- ****************** FEATURE SYSTEM ********************** -->
<!-- -->

<!ELEMENT featureSystem ((complexFeatureDefinition | closedFeatureDefinition)+) >

<!ELEMENT complexFeatureDefinition (comment*, name, description?, (complexFeatureDefinition | closedFeatureDefinition | complexFeatureDefinitionRef )+) >
<!ATTLIST complexFeatureDefinition
  id ID #REQUIRED
>
<!-- a complexFeatureDefinition consists of one or more other features, either closed or complex.
  The description is not used by PC-PATR but can help a user remember what the feature is for.
-->
<!ELEMENT complexFeatureDefinitionRef EMPTY >
<!ATTLIST complexFeatureDefinitionRef
  id ID #REQUIRED
  complex IDREF #REQUIRED
>
<!-- a complexFeatureDefinitionRef refers to another complexFeatureDefinition defined elsewhere in the feature system.
  It provides a way to reuse parts of the feature system.
-->
<!ELEMENT closedFeatureDefinition (comment*, name, description?, featureValueDefinition+) >
<!ATTLIST closedFeatureDefinition
  id ID #REQUIRED
>
<!-- a closedFeatureDefinition is a simple feature with one or more values.
  The description is not used by PC-PATR but can help a user remember what the feature is for.
-->
<!ELEMENT featureValueDefinition (comment*, name, description?) >
<!ATTLIST featureValueDefinition
```

```
  id ID #REQUIRED
>
<!-- a featureValueDefinition defines a value of a (closed) feature.
  The description is not used by PC-PATR but can help a user remember what the feature is for.
-->

<!ELEMENT name (#PCDATA) >
<!-- The name of the feature or other item. -->

<!-- -->
<!-- ****************** FEATURE TEMPLATE ********************** -->
<!-- -->
<!ELEMENT featureTemplates ((featureTemplate | comment)*) >
<!ELEMENT featureTemplate (comment*, featureTemplateDefinition) >
<!ATTLIST featureTemplate
  %constraintControl;
>
<!-- A feature template has optional comment lines and
  the feature definition.
 -->
<!ELEMENT featureTemplateDefinition ((featureTemplateName | terminalRef),
  (templateFeatureStructure | templateFeatureDisjunction | templateFeaturePath | templateFeaturePathDisjunction | featureTemplateRef |
  catValuePath | lexValuePath | glossValuePath | rootGlossValuePath | empty)+) >
<!-- A feature definition consists of two parts.
  The first is a name or a reference to a terminal symbol.
  The second is the content of the definition.
-->

<!ELEMENT featureTemplateName (#PCDATA) >
<!ATTLIST featureTemplateName
  id ID #IMPLIED
>
<!-- a featureTemplateName is the name of the feature template.
  It may have an id for when another feature template refers to it.
-->
<!ELEMENT templateFeatureStructure ((complexFeatureRef | featureRef | feature | templateFeatureDisjunction | subcatRef | catValue | lexValue | glossValue | rootGlossValue
  | collectionFeatureRef | featureTemplateRef | comment)+)>
<!-- a templateFeatureStructure defines a feature structure -->
<!ELEMENT featureRef ((featureRef | collectionFeatureRef | templateFeatureDisjunction | featureTemplateRef)?) >
<!ATTLIST featureRef
```

```
  featureStart IDREF #REQUIRED
  featureEnd IDREF #REQUIRED
  isDefault (no | yes) "no"
>
<!-- a featureRef refers to a feature path within in the feature system.
  It can be embedded in which case the featureEnd of the parent is a complexFeatureDefinitionRef and
  the startFeature of the embedded featureRef refers to the same feature as the complexFeatureDefinitionRef reference
  of its parent's featureEnd.
-->
<!ELEMENT collectionFeatureRef (noneRef | catValue | lexValue | glossValue | rootGlossValue | templateFeatureStructure | featureRef | featureTemplateRef) >
<!ATTLIST collectionFeatureRef
  collectionFeature IDREF #REQUIRED
  isDefault (no | yes) "no"
>
<!-- a collectionFeatureRef refers to a collection feature and has a path to other items -->
<!ELEMENT catValue EMPTY >
<!ATTLIST catValue
  symbol IDREF #REQUIRED
  isDefault (no | yes) "no"
>
<!-- a catValue refers to terminal or non-terminal symbol -->
<!ELEMENT lexValue (#PCDATA) >
<!ATTLIST lexValue
  isDefault (no | yes) "no"
>
<!ELEMENT glossValue (#PCDATA) >
<!ATTLIST glossValue
  isDefault (no | yes) "no"
>
<!ELEMENT rootGlossValue (#PCDATA) >
<!ATTLIST rootGlossValue
  isDefault (no | yes) "no"
>
<!ELEMENT subcatRef (noneRef | (firstRef, restRef) | subcatDisjunction) >
<!ATTLIST subcatRef
  isDefault (no | yes) "no"
>
<!-- a subcatRef is used for the special case of subcategorization lists.
  They have either a (first, rest) pair or a value of none (to end the list).
  It can also have a disjunctive value.
-->
```

```
<!ELEMENT noneRef EMPTY >
<!ELEMENT firstRef (complexFeatureRef | featureRef | templateFeatureDisjunction | noneRef
| featureTemplateRef | templateFeatureStructure | subcatDisjunction) >
<!ELEMENT restRef (complexFeatureRef | featureRef | templateFeatureDisjunction | noneRef
| featureTemplateRef | templateFeatureStructure | subcatDisjunction) >
<!-- firstRef and restRef elements contain the contents of first element of a subcategoriza-
tion list and the rest of the list, respecitively. -->
<!ELEMENT subcatDisjunction ((noneRef | (firstRef, restRef)), (noneRef | (firstRef, re-
stRef))+) >
<!ELEMENT complexFeatureRef ((complexFeatureRef | featureRef | feature | collection-
FeatureRef | templateFeatureDisjunction | subcatRef | catValue | comment)+) >
<!ATTLIST complexFeatureRef
  feature IDREF #REQUIRED
>
<!-- a complexFeatureRef is used to build a complex feature in a feature template.
  The feature attribute refers to a complexFeatureDefinition.
-->
<!ELEMENT templateFeatureDisjunction ((templateFeatureStructure | featureTemplateRef
| collectionFeatureRef | noneValue | featureValueRef | lexValue),
  (templateFeatureStructure | featureTemplateRef | collectionFeatureRef | noneValue | fea-
tureValueRef | lexValue | comment)+) >
<!ELEMENT templateFeaturePath ((templateFeaturePath | catValuePath | noneValuePath
| glossValuePath | lexValuePath | rootGlossValuePath)?)>
<!ATTLIST templateFeaturePath
  featureStart IDREF #REQUIRED
  featureEnd IDREF #REQUIRED
  isDefault (no | yes) "no"
>
<!-- a feature path consists of
a reference to a non-terminal or terminal symbol;
a reference to a starting complex or closed feature; and
a reference to an ending complex or closed feature or a feature value
(The intermediate nodes are implicit in the feature system.)
-->
<!ELEMENT templateFeaturePathDisjunction (templateFeaturePath?, featureValueRef, fea-
tureValueRef+) >
<!ATTLIST templateFeaturePathDisjunction
  featureStart IDREF #REQUIRED
  featureEnd IDREF #REQUIRED
>
<!ELEMENT catValuePath EMPTY >
<!ATTLIST catValuePath
  symbol IDREF #REQUIRED
  isDefault (no | yes) "no"
```

