LAMB
A Lexical Analyzer with Ambiguity Support

Luis Quesada, Fernando Berzal and Francisco J. Cortijo
Department of Computer Science and Artificial Intelligence, CITIC, University of Granada, Granada 18071, Spain
lquesada@decsai.ugr.es, fberzal@decsai.ugr.es, cb@decsai.ugr.es

Keywords: Language processor, lexical analyzer, scanner, lexer, lexical ambiguity, lexical analysis graph.

Abstract: Lexical ambiguities may naturally arise in language specifications. We present Lamb, a lexical analyzer that captures overlapping tokens caused by lexical ambiguities. This novel technique scans through the input string and produces a lexical analysis graph that describes all the possible sequences of tokens that can be found within the string. The lexical graph can then be fed as input to a parser, which will discard any sequence of tokens that does not produce a valid syntactic sentence. In summary, our approach allows a context-sensitive lexical analysis that supports lexically-ambiguous language specifications.

1 INTRODUCTION

A lexical analyzer, also called lexer or scanner, is a piece of software that processes an input string conforming to a language specification and produces a sequence of the tokens or terminal symbols found in it. The obtained sequence of tokens is then usually fed to a parser, also called syntactic analyzer, as the next step of a data translation, compilation or interpretation procedure.

Lexical ambiguities may show up in a language specification. They ocurr when an input string simultaneously corresponds to several token sequences. In order to solve them, traditional lexers allow asigning priorities to tokens (Levine et al., 1992).

However, the language developer may want similar substrings to be recognized as different sequences of tokens depending on their context. This cannot be achieved with the priority approximation.

Statistical lexers (Markov, 1971; Ephraim and Merhav, 2002; McCallum et al., 2000) may perform well in context-sensitive scenarios, but they require intensive training and, as token types are guessed, they do not guarantee that the obtained token sequence will be what the developer intended.

When it comes to programming languages, data specification languages, or limited natural languages scenarios, the syntactic rules are clear as to what should be accepted. The usage of statistical models introduces an unpredictable possibility of error during token recognition that would render scanning and parsing theoretically and pragmatically unfeasible.

Our proposal, Lamb (standing for *Lexical AMBiguity*), performs a lexical analysis that efficiently captures all the possible sequences of tokens and generates a lexical analysis graph that describes them all. The subsequent parsing process would discard any sequence of tokens that did not provide a valid syntactic sentence conforming to the language grammar. This solves the lexical ambiguity problem with formal correctness.

As research in lexers sets the basis for the application of parsers, it inherits their application fields: the processing of programming languages (Aho et al., 2006), the integration of data in data mining applications (Han et al., 2005), and natural language processing (Jurafsky and Martin, 2009).

2 BACKGROUND

Lex and *yacc* (Levine et al., 1992) are traditional lexer generator and parser generator, respectively.

When using a *lex*-generated lexer, tokens get assigned a priority based on the length of the performed matches and, if there is a tie, on the specification order.

The order of efficiency of a *lex*-generated lexer is $O(n)$, being n the input string length.

Statistical models as Hidden Markov Models (Markov, 1971; Ephraim and Merhav, 2002) or Maximum Entropy Markov Models (McCallum et al., 2000) consider the existence of implicit relationships between words, symbols, or characters that are close together in strings. These models need intensive corpus-based training and they produce results with associated implicit probabilities. Even though they can perform well in natural language processing, their training requirement is impractical for programming or data representation languages, especially when the syntactic rules provide all the needed context information to unequivocally identify tokens. Furthermore, the results are prone to interpretation errors that would render the analysis unusable.

The semi-syntactic lexer proposed in (Shyu, 1986) considers context information found in syntactic rules, but is not able to capture syntactic ambiguities for their further consideration.

3 LAMB

In contrast to the aforementioned techniques, Lamb is able to recognize and capture lexical ambiguities.

Our proposed algorithm takes as input the string to be scanned and a list of tokens associated to their corresponding regular expressions. It produces a lexical analysis graph in which each token is connected to its following and preceding tokens in the input sequence.

Our algorithm consists of two steps: the scanning step, which recognizes all the possible tokens in the input string; and the graph generation step, which computes the sets of preceding and following tokens for each token and builds the lexical analysis graph.

3.1 The Scanning Step

The algorithm in Figure 1 takes as input a string and a list of matchers, and produces a list of found tokens sorted by starting position. These tokens may overlap in the input string.

Each matcher consists of a regular expression and its corresponding *match* method, a *priority* value, and a *next* value.

The *match* method performs a match given the input string and a starting position in it, and returns the matched string.

