Declarative Algorithms for Language Theory

Juan Carlos Guzman[1]

Abstract

In this paper we present a high-level approach to implementing algorithms found in predictive parsing. The algorithms are implemented in logic, functional and object-oriented styles, while retaining their original structure. This allows for better understanding on how the techniques represented by the algorithms work.

Introduction

Courses on algorithms and theory of computing are typically taught by giving high-level mathematical descriptions of the operations at hand, and then presenting algorithms at a lower level, which could be easily implementable in the language of choice [1], [4], [5], [9]. This has been the norm, since the high-level notions have not been easy to implement directly. There is, however, a considerable gap between a typical algorithm presented in these texts, and the high-level definition that it implements. We claim in this paper that we can have higher-level descriptions of these algorithms, which can be implemented in current programming languages with ease. The descriptions presented here come from a functional interpretation of the algorithms. The high-level mathematical operators naturally map to higher-order functions. These could be directly implemented in languages like ML, Haskell, and Scheme, but can also be implemented in Prolog, and in Java and C++.

We work in this paper with the notions of first and follow sets, in the context of predictive parsing. We present solutions for first in Prolog, Scheme and Java, and for follow in Scheme, and discuss the equivalent solutions in Prolog and Java. Solutions for both problems exhibit the same style. We implement solutions that are adequate translations from the algorithms. We are more concerned with the expressiveness of the algorithm, than with the actual efficiency that they may have.

Grammars and Caches

We will use the following grammar for expressions. More precisely, assume $G = (N, S, P, E)$, where

$$
\begin{aligned}
N &= \{E, E', T, T', F\} \\
S &= \{a, +, *, (,)\} \\
P &= \{E \rightarrow T\,E', \\
&\quad E' \rightarrow +\,T\,E' \mid e, \\
&\quad T \rightarrow F\,T', \\
&\quad T' \rightarrow *\,F\,T' \mid e, \\
&\quad F \rightarrow a \mid (\,E\,)\}
\end{aligned}
$$

[1] Department of Computer Science, Southern Polytechnic State University, 1100 South Marietta Parkway, Marietta, GA 30060. e-mail: jguzman@spsu.edu

In the above description, *N* represents the set of nonterminals or variables of the grammar, S the alphabet of the generated language, *P* the set of productions and *E* the initial nonterminal. A terminal is represented in blue, and *e* corresponds to the empty string. The above grammar generates phrases like *a*, *a*+*a*, *a***a*+*a*, ((*a*+*a*)**a*)+*a*, etc.

Algorithms for computing first and follow sets need to associate sets to each nonterminal. This is accomplished by using the notion of a mapping, or environment, from nonterminals to sets of terminals. This is implemented in languages under various names. It is known in general as associative memory, or associative arrays, or caches, association lists in Lisp, and as maps in Java. We will use the language neutral name *cache* for them, and will use operations for creating an empty cache (*createEmptyCache*), for selecting the value of a key in the cache (*selCache*) and to assemble a cache by explicitly mapping keys to values $\langle key_1 ? \ value_1, \ldots, key_n ? \ value_n \rangle$..

Computing Fixpoints

Fixpoints – Pumping from Bottom

Intuitively, fixpoints are solutions to equations of the form

$$x = f(x)$$

where *f* is a known function and *x* is unknown.

Fixpoints are introduced in the denotational semantics mainly to provide meaning to recursive functions. In our case, we will use it to compute the reflexive and transitive closure of a function. The technique we will use to compute the fixpoint is known as "pumping from bottom", and is one in which a function *f*, for which the fixpoint is desired, is repeatedly applied to its argument until it has no effect on it. That is, we supply an initial argument, and 'pumping from bottom' provides us with a solution to the fixpoint equation. In short:

$$Ý \ f X = Z,$$

where Ý is the symbol that represents the fixpoint operation, *f* is the function for which the fixpoint is sought, X is the initial value, and *Z* is the actual fixpoint. The initial parameter *X* is interesting just to provide a seed to the function to operate. The computation of the fixpoint is not usually possible, unless certain restrictions are satisfied:

- The partial ordering implied by the information of the domain must be of finite height. In our case, the domain will be the powerset of terminals, and its partial ordering is set inclusion. This has finite height, with weakest element being the empty set and strongest element being the set of all terminals. We will also use tuples of sets for this domain, which are likewise of finite height.

