Fencing the Lamb: A Chart Parser for ModelCC

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

Abstract. Traditional grammar-driven language specification techniques constrain language designers to specific kinds of grammars. In contrast, model-based language specification techniques decouple language design from language processing. They allow the occurrence of ambiguities and the declarative specification of constraints for solving them. As a result, these techniques require general parser generators, which should be able to parse context-free grammars, handle ambiguities, and enforce constraints to disambiguate them as desired by the language designer. In this paper, we describe Fence, a bottom-up chart parser with lexical and syntactic ambiguity support. Fence accepts lexical analysis graphs outputted by the Lamb (Lexical AMBiguity) lexer and efficiently resolves ambiguities by means of the declarative specification of constraints. Both Lamb and Fence are part of the ModelCC model-based parser generator.

Keywords: Chart parser, context-free grammars, ambiguities, constraints.

1 Introduction

Traditional language specification techniques [2] require the developer to provide a textual specification of the language grammar.

In contrast, model-based language specification techniques [15,24,25,21] allow the specification of languages by means of data models annotated with constraints.

Model-based language specification has direct applications in the following fields: programming tools [1], domain-specific languages [9,10,18], model-driven software development [26], data integration [27], text mining [29,4], natural language processing [11], and the corpus-based induction of models [14].

Due to the nature of the aforementioned application fields, the specification of separate language elements may cause lexical and syntactic ambiguities. Lexical ambiguities occur when an input string simultaneously corresponds to several token sequences [19], which may also overlap. Syntactic ambiguities occur when a token sequence can be parsed in several ways.

The formal grammars of languages specified using model-based techniques may contain epsilon productions (such as $E ::= \epsilon$), infinitely recursive production sets (such as $A ::= c$, $A ::= B$, and $B ::= A$), and associativity, precedence, and custom constraints. Therefore, a parser that supports such specification is needed.

Our proposed algorithm, Fence [23], is a bottom-up chart parser that accepts a lexical analysis graph as input, performs an efficient syntactic analysis taking constraints into account, and produces a parse graph that represents all the possible parse trees. The parsing process discards any sequence of tokens that does not provide a valid syntactic sentence conforming to the language specification, which consists of a production set and a set of constraints. Fence implicitly performs a context-sensitive lexical analysis, as the parsing process determines which token sequences end up in the parse graph. Fence supports every possible construction in a context-free language with constraints, including epsilon productions and infinitely recursive production sets.

The combined use of the Lamb lexical analyzer [22] and Fence [23] allows the generation of processors for languages with ambiguities and constraints, and it renders model-based language specification techniques feasible. Indeed, ModelCC [24,25,21] is a model-based language specification tool that relies on Lamb and Fence to generate language processors.

Section 2 explains model-based language specification techniques and lexical analysis algorithms with ambiguity support. Section 3 introduces Fence, our parser with ambiguity and constraint support. Section 4 presents our conclusions and future work.

2 Background

Language processing tools traditionally divide the analysis into two separate phases; namely, scanning (or lexical analysis), which is performed by lexers, and parsing (or syntax analysis), which is performed by parsers. However, language processing tools based on scannerless parsers also exist.

Subsection 2.1 introduces model-based language specification techniques and the model-based parser generator ModelCC. Subsection 2.2 analyzes existing scanning algorithms with ambiguity support. Subsection 2.3 describes existing parsing algorithms.

2.1 Model-Based Language Specification

In its most general sense, a model is anything used in any way to represent something else. In such sense, a grammar is a model of the language it defines. In Software Engineering, data models are also common. Data models explicitly determine the structure of data. Roughly speaking, they describe the elements they represent and the relationships existing among them. From a formal point of view, it should be noted that data models and grammar-based language specifications are not equivalent, even though both of them can be used to represent data structures. A data model can express relationships a grammar-based language specification cannot. A data model does not need to comply with the constraints a grammar-based language specification has to comply with. Typically, describing a data model is generally easier than describing the corresponding grammar-based language specification.

In practice, when we want to build a complex data structure from the contents of a file, the implementation of the mandatory language processor needed to parse the file requires the software engineer to build a grammar-based language specification for the data as represented in the file and also to implement the conversion from the parse tree returned by the parser to the desired data structure, which is an instance of the data model that describes the data in the file.

