Pronouns, Second Edition (Modula-2 Version)

Kelly Roach
kellybrianroach@outlook.com
`https://www.kellyroach.com`

(12 December 2024)

Contents

Preface

Pronouns, Second Edition is a 2024 LaTeX-formatted version of the author's original 1980 Caltech M.S. thesis, *Pronouns* [27]. The content of *Pronouns, Second Edition* is substantially the same as the original *Pronouns* with the following principal differences:

- OCR'd content of *Pronouns* converted to modern LaTeX style.
- Misspellings, minor grammar points, numberings, and minor technical points are corrected.
- Capitalization changes.
- Figure captions shortened.
- PEP 8 Coding Style identifier spellings.
- Tikz figures replace partially hand-drawn TXT figures.
- LaTeX tabular tables replace TXT tables.
- LaTeX References replace TXT References.
- LaTeX Index added.
- Preface added.

Caltech's "Usage Policy", inherited from the original *Pronouns*, states:

Postscript and PDF versions of the original *Pronouns* are also available on the author's PLANETQUANTUM.COM website [26].

The author's M.S. thesis and undergraduate adviser was Frederick B. Thompson [34].

Introduction

Certain substitutions and abbreviations occur in English which are not well understood yet that we would like to understand better so that we may implement them in computer natural language systems intended for man-machine communication. These include **pronouns** and other **function words** like those below in Figure 0.1 acting both in isolation and with each other.

I	me	my	myself	mine	we
us	our	ours	ourselves	you	your
yours	yourself	yourselves	he	him	his
himself	she	her	hers	herself	it
its	itself	they	them	their	theirs
themselves	this	that	these	those	one
ones	oneself	other	others	all	none
some	any	each	which	what	who
whom	whose	another	do	does	did
done	doing	so			

Figure 0.1. Pronouns and Other Function Words

As well, we have noun phrases modified by **demonstratives**, Head Deletion, and Equi-NP Deletion. Bloomfield [2] defined **substitution** as a replacement operation.

> A **substitute** is a linguistic form or grammatical feature which, under certain conventional circumstances, replaces any one of a class of linguistic forms. Thus, in English, the substitute I replaces any singular-number substantive expression, provided that this substantive expression denotes the speaker of the utterance in which the substitute is used.

In this thesis we will be concerned with pronouns. Possibly because this will be the only chance we get, we should note the wide variety of substitution mechanisms in general. Examples (0.2)-(0.11) are from Sag [30].

(0.2) **Do It Anaphor**
Jerry won't prove that theorem; Alice will do it.
[do it = prove that theorem]

(0.3) **Sentential It Anaphor**
I believe that she means business and you'd better believe it too.
[it = that she means business]

(0.4) **Null Complement**
They asked me to leave but I refused ϕ.
[ϕ = to leave]

(0.5) **Ones Pronominalization**
Betsy has a blue car, and Randy has a red one.
[one = car]

(0.6) **Verb Phrase Deletion**
Joan wouldn't eat a Quarter Pounder, but Annie would ϕ.
[ϕ = eat a Quarter Pounder]

(0.7) **Sluicing**
Someone has drunk my entire six-pack of Schlitz Light, but I don't know who ϕ.
[ϕ = has drunk my entire six-pack of Schlitz-Light]

(0.8) **Stripping**
Gwendolyn snorts cocaine, but ϕ_1 not ϕ_2 in her own apartment.
[ϕ_1 = Gwendolyn (does), ϕ_2 = snort cocaine]

(0.9) **Gapping**
Erichman duped Haldeman and Nixon ϕ Mitchell.
[ϕ = duped]

(0.10) **Conjunction Reduction**
Mitchell lied to the committee and ϕ was sentenced last year.
[ϕ = Mitchell]

(0.11) **So Anaphor**
Mitchell said he was innocent and Nixon said so too.
[so = he was innocent]

To this list we can add pronominalizations. Examples (0.12)-(0.14) are from Lees and Klima [22].

(0.12) **Reflexive Pronominalization**
Mary's father supported himself.
[himself = Mary's father]

(0.13) **Pronominalization**
Mary's father supported her.
[her = Mary]

(0.14) **Reciprocal Pronominalization**
John and Mary kissed each other.
[each other = John and Mary]

And we might add (0.15) and (0.16) as well.

(0.15) **Head Deletion**
Joan's cat purrs but Mary's ϕ doesn't.
[ϕ = cat]

(0.16) **Equi-NP Deletion**
John is afraid of ϕ cutting himself.
[ϕ = John's]

Clearly, this list starts to grow very large with addition or refinement and it is probably safe to say that many volumes could be written on substitution processes without putting it to bed. This thesis is about pronouns and chaining of pronouns, and so is much narrower in scope. But this is not much comfort if the goals are not clearly in sight. We are just as lost in the middle of Lake Michigan as we are in the middle of the Pacific Ocean if we don't have a horizon to steer us by.

Part of the problem with investigations of anaphora today is that there is no horizon to steer by. Even though work on anaphora continues in an intelligent way, little progress is being made towards a really comprehensive theory. Instead we have a lot of scattered and independent results.

One goal of this thesis, besides talking about pronouns, is to seek out an algorithmic framework on which to build theory. Accordingly, various data structures such as nodes, C-S-N trees, and chaining tables are created for this purpose. Hopefully, the reader will recognize these data structures as too simplistic and will be moved to improve upon them. This thesis is, by no means at all, a solution to pronouns. At best, it may be a small compass in the middle of Lake Michigan, but this is our approach.

1 Fundamentals

1.1 Introduction

This chapter describes notation and basic ideas that will be used throughout this thesis. Hopefully, most of the notation described in this chapter is already familiar to the reader, but if not, then this chapter should be self-contained enough to be understandable by a reader with less experience.

1.2 Sentences

Sentences are numbered and are kept separate from the text of discussion for ease of reference. For example, (1.1) is from Huddleston [13] and is an example of a **Bach Peters sentence**.

(1.1) The boy who was fooling her kissed the girl who loved him.

Ungrammatical sentences are prefixed with an asterisk (*****) and **sentences of questionable grammaticality** are prefixed with a question mark (**?**). Here, (1.3) is from Chomsky [5].

(1.2) *John killed herself.
(1.3) ?Colorless green ideas sleep furiously.

Subscripts are used to indicate identity between constituents, meaning roughly that they mean the same thing or denote the same referent. More properly, we may think of constituents having the same subscript as being chained together. Below, (1.4) and (1.5) are from Bresnan [3].

(1.4) Some studentsI think theyI are smarter than theyI are.
(1.5) *Some studentsI think some studentsI are smarter than some studentsI are.

Sometimes we enclose information in **brackets** at the beginning or end of a sentence. This same notation is also sometimes used as an alternative to subscripts in identifying constituents. Here, (1.6) is from Bresnan [3], (1.7) and (1.8) are from Roberts [28] and (1.9) is from Bloom and Hayes [1].

(1.6) My uncle has never ridden a camel but his brother has, although it was lame. [it = camel]
(1.7) Men are mortal. [All men are mortal]
(1.8) Men are waiting. [Some men are waiting]
(1.9) [Seeing a picture of John Smith] That's John Smith.

A **deletion site** is indicated by a ϕ. Example (1.10) is from Hockett [12].

(1.10) I like the fresh candy better than the stale ϕ. [ϕ = candy]

Deletion sites arising from transformations like **Equi-NP Deletion** are treated similar to pronouns in this paper. Although there are many different kinds of deletion sites with distinct properties, we won't pay attention to this distinction in this thesis.

The symbol = is used between sentences to indicate that they are equivalent, while the symbol $\neq$ is used between sentences to indicate that they are not equivalent. Below, (1.11)-(1.14) are from Ross [29].

(1.11) If John can, he will do it. =
(1.12) If he can, John will do it.
(1.13) John will do it if he can. $\neq$
(1.14) He will do it if John can.

1.3 Noun Phrases

Quantified noun phrases are **noun phrases** modified by **quantifiers**. Examples (1.15)-(1.18) are quantified noun phrases.

(1.15) all female astronauts
(1.16) at least 10 sexual perverts
(1.17) many notorious criminals
(1.18) nearly a dozen Unicorns

Genitives are **possessive noun phrases**. Examples (1.19)-(1.22) are genitives.

(1.19) Uncle Iggy's
(1.20) my cobra's
(1.21) the Nazi war criminal's
(1.22) the alien creatures'

A noun phrase can be **generic**, **specific**, or **nonspecific**, indicated respectively by (1.23)-(1.25) from Kuno [16].

(1.23) A cat is a malicious animal. [generic]
(1.24) I have a cat at home, but hate it. [specific]
(1.25) I want to get a cat for myself. [nonspecific]

A plural noun phrase can be **collective** or **distributive**. Examples (1.26)-(1.28) are from Fauconnier [7].

(1.26) The men gathered. [collective]
(1.27) The men took off their hats. [distributive]
(1.29) The men carried the couch. [ambiguous]

Sentence (1.29) is **ambiguous** because it can mean either (1.30) or (1.31).

(1.30) Each man of the men carried the couch.
(1.31) The team of men carried the couch.

Smith [33] has also noticed this distinction. This explains why (1.32)-(1.35) below are ambiguous.

(1.32) John and Mary bought the new book by John Steinbeck.
(1.33) Bricks and stones make strong walls.
(1.34) George and Marmaduke have dogs.
(1.35) Gerry likes ice cream and cake.

1.4 Pronouns

Pronouns are cross-classified by person, plural, gender, animate, reflexive, attributive possessive, and predicative possessive features among others.

First person pronouns are given in (1.36).

(1.36) I, me, myself, my, mine, we, us, our, ours, ourselves.

Second person pronouns are given in (1.37).

(1.37) you, yourself, yourselves, your, yours

Third person pronouns are given in (1.38).

(1.38) she, he, it, they, her, him, them, herself, himself, itself, themselves, his, its, their, hers, theirs

Singular pronouns are given in (1.39).

(1.39) I, me, myself, my, mine, you, yourself, your, yours, she, he, it, her, him, herself, himself, itself, his, its, hers

Plural pronouns are given in (1.40).

(1.40) we, us, our, ours, ourselves, you, yourselves, your, yours, they, them, themselves, their, theirs

Pronouns with female gender are given in (1.41).

(1.41) she, her, herself, hers

Pronouns with male gender are given in (1.42).

(1.42) he, him, himself, his

Animate pronouns are given in (1.43).

(1.43) I, me, myself, mine, you, yourself, yourselves, your, yours, she, he, they, her, him, herself, himself, themselves, his, their, hers, theirs

Inanimate pronouns are given in (1.44).

(1.44) it, they, them, itself, themselves, its, theirs

Reflexive pronouns are given in (1.45).

(1.45) myself, yourself, yourselves, herself, himself, itself, themselves

Attributive possessive pronouns are given in (1.46).

(1.46) my, your, her, his, its, their

Predicative possessive pronouns are given in (1.47).

(1.47) mine, yours, hers, his, its, theirs

Besides the pronouns given above, we also have ones pronouns and reciprocal pronouns. **Ones pronouns** are given in (1.48).

(1.48) one, oneself, one's

Reciprocal pronouns are given in (1.49).

(1.49) each other, one another, each other's, one another's

1.5 Features

We use three kinds of **features** in this thesis. The symbol **+** indicates presence of a feature. The symbol **−** indicates absence of a feature. And the symbol **?** indicates that the presence or absence of a feature is either unspecified or not applicable. In the coming chapters, we will speak of agreement of features. A ? feature agrees with any other feature. The only time two features do not agree is when we are comparing a + and a − feature. Using = to indicate agreement and ≠ to indicate nonagreement, we have Figure 1.50.

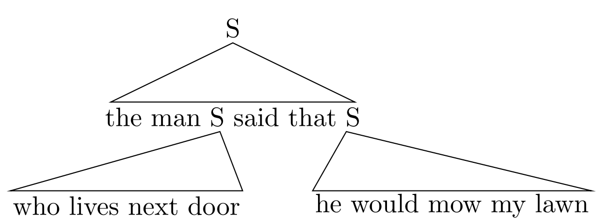
+ = +	+ = ?	+ ≠ −
? = +	? = ?	? = −
− ≠ +	− = ?	− = −

Figure 1.50. Agreement and Nonagreement between Features

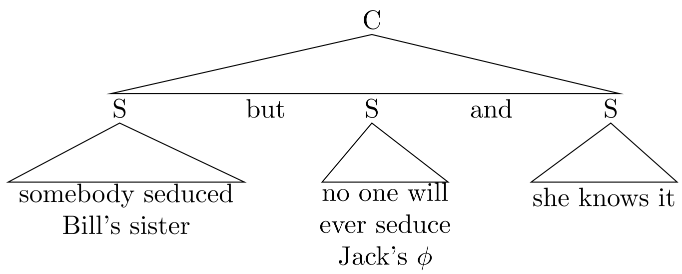
1.6 Parse Trees

Sentence **parse trees** are only drawn schematically in this thesis as extra detail is unnecessary. Parse trees shown more or less represent the surface structure of a sentence. Clause dominating nodes are labelled S and clause conjoining nodes are labelled C. In this thesis, genitives and adjectives are not treated as arising from transformations, but as occuring in the base component. Below, example (1.51) is from Huddleston [13] and example (1.52) is from Grosu [9].

(1.51) The man who lives next door said that he would mow my lawn.



(1.52) Somebody seduced Bill's sister, but no one will ever seduce Jack's and she knows it.



1.7 Clauses

Adverbial clauses are clauses beginning with an adverb. Some examples are (1.53)-(1.57) below.

(1.53) after Fido made a mess on the carpet
(1.54) before George kisses Betty
(1.55) since John is an asshole
(1.56) until Cathy behaves herself
(1.57) although Lile flunked all his classes

Clauses complemented with <u>that</u> are **that clauses**. Example (1.58) is a that clause.

(1.58) that Snoopy is a cat

Clauses modified by the **For-To Transformation** are **infinitive clauses**. Example (1.59) is an infinitive clause.

(1.59) for Ruth to choose

Clauses modified by the **Possessive-Ing Transformation** are **genitive clauses**. Example (1.60) is a genitive clause.

(1.60) Mary's kissing Bob

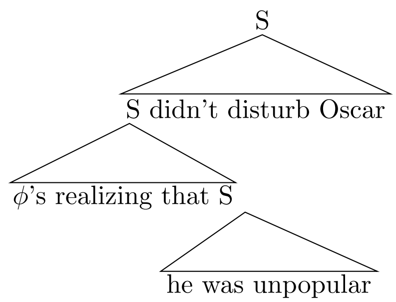
Clauses modified by **WH-Fronting Transformation** but not the **Question Transformation** and which modify noun phrases are **relative clauses**. Examples (1.61)-(1.65) are relative clauses.

(1.61) who ate five hamburgers
(1.62) that has a leaky faucet
(1.63) which doesn't run
(1.64) whom he gave it to

(1.65) whose life isn't worth a postage stamp

Clauses without embedded **subordinate clauses** are **simplex**. In example (1.66) from Ross[29], the simplexes are (1.67)-(1.69). In example (1.70), from Huddleston [13], the simplexes are (1.71)-(1.73). In example (1.74), from Huddleston, the simplexes are (1.75)-(1.77).

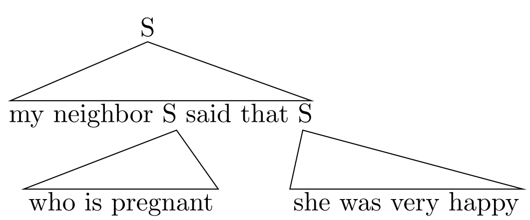
(1.66) Realizing that he was unpopular didn't disturb Oscar.



ϕ *precedes* he
ϕ *commands* he
ϕ *precedes* Oscar
Oscar *commands* ϕ
Oscar *commands* he

(1.67) S didn't disturb Oscar
(1.68) ϕ's realizing that S
(1.69) he was unpopular

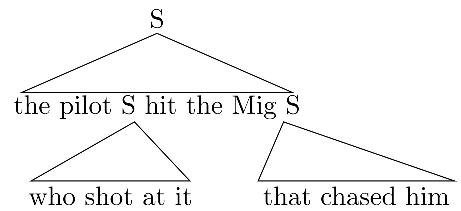
(1.70) My neighbor who is pregnant said that she was very happy.



neighbor *precedes* she
neighbor *commands* she

(1.71) my neighbor said that S
(1.72) who is pregnant
(1.73) she was very happy

(1.74) The pilot who shot at it hit the Mig that chased him.



pilot *precedes* him
pilot *commands* him
it *precedes* the Mig
Mig *commands* it

(1.75) the pilot hit the Mig
(1.76) who shot at it
(1.77) that chased him

1.8 Precedes and Commands

The ***precedes*** and ***commands*** relations, first described by Langacker [19], are defined below in (1.78) and (1.79).

(1.78) ***precedes*** **Relation**
A node A *precedes* another node B if

(a) neither A nor B ***dominates*** the other, and
(b) A occurs before B (in preorder traversal)

(1.79) ***commands*** **Relation**
A node A *commands* another node B if

(a) neither A nor B *dominates* the other, and
(b) the S-node that most immediately *dominates* A also *dominates* B

Another relation that will be useful is the ***is separate from*** relation defined below in (1.80).

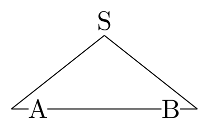
(1.80) ***is separate from*** **Relation**
A node R *is separate from* another node B if

(a) neither A nor B *dominates* the other, and
(b) the lowest node in the tree dominating A and B is a C-node.

We will see that *precedes*, *commands*, and *is separate from* are useful in determining when pronominalization is or isn't possible.

In example (1.81), A *precedes* B, A *commands* B, and E *commands* A. We don't have A *precedes* A, B *precedes* A, B *precedes* B, A *commands* A, or B *commands* B.

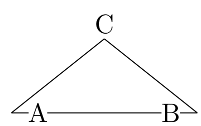
(1.81)



A *precedes* B
A *commands* B
B *commands* A

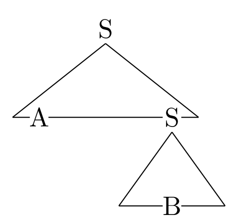
In (1.82), A *precedes* B, A *is separate from* B, and B *is separate from* A. In (1.83), A *precedes* B and A *commands* B. In (1.84), A *precedes* B and B *commands* A. In (1.85), A *precedes* B, H *is separate from* B, and B *is separate from* A.

(1.82)



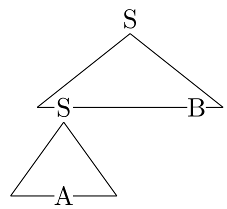
A *precedes* B
A *is separate from* B
A *precedes* B

(1.83)



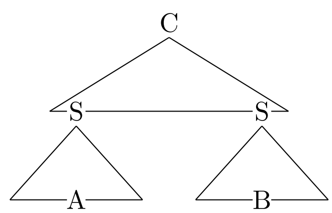
A *precedes* B
A *commands* B

(1.84)



A *precedes* B
B *commands* A

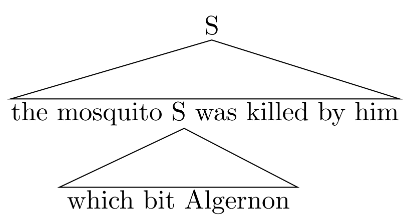
(1.85)



A *precedes* B
A *is separate from* B
B *is separate from* A

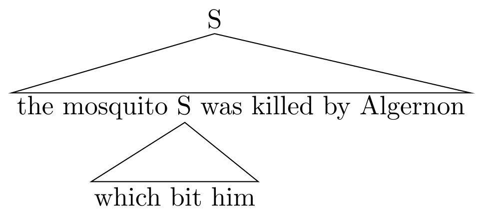
Examples (1.86)-(1.89) are from Langacker [19].

(1.86) The mosquito which bit Algernon was killed by him. [him = Algernon]



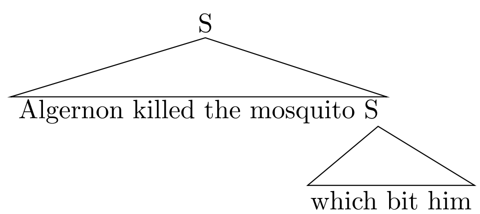
Algernon *precedes* him
him *precedes* Algernon

(1.87) The mosquito which bit him was killed by Algernon. [him = Algernon]



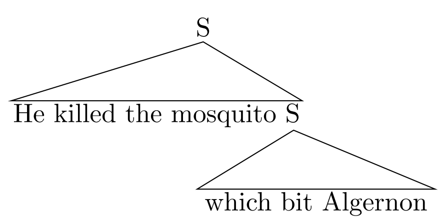
him *precedes* Algernon
Algernon *precedes* him

(1.88) Algernon killed the mosquito which bit him. [him = Algernon]



Algernon *precedes* him
him *commands* Algernon

(1.89) He killed the mosquito which bit Algernon. [he ≠ Algernon]



he *precedes* Algernon
Algernon *commands* he

The Precedes and Commands Rule, essentially as stated by Langacker [19], is given in (1.90) below.

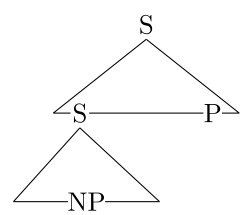
(1.90) **Precedes and Commands Rule**
A pronoun P may be used to pronominalize a noun phrase NP unless

(a) P *precedes* NP, and

(b) P *commands* NP or P *is separate from* NP

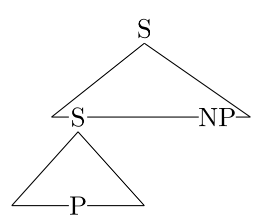
Note that the Precedes and Commands Rule explains the grammaticality and ungrammaticality of (1.86)-(1.89). These further examples from Ross [29] should drive the point home.