- The function *f* must be *monotonic* (the more information it receives in its parameter, the more information it returns). In short, if $X < Z$, then $f(X) \leq f(Z)$.

If these restrictions are satisfied, then the value *Z* computed by pumping from bottom is the weakest fixpoint for *f* that is stronger than *X*. To compute the weakest fixpoint of the function, just supply the bottom element of the domain—the empty set (Ø). In our case,

$$Ý \ f Ø = Z$$

We will be using a slightly more general version of the equation

$$x = f(x, y_1, \ldots, y_n)$$

where all y_i's are constants. This is equivalent to the equation introduced earlier, but more convenient to use.

Many operations on languages are just applications of closures on simpler operations. Such is the case of the computation of first and follow sets, crucial pieces of information needed to compute the LL(k) or LR(k) table of a predictive parser. These closure operations *are* fixpoints.

In general, to compute a closure on a property we need to start with an initial set that satisfy the property, we need to have an operation to find new elements that satisfy the property, and include them to the set, and we need to do this operation until no new elements can be added.

Implementation of 'Pumping from Bottom'

The algorithm has been written following a functional style, but it is fully translatable to other paradigms. The pumping from bottom operator is the most difficult element to implement, since it is a higher-order operation. Sure enough, most programming languages are first-order, with an add-on limited higher-order capability. In addition, the algorithm makes use of mathematical elements—mostly set operations—that may, or may not be available in your language. If they are not, then they should be implemented. In this example, we use the underlying list data structure to provide the needed set capabilities.

Prolog Implementation. In the following program code, pumpingFromBottom takes a (higher-order) predicate `F`, initial argument `X` to be pumped, the fixpoint version `FixX` of that argument, and all other arguments to the predicate as the list `Args`. The first clause implements the case where the fixpoint has indeed been reached, and the second clause implements the recursive case, where the call to `F` provides a new version of the parameter `X`, which is later pumped. Here, `FixX` is what is known as an 'out parameter', and `F` has been translated to a predicate form. The actual predicate call is built as `F(X,FixX|Args)`, where `X` and `Args` are the 'in parameters' to the `F` predicate, and `FixX` is the result of the predicate. Prolog does not allow variable number of arguments, so all of them are collectively passed as a list (`Args`). The function `F`, however, will receive all of them individually. `F` is the higher-order functional. The following code was implemented using Amzi! Prolog [2].

```
pumpingFromBottom(F,X,X,Args) :-
    Goal =.. [F,X,X|Args],
    call(Goal),!.

pumpingFromBottom(F,X,FixX,Args) :-
    Goal =.. [F,X,NewX|Args],
    call(Goal),!,
    pumpingFromBottom(F,NewX,FixX,Args).
```

Scheme Implementation. Translation to Scheme is simple, as the language supports higher-order functions and variable number of arguments. The implementation shown runs in DrScheme [7].

```
(define (pumping-from-bottom f x . ys)
  (let ((new-x (apply f x ys)))
    (if (equal? x new-x)
        x
        (apply pumping-from-bottom f new-x ys))))
```

Java Implementation. Translation to Java is a bit more complicated, due to restrictions in the type system. Further, Java does not support higher-order functions directly, but they can be implemented using an interface [6], [8]. The higher-order function that is to be "pumped" must then be wrapped within an interface (`ObjectFun`) which contains `f` as one of its required methods. On the bright side, the language has wonderful support for lists, sets, and mappings, which makes coding a breeze. The code for the actual routine is

```
public static Map pumpingFromBottom(ObjectFun fun, Map x, Map args) {
    Map newX = fun.f(x,args);
        if (x.equals(newX))
            return x;
        else
```

```
            return pumpingFromBottom(fun,newX,args);
}
```

