An Introduction to the Sundance and AutoSlog Systems

Ellen Riloff and William Phillips

UUCS-04-015

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

November 8, 2004

Abstract

This document describes the Sundance natural language processing system that has been developed at the University of Utah, as well as the AutoSlog and AutoSlog-TS extraction pattern learners that have been implemented on top of Sundance. Sundance is a shallow parser that also includes clause handling capabilities and an information extraction engine. AutoSlog and AutoSlog-TS are learning algorithms that can be used to generate extraction patterns automatically. This report includes: (1) a high-level overview of Sundance's design and capabilities, (2) a detailed description of the steps that Sundance performs when analyzing sentences, (3) a discussion of the capabilities of the AutoSlog and AutoSlog-TS learning algorithms, and (4) guidance on how to install, use, and customize the Sundance and AutoSlog systems for new users and applications.

Contents

1 Introduction

For several years now, the natural language processing group at the University of Utah has been developing an in-house natural language processing (NLP) system called Sundance. Sundance was originally developed to provide basic sentence processing and information extraction capabilities for the NLP research efforts at Utah. We never intended to make it publicly available – in fact, it never occurred to us that anyone else would want to use it. However, over the years we've had numerous requests for Sundance and have come to appreciate that it may be of value to other research groups as well. So, we have (rather belatedly) written this report to provide an overview of Sundance for people who wish to use it.

Sundance was designed to be a comprehensive shallow parser that performs partial parsing as well as some additional sentence processing needed for information extraction. As a result, Sundance does substantially more than a typical shallow parser, but it still adheres to the shallow parsing paradigm (i.e., there is no attempt to generate full parse trees or handle deeply nested structures). Shallow parsing is less ambitious than full parsing, but it is generally faster and more robust. Shallow parsers have been widely used for information extraction systems because they provide the speed and robustness needed to process large volumes of (sometimes ungrammatical) text and because partial parsing seems to be largely adequate for such tasks (Grishman, 1995).

In addition to parsing, Sundance also has the ability to activate and instantiate case frames for information extraction. Throughout this document, we will describe Sundance as having two separate functions: parsing and information extraction. This report also describes the AutoSlog and AutoSlog-TS systems, which are extraction pattern learning algorithms that have been implemented on top of Sundance and are included in the Sundance distribution package. Below, we summarize our NLP systems:

The Sundance Parser: Sundance performs a variety of syntactic functions, including sentence segmentation, morphological analysis, part-of-speech disambiguation, syntactic chunking, named entity recognition, syntactic role assignment (subject, direct object, and indirect object), clause segmentation, and subject inference.

The Sundance Information Extraction Engine: Sundance can activate case frame structures and instantiate them with information from a sentence. The case frames can be simple single-slot extraction patterns or more complex frame structures with multiple slots, thematic roles, and selectional restrictions.

The AutoSlog Extraction Pattern Generator: AutoSlog (Riloff, 1993; Riloff, 1996a) is a supervised learning algorithm for extraction pattern generation. As input, AutoSlog requires an annotated training corpus or (in our implementation) a list of nouns representing items that are targeted for extraction. The extraction patterns that AutoSlog learns are simple case frames that have just a single slot and no selectional restrictions. Thematic roles and selectional restrictions can be easily added by hand.

The AutoSlog-TS Extraction Pattern Generator: AutoSlog-TS (Riloff, 1996b) is a weakly supervised learning algorithm for extraction pattern generation. As input, AutoSlog-TS requires

one set of relevant texts and one set of irrelevant texts. The learner applies AutoSlog's heuristics exhaustively to generate every extraction pattern that appears in the corpus. It then computes statistics and ranks the extraction patterns based upon their strength of association with the relevant texts.

We have written this document to serve both as a general reference that explains how Sundance and AutoSlog work and as a technical manual that explains how to use them. In Section 2, we begin by providing a high-level overview of Sundance's design. In Section 3, we step through the various processes involved in Sundance's parsing mechanism. In Section 4, we describe Sundance's information extraction capabilities and explain how the AutoSlog and AutoSlog-TS learning algorithms can be used to generate extraction patterns. Section 5 gives instructions on how to install and use the Sundance and AutoSlog systems, and it provides some guidance on how Sundance and AutoSlog can be customized for new domains. The current version of Sundance at the time of this writing is v4.2.

2 An Overview of Sundance

Sundance[1] is a shallow parser that was specifically designed to support information extraction. Sundance is organized as a multi-layered architecture that includes three levels of sentence analysis: basic syntactic analysis, clause handling, and information extraction. Figure 1 shows the subprocesses involved at each level. The syntactic analysis functions perform low-level syntactic processing, including sentence segmentation, morphological analysis, part-of-speech disambiguation, syntactic chunking, and named entity recognition. The clause handling functions support clause recognition, syntactic role assignment, relative pronoun resolution, and subject inference. The information extraction functions perform case frame activation and instantiation.

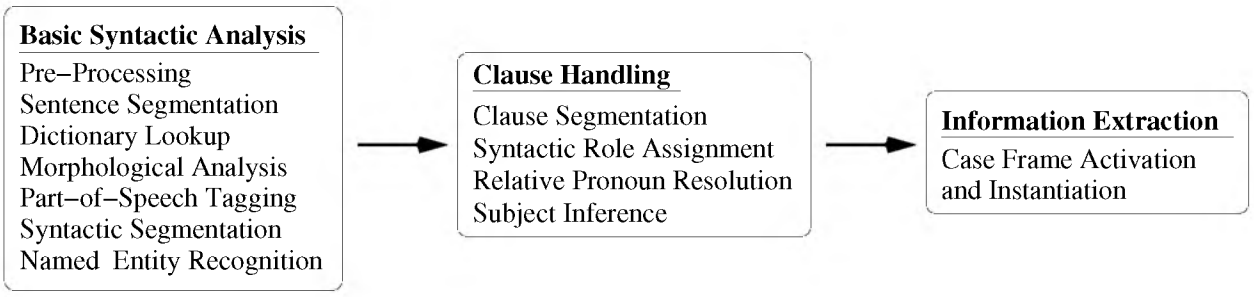

Figure 1: Sundance's sentence analysis processes

The motivation for Sundance's layered design is the idea of propagating constraints and preferences from the lowest level of analysis to the highest. Intuitively, Sundance makes the relatively easy decisions first and then uses those decisions to constrain the possibilities for the more difficult decisions.[2] Sundance does not use an explicit grammar, but instead relies on heuristics that represent syntactic (and sometimes semantic) constraints and preferences. The heuristics are domain-independent but are specific to the English language.

As Figure 1 shows, Sundance has more capabilities than most shallow parsers. Sundance is similar in spirit to CASS (Abney, 1990b; Abney, 1990a), which is also a partial parser that uses an incremental "easy-first" parsing strategy. However, Sundance does more linguistic analysis than CASS. As input, CASS takes a sentence that has already been assigned correct part-of-speech tags and has been bracketed for non-recursive NPs using Church's POS tagger (Church, 1989). In contrast, Sundance does its own part-of-speech disambiguation and NP bracketing, as well as clause segmentation, syntactic role assignment for subjects, direct objects, and indirect objects, relative pronoun resolution, and subject inference. Fidditch (Hindle, 1994) also goes beyond syntactic segmentation and identifies subjects and objects, resolves relative pronouns, and performs subject inference. But like CASS, Fidditch relies on a separate POS tagger to disambiguate parts-of-speech before its parsing begins, and it does not include information extraction capabilities.

[1] Sundance is an (admittedly forced) acronym for "Sentence UNDerstanding ANd Concept Extraction".

[2] This pipelined architecture also allows Sundance to perform variable-depth language processing by applying only the layers that are necessary for a task. For example, if an application only requires NP chunking, then only the syntactic analysis functions up to and including syntactic segmentation need to be performed. In principle, this design allows Sundance to run faster for applications that don't need all of its capabilities. In practice, however, Sundance is fast enough that we have never needed to turn off any of its subprocesses.

FASTUS (Appelt et al., 1993) is a partial parser that was also designed for information extraction. FASTUS uses cascaded finite-state machines to perform syntactic segmentation, but does not do as much clause-level analysis as Sundance. In contrast to Sundance's case frame representation, FASTUS uses "pseudo-syntax" to recognize regular expression extraction patterns consisting of syntactic segments.

Sundance is a complete NLP system that does everything from low-level morphological analysis to syntactic segmentation and clause-level handling as well as case frame instantiation for information extraction. In our research, Sundance has proved to be very robust when faced with ungrammatical or ill-formed input. For example, we have used Sundance extensively to process web pages (Furnkranz et al., 1998; Riloff and Jones, 1999), which often contain sentence fragments and text snippets that are not grammatically well-formed. Sundance attempts to parse whatever it receives and produces a shallow parse. Although it is not perfect, of course, in most cases it seems to do a reasonably good job of producing a plausible syntactic analysis. In the next section, we explain in detail the various steps that Sundance goes through when processing sentences.

3 The Sundance Parser

The purpose of this section is to describe Sundance's parsing mechanism in detail so that users can get a sense for what Sundance can do, what it can not do, and what its output looks like. Throughout this section, we also indicate which steps use external dictionaries or lists that can be easily modified for new applications.

3.1 Pre-Processing

Sundance begins by pushing each text through a pre-processor that tokenizes and normalizes the text. When appropriate, trailing punctuation marks are separated from adjacent words and replaced by special punctuation symbols (e.g., "," becomes ">COMMA"). Numbers are normalized by putting a special && symbol in front of them. Some written numbers are also converted to their numeric form (e.g., "three" becomes "&&3"). Possessives are normalized by removing the apostrophe and replacing it with an @ sign both in front of the word and in the position where the apostrophe was (e.g., "Fred's" becomes "@Fred@s").

Next, a phrasal lexicon is consulted to recognize phrases that should be treated as a single lexical item. For example, when the phrase "as well as" appears in a text, the three words are slammed together with underscores to become a single lexical item: "as_well_as". This helps to simplify parsing in cases where the lexicalized phrase can be assigned a single part-of-speech tag[3] (e.g., "as_well_as" is defined in Sundance's dictionary as a CONJUNCTION).

The phrasal lexicon is also used to expand contractions. To handle contractions, the phrasal lexicon expands a single word ("can't") into multiple known words ("can not"). This is in contrast to most other situations where the phrasal lexicon merges multiple words into a single lexicalized term. Using the phrasal lexicon to expand contractions obviates the need to define contractions explicitly in the dictionary. The phrasal lexicon can be easily augmented or modified without having to recompile Sundance. (see Section 7).

3.2 Sentence Segmentation

Given a document, Sundance first separates the text into sentences. Question marks, exclamation points, and semi-colons are fairly reliable end-of-sentence indicators by themselves, so Sundance simply separates sentences when one of these tokens is found.[4] Periods, however, are highly ambiguous. Abbreviations that end with a period may or may not indicate that the end of a sentence has been reached. For example, the word "U.S." sometimes appears at the end of a sentence and sometimes in the middle. To determine sentence boundaries when a period is found, Sundance uses end-of-sentence (EOS) heuristics.

First, Sundance maintains a list of common abbreviations called *non-EOS words*. Sundance's initial assumption is that these abbreviations do not represent the end of a sentence. However, this assumption can be overridden by subsequent heuristics that look at the word following the abbreviation. Sundance also maintains a list called *EOS followers* that consists of words that

[3]This also allows lexicalized phrases to be assigned semantic tags in the dictionary. For example, the phrase "Fort_Worth" can be labeled as a LOCATION in the dictionary, even though the words "fort" and "worth" independently should not have a LOCATION semantic tag.

[4]Sundance treats semi-colons as end-of-sentence markers because, in our experience, they usually separate word sequences that can be parsed as relatively independent sentence structures.

commonly begin a sentence, especially when the word begins with a capitalized letter (e.g., "The"). If the word following the abbreviation is in the EOS followers list, then Sundance infers that the abbreviation is in fact the end of the sentence. For example, if "U.S." is followed by a capitalized article such as "The", then Sundance assumes "U.S." is the end of the sentence. The lists of *non-EOS words* and *EOS followers* can be easily augmented or modified without having to recompile Sundance.

3.3 Dictionary Lookup

Sundance relies on manually defined dictionaries to get the basic syntactic and (optionally) semantic properties of a word. The dictionaries are not case-sensitive, so a capitalized entry will be considered to be the same word as a lower case entry. Each word in the dictionary is defined with all possible parts-of-speech that it can take. By default, all words are assumed to be in their root form in the dictionary. A morphological analyzer (described in Section 3.4) is used to automatically derive the morphological variants of words in the dictionary. Irregular morphological variants must be explicitly defined in the dictionary and their root forms must be specified.