```
>
<!-- a catValue refers to terminal or non-terminal symbol -->
<!ELEMENT noneValuePath (#PCDATA) >
<!ATTLIST noneValuePath
  isDefault (no | yes) "no"
>
<!ELEMENT lexValuePath (#PCDATA) >
<!ATTLIST lexValuePath
  isDefault (no | yes) "no"
>
<!ELEMENT glossValuePath (#PCDATA) >
<!ATTLIST glossValuePath
  isDefault (no | yes) "no"
>
<!ELEMENT rootGlossValuePath (#PCDATA) >
<!ATTLIST rootGlossValuePath
  isDefault (no | yes) "no"
>
<!ELEMENT featureValueRef EMPTY >
<!ATTLIST featureValueRef
  value IDREF #REQUIRED
  isDefault (no | yes) "no"
>
<!ELEMENT featureTemplateRef EMPTY >
<!ATTLIST featureTemplateRef
  template IDREF #REQUIRED
>
<!-- a featureTemplateRef refers to a feature template defined earlier in the file. -->
<!-- -->
<!-- ********************** PARAMETERS ********************** -->
<!-- -->

 <!ELEMENT parameters (comment*, startSymbol?, restrictor?, attributeOrder?, catego-
ryFeature?, lexicalFeature?, glossFeature?, rootGlossFeature?, comment*) >
 <!-- A parameter setting consists of zero or more comment lines,
   the parameter name, the parameter value, and an optional
   comment. -->

 <!ELEMENT startSymbol (comment*)>
 <!ATTLIST startSymbol
   symbol IDREF #REQUIRED
 >
 <!ELEMENT restrictor (templateFeaturePath+, comment*)>
```

```
<!ELEMENT attributeOrder ((catRef | lexRef | glossRef | rootGlossRef | featureNameRef
| subcat | first | rest)+, comment*) >
<!ELEMENT catRef EMPTY >
<!ATTLIST catRef
  cat IDREF #REQUIRED
>
<!ELEMENT lexRef EMPTY >
<!ATTLIST lexRef
  lex IDREF #REQUIRED
>
<!ELEMENT glossRef EMPTY >
<!ATTLIST glossRef
  gloss IDREF #REQUIRED
>
<!ELEMENT rootGlossRef EMPTY >
<!ATTLIST rootGlossRef
  rootGloss IDREF #REQUIRED
>
<!ELEMENT featureNameRef EMPTY >
<!ATTLIST featureNameRef
  feature IDREF #REQUIRED
>
<!ELEMENT subcat EMPTY >
<!ELEMENT first EMPTY >
<!ELEMENT rest EMPTY >
<!ELEMENT noneValue EMPTY >
<!ELEMENT empty EMPTY >
<!ELEMENT categoryFeature (name, comment*) >
<!ELEMENT lexicalFeature (name, comment*) >
<!ELEMENT glossFeature (name, comment*) >
<!ELEMENT rootGlossFeature (name, comment*) >
<!-- -->
<!-- ********************** LEXICAL RULE ********************** -->
<!-- -->
<!ELEMENT lexicalRules (lexicalRule*) >
<!ELEMENT lexicalRule (comment*, name, lexicalRuleDefinition+) >
<!-- A lexical rule has an name and a definition. -->
<!ELEMENT lexicalRuleDefinition (lexicalRuleFeaturePath, (lexicalRuleFeaturePath | fea-
tureValueRef), comment?) >
<!ELEMENT lexicalRuleFeaturePath EMPTY >
<!ATTLIST lexicalRuleFeaturePath
  featureStart IDREF #REQUIRED
  featureEnd IDREF #REQUIRED
>
```

```
<!-- -->
<!-- ****************** CONSTRAINT TEMPLATE ****************** -->
<!-- -->
<!ELEMENT constraintTemplates (constraintTemplate*) >
<!ELEMENT constraintTemplate (name, logicalExpression, comment*) >
<!ATTLIST constraintTemplate
  id ID #REQUIRED
>
<!-- A constraint template has a name and a logical expression. -->

<!-- -->
<!-- ******************* NON-TERMINAL ****************** -->
<!-- -->
<!ELEMENT nonTerminals (nonTerminal+) >
<!ELEMENT nonTerminal (comment*, name, description?) >
<!ATTLIST nonTerminal
  id ID #REQUIRED
>
<!-- A non-terminal has a name and definition and a required ID. -->

<!-- -->
<!-- ******************* TERMINAL ****************** -->
<!-- -->
<!ELEMENT terminals (terminal+) >
<!ELEMENT terminal (comment*, name, description?) >
<!ATTLIST terminal
  id ID #REQUIRED
>
<!-- A terminal symbol has a name and definition and a required ID. -->
<!ELEMENT description (#PCDATA) >
<!ELEMENT terminalRef EMPTY >
<!ATTLIST terminalRef
  terminal IDREF #REQUIRED
>
<!-- -->
<!-- ******************* COLLECTION FEATURES ****************** -->
<!-- -->
<!ELEMENT collectionFeatures (collectionFeature*) >
<!ELEMENT collectionFeature (comment*, name, description?) >
<!ATTLIST collectionFeature
  id ID #REQUIRED
>
<!-- a collectFeature is a feature that can contain various things:
  a complex feature, a category, none.
```

Therefore it cannot fit into the feature system and needs to be handled specially.
-->
<!-- ********** REVISION HISTORY ********** -->
<!-- -->
<!-- Version 0.2.0, January 10, 2012 -->
<!-- Flesh out constituent objects -->
<!-- Version 0.1.0, February 11, 2000 -->
<!-- First draft by Andy Black -->
<!-- This is for Stage 1: a simplified model of rules and
their constraints. Stage 2 will deal with detailing

rule contents in terms of their constituent objects. -->

APPENDIX C

EXAMPLE DATA FROM CASE STUDIES

Example (10) shows the interlinear text of the sentences used for the case studies in Chapter 4.

(10) a. saya di-jemput=nya
1SG PASS-met=3SG
I was by him. Sneddon (1996:248)

b. saya di-jemput olehnya
1SG PASS-met by=3SG
I was met by him. Sneddon (1996:248)

c. saya di-jemput dia
1SG PASS-met 3SG
I was met by him. Sneddon (1996:248)

d. dia kami jemput
3SG 1PL.EXCL met
He was met by us. Sneddon (1996:249)

e. buku ini tidak akan kami baca
book this NEG FUT 1PL.EXCL read
This book will not be read by us. Sneddon (1996:249)

f. buku ini sudah ku-baca
book this already 1SG-read
I have already read this book. Sneddon (1996:249)

g. buku ini harus kau-baca
book this must 2SG-read
You must read this book. Sneddon (1996:249)

APPENDIX D

TEXT FORMAT OF GRAMMAR USED FOR CASE STUDIES

Below is the final form of the grammar rules and templates used in the case studies in Chapter 4. This grammar is a fragment of the grammar generated by the PAWS Starter Kit with my own additions and changes. It is not intended to be a full grammar for Bahasa Indonesia and as such is missing rules and elements required for other constructions such as questions and imperatives.[1]

```
Let actorVoice be <head infl voice> = actor
Let dativeVoice be <head infl voice> = dative
Let goalVoice be <head infl voice> = goal
Let instrumentalVoice be <head infl voice> = instrumental
Let locativeVoice be <head infl voice> = locative
Let objectVoice be <head infl voice> = object
Let AdjP-final be <head type AdjP-final> = +
        <head type AdjP-initial> = -
Let AdjP-initial be <head type AdjP-initial> = +
        <head type AdjP-final> = -
Let AdvP-final be <head type AdvP-final> = +
        <head type AdvP-initial> = -
Let AdvP-initial be <head type AdvP-initial> = +
        <head type AdvP-final> = -
Let -alone be <head type stand-alone> = -
Let animate be <head agr animacy> = animate
Let -animate be <head agr animacy> = inanimate
Let causative be <head infl valence> = causative
Let class_animal be <head agr class> = living_things
Let comma be <head type comma> = +
Let comp be <head type comp> = +
Let comparative be <head type comparative> = +
```

[1] The XML version of the grammar contains rules and templates for handling other constructions. It is still a work in progress, but is available by e-mailing me at fhardison@gmail.com.