The *priority* value specifies the matcher priority. The value -1 is reserved for ignored patterns, which represent irrelevant text. The value 0 is reserved for tokens that are not affected by priority restrictions. Priority values 1 or higher represent token priorities, being the lower the value, the higher the priority. Whenever a token is found, no lower priority tokens will be looked for within the matched text.

```
for i in 0..input.length()-1:
  prio = -2
  if search[i] == SEARCH:
    anymatch = false
    for each matcher m in matcherlist:
      if (prio == -2 || prio >= m.prio ||
          m.prio == 0) && (prio != -1 &&
          next[j] < i):
        match = m.match(input,i)
        if match != null:
          anymatch = true
          prio = m.prio
          end = i+match.length()-1
          if search[end+1] == SKIP:
            search[end+1] = SEARCH
          if m.prio == -1: //ignored pattern
            for k in t.start..t.end:
              search[k] = NEVER
          else: //not ignored pattern
            t = new token(id=id,text=match,
                 type=m.type,start=i,end=end)
            tokenlist.add(t)
            id++
    if !anymatch:
      if search[i+1] == SKIP
        search[i+1] = SEARCH
```

Figure 1: Pseudocode of the scanning step in our lexical analysis algorithm.

The *next* value specifies the position before the next string index a match will be tried to be performed at. It defaults to -1.

The *search* array determines if an input string index has to be scanned, skipped, or never scanned (i.e. if an ignore pattern that contains it was found). It defaults to *SCAN* for the position 0 and *SKIP* for the rest of them.

The *prio* variable represents the last priority that has been matched in the current input position. Its value is -2 if no match was performed, -1 if an ignored element match was performed, and a higher value if any token of that priority has been found.

This step has a theoretical order of efficiency of $O(n^2 \cdot l)$, being n the input string length and l the number of matchers in the lexer.

3.2 The Graph Generation Step

The algorithm in Figure 2 goes through the identified token list in reverse order and efficiently computes the sets of preceding and following tokens for every token.

```
for i in tokenlist.size()-1..0:
  t = tokenlist[i]
  state = 0
  minend = input.length()+1
  for j in i+1..tokenlist.size()-1:
    tc = tokenlist[j]
    if state == 0 && tc.start>t.end:
      state = 1
    if state == 1 && tc.start>t.end:
      if tc.start>minend:
        break
      else:
        minend = min(minend,tc.end)
        t.addfollowing(tc)
        tc.addpreceding(t)
```

Figure 2: Pseudocode of the graph generation step in our lexical analysis algorithm.

The sets of preceding and following tokens of the token x are defined in Equation 1, being a, b, c tokens and x_{start} and x_{end} the starting and ending positions of the token x in the input string.

$$\begin{gathered} b \in FOLLOWING(a), a \in PRECEDING(b) \text{ iif} \\ a_{end} < b_{start} \,\&\, \nexists c, c_{start} > a_{end}, c_{end} < b_{start} \end{gathered} \tag{1}$$

After these sets have been computed for every token, any token whose preceding set is empty is added to the start token set of the lexical analysis graph.

This step theoretical order of efficiency of $O(tk)$, being t the number of tokens found and k the maximum number of tokens that follow a token in the graph. As $t \leq n \cdot l$, the theoretical order of efficiency of this step is $O(nlk)$.

Both scanning and graph generation steps together have an order of efficiency of $O(nl(k+n))$.

4 COMPARISON

We have implemented a simple proof of concept parser that allows a lexical analysis guided by a syntactic rule set. Its pseudocode is shown in Figure 3.

The *parse* method returns all the possible reductions using a rule given a starting symbol.

In the lexically-ambiguous language specification that describes the tokens listed in Figure 4, any sequence of digits separated with points could be considered either *Real* tokens or *Integer Point Integer* token sequences.

The syntactic rules shown in Figure 5 illustrate a scenario of lexical ambiguity sensitivity, as the consideration of the aforementioned tokens depends on the context. The expected parse of the input string "&5.2& /25.20/" is shown in Figure 6.

```
symbollist = tokenlist
do:
  flag = false
  for each rule r in rules:
    for each symbol s in symbollist:
      matches = r.parse(s)
      for each match m in matches:
        if !symbollist.contains(m):
          symbollist.add(m)
          if m is start symbol:
            validparses.add(m)
          flag = true
while flag = true
```

Figure 3: Pseudocode of the proof of concept parser supporting ambiguities.

```
(-|\+)?[0-9]+              Integer
(-|\+)?[0-9]+\.[0-9]+      Real
\.                         Point
\/                         Slash
\&                         Ampersand
```

Figure 4: Regular expressions and token names in the specification of our ambiguous language.

```
E ::= A B
A ::= Ampersand Real Ampersand
B ::= Slash Integer Point Integer Slash
```

Figure 5: Context-sensitive syntactic rules that solve lexical ambiguities.

When using a traditional lexer, the developer can assign the *Integer* token a greater priority than the *Real* token or the opposite way. The respective interpretations are shown in Figures 7 and 8.