Sundance uses a small set of approximately 20 part-of-speech tags. If a word can have more than one part-of-speech, then each possible part-of-speech should be entered on a separate line of the dictionary. By default, nouns are assumed to be singular and verbs are assumed to be in their base (root) form. Plural nouns can be defined using a "plural" specifier following the part-of-speech tag. Verb tenses can be defined with one of the following tense specifiers: (*past, present, participle*). The tense specifier should immediately follow the word's part-of-speech tag. A verb's root can be indicated by appending it to the word with a colon as a separator. For example, the verb *swam* is defined as the past tense of the verb *swim* with this dictionary entry: **SWAM:SWIM VERB PAST**. If no root is specified, then the word is assumed to be in a root form already. For example, the word "swim" would be defined like so: **SWIM VERB**.

A word can also be assigned a semantic class, if desired. Sundance uses a general-purpose semantic hierarchy by default, but specialized semantic hierarchies can be easily created for new domains (see Section 5.5). Each word in the dictionary can be assigned a semantic class from this hierarchy by adding the semantic class to the end of the word's dictionary entry. For example, the following dictionary entry will define the word "Friday" as belonging to the semantic class TIME: **FRIDAY NOUN TIME**. Semantic class labels are sometimes used by Sundance during parsing (e.g., TIME expressions are handled specially), but they are primarily used as selectional restrictions in case frames during information extraction. The dictionaries and semantic hierarchy can be easily augmented or modified without having to recompile Sundance.

3.4 Morphological Analysis

Given a word, Sundance first looks up the word in its dictionaries. If the word is present, then all parts-of-speech defined for it are gathered and considered during parsing. If a word is not found, then Sundance invokes a rule-based morphological analyzer to try to derive it. If successful, the morphological analyzer returns all parts-of-speech that are possible for the word, along with the root for each possible derivation. If morphology is unsuccessful, then the word is assigned an *unknown* part-of-speech tag. The morphological analysis rules can be easily augmented or modified without having to recompile Sundance.

3.5 Part-of-Speech Tagging (or a lack thereof)

A somewhat unusual aspect of Sundance is that it does not have an explicit part-of-speech tagger and it really doesn't consider part-of-speech tagging to be a distinct task. Initially, a word is assigned all parts-of-speech defined for it in Sundance's dictionaries, or all parts-of-speech assigned to it by the morphological analyzer. Many words are assigned multiple part-of-speech tags. Then Sundance goes about its business of syntactic segmentation, disambiguating part-of-speech tags along the way. Consequently, part-of-speech disambiguation is intermingled with parsing.

Given multiple part-of-speech tags for a word, Sundance uses heuristics to decide which tag(s) are most likely to be correct given the surrounding context. Some parsing heuristics remove inappropriate tags, while others select a particular tag from the set. Once Sundance determines what syntactic constituent a word belongs to, then it is satisfied and does not try to disambiguate part-of-speech assignments any further. Consequently, Sundance will often leave multiple part-of-speech tags assigned to a word. This is especially common with words that can be used as an adjective or a noun, such as *"light"*. If Sundance determines that *"light"* belongs inside an NP (as in *"a light fixture"*), then it creates the NP but does not attempt to determine whether *"light"* is functioning as an adjective or a noun in that NP. Both part-of-speech tags remain assigned to the word.

One side effect of this design decision, coupled with some careless programming, is that sometimes multiple part-of-speech tags are left assigned to a word even though it is clear which one Sundance decided is correct. For example, in the sentence *"The table is light"*, Sundance properly parses *"light"* as an adjectival phrase. However, both the adjective and noun tags remain assigned to it. Similarly, in the sentence *"I have a light"*, Sundance properly parses *"a light"* as a noun phrase, but both the adjective and noun tags remain assigned to it. This behavior stems from our emphasis on parsing rather than part-of-speech tagging per se. This behavior is also apparent with unknown words, which are initially assigned an *unknown* POS tag but eventually may be parsed as (say) a noun phrase or a verb phrase. In these cases, Sundance has implicitly decided that this word is (say) a noun or a verb, but the part-of-speech tag will still be listed as *unknown*. We hope to remedy this in future versions of Sundance, but in the meantime we advise Sundance users to focus on the syntactic constituents assigned to a word rather than the part-of-speech tags themselves. There are also some internal Sundance functions that can be used to determine how a word was recognized by Sundance (e.g., the "get_head_noun" function will return the head noun of an NP, regardless of what part-of-speech tags are listed in its tag set).

Another important aspect of Sundance's design is that the parsing heuristics have free rein to override the part-of-speech tags assigned to a word in the dictionary or by morphology. This is done very conservatively, for obvious reasons. But if a word simply must belong to a certain part-of-speech class given the words around it, then the parsing heuristics will force it to have that POS tag. For example, suppose the word *"shoot"* is defined only as a verb in the dictionary. Given the sentence *"The shoot was good."*, Sundance will infer that *"shoot"* cannot be a verb in this sentence because it immediately follows an article, *"the"*. Sundance will then override the dictionary, inferring that *"the shoot"* must be an NP and therefore *"shoot"* must be a noun. We have found this behavior to be highly reliable in Sundance and it often helps to compensate for omissions in the dictionary and with the parsing of slang expressions.

In the parse output, there are four labels that reveal how part-of-speech tags were assigned to

a word. LEX means that the part-of-speech tags were found in the dictionary. MOR means that the part-of-speech tags were derived via morphological analysis. INF-LEX means that the part-of-speech tags were assigned by grammatical inference heuristics. For example, the previous example where "shoot" is inferred to be a noun is due to a grammatical inference heuristic. INF-MOR means that the part-of-speech tags were assigned by morphological inference heuristics. These heuristics simply look for common suffixes that are virtually a dead give-away to a word's part-of-speech. For example, Sundance infers that unknown words ending in "tion" are nouns, which is almost always the case. The morphological inference heuristics kick in only as a last resort after true morphological analysis has failed. Both the INF-LEX and INF-MOR heuristics can override the dictionary and the morphological analyzer, and can also infer POS tags for unknown words.

3.6 Syntactic Segmentation

Sundance's syntactic segmentation phase recognizes noun phrases (NPs), prepositional phrases (PPs), verb phrases (VPs), and adjectival phrases (ADJPs). Sundance performs multiple passes over each sentence, beginning with the most reliable segmenters (NP and PP chunking) and using their output to constrain the possibilities for the remaining segmenters (VP and ADJP chunking). The motivation behind this approach is best illustrated with an example. Consider the following sentence:

(S1) The foo blorched a wumpy baz.

This sentence is nonsensical semantically, yet in our experience, people have no trouble parsing it. People seem to assign a grammatical structure to this sentence by recognizing grammatical constraints and morphological clues. People probably apply some of the following heuristics. (1) *foo* and *wumpy* must be part of a noun phrase because they are preceded by an article. Therefore they are probably adjectives or nouns. (2) *blorched* is a verb because it ends in "ed", and also perhaps because of its position in the sentence (i.e., it is in the middle of the sentence and surrounded by noun phrases). (3) *wumpy* is probably an adjective because it ends in "y" and is sandwiched between an article and a possible noun. (4) *baz* is probably a noun because it is preceded by a likely adjective and ends the sentence. All of these heuristics could be wrong, but together they support one another to produce a compelling picture of the sentence structure. If this sentence seems contrived, consider a more realistic sentence that we will use to demonstrate how Sundance incrementally parses a sentence:

(S2) The armed man shot the bear with a gun.

After part-of-speech assignments from the dictionary, $S2$ looks like this:

The	armed	man	shot	the	bear	with	a	gun
art	*adj/verb*	*noun/verb*	*adj/noun/verb*	*art*	*noun/verb*	*prep*	*art*	*noun/verb*

First, Sundance begins with NP segmentation. Sundance gathers up words that are likely to form simple (base) noun phrases. NP segmentation uses relatively conservative heuristics because its commitments limit the options available to the subsequent segmenters. For example, one heuristic creates an NP when it sees an article. First, it grabs the word following the article and adds it to the NP. The NP is then extended to include any subsequent words following it that have part-of-speech

tags that are consistent with being inside an NP. A variety of heuristics look at the part-of-speech tags assigned to the surrounding words and try to infer whether the word belongs in the NP or not. If the word remains questionable, the NP segmenter will not put the word into the NP, leaving it available for subsequent segmenters to use.

The NP segmenter implicitly performs part-of-speech disambiguation as a side effect of NP chunking. For example, S2 is shown below after NP segmentation. The words "armed", "man", "bear", and "gun" have all been disambiguated.

NP[The armed man]	shot	NP[the bear]	with	NP[a gun]
art adj noun	*adj/noun/verb*	*art noun*	*prep*	*art noun*

Immediately before and after NP segmentation, Sundance performs **entity recognition**, which tries to match lexico-syntactic patterns corresponding to named entities. For example, a date recognizer pattern might look for a month token followed by a number, a comma, and another number (e.g., "April 4, 1997"). The entity recognition rules can create new NPs (e.g., the April 4 date) or can assign an entity label to an existing NP (e.g., an NP might be relabeled as a NP-LOCATION). The next section describes Sundance's entity recognition module in more detail.

The next step of syntactic analysis is prepositional phrase (PP) segmentation, which creates PP structures. In most cases, the PP segmenter simply looks for a preposition followed by an NP. After PP segmentation, S2 looks like:

NP[The armed man]	shot	NP[the bear]	PP[with NP[a gun]]
art adj noun	*adj/noun/verb*	*art noun*	*prep art noun*

Sundance then performs VP segmentation, which creates verb phrase (VP) structures. Each VP is labeled as one of five types: *active*, *active_infinitive*, *passive*, *passive_infinitive*, or *infinitive*. The *active* and *passive* labels correspond to simple active voice and passive voice constructions. The *active_infinitive* and *passive_infinitive* labels correspond to a main verb (active or passive, respectively) immediately followed by an infinitive construction (e.g., "persuaded to invite" or "was persuaded to invite"). The *infinitive* label corresponds to simple infinitives (e.g., "to invite").

The VP segmenter is more aggressive than the earlier segmenters because it assumes that words that are still unassigned are not likely to belong to NPs or PPs. For example, one heuristic creates an infinitive VP when it sees the word "to" followed by a base verb. This heuristic would be reckless if the PP segmenter had not already grabbed instances of "to" that belong in prepositional phrases. S2 now looks like this:

NP[The armed man]	VP-active[shot]	NP[the bear]	PP[with NP[a gun]]
art adj noun	*verb*	*art noun*	*prep art noun*

Finally, the adjectival phrase (ADJP) segmenter looks for adjectival phrase constructions and puts them into ADJP structures. Once all of the syntactic segmenters have finished, a final "clean-up" phase looks for orphaned words that have not yet been assigned to any constituent. Sundance uses heuristics to push an orphaned word into an existing constituent, create a new constituent for it, or in some cases it will let it remain as an orphaned word.

To be fair, $S2$ is not a difficult sentence for most NLP systems to parse. But we believe that Sundance's approach to syntactic segmentation is natural and robust. One of Sundance's strengths

is that it creates syntactic segments in a bottom-up fashion, so it will usually generate a reasonable parse even when the input contains many unknown words or is ungrammatical. We have used Sundance extensively to process text on web pages, which are notoriously ungrammatical, and it has performed quite well.[5]

3.7 Entity Recognition

Sundance provides a special capability to label words or expressions as a semantic type by defining patterns. We created this pattern recognizer with named entity recognition in mind, but it is fairly general in nature and could be used for other purposes as well.

Sundance's **Recognizer Patterns** must be manually defined by the user.[6] They can specify combinations of words and/or constituents that represent a special type of entity. Each pattern can check for lexical, syntactic, and semantic properties associated with words or constituents.

Each pattern is defined as a rule where the left-hand side specifies a sequence of predicates that must match the input, and the right-hand side specifies the type of entity that will be created. Each predicate has constraints associated with it that must be satisfied for the rule to be successfully matched. The set of possible entity types can be found in the enumerated list "PatternType" in the file "src/tags.h". For example, that list contains the types "PATTERN_COMPANY" and "PATTERN_TIME", which means that the right-hand side of a rule could create entities of the type "NP-COMPANY" or "NP-TIME". The created entity must always be some type of NP. In theory, this mechanism is quite general and could apply to all constituent types, but in practice the current implementation has been inserted into Sundance before PP segmentation. Consequently, the current implementation only allows individual words, punctuation marks, and noun phrase constituents to be part of recognizer rules.

Figure 2 shows the set of predicates and associated constraints that can be used in the left-hand side of rules. The predicates can match either a noun phrase (NP), a punctuation mark (PUNC), or an individual word (WORD). The <NP> predicate simply matches any noun phrase. The <NP-HEAD> predicate matches any NP whose head noun satisfies the specified constraints. The <NP-ANYWORD> predicate matches an NP containing any word that satisfies the specified constraints. The <NP-ALLWORDS> predicate matches any NP such that all of its constituent words satisfy the specified constraints. The <PUNC> predicate matches any punctuation mark that satisfies the given constraints, and the <WORD> predicate matches any word that is not part of an NP and satisfies the given constraints.

