THE COMMON PATTERN SPECIFICATION LANGUAGE

Douglas E. Appelt

Artificial Intelligence Center
SRI International
333 Ravenswood Ave, Menlo Park, CA

Boyan Onyshkevych

R525
Department of Defense
Ft. Meade MD

ABSTRACT

This paper describes the Common Pattern Specification Language (CPSL) that was developed during the TIPSTER program by a committee of researchers from the TIPSTER research sites. Many information extraction systems work by matching regular expressions over the lexical features of input symbols. CPSL was designed as a language for specifying such finite-state grammars for the purpose of specifying information extraction rules in a relatively system-independent way. The adoption of such a common language would enable the creation of shareable resources for the development of rule-based information extraction systems.

1. THE NEED FOR CPSL

As researchers have gained experience with information extraction systems, there has been some convergence of system architecture among those systems based on the knowledge engineering approach of developing sets of rules more or less by hand, targeted toward specific subjects. Some rule-based systems have achieved very high performance on such tasks as name identification. Ideally, developers of information extraction systems should be able to take advantage of the considerable effort that has gone into the development of such high-performance extraction system components. Unfortunately, this is usually impossible, in part because each system has a native formalism for rule specification, and the translation of rules from one native formalism to another is usually a slow, difficult, and error-prone process that ultimately discourages the sharing of system components or rule sets.

Over the course of the TIPSTER program and other information extraction efforts, many systems have converged on an architecture based on matching regular expression patterns over the lexical features of words in the input texts. The Common Pattern Specification Language (CPSL) was designed to take advantage of this convergence in architecture by providing a common formalism in which finite-state patterns could be represented. This would then enable the development of shareable libraries of finite-state patterns directed toward specific extraction tasks, and hopefully remove one of the primary barriers to the fast development of high-performance information extraction systems. Together with common lexicon standards and annotation standards, a developer can exploit previous domain or scenario customization efforts and make use of the insights and the hard work of others in the extraction community. The CPSL was designed by a committee consisting of a number of researchers from the Government and all of the TIPSTER research sites involved in Information Extraction that are represented in this volume.

2. INTERPRETER ASSUMPTIONS

A pattern language is intended to be interpreted. Indeed, the interpreter is what gives the syntax of the language its meaning. Therefore, CPSL was designed with a loosely specified *reference interpreter* in mind. It was realized that extraction systems may not work exactly like the reference interpreter, and it was certainly not the goal of the designers to stifle creativity in system design. However, it was hoped that any system that implemented at least the functionality of the reference interpreter would, given appropriate lexicons, be able to used published sharable resources.

The functionality assumed to be implemented by the reference interpreter is as follows:

The interpreter implements cascaded finite-state transducers.

Each transducer accepts as input a sequence of annotations conforming to the Annotation object specification of the TIPSTER Architecture [Grishman, this volume]. The fundamental operation performed by the interpreter is to test whether the next annotation in sequence has an attribute with a value specified by the grammar being interpreted.

Each transducer produces as output a sequence of annotations conforming to the Annotation object specification of the TIPSTER Architecture.

The interpreter maintains a "cursor" marking the current position in the text. All possible rules are matched at each point. One of the matching rules is selected as a "best match" and is applied. The application of a rule results in the creation of new annotations, and in moving the "cursor" to a new position.

The interpreter does an initial tokenization and lexical lookup on the input. Each lexical input item is marked with a Word annotation, and attributes from the lexicon are associated with each annotation.

The interpreter provides an interface to any external functions to extend the functionality of the basic interpreter. Such functions should be used sparingly and be carefully documented. One example of a legitimate use would be to construct tables of information useful to subsequent coreference resolution.

To date, one interpreter has been developed that closely conforms to the specifications of the reference interpreter, namely the *TextPro* system, implemented by Appelt. The object code, together with a fairly comprehensive English lexicon and gazetteer, and a sample grammar for doing name recognition on Wall Street Journal texts is freely downloadable over the web at the following URL:

http://www.ai.sri.com/~appelt/TextPro/.

