Top Down Operator Precedence

Vaughan R. Pratt
Massachusetts Institute of Technology 1973

Work reported herein was supported in part at Stanford by the National Science Foundation under grant no GJ 992, and the Office of Naval Research under grant number N-00014-67-A-O112-0057 NR 044-402; by IBM under a post-doctoral fellowship at Stanford; by the IBM T.J. Watson Research Center, Yorktown Heights, N.Y.; and by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number N00014-70-O362-0006 and the National Science Foundation under contract number GJO0-4327. Reproduction in whole or in part is permitted for any purpose of the United States Government.

1. Survey of the Problem Domain

There is little agreement on the extent to which syntax should be a consideration in the design and implementation of programming languages. At one extreme, it is considered vital, and one may go to any lengths [Van Wijngaarden 1969, McKeeman 1970] to provide adequate syntactic capabilities. The other extreme is the spartan denial of a need for a rich syntax [Minsky 1970]. In between, we find some language implementers willing to incorporate as much syntax as possible provided they do not have to work hard at it [Wirth 1971].

In this paper we present what should be a satisfactory compromise for a respectably large proportion of language designers and implementers. We have in mind particularly

1. those who want to write translators and interpreters (soft, firm or hardwired) for new or extant languages without having to acquire a large system to reduce the labor, and
2. those who need a convenient yet efficient language extension mechanism accessible to the language user.

The approach described below is very simple to understand, trivial to implement, easy to use, extremely efficient in practice if not in theory, yet flexible enough to meet most reasonable syntactic needs of users in both categories (i) and (ii) above. (What is "reasonable" is addressed in more detail below). Moreover, it deals nicely with error detection.

One may wonder why such an "obviously" utopian approach has not been generally adopted already. I suspect the root cause of this kind of oversight is our universal preoccupation with BNF grammars and their various offspring: type 1 [Chomsky 1959], indexed [Aho 1968], macro [Fischer 1968], LR(k) [Knuth 1965], and LL(k) [Lewis 1968] grammars, to name a few of the more prominent ones, together with their related automata and a large body of theorems. I am personally enamored of automata theory per se, but I am not impressed with the extent to which it has so far been successfully applied to the writing of compilers or interpreters. Nor do I see a particularly promising future in this direction. Rather, I see automata theory as holding back the development of ideas valuable to language design that are not visibly in the domain of automata theory.

Users of BNF grammars encounter difficulties when trying to reconcile the conflicting goals of practical generality (coping simultaneously with symbol tables, data types and their inter-relations, resolution of ambiguity, unpredictable demands by the BNF user, top-down semantics, etc.) and theoretical efficiency (the guarantee that any translator using a given technique will run in linear time and reasonable space, regardless of the particular grammar used). BNF grammars alone do not deal adequately with either of these issues, and so they are stretched in some directions to increase generality and shrunk in others to improve efficiency. Both of these operations tend to increase the size of the implementation "life-support" system, that is, the software needed to pre-process grammars and to supervise the execution of the resulting translator. This makes these methods correspondingly less accessible and less pleasant to use. Also, the stretching operation is invariably done gingerly, dealing only with those issues that have been anticipated, leaving no room for unexpected needs.

I am thinking here particularly of the work of Lewis and Stearns and their colleagues on LL(k) grammars, table grammars, and attributed translations. Their approach, while retaining the precision characteristic of the mathematical sciences (which is unusual in what is really a computer-engineering and human-engineering problem), is tempered with a sensitivity to the needs of translator writers that makes it perhaps the most promising of the automata-theoretic approaches. To demonstrate its practicality, they have embodied their theory in an efficient Algol compiler.

A number of down-to-earth issues are not satisfactorily addressed by their system – deficiencies which we propose to make up in the approach below; they are as follows.

1. From the point of view of the language designer, implementer or extender, writing an LL(k) grammar, and keeping it LL(k) after extending it, seems to be a black art, whose main redeeming feature is that the life-support system can at least localize the problems with a given grammar. It would seem preferable, where possible, to make it easier for the user to write acceptable grammars on the first try, a property of the approach to be presented here.
2. There is no "escape clause" for dealing with non-standard syntactic problems (e.g. Fortran format statements). The procedural approach of this paper makes it possible for the user to deal with difficult problems in the same language he uses for routine tasks.
3. The life-support system must be up, running and debugged on the user's computer before he can start to take advantage of the technique. This may take more effort than is justifiable for one-shot applications. We suggest an approach that requires only a few lines of code for supporting software.
4. Lewis and Stearns consider only translators, in the context of their LL(k) system; it remains to be determined how effectively they can deal with interpreters. The approach below is ideally suited for interpreters, whether written in software, firmware or hardware.

2. Three Syntactic Issues

To cope with unanticipated syntactic needs, we adopt the simple expedient of allowing the language implementer to write arbitrary programs. By itself, this would represent a long step backwards; instead, we offer in place of the rigid structure of a BNF-oriented meta-language a modicum of supporting software, and a set of guidelines on how to write modular, efficient, compact and comprehensible translators and interpreters while preserving the impression that one is really writing a grammar rather than a program.

The guidelines are based on some elementary assumptions about the primary syntactic needs of the average programmer.

First, the programmer already understands the semantics of both the problem and the solution domains, so that it would seem appropriate to tailor the syntax to fit the semantics. Current practice entails the reverse.

Second, it is convenient if the programmer can avoid having to make up a special name for every object his program computes. The usual way to do this is to let the computation itself name the result -- e.g. the object which is the second argument of `+` in the computation `a+b*c` is the result of the computation `b*c`. We may regard the relation "is an argument of" as defining a class of trees over computations; the program then contains such trees, which need conventions for expressing linearly.

Third, semantic objects may require varying degrees of annotation at each invocation, depending on how far the particular invocation differs in intent from the norm [e.g. for loops that don't start from 1, or don't step by 1). The programmer needs to be able to formulate these annotations within the programming language.

There are clearly many more issues than these in the design of programming languages. However, these seem to be the ones that have a significant impact on the syntax aspects. Let us now draw inferences from the above assumptions.