There are four types of constraints: case (*case*=), lexical (*word*=), part-of-speech (*tag*=), and semantic (*sem*=). A case constraint looks at the capitalization of a word and has three possible values: *ALL_UPPER* (all letters must be upper case), *ALL_LOWER* (all letters must be lower case), and *CAPITALIZED* (the first letter must be capitalized). A lexical constraint requires an exact word match. A part-of-speech constraint requires that the word's tag set must contain the specified part-of-speech. A semantic constraint requires that the word belong to the given semantic class.

[5] Of course, this does not imply that Sundance produces grammatically perfect output for ungrammatical or ill-formed text. Given such text, however, it seems to do a reasonably good job at producing output that matches the input fairly well.

[6] Some recognizer patterns may already be defined in the Sundance distribution that you receive. But any additional patterns that you want must be added manually.

<NP>
<NP-HEAD:case=casevalue>
<NP-HEAD:sem=semanticfeature>
<NP-HEAD:tag=partofspeech>
<NP-HEAD:word=wordvalue>
<NP-ANYWORD:case=casevalue>
<NP-ANYWORD:sem=semanticfeature>
<NP-ANYWORD:tag=partofspeech>
<NP-ANYWORD:word=wordvalue>
<NP-ALLWORDS:case=casevalue>
<NP-ALLWORDS:sem=semanticfeature>
<NP-ALLWORDS:tag=partofspeech>
<NP-ALLWORDS:word=wordvalue>
<PUNC:word=punctuationtype>
<WORD:case=casevalue>
<WORD:sem=semanticfeature>
<WORD:tag=partofspeech>
<WORD:word=wordvalue>

Figure 2: Predicates and syntax for entity recognition rules

Figure 3 shows some rules and sample expressions that would be recognized by those rules.

Multiple predicates can also be used in a pattern via a conjunction operator (&) or a disjunction operator (|). For example, the following rule will label an NP as a PERSON if any word in the NP has the semantic class TITLE and all words in the NP are capitalized. This would recognize names such as "Dr. Pepper" and "Mr. John Smith".

<NP-ANYWORD:sem=TITLE&ALLWORDS:case=Capitalized> ⇒ NP-PERSON

As another example, the rule below contains both a conjunction and a disjunction, which allows it to recognize a wide variety of organization names. This rule recognizes a phrase as an organization if it consists of (1) an NP whose head noun is capitalized and belongs to the semantic class ORGANIZATION, (2) that NP is followed by the preposition "for" or "of", and (3) the preposition is followed by another NP. This pattern will recognize organization names such as "The National Library of Medicine" and "the Association for Computing Machinery".

<NP-HEAD:sem=Organization&HEAD:case=Capitalized>
<WORD:word=for|WORD:word=of>
<NP>
⇒ NP-ORGANIZATION

If more than one rule matches the same context, then the longer rule is given precedence over the shorter rule. If multiple rules of the same length apply, then the rule that is listed first in the entity recognition rules file is chosen.[7]

[7] By default, this is data/recognizer_patterns.txt.

Pattern:	<WORD:tag=NUMBER> <WORD:word=p.m.> → NP-TIME
Example:	*3 p.m.*
Pattern:	<WORD:sem=MONTH> <WORD:tag=NUMBER> → NP-TIME
Example:	*January 4*
Pattern:	<WORD:sem=MONTH> <WORD:tag=NUMBER> <PUNC:word=COMMA> <WORD:tag=NUMBER> → NP-TIME
Example:	*January 4 , 2001*
Pattern:	<NP-HEAD:word=Corp.> → NP-COMPANY
Example:	*International Business Machines Corp.*
Pattern:	<NP-ALLWORDS:case=Capitalized> <PUNC:word=COMMA> <NP-HEAD:sem=LOCATION> → NP-LOCATION
Example:	*Salt Lake City, Utah*

Figure 3: Sample recognizer rules and example phrases they would match

3.8 Clause Segmentation

Sundance divides each sentence into separate clauses based on the presence of relative pronouns and multiple verb phrases. Sundance scans each sentence from left to right looking for relative pronouns and VPs. When a new relative pronoun is found, a new clause is started. Similarly, when multiple active or passive VPs are found, a new clause is started for each one. Simple infinitives by themselves will not spawn the creation of a new clause. For example, the following sentence will be segmented into three clauses, as shown below:

John paid the man who played the piano and sang songs to entertain people at his party.

Clause #1: *John paid the man*
Clause #2: *who played the piano and*
Clause #3: *sang songs to entertain people at his party.*

3.9 Syntactic Role Assignment

The next step is syntactic role assignment. Sundance assigns three types of syntactic roles to noun phrases: subject (*subj*), direct object (*dobj*), and indirect object (*iobj*). Subjects must precede their VP, while direct objects and indirect objects must follow their VP. The *indirect object* label is assigned only when two NPs immediately follow a VP (in which case the first NP is assumed to be the indirect object and the second NP is assumed to be the direct object). The syntactic role assignments for the previous example sentence are shown here:

John paid the man who played the piano and sang songs to entertain people at his party.

Clause #1: $(John)_{SUBJ}$ *paid* $(the\ man)_{DOBJ}$
Clause #2: *who played* $(the\ piano)_{DOBJ}$ *and*
Clause #3: *sang* $(songs)_{DOBJ}$ *to entertain* $(people)_{DOBJ}$ *at his party.*

As the example shows, a clause can have multiple direct objects if it contains more than one VP. As we explained in Section 3.8, a clause can have only one primary VP (active or passive), but it can have multiple simple infinitive VPs and each infinitive VP can have its own direct object. However, each clause will have at most one subject.

3.10 (Attempts at) Relative Pronoun Resolution

When Sundance finds a clause that does not yet have a subject but contains a relative pronoun in the subject position (i.e., before the verb), then it attempts to resolve the pronoun. The current implementation only tries to resolve some relative pronouns (e.g., "who" and "that"). Sundance uses simple heuristics to choose an antecedent from the previous clause. Relative pronoun resolution is a difficult problem and Sundance's heuristics frequently make mistakes. But our hope is that it is right more often than it is wrong. Eventually, we hope to replace the simple heuristics with a more sophisticated pronoun resolution algorithm.

For the previous sentence, Sundance will (1) identify the NP "the man" as the antecedent for the relative pronoun "who", (2) replace "who" with the "the man", (3) and label "the man" as the subject of the second clause. The result at this point is shown below.

John paid the man who played the piano and sang songs to entertain people at his party.

Clause #1: $(John)_{SUBJ}$ *paid* $(the\ man)_{DOBJ}$
Clause #2: $(the\ man)_{SUBJ}$ *played* $(the\ piano)_{DOBJ}$ *and*
Clause #3: *sang* $(songs)_{DOBJ}$ *to entertain* $(people)_{DOBJ}$ *at his party.*

3.11 Subject Inference

When Sundance discovers a clause that does not have a subject, it assumes that one of the NPs in the previous clause is the subject of the current clause. For example, in the previous sentence Clause #3 still does not have a subject. So Sundance infers that one of the NPs from the previous clause must be the subject of the current clause and copies it over. The final syntactic role assignments for this sentence are shown below. As with relative pronoun resolution, the subject inference heuristics are far from perfect and Sundance often make mistakes. We hope to improve its performance in future releases.

John paid the man who played the piano and sang songs to entertain people at his party.

Clause #1: $(John)_{SUBJ}$ *paid* $(the\ man)_{DOBJ}$
Clause #2: $(the\ man)_{SUBJ}$ *played* $(the\ piano)_{DOBJ}$ *and*
Clause #3: $(the\ man)_{SUBJ}$ *sang* $(songs)_{DOBJ}$ *to entertain* $(people)_{DOBJ}$ *at his party.*

3.12 Sundance in Action: Example Sentences and Parse Output

In this section, we will show some examples of Sundance's parsing behavior to familiarize the reader with Sundance's output format. The sentences in this section were taken from *The Lorax* (Geisel, 1971). Dr. Seuss books can be challenging for NLP systems because they often contain made up words. Consequently, sentences from The Lorax will serve to illustrate how Sundance can infer the correct part-of-speech and syntactic segmentation for words that it has never seen before.

Figure 4 shows Sundance's output for a simple sentence. This sentence is parsed as a typical Subject/VP/Direct_Object structure. Two things worth noting are that (1) the contraction "won't" is expanded by the phrasal lexicon into the words "will not", and (2) the word "Once-ler" is correctly inferred to be the head of an NP because it immediately follows an article.

```
Original : You won't see the Once-ler.
PreProc : You WILL NOT see the Once-ler >PERIOD <EOS
CLAUSE:
    NP SEGMENT (SUBJ):
        [You (LEX)(PN(HUMAN))]
    VP SEGMENT (ACTIVE_VERB):
        [WILL (LEX)(AUX)]
        [NOT (LEX)(ADV)]
        [see (LEX)(V BASE)]
    NP SEGMENT (DOBJ):
        [the (LEX)(ART)]
        [Once-ler (INF-LEX)(ADJ) (N)]
    [>PERIOD (LEX)(PUNC)]
    [<EOS (?)]
```

Figure 4: A simple sentence from The Lorax

Figure 5 shows a line from The Lorax that is not a complete sentence but is an isolated noun phrase followed by an exclamation mark. Since Sundance is a partial parser, it can handle text fragments like this just fine. Although it does label *"The Truffula Trees"* as the subject, even though there is no verb in the sentence for it to be the subject of!

Figure 6 shows Sundance's parse output for a longer sentence. Sundance divides this sentence into two clauses and correctly handles three novel words: *"Lerkim"*, *"miff-muffered"*, and *"moof"*. Sundance isn't sure what to do with the two words *"cold"* and *"out"*, so they are not put into any syntactic structure and are left as isolated words. This example also shows that Sundance performs PP-attachment for prepositional phrases that contain the preposition *"of"* because these PPs virtually always attach to the immediately preceding constituent.[8] PP-attachment is not performed on any other types of PPs.

[8] Ironically, in this example the PP really shouldn't be attached to *'his own clothes"*. But in the vast majority of cases, attaching *'bf"*PPs to the preceding constituent will be correct.

```
Original : The Truffula Trees!
PreProc : The Truffula Trees >EXCLAMATION <EOS
CLAUSE:
    NP SEGMENT (SUBJ):
        [The (LEX)(ART)]
        [Truffula (INF-LEX)(ADJ) (N)]
        [Trees (root: tree) (LEX)(N PLURAL)]
    [>EXCLAMATION (LEX)(PUNC)]
    [<EOS (?)]
```

Figure 5: An isolated noun phrase from The Lorax

4 Information Extraction with Sundance and AutoSlog

Ultimately, our goal when developing Sundance was to use it for information extraction. Consequently, Sundance is more than just a parser. It also includes an information extraction engine that uses case frame structures to extract information from sentences.

Information extraction (IE) is a natural language processing task that involves identifying and extracting domain-specific information from text. For example, an IE system could be built to extract facts about corporate acquisitions from Wall Street Journal articles. The system might extract the names of the people and companies involved in the acquisition, the date of the acquisition, and the amount paid for the acquired company.

Most IE systems use domain-specific patterns or case frames to recognize relevant information in text. Some systems use extraction patterns that can be recognized with simple pattern matching or regular expressions and do not require any linguistic processing (e.g., first-order logic rules (Freitag, 1998) or wrappers (Kushmerick et al., 1997)), but many IE systems use extraction patterns that require syntactic processing (e.g., (Kim and Moldovan, 1993; Huffman, 1996; Riloff, 1996b; Riloff and Jones, 1999; Soderland et al., 1995)).

Sundance's case frame structures can be defined manually, which allows users to create structures with very specific syntactic and semantic constraints. Defining case frames by hand is tedious and time-consuming, however. As an alternative, we have also developed and implemented two learning algorithms, AutoSlog and AutoSlog-TS, that can generate simple case frame structures (a.k.a., *extraction patterns*) automatically.