(1.91) After John Adams woke up, he was hungry. [he = John Adams]
(1.92) That Oscar was unpopular didn't disturb him. [him = Oscar]
(1.93) For your brother to refuse to pay taxes would get him into trouble. [him = your brother]
(1.94) Anna's complaining about Peter infuriated him. [him = Peter]
(1.95) The possibility that Fred will be unpopular doesn't bother him. [him = Fred]



NP *precedes* P
P *commands* NP

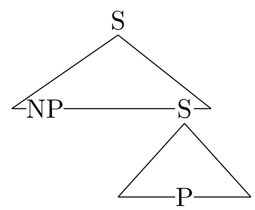
(1.96) After he woke up, John Adams was hungry. [he = John Adams]
(1.97) That he was unpopular didn't disturb Oscar. [he = Oscar]
(1.98) For him to refuse to pay taxes would get your brother into trouble. [him = your brother]
(1.99) Anna's complaining about him infuriated Peter. [him = Peter]
(1.100) The possibility that he will be unpopular doesn't bother Fred. [him = Fred]



P *precedes* NP
NP *commands* P

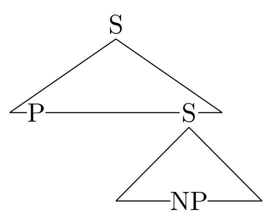
(1.101) John Adams was hungry after he woke up. [he = John Adams]

(1.102) Oscar wasn't disturbed that he was unpopular. [he = Oscar]
(1.103) It would get your brother into trouble for him to refuse to pay taxes. [him = your brother]
(1.104) Peter was infuriated at Anna's complaining about him. [him = Peter]
(1.105) Fred isn't bothered by the possibility that he will be unpopular. [he = Fred]



NP *precedes* P
NP *commands* P

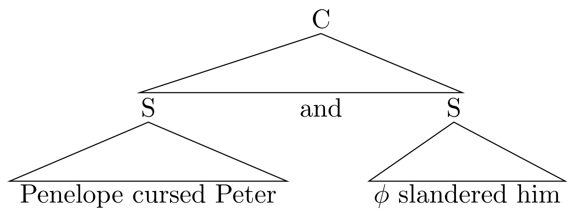
(1.106) *He was hungry after John Adams woke up. [he = John Adams]
(1.107) *He wasn't disturbed that Oscar was unpopular. [he = Oscar]
(1.108) *It would get him into trouble for your brother to refuse to pay taxes. [him = your brother]
(1.109) *He was infuriated at Anna's complaining about Peter. [he = Peter]
(1.110) *He isn't bothered by the possibility that Fred will be unpopular. [he = Fred]



P *precedes* NP
P *commands* NP

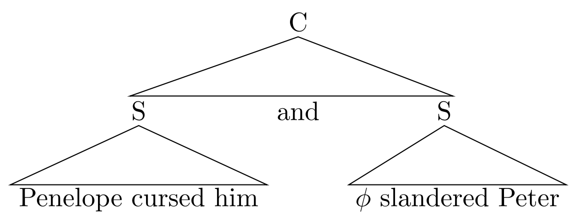
Examples (1.111) and (1.112) from Langacker [19] illustrate the Precedes and Commands Rule for **conjoined structures**.

(1.111) Penelope cursed Peter and slandered him. [him = Peter]



Peter *precedes* him
Peter *is separate from* him
him *is separate from* Peter

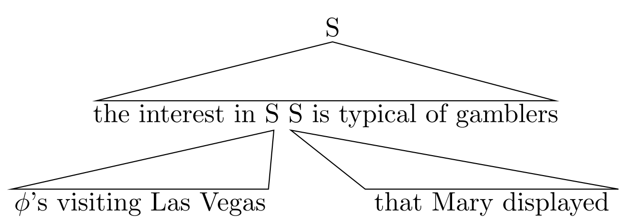
(1.112) *Penelope cursed him and slandered Peter. [him = Peter]



him *precedes* Peter
him *is separate from* Peter
Peter *is separate from* him

Examples (1.113) and (1.114) adapted from Chiba [4] involve Equi-NP Deletion.

(1.113) The interest in visiting Las Vegas that Mary displayed is typical of gamblers.



ϕ *precedes* Mary

2 Resolution Module

2.1 Introduction

In the previous chapter we touched upon some basic notions such as the *precedes*, *commands*, and *is separate from* relations. We will see in the coming chapters how these concepts give rise to a very promising approach to the problem of pronoun resolution.

The algorithm we shall describe won't be complete in the sense that we will elaborate and refine it in later chapters and after we are done it will need elaboration and refinement, but it will be set in firm soil so that we have a foundation on which to build. Because personal and reflexive pronouns are easiest, these are the pronouns we shall consider first. But before we go any farther, let us take time out to indicate something of the environment and structure of the module that does resolving of pronouns in a natural language system, the Resolution module.

2.2 Environment

The center of a natural language system is the Language Processor module which is divided into five submodules. These are the Language Driver, Preprocesor, Parser, Semantic Processor, and Output Processor as indicated in Figure 2.1.

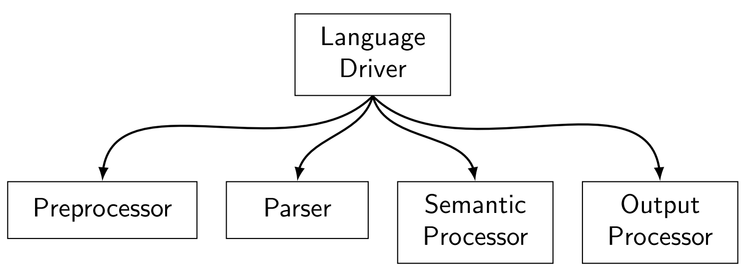


Figure 2.1. Submodules of the Language Processor

Briefly, from the point of view of the Language Processor, the following happens. A user types input at a terminal which is picked up by the Operating System of the natural language system. The Operating System maintains information about the user including the language version he is in as well as his state in that version. The user's state is known as his prefix. The Operating System, after picking up a user's input calls a Process Input routine of the Language Driver in the Language Processor. Once in the Language Driver, the first module to be called upon is the Preprocessor.

The Preprocessor in the Language Processor compresses blanks in the input string, straps right and left delimiters about it, recognizes and builds parsing graph arcs over identifiers and numbers, and looks the identifiers up in the lexicon. After calling the Preprocessor, the Language Driver calls the Parser.

The Parser in the Language Processor parses the output. of the Preprocessor using an algorithm such as the **Kay algorithm** and can handle any general rewrite rule grammar. Of course, since a sentence may be ambiguous, more than one system parse tree may be passed back by the Parser. If no good parsings are found, then the Syntax Diagnostics routine of the Syntax Diagnostics module of the natural language system is called. Otherwise, if there are good parsings, then the Language Driver calls the Semantic Processor on the output of the Parser.

The Semantic Processor is driven by the syntax of a system parse tree into making calls on semantic routines which can be postprocedures (called on their arguments after their arguments evaluate themselves), preprocedures (called on their arguments before their arguments evaluate themselves), and syntax procedures (called at syntax time during parsing before preprocedures and postprocedures are called during semantic processing). On return to the Language Driver, the Language Driver calls the Output Processor on the output of the Semantic Processor.

The Output Processor does some relatively menial processing such as removing duplicate lines from the output line list which will be sent back to the Operating System. The Output Processor is able to handle ambiguous output and removes diagnostic messages if at least one of the outputs is good.

On completion of the call on the Output Processor, the Language Driver returns to the Operating System and the Operating System displays the output line list on the user's terminal, at the same time updating its information on the user.

From the discussion of the *precedes*, *commands*, and *is separate from* relations in the previous chapter, we know that information about the syntax of the input sentence is critical to the resolving of pronouns in the input sentence. On the other hand, for semantic processing to carry out the processing it needs to carry out, the placing of information on the chaining of pronouns must already be placed in the system parse tree of the input sentence.

The logical conclusion of these two observations indicates that pronoun resolution takes place after parsing, but before semantic processing. This relationship of the Resolution module with the other modules of the Language Processor is indicated in Figure 2.2.

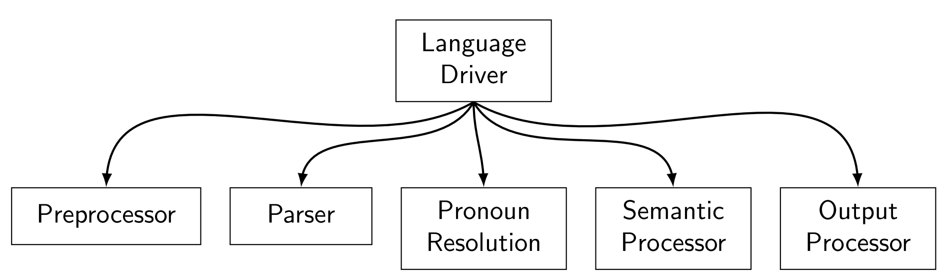


Figure 2.2. Resolution Module within the Language Processor

In practice, this formulation may not be quite correct because there can be other versions than English which will have nothing to do with the Pronoun Resolution module and so what we end up doing is making the Resolution module accessible via a semantic preprocedure which is associated with the parsing of the right delimiter of a sentence. So instead, what happens is that the first semantic preprocedure to be called will be the procedure which handles Pronoun Resolution.

2.3 Structure inside the Resolution Module

The Resolution module is partitioned into seven submodules besides a Global Declarations module. These are the Node Processor, Parser, Primary Utilities, Secondary Utilities, Table Processor, Table Interpreter, and Resolution Driver modules. The reader should not confuse the Parser of the Language Processor with the Parser of the Pronoun Resolution module which have entirely different functions. The relationship of these submodules of the Resolution module is indicated below in Figure 2.3.

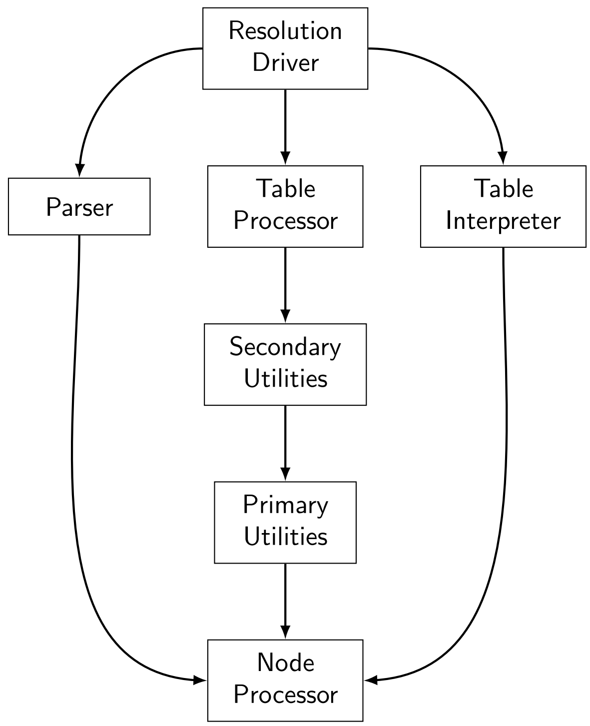


Figure 2.3. Structure of the Resolution Module

Not shown is the Global Declarations module which does not have any procedures itself, but merely defines data structures. The Global Declarations submodule is accessible by all other submodules of the Resolution module.

3 Global Declarations

The Global Declarations module defines the data structures accessible to other modules within the pronoun resolution module. The Global Declarations module is shown below in Figure 3.1.

```
unit globals;
    {Global Declarations Module}
    define
    const PNF=1; {Pronoun Feature}
          FPF=2; {First Person Feature}
          SPF=3; {Second Person Feature}
          TPF=4; {Third Person Feature}
          PLF=5; {Plural Feature}
          GNF=6; {Gender Feature}
          ANF=7; {Animate Feature}
          RPF=8; {Reflexive Feature}
          N_FEATURES=8; {Number of Features}
    type NodeId=(C_NODE,S_NODE,N_NODE,E_NODE); {Kinds of
                                                Nodes}
    Feature=(PLUS,MINUS,QUESTION); {Kinds of
                                    Features}
    Features=packed array[1..N_FEATURES] of
                                        Feature;
    StringPointer=^string;
    NodePointer=^node;
    Node=record
             number:integer;
             up_link,down_link,left_link,right_link,
             thread_link,np_link,chain_link,
             col_link:NodePointer;
             ftr:Features;
             case id:NodeId of
                 C_NODE,S_NODE:();
                 N_NODE:(lit:StringPointer;
                        end_col_link,pred_link,
                        succ_link:NodePointer);
                 E_NODE:(sub:char);
    end{node};
    { Node Fields According to Kind of Node
      (C,up,down,left,right,thread,number)
      (S,up,down,left,right,thread,number)
      (N,lit,ftr,up,down,left,right,thread,np,chain,col,
       end_col,pred,succ,number)
      (E,sub,ftr,np,chain,col)
    }
implement
begin
end{globals};
```

Figure 3.1. Global Declarations Module

Basically, our data structures are C-S-N trees, chaining tables, and the nodes they involve. It will help to get some feel for these data structures before we go on to other chapters.

3.1 Nodes

There are four kinds of **nodes**: C-nodes, S-nodes, N-nodes, and E-nodes. C-nodes, S-nodes, and N-nodes occur in C-S-N trees and correspond to conjoined structures, sentences, and noun phrases. E-nodes occur in chaining tables. The fields of the C-nodes, S-nodes, N-nodes, and E-nodes are as indicated in Figure 3.1.

3.2 C-S-N Trees

A **C-S-N tree** has three kinds of nodes: C-nodes, S-nodes, and N-nodes. Link fields which are relevant to C-S-N trees are `up_link`, `down_link`, `left_link`, `right_link`, `thread_link`, `pred_link`, and `succ_link`. An example of a C-S-N tree is given in Figure 3.2.

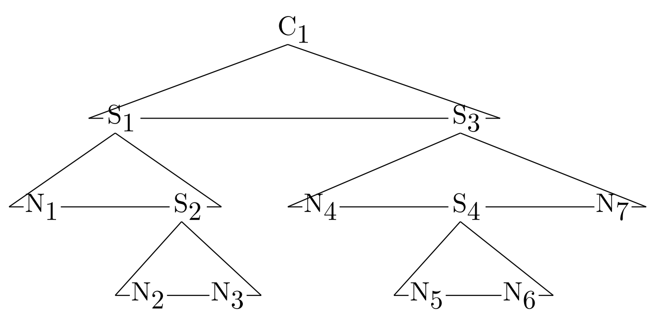


Figure 3.2. C-S-N Tree

3.3 Chaining Tables

A **chaining table** contains N-nodes, E-nodes, and one S-node for keeping track of the chaining table. Link fields relevant to chaining tables are `np_link`, `chain_link`, `col_link`, `end_col_link`, `pred_link`, and `succ_link`. Chaining tables and C-S-N trees are connected through their N-nodes. An example of a chaining table is given in Figure 3.3.

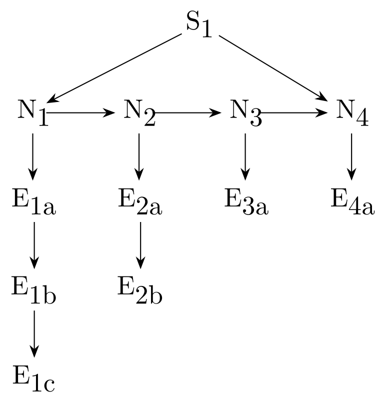


Figure 3.3. Chaining Table

3.4 C-Nodes

A **C-node** has the following fields: `up_link`, `down_link`, `left_link`, `right_link`, `thread_link`, and `number`. C-nodes correspond to conjoined sentences and conjoined subordinate clauses.

3.5 S-Nodes

An **S-node** has exactly the same fields as a C-node and is only distinguished from a C-node by its `NodeId`. S-nodes correspond to sentences and subordinate clauses.

3.6 N-Nodes

An **N-node** has the following fields: `lit`, `ftr`, `up_link`, `down_link`, `thread_link`, `np_link`, `chain_link`, `col_link`, `end_col_link`, `pred_link`, `succ_link`, and `number`. N-nodes correspond to noun phrases without attached subordinate clause modifiers.

3.7 E-Nodes

An **E-node** has the following fields: `sub`, `ftr`, `np_link`, `chain_link`, and `col_link`. An E-node may be thought of as a copy of its `np_link` with a slightly more defined set of features.

3.8 `lit` Field

The `lit` field of an N-node is a string pointer to the string that the N-node represents. The `lit` field is actually unnecessary in an N-node, but is convenient for displaying intermediate results. Procedure `view_Node_str` of the Node Processor and some other procedures that display intermediate results use this field.

3.9 sub Field

The sub field of an E-node is a character representing the subscript of the E-node. The sub field of an E-node, like the lit field of an N-node, is an unnecessary field, but is convenient for displaying intermediate results.

3.10 ftr Field

The ftr field of an N-node or E-node is an array of Feature's representing the feature set of the N-node or E-node to which it corresponds. A Feature can be a PLUS, MINUS, or QUESTION as described in the previous chapter. The offsets PNF, FPF, SPF, TPF, PLF, GNF, ANF, and RPF are used to access elements of the ftr array. The accessed elements are pronoun feature, first person feature, second person feature, third person feature, plural feature, gender feature, animate feature, and reflexive feature. The number of Feature's is N_FEATURES. Figure 3.4 shows some examples of the settings of ftr for some typical noun phrases.

	Features							
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
John	−	−	−	+	−	−	+	−
flowers	−	−	−	+	+	?	−	−
he	+	−	−	+	−	−	+	−
them	+	−	−	+	+	?	?	−
I	+	+	−	−	−	?	+	−
you	+	−	+	−	−	?	+	−
her	+	−	−	+	−	+	+	−
myself	+	+	−	−	−	?	+	+
herself	+	−	−	+	−	+	+	+
itself	+	−	−	+	−	?	−	+

Figure 3.4. ftr Settings for Some Typical Noun Phrases

3.11 up_link Field

The up_link field of a C-node, S-node, or N-node links to the parent node of the C-node, S-node, or N-node in the C-S-N tree in which it occurs. An example of a C-S-N tree with up_link's shown is given in Figure 3.5.

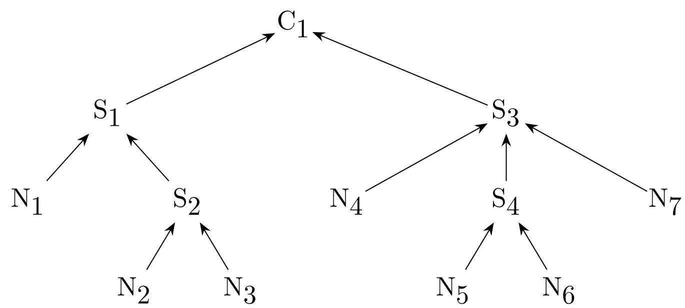


Figure 3.5. C-S-N Tree with `up_link`'s Shown

3.12 `down_link` Field

The `down_link` field of a C-node, S-node, or N-node links to the first child node of the C-node, S-node, or N-node in the C-S-N tree in which it occurs. An example of a C-S-N tree with `down_link`'s shown is given in Figure 3.6.

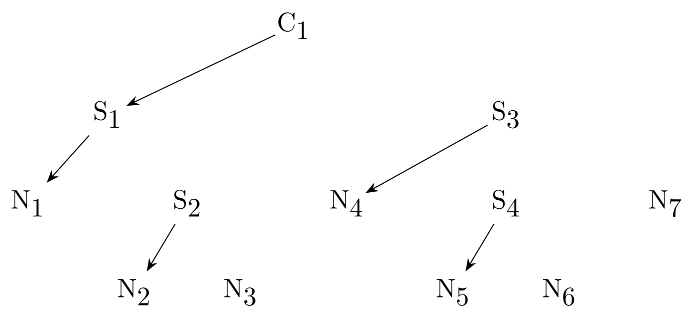


Figure 3.6. C-S-N Tree with `down_link`'s Shown

3.13 `left_link` Field

The `left_link` field of a C-node, S-node, or N-node links to the left brother node of the C-node, S-node, or N-node in the C-S-N tree in which it occurs. An example of a C-S-N tree with `left_link`'s shown is given in Figure 3.7.

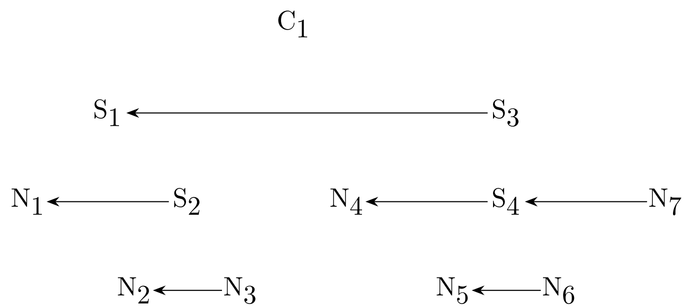


Figure 3.7. C-S-N Tree with `left_link`'s Shown

3.14 `right_link` Field

The `right_link` field of a C-node, S-node, or N-node links to the right brother node of the C-node, S-node, or N-node in the C-S-N tree in which it occurs. An example of a C-S-N tree with `right_link`'s shown is given in Figure 3.8.

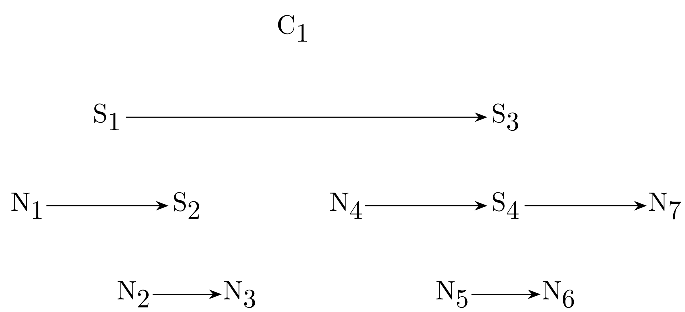


Figure 3.8. C-S-N Tree with `right_link`'s Shown

3.15 `thread_link` Field

The `thread_link` field of a C-node, S-node, or N-node links to the first node traversed after the C-node, S-node, or N-node in a preorder traversal of the C-S-N tree in which it occurs. An example of a C-S-N tree with `thread_link`'s shown is given in Figure 3.9.