2.1 Lexical Semantics versus Syntactic Semantics

The traditional mechanism for assigning meanings to programs is to associate semantic rules with phrase-structure rules, or equivalently, with classes of phrases. This is inconsistent with the following reasonable model of a programmer.

The programmer has in mind a set of semantic objects. His natural inclination is to talk about them by assigning them names, or tokens. He then makes up programs using these tokens, together with other tokens useful for program control, and some purely syntactic tokens. (No clear-cut boundary separates these classes.) This suggests that it is more natural to associate semantics with tokens than with classes of phrases.

This argument is independent of whether we specify program control explicitly, as in Algol-like languages, or implicitly, as in Planner-Conniver-like languages. In either case, the programmer wants to express his instructions or intentions concerning certain objects.

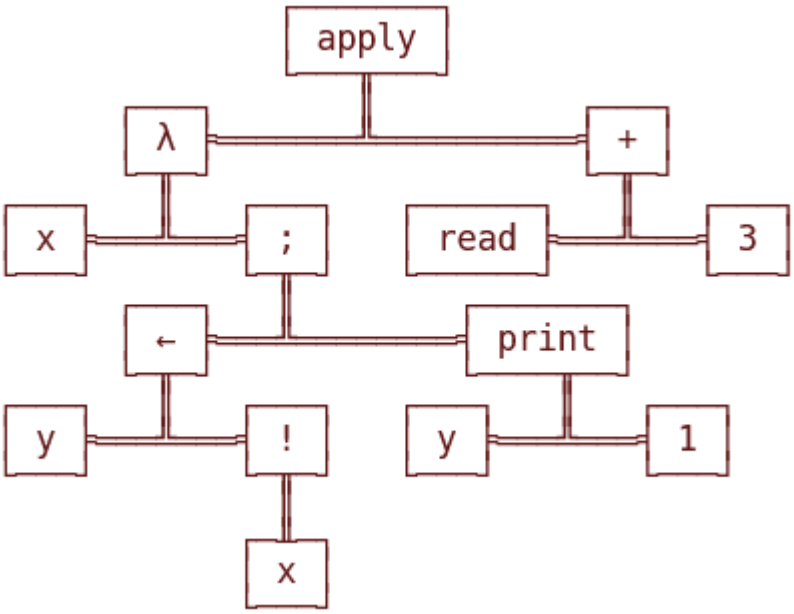
When a given class of phrases is characterized unambiguously by the presence of a particular token, the effect is the same, but this is not always the case in a BNF-style semantic specification, and I conjecture that the difficulty of learning and using a given language specified with a BNF grammar increases in proportion to the number of rules not identifiable by a single token. The existence of an operator grammar [Floyd 1963] for Algol 60 provides a plausible account of why people succeed in learning Algol, a process known not to be strongly correlated with whether they have seen the BNF of Algol.

There are two advantages of separating semantics from syntax in this way. First, phrase-structure rules interact more strongly than individual tokens because rules can share non-terminals whereas tokens have nothing to share. So our assignment of semantics to tokens has a much better chance of being modular than an assignment to rules. Thus one can tailor the language to one's needs by selecting from a library, or writing, the semantics of just those objects that one needs for the task in hand, without having to worry about preordained interactions between two semantic objects at the syntactic level. Second, the language designer is free to develop the syntax of his language without concern for how it will affect the semantics; instead, the semantics will affect decisions about the syntax. The next two issues (linearizing trees and annotating tokens) illustrate this point well. Thus syntax is the servant of semantics, an appropriate relationship since the substance of the message is conveyed with the semantics, Variations in syntax being an inessential trimming added on human-engineering grounds.

The idea of lexical semantics is implicit in the usual approach to macro generation, although the point usually goes unmentioned. I suspect many people find syntax macros [Leavenworth 1966] appealing for reasons related to the above discussion.

2.2 Conventions for Linearizing Trees

We argued at the beginning of section 2 that in order to economize on names the programmer resorted to the use of trees. The precedent has a long history of use of the same trick in natural language. Of necessity (for one-dimensional channels) the trees are mapped into strings for transmission and decoded at the other end. We are concerned with both the human and computer engineering aspects of the coding. We may assume the trees look like, e.g.



That is, every node is labelled with a token whose arguments if any are its subtrees. Without further debate we shall adopt the following conventions for encoding trees as strings.

1. The string contains every occurrence of the tokens in the tree, (which we call the *semantic tokens*, which include procedural items such as `if`, `;`) together with some additional *syntactic tokens* where necessary.
2. Subtrees map to contiguous substrings containing no semantic token outside that subtree.
3. The order of arguments in the tree is preserved. (Naturally these are oriented trees in general.)
4. A given semantic token in the language, together with any related syntactic tokens, always appear in the same place within the arguments; e.g. if we settle for `+a,b`, we may not use `a+b` as well. (This convention is not as strongly motivated as (i)-(iii); without it, however, we must be overly restrictive in other areas more important than this one.)

If we insist that every semantic token take a fixed number of arguments, and that it always precede all of its arguments (prefix notation) we may unambiguously recover the tree from the string (and similarly for postfix) as is well known. For a variable number of arguments, the LISP solution of having syntactic tokens (parentheses) at the beginning and end of a subtree's string will suffice.

Many people find neither solution particularly easy to read. They prefer...

```
ab² + cd² = 4 sin (a+b)
```

to...

```
= + * a ↑ b 2 * c ↑ d 2 * 4 sin + a b
```

or to...

```
(= (+ (* a (↑ b 2)) (* c (↑ d 2))) (* 4 (sin (+ a b))))
```

although they will settle for...

```
a*b↑2 + c*d↑2 = 4*sin(a+b)
```

in lieu of the first if necessary. (But I have recently encountered some LISP users claiming the reverse, so I may be biased.)

An unambiguous compromise is to require parentheses but move the tokens, as in...

```
(((a * (b ↑ 2)) + (c * (d ↑ 2))) = (4 * (sin (a + b))))
```

This is actually quite readable, if not very writable, but it is difficult to tell if the parentheses balance, and it nearly doubles the number of symbols. Thus we seem forced inescapably into having to solve the problem that operator precedence was designed for, namely the association problem. Given a substring `AEB` where `A` takes a right argument, `B` a left, and `E` is an expression, does `E` associate with `A` or `B`?