```
Let compareAdj be <head type compareAdj> = +
Let compareN be <head type compareN> = +
Let completive be <head infl aspect> = completive
Let compound be <head type compound> = +
Let -compound be <head type compound> = -
Let conjoins_DP be <head type conjoins_DP> = +
Let conjoins_IP be <head type conjoins_IP> = +
Let -conjoins_DP be <head type conjoins_DP> = -
Let -conjoins_IP be <head type conjoins_IP> = -
Let contemplative be <head infl aspect> = contemplative
Let continuative be <head infl aspect> = continuative
Let contrafactual be <head infl aspect> = contrafactual
Let copular be <head type copular> = +
Let CP-final be <head type CP-final> = +
        <head type CP-initial> = -
Let -CP-final be <head type CP-final> = -
Let CP-initial be <head type CP-initial> = +
        <head type CP-final> = -
Let -CP-initial be <head type CP-initial> = -
Let CP-specifier-initial be <head type CP-specifier-initial> = +
Let declarative be <head infl mood> = declarative
Let definite be <head type definite> = +
Let ditransitive be <head type ditransitive> = +
        <head type transitive> = +
Let DP-final be <head type DP-final> = +
        <head type DP-initial> = -
Let -DP-final be <head type DP-final> = -
Let DP-initial be <head type DP-initial> = +
        <head type DP-final> = -
Let -DP-initial be <head type DP-initial> = -
Let embedded_CP be <embedded cat> = CP
Let embedded_causative_IP be <head infl valence> = causative
        [embedded : [cat : IP
            head : [infl : [finite : -]]
        subject : [head : [type : [pro-drop : +]]]]]
Let embedded_causative_VP be <head infl valence> = causative
        [embedded : [cat : VP
            head : [infl : [finite : -]]]]
Let embedded_IP be <embedded cat> = IP
Let embedded_IPpro-dropOrCP be {[embedded : [cat : IP
            head : [subject : [head : [type : [pro-drop : +]]]]]]
        [embedded : [cat : CP
            head : [subject : [head : [type : [pro-drop : -]]]]]]}
Let embedded_finite be <embedded head infl finite> = +
```

```
        <embedded head type question> = !-
Let embedded_-finite be <embedded head infl finite> = -
        <embedded head type question> = !-
Let embedded_perfective be <embedded head infl aspect> = perfective
Let embedded_pro-drop be <embedded head subject head type pro-drop> = +
Let embedded_question be <embedded head type question> = +
Let embedded_-question be <embedded head type question> = -
Let embedded_question_allowed be {[embedded : [head : [type : [question : +]]]]
        [embedded : [head : [type : [question : -]]]]}
Let embedded_raising be {[embedded : [cat : IP
            head : [subject : [head : [type : [pro-drop : +]]]]]]
        [embedded : [cat : IP
            head : [subject : [head : [type : [pro-drop : -]]]
            infl : [finite : +]]]]}
Let embedded_subjunctive be <embedded head infl mood> = realis
Let emphatic be <head type emphatic> = +
Let equalAdj be <head type equalAdj> = +
Let equalN be <head type equalN> = +
Let exclusive be <head agr person> = first_exclusive
Let excl-initial be <head type excl-initial> = +
Let excl-final be <head type excl-final> = +
Let exist be <head type existential> = +
Let experiencer be <head subject head case> = dative
Let finite be <head infl finite> = +
Let -finite be <head infl finite> = -
Let first be <head agr person> = first
Let focus be <head type focus> = +
Let focus-initial be <head type focus-initial> = +
Let focus-final be <head type focus-final> = +
Let future be <head infl tense> = future
Let gerund be <head type gerund> = +
Let habitual be <head infl aspect> = habitual
Let human be <head agr animacy> = human
Let -human be <head agr animacy> = nonhuman
Let imperative be <head infl mood> = imperative
Let imperfective be <head infl aspect> = imperfective
Let inclusive be <head agr person> = first_inclusive
Let incompletive be <head infl aspect> = incompletive
Let indefinite be <head type definite> = -
        <head type relative> = -
Let infinitive be <head infl finite> = -
Let interrogative be <head infl mood> = interrogative
        <head type question> = +
Let -interrogative be <head type question> = -
```

<head infl mood> = interrogative
Let intransitive be <head type transitive> = -
Let irrealis be <head infl mood> = irrealis
Let locative be <head type locative> = +
Let manner be <head type manner> = +
Let mass be <head type mass> = +
Let modifies_Adj be <head type modifies_Adj> = +
Let modifies_Adv be <head type modifies_Adv temporal> = +
<head type modifies_Adv locative> = +
<head type modifies_Adv manner> = +
<head type modifies_Adv reason> = +
Let modifies_Adv-reason be <head type modifies_Adv reason> = -
<head type modifies_Adv locative> = +
<head type modifies_Adv manner> = +
<head type modifies_Adv temporal> = +
Let modifies_locative be <head type modifies_Adv locative> = +
Let modifies_manner be <head type modifies_Adv manner> = +
Let modifies_NP be <head type modifies_NP> = +
Let modifies_PP be <head type modifies_PP> = +
Let modifies_Q be <head type modifies_Q> = +
Let modifies_reason be <head type modifies_Adv reason> = +
Let modifies_temporal be <head type modifies_Adv temporal> = +
Let motion be <head type motion> = +
Let negative be <head infl polarity> = negative
Let negative_prefix be <head type prefix negative> = +
[negative]
Let negative_suffix be <head type suffix negative> = +
[negative]
Let negative-polarity be <head type negative-polarity> = +
[negative]
Let negative-polarity_prefix be <head type prefix negative-polarity> = +
[negative_prefix]
Let negative-polarity_suffix be <head type suffix negative-polarity> = +
[negative_suffix]
Let oblique be <head type oblique> = +
Let participle be <head type participle> = +
Let partitive be <head type partitive> = +
Let passive be <head infl valence> = passive
Let passive.optional be <head infl valence> = { passive active}
Let past be <head infl tense> = past
Let perception be <head type perception> = +
Let perfect be <head infl aspect> = perfect
Let perfective be <head infl aspect> = perfective
Let plural be <head agr number> = plural