In this section, we describe Sundance's IE capabilities as well as the AutoSlog and AutoSlog-TS learning algorithms. Section 4.1 explains how Sundance performs information extraction by instantiating case frame structures. Section 4.2 describes the AutoSlog learning algorithm that can generate extraction patterns given a list of targeted nouns and a training corpus. Section 4.3 presents the AutoSlog-TS learning algorithm, which is a derivative of AutoSlog that can generate extraction patterns using only unannotated "relevant" and "irrelevant" texts as input. Finally, Section 4.4 explains how the case frame templates used by AutoSlog and AutoSlog-TS are defined and how new templates can be created.

```
Original : He lurks in his Lerkim, cold under the roof, where he makes his own clothes
out of miff-muffered moof.
PreProc : He lurks in his Lerkim >COMMA cold under the roof >COMMA where he
makes his own clothes out of miff-muffered moof >PERIOD <EOS
CLAUSE:
    NP SEGMENT (SUBJ):
        [He (LEX)(PN SINGULAR(HUMAN))]
    VP SEGMENT (ACTIVE_VERB):
        [lurks (root: lurk) (MOR)(V PRESENT)]
    PP SEGMENT (PREP):
        [in (LEX)(PREP)]
        NP SEGMENT:
            [his (LEX)(ADJ)]
            [Lerkim (INF-LEX)(ADJ) (N)]
    [>COMMA (LEX)(PUNC)]
    [cold (LEX)(ADJ)]
    PP SEGMENT (PREP):
        [under (LEX)(PREP)]
        NP SEGMENT:
            [the (LEX)(ART)]
            [roof (LEX)(N SINGULAR)]
    [>COMMA (LEX)(PUNC)]
CLAUSE:
    [where (LEX)(C_M)]
    NP SEGMENT (SUBJ):
        [he (LEX)(PN SINGULAR(HUMAN))]
    VP SEGMENT (ACTIVE_VERB):
        [makes (root: make) (MOR)(V PRESENT)]
    NP SEGMENT (DOBJ):
        [his (LEX)(ADJ)]
        [own (LEX)(ADJ)]
        [clothes (LEX)(N SINGULAR)]
    [out (LEX)]
    Following PP attaches to: his own clothes
    PP SEGMENT (PREP):
        [of (LEX)(PREP)]
        NP SEGMENT:
            [miff-muffered (INF-LEX)(ADJ)]
            [moof (?)(UNK)]
    [>PERIOD (LEX)(PUNC)]
    [<EOS (?)]
```

Figure 6: A longer sentence from The Lorax

4.1 Sundance's Information Extraction Engine

Sundance can perform information extraction (IE) using structures called *case frames*. Unlike many IE systems that use superficial patterns based on surface-level features and regular expressions, Sundance's case frames rely heavily on the syntactic analysis and (optionally) semantic labels provided by Sundance.[9] In later sections when we discuss AutoSlog and AutoSlog-TS, we will use the term *extraction pattern* to describe the structures that perform information extraction. In our implementation, an *extraction pattern* is represented as a single-slot case frame. In general, however, case frame structures can include multiple slots and can be arbitrarily complex. So we will use the term *extraction pattern* to refer to single-slot case frames, usually in the context of AutoSlog and AutoSlog-TS, but it is worth noting that extraction patterns and case frames are basically the same thing to Sundance.

Each case frame can contain five things: (1) a name, (2) an anchor designation, (3) a set of activation functions and their arguments, (4) optionally, a type that assigns a category to the case frame, and (5) optionally, a set of slots to perform information extraction.[10] Figure 7 shows the syntax for case frame definitions.

CF:
Name: [*string*]
Anchor: [*constituent_variable*]([*word*])
Type: [*string*] (optional)
Act_Fcns: [*list of activation functions and their arguments*]
Slot: [*synrole*] <- [*slotname*] <- [*semclasses*] (optional)
.
.
.
Slot: [*synrole*] <- [*slotname*] <- [*semclasses*] (optional)

Figure 7: Case Frame Syntax

The **Name** designator assigns a name to the case frame, which can be any string. The name is only for ease of reference and is not used internally by Sundance.

Before we describe the rest of a case frame definition, let's introduce the notion of a *constituent variable*. A *constituent variable* is written as [*constituent type*][#], where a constituent type is a type of syntactic constituent such as an NP or VP. The number is simply an identifier for the variable. For example, VP1 is a variable representing a verb phrase constituent and VP2 is a variable representing a different verb phrase constituent.

As shown in Figure 7, each case frame has an **Anchor** designator that specifies a constituent variable and a word. Intuitively, this means that the case frame is looking for a syntactic constituent of that type with the designated word as its head. When a sentence is parsed that contains a

[9] In principle, other parsers could be used as well, but the current implementation is based on Sundance.

[10] We made slot definitions optional so that case frames can be used for applications that need to recognize a pattern but do not need to perform IE. Case frames were really designed for information extraction, however, and slots must always be defined for IE.

constituent matching the anchor specification, then the case frame is considered for activation. (The activation functions, which we will describe shortly, must be satisfied before the case frame will actually be activated.) For example, if the anchor designator is *VP1(kidnapped)* then the case frame will be considered for activation when the word "kidnapped" is found as the head of a verb phrase (VP). Similarly, if the anchor designator is *NP1(mayor)* then the case frame will be considered for activation when the word "mayor" is found as the head of a noun phrase (NP). The primary purpose of the anchor is to dictate how the slot extractions are done; we will return to this issue when we discuss the slot definitions.

The **Type** designator can be used to specify a category associated with the case frame. We typically use the type designator to define an event that is associated with expressions that the case frame will match. For example, case frames that are activated by expressions such as "was killed", "was assassinated", and "slain" might be assigned the type MURDER so that all extractions associated with those case frames will be understood to be facts about murder events.

The activation functions, **Act_Fcns**, are tests that must be satisfied for the case frame to become fully active. Typically, the activation functions used by Sundance look for general types of syntactic constructions. For example, one activation function matches VPs in the active voice (*active_verb_broad_p*), another activation function matches VPs in the passive voice (*passive_verb_broad_p*), and a different activation function matches NPs with a specific head noun (*noun_phrase_broad_p*).

For each activation function, one or more constituent variables must be specified as arguments. In most cases, the constituent variable is also given a word as an argument to define the lexico-syntactic pattern that the case frame is looking for. For example, *passive_verb_broad_p(VP1(detonated))* means that the case frame will be activated whenever a passive voice VP with the head word "detonated" is found. Similarly, *active_verb_broad_p(VP1(detonated))* means that the case frame will be activated whenever an active voice VP with the head word "detonated" is found.

In some cases, a case frame will have more than one activation function. All of the activation functions must be satisfied for the case frame to be activated. For example, consider a case frame that should be activated when it sees the expression: *faced trial*. This case frame would require two activation functions. The first function would be *active_verb_narrow_p(VP1(FACED))*[11], which looks for an active voice VP that has the verb "faced" as its head. The second function would be *has_dobj_following_r(VP1(FACED) NP1(TRIAL))*, which looks at the direct object following the VP containing "faced" and checks to see if its head is the word "trial". If both of those functions are satisfied, then the case frame will be activated.

It is important to remember that the case frames rely on Sundance's syntactic analysis so they will match any sentence that has the desired syntactic constituents even if the specified words are not adjacent. For example, the words "faced" and "trial" do not have to be adjacent for the previously mentioned case frame to fire, as long as "trial" is the direct object of "faced". This case frame would be activated by sentences such as: *"Saddam faced a long and lengthy trial"* and *"The mobster faced in a month a racketeering trial"*.

When a case frame becomes active, it extracts information from the current clause (i.e., the clause in which it was activated) by consulting its slots. We say that a case frame is *instantiated*

[11] The *active_verb_narrow_p* function only recognizes simple active voice VPs. In contrast, the *active_verb_broad_p function* recognizes both simple active voice VPs as well as VPs that contain an active voice verb followed by an infinitive, such as "wanted to see".

when its slots are filled. Each slot must define a syntactic role (*synrole*) from which information will be extracted. Currently, the possible values for a synrole are: *subj* [subject], *dobj* [direct object], *iobj* [indirect object], and *PP(prep)* [prepositional phrase], where *prep* can be any preposition. These syntactic roles are relative to the constituent that is specified as the case frame's anchor. For example, if the anchor is defined as *VP1(kidnapped)*, then the case frame will only extract subjects, direct objects, indirect objects, and prepositional phrases that are associated with the VP containing the word "kidnapped". When the anchor is a noun phrase, such as *NP1(trial)*, then typically only prepositional phrases are defined as slots. These PPs must attach to the NP containing the word "trial" in order to be extracted.

Optionally, a name can be assigned to each slot (*slotname*) to label the type of object that it will extract. In most cases, we use slotnames that correspond to thematic roles. For example, when the verb "kidnapped" appears in the active voice, we might want to label its subject as the *agent* or *perpetrator* of the kidnapping. When the verb "kidnapped" appears in the passive voice, we might want to label its subject as the *theme* or *victim* of the kidnapping.

Selectional restrictions can also be defined by assigning semantic constraints to a slot. If semantic classes are associated with a slot, information will be extracted by that slot only if it matches those semantic categories. If only one semantic class is listed, then an item must belong to that class to be extracted. If multiple semantic classes are desired, they can be listed either as a disjunction (an item must match at least one of the semantic classes) or as a conjunction (an item must match all of the semantic classes). Disjunctions are represented by separating the semantic classes by a comma and white space (e.g, "AIRCRAFT, AUTO") and conjunctions are represented by joining the semantic classes with a $\wedge$ symbol without any white space between them (e.g, "AIRCRAFT$\wedge$WEAPON"). A disjunction of conjunctions is also possible (e.g., "AIRCRAFT$\wedge$WEAPON, SHIP$\wedge$WEAPON, MISSILE").

An Example Case Frame

Figure 8 shows an example case frame designed to extract information about vehicle crashes. This case frame will be activated when it sees the verb "crashed" in the active voice. When it is activated, it will create a case frame of type Vehicle_Crash, indicating that a crash event has been identified in the text. Then it will try to fill the slots. To fill the first slot, Sundance grabs the subject of the verb "crashed" and checks to see if it has the semantic class AIRCRAFT or AUTO.[12] If so, then Sundance uses the subject to fill the slot and labels it as a Vehicle extraction.

To fill the second and third slots, Sundance grabs prepositional phrases with the preposition "in" that attach to the verb "crashed". If the PP has the semantic class LOCATION, then it is put into the second slot and labeled as a Place. If the PP has the semantic class TIME, then it is put into the third slot and labeled as a Date. If the PP doesn't belong to either the LOCATION or the TIME semantic class, then it isn't extracted at all. Figure 9 shows the IE output produced from a sample sentence using the case frame defined in Figure 8.

[12]The current implementation can check (1) the semantic classes of the head of the constituent and (2) classes assigned by the entity recognizer.

CF:	
Name:	example_caseframe
Anchor:	VP1(CRASHED)
Type:	Vehicle_Crash
Act_Fcns:	active_verb_broad_p(VP1(CRASHED))
Slot:	subj <- Vehicle <- AIRCRAFT, AUTO
Slot:	PP(in) <- Place <- LOCATION
Slot:	PP(in) <- Date <- TIME

Figure 8: Sample Case Frame Definition

The plane crashed in Utah in January in a blaze of fire.

CaseFrame: example_caseframe
Type: Vehicle_Crash
Trigger(s): (CRASHED)
Extractions:
Vehicle = "The plane" [AIRCRAFT]
Place = "Utah" [LOCATION]
Date = "January" [TIME]

Figure 9: An Instantiated Case Frame

4.2 AutoSlog

AutoSlog (Riloff, 1993; Riloff, 1996a) is a program that can be used to automatically generate extraction patterns (i.e., simple case frames) for a domain. The original AutoSlog system was developed long ago and based on a different parser, but we have reimplemented AutoSlog on top of Sundance. AutoSlog is a supervised learning algorithm that takes items targeted for extraction as input and generates a set of case frames to extract those items. A derivative of AutoSlog, called AutoSlog-TS, is a more weakly supervised learning algorithm that generates extraction patterns using only "relevant" and "irrelevant" texts as input. AutoSlog-TS will be described in more detail in the next section. However, AutoSlog-TS shares much in common with AutoSlog, so readers who are interested in AutoSlog-TS should read this section as well as the next one.

AutoSlog was designed to generate extraction patterns for a domain given an annotated corpus of texts in which the NPs targeted for extraction have been labeled. For example, a training corpus for a terrorism domain might contain annotations corresponding to the victims and perpetrators of terrorist attacks. AutoSlog originally focused only on noun phrase extraction, but our new implementation allows other syntactic constituents to be extracted as well (see Section 4.4).

For the sake of simplicity, our implementation of AutoSlog does not actually use an annotated corpus. Instead, a user gives AutoSlog a set of unannotated texts and a list of nouns that should be extracted. AutoSlog then finds every NP in the corpus whose head matches a noun in the list. Those NPs are assumed to be targeted for extraction (we will refer to these as the *targeted constituents*). One advantage of this approach is that there is no need for a manually annotated corpus as input. Instead, a user just needs to supply a list of nouns representing items of interest in the corpus (e.g., the names of victims and perpetrators in the corpus). There are several disadvantages of this approach, however. Every occurrence of a targeted noun will be extracted, so AutoSlog can not distinguish between instances that occur in a relevant context (e.g., references to a specific person in a terrorist event description) and those that occur in an irrelevant context (e.g., references to that same person in a story that has nothing to do with terrorism). Also, AutoSlog will extract all NPs with the targeted head noun, which may include noun phrases that do not match the intended item because of different modifiers. For example, if a targeted noun is "Clinton", then references to Bill Clinton, Hillary Clinton, and Chelsea Clinton will all be targeted for extraction. Despite these issues, we felt that the simplicity of requiring just a noun list as input was more practical than requiring manually annotated texts with a specific annotation format.