A simple convention would be to say `E` always associates to the left. However, in `print a + b`, it is clear that `a` is meant to associate with `+`, not `print`. The reason is that `(print a) + b` does not make any conventional sense, `print` being a procedure not normally returning an arithmetic value. The choice of `print (a + b)` was made by taking into account the data types of `print`'s right argument, `+`'s left argument, and the types returned by each. Thus the association is a function of these four types (call them a_A, r_A, a_B, r_B for the argument and result respectively of `A` and `B`) that also takes into account the legal coercions (implicit type conversions) Of course, sometimes both associations make sense, and sometimes neither. Also r_A or r_B may depend on the type of `E`, further complicating matters.

One way to resolve the issue is simply to announce the outcome in advance for each pair `A` and `B`, basing the choices on some reasonable heuristics. Floyd [1963] suggested this approach, called operator precedence. The outcome was stored in a table. Floyd also suggested a way of encoding this table that would work in a small number of cases, namely that a number should be associated with each argument position by means of precedence functions over tokens; these numbers are sometimes called "binding powers". Then `E` is associated with the argument position having the higher number. Ties need never occur if the numbers are assigned carefully; alternatively, ties may be broken by associating to the left, say. Floyd showed that Algol 60 could be so treated.

One objection to this approach is that there seems to be little guarantee that one will always be able to find a set of numbers consistent with one's needs. Another objection is that the programmer has to learn as many numbers as there are argument positions, which for a respectable language may be the order of a hundred. We present an approach to language design which simultaneously solves both these problems, without unduly restricting normal usage, yet allows us to retain the numeric approach to operator precedence.

The idea is to assign data types to classes and then to totally order the classes. An example might be, in ascending order, Outcomes (e.g., the pseudo-result of `print`), Booleans, Graphs (e.g. trees, lists, plexes), Strings, Algebraics (e.g. integers, complex nos, polynomials, real arrays) and References (as on the left side of an assignment.) We write `Strings < References`, etc.

We now insist that the class of the type at any argument that might participate in an association problem not be less than the class of the data type of the result of the function taking that argument. This rule applies to coercions as well. Thus we may use `<` since its argument types (Algebraics) are each greater than its result type (Boolean.) We may not write `length x` (where `x` is a string or a graph) since the argument type is less than the result type. However, `|x|` would be an acceptable substitute for `length x` as its argument cannot participate in an association problem.

Finally, we adopt the convention that when all four data types in an association are in the same class, the association is to the left.

These restrictions on the language, while slightly irksome, are certainly not as demanding as the LISP restriction that every expression have parentheses around it. Thus the following theorem should be a little surprising, since it implies that the programmer never need learn *any* associations!

Theorem 1

Given the above restrictions, every association problem has at most one solution consistent with the data types of the associated operators.

Proof

Let `...AEB...` be such a problem, and suppose `E` may associate with both `A` and `B`. Hence because `E` associates with $A\ [a_A] \geqq [r_A] \geqq [a_B] \geqq [r_B]$ (type `x` is in `class[x]`) since coercion is non-increasing, and the type class of the result of `...AE` is not greater than $[r_A]$, by an obvious inductive proof. Also for `E` with `B`, $[a_B] \geqq [r_B] \geqq [a_A] \geqq [r_A]$ similarly. Thus $[a_A] = [a_B]$, $[r_A] = [r_B]$, and $[a_A] = [r_B]$, that is, all four are in the same class. But the convention in this case is that `E` must associate with `A`, contradicting our assumption that `E` could associate with `B` as well.

This theorem implies that the programmer need not even think about association except in the homogeneous case (all four types in the same class), and then he just remembers the left-associativity rule. More simply, the rule is "always associate to the left unless it doesn't make sense".

What he does have to remember is how to write expressions containing a given token (e.g. he must know that one writes `|x|`, not `length x`) and which coercions are allowed. These sorts of facts are quite modular, being contained in the description of the token itself independently of the properties of any other token, and should certainly be easier to remember than numbers associated with each argument.

Given all of the above, the obvious way to parse strings (i.e. recover their trees) is, for each association problem, to associate to the left unless this yields semantic nonsense. Unfortunately, nonsense testing requires looking up the types r_A and a_B and verifying the existence of a coercion from r_A to a_B. For translation this is not serious, but for interpretation it might slow things down significantly. Fortunately, there is an efficient solution that uses operator precedence functions.

Theorem 2

Given the above restrictions on a language, there exists an assignment of integers to the argument positions of each token in the language such that the correct association, if any, is always in the direction of the argument position with the larger number, with ties being broken to the left.

Proof

First assign *even* integers (to make room for the following interpolations) to the data type classes. Then to each argument position assign an integer lying strictly (where possible) between the integers corresponding to the classes of the argument and result types. To see that this assignment has the desired property, consider the homogeneous and non-homogeneous cases in the problem `...AEB...` as before.

In the homogeneous case all four types are in the same class and so the two numbers must be equal, resulting in left association as desired. If two of the data types are in different classes, then one of the inequalities in $[a_A] \geqq [r_A] \geqq [a_B] \geqq [r_B]$ (assuming `E` associates with `A`) must be strict. If it is the first or third inequality, then `A`'s number must be strictly greater than `B`'s because of the strictness condition for lying between different argument and result type class numbers. If it is the second inequality then `A`'s number is greater than `B`'s because `A`'s result type class number is greater than `B`'s argument one. A similar argument holds if `E` associates with `B`, completing the proof.

Thus Theorem 1 takes care of what the programmer needs to know, and Theorem 2 what the computer needs to know. In the former case we are relying on the programmer's familiarity with the syntax of each of his tokens; in the latter, on the computer's agility with numbers. Theorem 2 establishes that the two methods are equivalent.