On the other hand, Lamb is able to capture all the possible token sequences in the form of a lexical analysis graph, as shown in Figure 9. The parsing of this graph would produce the only possible valid sentence, which, in turn, is based on the only valid lexical analysis possible. Both of them are shown in Figure 10.

5 CONCLUSIONS AND FUTURE WORK

We have presented Lamb, a lexer that supports lexical ambiguities. It performs a lexical analysis that efficiently captures all the possible sequences of tokens for lexically-ambiguous languages and it generates a lexical analysis graph that describes them all. Lamb supports assigning priorities to tokens as traditional

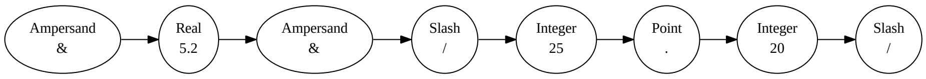

Figure 6: Intended lexical analysis.

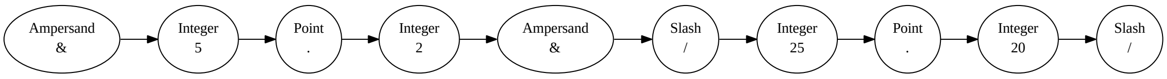

Figure 7: Lexical analysis, as produced by a traditional lexer, when the *Integer* token has a greater priority than the *Real* token.

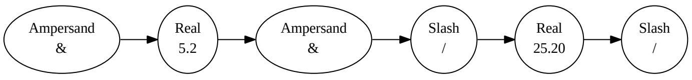

Figure 8: Lexical analysis, as produced by a traditional lexer, when the *Real* token has a greater priority than the *Integer* token.

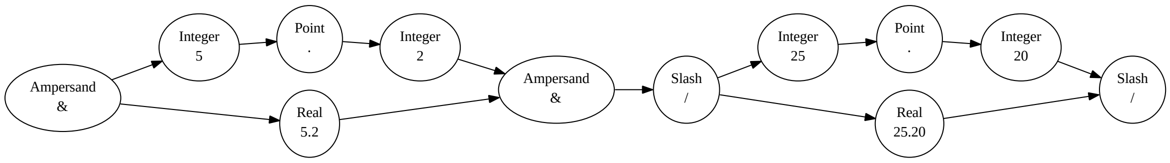

Figure 9: Lexical analysis, as produced by Lamb, when *Real* and *Integer* tokens share priority value.

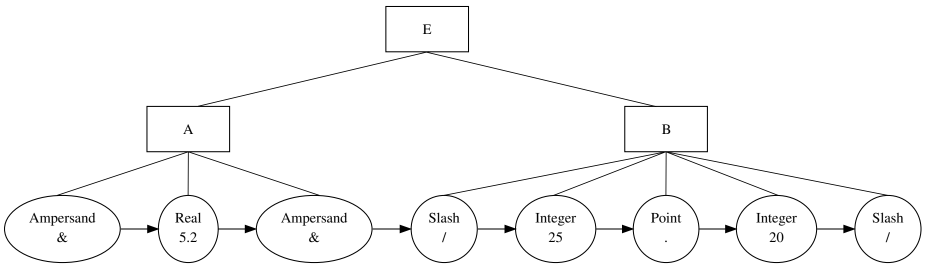

Figure 10: Correct syntactic analysis produced by applying an ambiguity-supporting parsing technique to the lexical analysis graph produced by Lamb and shown in Figure 9.

techniques do but, in contrast to them, it does not enforce these priorities to be set and it allows for priority values to be shared. Tokens with shared priorities are considered valid alternatives instead of mutually-exclusive options.

The lexical graph can be further processed in order to perform a context-sensitive lexical analysis guided by syntactic rules.

We plan to extend existing parsing techniques for them to accept lexical analysis graphs. We will also apply Lamb to modular languages, where token definitions may conflict and cause ambiguities.

Acknowledgements: Work partially supported by research project TIN2009-08296.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition.

Ephraim, Y. and Merhav, N. (2002). Hidden markov processes. *IEEE Transactions on Information Theory*, 48:1518–1569.

Han, J., Kamber, M., and Pei, J. (2005). *Data Mining: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2nd edition.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing*. Prentice Hall, 2nd edition.

Levine, J. R., Mason, T., and Brown, D. (1992). *lex&yacc*. O'Reilly, 2nd edition.

Markov, A. A. (1971). *Extension of the limit theorems of probability theory to a sum of variables connected in a chain*. R. Howard, Dynamic Probabilistic Systems volume 1, Appendix B. John Wiley and Sons.

McCallum, A., Freitag, D., and Pereira, F. (2000). Maximum entropy markov models for information extraction and segmentation. In *Proc. of the 17th International Conference on Machine Learning*, pages 591–598.

Shyu, Y.-H. (1986). From semi-syntactic lexical analyzer to a new compiler model. *ACM SIGPLAN Notices*, 21:149–157.