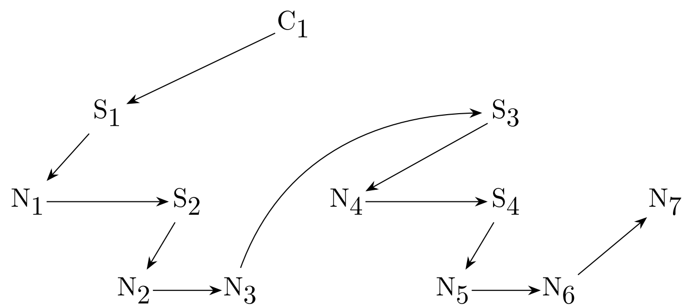


Figure 3.9. C-S-N Tree with `thread_link`'s Shown

3.16 `number` Field

C-node, S-node, or N-node have a `number` field which is the number that would be assigned to that node if the nodes of the C-S-N tree in which it occurs are numbered in a preorder traversal. An example of a C-S-N tree with `number` fields shown is given in Figure 3.10.

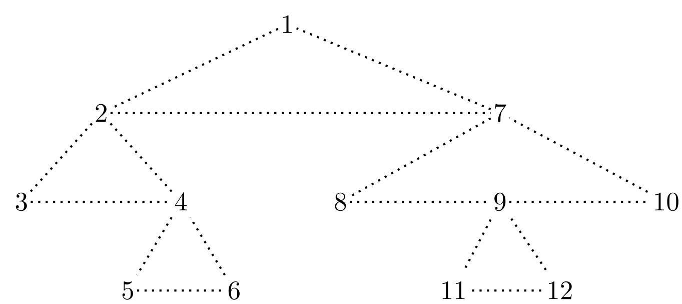


Figure 3.10. C-S-N Tree with `number` Fields Shown

3.17 `np_link` Field

For an E-node, the `np_link` is the N-node to which it is attached. Conceptually, we think of the E-node as being a copy of the N-node except for its subscript and different set of `Feature`'s, `chain_link`, and `col_link`. The `np_link` is just a way of avoiding duplication of information. For an N-node, the `np_link` is always itself. An example of a chaining table with `np_link`'s shown is given in Figure 3.11.

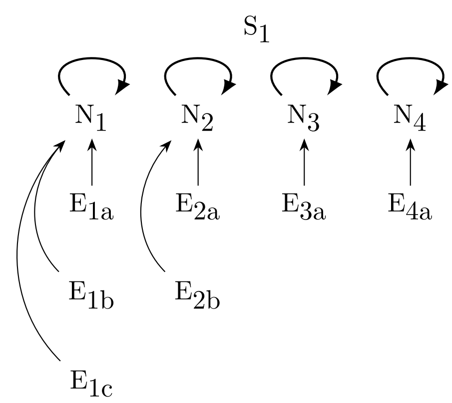


Figure 3.11. Chaining Table with up_link's Shown

3.18 chain_link Field

The `chain_link` of an E-node is another E-node representing the substitute to which the first E-node is attached. When chaining is obligatory, an N-node is chained to an N-node. An example of a chaining table with `chain_link`'s shown is given in Figure 3.12.

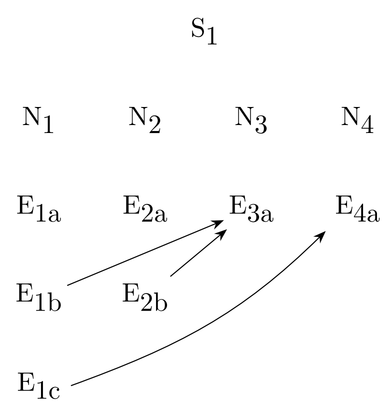


Figure 3.12. Chaining Table with chain_link's Shown

3.19 col_link Field

The `col_link` field of an E-node or N-node links together the elements of a column in a table. An N-node is always on top of a column with E-nodes underneath. An example of a chaining table with `col_link`'s shown is given in Figure 3.13.

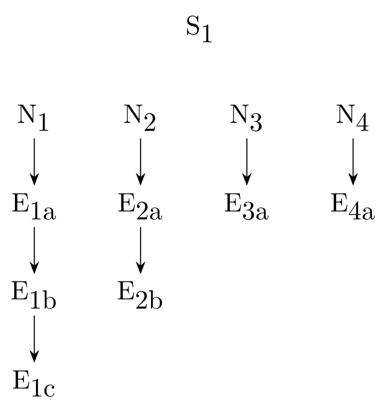


Figure 3.13. Chaining Table with `col_link`'s Shown

3.20 `end_col_link` Field

The `end_col_link` field of an N-node links to the end of the column of E-nodes lying under this N-node. An example of a chaining table with `end_col_link`'s shown is given in Figure 3.14.

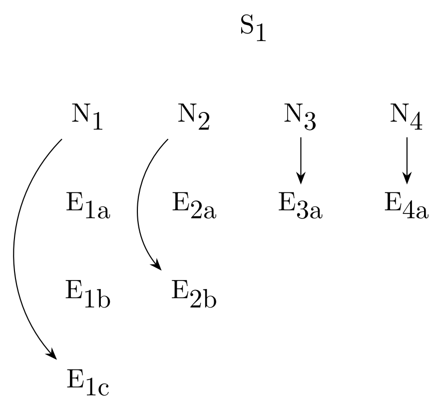


Figure 3.14. Chaining Table with `end_col_link`'s Shown

3.21 `pred_link` Field

The `pred_link` field of an N-node links to the preceding N-node found in a preorder traversal of the C-S-N tree in which it occurs. An example of a C-S-N tree with `pred_link`'s shown is given in Figure 3.15.

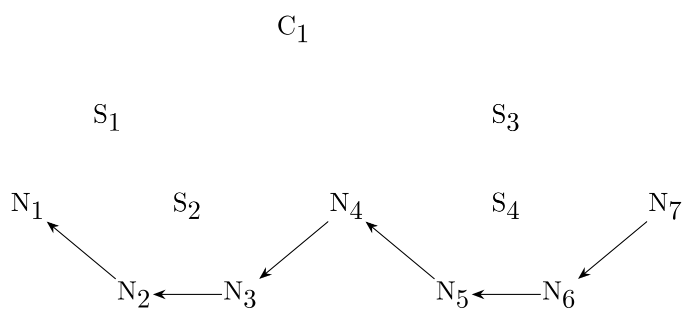


Figure 3.15. C-S-N Tree with `pred_link`'s Shown

An example of a chaining table with `pred_link`'s shown is given in Figure 3.16

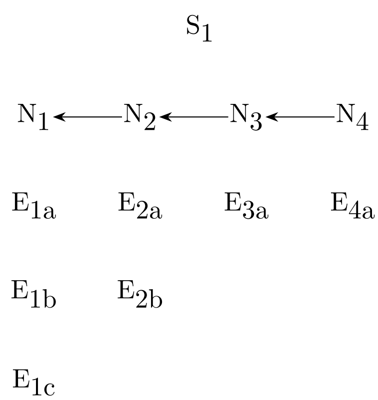


Figure 3.16. Chaining Table with `pred_link`'s Shown

3.22 `succ_link` Field

The `succ_link` field of an N-node links to the succeeding N-node found in a preorder traversal of the C-S-N tree in which it occurs. An example of a C-S-N tree with `succ_link`'s shown is given in Figure 3.17.

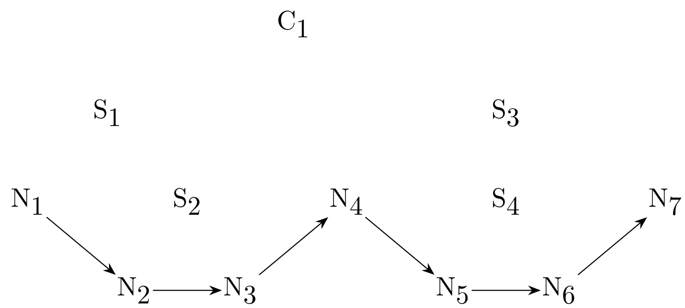


Figure 3.17. C-S-N Tree with `succ_link`'s Shown

An example of a chaining table with `succ_link`'s shown is given in Figure 3.18.

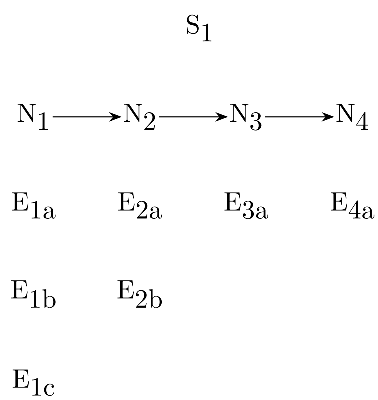


Figure 3.18. Chaining Table with `succ_link`'s Shown

4 Node Processor

The Node Processor module contains functions `new_c_node`, `new_s_node`, `new_n_node`, and `new_e_node` and has the skeleton shown below in Figure 4.1.

```
module node_proc;
    use globals;
    define function new_c_node:NodePointer;
           function new_s_node:NodePointer;
           function new_n_node:NodePointer;
           function new_e_node:NodePointer;
    implement
        (*$I NODE PROC*)
    end;
```

Figure 4.1. Skeleton of the Node Processor

new_c_node, new_s_node, new_n_node, and new_e_node generate, respectively, a new C-node, S-node, N-node, or E-node, with their fields initialized and are rather straightforward functions. These are shown below in Figures 4.2-4.5.

Function new_c_node returns a new C-node.

```
function new_c_node:NodePointer;
    var answer:NodePointer;
begin
    new(answer);
    with answer^ do begin
        id:=C_NODE;
        up_link:=nil;
        down_link:=nil;
        left_link:=nil;
        right_link:=nil;
        thread_link:=nil;
        number:=0;
    end;
    new_c_node:=answer;
end;
```

Figure 4.2. Function new_c_node

Function new_s_node returns a new S-node.

```
function new_s_node:NodePointer;
    var answer:NodePointer;
begin
    answer:=new_c_node;
    answer^.id:=S_NODE;
    new_s_node:=answer;
end;
```

Figure 4.3. Function new_s_node

Function new_n_node returns a new N-node.

```
function new_n_node:NodePointer;
    var answer:NodePointer;
        i:integer;
begin
    new(answer);
    with answer^ do begin
        id:=N_NODE:
        lit^:='':
        for i:=1 to N_FEATURES do ftr[i]:=QUESTION;
        up_link:=nil;
        down_link:=nil;
        left_link:=nil;
        right link:=nil;
        thread_link:=nil;
        np_link:=nil;
        chain_link:=nil;
        col_link:=nil;
        end_col_link:=nil;
        pred_link:=nil;
        succ_link:=nil;
        number:=0;
    end;
    new_n_node:=answer;
end;
```

Figure 4.4. Function new_n_node

Function new_e_node returns a new E-node.

```
function new_e_node:NodePointer;
    var answer:NodePointer;
        i:integer;
begin
    new(answer);
    with answer^ do begin
        id:=E_NODE:
        sub:=' ':
        for i:=1 to N_FEATURES do ftr[i]:=QUESTION;
        np_link:=nil;
        chain_link:=nil;
        col_link:=nil;
    end;
    new_e_node:=answer;
end;
```

Figure 4.5. Function `new_e_node`

4.1 Procedure `view_Node_str`

There is one output procedure in the Node Processor that has not been discussed above that we need to know about, because we will be looking at some of its output for short while. This is procedure `view_Node_str` which takes as an argument a `NodePointer` and outputs it in readable form. Otherwise, procedure `view_Node_str` does no processing of its own, and so we do not need to know the details of its inner workings. For us it is enough to be able to understand the output. Procedure `view_Node_str` has the form indicated in Figure 4.6.

```
procedure view_Node_str(n:NodePointer);
begin
    {output n in a readable form}
end;
```

Figure 4.6. Skeleton of Procedure `view_Node_str`

Some typical output of procedure `view_Node_str` is shown below in Figure 4.7 where a chaining table is listed. (Links from the chaining table to its associated C-S-N tree are also listed by procedure `view_Node_str`.)

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:5, th:3, nu:2)
3	(N, lit:June, ftr:[---+-++-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_b, pr:0, su:4, nu:3)
3_a	(E, sub:A, ftr:[---+-++-], np:3, ch:0, co:3_b)
3_b	(E, sub:B, ftr:[---+-++-], np:3, ch:6_a, co:0)
4	(N, lit:flowers, ftr:[---++?--], up:2, dn:0, lt:3, rt:0, th:5, np:4, ch:0, co:4_a, ec:4_b, pr:3, su:6, nu:4)
4_a	(E, sub:A, ftr:[---++?--], np:4, ch:0, co:4_b)
4_b	(E, sub:B, ftr:[---++?--], np:4, ch:7_a, co:0)
5	(S, up:1, dn:6, lt:2, rt:0, th:6, nu:5)
6	(N, lit:she, ftr:[+--+-++-], up:5, dn:0, lt:0, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_a, pr:4, su:7, nu:6)
6_a	(E, sub:A, ftr:[+--+-++-], np:6, ch:0, co:0)
7	(N, lit:them, ftr:[+--++??-], up:5, dn:0, lt:6, rt:0, th:0, np:7, ch:0, co:7_a, ec:7_a, pr:6, su:0, nu:7)
7_a	(E, sub:A, ftr:[+--++??-], np:7, ch:0, co:0)

Figure 4.7. Typical Output from Procedure `view_Node_str`

(`C` = C-node, `S` = S-node, `N` = N-node, `E` = E-node, `lit` = `lit` field, `sub` = `sub` field, `ftr` = `ftr` field, `up` = `up_link`, `dn` = `down_link`, `lt` = `left_link`, `rt` = `right_link`, `th` = `thread_link`, `nu` = `number`, `np` = `np_link`, `ch` = `chain_link`, `co` = `col_link`, `ec` = `end_col_link`, `pr` = `pred_link`, and `su` = `succ_link`)

5 Parser

The Parser module defines function `parse` and has the form shown below in Figure 5.1.

```
module parser;
    use globals,node_proc:
    type
        ParseObject=^record
            case isString:boolean of
                true:(strValue:string);
                false:(listValue:array of ParseObject);
    define function parse(obj:ParseObject):NodePointer;
    implement
        (*$I PARSER*)
    end;
```

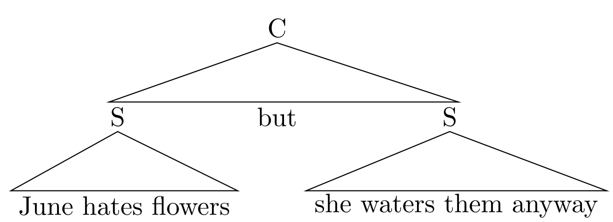
Figure 5.1. Skeleton of the Parser

Function `parse` accepts as input a **system focus** representation and **system parse tree** that has been generated by a computer natural language system. The output of `parse` is a C-S-N tree incorporating the information contained in the system parse tree and system focus. The system focus represents the natural language system's focus of attention. This will be gone into in more detail in Chapter 13.

The representation of the system tree inputted to `parse` is system dependent, and so the details of `parse` are also system dependent. As the internals of `parse` are heavily dependent upon and rather involved for any system, we won't go into the details of `parse` for any particular system here. Hopefully, the reader may glean enough information from the multitude of examples presented in this thesis to get an idea of what `parse` does. In any case, lack of an actual algorithm for `parse` isn't so bad since the ideas presented in this thesis are really still in an early stage and it is enough to concentrate on them.

Even though the input to the Parser is not well defined, the output is. The Parser builds from the system parse tree it is given the corresponding C-S-N tree with all `up_link`'s, `down_link`'s, `left_link`'s, `right_link`'s, `thread_link`'s, and `number`'s set to what is expected. Consider example (5.2) below.

(5.2) June hates flowers, but she waters them anyway.



When procedure parse is called on the system parse tree representing (5.2), we get the following output in Figure 5.3.

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
June	−	−	−	+	−	+	+	−
flowers	−	−	−	+	+	?	−	−
she	+	−	−	+	−	+	+	−
them	+	−	−	+	+	?	?	−

Nodes	
1	`(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)`
2	`(S, up:1, dn:3, lt:0, rt:5, th:3, nu:2)`

3	`(N, lit:June, ftr:[---+-++-], up:2, dn:0,`
	`lt:0, rt:4, th:4, np:3, ch:0, co:`3_a`, ec:`3_b`,`
	`pr:0, su:4, nu:3)`

4	`(N, lit:flowers, ftr:[---++?--], up:2, dn:0,`
	`lt:3, rt:0, th:5, np:4, ch:0, co:`4_a`, ec:`4_b`,`
	`pr:3, su:6, nu:4)`

5	`(S, up:1, dn:6, lt:2, rt:0, th:6, nu:5)`

6	`(N, lit:she, ftr:[+--+-++-], up:5, dn:0,`
	`lt:0, rt:7, th:7, np:6, ch:0, co:`6_a`, ec:`6_a`,`
	`pr:4, su:7, nu:6)`

7	`(N, lit:them, ftr:[+--++??-], up:5, dn:0,`
	`lt:6, rt:0, th:0, np:7, ch:0, co:`7_a`, ec:`7_a`,`
	`pr:6, su:0, nu:7)`

Figure 5.3. Typical Output from `parse`

Listing of nodes in Figure 5.3 is done by procedure `view_Node_str` of the Node Processor described in Chapter 4. The C-S-N parse tree is slightly more complicated when focusing is taken into account, but for the time being we will ignore its effects. We will discuss the effects of focusing on C-S-N parse trees in Chapter 13.

When Figure 5.3 is drawn as a tree, we get a structure like Figure 5.4.

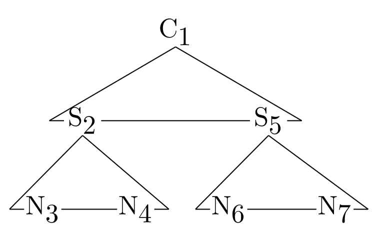


Figure 5.4. Output from `parse` Drawn as a Tree

6 Primary Utilities

The Primary Utilities module defines the boolean functions `precede`, `command`, and `separate` corresponding to the *precedes*, *commands*, and *is separate from* relations discussed in Chapter 1. The skeleton of the primary utilitites module is shown below in Figure 6.1.

```
module primary_uty;
    use globals;
    define function precede(n1,n2:NodePointer)
                                            :boolean;
           function command(n1,n2:NodePointer)
                                            :boolean;
           function separate(n1,n2:NodePointer)
                                            :boolean;
    implement
        (*$I PRIMARY UTY*)
    end;
```

Figure 6.1. Skeleton of the Primary Utilities

The `precede`, `command`, and `separate` functions do just what is expected. They are `true` if and only if the *precedes*, *commands*, and *is separate from* relations hold between their arguments. Along with function `dominate` which is used by `separate`, these functions are shown below in Figures 6.2-6.5.

Function `precede` is `true` if and only if `n1` *precedes* `n2`.

```
function precede(n1,n2:NodePointer):boolean;
begin
    if n1^.id=E_NODE then n1:=n1^.np_link;
    if n2^.id=E_NODE then n2:=n2^.np_link;
    precede:=n1^.number<n2^.number;
end;
```

Figure 6.2. Function `precede`

Function `dominate` is `true` if and only if `n1` *dominates* `n2`.

```
function dominate(n1,n2:NodePointer):boolean;
    label 100;
    var child:NodePointer;
begin
    if n1^.id=E_NODE then n1:=n1^.np_link;
    if n2^.id=E_NODE then n2:=n2^.np_link;
    dominate:=true;
    if n1=n2 then goto 100;
    child:=n1^.down_link;
    while child<>nil do begin
        if dominate(child,n2) then goto 100;
        child:=child^.right_link;
    end;
    dominate:=false;
100:;
end;
```

Figure 6.3. Function `dominate`

Function `command` is `true` if and only if `n1` *commands* `n2`.

```
function command(n1,n2:NodePointer):boolean;
begin
    {minimum cyclic node dominating n1 also dominates n2}
    if n1^.id=E_NODE then n1:=n1^.np_link;
    if n2^.id=E_NODE then n2:=n2^.np_link;
    command:=dominate(n1^.up_link,n2);
end;
```

Figure 6.4. Function `command`

Function `separate` is `true` if and only if `n1` *is separate from* `n2`.

```
function separate(n1,n2:NodePointer):boolean;
    var parent:NodePointer;
begin
    {minimum cyclic node dominating n1 that also dominates
     n2 is a C-node}
    if mi^.id=E_NODE then n1:=n1^.np_link;
    if n2^.id=E_NODE then n2:=n2^.np_link;
    parent:=n1^.up_link;
    while not dominate(parent, n2) do parent:=parent^.up_link
    separate:=parent^.id=C_NODE;
end;
```

Figure 6.5. Function `separate`

7 Secondary Utilities

The Secondary Utiltities module defines functions `sc`, `agr`, and `rnr`. These stand for Syntactic Conditions, Agreement, and the Reflexive Nonreflexive Rule. The skeleton of the Secondary Utilities module is shown below in Figure 7.1.

```
module secondary_uty;
    use globals,primary_uty;
    define function sc(n1,n2:NodePointer):boolean;
           function agr(n1,n2:NodePointer):boolean;
           function rnr(n1,n2:NodePointer):boolean;
    implement
        (*$I SECONDARY UTY*)
    end;
```

Figure 7.1. Skeleton of Secondary Utilities

7.1 Syntactic Conditions

As shown in Chapter 1, certain constraints such as the Precedes and Commands Rule apply in forward pronominalization. Function `sc` is `true` whenever these grosser syntactic constraints are met. In this thesis, we let `sc` be `true` when the Precedes and Commands Rule is satisfied, function `sc` is shown below in Figure 7.2.

```
function sc(n1,n2:NodePointer):boolean;
    {Syntactic Conditions}
begin
    sc:=not (precede(n1,n2) and (command(n1,n2)
                or separate(n1,n2)));
end;
```

Figure 7.2. Function `sc` (Syntactic Conditions)

7.2 Agreement

Besides satisfying Syntactic Conditions, there has to be agreement between a node and its chaining node. First person, second person, third person, plural, gender, and animate features have to agree in order for one node to chain to another. Function `agr` is shown below in Figure 7.3.

```
function agr(n1,n2:NodePointer):boolean;
    {Agreement}
    var ftr1,ftr2:features;
begin
    ftr1:=n1^.ftr;
    ftr2:=n2^.ftr;
    agr:=eq_feat(ftr1[FPF],ftr2[FPF]) and
         eq_feat(ftr1[SPF],ftr2[SPF]) and
         eq_feat(ftr1[TPF],ftr2[TPF]) and
         eq_feat(ftr1[PLF],ftr2[PLF]) and
         eq_feat(ftr1[GNF],ftr2[GNF]) and
         eq_feat(ftr1[ANF],ftr2[ANF]);
end;
```

Figure 7.3. Function `agr` (Agreement)

7.3 Equal Features

Function `agr` uses function `eq_feat`. `eq_feat` tests if two `Feature`'s are equal. As indicated in Chapter 1, `Feature`'s are equal unless a `PLUS` and `MINUS` are compared. Function `eq_feat` is shown below in Figure 7.4.

```
function eq_feat(x1,x2:feature):boolean;
    {Equal Features}
begin
    case x1 of
        PLUS:     eq_feat:=x2<>MINUS;
        MINUS:    eq_feat:=x2<>PLUS;
        QUESTION: eq_feat:=true;
    end;
end;
```

Figure 7.4. Function `eq_feat` (Equal Features)

7.4 Reflexive Nonreflexive Rule

The distinction between reflexive pronouns and nonreflexive pronouns is that reflexive pronouns cannot chain to an N-node that is outside of the same simplex in which it occurs, while a nonreflexive pronoun can. This rule will have to be modified later for genitives, but for now we can suppose that a nonreflexive pronoun must chain to an N-node outside of the same simplex in which it is in. Shown in Figure 7.5 is function `rnr` which is `true` when the reflexive nonreflexive rule is satisfied.

```
function rnr(n1,n2:NodePointer):boolean;
    {Reflexive Nonreflexive Rule}
    var ftr1:features;
begin
    n1:=n1^.np_link;
    n2:=n2^.np_link;
    ftr1:=n1^.ftr;
    case ftr1[RPF] of
        PLUS:      rnr:=n1^.up_link=n2^.up_link;
        QUESTION: {doesn't occur};
        MINUS:     rnr:=n1^.up_link<>n2^.up_link;
    end;
end;
```

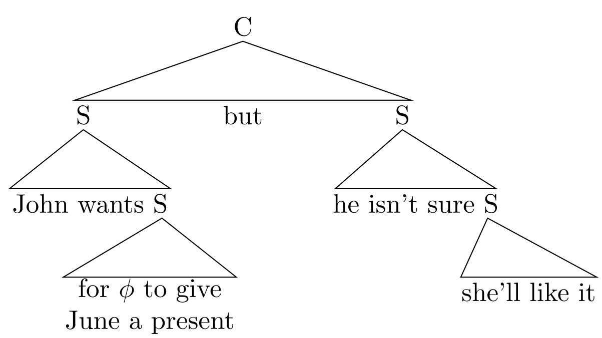
Figure 7.5. Function `rnr` (Reflexive Nonreflexive Rule)

8 Table Processor I

The Table Processor module defines function `chaining` which takes as input a C-S-N tree and returns its chaining table. The actions of function `chaining` in the Table Processor can only be understood by example, and this is what this chapter provides. In Chapter 9, we'll look at the actual algorithms and, in Chapter 10, we'll look at some actual output.