Exceptions to the left association rule for the homogeneous case may be made for classes as a whole without upsetting Theorem 2. This can be done by decrementing by 1 the numbers for argument positions to the right of all semantic tokens in that class, that is, the right binding powers. Then the programmer must remember the classes for which the exception holds. Applying this trick to some tokens in a class but not to others gives messy results, and so does not seem worth the extra effort required to remember the affected tokens.

The non-semantically motivated conventions about `and`, `or`, `+` and `↑` may be implemented by further subdividing the appropriate classes (here the Booleans and Algebraics) into pseudo-classes, e.g. `terms < factors < primaries`, as in the BNF for Algol 60. Then `+` is defined over terms, `*` over factors and `↑` over primaries, with coercions allowed from primaries to factors to terms. To be consistent with Algol, the primaries should be a right associative class.

While these remarks are not essential to the basic approach, they do provide a sense in which operator precedence is more than just an ad hoc solution to the association problem. Even if the language designers find these guidelines too restrictive, it would not contradict the fact that operator precedence is in practice a quite satisfactory solution, and we shall use it in the approach below regardless of whether the theoretical justification is reasonable. Nevertheless we would be interested to see a less restrictive set of conventions that offer a degree of modularity comparable with the above while retaining the use of precedence functions. The approach of recomputing the precedence functions for every operator after one change to the grammar is not modular, and does not allow flexible access to individual items in a library of semantic tokens.

An attractive alternative to precedence functions would be to dispose of the ordering and rely purely on the data types and legal coercions to resolve associations. Cases which did not have a unique answer would he referred back to the programmer, which would be acceptable in an on-line environment, but undesirable in batch mode. Our concern about efficiency for interpreters could be dealt with by having the outcome of each association problem marked at its occurrence, to speed things up on subsequent encounters. Pending such developments, operator precedence seems to offer the best overall compromise in terms of modularity, ease of use and memorizing, and efficiency.

The theorems of this section may be interpreted as theorems about BNF grammars, with the non-terminals playing the role of data type classes. However, this is really a draw-back of BNF; the non-terminals tempt one to try to say everything with just context-free rules, which brings on the difficulties mentioned in Section 1. It would seem preferable to refer to the semantic objects directly rather than to their abstraction in an inadequate language.

2.3 Annotation

When a token has more than two arguments, we lose the property of infix notation that the arguments are delimited. This is a nice property to retain, partly for readability, partly because complications arise, e.g., if – is to be used as both an infix and a prefix operator; (also has this property as an infix it denotes application, as a prefix, a no-op. Accordingly we require that all arguments be delimited by at least one token; such a grammar Floyd [1963] calls an operator grammar. Provided the number of arguments remains fixed it should be clear that no violence is done by the extra arguments to theorems 1 and 2, since the string of tokens and arguments including the two arguments at each end plays the same syntactic role as the single semantic token in the two argument case. We shall call the semantic tokens associated with a delimiter its parents.

An obvious choice of delimiters is commas. However, this is not as valuable as a syntactic token that documents the role of the argument following it. For example, `if a then b else c` is more readable (by a human) then `if a, b, c`. Other examples are `print x format f`, `i from s to f by d while c do b`, `log x base b`, `solve e using m`, `x between y and z`, etc.

Sometimes arguments may be frequently used constants, e.g., `for i from 1 to n by 1 while true do b`. If an argument is uniquely identified by its preceding delimiter, an obvious trick is to permit the omission of that argument and its token to denote that a default value should be used. Thus, we may abbreviate the previous example to `for i to n do b`, as in extended Algol 68. Other obvious defaults are `log x` for `log x base 2`, `if x then y` for `if x then y else nil`, and so on. Note that various arguments now may be involved in associations, depending on which ones are absent.

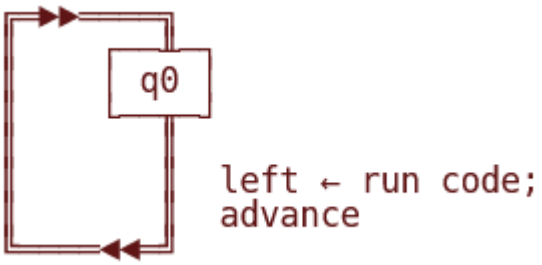
Another situation is that of the variable length parameter list, e.g., `clear a, b, c, d`. Commas are more appropriate here, although again we may need more variety, as in `turn on a on b off g on m off p off t` (in which the unnamed switches or bits are left as they are). All of these examples show that we want to be able to handle quite a variety of situations with default parameters and variable-length parameter lists. No claim is made that the above examples exhaust the possibilities, so our language design should make provision not only for the above, but for the unexpected as well. This is one reason for preferring a procedural embedding of semantics; we can write arbitrary code to find all the arguments when the language designer feels the need to complicate things.

3. Implementation

In the preceding section we argued for lexical semantics, operator precedence and a variety of ways of supplying arguments. In this section we reduce this to practice.

To combine lexical semantics with a procedural approach, we assign to each semantic token a program called its *semantic code*, which contains almost all the information about the token. To translate or interpret a string of tokens, execute the code of each token in turn from left to right.

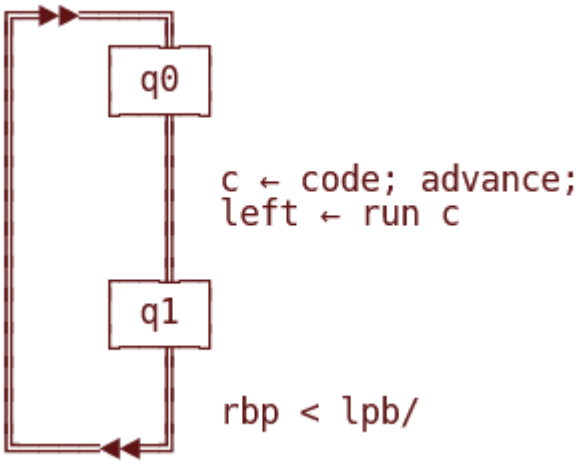
Many tokens will expect arguments, which may occur before or after the token. If the argument always comes before, as with unary postfix operators such as "!", we may parse expressions using the following one-state parser.