```
Let possessed be <head type possessed> = +
Let possessive be <head type possessive> = +
Let potential be <head infl aspect> = potential
Let present be <head infl tense> = present
Let progressive be <head infl aspect> = progressive
Let proper be <head type proper> = +
Let PP be <head type PP> = +
Let PP-final be <head type PP-final> = +
        <head type PP-initial> = -
Let PP-initial be <head type PP-initial> = +
        <head type PP-final> = -
Let P_prefix be <head type prefix P> = +
Let P_suffix be <head type suffix P> = +
Let question be <head type question> = +
Let -question be <head type question> = -
Let QP-final be <head type QP-final> = +
        <head type QP-initial> = -
Let QP-initial be <head type QP-initial> = +
        <head type QP-final> = -
Let quantifier be <head type quantifier> = +
Let quantifier_prefix be <head type prefix quantifier> = +
        [quantifier]
Let quantifier_suffix be <head type suffix quantifier> = +
        [quantifier]
Let reason be <head type reason> = +
Let realis be <head infl mood> = realis
Let reciprocal be <head type reciprocal> = +
Let reflexive be <head type reflexive> = +
Let relative be <head type relative> = +
Let relative_prefix be <head type prefix relative> = +
Let relative_suffix be <head type suffix relative> = +
Let root be <head type root> = +
Let +root be <head type root> = +
Let -root be <head type root> = -
Let second be <head agr person> = second
Let second_absolutive be {[head : [object : [head : [agr : [person : second]]]
            type : [transitive : +]]]
        [head : [subject : [head : [agr : [person : second]]]
            type : [transitive : -]]]}
Let second_ergative be <head subject head agr person> = second
        <head type transitive> = +
Let second_object be <head object head agr person> = second
Let second_subject be <head subject head agr person> = second
Let sentential be <head type sentential> = +
```

```
Let sentential_with_object be <head type sentential_with_object> = +
Let singular be <head agr number> = singular
Let singular_absolutive be {[head : [object : [head : [agr : [number : singular]]]
            type : [transitive : +]]]
        [head : [subject : [head : [agr : [number : singular]]]
            type : [transitive : -]]]}
Let singular_ergative be <head subject head agr number> = singular
        <head type transitive> = +
Let singular_object be <head object head agr number> = singular
Let singular_subject be <head subject head agr number> = singular
Let stand-alone be <head type stand-alone> = +
Let stative be <head infl aspect> = stative
Let subjunctive be <head infl mood> = realis
Let superlative_prefix be <head type prefix superlative> = +
Let superlative_suffix be <head type suffix superlative> = +
Let takes_Adv be <head type takes_Adv> = +
Let takes_DP be <head type takes_DP> = +
Let temporal be <head type temporal> = +
Let third be <head agr person> = third
Let third_absolutive be {[head : [object : [head : [agr : [person : third]]]
            type : [transitive : +]]]
        [head : [subject : [head : [agr : [person : third]]]
            type : [transitive : -]]]}
Let third_ergative be <head subject head agr person> = third
        <head type transitive> = +
Let third_object be <head object head agr person> = third
Let third_subject be <head subject head agr person> = third
Let topic be <head type topic> = +
Let topic-initial be <head type topic-initial> = +
Let topic-final be <head type topic-final> = +
Let transitive be <head type transitive> = +
Let unreal be <head infl aspect> = unreal
Let vocative be <head case> = vocative
Let wh be <head type wh> = +
Let -wh be <head type wh> = -
Let whQ be <head type whQ> = +
Let whQ_prefix be <head type prefix whQ> = +
Let whQ_suffix be <head type suffix whQ> = +
Let YNQ be <head type YNQ> = +
Let YNQ_prefix be <head type prefix YNQ> = +
Let YNQ_suffix be <head type suffix YNQ> = +
Let Adj be <cat> = Adj
        <head type ordinal> = !-
        <head type wh> = !-
```