So, let us consider sentence (8.1) below.

(8.1) John wants to give June a present, but he isn't sure she'll like it.



The Parser builds from the system parse tree of (8.1) the corresponding C-S-N tree with six N-nodes which have the `lit` fields and `ftr`'s indicated below in Figure 8.2.

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
John	−	−	−	+	−	−	+	−
ϕ	+	?	?	?	?	?	?	−
June	−	−	−	+	−	+	+	−
present	−	−	−	+	−	?	−	−
he	+	−	−	+	−	−	+	−
she	+	−	−	+	−	+	+	−
it	+	−	−	+	−	?	−	−

Figure 8.2. `lit` Fields and `ftr`'s of the N-Nodes

The C-S-N tree itself has the form of Figure 8.3 below.

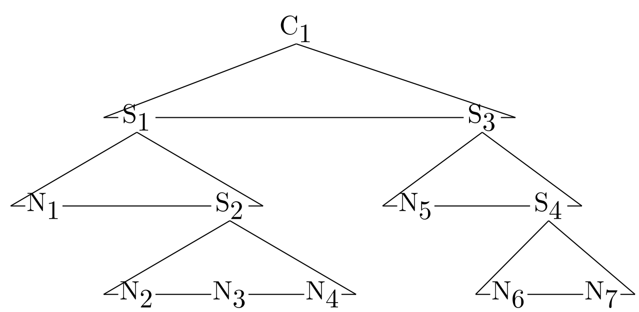


Figure 8.3. C-S-N Parse Tree

After `parse` is called, `chaining` is called. The first thing to happen is the initialization of the chaining table for the C-S-N tree. Below each N-node is suspended, by the `col_link` of the N-node, a new E-node with subscript `A`. Each new E-node has a `np_link` back to the N-node it is suspended from. As well, the `Feature`'s of each new E-node are copied from the N-node it is suspended from. Attached to the first and last N-nodes is an S-node to make it easy to keep track of the first and last N-nodes in the chaining table. The chaining table, as it looks immediately after initialization, is shown below in Figure 8.4.

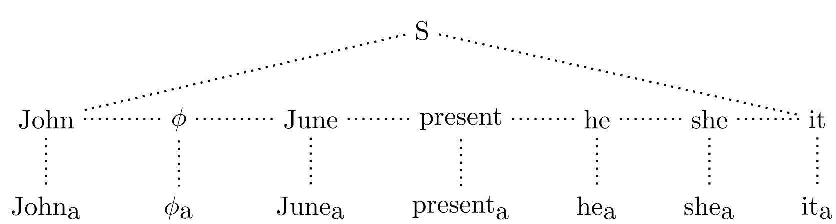


Figure 8.4. Chaining Table Immediately after Initialization

The `chaining` algorithm works by walking backwards across N-nodes in the top row and walking down columns of E-nodes. The `chaining` algorithm works on two N-nodes at a time. If the first is compatible with the second under Syntactic Conditions, Agreement, and the Reflexive Nonreflexive Rule, then the E-nodes underneath the first N-node that agree with the second N-node are `chain_link`'ed to copies of the second N-node.

The last N-node in the table is it, the chaining table begins with it. it can't chain to itself, so the second N-node in the description above becomes she and the `chaining` algorithm compares it to she. Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(it, she) = true
agr(it, she) = false
```

The `chaining` algorithm now moves from she to he and compares it to he. Again Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(it, he) = true
agr(it, he) = false
```

The `chaining` algorithm moves from he to present and compares it to present. This time, Syntactic Conditions, Agreement, and the Reflexive Nonreflexive Rule are satisfied.

```
sc(it, present) = true
agr(it, present) = true
rnr(it, present) = true
```

Since all three rules are satisfied, a chain from it_a to a copy of present may be created. This happens if it_a and present agree, and they do.

```
agr(it_a, present) = true
```

The `chaining` algorithm makes a new E-node copy of present, present_b, and hangs it below present. The `chain_link` of present_b is set to it_a and the semantic features of it_a, but not the syntactic features, are copied into the semantic features of present_b. After chaining present_b to it_a, the chaining table appears as shown in Figure 8.5.

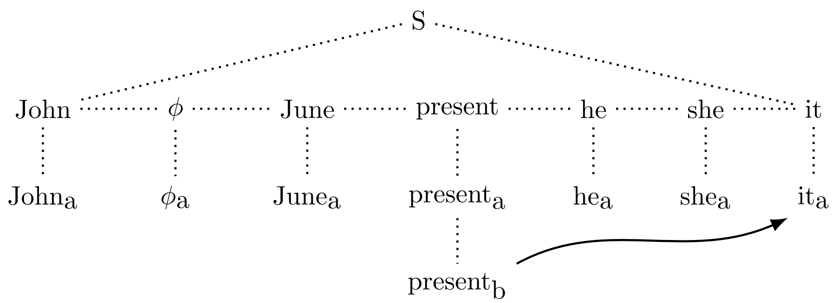


Figure 8.5. Chaining Table after Chaining present$_b$ to it$_a$

The `chaining` algorithm now moves from present to June. Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(it, June) = true
agr(it, June) = false
```

The `chaining` algorithm now moves from June to ϕ. This time, all three rules, Syntactic Conditions, Agreement, and the Reflexive Nonreflexive Rule are satisfied.

```
sc(it, φ) = true
agr(it, φ) = true
rnr(it, φ) = true
```

Since all three rules are satisfied, E-nodes under it that agree with ϕ chain to copies of ϕ. it$_a$ is compared to ϕ, and it is seen that they agree.

```
agr(it_a, φ) = true
```

The `chaining` algorithm makes a new E-node copy of ϕ, ϕ_b, and hangs it below ϕ. The `chain_link` of ϕ_b is set to it$_a$ and the semantic features of it$_a$ are copied into the semantic features of ϕ_b. After chaining ϕ_b to it$_a$, the chaining table appears as shown in Figure 8.6.

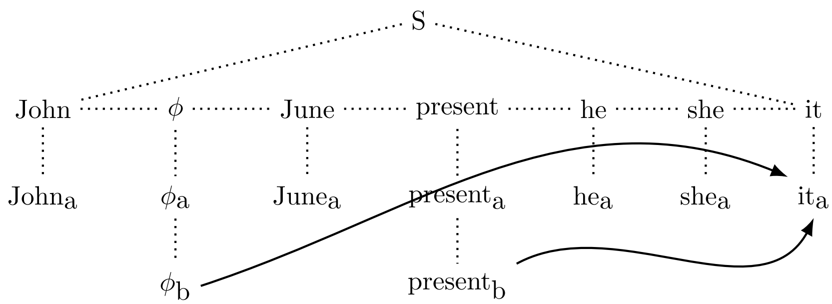


Figure 8.6. Chaining Table After Chaining $\underline{\phi_b}$ to $\underline{it_a}$

The `chaining` algorithm now moves to John and compares John to it. Syntactic Conditions hold, but Agreement does not.

```
sc(it, John) = true
agr(it, John) = false
```

Having exhausted all possible combinations with it, the `chaining` algorithm considers she.

The `chaining` algorithm tries comparing she to it, but Syntactic Conditions are not satisfied.

```
sc(she, it) = false
```

The `chaining` algorithm moves from it to she, but she can't chain to she, so the `chaining` algorithm moves to he. This time Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(she, he) = true
agr(she, he) = false
```

The `chaining` algorithm now moves from he to present where again Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(she, present) = true
agr(she, present) = false
```

The `chaining` algorithm moves from present to June. This time all three rules are satisfied.

```
sc(she, June) = true
agr(she, June) = true
rnr(she, June) = true
```

As all three rules are satisfied, E-nodes under she that agree with June chain to copies of June. she$_a$ is compared to June, and it is seen that they agree.

```
agr(she_a, June) = true
```

The `chaining` algorithm makes a new E-node copy of June, June$_b$, and hangs it below June. The `chain_link` of June$_b$ is set to she$_a$ and the semantic features of she$_a$ are copied into the semantic features of June$_b$. After chaining June$_b$ to she$_a$, the chaining table appears as shown in Figure 8.7.

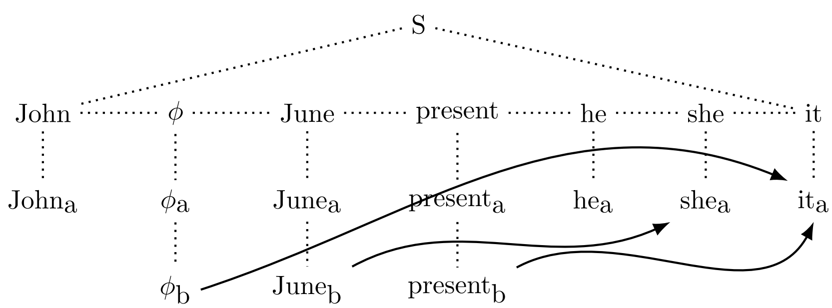


Figure 8.7. Chaining Table after Chaining June$_b$ to she$_a$

The `chaining` algorithm now moves from June to ϕ and compares she to ϕ. All three rules are satisfied.

```
sc(she, φ) = true
agr(she, φ) = true
rnr(she, φ) = true
```

Copies of ϕ are `chain_link`'ed to E-nodes under she that agree with ϕ. she$_a$ is compared to ϕ, and it is seen that they agree.

```
agr(she_a, φ) = true
```

A new E-node copy of ϕ, ϕ_c, is made and hung below ϕ. The `chain_link` of ϕ_c is set to she$_a$ and the semantic features of she$_a$ are copied into the semantic features of ϕ_c. After chaining ϕ_c to she$_a$, the chaining table appears as shown in Figure 8.8.

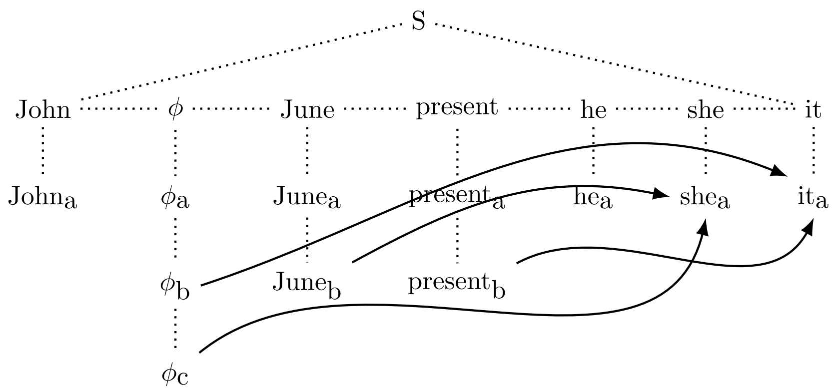


Figure 8.8. Chaining Table after Chaining $\underline{\phi_c}$ to $\underline{\text{she}_a}$

The `chaining` algorithm now moves from $\underline{\phi}$ to John and compares she to John. It is seen that Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(she, John) = true
agr(she, John) = false
```

This completes the creation of `chain_link`'s to E-nodes under she. The `chaining` algorithm now considers he.

he is compared to it, but it is seen that Syntactic Conditions aren't satisfied.

```
sc(he, it) = false
```

The `chaining` algorithm moves from it to she, but again, Syntactic Conditions aren't satisfied.

```
sc(he, she) = false
```

The `chaining` algorithm moves from she to he, but he can't chain to he, so the `chaining` algorithm moves from he to present. This time Syntactic Conditions are satisfied, but Agreement isn't.

```
sc(he, present) = true
agr(he, present) = false
```

The `chaining` algorithm moves from present to June, and similar results happen.

```
sc(he, June) = true
agr(he, June) = false
```

Next, the `chaining` algorithm moves from June to ϕ, and, this time, all three rules, Syntactic Conditions, Agreement, and the Reflexive Nonreflexive Rule, are satisfied.

```
sc(he, ϕ) = true
agr(he, ϕ) = true
rnr(he, ϕ) = true
```

Copies of he are `chain_link`'ed to E-nodes under ϕ that agree with he. he_a is compared to ϕ, and it is seen that they agree.

```
agr(he_a, ϕ) = true
```

A new E-node copy of ϕ, ϕ_d, is made and hung below ϕ. The `chain_link` of ϕ_d is set to he_a and the semantic features of he_a are copied into the semantic features of ϕ_d. After chaining ϕ_d to he_a, the chaining table appears as shown in Figure 8.9.

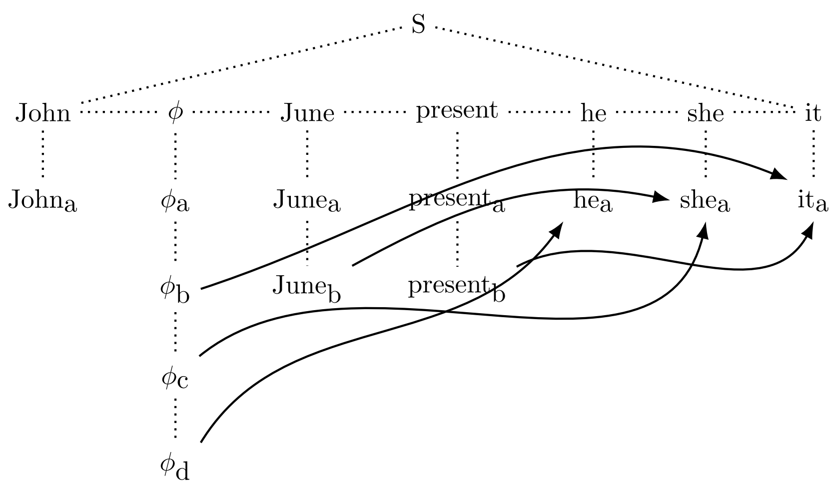


Figure 8.9. Chaining Table after Chaining ϕ_d to he_a

The `chaining` algorithm now moves from ϕ to John and he is compared to John. It is seen that all three rules are satisfied.

```
sc(he, John) = true
agr(he, John) = true
rnr(he, John) = true
```

So, copies of he are `chain_link`'ed to E-nodes under John that agree with he. he_a is compared to John, and it is seen that they agree.

```
agr(heₐ, John) = true
```

A new E-node copy of John, John_b, is made and hung below John. The `chain_link` of John_b is set to he_a and the semantic features of he_a are copied into the semantic features of John_b. After chaining John_b to he_a, the chaining table appears as shown in Figure 8.10.

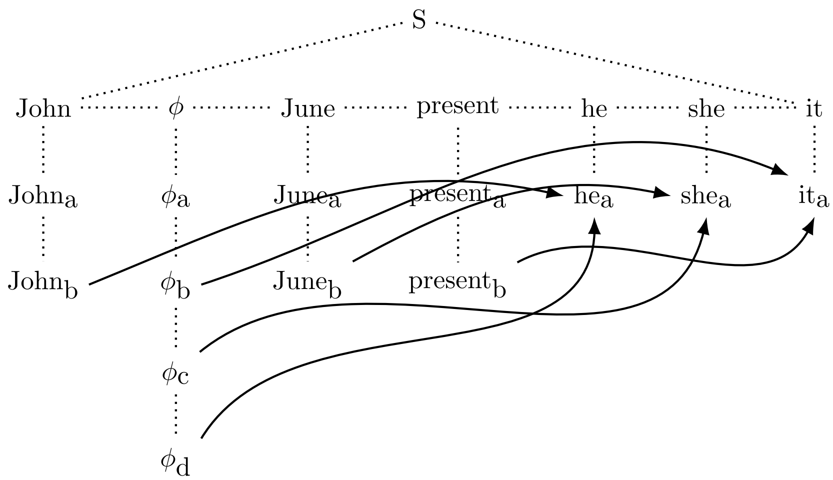


Figure 8.10. Chaining Table after Chaining John_b to he_a

Having completed the processing of he, the `chaining` algorithm considers present. present is not a pronoun though, so the `chaining` algorithm moves on to June. Similarly, June is not a pronoun, so the `chaining` algorithm now considers ϕ.

The `chaining` algorithm compares ϕ to it, and it is seen that Syntactic Conditions don't hold.

```
sc(φ, it) = false
```

The `chaining` algorithm moves from it to she, she to he, he to present, and present to June with little more success.

```
sc(φ, she) = false
sc(φ, he) = false
sc(φ, present) = false
sc(φ, June) = false
```

The `chaining` algorithm moves from June to ϕ, but ϕ can't chain to ϕ. So now, the `chaining` algorithm moves from ϕ to John. This time, all three rules are satisfied.

```
sc(φ, John) = true
agr(φ, John) = true
rnr(φ, John) = true
```

Copies of John are chain_link'ed to E-nodes under ϕ that agree with John. ϕ_a is compared to John, and it is seen that they agree.

```
agr(φa, John) = true
```

Thus, new copy of John, John_c is made. John_c is chain_link'ed to ϕ_a. The semantic features of ϕ_a are copied into John_c. After chaining John_c to ϕ_a, the chaining table appears as shown in Figure 8.11.

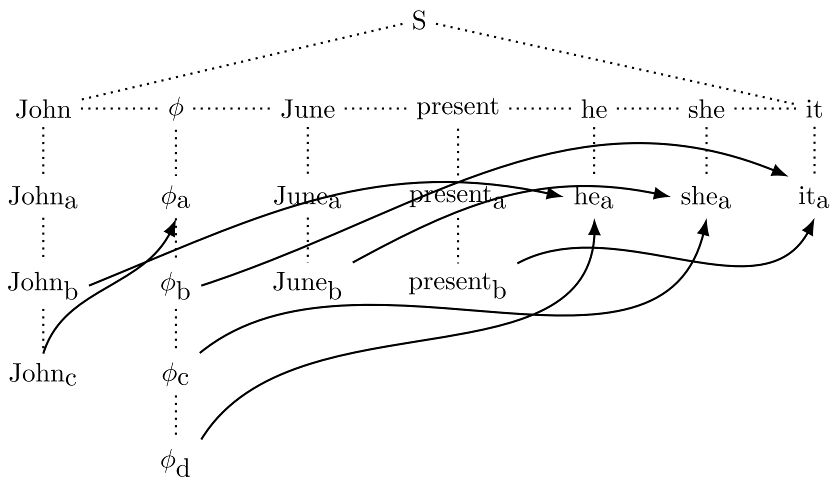


Figure 8.11. Chaining Table after Chaining John_c to ϕ_a

ϕ_b is compared to John, and it is seen that they don't agree.

```
agr(φb, John) = false
```

ϕ_b and John don't agree because when ϕ_b was chain_link'ed to it_a, the semantic features of it_a were copied into the semantic features of ϕ_b. Hence, the information that it_a was inanimate was copied into ϕ_b preventing a ridiculous chain: JohnX is chained to ϕ_b is chained to it_a. ϕ_c, which was chained to she_a, is compared to John, and it is seen that they don't agree.

```
agr(φc, John) = false
```

On the other hand, ϕ_d, which was chained to he_a, does agree with John.

```
agr(φd, John) = true
```

Thus, new copy of John, John_d is made. John_d is `chain_link`'ed to ϕ_d. The semantic features of ϕ_d are copied into the semantic features of John_d. After chaining John_d to ϕ_d, the chaining table appears as shown in Figure 8.12.