This parser is initially positioned at the beginning of the input. It runs the code of the current token, stores the result in a variable called `left`, advances the input, and repeats the process. If the input is exhausted, then by default the parser halts and returns the value of `left`. The variable `left` may be consulted by the code of the next token, which will use the value of `left` as either the translation or value of the left-hand argument, depending on whether it is translating or interpreting.

Alternatively, all arguments may appear on the right, as with unary prefix operators such as `log` and `sin`. In this case the code of a prefix operator can get its argument by calling the code of the following token. This process will continue recursively until a token is encountered (e.g., a variable or a constant) that does not require an argument. The code of this token returns the appropriate translation and then so does the code of each of the other tokens, in the reverse of the order in which they were called.

Clearly we want to be able to deal with a mixture of these two types of tokens, together with tokens having both kinds of arguments (infix operators). This is where the problem of association arises, for which we recommended operator precedence. We add a state to the parser, thus:

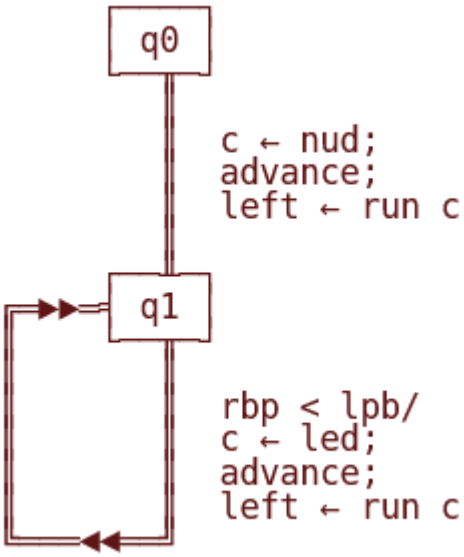


Starting in state `q0`, the parser interprets a token after advancing past that token, and then enters state `q1`. If a certain condition is satisfied, the parser returns to `q0` to process the next token; otherwise it halts and returns the value of `left` by default.

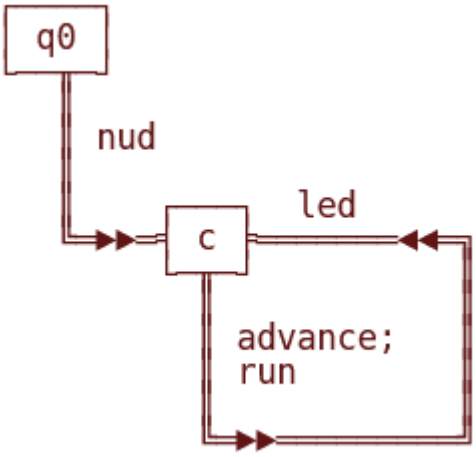
We shall also change our strategy when asking for a right-hand argument, making a recursive call of the parser itself rather than of the code of the next token. In making this call we supply the binding power associated with the desired argument, which we call the *rbp* (right binding power), whose value remains fixed as this incarnation of the parser runs. The *lbp* (left binding power) is a property of the current token in the input stream, and in general will change each time state `q1` is entered. The left binding power is the only property of the token not in its semantic code. To return to `q0` we require `rbp < lbp`. If this test fails, then by default the parser returns the last value of `left` to whoever called it, which corresponds to `A` getting `E` in `AEB` if `A` had called the parser that read `E`. If the test succeeds, the parser enters state `q0`, in which case `B` gets `E` instead.

Because of the possibility of there being several recursive calls of the parser running simultaneously, a stack of return addresses and right binding powers must be used. This stack plays essentially the same role as the stacks described explicitly in other parsing schemes.

We can embellish the parser a little by having the edge leaving `q1` return to `q1` rather than `q0`. This may appear wasteful since we have to repeat the `q0 - q1` code on the `q1 - q1` edge as well. However, this change allows us to take advantage of the distinction between `q0` and `q1`, namely that `left` is undefined in state `q0` and defined in `q1` -- that is, some expression precedes a token interpreted during the `q1 - q1` transition but not a token interpreted during the `q0 - q1` transition. We will call the code denoted by a token with (without) a preceding expression its *left (null) denotation* or *led* (nud). The machine becomes...



or by splitting transitions and using a stack instead of variables (the state equals the variable on the stack):



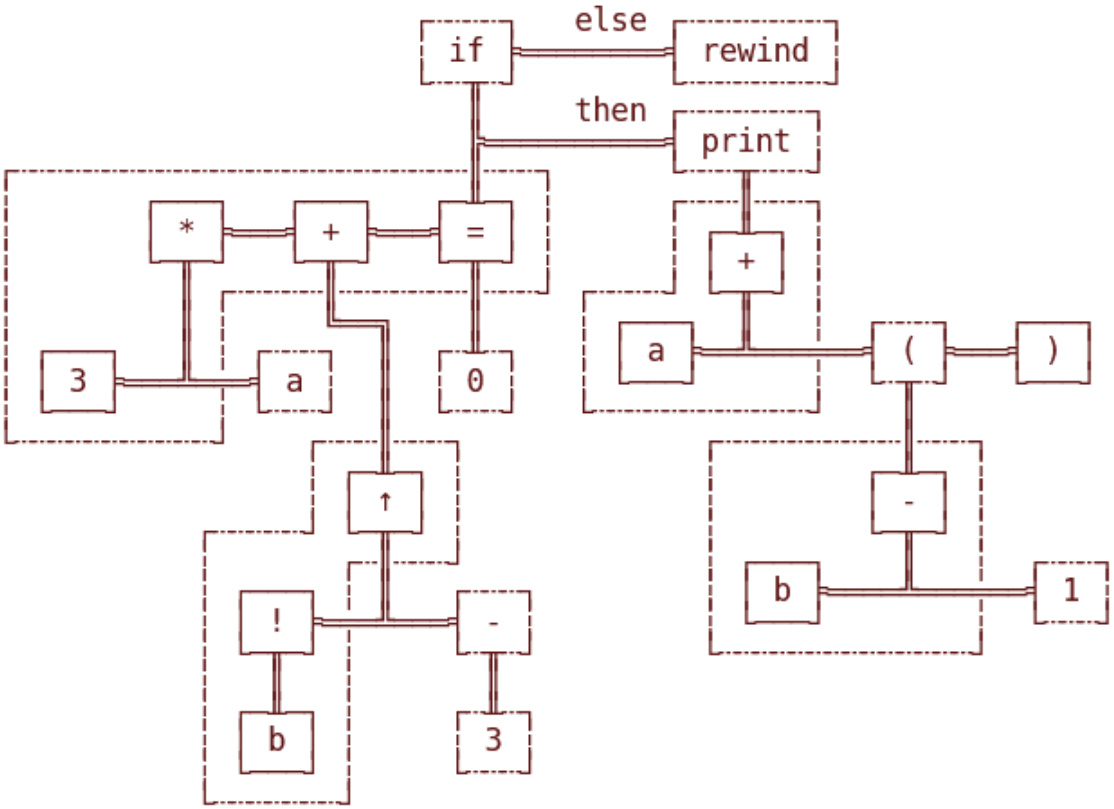
It now makes sense for a token to denote two different codes. For example, the nud of `-` denotes unary minus, and its led, binary minus. We may do the same for `/` (integer-to-semaphore conversion as in Algol 68, versus division), `(` (syntactic grouping, as in `a+(b×c)`, versus applications of variables or constants whose value is a function, as in `Y(F)`, `(λx.x²)(3)`, etc.), and `ε` (the empty string versus the membership relation).