Whenever the language specification has to be modified, the language designer has to manually propagate changes throughout the entire language processor tool chain, from the specification of the grammar defining the formal language (and its adaptation to specific parsing tools) to the corresponding data model. These updates are time-consuming, tedious, and error-prone. By making such changes labor-intensive, the traditional language processing approach hampers the maintainability and evolution of the language used to represent the data [13].

Moreover, it is not uncommon for different applications to use the same language. For example, the compiler, different code generators, and other tools within an IDE, such as the editor or the debugger, typically need to grapple with the full syntax of a programming language. Unfortunately, their maintenance typically requires keeping several copies of the same language specification in sync.

The idea behind model-based language specification is that, starting from a single abstract syntax model (ASM) that represents the core concepts in a language, language designers can develop one or several concrete syntax models (CSMs). These CSMs can suit the specific needs of the desired textual or graphical representation. The ASM-CSM mapping can be performed, for instance, by annotating the abstract syntax model with the constraints needed to transform the elements in the abstract syntax into their concrete representation.

This way, the ASM representing the language can be modified as needed without having to worry about the language processor and the peculiarities of the chosen parsing technique, since the corresponding language processor will be automatically updated.

Finally, as the ASM is not bound to a particular parsing technique, evaluating alternative and/or complementary parsing techniques is possible without having to propagate their constraints into the language model. Therefore, by using an annotated ASM, model-based language specification completely decouples language specification from language processing, which can be performed using whichever parsing techniques are suitable for the formal language implicitly defined by the abstract model and its concrete mapping.

A diagram summarizing the traditional language design process is shown in Figure 1, whereas the corresponding diagram for the model-based approach is shown in Figure 2. It should be noted that ASMs represent non-tree structures whenever language elements can refer to other language elements, hence the use of the ‘abstract syntax graph’ term.

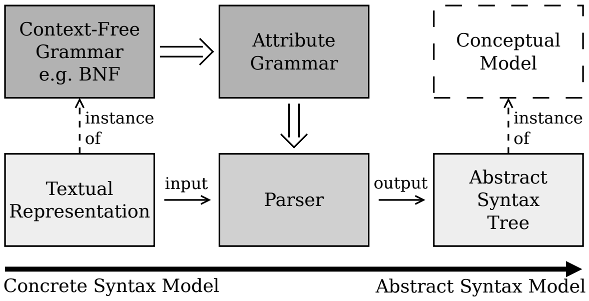

Fig. 1: Traditional language processing.

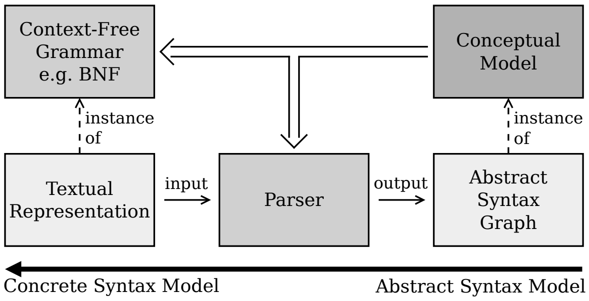

Fig. 2: Model-based language processing.

The ASM is built on top of basic language elements, which can be viewed as the tokens in the model-driven specification of a language. ModelCC provides the necessary mechanisms to combine those basic elements into more complex language constructs, which correspond to the use of concatenation, selection, and repetition in the syntax-driven specification of languages.

When the ASM represents a tree-like structure, a model-based parser generator is equivalent to a traditional grammar-based parser generator in terms of expression power. When the ASM represents non-tree structures, reference resolution techniques can be employed to make model-based parser generators more powerful than grammar-based ones, as we will see in the next Section.

In ModelCC, the constraints imposed over ASMs to define a particular ASM-CSM mapping can be declared as metadata annotations on the model itself. Now supported by all the major programming platforms, metadata annotations are often used in reflective programming and code generation. Table 1 summarizes the set of constraints supported by ModelCC for establishing ASM-CSM mappings between ASMs and their concrete representation in textual CSMs.