Once pumping from bottom is defined, it can be called with any *object* that has implemented `ObjectFun`. This interface is defined as

```
abstract interface ObjectFun {
      public abstract Map f(Map x, Map args);
}
```

Computing First and Follow Sets

For the sake of simplicity, we will assume that a single lookahead is sufficient, so the first and follow sets will compute single character lookaheads. You may recall that, for any grammar G [9],

$$first(\mathrm{a}) = \{\ \mathrm{trunc}(w) \mid \mathrm{a} \Rightarrow^* w\ \},\ \text{for}\ \mathrm{a} \in (N \cup \mathrm{S})^*,\ w \in \mathrm{S}^*$$

where

$$\mathrm{trunc}(\varepsilon) = \varepsilon$$

$$\mathrm{trunc}(aw) = a, \quad \text{for}\ a \in \mathrm{S},\ w \in \mathrm{S}^*$$

The notion of the first set is defined for all strings, but its computation is tightly tied on the values of these sets for each of the nonterminals of a grammar, i.e., $first(a) = a$, but $first(E+a) = \{a, (\}$ because E can derive in phrases starting with each of these terminals. This operation cannot be computed as defined, but we will be able to compute it by observing just the first characters that are generated. For this, we will use an auxiliary operation, bounded concatenation ($\oplus$), which takes two strings (or two string up to length 1), and returns the first character of the concatenated string. This operation is further generalized for sets of strings, with the following definition

$$F_1 \oplus F_2 = (F_1 - \{\ e\ \}) \cup F_2, \quad \text{if}\ e \in F_1$$

$$F_1 \oplus F_2 = F_1, \quad \text{otherwise}$$

We will have the operation `boundedConcat` implement precisely this operator.

Algorithm for First Sets

In order to be able to readily compute the first set for any string with respect to a grammar, we need to have computed these sets for all nonterminals of the grammar. For this, we will use a cache—an environment where nonterminals are associated to first sets. Therefore, we will start with a version of first that computes its values based on a cache (*first*), and later abstract this element out by computing the fixpoint of the cache (*First*). The fixpoint cache is computed by the function *firsts*. We also need to define the operation that shows how these caches are improved (*CachedFirst*)

$$\mathit{first\ G\ Cache}\ \varepsilon = \{\ \varepsilon\ \}$$

$$\mathit{first\ G\ Cache}\ a = \{\ a\ \}$$

$$\mathit{first\ G\ Cache}\ A = \mathit{selCache}\ A\ \mathit{Cache}, \quad \text{if}\ A \in N$$

$$\mathit{first\ G\ Cache}\ X\alpha = (\mathit{first\ G\ Cache}\ X) \oplus (\mathit{first\ G\ Cache}\ \alpha), \quad \text{if}\ X \in N \cup T \text{ and } \alpha \in (N \cup T)^*$$

$$\mathit{CachedFirst\ G\ Cache} = \langle \ldots A_i?\ ((\mathit{first\ G\ Cache}\ \alpha_1) \cup \ldots \cup (\mathit{first\ G\ Cache}\ \alpha_n))\ldots \rangle,$$

$$\text{for all}\ A_i \in N, \text{ and } A_i \rightarrow \alpha_j \in P$$

$$firsts\ G = Ý\ CachedFirst\ (createEmptyCache\ G)\ G$$

$$First\ G\ \alpha = first\ G\ (firsts\ G)\ \alpha$$

Note that the pumping from bottom operation improves the original cache until it finds a fixpoint. The following table summarizes the different caches that are computed:

N	First Caches (Iteration)				
	0	1	2	3	4
E	Ø	Ø	Ø	{ *a*, (}	{ *a*, (}
E'	Ø	{ +, *e* }	{ +, *e* }	{ +, *e* }	{ +, *e* }
T	Ø	Ø	{ *a*, (}	{ *a*, (}	{ *a*, (}
T'	Ø	{ *, *e* }	{ *, *e* }	{ *, *e* }	{ *, *e* }
F	Ø	{ *a*, (}	{ *a*, (}	{ *a*, (}	{ *a*, (}

Note that the fixpoint is reached at the 3rd iteration, but this is not known until the 4th iteration shows the same result. Also, note that *First* abstracts away the notion of the cache—it just computes it internally. The last cache contains the solution to the problem—the problem was solved by the use of pumping from bottom technique—but it is made available through the function *First*.

