Compiling and Analyzing Forth in Prolog

R. D. Dixon
David Hemmendinger
Department of Computer Science
Wright State University
Dayton, Ohio 45435

Abstract

A parser, a code generator, a semantic evaluator and an interpreter for Forth are written in Prolog. This is an investigative tool only and the semantic model includes a stack, input and output streams and segmented memory. The structure follows the usual Forth models but the somewhat more concise description emphasizes the exact manner in which Forth words control their environment. Compilers of this type together with Prolog compilers that generate Forth code make an interesting package that might be used to port both languages to a new machine, particularly a Forth machine. The abstraction of the Forth process in this manner may also make the environment more understandable to people inside and outside the Forth user community and thus allow both the wider acceptance of Forth and the generalization of Forth techniques to new languages.

Introduction

This paper contains the description of an abstraction of the Forth environment that is independent of Forth or any specific machine. Prolog has been chosen as the description language because it is concise, well understood, machine independent and executable.

The sections of the paper are independent and so may be read individually or in any order. The actual model is described in the last section. In the second section we discuss why a formal specification for Forth is useful and what the nature of a specification should be. In the third section we discuss why Prolog was chosen in this case. The fourth section contains the relationship of this paper to Forth standards.

Language Specifications

Syntax diagrams for Forth were presented at the 1982 Rochester Forth Conference by K. Moerman [MOE82]. Syntax diagrams and BNF grammars are useful because they give a short specification of the syntax that is allowable in the language. Many compilers are derived directly from the BNF grammar.

Any context-free syntactical specification of a programming language will accept or generate some strings or programs that have no reasonable interpretation by either a programmer or a compiler. Normally, the semantics of the language limits what is a valid program just as syntax does. Forth, because it uses a parameter stack to pass arguments, has a very loose syntax. Arguments may be generated a very long way, syntactically, from their use. Thus, the standard Forth compiler has no way at compile time to know if the proper arguments for a function will be available when it is called. Even under the assumption that "integer" is the only data type, Forth compilers still cannot determine the number of input and output arguments necessary for the proper operation of a colon definition.

A second deficiency of a BNF specification for Forth is that compiler implementations derived from BNF grammars usually contain the specification of the language syntax in their structure or in a processed table. A Forth compiler has most of its syntactic information distributed in the dictionary. It is this distributed nature of the compiler that permits the extensibility of the Forth language.

If the computer science community and management in general are to commit resources to projects involving Forth, then they must be convinced that there is a sound conceptual basis for the Forth environment. Although extensibility is generally viewed as a desirable characteristic, languages such as ALGOL 68 that have included it have not been commercially successful. Syntactic and semantic extensibility give the programmer the power to create new language features with unknown capabilities and perhaps bugs. One of the aims of the management of large computing projects is to restrict usage to certain well-known and understood constructs. Syntactic restrictions are used effectively but semantic restrictions are not practical. Every time a new layer of subroutines is introduced in a system, users of the higher layers are dealing with new semantic constructs whose meaning must be learned and whose reliability must be tested.

One of the things that makes Forth popular with its users is its syntactic extensibility; so it is desirable to specifically address reliability concerns in a manner other than restricting the use of extensibility. There are several ways in which that can be done:

1. Make the compilation and extension process better understood.
2. Have the user declare both the syntactic and semantic meaning of extensions.
3. Do a machine check of these declarations to verify consistency.
4. Eventually specify Forth in a way that is machine independent and precise and that can be tested as a specification.

Categorial grammars [ADJ35] are a much older method of syntax specification than the phrase structure grammars [CHO56] from which BNFs are derived. In categorial grammars each word has one or more categories to which it belongs. A category may be primitive (indivisible) or complex. Complex categories relate the word to the categories of adjacent words. Just as with phrase structure grammars, different category schemes yield different classes of languages. A simple category scheme leads to the class of context-free languages, the class into which most programming languages fall [BAR60].

A categorial grammar has a set of primitive categories, including a distinguished category, s, for accepted strings. The complex categories are of the form C1/C2, where C1 and C2 are categories. Categories combine according to a cancellation rule that is intended to suggest the cancellation of fractions:

C1/C2 combines with C2 to yield C1 (C1/C2 C2 → C1).

(Categorial grammars may also have other cancellation rules but we do not need them here.) As a simple example, the following category assignment permits the parsing of a fragment of Forth:

Forth Word	Category
`:`	(s/sc)/sym
`;`	sc
`dup`	sc/sc
`+`	sc/sc
`(any token)`	sym

The following steps yield the parse of the colon definition:

```
                 :            dog           dup           +        ;
step             |             |             |            |        |
                 |             |             |            |        |
0         (s/sc)/sym          sym
1                      s/sc                sc/sc        sc/sc       sc
2                      s/sc                sc/sc              sc
3                      s/sc                        sc
4                                   s
```

In an earlier paper [DIX86], the authors made a preliminary study of the application of categorial grammars to Forth. BNF grammars may be applied in different ways to obtain different styles of parsers and the same is true of categorial grammars. Both techniques lead to parsing automata. In the case of Forth, the compilers are usually written directly in terms of stack automata. We have chosen to use that model directly with categories, rather than numbers, on the stack. One should keep in mind that, using the techniques of [BAR60], there is a constructive procedure for the factorization of any context-free grammar onto the words of the language. This factorization can be used to form a word expert parser [RIE79]. In this system each word knows the syntactic roles it can play. Forth syntax alone is quite simple; however, the semantic interactions of words are quite complex. The nature of word expert parsers allows us to model the semantics as we parse. The semantic model provides a second stack that can be used in the definition of acceptable sentences. An automaton with two stacks can determine languages that are not context free [HOP69]. Semantic restrictions on a Forth-like language with typed data, reverse Polish operators and nested if-else-then structures produce a language which is not context free. The reason is that the image of the run-time stack and the nesting of the structures cannot be handled with a single stack.