Given a targeted constituent and a sentence, AutoSlog uses syntactic templates to heuristically identify an expression in the sentence that describes the conceptual role that the constituent plays. Figure 10 shows the set of syntactic templates originally developed for AutoSlog (Riloff, 1993; Riloff, 1996a). The left-hand column shows the syntactic templates and the right-hand column shows examples of extraction patterns learned from those templates. The templates are divided into three types, depending upon whether the targeted constituent is a subject (*subj*), direct object (*dobj*), or prepositional phrase (*obj-prep*). If the targeted constituent is a subject, for example, then only the subject heuristics apply.

All of the heuristics are applied to a sentence and the ones that match will generate patterns based on the specific words in the sentence. For example, consider the following sentence:

Luke Johnson was killed in Iraq by insurgents.

If *Luke Johnson* was a targeted constituent, then AutoSlog would generate the pattern *"<subj> was killed"* because *Luke Johnson* is the subject of the clause and template #1 is the only subject template that matches the clause. Template #1 will then be matched against the sentence and instantiated with the verb "killed". The learned pattern will extract subjects of the verb "killed" whenever it appears in the passive voice.

Syntactic Template	**Sample Learned Pattern**
1. <subj> passive_verb	<subj> was acquired
2. <subj> active_verb	<subj> sold
3. <subj> active_verb dobj	<subj> saw profit
4. <subj> verb infinitive	<subj> wanted to sell
5. <subj> auxbe noun	<subj> is CEO
6. <subj> auxhave noun	<subj> has debt
7. active_verb <dobj>	sold <dobj>
8. infinitive <dobj>	to sell <dobj>
9. verb infinitive <dobj>	agreed to buy <dobj>
10. noun auxbe <dobj>	CEO is <dobj>
11. noun auxhave <dobj>	buyer has <dobj>
12. infinitive prep <np>	to sell for <np>
13. active_verb prep <np>	will sell for <np>
14. passive_verb prep <np>	was sold for <np>
15. noun prep <np>	owned by <np>

Figure 10: AutoSlog's original syntactic templates and sample patterns

Similarly, if *insurgents* was a targeted constituent, AutoSlog would generate the pattern *"was killed by <np>"* because *insurgents* is in a PP and template #14 matches the sentence. The learned pattern will extract all NPs that occur as the object of the preposition "by" and attach to passive forms of the verb "killed". Note that the words "killed" and "by" do not have to be adjacent.

AutoSlog's heuristics are not perfect, so some of the patterns that AutoSlog generates are not desirable. Consequently, a human must manually review the resulting extraction patterns and decide which ones to keep. The user may also wish to label each slot with a thematic or conceptual role indicating what type of item will be extracted. For example, the NP extracted by the pattern *"<subj> was killed"* will be a victim, so the pattern could be re-written as *"<victim> was killed"*. Similarly, the pattern *"was killed by <np>"* will extract perpetrators, so the pattern could be re-written as *"was killed by <perpetrator>"*.[13]

Figure 11 shows a sample sentence and the extraction patterns that AutoSlog would produce if all of the NPs in the sentence were targeted for extraction. The NPs are underlined. The left-hand side shows the actual case frames structures that AutoSlog generates. The right-hand side shows an intuitive (short-hand) description of the learned extraction patterns. Notice that there are seven NPs in the sentence, but only six case frames are learned. This is because none of AutoSlog's syntactic

[13]These are all shorthand notations. In actuality, each extraction pattern is a case frame structure and a slotname would be assigned corresponding to *victim* or *perpetrator*.

templates match the context surrounding the NP "one", so no case frames are learned to extract it. Also notice that there are two patterns that were created to extract the subject of "injuring", even though "injuring" has no explicit subject in the sentence. During parsing, the subject inference rules kicked in and inferred that "take-off" is the implicit subject of "injuring", so the patterns were created in response to the subject NP "take-off". Figure 12 shows how Sundance would subsequently use the learned case frames to extract information from the same sentence.

4.3 AutoSlog-TS

AutoSlog-TS[14] is a derivative of AutoSlog that uses the same syntactic heuristics to learn extraction patterns, but AutoSlog-TS does not require annotated training data or a list of targeted nouns as input. Instead, AutoSlog-TS learns from two sets of unannotated texts: one collection of texts that are *relevant* to the topic of interest, and one collection of texts that are *irrelevant* to the topic of interest. For example, a user who wants to learn extraction patterns for a terrorism domain should provide one set of texts that describe terrorist events and one set of texts that do not describe terrorist events.[15] In a nutshell, AutoSlog-TS generates every possible pattern it discovers in the corpus and then computes statistics based on how often each pattern appears in the relevant texts versus the irrelevant texts. AutoSlog-TS then produces a ranked list of extraction patterns coupled with statistics indicating how strongly each pattern is associated with the domain.

AutoSlog-TS learns extraction patterns by performing two passes over the corpus. In the first pass, AutoSlog-TS applies AutoSlog's syntactic templates to the texts in an exhaustive fashion. That is, AutoSlog-TS generates patterns to extract every noun phrase in the corpus by applying the syntactic templates to every context surrounding a noun phrase. Whenever a template matches, an extraction pattern is generated by instantiating the template based on the words in the context. One way to think about this process is that it is analogous to asking AutoSlog to generate patterns to extract every noun phrase it sees. This step produces an enormous set of extraction patterns, often generating tens of thousands of distinct patterns. Collectively, these patterns are capable of extracting nearly every NP in the corpus.[16]

The second pass of AutoSlog-TS applies the learned patterns to the corpus and collects statistics based on how often each pattern appears in the relevant texts versus the irrelevant texts.[17] For each extraction pattern, $pattern_i$, AutoSlog-TS computes two frequency counts: $totalfreq_i$ is the number of times that the pattern appears anywhere in the corpus, and $relfreq_i$ is the number of

[14]For those curious about the name, AutoSlog-TS stands for "AutoSlog - The Sequel". This name was intended to be temporary while I tried to think of something more clever, but alas, I never did.

[15]It is best if the relevant and irrelevant texts are of a somewhat similar nature because AutoSlog-TS will focus on the differences between them. For example, if the irrelevant texts cover a dramatically different topic, such as chemistry, then any pattern common in terrorism articles but not in chemistry articles will be thought to be terrorism-related. This may include any pattern that mentions a weapon, the police, or the media. If, however, the irrelevant texts include stories about (say) natural disasters or petty crime, then AutoSlog-TS will do a better job of distinguishing patterns that are truly terrorism-related from those that represent (say) injuries, weapons, and police activities that have nothing to do with terrorism.

[16]There will be some NPs that cannot be extracted because their surrounding context does not match any of the syntactic templates.

[17]Our implementation will generate these statistics after both the first pass and the second pass. However, the numbers produced after the first pass are only preliminary. The numbers produced after the second pass are the official, most accurate AutoSlog-TS statistics.

Input Sentence: *A light plane crashed at a Jerusalem airport after take-off injuring six people, one of them critically, police said.*

Case Frame Structure	Shorthand Pattern Description
Name: <subj>_active_verb_1 **Anchor:** VP1(CRASHED) **Act_Fcns:** active_verb_broad_p(VP1(CRASHED)) **Slot:** subj	<subj> crashed
Name: active_verb_prep<NP>_1 **Anchor:** VP1(CRASHED) **Act_Fcns:** active_verb_narrow_p(VP1(CRASHED)) **Slot:** PP(AT)	crashed at <np>
Name: active_verb_prep<NP>_2 **Anchor:** VP1(CRASHED) **Act_Fcns:** active_verb_narrow_p(VP1(CRASHED)) **Slot:** PP(AFTER)	crashed after <np>
Name: <subj>_active_verb_2 **Anchor:** VP1(INJURING) **Act_Fcns:** active_verb_broad_p(VP1(INJURING)) **Slot:** subj	<subj> injuring
Name: <subj>_active_verb_dobj_1 **Anchor:** VP1(INJURING) **Act_Fcns:** active_verb_narrow_p(VP1(INJURING)) has_dobj_following_r(VP1(INJURING) NP1(PEOPLE)) **Slot:** subj	<subj> injuring people
Name: active_verb_<dobj>_1 **Anchor:** VP1(INJURING) **Act_Fcns:** active_verb_narrow_p(VP1(INJURING)) **Slot:** dobj	injuring <dobj>
Name: noun_prep<NP>_1 **Anchor:** NP1(ONE) **Act_Fcns:** noun_phrase_broad_p(NP1(ONE)) **Slot:** PP(OF)	one of <np>
Name: <subj>_active_verb_3 **Anchor:** VP1(SAID) **Act_Fcns:** active_verb_broad_p(VP1(SAID)) **Slot:** subj	<subj> said

Figure 11: Extraction patterns (simple case frames) generated by AutoSlog

Parsed Sentence: *A light plane crashed at a Jerusalem airport after take-off Friday >COMMA injuring &&6 people >COMMA one of them critically >COMMA police police said >PERIOD <EOS*

CaseFrame: <subj>_active_verb_1
Trigger(s): (CRASHED)
Extractions:
SUBJ_Extraction = *'A light plane"*[NO ATTRIBUTES]

CaseFrame: active_verb_prep<NP>_1
Trigger(s): (CRASHED)
Extractions:
PP(AT)_Extraction = *'a Jerusalem airport"*[NO ATTRIBUTES]

CaseFrame: active_verb_prep<NP>_2
Trigger(s): (CRASHED)
Extractions:
PP(AFTER)_Extraction = *'take-off"*[NO ATTRIBUTES]

CaseFrame: <subj>_active_verb_2
Trigger(s): (INJURING)
Extractions:
SUBJ_Extraction: *'take-off"*[NO ATTRIBUTES]

CaseFrame: <subj>_active_verb_dobj_1
Trigger(s): (INJURING,PEOPLE)
Extractions:
SUBJ_Extraction: *'take-off"*[NO ATTRIBUTES]

CaseFrame: active_verb_<dobj>_1
Trigger(s): (INJURING)
Extractions:
DOBJ_Extraction = *"&&6 people"*[NO ATTRIBUTES]

CaseFrame: noun_prep<NP>_1
Trigger(s): (ONE)
Extractions:
PP(OF)_Extraction = *'them"*[NO ATTRIBUTES]

CaseFrame: <subj>_active_verb_3
Trigger(s): (SAID)
Extractions:
SUBJ_Extraction = *'police"*[NO ATTRIBUTES]

Figure 12: The learned extraction patterns applied to a sentence by Sundance

times that the pattern appears in the relevant texts. AutoSlog-TS then computes two statistical measures from these frequency counts:

- A conditional probability estimate of the likelihood that a story is relevant given that a specific pattern, $pattern_i$, appears in the story.

$$P(relevant \mid pattern_i) = \frac{relfreq_i}{totalfreq_i}$$

- The RlogF measure (Riloff, 1996b), which weights (balances) a pattern's conditional probability estimate with the log of its frequency.

$$RlogF(pattern_i) = log_2(relfreq_i) * P(relevant \mid pattern_i)$$

An extraction pattern will receive a high RlogF score if either its probability is very high and its frequency is moderately high or if its frequency is very high and its probability is moderately high. Intuitively, this rewards patterns that are very strongly correlated with the relevant texts or are moderately correlated with the relevant texts but occur very frequently in them. For example, a pattern that has a probability of .75 and frequency of 500 is often just as valuable as a pattern that has a probability of .95 and a frequency of 10 because the more frequent pattern is so common that many extractions will be missed if the pattern is not used.

Our implementation of AutoSlog-TS produces a list of all the learned patterns and, for each one, computes the four statistics just described: *totalfreq*, *relfreq*, *prob*, and *RlogF*. It is then up to the user to rank the patterns based on the statistic(s) of their choosing and to decide what threshold values to use to isolate the most important and most reliable patterns for their application domain. In our IE research, we have found that the *RlogF* measure does a good job of lifting the most important extraction patterns to the top of the list. We usually select the best patterns using both an *RlogF* threshold and a *totalfreq* threshold, since the statistics on low frequency patterns are usually not reliable enough to depend on. For classification tasks where precision is paramount (e.g., (Riloff and Wiebe, 2003)), we have ranked the patterns based on the *prob* value instead of *RlogF* because the probability measure identifies patterns that have the strongest correlation with the relevant texts.