Implementation of First Sets

Here we will present Prolog, Scheme and Java implementations of for the *first* set computation

Prolog Implementation. Prolog's limitations are more apparent at this point. It lacks iterators, which are basic tools for list processing. They could have been introduced as higher-order predicates, but that would change the character of the language. Instead, the hand-coded `recomputeFiRHSs` is the needed iterator. Prolog also lacks simple mechanisms for abstracting the names of the supporting predicates. Prolog's straightforward implementation of the function *First*(*G*,a) by recomputing the fixpoint cache for each call to *First* would be highly inefficient. Another alternative would be to *assert* the cache, but this is highly inconvenient. Finally, another alternative is to use *first*, with the fixpoint cache, and this is exactly the implemented function `first`.

```
first(Cache,[],[[]]) :- !.
first(Cache,[X|Xs],Y) :-
    first(Cache,X,Yl),
    first(Cache,Xs,Yr),
    boundedconcat(Yl,Yr,Y).
first(Cache,X,Y) :-
    nonterminal(X,Cache),!,
    selCache(X,Cache,Y).
first(_,X,[X]) :-
    atomic(X).

cachedFirst(_,[],[]) :- !.
cachedFirst(Cache,[[A,NewFirstSet]|RNewCache],[[A|RHSs]|RGrammar]) :-
    recomputeFiRHSs(Cache,RHSs,NewFirstSet),
    cachedFirst(Cache,RNewCache,RGrammar).

recomputeFiRHSs(Cache,[],[]) :- !.
recomputeFiRHSs(Cache,[RHS|RHSs],NewFirstSet) :-
    first(Cache,RHS,FirstRHS),
    recomputeFiRHSs(Cache,RHSs,FirstRHSs),
```

```
    union(FirstRHS,FirstRHSs,NewFirstSet).

firsts(Grammar,Cache) :-
    createCache(Grammar,BottomCache),
    pumpingFromBottom(cachedFirst,Cache,BottomCache,[Grammar]).
```

Scheme Implementation. Scheme iterators `map` and `fold` are just the right tools for solving the problem. Although Scheme allows for local function definitions, and hence some information hiding, it seemed unnecessary to do it here, since it would create a larger S-expression. After all, lisp programmers are used to a flat environment. The *First* function will not be explicitly defined. However, Lisp programmers have tools to specialize `first` with the fixpoint cache into the function *First*.

```
 (define (first cache word)
  (cond ((null? word)
         '(()))
        ((list? word)
         (bounded-concat (first cache (car word))
                         (first cache (cdr word))))
        ((nonterminal? word cache)
         (sel-cache word cache))
        (else
         (list word))))

(define (cached-first cache grammar)
  (let ((f (lambda (A-prods)
             (let ((A (car A-prods))
                   (prods (cdr A-prods)))
               (list A
                     (qs
                      (fold union
                            '()
                            (map (lambda (RHS)
                                   (first cache RHS))
                                 prods)))))))))
    (map f grammar)))

(define (firsts grammar)
  (pumping-from-bottom cached-first (create-cache grammar) grammar))
```