```
        <head type sentential> = !-
        <head type clausal-comp> = !-
        <head type focusmarked> = -
        <head type gerund> = -
        <head type participle> = -
        <head type comma> = !-
        <head type topic> = !-
        <head type prefix conj> = !-
        <head type suffix conj> = !-
        [subcat : [first : [cat : NP]
                   rest : [subcat : [first : none
                   rest : none]]]]
Let Adv be <cat> = !Adv
        <head type wh> = !-
        <head type manner> = !-
        <head type existential> = !-
        <head type takes_Adv> = !-
        <head type takes_DP> = !-
        <head type sentential> = !-
        <embedded cat> = !none
        <head type gerund> = -
        <head type participle> = -
        <head type comma> = !-
        <head type prefix conj> = !-
        <head type suffix conj> = !-
Let Aux be <cat> = !Aux
        <head infl valence> = !active
        <head infl polarity> = !positive
        <head type gerund> = -
        <head type participle> = -
        <head type existential> = -
        <head infl finite> = !+
        <head type comma> = !-
        <head type topic> = !-
        <head type prefix conj> = !-
        <head type suffix conj> = !-
Let C be <cat> = !C
        <head type question> = !-
        <head type relative> = !-
        <embedded cat> = !none
        <head type comma> = !-
        <head type prefix conj> = !-
        <head type suffix conj> = !-
Let Case be <cat> = !Case
```

<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Conj be <cat> = !Conj
<head infl polarity> = !positive
<head agr number> = !singular
<head type comma> = !-
<head type comparative> = !-
<head type compareAdj> = !-
<head type compareN> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Deg be <cat> = !Deg
<head type modifies_Adj> = !-
<head type modifies_Adv temporal> = !-
<head type modifies_Adv locative> = !-
<head type modifies_Adv manner> = !-
<head type modifies_Adv reason> = !-
<head type modifies_NP> = !-
<head type modifies_PP> = !-
<head type modifies_Q> = !-
<head type quantifier> = !-
<head type wh> = !-
<head type relative> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type comma> = !-
<head type topic> = !-
<head type comparative> = !-
<head type equalAdj> = !-
<head type equalN> = !-
<head type superlative> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Dem be <cat> = Dem
<head type wh> = -
<head agr person> = third
<head type relative> = -
<head type temporal> = -
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type emphatic> = -

<head type pronoun> = -
<head type comma> = -
<head type topic> = -
<head type prefix conj> = -
<head type suffix conj> = -
Let Det be <cat> = !Det
<head type wh> = !-
<head agr person> = !third
<head type relative> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Excl be <cat> = Excl
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Greet be <cat> = Greet
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Intj be <cat> = Intj
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let N be <cat> = N
<head agr person> = !third
<head agr animacy> = !inanimate
<head possessor head agr person> = !none
<head type prefix case> = !none
<head type suffix case> = !none
<gap first> = none
<head type wh> = -
<head type reciprocal> = !-
<head type reflexive> = !-
<head type sentential> = !-
<head type proper> = !-
<head type relative> = !-
<head type mass> = !-
<head type temporal> = !-
<head embedded cat> = none

<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
<head case> = !none

Let Num be <cat> = Num
<head type wh> = -
<head agr number> = plural
<head type relative> = -
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type quantifier> = +
<head type comma> = -
<head type topic> = -
<head type prefix conj> = -
<head type suffix conj> = -
<head type numeric> = +
<head type ordinal> = !-

Let ordinal be <head type ordinal> = +
<head agr number> = !plural
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-

Let P be <cat> = P
<head infl valence> = !active
<head type wh> = -
<head type partitive> = !-
<head type temporal> = !-
<head type sentential> = !-
<head type gerund> = -
<head type participle> = -
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
<head type ref_phrase_head> = !-

Let Poss be <cat> = Poss

[genitive]
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Poss_ergative be <cat> = Poss
[ergative]
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Poss_absolutive be <cat> = Poss
[absolutive]
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let Pron be <cat> = Pron
<head type wh> = !-
<head agr animacy> = !animate
<head type reciprocal> = !-
<head type reflexive> = !-
<head type possessive> = !-
<head type relative> = !-
<head type temporal> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = !+
<head type emphatic> = !-
<head type locative> = !-
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let PropN be <cat> = N
<head agr animacy> = !animate
<head agr person> = !third
<head possessor head agr person> = !none
<head type prefix case> = !none
<head type suffix case> = !none
<head type wh> = -
<head type proper> = +
<head type reciprocal> = !-
<head type reflexive> = !-
<head type sentential> = !-
<head type relative> = !-

<head type mass> = !-
<head type temporal> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
<head case> = !none

Let Q be <cat> = Q
<head type wh> = -
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type quantifier> = +
<head type comma> = -
<head type topic> = -
<head type prefix conj> = -
<head type suffix conj> = -
[head : [type : [numeric : -]]]

Let V be <cat> = V
<head type sentential> = !-
<head infl valence> = !active
<head type sentential_with_object> = !-
<head type copular> = !-
<head type existential> = !-
<head type perception> = !-
<head type motion> = !-
<head type reciprocal> = !-
<head type reflexive> = !-
<head infl finite> = !+
<head infl polarity> = !positive
<head type participle> = !-
<head infl mood> = !declarative
<head embedded cat> = none
<head type focusmarked> = -
<head type pronoun-fronted> = !-
<head infl voice> = !actor
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-

Let makeAdj be <cat> = Adj

<head type ordinal> = !-
<head type wh> = !-
<head type sentential> = !-
<head type clausal-comp> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-

Let makeAdv be <cat> = Adv
<head type wh> = !-
<head type manner> = !-
<head type existential> = !-
<head type takes_Adv> = !-
<head type takes_DP> = !-
<head type sentential> = !-
<head embedded cat> = !none
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-

Let makeArt be <cat> = Art
<head type wh> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-

Let makeAux be <cat> = Aux
<head infl valence> = !active
<head infl polarity> = !positive
<head type gerund> = -
<head type participle> = -
<head type existential> = -
<head infl finite> = !+
<head type pronoun> = -

<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeC be <cat> = C
<head type question> = !-
<head type relative> = !-
<head embedded cat> = !none
<head type pronoun> = -
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeConj be <cat> = Conj
<head infl polarity> = !positive
<head agr number> = !singular
<head type pronoun> = -
<head type comma> = !-
<head type comparative> = !-
<head type compareAdj> = !-
<head type compareN> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeDeg be <cat> = Deg
<head type modifies_Adj> = !-
<head type modifies_Adv temporal> = !-
<head type modifies_Adv locative> = !-
<head type modifies_Adv manner> = !-
<head type modifies_Adv reason> = !-
<head type modifies_NP> = !-
<head type modifies_PP> = !-
<head type modifies_Q> = !-
<head type quantifier> = !-
<head type wh> = !-
<head type relative> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type comparative> = !-
<head type equalAdj> = !-
<head type equalN> = !-
<head type superlative> = !-
<head type prefix conj> = !-

<head type suffix conj> = !-
Let makeDem be <cat> = Dem
<head type wh> = !-
<head agr person> = !third
<head type relative> = !-
<head type temporal> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type emphatic> = !-
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeDet be <cat> = Det
<head type wh> = !-
<head agr person> = !third
<head type relative> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeExcl be <cat> = Excl
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeFocusM be <cat> = FocusM
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeGreet be <cat> = Greet
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeInitialConj be <cat> = InitConj
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeIntj be <cat> = Intj

<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeN be <cat> = N
<head agr person> = !third
<head agr animacy> = !inanimate
<head possessor head agr person> = !none
<head type prefix case> = !none
<head type suffix case> = !none
<head type wh> = -
<head type reciprocal> = !-
<head type reflexive> = !-
<head type sentential> = !-
<head type proper> = !-
<head type relative> = !-
<head type mass> = !-
<head type temporal> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
<head case> = !none
Let makeNum be <cat> = Num
<head type ordinal> = !-
<head type wh> = !-
<head agr number> = !plural
<head type relative> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type quantifier> = +
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeP be <cat> = P
<head infl valence> = !active
<head type wh> = -
<head type partitive> = !-

<head type temporal> = !-
<head type sentential> = !-
<head type gerund> = -
<head type participle> = -
<head type pronoun> = -
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makePron be <cat> = Pron
<head type wh> = !-
<head agr animacy> = !animate
<head type reciprocal> = !-
<head type reflexive> = !-
<head type possessive> = !-
<head type relative> = !-
<head type temporal> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type pronoun> = +
<head type emphatic> = !-
<head type locative> = !-
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeQ be <cat> = Q
<head type wh> = !-
<head type focusmarked> = -
<head type gerund> = -
<head type participle> = -
<head type quantifier> = +
<head type pronoun> = -
<head type comma> = !-
<head type topic> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeTopicM be <cat> = TopicM
<head type pronoun> = -
<head type comma> = !-
<head type prefix conj> = !-
<head type suffix conj> = !-
Let makeV be <cat> = V
<head infl valence> = !active

```
        <head type sentential> = !-
        <head type sentential_with_object> = !-
        <head type copular> = !-
        <head type existential> = !-
        <head type perception> = !-
        <head type motion> = !-
        <head type reciprocal> = !-
        <head infl finite> = !+
        <head infl polarity> = !positive
        <head type gerund> = !-
        <head type participle> = !-
        <head infl mood> = !declarative
        <embedded cat> = !none
        <head type focusmarked> = -
        <head type pronoun-fronted> = !-
        <head infl voice> = !actor
        <head type comma> = !-
        <head type topic> = !-
        <head type prefix conj> = !-
        <head type suffix conj> = !-
Let class_inanimate be <head agr class> = inanimate
Let class_human be <head agr class> = human
Let new be [subcat : [first : [head : [cat : Q]]
                rest : none]]
Let takesQpComp be [subcat : [first : [head : [cat : Num]]
                rest : none]]
Let transitiveTFH be [head : [type : [transitivity : transitive]]]
        [subcat : [first : [cat : NP]
                rest : [subcat : [first : [cat : NP]
                rest : none]]]]
Let ditransitiveTFH be [head : [type : [transitivity : ditransitive]]]
        [subcat : [first : [cat : NP]
                rest : [subcat : [first : [cat : NP]
                rest : [cat : PP]]]]]
Let intransitiveTFH be [head : [type : [transitivity : intransitive]]]
        [subcat : [first : [cat : NP]
                rest : [subcat : [first : none
                rest : none]]]]
Let ditransNPPP be [subcat : [first : [cat : NP]
                rest : [cat : PP]]]
Let ditransNPNP be [subcat : [first : [cat : NP]
                rest : [cat : NP]]]
Let kanPlus be [head : [type : [transitivity_class : [kan : +]]]]
Let kanMinus be [head : [type : [transitivity_class : [kan : -]]]]
```