It should be noted that model-based language specification techniques allow lexical and syntactic ambiguities: each language element is defined as a separate and independent entity, even when their pattern specification or syntactic specification are in conflict. Therefore, model-based language specification techniques

Constraints on...	Annotation	Function
Patterns	@Pattern	Pattern definition of basic language elements.
	@Value	Field where the matched input will be stored.
Delimiters	@Prefix	Element prefix(es).
	@Suffix	Element suffix(es).
	@Separator	Element separator(s).
Cardinality	@Optional	Optional elements.
	@Minimum	Minimum element multiplicity.
	@Maximum	Maximum element multiplicity.
Evaluation order	@Associativity	Element associativity (e.g. left-to-right).
	@Composition	Eager or lazy policy for nested composites.
	@Priority	Element precedence.
References	@ID	Identifier of a language element.
	@Reference	Reference to a language element.

Table 1: Summary of the basic metadata annotations supported by ModelCC.

require a lexical analysis algorithm with ambiguity support and a corresponding parsing algorithm with lexical and syntactic ambiguity support.

2.2 Lexical Analysis Algorithms with Ambiguity Support

Given a language specification describing the tokens listed in Figure 3, the string "&5.2& /25.20/" can correspond to the four different lexical analysis alternatives shown in Figure 4, depending on whether the sequences of digits separated by points are considered real numbers or integer numbers separated by points.

The productions shown in Figure 5 illustrate a scenario of lexical ambiguity sensitivity. Sequences of digits separated by points should be considered either *Real* tokens or *Integer Point Integer* token sequences depending on the surrounding tokens, which may be either *Ampersand* tokens or *Slash* tokens. The desired result of analyzing the input string "&5.2& /25.20/" is shown in Figure 6.

The further application of a parser supporting lexical ambiguities would produce the only possible valid sentence, which, in turn, would be based on the only valid lexical analysis for our example. The intended results are shown in Figure 8.

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

Fig. 3: Specification of token types as regular expressions for a lexically-ambiguous language.

- `Ampersand Integer Point Integer Ampersand Slash Integer Point Integer Slash`
- `Ampersand Integer Point Integer Ampersand Slash Real Slash`
- `Ampersand Real Ampersand Slash Integer Point Integer Slash`
- `Ampersand Real Ampersand Slash Real Slash`

Fig. 4: Different possible token sequences in the input string "&5.2& /25.20/" due to the lexically-ambiguous language specification shown in Figure 3.

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

Fig. 5: Context-sensitive productions that resolve the ambiguities in Figure 4.

The Lamb lexical analyzer [22] captures all possible sequences of tokens within a given input string and it generates a lexical analysis graph that describes them all, as shown in Figure 7. In these graphs, each token is linked to its preceding and following tokens. There may also be several starting tokens. Each path in these graphs describes a possible sequence of tokens that can be found within the input string.

To the best of our knowledge, the only way to process lexical analysis graphs consists of extracting the different paths from the graph and parse each of them. This process is inefficient, as partial parsing trees that are shared among different token sequences have to be created several times.

2.3 Syntactic Analysis Algorithms

Traditional efficient parsers for restricted context-free grammars, such as the LL [20], LR [16], LALR [5,7], and SLR [6] parsers, do not consider ambiguities in syntactic analysis, so they cannot be used to parse ambiguous languages. The efficiency of these parsers is $O(n)$, being n the token sequence length.

Generalized LR (GLR) parsers [17] parse in linear to cubic time, depending on how closely the grammar conforms to the underlying LR strategy. The time required to run the algorithm is proportional to the degree of nondeterminism in the grammar. The Universal parser [28] is a GLR parser used for natural language processing. However, it fails for grammars with epsilon productions and infinitely recursive production sets.

Existing chart parsers for unrestricted context-free grammar parsing, as the CYK parser [30,12] and the Earley parser [8], can consider syntactic ambiguities but not lexical ambiguities. The efficiency of these general context-free grammar parsers is $O(n^3)$, being n the token sequence length.

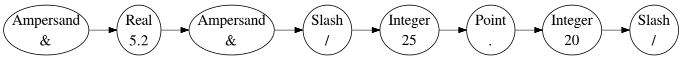

Fig. 6: Desired lexical analysis of the lexically ambiguous "&5.2& /25.20/" input string.