A possibly more important role for nuds and leds is in error detection. If a token only has a nud and is given a left argument, or only has a led and is not given a left argument, or has neither, then non-existent semantic code is invoked, which can be arranged to result in the calling of an error routine.

So far we have assumed that semantic code optionally calls the parser once, and then returns the appropriate translation. One is at liberty to have more elaborate code, however, when the code can read the input (but not backspace it), request and use arbitrary amounts of storage, and carry out arbitrary computations in whatever language is available (for which an ideal choice is the language being defined). These capabilities give the approach the power of a Turing machine, to be used and abused by the language implementer as he sees fit. While one may object to all this power on the ground that obscure language descriptions can then be written, for practical purposes the same objection holds for BNF grammars, of which some quite obscure yet brief examples exist. In fact, the argument really runs the other way; the cooperative language implementer can use the extra power to produce more comprehensible implementations, as we shall see in section 4.

One use for this procedural capability is for the semantic code to read the delimiters and the arguments following them if any. Clearly any delimiter that might come directly after an argument should have a left binding power no greater than the binding power for that argument. For example, the nud of `if`, when encountered in the context `if a then b else c`, may call the parser for `a`, verify that `then` is present, advance, call the parser for `b`, test if `else` is present and if so then advance and call the parser a third time. (This resolves the "dangling else" in the usual way.) The nud of `(` will call the parser, and then simply check that `)` is present and advance the input. Delimiters of course may have multiple parents, and even semantic code, such as `|`, which might have a nud ('absolute value of' as in `|x|`'), and two parents, itself and '→' (where `a→b|c` is shorthand for `if a then b else c`). The ease with which mandatory and optional delimiters are dealt with constitutes one of the advantages of the top-down approach over the conventional methods for implementing operator precedence.

The parser's operation may perhaps be better understood graphically. Consider the example `if 3*a + b!↑-3 = 0 then print a + (b-1) else rewind`. We may exhibit the tree recovered by the parser from this expression as in the diagram below. The tokens encountered during one incarnation of the parser are enclosed in a dotted circle, and are connected via down-and-left links, while calls on the parser are connected to their caller by down-and-right links. Delimiters label the links of the expression they precede, if any. The no-op `(` is included, although it is not really a semantic object.



The major difference between the approach described here and the usual operator precedence scheme is that we have modified the Floyd operator precedence parser to work top-down, implementing the stack by means of recursion, a technique known as recursive descent. This would appear to be of no value if it is necessary to implement a stack anyway in order to deal with the recursion. However, the crucial property of recursive descent is that the stack entries are no longer just operators or operands, but the environments of the programs that called the parser recursively. When the programs are very simple, and only call the parser once, this environment gives us no more information than if we had semantic tokens themselves on the stack. When we consider more complicated sorts of constructions such as operators with various default parameters the technique becomes more interesting.

While the above account of the algorithm should be more or less self-explanatory, it may be worth while summarizing the properties of the algorithm a little more precisely.

Definition

An *expression* is a string `S` such that there exists a token `t` and an environment `E` in which if the parser is started with the input at the beginning of `St`, it will stop with the input at `t`, and return the *interpretation of `S` relative to* `E`.

Properties

1. When the semantic code of a token `t` is run, it begins with the input positioned just to the right of that token, and it returns the interpretation of an expression ending just before the final position of the input, and starting either at `t` if `t` is a nud, or if `t` is a led then at the beginning of the expression of which `left` was the interpretation when the code of `t` started.
2. When the parser returns the interpretation of an expression `S` relative to environment `E`, `S` is immediately followed by a token with `lbp` ≦ `rbp` in `E`.
3. The led of a token is called only if it immediately follows an expression whose interpretation the parser has assigned to `left`.
4. The `lbp` of a token whose led has just been called is greater than the `rbp` of the current environment.
5. Every expression is either returned by the parser or given to the following led via `left`.
6. A token used only as a nud does not need a left binding power.

These properties are the ones that make the algorithm useful. They are all straightforward to verify. Property (i) says that a semantic token pushes the input pointer off the right end of the expression whose tree it is the root. Properties (ii), (iv) and (v) together completely account for the two possible fates of the contents of `left`. Property (iii) guarantees that when the code of a led runs, it has its left hand argument interpreted for it in `left`. There is no guarantee that a nud is never preceded by an expression; instead, property (v) guards against losing an expression in `left` by calling a nud which does not know the expression is there. Property (vi) says that binding powers are only relevant when an argument is involved.