```
Let Copula be [head : [type : [transitivity : transitive]]]
Let RelativeYang be [subcat : [first : [head : [type : [relative : +]]]
                    rest : none]]
          [head : [type : [relative : +]]]
Let dependant_quantifier be [head : [type : [free_quantifier : -]]]
Let headsRefPhrase be [head : [type : [ref_phrase_head : +]]]
Let bukanTemplate be [subcat : [first : [cat : NP]
                    rest : none]]
Let ditransitiveKanTFH be [subcat : [first : [cat : NP]
                    rest : [subcat : [first : [cat : NP]
                    rest : [cat : NP]]]]]
Let PassiveType1 be [head : [agr : [person : third]]subcat : [first : [cat : NP]
                    rest : [subcat : {[first : [cat : NP]
                    rest : none]
          [first : [cat : PP]
                    rest : none]
          [first : none
                    rest : none]}]]]
          <head infl valence> = passive
Let AgentAffixedPassiveTransitiveVerb be [subcat : [first : [cat : NP]
                    rest : [subcat : [first : none
                    rest : none]]]]
          <head infl valence> = passive
Let intransitiveTFHMotion be [subcat : [first : [cat : NP]
                    rest : {[first : [cat : PP]
                    rest : none]}]]
rule {ruleSIP} | S IP
S = IP
 <S head> = <IP head>
 <S gap> = <IP gap>
 <IP head type root> = +
 <IP gap first> = none
 <IP gap rest> = none
rule {ruleSNpPredP} | IP Np I'
IP = NP I'
 <IP head> = <I' head>
 <NP head type possessive> = -
 <I' subcat first> = <NP>
 <I' subcat rest subcat first> = none
 <I' subcat rest subcat rest> = none
 <IP gap first> = <I' gap first>
 <IP gap rest> = <I' gap rest>
rule {ruleSAdvPS} | IP AdvP IP
IP_1 = AdvP IP_2
```

<IP_1 head> = <IP_2 head>
rule {rulessadvp} | IP IP advP
IP_1 = IP_2 AdvP
<IP_1 head> = <IP_2 head>
rule {rulePredBar_NpVpOrPp} | Indonesian can have VP, PP, NP, or AdjP as predicate center
| IBar Np Vp Pp AdjP or QP
I' = { VP / PP / NP / AdjP / QP }
<I' head> = <VP head>
<I' gap> = <VP gap>
<VP head type verbheaded> = +
<I' head> = <PP head>
<I' gap> = <PP gap>
<I' head> = <NP head>
<I' gap> = <NP gap>
<I' head> = <AdjP head>
<I' gap> = <AdjP gap>
<I' head> = <QP head>
<AdjP head type substantive> = -
<AdjP subcat first> = <I' subcat first>
<I' subcat first> = <VP subcat first>
<I' subcat rest> = <VP subcat rest>
rule {ruleVbarAdvPVbar} | IBar AdvP IBar
I'_1 = AdvP I'_2
<I'_1 head> = <I'_2 head>
<I'_1 head type prefix copular> = -
rule {rulePredBarAuxPredBar} | IBar Aux IBar
I'_1 = Aux I'_2
<I'_1 head> = <I'_2 head>
<I'_1 gap> = <I'_2 gap>
<I'_2 head type copular> = -
<I'_1 subcat first> = <I'_2 subcat first>
<I'_1 subcat rest> = <I'_2 subcat rest>
rule {ruleIPCopIBar} | I' Cop I'
I'_1 = Cop I'_2
<I'_1 head> = <I'_2 head>
<I'_2 head> == ~[type : [verbheaded : [+ : ^]]]
<I'_2 head type prefix copular> = -
<I'_1 head type suffix copular> <= +
rule {rulePredPConjoined} | IBar (initConj) IBar Conj IBar
I'_1 = (InitConj) I'_2 Conj I'_3
<I'_1 head> = <I'_2 head>
rule {ruleVpVbar} | Vp V'
VP = V'

```
 <VP head> = <V' head>
 <VP gap> = <V' gap>
 <VP subcat first> = <V' subcat first>
 <VP subcat rest subcat first> = <V' subcat rest subcat first>
 <V' subcat rest subcat first> = none
rule {ruleVbarV} | V' V
V' = V
 <V' head> = <V head>
 <V' subcat first> = <V subcat first>
 <V' subcat rest subcat first> = <V subcat rest subcat first>
 <V' subcat rest subcat rest> = <V subcat rest subcat rest>
rule {rulevbarVPP} | V' V' Pp
V'_1 = V'_2 PP
 <V'_1 head> = <V'_2 head>
 <V'_2 subcat rest subcat first> = <PP>
 <V'_1 subcat rest subcat first> = <V'_2 subcat rest subcat rest>
rule {ruleVbarVAdvP} | Vbar V AdvP
V' = V AdvP
<V' head> = <V head>
rule {ruleVBarV[embedded, transitive]} | V' V[transitive, embedded]
V' = V
 <V' head> = <V head>
 <V' head type root> = -
 <V' subcat first> = <V subcat first>
{
 <V head type transitivity> = transitive
 <V' gap first> = <V subcat rest subcat first>
 <V' gap rest> = <V subcat rest subcat rest>
 <V' subcat rest subcat rest> = none
/
 <V head type transitivity> = intransitive
}| subject gap and object gap
rule {ruleVbarVbarNp} | V' V' NP
V'_1 = V'_2 NP
 <V'_1 head> = <V'_2 head>
 <V'_2 subcat rest subcat first> = <NP>
 <V'_1 subcat rest subcat first> = <V'_2 subcat rest subcat rest>
rule {ruleVbarNPVbarPassiveAgent} | V' NP V' - passive agent
V'_1 = NP V'_2
 <V'_1 head> = <V'_2 head>
 <V'_2 head infl valence> = passive
 <V'_2 subcat rest subcat first> = <NP>
 <V'_1 subcat first> = <V'_2 subcat first>
```

```
 rule {rule75} | NP option Prona - pronoun,dem,quantifiers w/o head noun (optional PP
after)
 NP = { Pron / AdjP / Q / Num / Deg } (PP )
  <NP head> = <Pron head>
  <NP gap first> = none
  <Pron head type possessive> = -
  <Pron head type reflexive> = -
  <Pron head type reciprocal> = -
  <NP head> = <AdjP head>
  <NP head> = <Q head>
  <NP head> = <Num head>
  <NP head> = <Deg head>
  <Deg head type quantifier> = +
  <AdjP head type wh> = -
  <PP head type stranded> = -
  <PP head type sentential> = -
  <NP head type prefix poss> = -
  <NP head type suffix poss> = -
  <NP head type comma> <= <PP head type comma>
  <NP option> = Prona
  <NP gap> = <AdjP gap>
  <Num head type free_quantifier> = +
| these use DP rule "Pron"
 | so "which" not separate
 | PP must have overt complement
 | sentential not within NP
 | <PP head type partitive> = +
 | can't be possessors
 | can't be possessors
 rule {npDegNp} | NP - Deg NP
 NP_1 = Deg NP_2
  <NP_1 head> = <NP_2 head>
  <NP_1 gap> = <NP_2 gap>
 rule {NPnbardemfinal} | Np nbar dem final
 NP = { N' / N" } Dem
  <NP head> = <N' head>
  <NP head> = <N" head>
 rule {npNbarNpPos} | NBar Nbar Np[poss]
 NP_1 = { N' / N" } NP_2
  <NP_2 head type possessive> = +
  <NP_1 gap> = <N' gap>
  <NP_1 gap> = <N" gap>
  <NP_1 head> = <N' head>
  <N' head type possessed> = +
```

<N" head type possessed> = +
<NP_1 head> = <N" head>
<NP_2 head type ordinal> = -
<NP_2 gap first> <= none
rule {npnbar} | NP Nbar
NP = N'
<NP head> = <N' head>
<NP gap> = <N' gap>
rule {npdetnbar} | Np Det NBar
NP = Det N'
<NP head> = <N' head>
rule {NPfinal_qp} | NBar Nbar Qp final
N'_1 = N'_2 QP
<QP head type ordinal> = +
<N'_1 head> = <N'_2 head>
<QP head type numeric> = +
rule {npQpNbar} | Nbar Qp Nbar
N'_1 = QP N'_2
<N'_1 head> = <N'_2 head>
rule {npnbaradjp} | NBar Nbar AdjP
N'_1 = N'_2 AdjP
<N'_1 head> = <N'_2 head>
<N'_1 gap> = <AdjP gap>
<AdjP subcat first> =
[cat : NP]
rule {nbarN} | Nbar N
N' = N
<N' head> = <N head>
<N' gap> = <N gap>
rule {nbarHonN} | Nbar Hon N
N' = Hon N
<N' head> = <N head>
rule {ruleNbarAdjPNBar} | NBar AdjP NBar
N'_1 = AdjP N'_2
<N'_1 head> = <N'_2 head>
rule {ruleNBarNBarPP} | Nbar Nbar PP
N'_1 = N'_2 PP
<N'_1 head> = <N'_2 head>
rule {ruleNBarDem} | NBar Dem[-pos]
N' = Dem
<N' head> = <Dem head>
<N' head type possessive> = -
<N' gap> = none
rule {ruleNBarNCP} | N is a stop gap for NP gaps in CP