Java Implementation. As before, the type system adds to the complexity of the solution. In order to solve this problem, the `cachedFirst` method—the function that is to complete the meaning of pumping-from-bottom—has to be coded in a separate class, which must implement the `ObjectFun` interface. We will show just the relevant methods to the solution.

```
class CachedFirst implements ObjectFun {

    ......

    HashSet first(Map grammar, Map cache, List word) {
        String emptyString[]={""};
        HashSet acc = new HashSet(Arrays.asList(emptyString));
        for (Iterator iterator=word.iterator(); iterator.hasNext(); ) {
            String x = (String) iterator.next();
            if (nonterminal(x)) {
                acc = boundedConcat(acc, (HashSet) cache.get(x));
            }
            else {
```

```
                HashSet tmp = new HashSet();
                tmp.add(x);
                acc = boundedConcat(acc, tmp);
            }
        }
        return acc;
    }

    public Map f(Map cache, Map grammar) {
        Map newCache = new TreeMap();
        for (Iterator it=grammar.keySet().iterator();it.hasNext();) {
            String LHS = (String) it.next();
            HashSet RHSs = (HashSet) grammar.get(LHS);
            HashSet newSet = new HashSet();
            for (Iterator it2=RHSs.iterator(); it2.hasNext(); ) {
                List RHS = (List) it2.next();
                newSet.addAll(first(grammar,cache,RHS));
            }
            newCache.put(LHS,newSet);
        }
        return newCache;
    }
}


public class PredictiveParsing {

    ……
    public static Map firsts(Map grammar) {
        return pumpingFromBottom(new CachedFirst(),newCache(grammar),grammar);
    }
   ……
}
```

Algorithm of Follow Sets

The follow sets are described in way analogous to the first sets. However, they require the information from the grammar in a different way—the algorithm requests the 'right contexts' of nonterminals. This is not apparent in the algorithm, but it will surface in the implementations. The *follow* operation is defined only for nonterminals, and it just seeks the value of the nonterminals in the supplied cache. The *Follow* operation, on the other hand, computes the fixpoint cache, and therefore abstracts the cache from the operation. The *CachedFollow* operation indicates how a cache is improved.

$$follow\ G\ Cache\ A \;=\; selCache\ A\ Cache, \qquad \text{if } A \in N$$

$$CachedFollow\ G\ Cache \;=$$

$$\langle \ldots A_i?\ \ ((First\ G\ \alpha_1) \oplus (selCache\ B_1\ Cache)) \cup \ldots \cup ((First\ G\ \alpha_n) \oplus (selCache\ B_n\ Cache))) \ldots \rangle,$$

$$\text{for all } A_i \in N, \text{ and } B_j \rightarrow \beta_j A_i \alpha_j \in P$$

$$Follows\ G \;=\; (\acute{Y}\ CachedFollow\ (createEmptyCache\ G)\ G)$$

$$Follow\ G\ A \;=\; follow\ G\ (Follows\ G)\ A$$

As in the case of the first sets, the pumping from bottom operation improves the original cache until it finds a fixpoint. The following table summarizes the different caches that have been computed:

N	Follow Caches (Iteration)				
	0	1	2	3	4
E	Ø	{ $,) }	{ $,) }	{ $,) }	{ $,) }
E'	Ø	Ø	{ $,) }	{ $,) }	{ $,) }
T	Ø	{ + }	{ $,), + }	{ $,), + }	{ $,), + }
T'	Ø	Ø	{ + }	{ $,), + }	{ $,), + }
F	Ø	{ *a*, (}	{ +, * }	{ $,), +, * }	{ $,), +, * }

Note that, again, the fixpoint is reached at the 3rd iteration, and that this fact is known after computing the 4th iteration. Also, note that *First* abstracts away the notion of the cache—it just computes it internally. The last cache contains the solution to the problem—the problem was solved by the use of pumping from bottom technique—but it is made available through the function *First*.

Implementation of Follow Sets

It is customary to obtain an intermediate data structure (a cache) that has readily access to the information of the right-contexts. This seems as a departure from the previous algorithm, but it really is not. Following is the set of equations that first and follow sets compute in order to obtain their values:

N	***first*(*N*)**	***follow*(*N*)**
E	*first*(*T*)⊕*first*(*E'*)	*first*($)⊕*follow*(*E*) ∪ *first*())⊕*follow*(*F*)
E'	*first*(+)⊕*first*(*T*)⊕*first*(*E'*) ∪ *first*(ε)	*first*(*e*)⊕*follow*(*E*) ∪ *first*(*e*)⊕*follow*(*E'*)
T	*first*(*F*)⊕*first*(*T'*)	*first*(*E'*)⊕*follow*(*E*) ∪ *first*(*E'*)⊕*follow*(*E'*)
T'	*first*(*)⊕*first*(*F*)⊕*first*(*T'*) ∪ *first*(ε)	*first*(*e*)⊕*follow*(*T*) ∪ *first*(*e*)⊕*follow*(*T'*)
F	*first*(*a*) ∪ *first*(()⊕*first*(*E*)⊕*first*())	*first*(*T'*)⊕*follow*(*T*) ∪ *first*(*T'*)⊕*follow*(*T'*)

