# Multi-Pass Multi-Strategy NLP

**Amnon Meyers**
Text Analysis International, Inc.
1669-2 Hollenbeck Ave. #501
Sunnyvale, CA 94087 USA
amnon.meyers@textanalysis.com

**Abstract**

We describe a novel multi-pass, multi-strategy architecture for natural language processing (NLP). The commercial integrated development environment (IDE), VisualText(TM), and the associated NLP++(TM) programming language, as well as derived applications, serve to illustrate the architecture and methodology.

## 1 Introduction

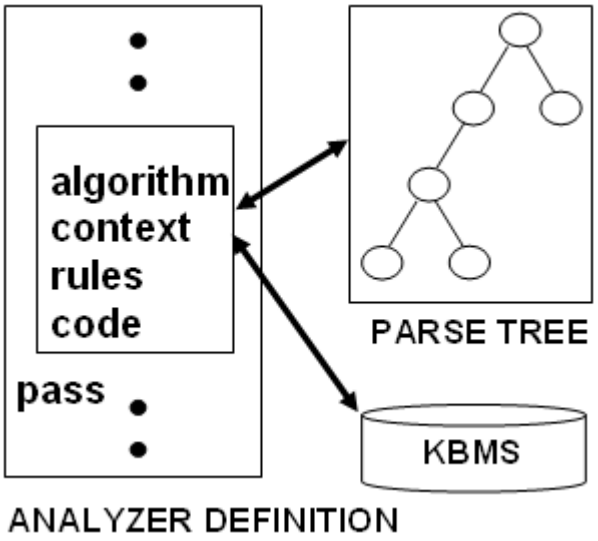


Figure 1. The Multi-Pass Architecture

Practical NLP systems must encompass a variety of methods in order to support lexical, syntactic, semantic, discourse, and pragmatic processing for a real-world task. Two primary ways of achieving this are: (1) develop the individual components and integrate them later; (2) develop the overall system in integrated fashion.

We focus here on method (2). Further, we describe an architecture (Figure 1) in which an arbitrary number of processing passes may be elaborated, where each pass invokes its own processing algorithm and uses its own data. For example, one pass may use a pattern-based matching algorithm and an associated set of rules to recognize location constructs in a text.

Each pass can be viewed as a YACC or Bison grammar with its attendant code actions.

The passes of the multi-pass architecture are constrained to share a single parse tree. Each pass receives the cumulative parse tree, elaborates it further, then hands it to a subsequent pass. While the constraint to a single parse tree appears to limit the handling of ambiguity, we touch on various methods that address ambiguity within the architecture.

In addition to managing a unique parse tree, the passes may also update and access an integrated knowledge base (KB), as well as general programmatic data structures. The VisualText IDE (TAI, 2003) uses a hierarchical knowledge base management system (KBMS), for mapping knowledge in a more natural fashion than a relational database.

A unifying programming language is a key component of the architecture, enabling NLP developers to manage the passes, parse tree, associated knowledge and programming constructs.

Some advantages of the architecture:

- Integration. A spectrum of strategies and levels of granularity can interact within a single framework.
- Modularity, extensibility, maintainability. Passes can be elaborated such that each performs a simple and modular task.
- Flexibility. Passes may be added to the framework in any order, so that work that was initially glossed over or forgotten may be added in at a later point.
- Feedback, deferral. Rather than making uninformed decisions upfront, passes can be implemented to gather data and "try things out", in order to increase confidence in subsequent processing decisions.
- Context. By splitting the NLP system into multiple passes, each pass may be constrained to operate on particular contexts. For example, a parsing pass might work only within noun phrases or within the header zones of a text being analyzed.
- Automation. Passes within the architecture can dynamically create and execute new passes, e.g., based on patterns learned from a corpus.

The architecture can be readily embedded within a programming language and implemented within a graphical development environment, affording substantial efficiencies for language engineering. Analyzers and sequences of passes can be configured for reuse as templates or as libraries.

While the architecture accommodates various application types, such as information extraction, full-scale parsing, language generation and translation, we shall assume that the primary application under discussion is "analysis", such as information extraction.