```
 | Nbar N CP
 N' = N CP
  <N' head> = <N head>
  <N' gap> = <N gap>
  <CP head type relative> = +
  <N' gap first> = <CP gap rest>
  <CP stop_gap> =
          [cat : NP]
<CP stop_gap> <= <CP gap first>
rule {ruleCPCBar} | CP CBar
 CP = C'
  <CP head> = <C' head>
  <CP gap> = <C' gap>
 rule {ruleCBarCIP} | C' C IP
 C' = C IP
  <C' head> = <C head>
  <C' gap> = <IP gap>
  <C subcat first> = <IP>
  <IP head type root> = -
 rule {rule85} | PP option conj - conjoined PPs
 PP_1 = (InitConj ) PP_2 Conj PP_3
  <PP_1 head> = <PP_3 head>
  <PP_1 conjoined> = +
  <PP_2 conjoined> = -
  <PP_1 option> = conj | limit recursion
 rule {rule86} | PP option conjNone - conjoined PPs
 PP_1 = PP_2 PP_3
  <PP_1 head> = <PP_3 head>
  <PP_1 conjoined> = +
  <PP_2 conjoined> = -
  <PP_3 head type prefix conj> = -
  <PP_2 head type suffix conj> = -
  <PP_2 head type comma> = -
  <PP_1 option> = conjNone | limit recursion
 rule {rule87} | PP option 0 - no modifiers
 PP = P'
  <PP head> = <P' head>
  <PP option> = 0
 rule {rule92} | PBar option 2c - prepositions, temporal AdvP complement
 P' = P AdvP
  <P' head> = <P head>
  <P head type PP-initial> = +
  <AdvP head type temporal> = +
  <P' head type stranded> = -
```

<P head type comma> = -
<P' head type comma> <= <AdvP head type comma>
<P' head type suffix poss> <= <AdvP head type suffix poss>
<P' head type temporal> <= <AdvP head type temporal>
<P' head type locative> <= <AdvP head type locative>
<P' head type sentential> <= -
<P' option> = 2c | not missing a complement
rule {rulePBarNP} | PBar NP[-pos]
P' = P NP
<P' head> = <P head>
<NP head type possessive> = -
rule {rule95} | AdjP option conj - conjoined AdjPs
AdjP_1 = (InitConj) AdjP_2 Conj AdjP_3
<AdjP_1 head> = <AdjP_3 head>
<AdjP_1 conjoined> = +
<AdjP_2 conjoined> = -
<AdjP_1 head type clausal-comp> = -
<AdjP_1 head type prefix> <= <AdjP_2 head type prefix>
<AdjP_1 option> = conj| limit recursion
rule {rule96} | AdjP option conjNone - conjoined AdjPs
AdjP_1 = AdjP_2 AdjP_3
<AdjP_1 head> = <AdjP_3 head>
<AdjP_1 conjoined> = +
<AdjP_2 conjoined> = -
<AdjP_1 head type clausal-comp> = -
<AdjP_3 head type prefix conj> = -
<AdjP_2 head type suffix conj> = -
<AdjP_2 head type comma> = -
<AdjP_1 head type prefix> <= <AdjP_2 head type prefix>
<AdjP_1 option> = conjNone | limit recursion
rule {rule98} | AdjP option 0 - no modifiers
AdjP = Adj'
<AdjP head> = <Adj' head>
<AdjP option> = 0
<AdjP gap first> = none
<AdjP subcat first> = <Adj' subcat first>
rule {rule99} | AdjP option 1m - degree or AdvP modifiers initial
AdjP = { Deg / AdvP } Adj'
<AdjP head> = <Adj' head>
<Deg head type modifies_Adj> = +
<AdvP head type manner> = +
<Deg head type AdjP-initial> = +
<AdvP head type AdjP-initial> = +
<Deg head type comma> = -

<AdvP head type comma> = -
<AdjP head type prefix> <= <Deg head type prefix>
<AdjP head type prefix> <= <AdvP head type prefix>
<AdjP option> = 1m
rule {rule100} | AdjP option 2m - degree or AdvP modifiers final
AdjP = Adj' { Deg / AdvP }
<AdjP head> = <Adj' head>
<Deg head type modifies_Adj> = +
<AdvP head type manner> = +
<Deg head type AdjP-final> = +
<AdvP head type AdjP-final> = +
<Adj' head type comma> = -
<AdjP head type comma> <= <Deg head type comma>
<AdjP head type comma> <= <AdvP head type comma>
<AdjP head type suffix> <= <Deg head type suffix>
<AdjP head type suffix> <= <AdvP head type suffix>
<AdjP option> = 2m
rule {rule101} | AdjP option 3m - degree or AdvP modifiers both sides
AdjP = { Deg_1 / AdvP_1 } Adj' { Deg_2 / AdvP_2 }
<AdjP head> = <Adj' head>
<AdvP_1 head type manner> = +
<AdvP_2 head type manner> = +
<Deg_1 head type modifies_Adj> = +
<Deg_2 head type modifies_Adj> = +
<Deg_1 head type AdjP-initial> = +
<Deg_2 head type AdjP-final> = +
<Deg_1 head type comma> = -
<AdvP_1 head type comma> = -
<Adj' head type comma> = -
<AdjP head type comma> <= <Deg_2 head type comma>
<AdjP head type comma> <= <AdvP_2 head type comma>
<AdjP head type prefix> <= <Deg_1 head type prefix>
<AdjP head type suffix> <= <Deg_2 head type suffix>
<AdjP head type prefix> <= <AdvP_1 head type prefix>
<AdjP head type suffix> <= <AdvP_2 head type suffix>
<AdjP option> = 3m
rule {rule102} | Adj' option 0 - no complements
Adj' = Adj
<Adj' head> = <Adj head>
<Adj' head type clausal-comp> = -
<Adj' option> = 0
<Adj' subcat first> = <Adj subcat first>
rule {ruleAdjBarAdjPP} | Adj' Adj PP - referent phrase
Adj' = Adj PP

<Adj' head> = <Adj head>
<Adj head type ref_phrase_head> = +
rule {ruleAdjBarAdjCp} | Adj' Adj CP
Adj' = Adj CP
<Adj' head> = <Adj head>
<Adj stop_gap> = <Adj subcat first>
<Adj' gap first> = <CP gap rest>
<CP stop_gap> <= <CP gap first>
rule {rule104} | AdvP option 0 - no modifiers
AdvP = Adv'
<AdvP head> = <Adv' head>
<AdvP option> = 0
rule {ruleAdvPModifiers} | AdvP option 1 - initial modifiers
AdvP = QP Adv'
<AdvP head> = <Adv' head>
rule {rule105} | AdvBar option 0 - no complements
Adv' = Adv
<Adv' head> = <Adv head>
<Adv' option> = 0
rule {rule106} | AdvBar option 1f - Adv complements final
Adv' = Adv_1 Adv_2
<Adv' head> = <Adv_1 head>
<Adv_1 head type temporal> = +
<Adv_2 head type temporal> = +
<Adv_1 head type takes_Adv> = +
<Adv_1 head type comma> = -
<Adv' head type comma> <= <Adv_2 head type comma>
<Adv' head type suffix> <= <Adv_2 head type suffix>
<Adv' option> = 1f
rule {rule107} | AdvBar option 2f - DP complements final
Adv' = Adv NP
<Adv' head> = <Adv head>
<Adv head type takes_DP> = +
<Adv head type comma> = -
<Adv' head type comma> <= <NP head type comma>
<Adv' head type suffix> <= <NP head type suffix>
<Adv' option> = 2f
rule {rule109} | AdvBar option 4f - CP complements final
Adv' = Adv CP
<Adv' head> = <Adv head>
<Adv head type sentential> = +
<Adv head embedded> = <CP>
<CP head type question> = -
<CP head type relative> = -

<CP head type root> = -
<Adv head type comma> = -
<Adv' head type comma> <= <CP head type comma>
<Adv' head type suffix> <= <CP head type suffix>
<Adv' option> = 4f
rule {ruleAdvBarAdvNp} | AdvBar Adv Np
Adv' = Adv NP
<Adv' head> = <Adv head>
rule {rule110} | QP option 0 - no modifiers
QP = Q
<QP head> = <Q head>
<QP option> = 0
rule {rule111} | QP option 1 - modifiers initial
QP = Deg Q'
<QP head> = <Q' head>
<Deg head type modifies_Q> = +
<Deg head type QP-initial> = +
<Deg head type comma> = -
<QP head type prefix> <= <Deg head type prefix>
<QP option> = 1
rule {rule113} | QP option Num0i - no modifiers, head initial
QP = Num_1 (Conj_1) (Num_2 (Conj_2) (Num_3 (Conj_3) (Num_4 (Conj_4) (Num_5 (Conj_5) (Num_6)))))
<QP head> = <Num_1 head>
<Num_1 head type quantifier> = +
<Conj_1 head type conjoins_DP> = +
<Conj_2 head type conjoins_DP> = +
<Conj_3 head type conjoins_DP> = +
<Conj_4 head type conjoins_DP> = +
<Conj_5 head type conjoins_DP> = +
<QP option> = Num0i
<Num_1 head type free_quantifier> = +
rule {qpQbar} | Qp Qbar
QP = Q'
<QP head> = <Q' head>
rule {ruleQBarQ} | QBar Q
Q' = Q
<Q' head> = <Q head>
rule {qbarNumQ} | Qbar Num Q
Q' = Num Q
<Q' head> = <Q head>
<Q headsubcat first cat> = <Num subcat none>
Parameter Start symbol is S.
|