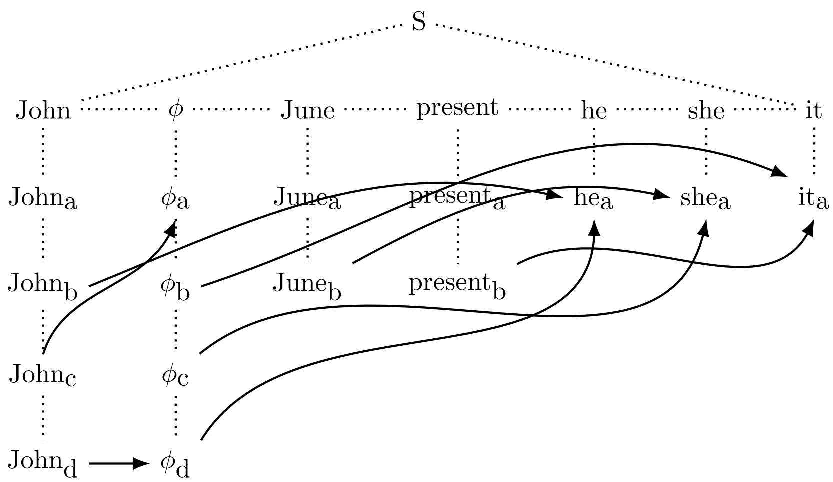


Figure 8.12. Chaining Table after Chaining John_d to ϕ_d

Having completed chaining to ϕ, the `chaining` algorithm moves to John. John is not a pronoun, so the `chaining` algorithm now stops as it has reached the end of the chaining table. This makes Figure 8.12, above, the finished chaining table.

9 Table Processor II

From Chapter 8 we know that the Table Processor module defines function `chaining` which takes as input a C-S-N tree and which returns as output the chaining table of the inputted C-S-N tree. In Chapter 8, we illustrated the kind of processing the Table Processor does by working through in detail a typical example. In this chapter, we will go into the particulars of the Table Processor algorithms. In Chapter 10, we'll look at some actual output.

The skeleton of the Table Processor module is shown below in Figure 9.1. The Table Processor module defines function `chaining`.

```
module table_proc;
    use globals,secondary_uty,node_proc;
    define function chaining(tree:NodePointer)
                                   :NodePointer;
    implement
        (*$I TABLE PROC*)
    end;
```

Figure 9.1. Skeleton of the Table Processor

Function `chaining` is the algorithm we described by example in Chapter 8. `chaining` takes as input a C-S-N tree and returns the chaining table of the inputted C-S-N tree. Function `chaining` is shown below in Figure 9.2.

```
function chaining(tree:NodePointer):NodePointer;
    var n1:NodePointer;
begin
    init_table(tree);
    n1:=table^.right_link;
    while n1<>nil do with n1^ do begin
        if ftr[PNF]=PLUS then chaining_n(n1);
        n1:=pred_link;.
    end;
    chaining:=table;
end;
```

Figure 9.2. Function `chaining`

The first thing function `chaining` does is to call `init_table` which initializes the chaining table as described in the previous chapter. Procedure `init_table` is shown in Figure 9.3.

```
procedure init_table(tree:NodePointer);
    var first,last,n:NodePointer;
        i:integer;
begin
    first:=nil;
    last:=nil;
    n:=tree:
    while n<>nil do with n^ do begin
        if id=N_NODE then begin
            col_link:=new_e_node:
            with col_link^ do begin
                for i:=1 to features do
                ftr[i]:=n^.ftr[i];
                np_link:=n;
                sub:='A';
            end;
            end_col_link:=col_link;
            pred_link:=last;
            if last=nil then first:=n
            else last^.succ_link:=n;
            last:=n;
        end;
        n:=thread_link;
    end;
    table:=new_s_node:
    with table^ do begin
        left_link:=first;
        right_link:=last;
    end;
end;
```

Figure 9.3. Procedure `init_table`

Below each N-node is hung a new E-node with `Feature`'s copied from the N-node and `np_link` back to the N-node. The `col_link`'s and `end_col_link`'s of the N-nodes are updated accordingly. `pred_link`'s and `succ_link`'s are set in `init_table` using the `thread_link`'s which were established by the Parser. Finally, at the end of the procedure, `table`, a variable global inside the Table Processor, has its `left_link` and `right_link` set to the first and last N-node.

For each N-node that is a pronoun, function `chaining` calls procedure `chaining_n`. `chaining_n` calls `refl_chaining` or `non_refl_chaining` depending on whether or not the inputted N-node is reflexive or not. Procedure `chaining_n` is shown below in Figure 9.4.

```
procedure chaining_n(n1:NodePointer);
begin
    case n1^.ftr[RPF] of
        PLUS:     refl_chaining(n1);
        QUESTION: {can't happen};
        MINUS:    non_refl_chaining(n1);
    end;
end;
```

Figure 9.4. Procedure `chaining_n`

Procedure `non_refl_chaining` handles nonreflexive pronouns. Procedure `non_refl_chaining` is shown below in Figure 9.5.

```
procedure non_refl_chaining(n1:NodePointer);
    {Nonreflexive Chaining}
    var n2:NodePointer;
begin
    {try nps other than n1}
    n2:=table^.right_link; {last N-node}
    while n2<>nil do begin
        if n2<>n1 then chaining_n_to_n(n1,n2);
        n2:=n2^.pred_link;
    end;
end;
```

Figure 9.5. Procedure `non_refl_chaining`

`non_refl_chaining` calls `chaining_n_to_n` on the inputted N-node with each N-node in the chaining table except itself. This takes care of creating all chains to E-nodes lying under the inputted N-node.

Procedure `refl_chaining` is very similar to `non_refl_chaining` and is shown below in Figure 9.6.

```
procedure refl_chaining(n1:NodePointer);
    var n2:NodePointer;
begin
    {try ps preceding n1 in same simplex}
    n2:=simplex_pred(n1);
    while n2<>nil do begin
        if n2<>n1 then chaining_n_to_n(n1,n2);
        n2:=simplex_pred(n2);
    end;
end;
```

Figure 9.6. Procedure `refl_chaining`

Since the N-node inputted to refl_chaining is reflexive, refl_chaining only calls chaining_n_to_n on the inputted N-node with each preceding N-node within the same simplex as the inputted N-node.

Function simplex_pred, which is used by procedure refl_chaining, simply returns the N-node that preceeds the inputted N-node in the same simplex. Function simplex_pred is shown below in Figure 9.7.

```
function simplex_pred(n1:NodePointer):NodePointer;
    label 100;
    var answer:NodePointer;
begin
    answer:=n1;
    while true do with answer^ do begin
        answer:=left_link;
        if answer=nil then goto 100;
        if id=N_NODE then goto 100;
    end;
    100: simplex_pred:=answer;
end;
```

Figure 9.7. Function simplex_pred

Procedure chaining_n_to_n is called by procedures refl_chaining and non_refl_chaining and is shown below in Figure 9.8.

```
procedure chaining_n_to_n(n1,n2:NodePointer);
    label 100;
    var old_end_col_link,e1:NodePointer;
begin
    if not sc(n1,n2) then goto 100;
    if not agr(n1,n2) then goto 100;
    if not rnr(n1,n2) then goto 100;
    old_end_col_link:=n1^.end_col_link;
    e1:=n1;
    while e1<>old_end_col_link do begin
        e1:=e1^.col_link;
        if e1<>nil then chaining_e_to_n(e1,n2);
    end;
    100:;
end;
```

Figure 9.8. Procedure chaining_n_to_n

If Syntactic Conditions, Agreement, and the Reflexive Nonreflexive Rule hold, then procedure chaining_e_to_n is called on each E-node lying underneath the first N-node.

Procedure chaining_e_to_n, which is called by procedure chaining_n_to_n, is shown below in Figure 9.9.

```
procedure chaining_e_to_n(e1,n2:NodePointer);
begin
    if agr(e1,n2) then new_chain(e1,n2);
end;
```

Figure 9.9. Procedure chaining_e_to_n

If the inputted E-node agrees with the inputted N-node, then a new chain is created from a copy of the inputted N-node to the inputted E-node by calling procedure new_chain.

Procedure new_chain, which is called by chaining_e_to_n, is shown below in Figure 9.10.

```
procedure new_chain(e1,n2:NodePointer);
    var n:NodePointer;
        i:integer;
begin
    {add n2^e1, with right subscript and features}
    n:=new_e_node:
    with n^ do begin
        np_link:=n2;
        chain_link:=ei;
        sub:=chr(ord(n2^.end_col_link^.sub)+1);
        {n2 nonsyntactic QUESTION (?) features replaced
         by corresponding e1 features}
        for i:=1 to features do begin
            if (ftr[i]=QUESTION) and (i<>RPF)
            then ftr[i]:=e1^.ftr[i];
        end;
    end;
    with n2^ do begin
        end_col_link^.col_link:=n:
        end_col_link:=n;
    end;
end{new_chain};
```

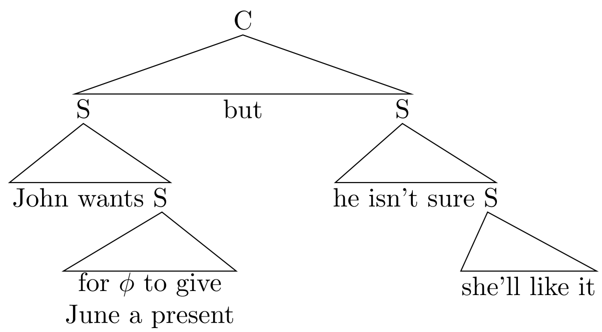
Figure 9.10. Procedure new_chain

Procedure new_chain creates a copy of the inputted N-node and chain_link's it to the inputted E-node. Semantic Feature's are copied from the inputted E-node to the copy of the inputted N-node.

10 Table Processor III

The last two chapters have been devoted to describing the Table Processor. It is time for some real examples: The first example we present in this chapter was produced using procedure `view_Node_str` of the Node Processor along with some intermittent write statements indicating when LIP are entering and exiting some of the more important routines and some or their results. We start off with the example first presented in Chapter 8.

(10.1) John wants to give June a present, but he isn't sure she'll like it.



Processing (10.1) with some intermediate output gives the listing shown below. This is somewhat verbose, but later examples will be cleaner and shorter, though less detailed output.

	Features							
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
John	−	−	−	+	−	−	+	−
ϕ	+	?	?	?	?	?	?	−
June	−	−	−	+	−	+	+	−
present	−	−	−	+	−	?	−	−
he	+	−	−	+	−	−	+	−
she	+	−	−	+	−	+	+	−
it	+	−	−	+	−	?	−	−

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:0, ec:0, pr:0, su:0, nu:3)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:0, ec:0, pr:0, su:0, nu:5)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:0, ec:0, pr:0, su:0, nu:6)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:0, ec:0, pr:0, su:0, nu:7)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:0, ec:0, pr:0, su:0, nu:9)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:0, ec:0, pr:0, su:0, nu:11)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:0, ec:0, pr:0, su:0, nu:12)

```
chaining
    init_table
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_a, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_a, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_a, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_a, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
init_table: exiting
chaining_n(it)
    non_refl_chaining(it)
        chaining_n_to_n(it, she)
            sc(it, she) = true
            agr(it, she) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(it, he)
            sc(it, he) = true
            agr(it, he) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(it, present)
            sc(it, present) = true
            agr(it, present) = true
            rnr(it, present) = true
            chaining_e_to_n(it_a, present)
                agr(it_a, present) = true
                new_chain(it_a, present)
                    new_chain: create present_b
                    new_chain: create present_b^it_a
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_a, pr:0, su:5, nu:3)
$\mathbf{3_a}$	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_a, pr:3, su:6, nu:5)
$\mathbf{5_a}$	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_a, pr:5, su:7, nu:6)
$\mathbf{6_a}$	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
$\mathbf{7_a}$	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
$\mathbf{7_b}$	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
$\mathbf{9_a}$	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
$\mathbf{11_a}$	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
$\mathbf{12_a}$	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(it, June)
    sc(it, June) = true
    agr(it, June) = false
chaining_n_to_n: exiting
chaining_n_to_n(it, ϕ)
    sc(it, ϕ) = true
    agr(it, ϕ) = true
    rnr(it, ϕ) = true
    chaining_e_to_n(it_a, ϕ)
        agr(it_a, ϕ) = true
        new_chain(it_a, ϕ)
            new_chain: create ϕ_b
            new_chain: create ϕ_b^it_a
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_a, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_b, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_a, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(it, John)
            sc(it, John) = true
            agr(it, John) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(she)
    non_refl_chaining(she)
        chaining_n_to_n(she, it)
            sc(she, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(she, he)
            sc(she, he) = true
            agr(she, he) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(she, present)
            sc(she, present) = true
            agr(she, present) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(she, June)
            sc(she, June) = true
            agr(she, June) = true
            rnr(she, June) = true
            chaining_e_to_n(she_a, June)
                agr(she_a, June) = true
                new_chain(she_a, June)
                    new_chain: create June_b
                    new_chain: create June_b^she_a
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_a, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_b, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
6_b	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(she, φ)
    sc(she, φ) = true
    agr(she, φ) = true
    rnr(she, φ) = true
    chaining_e_to_n(she_a, φ)
        agr(she_a, φ) = true
        new_chain(she_a, φ)
            new_chain: create φ_C
            new_chain: create φ_C^she_a
```

	Nodes
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_a, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_c, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:5_c)
5_c	(E, sub:C, ftr:[+--+-++-], np:5, ch:11_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
6_b	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(she, John)
            sc(she, John) = true
            agr(she, John) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(he)
    non_refl_chaining(he)
        chaining_n_to_n(he, it)
            sc(he, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, she)
            sc(he, she) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, present)
            sc(he, present) = true
            agr(he, present) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, June)
            sc(he, June) = true
            agr(he, June) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, φ)
            sc(he, φ) = true
            agr(he, φ) = true
            rnr(he, φ) = true
            chaining_e_to_n(he_a, φ)
                agr(he_a, φ) = true
                new_chain(he_a, φ)
                    new_chain: create φ_d
                    new_chain: create φ_d^he_a
```

	Nodes
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_a, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_d, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:5_c)
5_c	(E, sub:C, ftr:[+--+-++-], np:5, ch:11_a, co:5_d)
5_d	(E, sub:D, ftr:[+--+--+-], np:5, ch:9_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
6_b	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(he, John)
    sc(he, John) = true
    agr(he, John) = true
    rnr(he, John) = true
    chaining_e_to_n(he_a, John)
        agr(he_a, John) = true
        new_chain(he_a, John)
            new_chain: create John_b
            new_chain: create John_b^he_a
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_b, pr:0, su:5, nu:3)
$\mathbf{3_a}$	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:3_b)
$\mathbf{3_b}$	(E, sub:B, ftr:[---+--+-], np:3, ch:9_a, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_d, pr:3, su:6, nu:5)
$\mathbf{5_a}$	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
$\mathbf{5_b}$	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:5_c)
$\mathbf{5_c}$	(E, sub:C, ftr:[+--+-++-], np:5, ch:11_a, co:5_d)
$\mathbf{5_d}$	(E, sub:D, ftr:[+--+--+-], np:5, ch:9_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
$\mathbf{6_a}$	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
$\mathbf{6_b}$	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
$\mathbf{7_a}$	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
$\mathbf{7_b}$	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
$\mathbf{9_a}$	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
$\mathbf{11_a}$	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
$\mathbf{12_a}$	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(φ)
    non_refl_chaining(φ)
        chaining_n_to_n(φ, it)
            sc(φ, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, she)
            sc(φ, she) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, he)
            sc(φ, he) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, present)
            sc(φ, present) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, June)
            sc(φ, June) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, John)
            sc(φ, John) = true
            agr(φ, John) = true
            rnr(φ, John) = true
            chaining_e_to_n(φa, John)
                agr(φa, John) = true
                new_chain(φa, John)
                    new_chain: create Johnc
                    new_chain: create Johnc^φa
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_c, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:3_b)
3_b	(E, sub:B, ftr:[---+--+-], np:3, ch:9_a, co:3_c)
3_c	(E, sub:C, ftr:[---+--+-], np:3, ch:5_a, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_d, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:5_c)
5_c	(E, sub:C, ftr:[+--+-++-], np:5, ch:11_a, co:5_d)
5_d	(E, sub:D, ftr:[+--+--+-], np:5, ch:9_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
6_b	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

```
    new_chain: exiting
chaining_e_to_n: exiting
chaining_e_to_n(φb, John)
    agr(φb, John) = false
chaining_e_to_n: exiting
chaining_e_to_n(φc, John)
    agr(φc, John) = false
chaining_e_to_n: exiting
chaining_e_to_n(φd, John)
    agr(φd, John) = true
    new_chain(φd, John)
        new_chain: create Johnd
        new_chain: create Johnd^φd
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_d, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:3_b)
3_b	(E, sub:B, ftr:[---+--+-], np:3, ch:9_a, co:3_c)
3_c	(E, sub:C, ftr:[---+--+-], np:3, ch:5_a, co:3_d)
3_d	(E, sub:D, ftr:[---+--+-], np:3, ch:5_d, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_d, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:5_c)
5_c	(E, sub:C, ftr:[+--+-++-], np:5, ch:11_a, co:5_d)
5_d	(E, sub:D, ftr:[+--+--+-], np:5, ch:9_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
6_b	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

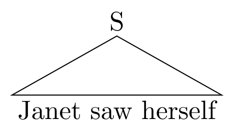
```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
  chaining_n: exiting
chaining: exiting
```

Nodes	
1	(C, up:0, dn:2, lt:0, rt:0, th:2, nu:1)
2	(S, up:1, dn:3, lt:0, rt:8, th:3, nu:2)
3	(N, lit:John, ftr:[---+--+-], up:2, dn:0, lt:0, rt:4, th:4, np:3, ch:0, co:3_a, ec:3_d, pr:0, su:5, nu:3)
3_a	(E, sub:A, ftr:[---+--+-], np:3, ch:0, co:3_b)
3_b	(E, sub:B, ftr:[---+--+-], np:3, ch:9_a, co:3_c)
3_c	(E, sub:C, ftr:[---+--+-], np:3, ch:5_a, co:3_d)
3_d	(E, sub:D, ftr:[---+--+-], np:3, ch:5_d, co:0)
4	(S, up:2, dn:5, lt:3, rt:0, th:5, nu:4)
5	(N, lit:ϕ, ftr:[+??????-], up:4, dn:0, lt:0, rt:6, th:6, np:5, ch:0, co:5_a, ec:5_d, pr:3, su:6, nu:5)
5_a	(E, sub:A, ftr:[+??????-], np:5, ch:0, co:5_b)
5_b	(E, sub:B, ftr:[+--+-?--], np:5, ch:12_a, co:5_c)
5_c	(E, sub:C, ftr:[+--+-++-], np:5, ch:11_a, co:5_d)
5_d	(E, sub:D, ftr:[+--+--+-], np:5, ch:9_a, co:0)
6	(N, lit:June, ftr:[---+-++-], up:4, dn:0, lt:5, rt:7, th:7, np:6, ch:0, co:6_a, ec:6_b, pr:5, su:7, nu:6)
6_a	(E, sub:A, ftr:[---+-++-], np:6, ch:0, co:6_b)
6_b	(E, sub:B, ftr:[---+-++-], np:6, ch:11_a, co:0)
7	(N, lit:present, ftr:[---+-?--], up:4, dn:0, lt:6, rt:0, th:8, np:7, ch:0, co:7_a, ec:7_b, pr:6, su:9, nu:7)
7_a	(E, sub:A, ftr:[---+-?--], np:7, ch:0, co:7_b)
7_b	(E, sub:B, ftr:[---+-?--], np:7, ch:12_a, co:0)
8	(S, up:1, dn:9, lt:2, rt:0, th:9, nu:8)
9	(N, lit:he, ftr:[+--+--+-], up:8, dn:0, lt:0, rt:10, th:10, np:9, ch:0, co:9_a, ec:9_a, pr:7, su:11, nu:9)
9_a	(E, sub:A, ftr:[+--+--+-], np:9, ch:0, co:0)
10	(S, up:8, dn:11, lt:9, rt:0, th:11, nu:10)
11	(N, lit:she, ftr:[+--+-++-], up:10, dn:0, lt:0, rt:12, th:12, np:11, ch:0, co:11_a, ec:11_a, pr:9, su:12, nu:11)
11_a	(E, sub:A, ftr:[+--+-++-], np:11, ch:0, co:0)
12	(N, lit:it, ftr:[+--+-?--], up:10, dn:0, lt:11, rt:0, th:0, np:12, ch:0, co:12_a, ec:12_a, pr:11, su:0, nu:12)
12_a	(E, sub:A, ftr:[+--+-?--], np:12, ch:0, co:0)

The previous example should give enough details away to satisfy the reader's curiosity, but the form of the previous example is rather burdensome. From now on, we'll keep to a more concise, if less detailed, output. To indicate a `chain_link` between two E-nodes, we use the symbol ^.

Below are some more examples.