We refer to the NLP++ programming language (TAI, 2003) and its implementation in the commercial VisualText IDE as exemplars of this methodology.

We discuss applications built using the multi-pass methodology, including the TAIParse general analyzer and Resume Analyzer developed with NLP++.

## 2 Background

TexUS (de Hilster and Meyers, 1991; Meyers and de Hilster, 1992) constitutes an early version of a multi-pass, multi-strategy NLP architecture. TexUS lacked a unifying programming language, but enabled passes to be matched with rule sets and declaratively maintained the NLP system definition, knowledge, rules, and even parse trees within a unitary KBMS. With a scant 2 person-month customization effort using a 9-month-old IDE, a TexUS-built analyzer achieved an impressive result at MUC-3 (de Hilster and Meyers, 1991).

A treatment of NLP architectures and integrated development environments appears in (TAI, 2001).

Architectures based mainly on a single paradigm such as cascaded finite state transducers, e.g., INTEX (Silberztein, 2000) and FASTUS (Appelt et al, 1993), or pattern-based processing, e.g., DEFT (Noah and Weeks, 1993), are known, as are hybrid architectures, e.g., GATE (Gaizauskas et al, 1996) and PAKTUS (Loatman, 1991).

## 3 TAIParse Application

We describe the architecture as applied to the TAIParse[1] general analyzer, which performs lexical, syntactic, and semantic processing in integrated fashion.

A guiding philosophy in elaborating TAIParse is to use knowledge only as necessary, that is, to prefer heuristic methods wherever possible rather than embedding large lists of data in the analyzer.

TAIParse currently comprises about 120 passes, most of which use a workhorse pattern-based pass algorithm, and some of which use a recursive grammar algorithm (e.g., for collecting lists). The passes can be roughly segregated into regions as follows

Tokenize
Zoner
Lexical
Phrasal
Segmenter
Parser
Semantics
Discourse

A standard first pass performs tokenization, that is, conversion of the characters of an input file to tokens or units of alphabetic, numeric, punctuation, and whitespace characters. Other passes are also concerned with tokenization issues, for example,

[1] Analyzer project and definition available from http://www.textanalysis.com; requires VisualText to run.

reasoning about whether two tokens should be joined (e.g., "cross current").

The Zoner is interesting in that it performs parsing operations and other evidence-gathering on isolated lines of text. In this way, it characterizes individual lines and the relationships among them, in order to decide upon the best separation of sentences, paragraphs, headers, and other text regions. One type of evidence pertains to the start and end of each line. For example, if a line ends with an English function word such as "the", this adds evidence for the presence of a prose or sentential region of lines.

While TAIParse minimizes the use of pre-built knowledge, a list of English words and their possible syntax classes is included in the system. The Lexical passes utilize that knowledge to characterize words (e.g., as known, unknown, spelling errors). Part-of-speech (POS) tagging is distributed throughout TAIParse. Syntactically unambiguous words are tagged early on by the Lexical passes, but tagging of ambiguous words is deferred to passes dealing with clausal patterns, in order to utilize context to enable informed POS tagging.

As the passes unfold, nodes for lines and white space may be excised from the parse tree. This can be done selectively within nodes of the parse tree labeled as text regions.

Phrasal passes occur at various points to recognize relevant idioms and collocations.

The Segmenter creates nodes (called segments) primarily based on boundaries such as English function words ("the", "is", "of") and prose punctuation. Subsequent passes then reason about the content and structure of isolated segments, as well as about the context surrounding segments. In some cases, segments are re-segmented, for example to separate a verb from the start of a noun phrase. Since actions deemed to be erroneous may be undone or redone, this is one means for handling ambiguous constructs.

The Parser passes are primarily pattern-based rather than relying on recursive grammar algorithms. Segment resolution passes are interspersed with chunking and "parsing" passes, so as to use feedback to assign segments and their boundaries with greater confidence.