4. Examples

For the examples we shall assume that `lbp`, `nud` and `led` are really the functions `lbp(token)`, `nud(token)` and `led(token)`. To call the parser and simultaneously establish a value for `rbp` in the environment of the parser, we write `parse(rbp)`, passing `rbp` as a parameter. Then a `led` runs, its left hand argument's interpretation is the value of the variable `left`, which is local to the parser calling that `led`.

Tokens without an explicit `nud` are assumed to have for their `nud` the value of the variable `nonud`, and for their `led`, `noled`. Also the variable `self` will have as value the token whose code is missing when the error occurs.

In the language used for the semantic code, we use `a ← b` to define the value of expression `a` to be the value of expression `b` (not `b` itself); also, the value of `a ← b` is that of `b`. The value of an expression is itself unless it has been defined explicitly by assignment or implicitly by procedure definition; e.g., the value of `3` is `3`, of `1+1`, `2`. We write `'a'` to mean the expression `a` whose value is `a` itself, as distinct from the value of `a`, e.g. `'1+1'` must be evaluated twice to yield `2`.

A string `x` is written `"x"` this differs from `'x'` only in that `x` is now assumed to be a token, so that the value of `"1+1"` is the token `1+1`, which does not evaluate to `2` in general. To evaluate `a`, then `b`, returning the value of `b`, write `a;b`. If the value of `a` is wanted instead, write `a&b`. (These are for side-effects.) We write `check x` for `if token = x then advance else (print "missing"; print x; halt)`. Everything else should be self-explanatory. (Since this language is the one implemented in the second example, it will not hurt to see it defined and used during the first.)

We give specifications, using this approach, of an on-line theorem prover, and a fragment of a small general-purpose programming language. The theorem prover is to demonstrate that this approach is useful for other applications than just programming languages. The translator demonstrates the flexibility of the approach.

For the theorem prover's semantics, we assume that we have the following primitives available:

1. `generate`; this returns the bit string 0^k1^k and also doubles `k`, assumed `1` initially.
2. `boole(m,x,y)`: forms the bitwise boolean combination of strings `x` and `y`, where `m` is a string of four bits that specifies the combination in the obvious way (`1000 = and`, `1110 = or`, `1001 = eqv` etc). If one string is exhausted before the other, boole continues from the beginning of the exhausted string, cycling until both strings are exhausted simultaneously. Boole is not defined for strings of other than 0's and 1's.
3. `x isvalid`: a predicate that holds only when `x` is a string of all ones.

We shall use these primitives to write a program which will read a zero-th order proposition, parse it, determine the truth-table column for each subtree in the parse, and print "theorem" or ”non-theorem" when "?” is encountered at the end of the proposition, depending on whether the whole tree returns all ones.

The theorem prover is defined by evaluating the following expression.

```
nonud   ←  'if null led(self) then nud(self) ← generate else (
               print self;
               print "has no argument"
           )';

led("?") ← 'if left isvalid then print "theorem" else print "non-theorem";
               parse 1';

lbp("?") ← 1;

nud("(") ← 'parse 0 & check ")"';

lbp(")") ← 0;

led("→") ← 'boole("1101", left, parse 1)';

lbp("→") ← 2;

led("∨") ← 'boole("1110", left, parse 3)';

lbp("∨") ← 3;

led("∧") ← 'boole("1000", left, parse 4)';

lbp("∧") ← 4;

nud("~") ← 'boole("0101", parse 5, "0")'
```

To run the theorem prover, evaluate `k←1; parse 0`.

For example, we might have the following exchange:

```
(a→b)∧(b→c)→(a→c)? theorem
a? non-theorem
a∨~a? theorem
```

until we turn the machine off somehow.

The first definition of the program deals with new variables; which is anything without a prior meaning that needs a nud. The first new variable will get the constant `01` for its nud, the next `0011`, then `00001111`, etc. Next, `?` is defined to work as a delimiter; it responds to the value of its left argument (the truth-table column for parses a list of expressions delimited by the whole proposition), processes the next proposition by calling the parser, and returns the result to the next level parser. This parser then passes it to the next `?` as *its* left argument, and the process continues, without building up a stack of `?`s as `?` is left associative.

Next, `(` is defined to interpret and return an expression, skipping the following `)`. The remaining definitions should be self-explanatory. The reader interested in how this approach to theorem provers works is on his own as we are mainly concerned here with the way in which the definitions specify the syntax and semantics of the language.

The overhead of this approach is almost negligible. The parser spends possibly four machine cycles or so per token (not counting lexical analysis), and the semantics can be seen to do almost nothing; only when the strings get longer than a computer word need we expect any significant time to be spent by the logical operations. For this particular interpreter, this efficiency is irrelevant; however, for a general purpose interpreter, if we process the program so the lexical items become pointers into a symbol table, then the efficiency of interpreting the resulting string would be no worse than interpreting a tree using a tree-traversing algorithm as in LISP interpreters.

For the next example we describe a translator from the language used in the above to trees whose format is that of the internal representation of LISP s-expressions, an ideal intermediate language for most compilers.

In the example we focus on the versatility the procedural approach gives us, and the power to improve the descriptive capacity of the metalanguage we get from bootstrapping. Some of the verbosity of the theorem prover can be done away with in this way.

We present a subset of the definitions of tokens of the language L; all of them are defined in L, although in practice one would begin with a host language H (say the target language, here LISP) and write as many definitions in H as are sufficient to define the reast in L. We do not give the definitions of `nilfix`, `prefix`, `infix` or `infixr` here; however, they perform assignments to the appropriate objects; e.g. `(nilfix a b)` performs `nud(a)←'b'`, `(prefix a b c)` sets `bp←b` before perfomring `nud(a)←'c'`, `(infix a b c)` does the same as `(prefix a b c)` except that the led is defined instead and also `lbp(a)←b` is done, and `infixr` is like `infix` except that `bp←b-1` replaces `bp←b`. The variable `bp` is available for use for calling the parser when reading `c`. Also `(delim x)` does `lbp(x)←0`. The function `(a getlist b)` parses a list of expressions delimited by `a`s, parsing each one by calling `parse b`, and it returns a LISP list of the results.