Figure 13 shows relevant and irrelevant text samples and the output that AutoSlog-TS would produce given these text samples as input. Note that all patterns that have *RelFreq* ≤ 1 will receive an *RlogF* score of 0 because $log_2(1)$ is 0.

Relevant Text

CNN reported that three people died today in Bogota in a terrorist event. Armed guerrillas shot 3 judges with a machine gun. They died in an attack at the court house. The FMLN claimed responsibility for the death of the judges and claimed that the death of more judges would soon follow.

Irrelevant Text

The Los Angeles Times reported that Marlon Brando died today in California. Marlon Brando died at the UCLA Hospital at the age of 80. Sources claimed that he had been diagnosed with pulmonary fi brosis.

TotalFreq	RelFreq	Prob	RlogF	Pattern
4	3	0.750	1.189	*died in <np>*
2	2	1.000	1.000	*death of <np>*
3	2	0.667	0.667	*<subj> claimed*
4	2	0.500	0.500	*<subj> died*
1	0	0.000	0.000	*was diagnosed with <np>*
1	0	0.000	0.000	*<subj> was diagnosed*
1	0	0.000	0.000	*age of <np>*
1	1	1.000	0.000	*<subj> follow*
1	1	1.000	0.000	*responsibility for <np>*
1	1	1.000	0.000	*claimed <dobj>*
1	1	1.000	0.000	*<subj> claimed responsibility*
3	1	0.333	0.000	*died at <np>*
1	1	1.000	0.000	*shot with <np>*
1	1	1.000	0.000	*shot <dobj>*
1	1	1.000	0.000	*<subj> shot judges*
1	1	1.000	0.000	*<subj> shot*
2	1	0.500	0.000	*<subj> reported*

Figure 13: AutoSlog-TS Example

4.4 Defining New Case Frame Templates for AutoSlog and AutoSlog-TS

A feature that we recently added to AutoSlog and AutoSlog-TS is the ability to explicitly define the *case frame templates* that they use for learning. In Section 4.2, Figure 10 showed the syntactic templates that were designed for the original AutoSlog and AutoSlog-TS algorithms. Those templates are already implemented in our current system. However, we thought it would be useful for users to be able to design their own case frame templates for new applications. The current implementation of AutoSlog permits this by storing the case frame templates in a special file called "caseframe_templates.txt" that can be easily augmented or modified.

Figure 14 shows an example of a properly formatted case frame templates file. Each template begins with the string "NAME: " (space after colon is required). Following this is a single string (without any whitespace) denoting the name of the template. The value of the string has no significance to AutoSlog other than giving a name to case frames that are instantiated from that template. The next line begins with "ANCHOR: " and specifies a constituent variable that the case frame will use as its anchor.

The next line begins with "ACT_FCNS: ". Activation functions are the predicates that must be satisfied for a case frame to be created. The activation_functions that are already recognized by Sundance can be found in the file src/activationFcnPatterns.C. At least one activation function must be specified. When multiple activation functions are specified, they must all be satisfied for the case frame to be created (see Section 4.2 for details on the purpose of activation functions). The names given to the activation functions are case-sensitive.

Activation functions come in two varieties: *predicate* functions, which have the suffix _p, and *return* functions, which have the suffix _r. Predicate activation functions simply return true or false. Return activation functions also return true or false, but in addition they return a value as a side effect of processing. The value to be returned is specified by giving the return function one constituent variable that has an ampersand in front of it. For example, the *<subj>_active_verb_dobj* function shown in Figure 14 uses the return function *has_dobj_following_r*, which will put its return value in the NP1 variable. In essence, this function checks to see whether the VP has a direct object following it, and if so it puts that direct object in the NP1 variable.

The final part of a case frame template is denoted by the keyword "EXT: ". This parameter is optional, but if you want the case frame to extract something then this slot must be defined. If included, it is followed by one or more keywords that denote what part of the clause should be extracted. The current legal keywords are: *np*, *subj*, *dobj*, and *prep*. We expect to add more extraction options in the future.

To illustrate how a new template would be defined, suppose you wish to create a pattern that will extract the direct object of an active verb phrase, but only if the active verb is in the past tense and it isn't preceded by a negative word. First, you must decide what to name the template. Perhaps you would name it *"active_verb_past_not_negative_<dobj>"*. Since this pattern is designed to extract the direct object of a verb phrase, you would designate its anchor to be a verb phrase constituent variable (e.g., VP1).[18] Next you need to decide which activation functions should be included. You would almost certainly choose *active_verb_narrow_p* to look for an active verb phrase. Since active_verb doesn't require the verb to be in the past tense, and it doesn't check for the presence of

[18]This is because the role to be extracted, *dobj*, should be the direct object of the verb phrase.

```
NAME: <subj>_passive_verb
ANCHOR: VP1
ACT_FCNS: passive_verb_broad_p(VP1)
EXT: subj

NAME: <subj>_active_verb_dobj
ANCHOR: VP1
ACT_FCNS: active_verb_narrow_p(VP1)
                has_dobj_following_r(VP1, &NP1)
EXT: subj

NAME: noun_prep<NP>
ANCHOR: NP1
ACT_FCNS: noun_phrase_broad_p(NP1)
EXT: prep
```

Figure 14: Three Sample Case Frame Templates

negative words, you will need to add more activation functions as constraints. You might create a new activation function called *no_negate_verb_p*, which can be added to the list of activation functions to make sure the verb is not negated, as well as a function called *past_tense_verb_p* which would check that the verb phrase contains a past tense verb. Now you have all the constraints you need, but you still need to define the items that the template will extract. The direct object of the verb should be extracted, so you would define *dobj* as the extraction type. The new case frame template will look like this:

```
NAME: active_verb_past_not_negative_<dobj>
ANCHOR: VP1
ACT_FCNS: active_verb_narrow_p(VP1)
                no_negate_verb_p(VP1)
                past_tense_verb_p(VP1)
EXT: dobj
```

You might notice that the current list of items that can be extracted all refer to noun phrases. This is because AutoSlog was created for information extraction tasks, which tend to look for noun phrases. However, our implementation is flexible enough to allow for additional extraction types to be created, such as adjective_phrases and verb_phrases, or even head_words only. We expect to be augmenting the types of extractions that are possible in the future.

5 How to Use Sundance and AutoSlog

In this section, we will review the mechanics involved in installing and running Sundance and AutoSlog. All instructions assume that you are installing Sundance on a linux or unix operating system. Sundance can be used either as a stand-alone application or as an API. We will explain how to use Sundance in both ways. AutoSlog is primarily meant to be used as a stand-alone application, so we will only review that mode of operation.

5.1 Installing Sundance and AutoSlog

Sundance and AutoSlog are packaged together in a single installation file, which we will call the *Sundance distribution package*. This file is distributed as a tar'ed and gzip'ed file. The name of the file will be *"sundance-<number>.tar.gz"*, where the number corresponds to the version of the Sundance distribution you have received. For example, if you received version 4.2, then the file will be named sundance-4.2.tar.gz . To install the software, you should follow these steps:

1. Decide on a top-level directory under which you would like to install Sundance. Sundance will create its own subdirectory containing all of its files in the top-level directory that you choose.

2. Move the Sundance distribution package to this directory.

3. Unzip and untar the distribution package. Here are details for how to unzip and untar files:

   To unzip the file, on the command line type:
   `gunzip sundance-<number>.tar.gz`
   This will produce a file called `sundance-<number>.tar`

   To untar the file, on the command line type:
   `tar -xvf sundance-<number>.tar`
   This will produce a directory called `sundance-<number>`, with the Sundance source and data files placed underneath it.

4. Enter the `sundance-<number>` directory. You will find the following sub-directories: bin/, data/, docs/, include/, lib/, man/, regression/, scripts/, src/, and tools/.

5. Enter the scripts/ directory.

6. Run the executable script *install_sundance*.[19] You must be in the scripts/ directory when you run *install_sundance*! This script will perform many functions, such as setting the current directory path in files to allow for correct installation. It will then compile the source code and

[19] The *install_sundance* script requires that your system has the csh shell installed and looks for it in /bin. If you do not have csh installed, you will have to perform the procedures found in install_sundance by hand. This basically requires replacing placeholders in various fi les with your current directory path, copying these fi les to different directories, and then running the Makefi le in the src/ directory. The steps are pretty straightforward for anyone with solid programming skills and minor familiarity with gawk. Otherwise, it will be necessary to install csh to complete the installation.

create the Sundance and AutoSlog binaries. This will take a few minutes. You may receive some warning messages during compilation, but they should not affect correct installation. We try to ensure that Sundance is compatible with recent releases of g++, but if you have compilation problems then you may want to check the README file to see which version of g++ this Sundance package is known to be compatible with.

After running the *install_sundance* script, you should have a working installation of Sundance! The Sundance and AutoSlog binaries will be located in the bin/ directory. You will find a file called *.nlp-opts* in the data/ directory. This file specifies the location of the dictionaries and data files that Sundance uses. If you wish to have Sundance use different dictionaries or data files, you can change the .nlp-opts file or make a local copy of it in the directory from which you plan to work and modify that copy. Sundance first checks the local working directory for a .nlp-opts file. If one isn't there, then Sundance uses the version in its data/ directory. However, if you plan to use customized dictionaries or data files, we recommend that you set up a new *domain* under Sundance and put the domain-specific files there. Section 5.5 explains how to set up a new domain under Sundance.

5.2 Using Sundance as a Stand-Alone Application

Sundance can be run by invoking the "nlp" executable in the bin/ directory. Figure 15 shows the command line options that it can take.

Command line option	Explanation
-wt	word tagging mode
-seg	segmentation (parsing) mode
-ext	information extraction mode
-f <filename>	give text file to parse
-l <slist>	give a list of text files to parse
-o <string>	append string to front of output filename (with -f option only)
-o <output-directory>	put output files in this directory (with -l option only)
-d <domain>	use dictionaries and data files for the specified domain
-c <caseframefile>	give a set of case frames to use for information extraction (-ext mode only)
-p shepherd	print extraction output in concise mode (-ext mode only)
-trace	show trace output for debugging purposes
-h	help

Figure 15: Command line options for Sundance (nlp)

Sundance can be run in 3 different modes: *-wt* (word tagging), *-seg* (segmentation), or *-ext* (extraction). These modes are mutually exclusive, so Sundance can be run in only one mode at a time.[20] *Word tagging mode (-wt)* is really just for debugging. It simply reports the part-of-speech tags that are found for a word first by looking up the word in Sundance's dictionaries, and

[20] In fact, the modes are cumulative. Segmentation mode includes word tagging, and extraction mode includes both word tagging and segmentation.

if that fails, by applying morphological analysis. *Segmentation mode (-seg)* performs parsing, as described in Section 3. *Extraction mode (-ext)* performs information extraction, as described in Section 4.

Input/Output Arguments

The -f option specifies a single file that Sundance should process. The -l option specifies a list of files that Sundance should process. In this case, the -l argument should be followed by what we call an *slist* file, which simply contains a directory path on the first line followed by one individual file name on each succeeding line. The directory path will be appended to the file names when Sundance looks for the files. Subdirectory names can be included as part of individual file names, allowing Sundance to access files in different subdirectories underneath a common directory. Figure 16 shows a sample *slist* file.

```
/home/riloff/dr_seuss/stories/
TheLorax.txt
Grinch.txt
CatInHat/CatInHatOriginal.txt
CatInHat/CatInHatComesBack.txt
```

Figure 16: A sample *slist* file

If neither the -f option nor the -l option is given, then Sundance will enter interactive mode by default. Sundance can be used interactively by typing sentences at the prompt. To exit the program, a user must type Control-D. Using Sundance interactively can be useful to familiarize oneself with Sundance, or to see how Sundance processed an individual sentence or text snippet.

When using the -f option to specify an input file, by default Sundance will produce an output file that has exactly the same name but with a suffix appended to it. It will append a *.tags* suffix to files processed in -wt mode, a *.sundance* suffix to files processed in -seg mode, and a *.cases* suffix to files processed in -ext mode. In combination with the -f option, the -o option can be used to append a string to the beginning of the output file name. For example:

```
nlp -seg -f Lorax.txt -o July28_
```

will produce an output file named `July28_Lorax.txt.sundance`. When using the -l option to specify a list of input files, the -o argument should be a directory where the output files are to be placed.

Other Arguments

As we will describe in Section 5.5, Sundance allows users to set up special dictionary and data files for a specific "domain". Using this capability, a user can customize Sundance for a specialized domain. To use domain-specific dictionary and data files, Sundance should be invoked with the "-d <domain>" option, where the domain can be one of the predefined domains, or a user-created domain as described in Section 5.5.