3. A DESCRIPTION OF CPSL

A CPSL grammar consists of three parts: a *macro definition part, a declarations part,* and *a rule definition part.* The declaration section allows the user to declare the name of the grammar, since most extraction systems will employ multiple grammars to operate on the input in sequential phases. The grammar name is declared with the statement

```
Phase: <grammar_name>
```

The Input declaration follows the Phase declaration, and tells the interpreter which annotations are relevant for consideration by this phase. For example, a name recognizer will probably operate on Word annotations, while a parser may operate on Word and NamedEntity annotations. If there are multiple annotation types declared in the Input declaration, the first annotation in the list is considered the "default" annotation type. The importance of the default annotation will be explained under the discussion of quoted strings. Any other annotations are invisible to the interpreter, as well as any text that might be annotated exclusively by annotations of ignored types. A typical Input declaration would be:

```
Input: Word, NamedEntity
```

Finally, the language supports an Options declaration, where the user can specify implementation-dependent interpreter options to be used when interpreting the grammar.

3.1 The Rules Section

The rules section is the core of the grammar. It consists of a sequence of rules, each with a name and an optional priority. The general syntax of a rule definition is

```
Rule: <rule_name>
Priority: <integer>

<rule_pattern_part> -->
<rule_action_part>
```

Rules have names primarily for the implementation dependent convenience of error printing and tracing modules. Priority values can be any integer, and indicate to the interpreter whether this rule is to take precedence over other rules. The implementation of priority in the reference interpreter is that the rule matching the most annotations in the input stream is preferred over any rule matching fewer annotations, and if two rules match the same number of annotations, the rule with the highest priority is preferred. If several rules match that have the

same priority, then the rule declared earlier in the file is preferred. Interpreters should adopt this priority semantics by default. If another priority semantics is implemented, the grammar writer can select it in the Options declaration.

The reference interpreter is assumed to maintain a "cursor" pointer marking its position in the chunk of input currently being processed. The interpreter matches each rule pattern part against the sequence of annotations of the declared input type. If no rules match, then the cursor is moved past one input annotation. If one or more rule pattern parts match at the current cursor position, the interpreter selects the "best" match according to the priority criteria discussed above, and executes the rule action part for that rule. Finally, the interpreter moves the cursor past the text matched by the main body part of the rule pattern part. This process is repeated until the cursor is finally moved to the end of the current input chunk.

The Rule Pattern Part.

The pattern part of the rules consists of a prefix pattern, a body pattern, and a postfix pattern. The prefix and postfix patterns are both optional, but the body is mandatory. The syntax is as follows:

```
< prefix_pattern > body_pattern
      < postfix_pattern >
```

When pattern matching begins, the reference interpreter assumes that the initial cursor position is between the prefix pattern and the body pattern. If the annotations to the immediate left of the cursor match the prefix pattern, then the body pattern is matched. If that match is successful, then the postfix pattern is matched. If all three matches are successful, then the pattern is deemed a successful match. Following success and execution of the rule's action part, the cursor is moved to the point in the text after which the body pattern matched, but before the postfix pattern, if any.

Each of the constituents in the above rule is defined the same way. They are grouped (and optionally labeled) sequences of pattern elements. Labels are only useful in the central body pattern, because the annotations matched in the body pattern can be operated on by the action part of the rule. When a new annotation is created from a label in the body pattern, the new annotation receives a span consisting of the first through last characters covered by the spans of the matched annotations.

Groups of pattern elements are enclosed with parentheses, and are optionally followed by a label. There are two types of label expressions, indicated by ":" and "+:" characters, respectively. When used in the pattern part of a rule, the ":" label references the last-matched annotation within its scope. The "+:" annotation, on the other hand, refers to the entire set of annotations matched within its scope. Here is an example of labels used in a pattern:

```
(("douglas"):firstName "appelt")
          +:wholeName
```

In this example, the label "`firstName`" refers to the annotation spanning "`douglas`", and the label "`wholeName`" refers to the set of annotations {"`douglas`" "`appelt`"}.