The object is to translate, for example, `a+b` into `(PLUS a b)`, `a;b` into `(PROG2 a b)`, `a&b` into `(PROG2 nil a b)`, `-a` into `(MINUS a)`, `λx,y,...,z;a` into `(LAMBDA (x y ... z) a)`, etc. These target objects are LISP lists, so we will use `[` to build them; `[a, b,...,c]` translates into `(LIST a b ... c)`.

```
nilfix  right          ["PARSE", bp] $
infixr  ;          1   ["PROG2", left, right] $
infixr  &          1   ["PROG2", nil, left, right] $
prefix  is         1   ["LIST", right, 'left', ["PARSE", bp]] $
infix   $          1   [print eval left; right] $
prefix  delim      99  ["DELIM", token & advance] $
prefix  '          0   ["QUOTE", right & check "'"] $
delim   ' $
prefix  [          0   ("LIST" . "," getlist bp & check "]") $
delim   ] $
delim   , $
prefix  (          0   (right & check ")") $
delim   ) $
infix   (          25  (left . if token ≠ ")" then ("," getlist 0) &
                           check ")" else nil $
infix   getlist    25  is "GETLIST" $
prefix  if         2   ["COND", [right, check "then"; right]] @
                           (if token = "else" then (advance; [[right]])) $
delim   then $
delim   else $
nilfix  advance        ["ADVANCE"] $
prefix  check      25  ["CHECK", right] $
infix   ←          25  ["SETQ", left, parse(1)] $
prefix  λ          0   ["LAMBDA", ",", getlist 25 & check ";", right] $
prefix  +          20  right $
infix   +          20  is "PLUS" $
prefix  -          20  ["MINUS", right] $
infix   -          20  is "DIFFERENCE" $
infix   ×          21  is "TIMES" $
infix   ÷          21  is "QUOTIENT" $
infixr  ↑          22  is "EXPT" $
infixr  ↓          22  is "LOG" $
prefix  |          0   ["ABS", right & check "|"] $
delim   | $
infixr  @          14  is "APPEND" $
infixr  .          14  is "CONS" $
prefix  α          14  ["CAR", right] $
prefix  β          14  ["CDR", right] $
infix   ε          12  is "MEMBER" $
infix   =          10  is "EQUAL" $
infix   ≠          10  ["NOT", ["EQUAL", left, right]] $
infix   <          10  is "LESSP" $
infix   >          10  is "GREATERP"
```

and so on.

The reader may find some of the bootstrapping a little confusing. let us consider the definitions of `right` and `+`. The former is equivalent to `nud(right) ← '["PARSE", bp]'`.

The latter is equivalent to `nud(+) ← 'parse(20)' and led(+) ← '["PLUS", left, parse(20)]`, because when the nud of right is encountered while reading the definitions of `+`, it is evaluated by the parser in an environment where `bp` is `20` (assigned by prefix/infix).

It is worth noting how effectively we made use of the bootstrapping capability in defining `is`, which saved a considerable amount of typing. With more work, one could define even more exotic facilities. A useful one would be the ability to describe the argument structure of operators using regular expressions.

The `is` facility is more declarative than imperative in flavor, even though it is a program. This is an instance of the boundary between declaratives and imperatives becoming fuzzy. There do not appear to be any reliable ways of distinguishing the two in general.

5. Conclusions

We argued that BNF-oriented approaches to the writing of translators and interpreters were not enjoying the success one might wish for. We recommended lexical semantics, operator precedence and a flexible approach to dealing with arguments. We presented a trivial parsing algorithm for realizing this approach, and gave examples of an interpretive theorem prover and a translator based on this approach.

It is clear how this approach can be used by translator writers. The modularity of the approach also makes it ideal for implementing extensible languages. The triviality of the parser makes it easy to implement either in software or hardware, and efficient to operate. Attention was paid to some aspects of error detection, and it is clear that type checking and the like, though not exemplified in the above, can be handled in the semantic code. And there is no doubt that the procedural approach will allow us to do anything any other system could do, although conceivably not always as conveniently.

The system has so far found two practical applications. One is as the "front-end" for the SCRATCH-PAD system of Greismer and Jenks at IBM Yorktown Heights. The implementation was carried out by Fred Blair. The other application is the syntactic component of Project MAC's Mathlab system at MIT, MACSYMA, where this approach added to MACSYMA extension facilities not possible with the previous precedence parser used in MACSYMA. The implementer was Michael Genesreth.

6. Acknowledgments

I am indebted to a large number of people who have discussed some of the ideas in this paper with me. In particular I must thank Michael Fischer for supplying many valuable ideas relevant to the implementation, and for much programming help in defining and implementing CGOL, a pilot language initially used to break in and improve the system, but which we hope to develop further in the future as a desirable programming language for a large number of classes of users.

7. References

Aho, A.V. 1968. Indexed Grammars. JACM 15 4, 647-671

Chomsky, N. 1959. On certain formal properties of grammar. Information and Control, 2, 2, 137-167.

Fischer, M.J. 1968. Macros with Grammar-like Productions. Ph. D. Thesis, Harvard University.

Floyd, R.W. 1963. Syntactic Analysis and Operator Precedence. JACM 10, 3, 316-333.

Knuth, D.E. 1965. On the translation of languages from left to right, Information and Control, 8, 6, 607-639

Leavenworth, B.M. Syntax macros and extended translation. CACM 9, 11, 790-793. 1966.

Lewis, P.M., and R.E. Stearns. 1968. Syntax-directed transduction, JACM 15, 3, 465-488.

McKeeman, W.M., J.J. Horning and 57B. Wortman. 1970. A Compiler Generator. Prentice-Hall Inc. Englewood Cliffs, N.J.

Minsky, M.L. 1970. Form and Content in Computer Science. Turing Lecture, JACM 17, 2, 197-215.

Van Wijngaarden, A., B.J. Mailloux, J.E.L. Peck and C.H.A. Koster. 1969. Report on the Algorithmic Language ALGOL 68. Mathematisch Centrum, Amsterdam, MR 101.

Wirth, N. 1971. The programming language PASCAL. Acta Informatica, 1, 35-68.