The "-c <caseframefile>" option gives Sundance a specific set of case frames to use for information extraction. If you are working within a domain, then a case frames file may already

be specified for that domain in its .domain-spec file (see Section 5.5). But if you wish to use a different set of case frames, or are running the general version of Sundance without a specified domain, then the -c option should be used. A file of case frames can be created manually, or can be generated automatically by AutoSlog or AutoSlog-TS.

The "-p shepherd" option is a concise printing mode that can be used for information extraction output. In this mode, Sundance simply prints each extracted noun phrase followed by an asterisk followed by the name of the case frame that extracted it. For example, given the sentence: *"John F. Kennedy was killed by Lee Harvey Oswald with a gun."*, Sundance will produce the following output:

John F. Kennedy * killed_1
Lee Harvey Oswald * killed_2
a gun * killed_3

where the killed_1 case frame represents the pattern *"<subj> killed"*, the killed_2 case frame represents the pattern *"killed by <np>"*, and the killed_3 case frame represents the pattern *"killed with <np>"*. We have found the "-p shepherd" printing mode to be very useful for gathering extraction data over a large corpus.

Finally, the "-trace" option prints Sundance's parse output after each stage of the parsing process. This is very verbose and should be used only for debugging purposes. It can be helpful to discover exactly where in the parsing process a problem is occurring.

5.3 Using Sundance as an API

Sundance also provides for use as an API. Using Sundance as an API allows users to embed calls to Sundance in other application programs, and it allows users to directly access Sundance's internal data structures. This may be useful if more information is needed from Sundance than what it prints by default. However, accessing Sundance's data structures requires knowledge of the Sundance code, which is beyond the scope of this report. We recommend that people who have minimal familiarity with Sundance should use it as an API only to embed it in other applications in a straightforward manner. Since Sundance is written in C++, we will only discuss it's usage as an API in C++ code. The Makefile located in src/ does allow for dynamic libraries to be created, however, making it possible to also call Sundance from within java code.

There are three things that need to be done to set up the API. First, the `sundance.h` file needs to be included in your source code. This file contains references to many header files that you will need. This can be added to your source code by including the statement:

```
#include "sundance.h"
```

Next, you will need to initialize Sundance before using it. Simply include a call to the global function "init_sundance()" before using Sundance. This assumes that you will be using Sundance in segmentation (parsing) mode, and you will not be running it in a special domain. If you wish to use Sundance in extraction mode or want to invoke it with additional arguments (e.g., to specify a domain), you will need to call the "init_sundance_CL(argc, argv)" initialization function, where argc is an integer and argv is a char**. This will allow you to specify the -ext and -d arguments on the command line and your program will pass these arguments to Sundance.

The third step is to add the location of the Sundance files and static library to your Makefile. Let's assume that SUNDANCE_HOME is the full pathname of your Sundance directory. Then you can set up the variable SUNDANCE_INCLUDE in your Makefile to point to your Sundance files, and use the variable SUNDANCE_LIB to specify the Sundance library information. You should set up these variables in your Makefile as follows:

SUNDANCE_INCLUDE = -I $(SUNDANCE_HOME)/src/ -I $(SUNDANCE_HOME)/include/
SUNDANCE_LIB = -L $(SUNDANCE_HOME)/lib/ -l sundance

These variables can then be used as needed for compiling and linking to your code throughout your Makefile. If you are not familiar with how to do this, there is a simple example of a Makefile that uses Sundance as an API in docs/sample_code/.

You are now ready to use the API. To do this you will first want to declare an object of the Sentence class. You can then apply the input operator directly to this object from an input stream. This stream can either be cin or a file stream that you have opened. The input operator will remove one sentence at a time from your stream. To parse this sentence, invoke the "process()" method that belongs to the Sentence class. You can then use the output operator on your object to print the parsed sentence to an output stream. Figure 5.3 shows a simple example of code that uses Sundance as an API. This program prints out the parse of sentences that are entered interactively by a user, until Control-D exits the program.

```
#include <iostream.h>
#include "sundance.h"

int main(int argc, char** argv) {

  // Arguments to Sundance can be specified on the command line
  init_sundance_CL(argc, argv);
  Sentence Sent;

  // Print a welcome message and a prompt.
  cout << "Welcome to Sample Program 1. "
  cout << "Quit by pressing Ctrl-D" << endl;
  cout << ">";

  // Setup a loop which reads input from standard in
  while (cin >> Sent) {
    Sent.process();
    cout << S;
    cout << ">";
  }
}
```

Figure 17: Sample API program

When using the Sundance API in extraction mode, instead of defining a Sentence object and calling the process() function, you must invoke the global function "do_extraction(istream&, ostream&)". This will process all of the sentences sent in via the input stream ("istream"). The resulting extractions will be sent to the output stream ("ostream").

5.4 Using AutoSlog and AutoSlog-TS as Stand-Alone Applications

AutoSlog can be run by invoking the "aslog" executable in the bin/ directory. Figure 18 shows the command line options that AutoSlog can take. Many of the options are the same as those for Sundance, so here we will only describe the ones that are specific to AutoSlog, which are the last five options shown in Figure 18. The "aslog" executable serves double duty as both an implementation of the AutoSlog supervised learning algorithm (see Section 4.2), and the AutoSlog-TS weakly supervised learning algorithm (see Section 4.3). In the following sections we describe how to use it in both ways.

Command line option	Explanation
-f <filename>	give a text file to parse
-l <slist>	give a list of text files to parse
-o <string>	append string to front of output filename (with -f option only)
-o <output-directory>	put output files in this directory (with -l option only)
-d <domain>	use dictionaries and data files for the specified domain
-h	help
-n <nouns-file>	give a list of targeted nouns to AutoSlog
-r <rel-file>	give a list identifying the relevant texts for AutoSlog-TS
-2 <caseframefile>	perform the second pass of AutoSlog-TS with this case frame file
-u <number>	give frequency cutoff for case frames produced by AutoSlog-TS
-t <caseframe_templates>	specify case frame templates to use for learning

Figure 18: Command line options for AutoSlog and AutoSlog-TS (aslog)

5.4.1 Running AutoSlog

To run the AutoSlog algorithm, you must use the -n argument followed by the name of a file that contains the targeted nouns. This file should simply contain nouns, one per line. AutoSlog will generate a pattern to extract every noun phrase whose head noun matches a noun in this file. The case frames that are generated will be put in a file called *<filename>.aslog*, where *<filename>* is the text file that was processed. If the -l option is used, multiple .aslog files will be created, one for each file in the list.

5.4.2 Running AutoSlog-TS

The first pass

To run the AutoSlog-TS algorithm, you should call "aslog" with the -l, -r, and -u arguments. AutoSlog-TS takes one set of "relevant" texts and one set of "irrelevant" texts as input, and then generates extraction patterns found in those texts along with statistics indicating how strongly each

pattern is correlated with the relevant texts. Both the relevant and the irrelevant text files should be included in the slist file that is specified with the -l option. The "-r <rel-file>" argument then indicates which of those texts are relevant (i.e., which ones correspond to the topic of interest). The <rel-file> should be a file that simply lists the names of the relevant text files, one per line. There should not be directory paths in the <rel-file>.[21] The "aslog" program will generate every extraction pattern that appears in the texts, along with statistics indicating how many occurrences of each pattern were found in the relevant texts vs. the irrelevant texts. The -u <number> argument specifies a cutoff value based on the frequency of occurrences. Given the number N, aslog will only retain case frames that appeared $\geq N$ times in the corpus (the relevant and irrelevant texts combined). This is a useful argument because the number of generated case frames can be enormous, often in the tens of thousands. Eliminating the case frames that were seen only once in the corpus, for example, is often worthwhile because those case frames typically aren't very useful and discarding them can cut the size of the learned case frame file in half. To discard case frames that have a frequency of exactly 1, you should use the argument "-u 2". To retain all of the learned case frames, you should give the argument "-u 1".

```
CF:
Name: <subj>_passive_verb_1
Anchor: VP1(PURCHASED)
Act_Fcns: passive_verb_broad_p(VP1(PURCHASED))
Slot: subj
# frequency: 4
# relevant_freq: 3
# cond. probability: 0.750
# relevance score: 1.189
-------------------------------------------------
CF:
Name: infinitive_verb_prep<np>_1
Anchor: VP1(PURCHASE)
Act_Fcns: infinitive_verb_p(VP1(PURCHASE))
Slot: PP(WITH)
# frequency: 5
# relevant_freq: 4
# cond. probability: 0.400
# relevance score: 0.400
```

Figure 19: Sample .caseframes file produced by AutoSlog-TS

When the first pass of AutoSlog-TS is finished, two files will be created: "<slist>.caseframes" and "<slist>.stats1", where slist is the name of the file specified by the -l argument. The ".caseframes" file contains the actual case frame structures generated by AutoSlog-TS. This case frame

[21] There is one problem that can arise from the lack of directory paths in this file. If two files under different directories have the same name, but one is relevant and the other is irrelevant, there is no way to distinguish them. If the file name is put in the rel-file, both files will be considered relevant. If it is not put in the rel-file, both files will be considered irrelevant.

file can be given to Sundance to use for information extraction. Figure 19 shows an example .caseframes file. There are statistics printed below each case frame, but they are commented out and will not be read by Sundance. We print the statistics in this file simply for convenience, so that people can see how each case frame performed during the training phase.

The second file created by AutoSlog-TS is a ".stats" file, which contains the statistics associated with each case frame in a tabulated format (see Section 4.3 for descriptions of these statistics). Figure 20 shows an example .stats file. The first column shows the frequency of the case frame in the corpus. The second column shows its frequency in the relevant texts only. The third column shows the conditional probability $P(relevant \mid pattern_i)$ and the fourth column is the RlogF score. The fifth column shows a shorthand notation for each case frame. These are not actual case frame structures, but simply a brief description of the pattern that the case frame matches. The purpose of this .stats file is to have a file of tabulated statistics that can be easily sorted with the unix `sort` command. This will prove to be useful very shortly when we discuss the case frame filtering step. This .stats file is meant for inspection purposes only and is not used by Sundance or AutoSlog-TS in any way.

4	3	0.750	1.189	"passive_verb_broad_p PURCHASED"
5	4	0.400	0.400	"infinitive_verb_p PURCHASE WITH"

Figure 20: Sample .stats file produced by AutoSlog-TS

The statistics generated by the first pass of AutoSlog-TS are only preliminary. To get the final, most accurate statistics associated with the AutoSlog-TS algorithm, you must perform the second pass of the algorithm.

The second pass

The second pass of AutoSlog-TS is invoked by calling "aslog" with a "-2 <caseframefile>" argument, where <caseframefile> is the file of case frames generated by the first pass of the algorithm. The rest of the arguments are the same as those for the first pass: -l, -r, and -u options should be used with the same filenames supplied during the first pass. As output, the second pass also produces two files that are analogous to those produced during the first pass: "<slist>.caseframes_2" and "<slist>.stats2", where slist is the name of the file specified by the -l argument. The .caseframes_2 file contains the complete set of case frames learned by AutoSlog-TS after its second pass, and the .stats2 file contains the final tabulated statistics associated with those case frames.

Filtering the case frames

The ".caseframes_2" file literally contains every case frame that AutoSlog-TS discovered in the corpus (minus those discarded based on the cutoff frequency specified with the -u option). Typically, the vast majority of these case frames are not useful for a specific information extraction task. To separate out the case frames that are useful, the user must filter the ".caseframes_2" file to find the case frames that are most strongly associated with the domain. The purpose of the ".stats2" file is simply to list the various statistics associated with each case frame so that users can easily

sort the case frames based on these statistics (say, using the unix `sort` command) and then decide what thresholds they want to use.

Once you have decided on the best thresholds for your application, you must use the `filter` program in the tools/ directory and apply these thresholds to the ".caseframes_2" file. There is a *README_filter* file in the tools/ directory explaining the types of thresholds that are possible and how to apply them. Simply run the `filter` program on the ".caseframes_2" file, and the resulting output file will contain only the case frames that pass those thresholds.

An example of how to run AutoSlog-TS

As an example, here is a complete execution sequence that could be used to run AutoSlog-TS:

```
aslog -r relfiles.txt -l corpus.slist -u 2

aslog -2 corpus.slist.caseframes -r relfiles.txt
      -l corpus.slist -u 2

filter -f corpus.slist.caseframes_2 -o corpus.slist.filtered_cfs
       -freq 5 -prob .80
```

This process will produce a file called `corpus.slist.filtered_cfs` that will contain all of the case frames generated by AutoSlog-TS that occurred at least 5 times in the corpus and also had at least 80% of those occurrences in relevant texts. This file represents the final set of case frames that should be used for information extraction.