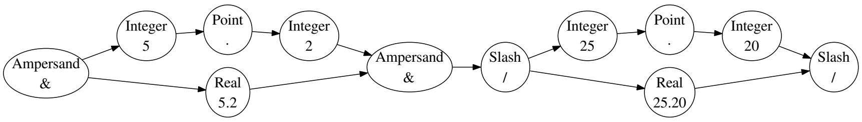

Fig. 7: Lexical analysis graph, as produced by the Lamb lexer.

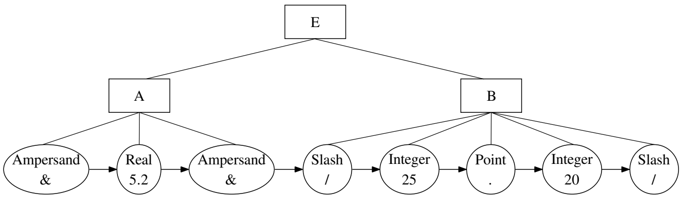

Fig. 8: Syntactic analysis graph, as produced by applying a parser that supports lexical ambiguities to the lexical analysis graph shown in Figure 7. Squares represent nonterminal symbols found during the parsing process.

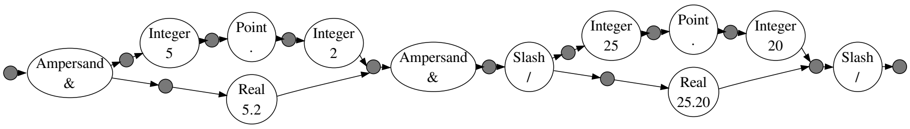

Fig. 9: Extended lexical analysis graph corresponding to the lexical analysis graph shown in Figure 7. Gray nodes represent cores.

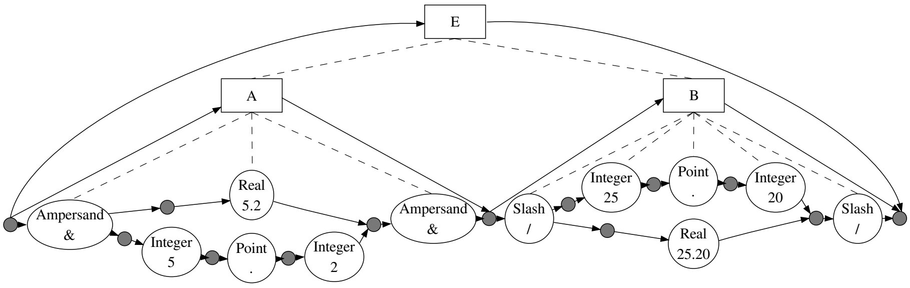

Fig. 10: Parse graph corresponding to the extended lexical analysis graph shown in Figure 9. Squares represent nonterminal symbols found during the parsing process. Dotted lines represent the explicit parse graph node.

3 Fence

In this section, we introduce Fence, an efficient bottom-up chart parser that produces a parse graph that contains as many root nodes as different parse trees exist for a given ambiguous input string.

In contrast to the parsing techniques mentioned in the previous section, Fence is able to process lexical analysis graphs and, therefore, it efficiently considers lexical ambiguities.

Fence also considers syntactic ambiguities, allows the specification of constraints, and supports every possible context-free language construction, particularly epsilon productions and infinitely recursive production sets.

The Fence parsing algorithm consists of three consecutive phases: the extended lexical analysis graph construction phase, the chart parsing phase, and the constraint enforcement phase.

Subsection 3.1 introduces the terminology used in this section. Subsection 3.2 describes the extended lexical analysis graph construction phase. Subsection 3.3 describes the chart parsing phase. Subsection 3.4 describes the constraint enforcement phase.

3.1 Terminology

A context-free grammar is formally defined [3] as the tuple (N, Σ, P, S), where:

- N is the finite set of nonterminal symbols of the language, sometimes called syntactic variables, none of which appear in the language strings.
- Σ is the finite set of terminal symbols of the language, also called tokens, which constitute the language alphabet (i.e. they appear in the language strings). Therefore, Σ is disjoint from N.
- P is a finite set of productions, each one of the form $N \rightarrow (\Sigma \cup N)^*$, where $*$ is the Kleene star operator, $\cup$ denotes set union, the part before the arrow is called the left-hand side (LHS) of the production, and the part after the arrow is called the right-hand side (RHS) of the production.
- S is a distinguished nonterminal symbol, $S \in N$: the grammar start symbol.