Pattern elements are constraints on the type, selected attributes and values of the next annotations in the input stream. The basic form of an attribute constraint is

```
Annotation_type.attribute <rel>
            <value>
```

The annotation_type must be one of the types listed on the Input declaration for this grammar. The attribute must be one of the attributes defined for that annotation type. The <rel> element is one of the relations appropriate for the attribute type. Possible relations are equal (==), not equal (!=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=). The <value> element can be a constant of any type known to the interpreter, or it can refer to an annotation matched in the pattern part. The data types supported by the reference interpreter are integer, floating point number, Boolean, string, symbol, a reference to another annotation, or sets of any of those types. The reference interpreter does not treat symbols and strings differently, except that if a symbol contains any non-alphanumeric characters, it must be enclosed in string quotes in order to be parsed correctly by the grammar compiler.

A pattern element consists of constraints in the above form, enclosed in brace characters. For example:

```
{Word.N == true,
Word.number == singular}
```

would match an annotation that has a Boolean "N" attribute with value true, and a character "number" attribute whose value is "`singular`."

The reference interpreter assumes that if an attribute is not present on an annotation, it will be treated as though

it were a Boolean attribute with value *false*. Reasonable type coercion is done when comparing values of different types.

An abbreviation allows an entire pattern element to be replaced by a quoted string. This is shorthand for constraining the lemma attribute of the default input annotation for this grammar to be the specified string. For example, if annotation type Word were declared to be the default input type for the current grammar then the pattern element

"howdy"

would be exactly equivalent to typing

```
{Word.lemma == "howdy"}.
```

The reference interpreter assumes that the value of the lemma attribute is the character sequence that is used to look the word up in the lexicon to obtain its other lexical attributes.

In addition to being sequenced in groups, pattern elements can be combined with one of several regular expression operators. Possible operations include

Alternation: `(arg1 | arg2 | ... | argn)`
Iteration: `(arg1 arg2 ... argn)*` or `(arg1 arg2 ... argn)+`
Optionality: `(arg1 arg2 ... argn)?`

As you would expect, * matches zero or more occurrences of its argument, + matches one or more occurrences, and ? matches zero or one occurrences.

Finally, a pattern element can be a call to an external function. An external function call is simply the name of the function followed by parameters enclosed in square brackets. The function must be defined to return a Boolean value, and it can take any number of arguments, which can be references to annotations and attributes bound by labels defined to the left of where the external function call appears in the pattern. If the function returns *true,* the pattern matching continues, and it fails if the function returns *false.*

The Rule Action Part

The rule action part tells the interpreter what to do when the rule pattern part matches the input at the current position, and consists of a comma-separated list of action specifications. The basic form of an action specification is

```
annotation/attribute
<assignment_operator> <value>
```

The annotation/attribute specification is an instruction to the interpreter to build a new annotation. The annotation/attribute specification has the following syntax:

```
:<label>.<annotation_type>.
        <attribute>
```

The label must be one of the labels defined in the pattern part of the rule. Also, the label must have been bound during the pattern-matching phase. For example, a label in an optional element that was not matched would be unbound, and generate a runtime error. The annotation type of the newly created annotation can be *any* annotation type. The attribute is optional. If present, it means to assign the value on the right hand of the assignment operator to the indicated attribute on the newly created annotation. If the attribute is not present, then the only legal value on the right hand of the assignment operator is "@", which tells the interpreter to create an annotation spanning the specified label, but which has no attributes.

The binding and the type of the label determine the span set of the newly created annotation. If the label was defined with ":", the annotation has a single span, which is the first through the last character of the annotations in the group to which the label is attached. If the label was defined with "+:", the new annotation has a set of spans, where each span in the set is obtained from one of the annotations in the group to which the label is attached.

When the reference interpreter is evaluating an assignment statement, it looks for an annotation of the type specified on the left -hand side that has the exact span specified by the label. If one exists, then that one is used to complete the assignment operation. Otherwise, a new annotation is created. This functionality allows one to assign values to multiple attributes on a single annotation by using a sequence of assignment actions with the same label and annotation type.