5.4.3 Using New Case Frame Templates

As described in Section 4.4, users have the ability to define their own case frame templates for AutoSlog and AutoSlog-TS to use during learning. The case frame templates that AutoSlog uses are read from a text file at runtime. If AutoSlog is run in a specified domain, then the case frame templates file specified by the .domain-spec file will be used. If you wish to use a different set of case frame templates, you can specify the file with the "-t <caseframe_templates>" command line option. The example below shows a call to AutoSlog that tells it to use the file "my_templates.txt" as the case frame templates.

```
aslog -l corpus.slist -n nouns.txt -t my_templates.txt
```

5.5 Setting up a Specialized Domain

Sundance was designed to be a general-purpose shallow parser that should do a reasonably good job of parsing texts in any domain. But a central aspect of its design was also to support domain-specific text processing where specialized dictionaries and data files might do a better job than general resources. Sundance uses general dictionaries and general data files in its default mode of operation. However, Sundance also gives users the ability to set up a specific *domain* that contains special dictionaries and data files. These files can augment or replace the default files. A new

domain can be defined without having to alter any of Sundance's source code. To create a new domain, three things must be done.

(1) Create a subdirectory for the domain under data/.

(2) Define the new domain in the .nlp-opts file. The .nlp-opts file contains a section consisting of several lines that begin with "domain_spec". These lines list the domains that currently exist and indicate where their specification files are located. By convention, each specification file ends with the extension ".domain-spec" and should be located in a directory called data/*domain*/. For example, if you wanted to create a domain called "sports", you might add a line to the .nlp-opts file like this:

```
domain_spec sports /home/name/sundance-v4.2/data/sports/.domain-spec
```

(3) Create the .domain-spec file that you just listed. The format for this file can be easily copied from an existing .domain-spec file, such as the one already defined for the terrorism domain in data/terror/.domain-spec. This file tells Sundance the location of the special dictionaries and data files that are associated with the domain. For some domains, you may wish to significantly modify the behavior of Sundance by creating many specialized dictionaries and data files. For other domains, you may wish to create only a single specialized dictionary or data file. When setting up a new domain, you have the option to create domain-specific versions of any of the following files:

Case Frame Dictionary (*caseframefile*)**:** A set of case frames to extract information from texts in the domain.

Append Dictionary (*domain_append_dict*)**:** A dictionary of domain-specific words. When running Sundance in your domain, Sundance's default dictionary as well as the domain-specific append dictionary will be accessed (i.e., the definitions in the append dictionary are effectively unioned with those in the general dictionary). The append dictionary should contain domain-specific words that you might not want to put in the general dictionary for several reasons. (1) Highly-specialized terms (e.g., medical terms) would virtually never be needed outside of the domain. Putting them in a separate domain-specific dictionary will keep the size of the general dictionary more manageable. (2) Some words may have a special usage within a domain. For example, in a general dictionary the word "apple" might simply be defined as a noun. But in a business domain, most occurrences of the word "apple" might refer to the company Apple Computer. In this case, it would make sense to assign the semantic category COMPANY to the word "apple" in the append dictionary for a business domain.

Override Dictionary (*domain_override_dict*)**:** A dictionary of domain-specific words that have a different use in the specialized domain than they normally have. When a word is defined in the override dictionary, the new definitions become the sole definitions for the word. Any definitions of the word that are found in Sundance's default dictionary will be ignored. For example, the word *pound* is frequently mentioned in business domains as a form of money. A general dictionary would define *pound* as both a verb and a noun. Ideally, Sundance's parsing heuristics should correctly disambiguate the word, but its disambiguation heuristics are not perfect. If Sundance were to consistently misparse *pound* as a verb in business texts, then

one solution would be to redefine *pound* only as a noun in the business domain's override dictionary. Assuming that *pound* really is used as a noun most of the time in that domain, this would probably improve parsing performance. The override dictionary can also be used to override semantic features assigned to a word. For example, if the word "apple" was explicitly defined as a FOOD in the general dictionary, we could redefine it as a COMPANY in the business domain's override dictionary and the FOOD label would be ignored.

Phrasal Dictionary (*domain_phrases_file*): A dictionary of lexicalized phrases that are relevant to the domain. These phrases will be unioned with those in the Sundance's default phrasal dictionary.

EOS Files (*domain_eos_followers_file* and *domain_non_eos_file*): Additional lists of EOS_followers and Non_EOS words can be specified for the domain. These lists will be unioned with the default EOS lists used by Sundance.

Semantic Hierarchy (*domain_semantic_hierarchy*): A semantic hierarchy for the domain. The domain-specific semantic hierarchy will replace the general one.

Recognizer Patterns (*domain_recognizerpatternfile*): A new set of recognizer patterns for the domain. To create a new entity type (e.g., NP-LOCATION and NP-COMPANY), the new type must be added to Sundance in three places. (1) It must be added to the PatternType enumeration list in src/tags.h. (2) It must be added to the PatternTypeToString function in src/tags.C. (3) It must be added to the StringToPatternType function in src/tags.C.

5.6 Summary

Well, I think that does it! You are now fully armed with all the knowledge you should need to use Sundance and AutoSlog effectively. Although we can offer no promises in terms of producing new releases of our software in any specific time frame, we welcome any feedback that you may have, such as bug reports, problematic parsing or learning behavior, general suggestions for improvement, and reports of positive experiences as well. Please send all comments to riloff@cs.utah.edu.

6 Acknowledgments

This research was supported in part by the National Science Foundation under awards IRI-9704240 and IIS-0208985, and this material is based upon work supported by the Advanced Research and Development Activity (ARDA) under Contract No. NBCHC040012. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARDA or the Department of Interior-National Business Center (DOI-NBC). We are very grateful to several programmers who helped to develop the Sundance and AutoSlog systems over the years, most notably David Bean who wrote substantial portions of the Sundance code. We are also grateful to Siddharth Patwardhan for reviewing a draft of this report.

7 Appendix A: The Source and Data Files

Tables 1 and 2 give a brief overview of the source code files used by Sundance and AutoSlog. The first column shows the file name and the second column briefly describes the contents of the file. At the end of each description we indicate which program (nlp or aslog) primarily uses the code in the file.[22] All these files are found under the src/ or include/ directories.

Source File	Description
activationFcns.h/.C	matches the extraction patterns with supplied clauses; finds potential extractions (nlp)
activationFcnPatterns.C	checks if constraints for activationFcns met in creating case frames, part of processFile class (aslog)
aslogMain.C	main driver function (aslog)
caseFrameTemplates.h/.C	reads in definitions of case frame templates (aslog)
caseFrames.h/.C	provides data structures (d.s.) and operations needed for working with case frames (aslog & nlp)
clause.C	clause handling functions (nlp)
clist.h/.C	contains constituentlist class that is used primarily by Constituent class to implement the class member children (nlp)
commandLine.h/.C	stores global variables obtained from the command line (aslog)
constituent.h/.C	the main class that contains sentence components. Words, phrases, and clauses are all constituents (nlp)
date.h/.C	a constituent just for maintaining numerical dates (nlp)
dictionary.h/.C	reads in and stores all words from dictionaries (text files). Provides access to words and also performs morphology (nlp)
extractions.h/.C	class that holds extractions and related information (nlp)
main.C	main driver function (nlp). Includes command line parsing and text handling functionalities.
miscfunc.h/.C	includes main functions for wordtagging, parsing, and extracting. Also includes many functions implementing details of parsing (nlp)
miscglobals.h/.C	global information needed for parsing (nlp)
morphology.h/.C	morphological analysis functions (nlp)
paren.h/.C	a constituent for some punctuation handling (nlp)
pos.h/.C	a part of speech class used by the word class. Also contains classes with supporting information for the parts of speech (nlp)
prepro.h/.C	contains global information needed to perform the preprocessing steps of Sundance, including sentence breaks (nlp)
preproMain.C	main driver function (prepro)

Table 1: Source Files (Part I)

[22] AutoSlog is built on top of Sundance, so technically AutoSlog uses all of Sundance's code as well as its own.

Source File	Description
processFile.h/.C	finds all case frames in text that can be created from given case frame templates (aslog)
recogpatterns.h/.C	named entity recognition functions (nlp)
segment.C	implementation for the sentence class methods related to segmenting. Implemented to reduce size of Sentence class file (nlp)
semtree.h/.C	semantic hierarchy functions (nlp)
sense.h/.C	word sense handling functions – not currently used but still integrated into the code (nlp)
sentence.h/.C	holds words in sentence and implements much of segmenting, clause handling, and extracting (nlp)
sundance.h	contains header files needed to use Sundance as an API (nlp)
synrole.h/.C	specifies all possible syntactic roles a phrase can take (nlp)
tags.h/.C	contains enumerated legal values for POS, syntactic features, patterntypes, and constituentTypes (nlp)
texthandler.h/.C	functionality for dealing with different types of text input, including cin (nlp)
word.h/.C	class with all d.s. and functionality needed for words (nlp)
dst.h	a template for a digital search tree (all)
filehelp.h/.C	helper functions for retrieving data from files, including skipping headers for domains (nlp)
hash.h	template for a hashtable (all)
intset.h/.C	an integer set class used for represented enumerated list members (all)
list.h	a template for a list class used throughout Sundance – has features of both a list and vector class (all)
ntree	template for a ntree (all)
strutil.h/.C	utility for retrieving and sending list of sunstr from files (all)
sunstr.h/.C	a string class used throughout Sundance (all)

Table 2: Source Files (Part II)

Data File	Description
.domain-spec	defines location of files for a domain
.nlp-opts	defines location of files for Sundance (nlp)
append_dict.txt	domain-specific dictionary to be appended to the default dictionary
caseframe_templates.txt	defines all the case frame templates for aslog
caseframes.txt	case frames for use by nlp -ext
closed-dict.txt	default dictionary of closed-class words
dictionary.txt	default dictionary of open-class words
eos_followers.txt	end-of-sentence follower words
non_eos.txt	words that are not end-of-sentence words
override.txt	domain-specific dictionary to override default dictionary
phrasal.txt	phrasal lexicon
recognizer_patterns.txt	rules for recognizer_patterns
rules.txt	morphology rules
semtree.txt	semantic hierarchy

Table 3: Dictionaries and Data Files

References

Steven Abney. 1990a. Partial parsing with finite-state cascades. In John Carroll, editor, *Workshop on Robust Parsing (ESSLLI '96)*, pages 8–15.

Steven Abney. 1990b. Rapid incremental parsing with repair. In *Proceedings of the 6th New OED Conference: Electronic Text Research*, pages 1–9. University of Waterloo.

D. Appelt, J. Hobbs, J. Bear, D. Israel, and M. Tyson. 1993. FASTUS: a finite-state processor for information extraction from real-world text. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*.

K. Church. 1989. A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text. In *Proceedings of the Second Conference on Applied Natural Language Processing*.

Dayne Freitag. 1998. Toward General-Purpose Learning for Information Extraction. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics*.

J. Furnkranz, T. Mitchell, and E. Riloff. 1998. A Case Study in Using Linguistic Phrases for Text Categorization from the WWW. In *Working Notes of the AAAI/ICML Workshop on Learning for Text Categorization.*

T. Geisel. 1971. *The Lorax*. Random House, New York.

R. Grishman. 1995. The NYU System for MUC-6 or Where's the Syntax? In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*, San Francisco, CA. Morgan Kaufmann.

Donald Hindle. 1994. A parser for text corpora. In A. Zampolli, editor, *Computational Approaches to the Lexicon*, pages 103–151. Oxford University Press.

S. Huffman. 1996. Learning information extraction patterns from examples. In Stefan Wermter, Ellen Riloff, and Gabriele Scheler, editors, *Connectionist, Statistical, and Symbolic Approaches to Learning for Natural Language Processing*, pages 246–260. Springer-Verlag, Berlin.

J. Kim and D. Moldovan. 1993. Acquisition of Semantic Patterns for Information Extraction from Corpora. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications*, pages 171–176, Los Alamitos, CA. IEEE Computer Society Press.

N. Kushmerick, D. Weld, and R. Doorenbos. 1997. Wrapper Induction for information extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 729–735.

E. Riloff and R. Jones. 1999. Learning Dictionaries for Information Extraction by Multi-Level Bootstrapping. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*.

E. Riloff and J. Wiebe. 2003. Learning Extraction Patterns for Subjective Expressions. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*.

E. Riloff. 1993. Automatically Constructing a Dictionary for Information Extraction Tasks. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 811–816. AAAI Press/The MIT Press.

E. Riloff. 1996a. An Empirical Study of Automated Dictionary Construction for Information Extraction in Three Domains. *Artificial Intelligence*, 85:101–134.

E. Riloff. 1996b. Automatically Generating Extraction Patterns from Untagged Text. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1044–1049. The AAAI Press/MIT Press.

S. Soderland, D. Fisher, J. Aseltine, and W. Lehnert. 1995. CRYSTAL: Inducing a conceptual dictionary. In *Proc. of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1314–1319.