A dotted production is of the form $N \rightarrow (\Sigma \cup N)^*.(\Sigma \cup N)^*$, where the dot indicates that the RHS symbols before the dot have already been matched with a substring of the input string.

A handle is a tuple $(dottedproduction, [start, end])$, where $start$ and end identify the substring of the input string that matched the dotted production RHS symbols before the dot. Each handle can be used during the parsing process to match a rule RHS symbol with a node representing either a token or a nonterminal symbol (namely, SHIFT actions in LR-like parsers) or perform a reduction (namely, REDUCE actions in LR-like parsers).

A core is a set of handles.

3.2 Extended Lexical Analysis Graph Construction Phase

In order to efficiently perform the parsing process, Fence converts the input lexical analysis graph (LA graph) into an extended lexical analysis graph (ELA graph) that stores information about partially applied productions (namely, handles) in data structures (namely, cores).

In an ELA graph, tokens are not linked to their preceding and following tokens, but to their preceding and following cores. Cores are, in turn, linked to their preceding and following token sets. For example, the ELA graph corresponding to the LA graph in Figure 7 is shown in Figure 9.

The conversion from the LA graph to the ELA graph is performed by completing the LA graph with cores. A *starting* core is linked to the tokens with an empty preceding token set. A *last* core is linked from the tokens with an empty following token set. Finally, for each one of the other tokens in the LA graph, a preceding core is linked to it. Links between tokens in the LA graph are converted into links from tokens to the cores preceding each token of their following token set in the ELA graph.

3.3 Chart Parsing Phase

The Fence chart parsing phase processes the ELA graph and generates an implicit parse graph (I-graph). Nodes in the I-graph are described as $(start, end, symbol)$ tuples, where *start* and *end* identify the substring of the input string, and *symbol* identifies the production LHS. It should be noted that ambiguities, both lexical and syntactic, are implicit in the I-graph nodes, as they contain no information about their contents. The I-graph contains a set of starting nodes, each of which may represent several parse tree roots. The parsing itself is performed by progressively applying productions and storing handles in cores.

The grammar productions with an empty RHS (i.e. epsilon productions) are removed from the grammar and their LHS symbol is stored in the *epsilonSymbols* set. This set allows these parse symbols being skipped when found in a production, as if a reduction using the epsilon production were applied.

The agenda is a stack of $(handle, node)$ in which the node can match the symbol after the dot in the dotted rule of the handle. It is initially empty.

The *alreadyGenerated* handle set contains all the agenda entries ever generated and inhibits the generation of duplicate entries.

The parser is initialized by generating a handle for each production and adding them to every core, as shown in Figure 12.

The *addHandle* procedure in Figure 11 is responsible for adding a handle to a core. It also adds the corresponding agenda entries for that handle with the nodes that follow the core and match the symbol after the dot in the dotted production of the handle. It should be noted that the *addHandle* procedure considers epsilon productions: if a production RHS symbol is in the *epsilonSymbols* set, both the possibilities of it being reduced or not by that production are considered; that is, a new handle that skips that element is added to the same core. It should also

```
procedure addHandle(Production p, int matched, ImplicitNode first,
                    ImplicitNode n, Stack<[Handle,ImplicitNode]> agenda):
  offset = 0
  do:
    next = matched+offset
    nextSymbol = p.right[next].symbol
    h = new Handle(p,next,first,first.startIndex)
    if !n.core.contains(h):
      n.core.add(h)
    if n.symbol == nextSymbol:
      if !alreadyGenerated.contains([h,n]):
        agenda.push([h,n])
        alreadyGenerated.add(]h,n])
    offset++
  while epsilonSymbols.contains(nextSymbol) && next<r.right.size
```

Fig. 11: Pseudocode of the ancillary *addHandle* procedure.

```
agenda = {}
for each Production p in productionSet:
  for each ImplicitNode n in nodeSet:
    addHandle(p,0,n,n,agenda)
```

Fig. 12: Pseudocode of the chart parser initialization.

be noted that element are skipped iteratively, as many consecutive RHS symbols of a production could be in the *epsilonSymbols* set.