CPSL includes two assignment operators: "=" and "+=". The former operator is the basic assignment operator. The latter operator assumes that the left hand operand represents a set, and the right hand element is added to the set by the assignment.

In addition to assignment statements, the action part of a rule can contain simple conditional expressions. The conditional expression can refer to the attributes of

annotations bound during the pattern match. Simple conjunction and disjunction operators (& and |) are provided for multiple conditional clauses, however, the language does not define a full Boolean expression syntax with parentheses and operator precedence. The clauses are simply evaluated left to right. The THEN and ELSE clauses of the conditional consist of a Here is an example of a conditional expression:

```
(IF :1.Word.lemma != false
   THEN
 :rhs.DateTime.lemma = :1.Word.lemma)
```

Action specifications can also be calls to external functions, invoked as before, by the name of the function followed by a list of parameters enclosed in square brackets. External functions can return a value or be defined as void. If the function returns a value, it can appear on the right-hand side of an assignment statement. Otherwise, the external function call appears as an entire action specification.

CPSL does not specify how the interface between the interpreter and the external function should be implemented. Each implementation is free to define its own API.

3.2 The Macro Definition Section

The grammar writer can optionally define macros at the beginning of a grammar definition file. CPSL macros are pure text substitution macros with the following twist: each macro consists of a pattern part and an action part, just like a CPSL rule. The macro is invoked by writing its name followed by an argument list delimited by double angle brackets somewhere in the pattern part of a rule. When the compiler encounters a macro call in the pattern part of the rule, it binds the parameters in the call to the variables in the macro definition prototype. The parameter bindings are substituted for occurrences of the parameters in the macro's pattern part, and the expanded pattern part is then inserted into the rule's pattern part in place of the macro call. Then, parameter substitution is performed on the macro's action part, and the resulting action specification is then added to the beginning of the rule's action part. It is permitted for the pattern part of a macro definition to contain references to other macros, so this macro substitution process is iterated until no more macro substitutions are possible.

Here is an example of a macro definition:

```
Short_and_stupid[X,lbl] ==>
   {Word.X == true, Word.ADJ == false}
       -->
   :lbl.Item.X = true, ;;
```

An invocation of the above macro:

```
Rule: foo
   (Short_and_stupid<<N,myLabel>>)
      :myLabel
   -->
   :myLabel.Item.type = stupid
```

would result in the following rule being compiled:

```
Rule: foo
   ({Word.N == true,
        Word.ADJ == false}):myLabel
      -->
   :myLabel.Item.N = true,
   :myLabel.Item.type = stupid
```

Macros can be used to automatically generate some very complicated rules, and when used judiciously can considerably improve their readability and comprehensibility.