As can be seen, the equations for first mimic the production structure, whereas those for follow do not. So while it seems reasonable to iterate over the grammar for computing the first sets, it is advantageous to compute the structure of the right column as a basis of computing follow. This is not strictly necessary. The encoding of these operations, as a cache will be called `FollowEnc`.

The higher-order operation introduced before (pumping from bottom) can be reused in this algorithm. There are no more significant challenges. Here we present the Scheme implementation.

Scheme Implementation. Besides the use of the map and fold iterators, we are creating a specialized first function to be used with these functions.

```
(define (follow cache A)
  (sel-cache A cache))

(define (cached-follow cache first followEnc)
  (let ((f (lambda (A-specs)
             (let ((A (car A-specs))
                   (specs (cdr A-specs)))
               (list A
                     (qs
                      (fold union
                            '()
                            (map (lambda (spec)
```

```
                                    (bounded-concat
                                     (first (cadr spec))
                                     (follow cache (car spec))))
                                   specs))))))))
    (map f followEnc)))

(define (follows followEnc first)
  (pump-from-bottom cached-follow (create-cache followEnc) first followEnc))
```

Note that the call to this function should supply the specialized first function: `(lambda (N) (first Cache N))`, where `Cache` is the previously computed cache for the first sets.

Java Implementation. The Java implementation requires a new implementation of `ObjectFun` (`CachedFollow`). This implementation can be supplied with the first cache. A slightly more sophisticated implementation calls for a redesign of the `CachedFirst` class, so that it hides the cache, and just makes a method `first` public.

Conclusion

In this paper, we have presented a higher-level exposition of algorithms for computing the first and follow sets, necessary for predictive parsing, and have given algorithms implemented in logic, functional and object-oriented programming. This idea can be extended to a variety of algorithms that appear in theory of languages, and in other areas of Computer Science. The value of these implementations is how direct high-level mathematical notions are translated to real programs. We expect that newer notions in the field of computing, such as patterns and frameworks [3], [10] may supplement the exposition of these notions, for their better understanding.

The algorithms presented in this paper have been used in a course on Concepts of Programming Languages, both at undergraduate and at graduate level. Students are requested to implement parts of it in any of the programming languages used.

References

1. Aho, A., Sethi, R. and Ullman, J. (1986) Compilers—Principles, Techniques and Tools, Addison Wesley.

2. Amzi! Inc., *Amzi! Prolog*, http://www.amzi.com/ .

3. Gamma, E, Helm, R. et al (1995) Design Patterns, Addison Wesley.

4. Hopcroft, J., Motwani, R. and Ullman, J. (2001) Introduction to Automata Theory, Languages, and Computation (2nd Ed.), Addison Wesley.

5. Lewis, H. and Papadimitriou, C. (1998) Elements of the Theory of Computation (2nd Ed.), Prentice Hall.

6. Naugler, D. (2003) Functional Programming in Java," In *The Journal of Computing in Small Colleges*, The Consortium for Computing in Small Colleges.

7. PLT, *DrScheme*, http://www.drscheme.org/.

8. Setzer, A. (2002) "Java as a Functional Programming Language," In *Types 2002, Proceedings of the workshop Types for Proofs and Programs*, Lecture Notes in Computer Science 2646, Springer-Verlag.

9. Sudkamp, T. (1997) Languages and Machines, Addison Wesley.

10. Szyperski, C. (2002) Component Software, Addison Wesley.

Juan Carlos Guzmán

Ing. en Computación, Universidad Simón Bolívar, Caracas, Venezuela, 1983. M.S., M.Phil., Yale University, 1986. Ph.D., Yale University, 1993. He has been a faculty member of Southern Polytechnic State University, in Marietta, Georgia, since 1999.