The parsing process consists in iteratively extracting entries consisting of handles and nodes from the agenda and matching the next symbol of the RHS of the handle production with the node. The handles whose productions are successfully matched are added to the cores following the node and the agenda is updated with the entries that contain any of the newly generated handles. In case all the symbols of a production RHS match a sequence of nodes, a new node is generated by reducing them. The new *node* start index is obtained from the handle, its *end* position is obtained from the last node matched, and its *symbol* is the LHS symbol of the production. When a newly generated node only has the *starting* core in its preceding core set and the *final* core in its following core set, and its *symbol* corresponds to the initial symbol of the grammar, it is added to the parse graph starting node set, which means that that node represents a valid parse. The pseudocode for this process is shown in Figure 13.

The result of the chart parsing phase is an I-graph, which the constraint enforcement phase accepts as input.

The Fence chart parsing phase order of efficiency is theoretically equivalent to existing Earley chart parsers. That is, $O(n^3)$ in the general case, $O(n^2)$ for

```
while !agenda.empty:
  [h,n] = agenda.pop()
  if h.dotposition == h.production.right.size-1:
    // Production matched all its elements. i.e. Reduction
    nn = new ImplicitNode(h.startIndex,n.endIndex,p.left.symbol)
    h.first.core.following.add(nn)
    nn.preceding.add(h.first.core)
    for each Core c in n.following:
      c.preceding.add(nn)
      nn.following.add(c)
    for each Handle hn in h.first.core.waitingFor(nn.symbol):
      hadd = new Handle(hn.production,hn.next,hn.first,hn.startIndex)
      agenda.push([hadd,n])
  else:
    // i.e. Shift
    for each Core c in n.following:
      for each ImplicitNode nnext in c.following:
        addHandle(h.production,h.next+1,h.first,h.startIndex,agenda)
```

Fig. 13: Pseudocode of the Fence parsing phase.

unambiguous grammars, and $O(n)$ for almost all LR(k) grammars, being n the length of the input string.

3.4 Constraint Enforcement Phase

The Fence constraint enforcement phase processes the I-graph and generates an explicit parse graph (E-graph, or just parse graph) by enforcing the constraints defined for the language. Nodes in the E-graph that represent tokens are still defined as $(start, end, symbol)$ tuples. Nodes in the E-graph that represent nonterminal symbols reference the list of nodes that matched the production used to generate those nodes. It should be noted that ambiguities, both lexical and syntactic, are explicit in the E-graph, as it represents several parse trees corresponding to all the possible interpretations of the input string. The E-graph contains a set of starting nodes, each of which represents a parse tree root. Constraint enforcement is performed by converting each implicit node into every possible explicit node sequence that can be derived from the implicit node and satisfies the specified constraints; that is, by expanding the each implicit node.

Only the nodes that conform valid parse trees are needed in the parse graph. In order to generate only these nodes, each one of the implicit nodes in the starting node set of the I-graph is recursively expanded using memoization. Each possible resulting explicit node is the root of a parse tree in the E-graph.

Algorithm Description The expansion of an implicit node is performed by finding every possible reduction of a sequence of explicit nodes that generates

```
procedure expand(ImplicitNode n, Set<ImplicitNode> history,
                 Map<ImplicitNode,Set<Node>> alreadyExpanded)
                 returns Set<Symbol>:
  if alreadyExpanded.contains(n): // memoization
    return alreadyExpanded.get(n)
  else:
    // the history set avoids infinite loop in recursive production sets
    if !history.contains(n):
      history.add(n);
      // try to apply every production
      for each Production p with LHS symbol == n.symbol:
        for every ImplicitNode pn with startIndex == n.startIndex:
          if pn != n && pn.endIndex<=n.endIndex:
            if p.mayMatch(pn.symbol):
              // apply production p to each expanded symbol of pn
              pn.expandeds = expand(pn,history,alreadyExpanded)
              for each Node nn in pn.expandeds:
                out += apply(p,nn,0,{},alreadyExpanded,history)
    alreadyExpanded.put(n,out)
    return out
```

Fig. 14: Pseudocode of the *expand* procedure that obtains every possible derivation of a given node in the parse graph.

that node. Each one of these reductions produces an explicit node. Whenever an implicit node is found and needed in order to make the reductions progress, it is expanded recursively. It should be noted that this procedure is different from parsing itself in that the actual bounds of the reductions for every node are known.