4. A FORMAL DESCRIPTION OF CPSL

The following is a BNF description of the common pattern specification language:

```
<GRAMMAR>  ::= <MACROS> <DECLARATIONS> <RULES>

 ----------------- Declarations ---------------

<DECLARATIONS> ::= <DECL> (<DECLARATIONS>)

<DECL> ::= <DECL_TYPE> : <SYMBOL_LIST>

<DECL_TYPE> ::= Phase | Input | Options
```

```
<SYMBOL_LIST> ::= <SYMBOL> (, <SYMBOL_LIST>)

--------------------- Macros  ------------------

<MACROS> ::= <MACRO> (<MACROS>)

<MACRO> ::= <MACRO_HEADER> ==>

              <PAT_PART> --> <ACT_PART> ;;

<MACRO_HEADER> ::= <SYMBOL> [ <PARAM_LIST> ]

<PAT_PART> ::= any characters except --> and ;;

<ACT_PART> ::= any characters except --> and ;;

<PARAM_LIST> ::= <SYMBOL> ( , <PARAM_LIST> )

<MACRO_INVOCATION> ::= <SYMBOL> << <ARG_LIST> >>

<ARG_LIST> ::= <ARG> (, <ARG_LIST>)

<ARG> ::= any characters except ; and >>

--------------------- Rules  ------------------

<RULES> ::= <RULE> ( <RULES> )

<RULE> ::= <NAME_DECL> (<PRIORITY_DECL>) <BODY>

<NAME_DECL> ::= Rule : <SYMBOL>

<PRIORITY_DECL> ::= Priority : <NUMBER>

<BODY> ::= <CONSTRAINTS> --> <ACTIONS>

<CONSTRAINTS> ::= ( < <CONSTRAINT_GROUP> > )

                  <CONSTRAINT_GROUP>

                ( < <CONSTRAINT_GROUP> > )

<CONSTRAINT_GROUP> ::= <PATTERN_ELEMENTS>

                    ( | CONSTRAINT_GROUP)

<PATTERN_ELEMENTS> ::= <PATTERN_EMEMENT>

                        (<PATTERN_ELEMENTS>)

<PATTERN_ELEMENT> ::= <BASIC_PATTERN_ELEMENT> |

    "(" <CONSTRAINT_GROUP> ")" <KEENE_OP> <BINDING> |

    "(" <CONSTRAINT_GROUP> ")" |

    "("  CONSTRAINT_GROUP ")" <KLEENE_OP> |

    "(" <CONSTRAINT_GROUP> ")" <BINDING>
```

```
<KLEENE_OP> ::= * | + | ?

<BINDING> ::= <INDEX_OP> : <LABEL>

<INDEX_OP> ::= : | +:

<LABEL> ::= <SYMBOL> | <NUMBER>

<BASIC_PATTERN_ELEMENT> ::= { <C_EXPRESSION> } |
                        <QUOTED_STRING> |
                        <SYMBOL> |
                        <FUNCTION_CALL>

<FUNCTION_CALL> ::= <SYMBOL> "[" <FARG_LIST> "]"

<FARG_LIST> ::= nil | <FARG> (",," <FARG_LIST> )

<FARG> ::= <VALUE> | (^) <INDEX_EXPRESSION>

<C_EXPRESSION> ::= <CONSTRAINT>
                    ( ",," <C_EXPRESSION> )

<CONSTRAINT> ::= <ATTRSPEC> <TEST_OP> <VALUE> |
                  <ANNOT_TYPE>

<ATTRSPEC> ::= <ANNOT_TYPE> . <SYMBOL>

<ANNOT_TYPE> ::= <SYMBOL> | <ANY>

TEST_OP ::= == | != | >= | <= | < | >

<VALUE> ::= <NUMBER> | <QUOTED_STRING> | <SYMBOL>
|           true | false

<ACTIONS> ::= <ACTION_EXP) ( , <ACTIONS>)

<ACTION_EXP> ::= <IF_EXP> | <SIMPLE_ACTION>

<IF_EXP> ::= "(" IF <A_C_EXPRESSION>
               THEN <ACTIONS> ")" |
           "(" IF <A_C_EXPRESSION> THEN <ACTIONS>
               ELSE <ACTIONS> ")"

<A_C_EXPRESSION> ::= <A_CONSTRAINT>
                   (<BOOLEAN_OP> <A_C_EXPRESSION>)

<BOOLEAN_OP> ::= & | "|"

<A_CONSTRANT> ::= <INDEX_EXPRESSION>
```

```
                    <TEST_OP> <VALUE>

<SIMPLE_ACTION> ::= <ASSIGNMENT> |

                    <FUNCTION_CALL>

<ASSIGNMENT> ::= <INDEX_EXPRESSION> = @   |

                 <INDEX_EXPRESSION> < ASSIGN_OP >

             (<VALUE> | <INDEX_EXPRESSION> |

                    <FUNCTION_CALL>)

<ASSIGN_OP> ::= = | +=

<INDEX_EXPRESSION> ::= : <INDEX> <FIELD>

<FIELD> ::= . <ANNOT_TYPE> ( . <SYMBOL> )
```

REFERENCES

1. Grishman, Ralph et al. *The TIPSTER Architecture* (this volume)