(10.2) Janet saw herself.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Janet	−	−	−	+	−	+	+	−
herself	+	−	−	+	−	+	+	+

```
chaining
    init_table
```

Chaining	
Janet	**herself**
Janet$_a$	herself$_a$

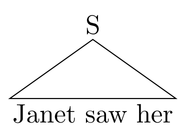
```
    init_table: exiting
    chaining_n(herself)
        refl_chaining(herself)
            simplex_pred(herself)
            simplex_pred:  Janet
            chaining_n_to_n(herself, Janet)
                sc(herself, Janet) = true
                agr(herself, Janet) = true
                rnr(herself, Janet) = true
                chaining_e_to_n(herself_a, Janet)
                    agr(herself_a, Janet) = true
                    new_chain(herself_a, Janet)
                        new_chain: create Janet_b
                        new_chain: create Janet_b^herself_a
```

Chaining	
Janet	**herself**
Janet$_a$	herself$_a$
Janet$_b$ˆherself$_a$	

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    simplex_pred(Janet)
    simplex_pred:
  refl_chaining: exiting
 chaining_n: exiting
chaining: exiting
```

Chaining	
Janet	**herself**
Janet$_a$	herself$_a$
Janet$_b$ˆherself$_a$	

(10.3) Janet saw her.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Janet	−	−	−	+	−	+	+	−
her	+	−	−	+	−	+	+	−

```
chaining
    init_table
```

Chaining	
Janet	**her**
Janet$_a$	her$_a$

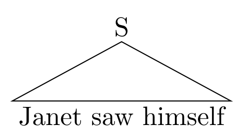
```
    init_table: exiting
    chaining_n(her)
        non_refl_chaining(her)
            chaining_n_to_n(her, Janet)
                sc(her, Janet) = true
                agr(her, Janet) = true
                rnr(her, Janet) = true
                chaining_e_to_n(her_a, Janet)
                    agr(her_a, Janet) = true
                    new_chain(her_a, Janet)
                        new_chain: create Janet_b
                        new_chain: create Janet_b^her_a
```

Chaining	
Janet	**her**
Janet$_a$ Janet$_b$^her$_a$	her$_a$

```
                    new_chain: exiting
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining	
Janet	**her**
Janet$_a$ Janet$_b$^her$_a$	her$_a$

(10.4) *Janet saw himself.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Janet	−	−	−	+	−	+	+	−
himself	+	−	−	+	−	−	+	+

```
chaining
    init_table
```

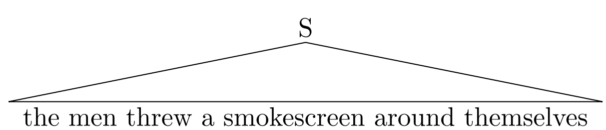
Chaining	
Janet	**himself**
Janet$_{a}$	himself$_{a}$

```
    init_table: exiting
    chaining_n(himself)
        refl_chaining(himself)
            simplex_pred(himself)
            simplex_pred:  Janet
            chaining_n_to_n(himself, Janet)
                sc(himself, Janet) = true
                agr(himself, Janet) = false
            chaining_n_to_n: exiting
            simplex_pred(Janet)
            simplex_pred:
        refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining	
Janet	**himself**
Janet$_{a}$	himself$_{a}$

Examples (10.5)-(10.11) are from Lees and Klima [22].

(10.5) The men threw a smokescreen around themselves.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
men	−	−	−	+	+	−	+	−
smokescreen	−	−	−	+	−	?	−	−
themselves	+	−	−	+	+	?	?	+

```
chaining
    init_table
```

Chaining		
men	**smokescreen**	**themselves**
men_a	smokescreen_a	themselves_a

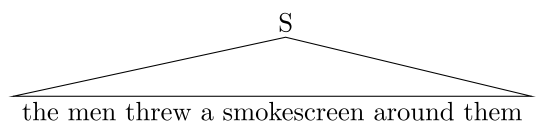
```
    init_table: exiting
    chaining_n(themselves)
        refl_chaining(themselves)
            simplex_pred(themselves)
            simplex_pred:  smokescreen
            chaining_n_to_n(themselves, smokescreen)
                sc(themselves, smokescreen) = true
                agr(themselves, smokescreen) = false
            chaining_n_to_n: exiting
            simplex_pred(smokescreen)
            simplex_pred:  men
            chaining_n_to_n(themselves, men)
                sc(themselves, men) = true
                agr(themselves, men) = true
                rnr(themselves, men) = true
                chaining_e_to_n(themselves_a, men)
                    agr(themselves_a, men) = true
                    new_chain(themselves_a, men)
                        new_chain: create men_b
                        new_chain: create men_b^themselves_a
```

Chaining		
men	**smokescreen**	**themselves**
men_a $\text{men}_b\hat{\ }\text{themselves}_a$	smokescreen_a	themselves_a

```
                    new_chain: exiting
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
            simplex_pred(men)
            simplex_pred:
        refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining		
men	**smokescreen**	**themselves**
men$_a$	smokescreen$_a$	themselves$_a$
men$_b$ˆthemselves$_a$		

(10.6) The men found a smokescreen around them.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
men	−	−	−	+	+	−	+	−
smokescreen	−	−	−	+	−	?	−	−
them	+	−	−	+	+	?	?	−

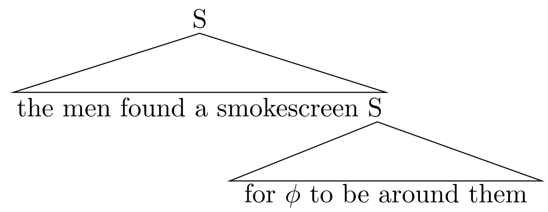
```
chaining
    init_table
```

Chaining		
men	**smokescreen**	**them**
men$_a$	smokescreen$_a$	them$_a$

```
    init_table: exiting
    chaining_n(them)
        non_refl_chaining(them)
            chaining_n_to_n(them, smokescreen)
                sc(them, smokescreen) = true
                agr(them, smokescreen) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(them, men)
                sc(them, men) = true
                agr(them, men) = true
                rnr(them, men) = false
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining		
men	**smokescreen**	**them**
men$_a$	smokescreen$_a$	them$_a$

(10.7) The men found a smokescreen to be around them.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
men	−	−	−	+	+	−	+	−
smokescreen	−	−	−	+	−	?	−	−
them	+	−	−	+	+	?	?	−

```
chaining
    init_table
```

Chaining		
men	**smokescreen**	**them**
men$_a$	smokescreen$_a$	them$_a$

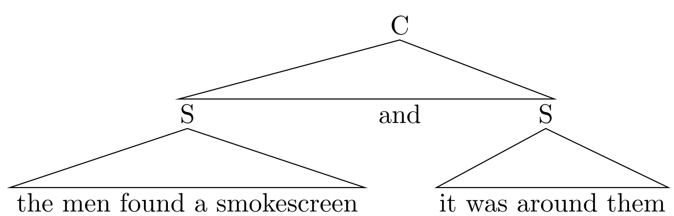
```
    init_table: exiting
    chaining_n(them)
        non_refl_chaining(them)
            chaining_n_to_n(them, smokescreen)
                sc(them, smokescreen) = true
                agr(them, smokescreen) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(them, men)
                sc(them, men) = true
                agr(them, men) = true
                rnr(them, men) = true
                chaining_e_to_n(them_a, men)
                    agr(them_a, men) = true
                    new_chain(them_a, men)
                        new_chain: create men_b
                        new_chain: create men_b^them_a
```

Chaining		
men	**smokescreen**	**them**
men$_a$ men$_b$ˆthem$_a$	smokescreen$_a$	them$_a$

```
                    new_chain: exiting
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining		
men	**smokescreen**	**them**
men$_a$ men$_b$ˆthem$_a$	smokescreen$_a$	them$_a$

(10.8) The men found a smokescreen and it was around them.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
men	−	−	−	+	+	−	+	−
smokescreen	−	−	−	+	−	?	−	−
it	+	−	−	+	−	?	−	−
them	+	−	−	+	+	?	?	−

```
chaining
    init_table
```

Chaining			
men	**smokescreen**	**it**	**them**
men$_a$	smokescreen$_a$	it$_a$	them$_a$

```
init_table: exiting
chaining_n(them)
    non_refl_chaining(them)
        chaining_n_to_n(them, it)
            sc(them, it) = true
            agr(them, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(them, smokescreen)
            sc(them, smokescreen) = true
            agr(them, smokescreen) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(them, men)
            sc(them, men) = true
            agr(them, men) = true
            rnr(them, men) = true
            chaining_e_to_n(them_a, men)
                agr(them_a, men) = true
                new_chain(them_a, men)
                    new_chain: create men_b
                    new_chain: create men_b^them_a
```

Chaining			
men	**smokescreen**	**it**	**them**
$\texttt{men}_a$ $\texttt{men}_b$ˆ$\texttt{them}_a$	$\texttt{smokescreen}_a$	$\texttt{it}_a$	$\texttt{them}_a$

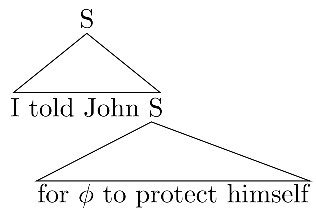
```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(it)
    non_refl_chaining(it)
        chaining_n_to_n(it, them)
            sc(it, them) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(it, smokescreen)
            sc(it, smokescreen) = true
            agr(it, smokescreen) = true
            rnr(it, smokescreen) = true
            chaining_e_to_n(it_a, smokescreen)
                agr(it_a, smokescreen) = true
                new_chain(it_a, smokescreen)
                    new_chain: create smokescreen_b
                    new_chain: create smokescreen_b^it_a
```

Chaining			
men	**smokescreen**	**it**	**them**
men$_a$	smokescreen$_a$	it$_a$	them$_a$
men$_b$^them$_a$	smokescreen$_b$^it$_a$		

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(it, men)
            sc(it, men) = true
            agr(it, men) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining: exiting
```

Chaining			
men	**smokescreen**	**it**	**them**
men$_a$	smokescreen$_a$	it$_a$	them$_a$
men$_b$^them$_a$	smokescreen$_b$^it$_a$		

(10.9) I told John to protect himself.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
I	+	+	−	−	−	?	+	−
John	−	−	−	+	−	−	+	−
ϕ	+	?	?	?	?	?	?	−
himself	+	−	−	+	−	−	+	+

```
chaining
    init_table
```

Chaining			
I	**John**	**ϕ**	**himself**
I_a	John_a	ϕ_a	$\mathrm{himself}_a$

```
    init_table: exiting
    chaining_n(himself)
        refl_chaining(himself)
            simplex_pred(himself)
            simplex_pred:  φ
            chaining_n_to_n(himself, φ)
                sc(himself, φ) = true
                agr(himself, φ) = true
                rnr(himself, φ) = true
                chaining_e_to_n(himself_a, φ)
                    agr(himself_a, φ) = true
                    new_chain(himself_a, φ)
                        new_chain: create φ_b
                        new_chain: create φ_b^himself_a
```

Chaining			
I	**John**	**ϕ**	**himself**
I$_a$	John$_a$	ϕ_a	himself$_a$
		ϕ_b^himself$_a$	

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    simplex_pred(φ)
    simplex_pred:
refl_chaining: exiting
chaining_n: exiting
chaining_n(φ)
    non_refl_chaining(φ)
        chaining_n_to_n(φ, himself)
            sc(φ, himself) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, John)
            sc(φ, John) = true
            agr(φ, John) = true
            rnr(φ, John) = true
            chaining_e_to_n(φa, John)
                agr(φa, John) = true
                new_chain(φa, John)
                    new_chain: create Johnb
                    new_chain: create Johnb^φa
```

Chaining			
I	**John**	**ϕ**	**himself**
I$_a$	John$_a$	ϕ_a	himself$_a$
	John$_b$^ϕ_a	ϕ_b^himself$_a$	

```
        new_chain: exiting
    chaining_e_to_n: exiting
    chaining_e_to_n(φb, John)
        agr(φb, John) = true
        new_chain(φb, John)
            new_chain: create Johnc
            new_chain: create Johnc^φb
```

Chaining			
I	**John**	**ϕ**	**himself**
I_a	$John_a$	ϕ_a	$himself_a$
	$John_b\hat{}\phi_a$	$\phi_b\hat{}himself_a$	
	$John_c\hat{}\phi_b$		

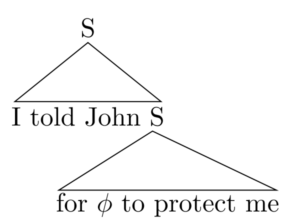
```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(ϕ, I)
    sc(ϕ, I) = true
    agr(ϕ, I) = true
    rnr(ϕ, I) = true
    chaining_e_to_n(ϕa, I)
        agr(ϕa, I) = true
        new_chain(ϕa, I)
            new_chain: create Ib
            new_chain: create Ib^ϕa
```

Chaining			
I	**John**	**ϕ**	**himself**
I_a	$John_a$	ϕ_a	$himself_a$
$I_b\hat{}\phi_a$	$John_b\hat{}\phi_a$	$\phi_b\hat{}himself_a$	
	$John_c\hat{}\phi_b$		

```
                    new_chain: exiting
                chaining_e_to_n: exiting
                chaining_e_to_n(φ_b, I)
                    agr(φ_b, I) = false
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
    chaining_n(I)
        non_refl_chaining(I)
            chaining_n_to_n(I, himself)
                sc(I, himself) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(I, φ)
                sc(I, φ) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(I, John)
                sc(I, John) = false
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining			
I	**John**	**ϕ**	**himself**
$\mathtt{I}_a$	$\mathtt{John}_a$	ϕ_a	$\mathtt{himself}_a$
$\mathtt{I}_b$ ˆ ϕ_a	$\mathtt{John}_b$ ˆ ϕ_a	ϕ_b ˆ $\mathtt{himself}_a$	
	$\mathtt{John}_c$ ˆ ϕ_b		

(10.10) I told John to protect me.



	Features							
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
I	+	+	−	−	−	?	+	−
John	−	−	−	+	−	−	+	−
ϕ	+	?	?	?	?	?	?	−
me	+	+	−	−	−	?	+	−

```
chaining
    init_table
```

Chaining			
I	**John**	ϕ	**me**
I_a	$John_a$	ϕ_a	me_a

```
    init_table: exiting
    chaining_n(me)
        non_refl_chaining(me)
            chaining_n_to_n(me, ϕ)
                sc(me, ϕ) = true
                agr(me, ϕ) = true
                rnr(me, ϕ) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(me, John)
                sc(me, John) = true
                agr(me, John) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(me, I)
                sc(me, I) = true
                agr(me, I) = true
                rnr(me, I) = true
                chaining_e_to_n(me_a, I)
                    agr(me_a, I) = true
                    new_chain(me_a, I)
                        new_chain: create I_b
                        new_chain: create I_b^me_a
```

Chaining			
I	**John**	ϕ	**me**
I_a I_b^me_a	$John_a$	ϕ_a	me_a

```
        new_chain: exiting
      chaining_e_to_n: exiting
    chaining_n_to_n: exiting
  non_refl_chaining: exiting
chaining_n: exiting
chaining_n(φ)
  non_refl_chaining(φ)
    chaining_n_to_n(φ, me)
      sc(φ, me) = false
    chaining_n_to_n: exiting
    chaining_n_to_n(φ, John)
      sc(φ, John) = true
      agr(φ, John) = true
      rnr(φ, John) = true
      chaining_e_to_n(φa, John)
        agr(φa, John) = true
        new_chain(φa, John)
          new_chain: create John_b
          new_chain: create John_b^φa
```

Chaining			
I	**John**	**ϕ**	**me**
I_a	$John_a$	ϕ_a	me_a
I_b ^ me_a	$John_b$ ^ ϕ_a		

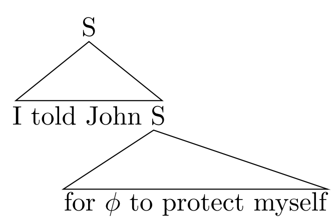
```
        new_chain: exiting
      chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    chaining_n_to_n(φ, I)
      sc(φ, I) = true
      agr(φ, I) = true
      rnr(φ, I) = true
      chaining_e_to_n(φa, I)
        agr(φa, I) = true
        new_chain(φa, I)
          new_chain: create I_c
          new_chain: create I_c^φa
```

Chaining			
I	**John**	**ϕ**	**me**
I_a	$John_a$	ϕ_a	me_a
I_b ^ me_a	$John_b$ ^ ϕ_a		
I_c ^ ϕ_a			

```
                    new_chain: exiting
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
    chaining_n(I)
        non_refl_chaining(I)
            chaining_n_to_n(I, me)
                sc(I, me) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(I, φ)
                sc(I, φ) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(I, John)
                sc(I, John) = false
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining			
I	**John**	ϕ	**me**
I_a	$John_a$	ϕ_a	me_a
I_b ˆ me_a	$John_b$ ˆ ϕ_a		
I_c ˆ ϕ_a			

(10.11) I told John to protect myself.



Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
I	+	+	–	–	–	?	+	–
John	–	–	–	+	–	–	+	–
ϕ	+	?	?	?	?	?	?	–
myself	+	+	–	–	–	?	+	+

```
chaining
    init_table
```

Chaining			
I	**John**	**ϕ**	**myself**
I_a	John_a	ϕ_a	myself_a

```
    init_table: exiting
    chaining_n(myself)
        refl_chaining(myself)
            simplex_pred(myself)
            simplex_pred:  φ
            chaining_n_to_n(myself, φ)
                sc(myself, φ) = true
                agr(myself, φ) = true
                rnr(myself, φ) = true
                chaining_e_to_n(myself_a, φ)
                    agr(myself_a, φ) = true
                    new_chain(myself_a, φ)
                        new_chain: create φ_b
                        new_chain: create φ_b^myself_a
```

Chaining			
I	**John**	**ϕ**	**myself**
I_a	John_a	ϕ_a ϕ_bˆmyself_a	myself_a

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    simplex_pred(φ)
    simplex_pred:
refl_chaining: exiting
chaining_n: exiting
chaining_n(φ)
    non_refl_chaining(φ)
        chaining_n_to_n(φ, myself)
            sc(φ, myself) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, John)
            sc(φ, John) = true
            agr(φ, John) = true
            rnr(φ, John) = true
            chaining_e_to_n(φa, John)
                agr(φa, John) = true
                new_chain(φa, John)
                    new_chain: create Johnb
                    new_chain: create Johnb^φa
```

Chaining			
I	**John**	**ϕ**	**myself**
I_a	$John_a$	ϕ_a	$myself_a$
	$John_b$^ϕ_a	ϕ_b^$myself_a$	

```
                new_chain: exiting
            chaining_e_to_n: exiting
            chaining_e_to_n(φb, John)
                agr(φb, John) = false
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, I)
            sc(φ, I) = true
            agr(φ, I) = true
            rnr(φ, I) = true
            chaining_e_to_n(φa, I)
                agr(φa, I) = true
                new_chain(φa, I)
                    new_chain: create Ib
                    new_chain: create Ib^φa
```

Chaining			
I	**John**	**ϕ**	**myself**
$\mathtt{I}_a$	$\mathtt{John}_a$	ϕ_a	$\mathtt{myself}_a$
$\mathtt{I}_b$^ϕ_a	$\mathtt{John}_b$^ϕ_a	ϕ_b^$\mathtt{myself}_a$	

```
          new_chain: exiting
      chaining_e_to_n: exiting
      chaining_e_to_n(φb, I)
          agr(φb, I) = true
          new_chain(φb, I)
              new_chain: create Ic
              new_chain: create Ic^φb
```

Chaining			
I	**John**	**ϕ**	**myself**
$\mathtt{I}_a$	$\mathtt{John}_a$	ϕ_a	$\mathtt{myself}_a$
$\mathtt{I}_b$^ϕ_a	$\mathtt{John}_b$^ϕ_a	ϕ_b^$\mathtt{myself}_a$	
$\mathtt{I}_c$^ϕ_b			

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
  chaining_n: exiting
  chaining_n(I)
    non_refl_chaining(I)
        chaining_n_to_n(I, myself)
            sc(I, myself) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I, φ)
            sc(I, φ) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I, John)
            sc(I, John) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
  chaining_n: exiting
chaining: exiting
```

Chaining			
I	**John**	**ϕ**	**myself**
I$_a$	John$_a$	ϕ_a	myself$_a$
I$_b$^ϕ_a	John$_b$^ϕ_a	ϕ_b^myself$_a$	
I$_c$^ϕ_b			

11 Table Interpreter

The Table Interpreter module defines function `interpret` and has the form shown in Figure 11.1.

```
module table_interp;
    use globals,node_proc;
    define function interpret(nnodes:array of NodePointer)
        :array of array of array of NodePointer;
    implement
        (*$I TABLE INTERP*)
    end;
```

Figure 11.1. Skeleton of the Table Interpreter

Basically, after the chaining table is created, a number of chains are implicitly defined by the chaining table and it is the job of the Table Interpreter to mesh these chains back into copies of the system tree, returning all trees defined by legitimate interpretations.

A nonpronominal E-node with the E-nodes that are traced by walking down `chain_link`'s until `nil` `chain_link` is reached constitute a **chain**. A set of chains defined by the chaining table which cover all the pronominal N-nodes and do not intersect constitute a legitimate **interpretation**.

Take the table given in Figure 11.2 as an example.

Chaining						
John	**ϕ**	**June**	**present**	**he**	**she**	**it**
John$_a$	ϕ_a	June$_a$	present$_a$	he$_a$	she$_a$	it$_a$
John$_b$^he$_a$	ϕ_b^it$_a$	June$_b$^she$_a$	present$_b$^it$_a$			
John$_c$^ϕ_a	ϕ_c^she$_a$					
John$_d$^ϕ_d	ϕ_d^he$_a$					

Figure 11.2. Typical Chaining Table

The chains present in Figure 11.2 are exactly (11.3)-(11.10) given below. Note that no chain begins with a pronoun. (Here, we are staying with the convention of Chapter 10 where a "^" symbol indicates a `chain_link`).

(11.3) John$_a$

(11.4) John$_{b}$ˆhe$_{a}$
(11.5) John$_{c}$ˆϕ_{a}
(11.6) John$_{d}$ˆϕ_{d}ˆhe$_{a}$
(11.7) June$_{a}$
(11.8) June$_{b}$ˆshe$_{a}$
(11.9) present$_{a}$
(11.10) present$_{b}$ˆit$_{a}$

The only interpretation derivable from Figure 11.2 is (11.11).

(11.11) John$_{d}$ˆϕ_{d}ˆhe$_{a}$ June$_{b}$ˆshe$_{a}$ present$_{b}$ˆit$_{a}$

Exactly how chaining information from a table to system parse tree will have to be system dependent, but we can imagine that noun phrases in a system parse tree are some kind of list elements which have linked to them, among other things, lists corresponding to their semantics. It is up to the Table Interpreter to set any `chain_link`'s inside the semantics of the noun phrases of the system parse tree. The Semantic Processor module of the system should then be powerful enough to be able to handle the kind of coordination that `chain_link`'s imply.

This strategy has a number of possibilities that simple methods of **coreferencing** are just not able to handle. Consider sentence (11.12) for example.

(11.12) Jack's house burned down, but he rebuilt it.

We can't really say that it **corefers** with Jack's house as Jack's house is some object that existed in the past and has stopped existing while it refers to some new object. This does not mean that it cannot chain from Jack's house, however, and indeed it should. The information the Semantic Processor module needs to give meaning to it is contained in Jack's house, and so there must be a `chain_link` from Jack's house to it in order for the Semantic Processor to give meaning to it.

A similar result holds for quantifiers. We see that (11.13) is not equivalent to (11.14).

(11.13) Every connoisseur loves his wine and cheese. $\neq$
(11.14) Every connoisseur loves every connoisseur's wine and cheese.

Quite clearly, his cannot be replaced by every connoisseur and preserve the meaning of the sentence. Instead, (11.13) has more the meaning given by (11.15).