The *expand* procedure in Figure 14 expands an implicit node by applying every possible production that could generate it and produces a set of explicit nodes. The use of the *history* set inhibits entering an infinite loop when processing infinitely recursive production sets, as it avoids the expansion of a node as an indirect requirement of expanding the same node.

The *apply* procedure in Figure 15 applies a production by matching the RHS symbol given by the $matched + 1$ index of it with the n node, expanding the nodes that follows it, and recursively applying the next RHS symbols of the production.

The *checkConstraints* procedure is the responsible for the enforcement of the constraints specified by the developer.

Supported Constraints Fence supports associativity constraints, selection precedence constraints, composition precedence constraints, and custom-designed constraints.

The fact that the constraint check is performed during the graph expansion improves the parser performance, as the sooner constraints are applied, the more

```
procedure apply(Production p, Node n, int matched, List<Node> content,
                Map<ImplicitNode,Set<Node>> alreadyExpanded,
                Set<ImplicitNode> history) returns Set<Node>:
  if matched == p.right.size:
    n = new Node(p.symbol,p,content)
    if checkConstraints(n):
      return {n}
  else:
    offset = 0
    next = matched+offset
    do:
      if p.right[next].symbol == n.symbol:
        for each ImplicitNode pn in n.followingNodes():
          if pn is the next symbol to match in the production:
            // keep applying production to each expanded symbol of pn
            expandeds = expand(pn,history,alreadyExpanded)
            for each Node nn in expandeds:
              out += apply(p,nn,next+1,content+n,alreadyExpanded,history)
      offset++
      next = matched+offset
    while epsilonSymbols.contains(nextSymbol) && next<r.right.size &&
          p.right[next].symbol == n.symbol
    return out
```

Fig. 15: Pseudocode of the ancillary *apply* procedure that applies a production.

interpretations are discarded. For example, in the case of a binary expression with left-to-right associative operators, the string "2+5+3+5+6+2+1+5+6+3" can be expanded in 10! possible ways when not considering the associativity constraint, and in just 1 possible way when considering it.

- **Associativity constraints** allow the specification of the associative property for binary operators. The application of a production is inhibited when any of the nodes that matches its RHS symbols has an associativity constraint and is followed (for left-to-right associativity constraints), preceded (for right-to-left associativity constraints), or either followed or preceded (for non-associative associativity constraints) by a node that was derived using the same production.
- **Selection precedence constraints** allow the resolution of syntactic ambiguities caused by different explicit nodes (i.e. interpretations) resulting from a single implicit node. For example, a *Statement* can be either an *OutputStatement* or a *FunctionCall*. Both *OutputStatement* and *FunctionCall* can match the input string "output(var);", therefore *OutputStatement* can be set to precede *FunctionCall*, which will inhibit that string from being considered a function call. The application of a production is inhibited when it is preceded by a different production and both of them match the same node sequence.

- **Composition precedence constraints** allow the resolution of syntactic ambiguities when a node derived using a production cannot be derived using another production. For example, one of the productions *ConditionalStatement ::= "if" Expression Sentence* and *ConditionalStatement ::= "if" Expression Sentence "else" Sentence* can be set to precede the other one in order to resolve the ambiguity in "if expr1 if expr2 sent1 else sent2", in which "else sent2" could be assigned to either the inside or outside conditional sentence. The application of a production is inhibited when it precedes any of the productions used to derive the nodes that matched its RHS symbols.
- **Custom-designed constraints** allow the specification of any other constraints (e.g. semantic constraints). In order to enforce custom-designed constraints, an evaluator can be assigned to any production. Whenever a node is generated, the evaluator of the production used to derive it gets executed and determines whether the node satisfies the constraint or not. In the later case, its generation is inhibited. Custom-designed constraints provide a very extensible framework which allows developers to design complex syntactic or semantic constraints (e.g. probabilistic constraints, corpus-based constraints) that effectively limit the possible interpretations of an input string and, as a side effect, improve the performance of the parser, as pruned partial interpretations are discarded as soon as they do not fulfill the constraints.