```
| Lexical rules
|
Define PassiveType2 as
 <out head infl valence> = passive
 <out subcat first> = <in subcat first>
 <out subcat rest> = <in subcat rest>
```

REFERENCES

Black, Cheryl A. 1997. PC-PATR implementation of GB syntax. *SIL Eletronic Working Papers* 1997(007).

Black, Cheryl A. & H. Andrew Black. 2012. PAWS Starter Kit. Online: http://carla.sil.org/paws.htm. (Accessed 31 Jan 2013.)

Black, Cheryl A. & H. Andrew Black. 2009. PAWS: Parser and writer for syntax: drafting syntactic grammars in the third wave. *SIL Forum for Language Fieldwork* 2009(002).

Black, H. Andrew. 2009a. PcPatr Browser. Online: http://lingtransoft.info/apps/pcpatr-browser. (Accessed 26 Mar 2013.)

Black, H. Andrew. 2009b. LingTree. Online: http://lingtransoft.info/apps/lingtree. (Accessed 26 Mar 2013.)

Black, H. Andrew. 2012. *PC-PATR in XML.* Online: http://sourceforge.net/projects/pcpatrinxml/. (Accessed 3 May 2013.)

Crabbé, Benoit, Denys Duchier, Joseph Le Roux, & Yannick Parmentier. 2008. XMG - eXtensible MetaGrammar. Online: https://sourcesup.renater.fr/xmg/. (Accessed 23 Feb 2013.)

Dardjowidjojo, Soenjono. 1978. *Sentence patterns of Indonesian.* Jakarta: Djambatan.

Gomolka, Andreas & Bernhard Humm. 2012. Structued editors: old hat or future vision? *Communications in Computer and Information Science* 0275.82-97. Online: https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Gomolka_Humm_-_Structure_Editors__Springer_ENASE_.pdf. (Accessed 16 April 2012.)

Guillaume, Bruno. 2012a. Leopar. Online: http://leopar.loria.fr/doku.php?id=leopar:leopar. (Accessed 22 Feb 2013.)

Guillaume, Bruno. 2012b. Lexicomp. Online: http://leopar.loria.fr/doku.php?id=lexicomp:lexicomp. (Accessed 23 Feb 2012.)

Guillaume, Bruno, & Guy Perrier. 2010. Interaction grammars. *Research on Language and Computation* 00.1-44.

Larasati, Septina Dian. 2012. IDENTIC Corpus: morphplogically enriched Indonesian - English parallel corpus (poster session). LREC 2012. Istanbul.

Lewis, M. Paul. 2009. *Ethnologue: languages of the world.* 16 Edition. Dallas: SIL International. Online: http://www.ethnologue.com/.

McConnel, Stephen. 2006. PC-PATR reference manual. Online: http://www.sil.org/pcpatr/manual/pcpatr.html. (Accessed 10 February 2012.)

Parmentier, Yannick. 2005. XMG/Documentation. Online: http://wiki.loria.fr/wiki/XMG/Documentation. (Accessed 15 Mar 2013.)

Schematron. 2010. Schematron. Online: http://www.schematron.com/. (Accessed 3 June 2013.)

Shieber, Stuart M. 2003. *An introduction to unification-based approaches to grammar.* Brookline: Microtome.

Shieber, Stuart M. 1985. Evidence against the context-freeness of natural language. *Linguistiics and Philosophy* 8.333-43. Online: http://nrs.harvard.edu /urn-3:HUL.InstRepos:2026618.

SIL International. 2005. Test data files. Online: ftp://ftp.sil.org/software/unix/pc-parse-tst.zip. (Accessed 20 Mar 2013.)

Sleator, Daniel, D. K. & Davy Temperly. 1991. Parsing English with link grammar. School of Computer Science. Carnagie Mellon University.

Sneddon, James Neil. 1996. *Indonesian: a comprehenisve grammar.* New York: Routledge.

Stanford Natural Language Processing Group. 2013a. The Stanford Parser: A statistical parser. Online: http://nlp.stanford.edu/software/lex-parser.shtml. (Accessed 22 Feb 2013.)

Stanford Natural Language Processing Group. 2013b. Stanford Parser FAQ. Online: http://nlp.stanford.edu/software/parser-faq.shtml. (Accessed 19 Mar 2013.)

Temperly, Davy. 1999. An Introduction to the Link Grammar Parser. Online: http://www.link.cs.cmu.edu/link/dict/introduction.html. (Accessed 15 Mar 2013.)

Temperly, Davy, Daniel Sleator, & John Lafferty. 2012. Link Grammar. Online: http://www.link.cs.cmu.edu/link/. (Accessed 15 Mar 2013.)