(11.15) (For all x: x is a connoisseur)(x loves x's wine and cheese.)

A number of other examples are pointed out by Bresnan [3].

(11.16) All Italians think they are handsome. $\neq$

(11.17) All Italians think all Italians are handsome.
(11.18) Every Italian thinks he is handsome. ≠
(11.19) Every Italian thinks every Italian is handsome.
(11.20) Any Italian would die for his mother. ≠
(11.21) Any Italian would die for and Italian's mother.
(11.22) Every Italian thinks that he alone is handsome. ≠
(11.23) *Every Italian thinks that every Italian alone is handsome.
(11.24) One girl claimed that she herself could read Homer. ≠
(11.25) *One girl claimed that one girl herself could read Homer.

It appears that the proper interpretation for a pronoun chained to quantified noun phrase within the **scope of quantification** is for the pronoun to act as a **bound variable**.

When the pronoun is outside the scope of quantification, it is a different story. Consider (11.26) and (11.27) from Evans [6].

(11.26) John owns some sheep and Harry vaccinates them.
(11.27) Mary danced with many boys and they found her interesting.

This time the pronouns are chaining to quantified noun phrases, but do not themselves lie within the scope of quantification. Instead, they appear to refer to the **range of the quantification**.

Similar results hold for (11.28)-(11.31) from Sidner [32].

(11.28) John lost a pen yesterday and Bill found one today.
(11.29) John claimed to have found the solution to the problem, but Bill was sure he had found it.
(11.30) John wants to catch a fish and eat it for supper.
(11.31) No one would put the blame on himself.

The problems mentioned above are all rather tricky, but viewing them from the vantage point of chaining sheds more light on them than viewing them through some kind of coreference. The moral of the story seems to be that anaphora is not coreference.

Using the Table Interpreter now, we present some more examples.

(11.32) Sue told Sandy about herself.

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Sue	−	−	−	+	−	+	+	−
Sandy	−	−	−	+	−	+	+	−
herself	+	−	−	+	−	+	+	+

```
chaining
    init_table
```

Chaining		
Sue	**Sandy**	**herself**
Sue$_a$	Sandy$_a$	herself$_a$

```
init_table: exiting
chaining_n(herself)
    refl_chaining(herself)
        simplex_pred(herself)
        simplex_pred:  Sandy
        chaining_n_to_n(herself, Sandy)
            sc(herself, Sandy) = true
            agr(herself, Sandy) = true
            rnr(herself, Sandy) = true
            chaining_e_to_n(herself_a, Sandy)
                agr(herself_a, Sandy) = true
                new_chain(herself_a, Sandy)
                    new_chain: create Sandy_b
                    new_chain: create Sandy_b^herself_a
```

Chaining		
Sue	**Sandy**	**herself**
Sue$_a$	Sandy$_a$ Sandy$_b$ˆherself$_a$	herself$_a$

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        simplex_pred(Sandy)
        simplex_pred:  Sue
        chaining_n_to_n(herself, Sue)
            sc(herself, Sue) = true
            agr(herself, Sue) = true
            rnr(herself, Sue) = true
            chaining_e_to_n(herself_a, Sue)
                agr(herself_a, Sue) = true
                new_chain(herself_a, Sue)
                    new_chain: create Sue_b
                    new_chain: create Sue_b^herself_a
```

Chaining		
Sue	**Sandy**	**herself**
Sue$_a$ Sue$_b$ˆherself$_a$	Sandy$_a$ Sandy$_b$ˆherself$_a$	herself$_a$

```
            new_chain: exiting
          chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        simplex_pred(Sue)
        simplex_pred:
      refl_chaining: exiting
    chaining_n: exiting
  chaining: exiting
```

Chaining		
Sue	**Sandy**	**herself**
$\texttt{Sue}_a$	$\texttt{Sandy}_a$	$\texttt{herself}_a$
$\texttt{Sue}_b$^$\texttt{herself}_a$	$\texttt{Sandy}_b$^$\texttt{herself}_a$	

Interpretations
$\texttt{Sue}_b$^$\texttt{herself}_a$
$\texttt{Sandy}_b$^$\texttt{herself}_a$

(11.33) *Jill kept talking about himself.

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Jill	−	−	−	+	−	+	+	−
himself	+	−	−	+	−	−	+	+

```
  chaining
    init_table
```

Chaining	
Jill	**himself**
$\texttt{Jill}_a$	$\texttt{himself}_a$

```
    init_table: exiting
    chaining_n(himself)
        refl_chaining(himself)
            simplex_pred(himself)
            simplex_pred:  Jill
            chaining_n_to_n(himself, Jill)
                sc(himself, Jill) = true
                agr(himself, Jill) = false
            chaining_n_to_n: exiting
            simplex_pred(Jill)
            simplex_pred:
        refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining	
Jill	**himself**
$\texttt{Jill}_a$	$\texttt{himself}_a$

Interpretations
NONE

(11.34) Does Jack's making a pig of himself bother Bill?

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Jack's	−	−	−	+	−	−	+	−
pig	−	−	−	+	−	?	+	−
himself	+	−	−	+	−	−	+	+
Bill	−	−	−	+	−	−	+	−

```
chaining
    init_table
```

Chaining			
Jack's	**pig**	**himself**	**Bill**
$\texttt{Jack's}_a$	$\texttt{pig}_a$	$\texttt{himself}_a$	$\texttt{Bill}_a$

```
    init_table: exiting
    chaining_n(himself)
        refl_chaining(himself)
            simplex_pred(himself)
            simplex_pred:  pig
            chaining_n_to_n(himself, pig)
                sc(himself, pig) = true
                agr(himself, pig) = true
                rnr(himself, pig) = true
                chaining_e_to_n(himself_a, pig)
                    agr(himself_a, pig) = true
                    new_chain(himself_a, pig)
                        new_chain: create pig_b
                        new_chain: create pig_b^himself_a
```

Chaining			
Jack's	**pig**	**himself**	**Bill**
Jack's$_a$	pig$_a$ pig$_b$^himself$_a$	himself$_a$	Bill$_a$

```
                    new_chain: exiting
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
            simplex_pred(pig)
            simplex_pred:  Jack's
            chaining_n_to_n(himself, Jack's)
                sc(himself, Jack's) = true
                agr(himself, Jack's) = true
                rnr(himself, Jack's) = false
            chaining_n_to_n: exiting
            simplex_pred(Jack's)
            simplex_pred:
        refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining			
Jack's	**pig**	**himself**	**Bill**
Jack's$_a$	pig$_a$ pig$_b$^himself$_a$	himself$_a$	Bill$_a$

Interpretations
$\texttt{pig}_b$^$\texttt{himself}_a$

(11.35) John wants to give June a present, but he is afraid she won't like it.

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
John	−	−	−	+	−	−	+	−
ϕ	+	?	?	?	?	?	?	−
June	−	−	−	+	−	+	+	−
present	−	−	−	+	−	?	−	−
he	+	−	−	+	−	−	+	−
she	+	−	−	+	−	+	+	−
it	+	−	−	+	−	?	−	−

```
chaining
    init_table
```

Chaining						
John	ϕ	**June**	**present**	**he**	**she**	**it**
$\texttt{John}_a$	ϕ_a	$\texttt{June}_a$	$\texttt{present}_a$	$\texttt{he}_a$	$\texttt{she}_a$	$\texttt{it}_a$

```
    init_table: exiting
    chaining_n(it)
        non_refl_chaining(it)
            chaining_n_to_n(it, she)
                sc(it, she) = true
                agr(it, she) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(it, he)
                sc(it, he) = true
                agr(it, he) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(it, present)
                sc(it, present) = true
                agr(it, present) = true
                rnr(it, present) = true
                chaining_e_to_n(it_a, present)
                    agr(it_a, present) = true
                    new_chain(it_a, present)
                        new_chain: create present_b
                        new_chain: create present_b^it_a
```

Chaining						
John	**ϕ**	**June**	**present**	**he**	**she**	**it**
John$_a$	ϕ_a	June$_a$	present$_a$ present$_b$ˆit$_a$	he$_a$	she$_a$	it$_a$

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(it, June)
    sc(it, June) = true
    agr(it, June) = false
chaining_n_to_n: exiting
chaining_n_to_n(it, φ)
    sc(it, φ) = true
    agr(it, φ) = true
    rnr(it, φ) = true
    chaining_e_to_n(it_a, φ)
        agr(it_a, φ) = true
        new_chain(it_a, φ)
            new_chain: create φ_b
            new_chain: create φ_bˆit_a
```

Chaining						
John	**ϕ**	**June**	**present**	**he**	**she**	**it**
John$_a$	ϕ_a ϕ_bˆit$_a$	June$_a$	present$_a$ present$_b$ˆit$_a$	he$_a$	she$_a$	it$_a$

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(it, John)
            sc(it, John) = true
            agr(it, John) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(she)
    non_refl_chaining(she)
        chaining_n_to_n(she, it)
            sc(she, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(she, he)
            sc(she, he) = true
            agr(she, he) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(she, present)
            sc(she, present) = true
            agr(she, present) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(she, June)
            sc(she, June) = true
            agr(she, June) = true
            rnr(she, June) = true
            chaining_e_to_n(she_a, June)
                agr(she_a, June) = true
                new_chain(she_a, June)
                    new_chain: create June_b
                    new_chain: create June_b^she_a
```

Chaining						
John	**ϕ**	**June**	**present**	**he**	**she**	**it**
John_a	ϕ_a	June_a	present_a	he_a	she_a	it_a
	ϕ_b^it_a	June_b^she_a	present_b^it_a			

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(she, φ)
    sc(she, φ) = true
    agr(she, φ) = true
    rnr(she, φ) = true
    chaining_e_to_n(she_a, φ)
        agr(she_a, φ) = true
        new_chain(she_a, φ)
            new_chain: create φ_c
            new_chain: create φ_c^she_a
```

Chaining						
John	**ϕ**	**June**	**present**	**he**	**she**	**it**
John_a	ϕ_a ϕ_bˆit_a ϕ_cˆshe_a	June_a June_bˆshe_a	present_a present_bˆit_a	he_a	she_a	it_a

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(she, John)
            sc(she, John) = true
            agr(she, John) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(he)
    non_refl_chaining(he)
        chaining_n_to_n(he, it)
            sc(he, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, she)
            sc(he, she) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, present)
            sc(he, present) = true
            agr(he, present) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, June)
            sc(he, June) = true
            agr(he, June) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(he, φ)
            sc(he, φ) = true
            agr(he, φ) = true
            rnr(he, φ) = true
            chaining_e_to_n(he_a, φ)
                agr(he_a, φ) = true
                new_chain(he_a, φ)
                    new_chain: create φ_d
                    new_chain: create φ_d^he_a
```

Chaining						
John	ϕ	**June**	**present**	**he**	**she**	**it**
John_a	ϕ_a ϕ_b^it_a ϕ_c^she_a ϕ_d^he_a	June_a June_b^she_a	present_a present_b^it_a	he_a	she_a	it_a

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(he, John)
    sc(he, John) = true
    agr(he, John) = true
    rnr(he, John) = true
    chaining_e_to_n(he_a, John)
        agr(he_a, John) = true
        new_chain(he_a, John)
            new_chain: create John_b
            new_chain: create John_b^he_a
```

Chaining						
John	ϕ	**June**	**present**	**he**	**she**	**it**
$\mathtt{John}_{\mathtt{a}}$	$\phi_{\mathtt{a}}$	$\mathtt{June}_{\mathtt{a}}$	$\mathtt{present}_{\mathtt{a}}$	$\mathtt{he}_{\mathtt{a}}$	$\mathtt{she}_{\mathtt{a}}$	$\mathtt{it}_{\mathtt{a}}$
$\mathtt{John}_{\mathtt{b}}\hat{}\,\mathtt{he}_{\mathtt{a}}$	$\phi_{\mathtt{b}}\hat{}\,\mathtt{it}_{\mathtt{a}}$	$\mathtt{June}_{\mathtt{b}}\hat{}\,\mathtt{she}_{\mathtt{a}}$	$\mathtt{present}_{\mathtt{b}}\hat{}\,\mathtt{it}_{\mathtt{a}}$			
	$\phi_{\mathtt{c}}\hat{}\,\mathtt{she}_{\mathtt{a}}$					
	$\phi_{\mathtt{d}}\hat{}\,\mathtt{he}_{\mathtt{a}}$					

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(ϕ)
    non_refl_chaining(ϕ)
        chaining_n_to_n(ϕ, it)
            sc(ϕ, it) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(ϕ, she)
            sc(ϕ, she) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(ϕ, he)
            sc(ϕ, he) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(ϕ, present)
            sc(ϕ, present) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(ϕ, June)
            sc(ϕ, June) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(ϕ, John)
            sc(ϕ, John) = true
            agr(ϕ, John) = true
            rnr(ϕ, John) = true
            chaining_e_to_n(ϕ_a, John)
                agr(ϕ_a, John) = true
                new_chain(ϕ_a, John)
                    new_chain: create John_c
                    new_chain: create John_c^ϕ_a
```

Chaining						
John	ϕ	**June**	**present**	**he**	**she**	**it**
John$_a$ John$_b$^he$_a$ John$_c$^ϕ_a	ϕ_a ϕ_b^it$_a$ ϕ_c^she$_a$ ϕ_d^he$_a$	June$_a$ June$_b$^she$_a$	present$_a$ present$_b$^it$_a$	he$_a$	she$_a$	it$_a$

```
            new_chain: exiting
        chaining_e_to_n: exiting
        chaining_e_to_n(φb, John)
            agr(φb, John) = false
        chaining_e_to_n: exiting
        chaining_e_to_n(φc, John)
            agr(φc, John) = false
        chaining_e_to_n: exiting
        chaining_e_to_n(φd, John)
            agr(φd, John) = true
            new_chain(φd, John)
                new_chain: create Johnd
                new_chain: create Johnd^φd
```

Chaining						
John	ϕ	**June**	**present**	**he**	**she**	**it**
John_a	ϕ_a	June_a	present_a	he_a	she_a	it_a
$\text{John}_b\hat{}\text{he}_a$	$\phi_b\hat{}\text{it}_a$	$\text{June}_b\hat{}\text{she}_a$	$\text{present}_b\hat{}\text{it}_a$			
$\text{John}_c\hat{}\phi_a$	$\phi_c\hat{}\text{she}_a$					
$\text{John}_d\hat{}\phi_d$	$\phi_d\hat{}\text{he}_a$					

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
  non_refl_chaining: exiting
 chaining_n: exiting
chaining: exiting
```

Chaining						
John	ϕ	**June**	**present**	**he**	**she**	**it**
John_a	ϕ_a	June_a	present_a	he_a	she_a	it_a
$\text{John}_b\hat{}\text{he}_a$	$\phi_b\hat{}\text{it}_a$	$\text{June}_b\hat{}\text{she}_a$	$\text{present}_b\hat{}\text{it}_a$			
$\text{John}_c\hat{}\phi_a$	$\phi_c\hat{}\text{she}_a$					
$\text{John}_d\hat{}\phi_d$	$\phi_d\hat{}\text{he}_a$					

Interpretations		
$\text{John}_d\hat{}\phi_d\hat{}\text{he}_a$	$\text{June}_b\hat{}\text{she}_a$	$\text{present}_b\hat{}\text{it}_a$

(11.36) Ernie doesn't like Bernie, because he is such an asshole.

Features								
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**
Ernie	−	−	−	+	−	−	+	−
Bernie	−	−	−	+	−	−	+	−
he	+	−	−	+	−	−	+	−
asshole	−	−	−	+	−	?	+	−

```
chaining
    init_table
```

Chaining			
Ernie	**Bernie**	**he**	**asshole**
Ernie$_a$	Bernie$_a$	he$_a$	asshole$_a$

```
    init_table: exiting
    chaining_n(he)
        non_refl_chaining(he)
            chaining_n_to_n(he, asshole)
                sc(he, asshole) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(he, Bernie)
                sc(he, Bernie) = true
                agr(he, Bernie) = true
                rnr(he, Bernie) = true
                chaining_e_to_n(he_a, Bernie)
                    agr(he_a, Bernie) = true
                    new_chain(he_a, Bernie)
                        new_chain: create Bernie_b
                        new_chain: create Bernie_b^he_a
```

Chaining			
Ernie	**Bernie**	**he**	**asshole**
Ernie$_a$	Bernie$_a$ Bernie$_b$ˆhe$_a$	he$_a$	asshole$_a$

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(he, Ernie)
            sc(he, Ernie) = true
            agr(he, Ernie) = true
            rnr(he, Ernie) = true
            chaining_e_to_n(he_a, Ernie)
                agr(he_a, Ernie) = true
                new_chain(he_a, Ernie)
                    new_chain: create Ernie_b
                    new_chain: create Ernie_b^he_a
```

Chaining			
Ernie	**Bernie**	**he**	**asshole**
$Ernie_a$	$Bernie_a$	he_a	$asshole_a$
$Ernie_b$^he_a	$Bernie_b$^he_a		

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
  chaining_n: exiting
chaining: exiting
```

Chaining			
Ernie	**Bernie**	**he**	**asshole**
$Ernie_a$	$Bernie_a$	he_a	$asshole_a$
$Ernie_b$^he_a	$Bernie_b$^he_a		

Interpretations
$Ernie_b$^he_a
$Bernie_b$^he_a

12 Genitives

Very little modification to what has been said so far is necessary to implement attributive possessive pronouns. Recall that the attributive possessive pronouns are those pronouns listed in (12.1).

(12.1) my, our, your, her, his, its, their

Examining sentences like (12.2)-(12.5) reveals that reflexive pronouns don't chain to genitives within the same simplex. On the other hand, nonreflexive pronouns can.

(12.2) Mary's father killed himself.
(12.3) *Mary's father killed him.
(12.4) *Mary's father killed herself.
(12.5) Mary's father killed her.

The same conclusions also hold for **of-genitives**. Compare sentences (12.6)-(12.9) to (12.2)-(12.5).

(12.6) The father of Mary killed himself.
(12.7) *The father of Mary killed him.
(12.8) *The father of Mary killed herself.
(12.9) The father of Mary killed her.

The easiest way to handle genitives, apparently, is to introduce a new `Feature`, `GEN`, for genitive and to modify the Reflexive Nonreflexive Rule to handle genitives. The new form of the Reflexive Nonreflexive Rule is shown below in Figure 12.10.

```
function rnr(n1,n2:NodePointer):boolean;
    {Reflexive Nonreflexive Rule}
    var ftr1,ftr2:features;
begin
    n1:=n1^.np_link;
    n2:=n2^.np_link;
    ftr1:=n1^.ftr;
    ftr2:=n2^.ftr:
    if ftr2[GEN]=PLUS then rnr:=false
    else case ftr1[RPF] of
        PLUS:     rnr:=(n1^.up_link=n2^.up_link)
                        and (ftr1[GEN]==MINUS);
        QUESTION: {doesn't occur};
        MINUS:    rnr:=(n1^.up_link<>n2^.up_link)
                        or (ftr1[GEN]<>MINUS);
    end;
end;
```

Figure 12.10. New Reflexive Nonreflexive Rule

The examples following illustrate the interpretation of attributive possessive pronouns and pronouns in the context of genitives.

(12.11) Mary's mother cooks only for herself.

Features									
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**	**GEN**
Mary's	−	−	−	+	−	+	+	−	+
mother	−	−	−	+	−	+	+	−	−
herself	+	−	−	+	−	+	+	+	−

(12.12) Mary's mother cooks only for her.