The Semantic and Discourse passes utilize the parse tree, parse tree "semantics" (i.e., data layered into the parse tree nodes), and data schemas within the KB in order to perform tasks such as anaphora resolution, correlation of the events described in a text, and application-specific processing.

In summary, the architecture allows analyzers such as TAIParse to use characterization, feedback, deferral, and context in order to make informed decisions during the analysis of a text.

## 4 Other Applications

A Resume Analyzer prototype[2] for web resumes is noteworthy in several respects. With 6 person-months of development and elaboration of 250 passes, it achieves about an 80% F-Measure (90% precision and 75% recall) in extracting contact, experience, and education records from unseen resumes. Resumes are difficult to process accurately due to a multiplicity of formats and mangling of formats in converting, e.g., HTML to plain text.

The Resume Analyzer makes substantial use of automated generation of passes from annotated resumes. It features a sophisticated zoning capability, as well as a confidence-based treatment of generalized capitalized phrases. E.g., some passes reason about boundaries between capitalized phrases and criteria for merging proximate capitalized phrases.

Some recent applications based on VisualText and NLP++ involve analyzers invoking each other as experts. For example, a focused crawler is under development in which one analyzer manages the list of URLs to be crawled and tracks URLs that have been visited, while invoking a second analyzer to reason about links to pursue in a website in order to locate an organization's employees. A third analyzer specializes in web pages containing lists of people, while a fourth specializes in processing an individual's home page.

## 5 NLP++ Programming Language

An architecture for NLP is incomplete without a programming language to serve as its "glue." While a standard programming language with added libraries for the architecture is helpful, our experience indicates that a programming language specialized for the architecture substantially enhances its value.

For the multi-pass, multi-strategy architecture described here, the programming language should

[2] Also available from http://www.textanalysis.com.

define the analyzer, including algorithms, rules, and associated code.

```
@CODE
  G("count nps") = 0;      # Initialize counter for nps.

  # Create a KB concept for storing noun phrases.
  G("kb nps") = makeconcept(findroot(), "nps");
@@CODE

# Constrain rule-matching context to _sentence nodes.
@PATH _ROOT _paragraph _sentence

# Recognize a noun phrase.
@POST
  ++G("count nps");  # Increment global count of nps.
  ++X("nps",2);      # Increment counter in _paragraph node.
  S("nouns") = N(5); # _np node gets a pointer to _nouns node.

  # Make a KB concept for this noun phrase.
  L("con") = makeconcept(G("kb nps"), str(G("count nps")));

  # Add the noun phrase's text as an attribute named "text"
  # attached to the KB concept representing the noun phrase.
  replaceval(L("con"),"text", phrasetext());

  single();    # Normal rule reduce to create _np node.
@RULES
_np <-
    _det
    _xWILD [star match=(_adv _advl) group=_advls]
    _quan
    _adj
    _xWILD [plus match=(_noun) group=_nouns]
    @@
```

Figure 2. NLP++ Pass File Code Sample

It should address the parse tree, the semantic information decorating the parse tree, and the objects within the associated knowledge base. It should provide built-in functions that support all the needs of the NLP developer.

Figure 2 depicts a sample of NLP++ code for a single pass file with a single rule, which we describe to give a flavor of a language tailored for an NLP architecture. Some lines are explained by the comments that precede them or that occur at the end of the line.

We assume that the pass algorithm is pattern-based, though the same syntax works for recursive-grammar passes as well.

NLP++ uses the at-sign ("@") to mark boundaries in a pass file. Thus, @CODE denotes the start of a rule-independent code region, while the optional @@CODE denotes the end of that region of the pass file. The @CODE region executes before rules, if any, in the pass file are executed by the (pattern-based) pass algorithm. A @DECL region, not shown, can precede the @CODE region, serving as a locus for user-defined NLP++ functions (which may be invoked anywhere in the analyzer).

Within the @CODE region, a global variable named "count nps" is created with initial value zero. G refers to global variables (whose scope is analyzer-wide for the analysis of the current input text).