The result of the constraint enforcement step is an E-graph or parse graph, such as the one shown in Figure 10.

The Fence constraint enforcement phase improves the performance of traditional techniques phases in practice, as all constraints are applied at the earliest possible time, thus discarding possibilities that would otherwise be processed later.

4 Conclusions and Future Work

We have presented Fence, an efficient bottom-up chart parsing algorithm with lexical and syntactic ambiguity support. Its constraint-based ambiguity resolution mechanism enables the use of model-based language specification in practice. In fact, the ModelCC model-based language specification tool [24,25,21] generates Fence parsers.

Fence accepts a lexical analysis graph as input, performs syntactic analysis conforming to a formal context-free grammar specification and a set of constraints, and produces as output a compact representation of the set of parse trees accepted by the language.

Fence applies constraints while expanding the parse graph. Thus, it improves the performance of traditional techniques in practice, as the sooner constraints are applied, the less processing time and memory the parser will require.

In the future, we plan to apply ModelCC, Lamb, and Fence to natural language processing.

Acknowledgments

Work partially supported by research project TIN2009-08296 and research project TIN2012-36951.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.
2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing & Volume II: Compiling*. Prentice Hall, Englewood Cliffs, N.J., 1972.
3. N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–123, 1956.
4. V. Crescenzi and G. Mecca. Automatic information extraction from large websites. *Journal of the ACM*, 51:731–779, 2004.
5. F. L. DeRemer. Practical translators for LR(k) languages. Technical report, Cambridge, MA, USA, 1969.
6. F. L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.
7. F. L. DeRemer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, 1982.
8. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 26:57–61, 1983.
9. M. Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.
10. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, vol. 28, no. 4es, art. 196, 1996.
11. D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, 2nd edition, 2009.
12. T. Kasami and K. Torii. A syntax-analysis procedure for unambiguous context-free grammars. *Journal of the ACM*, 16:423–431, 1969.
13. L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*, pages 918–932, 2010.
14. D. Klein. Christopher d. manning. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics (ACL '04)*, pages 478–485, 2004.
15. A. Kleppe. Towards the generation of a text-based IDE from a language metamodel. In *Lecture Notes in Computer Science*, volume 4530, pages 114–129, 2007.
16. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.
17. B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer Berlin / Heidelberg, 1974.
18. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, 2005.

19. J. R. Nawrocki. Conflict detection and resolution in a lexical analyzer generator. *Information Processing Letters*, 38:323–328, 1991.
20. A. Oettinger. Automatic syntactic analysis and the pushdown store. In *Proc. of the Symposia in Applied Math*, volume 12, pages 104–129, 1961.
21. L. Quesada. A model-driven parser generator with reference resolution support. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 394–397, 2012.
22. L. Quesada, F. Berzal, and F. J. Cortijo. Lamb — a lexical analyzer with ambiguity support. In *Proceedings of the 6th International Conference on Software and Data Technologies*, volume 1, pages 297–300, 2011.
23. L. Quesada, F. Berzal, and F. J. Cortijo. Fence — a context-free grammar parser with constraints for model-driven language specification. In *Proceedings of the 7th International Conference on Software Paradigm Trends*, pages 5–13, 2012.
24. L. Quesada, F. Berzal, and J.-C. Cubero. A language specification tool for model-based parsing. In *Proceedings of the 12th International Conference on Intelligent Data Engineering and Automated Learning. Lecture Notes in Computer Science*, volume 6936, pages 50–57, 2011.
25. L. Quesada, F. Berzal, and J.-C. Cubero. A tool for model-based language specification. *ArXiv e-prints*, 2011. http://arxiv.org/abs/1111.3970.
26. D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39:25–31, 2006.
27. P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison Wesley, 2006.
28. M. Tomita and J. G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, volume 2, pages 718–721, 1987.
29. J. Turmo, A. Ageno, and N. Cataà. Adaptive information extraction. *ACM Computing Surveys*, vol. 38, no. 2, art. 4, 2006.
30. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.