Features									
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**	**GEN**
Mary's	−	−	−	+	−	+	+	−	+
mother	−	−	−	+	−	+	+	−	−
her	+	−	−	+	−	+	+	−	?

```
chaining
    init_table
```

Chaining		
Mary's	**mother**	**her**
Mary's$_a$	mother$_a$	her$_a$

```
    init_table: exiting
    chaining_n(her)
        non_refl_chaining(her)
            chaining_n_to_n(her, mother)
                sc(her, mother) = true
                agr(her, mother) = true
                rnr(her, mother) = true
                chaining_e_to_n(her_a, mother)
                    agr(her_a, mother) = true
                    new_chain(her_a, mother)
                        new_chain: create mother_b
                        new_chain: create mother_b^her_a
```

Chaining		
Mary's	**mother**	**her**
Mary's$_a$	mother$_a$ mother$_b$ˆher$_a$	her$_a$

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
        chaining_n_to_n(her, Mary's)
            sc(her, Mary's) = true
            agr(her, Mary's) = true
            rnr(her, Mary's) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
  chaining_n: exiting
chaining: exiting
```

Chaining		
Mary's	**mother**	**her**
$Mary's_a$	$mother_a$	her_a
	$mother_b$ˆher_a	

Interpretations
$mother_b$ˆher_a

(12.13) Mary's mother cooks only for her mother.

Features									
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**	**GEN**
Mary's	−	−	−	+	−	+	+	−	+
$mother_1$	−	−	−	+	−	+	+	−	−
her	+	−	−	+	−	+	+	−	?
$mother_2$	−	−	−	+	−	+	+	−	−

```
chaining
  init_table
```

Chaining			
Mary's	**$mother_1$**	**her**	**$mother_2$**
$Mary's_a$	$mother_{1a}$	her_a	$mother_{2a}$

```
    init_table: exiting
    chaining_n(her)
        non_refl_chaining(her)
            chaining_n_to_n(her, mother2)
                sc(her, mother2) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(her, mother1)
                sc(her, mother1) = true
                agr(her, mother1) = true
                rnr(her, mother1) = true
                chaining_e_to_n(hera, mother1)
                    agr(hera, mother1) = true
                    new_chain(hera, mother1)
                        new_chain: create mother1b
                        new_chain: create mother1b^hera
```

Chaining			
Mary's	**mother$_1$**	**her**	**mother$_2$**
Mary's$_a$	mother$_{1a}$ mother$_{1b}$^her$_a$	her$_a$	mother$_{2a}$

```
                    new_chain: exiting
                chaining_e_to_n: exiting
            chaining_n_to_n: exiting
            chaining_n_to_n(her, Mary's)
                sc(her, Mary's) = true
                agr(her, Mary's) = true
                rnr(her, Mary's) = false
            chaining_n_to_n: exiting
        non_refl_chaining: exiting
    chaining_n: exiting
chaining: exiting
```

Chaining			
Mary's	**mother$_1$**	**her**	**mother$_2$**
Mary's$_a$	mother$_{1a}$ mother$_{1b}$^her$_a$	her$_a$	mother$_{2a}$

Interpretations
mother$_{1b}$^her$_a$

13 Focusing

Extrasentential anaphora and **ellipsis** is possible through the maintenance of a **focus** of conversation. This maintenance is known as **focusing** and has been described at length by Grosz [10] and Sidner [32]. By focus of conversation, we mean the common view of the participants of conversation of what their conversation is about. Focusing is useful because it allows the participants of conversation to avoid redundant repetition of old material. Assuming focusing is desirable in a computer natural language system, how do we implement it?

Grosz has examined task dialogues in which an expert helps an apprentice to assemble a mechanical air compressor. She finds it convenient to represent the focus of conversation as a set of overlapping **focus spaces**, where each focus space is a collection of objects. One focus space is active and the others are open. When a focus space is no longer needed, it is closed. One of Grosz's assumptions is that goals and subgoals are definable and recognizable in a task dialogue system with the consequence that in any conversation there is an open focus space hierarchy with the active focus space at the bottom of the hierarchy.

Sidner has approached the problem of focusing from a different perspective by analyzing monologues. For Sidner, focus is kept track of through a **discourse focus**, **actor focus**, **potential discourse foci**, **potential actor foci**, **discourse focus stack**, and **actor focus stack**. Sidner's work, which came after Grosz's, is very commendable for the algorithms she presents, although most of these are fairly sketchy.

In our approach, we will treat the focus of conversation as a collection of nonpronominal N-nodes. Among the N-nodes that we would ordinairly expect to always be in focus are the $\underline{\text{I}}$ and $\underline{\text{you}}$ of a conversation. To get a handle on the focused N-nodes, we dominate them by an S-node just as if they all had occurred in one simplex. So, for example, if $\underline{\text{I}_0}$ and $\underline{\text{you}_0}$ are the nonpronominal N-nodes currently in focus, then the current focus representation is given by a structure like Figure 13.1.

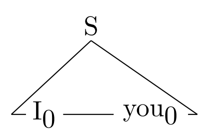
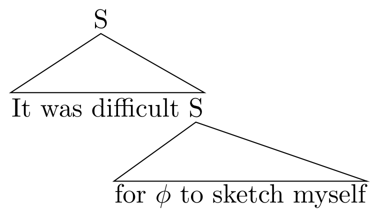


Figure 13.1. Typical Focus Representation

When it comes time to analyze a sentence, the current focus representation is attached to the C-S-N parse tree of the sentence via a C-node which *dominates* them both. This makes the focused N-nodes available to the N-nodes of the C-S-N parse tree for chaining.

As an example, suppose $\underline{\text{I}_0}$ and $\underline{\text{you}_0}$ are in focus and the current input sentence is (13.2) from Grinder [8].

(13.2) It was difficult to sketch myself.



The C-S-N parse tree of (13.2) will have a form like that indicated below in Figure 13.3.

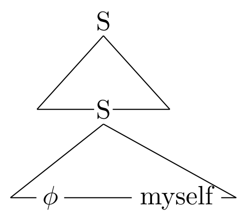


Figure 13.3. C-S-N Parse Tree of (13.2)

As the parse tree now stands in Figure 2, myself may chain to ϕ, but ϕ does not have an N-node to chain to. Thus, there are no legitimate interpretations without focusing. With focusing, the Parser attaches the current focus representation containing I_0 and you_0 to the C-S-N parse tree by a C-node obtaining the new C-S-N parse tree shown in Figure 13.4.

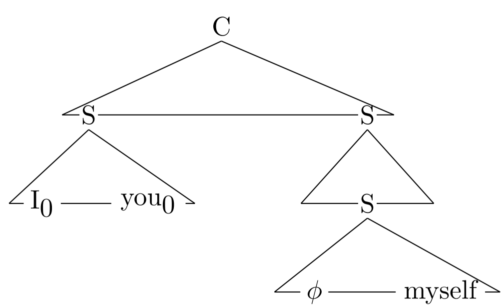


Figure 13.4. C-S-N Parse Tree with Focusing

Now, myself can chain from ϕ and ϕ can chain from I_0 giving us a legitimate interpretation of the C-S-N tree.

This kind of strategy explains a number of other examples from Grinder. Grinder lists (13.5a)-(13.11a) as grammatical.

(13.5a) It was difficult for me to sketch myself.

(13.6a) It was difficult for you to sketch yourself.
(13.7a) It was difficult for him to sketch himself.
(13.8a) It was difficult for her to sketch herself.
(13.9a) It was difficult for us to sketch ourselves.
(13.10a) It was difficult for you to sketch yourselves.
(13.11a) It was difficult for them to sketch themselves.

After Equi-NP Deletion, Grinder lists only (13.5b), (13.6b), (13.9b), and (13.10b) as grammatical.

(13.5b) It was difficult to sketch myself.
(13.6b) It was difficult to sketch yourself.
(13.7b) *It was difficult to sketch himself.
(13.8b) *It was difficult to sketch herself.
(13.9b) It was difficult to sketch ourselves.
(13.10b) It was difficult to sketch yourselves.
(13.11b) *It was difficult to sketch themselves.

The probable reason this comes about is that we are used to thinking of <u>I</u>, <u>you</u> singular, <u>us</u>, and <u>you</u> plural as always being in focus, while referents for <u>he</u>, <u>she</u>, and <u>they</u> are ordinairly not in focus. Needless to say, if referents for <u>he</u>, <u>she</u>, or <u>they</u> are in focus, the situation changes completely. This is shown by (13.12)-(13.14).

(13.12a) Nurse Bob Breezy gave up drawing.
(13.12b) [Bob] It was difficult to sketch himself.

(13.13a) Astronaut Linda Smith gave up drawing.
(13.13b) [Linda] It was difficult to sketch herself.

(13.14a) The bank embezzlers gave up drawing.
(13.14b) [the bank embezzlers] It was difficult to sketch themselves.

To indicate that various N-nodes are in focus, we bracket them at the beginning of a sentence. Thus (13.15)-(13.17) are not interpretable while (13.18)-(13.20) are.

(13.15) *It was difficult to sketch himself.

(13.16) *It was difficult to sketch herself.

(13.17) *It was difficult to sketch themselves.

(13.18) [Bob] It was difficult to sketch himself.

(13.19) [Linda] It was difficult to sketch herself.

(13.20) [the bank embezzlers] It was difficult to sketch themselves.

The following examples involve resolution through focusing.

(13.21) It was difficult to sketch myself.

Features									
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**	**GEN**
I_0	+	+	−	−	−	?	+	−	−
you_0	+	−	+	−	?	?	+	−	−
ϕ	+	?	?	?	?	?	?	−	−
myself	+	+	−	−	−	?	+	+	−

```
chaining
    init_table
```

Chaining			
I_0	**you_0**	**ϕ**	**myself**
I_{0a}	you_{0a}	ϕ_a	$myself_a$

```
    init_table: exiting
    chaining_n(myself)
        refl_chaining(myself)
            simplex_pred(myself)
            simplex_pred:  φ
            chaining_n_to_n(myself, φ)
                sc(myself, φ) = true
                agr(myself, φ) = true
                rnr(myself, φ) = true
                chaining_e_to_n(myself_a, φ)
                    agr(myself_a, φ) = true
                    new_chain(myself_a, φ)
                        new_chain: create φ_b
                        new_chain: create φ_b^myself_a
```

Chaining			
I_0	**you_0**	**ϕ**	**myself**
I_{0a}	you_{0a}	ϕ_a ϕ_bˆ$myself_a$	$myself_a$

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    simplex_pred(φ)
    simplex_pred:
refl_chaining: exiting
chaining_n: exiting
chaining_n(φ)
    non_refl_chaining(φ)
        chaining_n_to_n(φ, myself)
            sc(φ, myself) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, you0)
            sc(φ, you0) = true
            agr(φ, you0) = true
            rnr(φ, you0) = true
            chaining_e_to_n(φa, you0)
                agr(φa, you0) = true
                new_chain(φa, you0)
                    new_chain: create you0b
                    new_chain: create you0b^φa
```

Chaining			
I0	**you0**	**φ**	**myself**
I0a	you0a you0b^φa	φa φb^myselfa	myselfa

```
            new_chain: exiting
        chaining_e_to_n: exiting
        chaining_e_to_n(φb, you0)
            agr(φb, you0) = false
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    chaining_n_to_n(φ, I0)
        sc(φ, I0) = true
        agr(φ, I0) = true
        rnr(φ, I0) = true
        chaining_e_to_n(φa, I0)
            agr(φa, I0) = true
            new_chain(φa, I0)
                new_chain: create I0b
                new_chain: create I0b^φa
```

Chaining			
I_0	**you_0**	**ϕ**	**myself**
I_{0a}	you_{0a}	ϕ_a	$myself_a$
I_{0b}ˆϕ_a	you_{0b}ˆϕ_a	ϕ_bˆ$myself_a$	

```
    new_chain: exiting
chaining_e_to_n: exiting
chaining_e_to_n(φb, I0)
    agr(φb, I0) = true
    new_chain(φb, I0)
        new_chain: create I0c
        new_chain: create I0cˆφb
```

Chaining			
I_0	**you_0**	**ϕ**	**myself**
I_{0a}	you_{0a}	ϕ_a	$myself_a$
I_{0b}ˆϕ_a	you_{0b}ˆϕ_a	ϕ_bˆ$myself_a$	
I_{0c}ˆϕ_b			

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(you0)
    non_refl_chaining(you0)
        chaining_n_to_n(you0, myself)
            sc(you0, myself) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(you0, φ)
            sc(you0, φ) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(you0, I0)
            sc(you0, I0) = true
            agr(you0, I0) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(I0)
    non_refl_chaining(I0)
        chaining_n_to_n(I0, myself)
            sc(I0, myself) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I0, φ)
            sc(I0, φ) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I0, you0)
            sc(I0, you0) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining: exiting
```

Chaining			
I_0	**you_0**	**ϕ**	**myself**
I_{0a}	you_{0a}	ϕ_a	$myself_a$
I_{0b}^ϕ_a	you_{0b}^ϕ_a	ϕ_b^$myself_a$	
I_{0c}^ϕ_b			

Interpretations
I_{0c}^ϕ_b^$myself_a$

(13.22) [toy] Give me that!

Features									
	PNF	**FPF**	**SPF**	**TPF**	**PLF**	**GNF**	**ANF**	**RPF**	**GEN**
I_0	+	+	−	−	−	?	+	−	−
you_0	+	−	+	−	?	?	+	−	−
toy	−	−	−	+	−	?	?	−	−
ϕ	+	?	?	?	?	?	?	−	−
me	+	+	−	−	−	?	+	−	−
that	+	−	−	+	−	?	?	−	−

```
chaining
    init_table
```

Chaining					
I_0	**you_0**	**toy**	**ϕ**	**me**	**that**
I_{0a}	you_{0a}	toy_a	ϕ_a	me_a	$that_a$

```
    init_table: exiting
    chaining_n(that)
        non_refl_chaining(that)
            chaining_n_to_n(that, me)
                sc(that, me) = true
                agr(that, me) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(that, φ)
                sc(that, φ) = true
                agr(that, φ) = true
                rnr(that, φ) = false
            chaining_n_to_n: exiting
            chaining_n_to_n(that, toy)
                sc(that, toy) = true
                agr(that, toy) = true
                rnr(that, toy) = true
                chaining_e_to_n(that_a, toy)
                    agr(that_a, toy) = true
                    new_chain(that_a, toy)
                        new_chain: create toy_b
                        new_chain: create toy_b^that_a
```

Chaining					
I_0	**you_0**	**toy**	**ϕ**	**me**	**that**
I_{0a}	you_{0a}	toy_a toy_b^$that_a$	ϕ_a	me_a	$that_a$

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
    chaining_n_to_n(that, you0)
        sc(that, you0) = true
        agr(that, you0) = false
    chaining_n_to_n: exiting
    chaining_n_to_n(that, I0)
        sc(that, I0) = true
        agr(that, I0) = false
    chaining_n_to_n: exiting
non_refl_chaining: exiting
chaining_n: exiting
chaining_n(me)
non_refl_chaining(me)
    chaining_n_to_n(me, that)
        sc(me, that) = false
    chaining_n_to_n: exiting
    chaining_n_to_n(me, φ)
        sc(me, φ) = true
        agr(me, φ) = true
        rnr(me, φ) = false
    chaining_n_to_n: exiting
    chaining_n_to_n(me, toy)
        sc(me, toy) = true
        agr(me, toy) = false
    chaining_n_to_n: exiting
    chaining_n_to_n(me, you0)
        sc(me, you0) = true
        agr(me, you0) = false
    chaining_n_to_n: exiting
    chaining_n_to_n(me, I0)
        sc(me, I0) = true
        agr(me, I0) = true
        rnr(me, I0) = true
        chaining_e_to_n(mea, I0)
            agr(mea, I0) = true
            new_chain(mea, I0)
                new_chain: create I0b
                new_chain: create I0b^mea
```

Chaining					
I_0	**you_0**	**toy**	**ϕ**	**me**	**that**
I_{0a} I_{0b}^me_a	you_{0a}	toy_a toy_b^$that_a$	ϕ_a	me_a	$that_a$

```
            new_chain: exiting
        chaining_e_to_n: exiting
    chaining_n_to_n: exiting
non_refl_chaining: exiting
chaining_n: exiting
chaining_n(φ)
    non_refl_chaining(φ)
        chaining_n_to_n(φ, that)
            sc(φ, that) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, me)
            sc(φ, me) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(φ, toy)
            sc(φ, toy) = true
            agr(φ, toy) = true
            rnr(φ, toy) = true
            chaining_e_to_n(φa, toy)
                agr(φa, toy) = true
                new_chain(φa, toy)
                    new_chain: create toyc
                    new_chain: create toyc^φa
```

Chaining					
I_0	**you_0**	**toy**	**ϕ**	**me**	**that**
I_{0a} I_{0b}^me_a	you_{0a}	toy_a toy_b^$that_a$ toy_c^ϕ_a	ϕ_a	me_a	$that_a$

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(φ, you0)
    sc(φ, you0) = true
    agr(φ, you0) = true
    rnr(φ, you0) = true
    chaining_e_to_n(φa, you0)
        agr(φa, you0) = true
        new_chain(φa, you0)
            new_chain: create you0b
            new_chain: create you0b^φa
```

Chaining					
I0	**you0**	**toy**	**φ**	**me**	**that**
I0a I0b^mea	you0a you0b^φa	toya toyb^thata toyc^φa	φa	mea	thata

```
        new_chain: exiting
    chaining_e_to_n: exiting
chaining_n_to_n: exiting
chaining_n_to_n(φ, I0)
    sc(φ, I0) = true
    agr(φ, I0) = true
    rnr(φ, I0) = true
    chaining_e_to_n(φa, I0)
        agr(φa, I0) = true
        new_chain(φa, I0)
            new_chain: create I0c
            new_chain: create I0c^φa
```

Chaining					
I0	**you0**	**toy**	**φ**	**me**	**that**
I0a I0b^mea I0c^φa	you0a you0b^φa	toya toyb^thata toyc^φa	φa	mea	thata

```
                new_chain: exiting
            chaining_e_to_n: exiting
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(you0)
    non_refl_chaining(you0)
        chaining_n_to_n(you0, that)
            sc(you0, that) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(you0, me)
            sc(you0, me) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(you0, φ)
            sc(you0, φ) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(you0, toy)
            sc(you0, toy) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(you0, I0)
            sc(you0, I0) = true
            agr(you0, I0) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining_n(I0)
    non_refl_chaining(I0)
        chaining_n_to_n(I0, that)
            sc(I0, that) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I0, me)
            sc(I0, me) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I0, φ)
            sc(I0, φ) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I0, toy)
            sc(I0, toy) = false
        chaining_n_to_n: exiting
        chaining_n_to_n(I0, you0)
            sc(I0, you0) = false
        chaining_n_to_n: exiting
    non_refl_chaining: exiting
chaining_n: exiting
chaining: exiting
```

Chaining					
I_0	**you_0**	**toy**	**ϕ**	**me**	**that**
I_{0a} I_{0b}^me_a I_{0c}^ϕ_a	you_{0a} you_{0b}^ϕ_a	toy_a toy_b^$that_a$ toy_c^ϕ_a	ϕ_a	me_a	$that_a$

Interpretations
you_{0b}^ϕ_a toy_b^$that_a$
I_{0b}^me_a you_{0b}^ϕ_a toy_b^$that_a$
I_{0c}^ϕ_a toy_b^$that_a$

References

[1] David Bloom and David G. Hayes. “Designation in English”. In: *Anaphora in Discourse*. Ed. by John Hinds. Edmonton, Alberta, Canada: Linguistic Research, Inc., 1978.

[2] Leonard Bloomfield. *Language*. New York: Holt, Rinehart and Winston, Inc., 1933.

[3] Joan Bresnan. “A Note on the Notion ’Identity of Sense Anaphora’”. In: *Linguistic Inquiry* 2.4 (1971), pp. 589–596.

[4] Shuji Chiba. “A Note on Equi-NP Deletion”. In: *Linguistic Inquiry* 2.4 (1971), pp. 539–540.

[5] Noam Chomsky. *Syntactic Structures*. The Hague, The Netherlands: Mouton & Co., Publishers, 1957.

[6] Gareth Evans. “Pronouns, Quantifiers, and Relative Clauses (I)”. In: *Canadian Journal of Philosophy* 7.3 (1977), pp. 467–536.

[7] Gilles Fauconnier. “Do Quantifiers Branch?” In: *Linguistic Inquiry* 6.4 (1975), pp. 555–567.

[8] John Grinder. “Chains of Coreference”. In: *Linguistic Inquiry* 2.2 (1971), pp. 183–202.

[9] Alexander Grosu. “On the Nonunitary Nature of the Coordinate Structure Constraint”. In: *Linguistic Inquiry* 4.1 (1973), pp. 88–92.

[10] Barbara J. Grosz. *The Representation and Use of Focus in Dialogue Understanding*. Technical Note 151. Menlo Park, California: Stanford Research Institute, 1977.

[11] Jorge Hankamer and Ivan Sag. “Deep and Surface Anaphora”. In: *Linguistic Inquiry* 7.3 (1976), pp. 391–426.

[12] Charles F. Hockett. *A Course in Modern Linguistics*. New York: The Macmillan Company, 1958.

[13] Rodney Huddleston. “A Survey of the Crossing-Coreference Controversy”. In: *Papers in Linguistics* 11.3-4 (1978), pp. 295–319.

[14] Ray S. Jackendoff. “Any Vs. Every”. In: *Linguistic Inquiry* 3.1 (1972), pp. 119–120.

[15] Ray S. Jackendoff. *Semantic Interpretation in Generative Grammar*. Cambridge, Massachusetts: MIT Press, 1972.

[16] Susumu Kuno. “Lexical and Contextual Meaning”. In: *Linguistic Inquiry* 5.3 (1974), pp. 469–477.

[17] Susumu Kuno. “Pronominalization, Reflexivization, and Direct Discourse”. In: *Linguistic Inquiry* 3.2 (1972), pp. 161–195.

[18] George Lakoff and John R. Ross. “A Note on Anaphoric Islands and Causatives”. In: *Linguistic Inquiry* 3.1 (1972), pp. 121–125.

[19] Ronald W. Langacker. “On Pronominalization and the Chain of Command”. In: *Modern Studies in English*. Ed. by David A. Reible and Sanford A. Schane. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1969.

[20] D. Terence Langendoen. *Essentials of English Grammar*. New York: Holt, Rinehart and Winston, Inc., 1970.

[21] Robert B. Lees. *The Grammar of English Nominalizations*. The Hague, The Netherlands: Mouton & Co. Publishers, 1963.

[22] Robert B. Lees and Edward S. Klima. “Rules for English Pronominalization”. In: *Language* 39.1 (1963), pp. 17–28.

[23] Paul M. Postal. “On Coreferential Complement Subject Deletion”. In: *Linguistic Inquiry* 1.4 (1970), pp. 439–500.

[24] Paul M. Postal. “On So-Called ‘Pronouns’ in English”. In: *The 19th Monograph on Language and Linguistics*. Ed. by F. Dinneen. Washington, D.C.: Georgetown University Press, 1966.

[25] Randolph Quirk and Sidney Greenbaum. *A Concise Grammar of Contemporary English*. New York: Harcourt Brace Jovanovich, Inc., 1973.

[26] Kelly Roach. *Kelly Roach's Caltech M.S. Thesis*. URL: https://www.planetquantum.com/Pronouns/Index.htm.

[27] Kelly Roach. *Pronouns*. 1988. DOI: 10.7907/mf427-dra49.

[28] Paul Roberts. *Modern Grammar*. New York: Harcourt, Brace & World, Inc., 1967.

[29] John R. Ross. “On the Cyclic Nature of English Pronominalization”. In: *To Honor Roman Jakobson*. Vol. III. The Hague, The Netherlands: Mouton & Co., 1967.

[30] Ivan A. Sag. “The Nonunity of Anaphora”. In: *Linguistic Inquiry* 10.1 (1979), pp. 152–164.

[31] Mario Saltarelli. “Focus on Focus: Propositional Generative Grammar”. In: *Studies Presented to Robert B. Lees by His Students*. Ed. by Jerrold M. Sadock and Anthony L. Vanek. Edmonton, Alberta, Canada: Linguistic Research, Inc., 1970.

[32] Candace L. Sidner. *Towards a Computational Theory of Definite Anaphor Comprehension in English Discourse*. Technical Report 537. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory, 1979.

[33] Carlota S. Smith. “Ambiguous Sentences with And”. In: *Modern Studies in English*. Ed. by David A. Reibel and Sanford R. Schane. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1969.

[34] Douglas Smith. *Frederick B. Thompson*. 2014. URL: https://www.caltech.edu/about/news/frederick-b-thompson-43160.

[35] Thomas Wasow. “Anaphoric Pronouns and Bound Variables”. In: *Language* 51.2 (1975), pp. 368–383.

[36] Sayo Yotsukura. *The Articles in English*. The Hague, The Netherlands: Mouton & Co., Printers, 1970.

Index