Next, a global variable called "kb nps" is created. Its value is a KB concept named "nps", placed directly under the root of the hierarchical KB. The built-in NLP++ function *findroot* returns the root of the KB hierarchy, while the built-in *makeconcept(parent_concept, child_name_string)* creates a child-concept under a given KB concept.

The @PATH marker specifies the parse tree context in which patterns of the current pass will be matched. In this case, the pass algorithm searches directly under the root of the parse tree

(named "_ROOT") for nodes named "_paragraph" and below them for nodes named "_sentence." Rules in the pass file are constrained to match only under such "_sentence" nodes of the parse tree. Such context specifiers enable fine-grained control of pattern-matching within the NLP system and can help reduce spurious pattern matching.

Next is a @POST region, which contains code or actions to execute whenever a rule (within the immediately following @RULES region) matches a phrase directly under a "_sentence" node. C++ -like syntax is used to increment some variables. The X variable reference enables nodes in the @PATH or context to be accessed, while S references the suggested or reduced concept of the subsequent rules, N references the nodes in the right-hand-side phrase of rules, and L references local variables.

Explicit variable references help manage the many contexts and scopes inherent in NLP++, which effectively integrates pass, rule, and code syntax.

The one rule shown (starting with the line "_np <-") is representative of the rule or pattern syntax of NLP++. In this case "_np" denotes the suggested or reduce node to be created when the phrase following the "<-" arrow matches. The phrase consists of five elements, each on a separate line. The elements are _det, _xWILD, _quan, _adj, _xWILD. Each element is optionally followed by specifications within square brackets. The first _xWILD denotes a wildcard constrained to match _adv or _advl (e.g., an adverbial node), while the second _xWILD is constrained to match one or more adjacent nodes named _noun and to group them under a new node called "_nouns." The @@ marker denotes the end of the rule.

NLP++ exemplifies a language tailored for a multi-pass architecture, and which encompasses passes, code, contexts, rules, parse tree nodes, and knowledge base objects, as well as managing a variety of scopes and programming contexts (pass vs. rules vs. code).

## 6 Conclusion

We have described a multi-pass, multi-strategy architecture that operates on a single shared parse tree per text being processed. Each pass may execute its own algorithm and use its own data.

An associated KBMS can serve to store information across multiple input texts, as well as serving as a "sandbox" for arbitrary knowledge representation schemas. Primary uses are for managing entities in semantic and discourse processes of the NLP system.

Critical to the utility of the methodology described is a programming language that addresses all the facets of the architecture, including passes, parse trees, parse tree semantics, contexts, and knowledge base objects.

NLP++ and its implementation within the VisualText IDE, as well as derived applications, exemplify this type of architecture and language engineering methodology in practice.

## References


Douglas E. Appelt, Jerry R. Hobbs, John Bear, David Israel, Mabry Tyson. 1993. FASTUS: A Finite-State Processor for Information Extraction from Real-World Text. *IJCAI*. 1172-1178.

David de Hilster and Amnon Meyers. 1991. Description of the INLET System Used for MUC-3. *Proceedings of MUC-3*. DARPA. 178-182.

Robert Gaizauskas, Hamish Cunningham, Yorick Wilks, Peter Rodgers, Kevin Humphreys. 1996. GATE: An Environment to Support Research and Development in Natural Language Engineering. *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*. Toulouse, France.

Bruce Loatman. 1991. Description of the PAKTUS System Used for MUC-3. *Proceedings of MUC-3*. DARPA. 191-199.

Amnon Meyers and David de Hilster. 1992. Description of the TexUS System as Used for MUC-4. *Proceedings of MUC-4*. DARPA. 207-214.

William W. Noah and Rollin V. Weeks. 1993. Description of the DEFT System as Used for MUC-5. *Proceedings of MUC-5*. 237-248.

Max Silberztein. 2000. INTEX: An FST Toolbox. *Theoretical Computer Science*, 231(1):33-46.

TAI. 2001. *Integrated Development Environments for Natural Language Processing*. Text Analysis International. http://www.textanalysis.com/TAI-IDE-WP.pdf

TAI. 2003. VisualText Help. Text Analysis International. http://www.textanalysis.com/help/help.htm