Given this inherent complexity, even without extensibility, it is clearly a difficult job to describe Forth precisely. On the other hand, this complexity makes it all the more important that the description be done.

Prolog as the Specification Language

The pleasure of embedding other languages in Forth has a complementary delight: embedding Forth in other languages. Threaded languages have been embedded in other languages since their inception. The fig-FORTHs were embedded in assemblers or at least cross-compiled with assemblers. The ease of generating or transporting Forth with an assembler or with a cross-compiler written in Forth or another high-level language has been partly responsible for its wide use. This process is simple enough to be understood by users as well as by systems programmers.

C has been implemented on many machines under Unix and other similar operating systems which yield a uniform I/O structure. This has made popular the use of Forths written in C. The question of the efficiency of such techniques can be raised. Because most Forth programs are written at least one layer of subroutine above the kernel, the difference in the execution speeds of these implementations and the ordinary assembled implementation is not as great as one might expect, particularly if one optimizes the implementation of a few important operations.

This research has not been concerned with questions of efficiency or even with obtaining a complete implementation. The research had its roots in the 1985 Rochester Forth Conference. Discussions in the working groups suggested that some Forth products and, in particular, Forth machines might suffer because the Forth market, enthusiastic though it might be, is not large enough to support large capital investments. The goal here is to produce a workbench where present and future language features can be studied.

Prolog is a language that allows the user to state facts. In describing a language, it is convenient to be free of the yoke of specifying detailed flow. Prolog has a sophisticated pattern-matching facility that handles much of what constitutes a language description. Prolog also has list structures that, together with the first two features, allow the specification of the actions of the words in the dictionary in picture-book form.

The semantics of the Forth language subset given here are operational in the sense that they are described in terms of an implicit abstract stack machine, the actions of which are derived from the semantics of Prolog. As a result, this Prolog description of a subset of Forth, while nonprocedural, is also executable and so allows the confirmation of the designer's understanding of the specification. Finally, Prolog is concise: the total description with a small sample dictionary is only a few pages. This allows the designer to play and experiment without great effort or expense.

Forth Standards

At the 1986 Rochester Forth Conference, Hans Nieuwenhuyzen raised questions about the process by which Forth standards are determined and suggested that a machine-independent and highly transportable core should be developed and that other words should be gathered into groups that would be available for different users. Solntseff has taken steps toward the definition of a Forth abstract machine in [SOL82,83,84]. The authors have not been a part of the Forth standards efforts but those suggestions have guided this work, which we hope will support that movement. It is clear that standards that do not gain the acceptance of the Forth community cannot be of much help in gaining wider application of Forth tools.

Details of the Forth Model

The model presented here has several functions. First, an interpreter is modeled that provides for direct user interaction. A facility for compiling new definitions that are entered by the user is modeled. These compiled words are analyzed by the system and the analysis is displayed. The compiled words may also be executed. The model is based on several explicit structures:

1. The standard input stream **IStr** (Forth word tokens).
2. The argument stack **S** (typed values such as integers, addresses, categories).
3. Code strings **C** (references to Forth instructions).
4. A segmented memory model.
5. The standard output stream.
6. A flat dictionary for words.

The return stack is implicit and not available for manipulation. Values in the stack and memory all have type designations. The input/output streams and the code strings contain symbols. Forth words manipulate these structures in four states:

1. Interpret state.
2. Compile state.
3. Docol state.
4. Analyze state.

Dictionary Entries

The action of each word is defined by its dictionary entry. The dictionary entry consists of a **run** predicate which specifies the word's behavior in interpret, docol and analyze states. The **comp** predicate specifies the word's action in compile state. The state is determined by which of four predicate loops is controlling the processing: **interpret**, **compile**, **docol** or **analyze**. These predicates are defined in tail recursive fashion, which is the Prolog representation of a loop.

The **run** and **compile** predicates in the dictionary each have the following arguments:

1. The word name (e.g., '+').
2. The input stream action (e.g., **Istr=>Istr**).
3. The argument stack action (e.g., **[n(A),n(B) | S]=>[n(D) | S]**).
4. The code string action (e.g., **C=>C**).

Side effects and some semantic actions are represented to the right of the :- (e.g., **n(A)+n(B)=>n(D)**). Thus, the run statement for + in the dictionary is:

```
run('+',Istr=>Istr,[n(A),n(B) | S]=>[n(D) | S],C=>C) :- n(A)+n(B)=>n(D) .
```

The meaning of this statement is that + does not manipulate the input stream or the code string. It does have an effect on the argument stack which is to replace the top two numeric elements of the stack by a single element (**n** indicates numeric type). The symbol => is used to denote the before and after states of structures in arguments. As a predicate, as in **n(A)+n(B)=>n(D)**, it is read as "reduces to." This predicate is defined to implement the combinational semantics of Forth in terms of Prolog. The **comp** dictionary entry for + has similar structure but an additional argument for the name of the definition being compiled:

```
comp('+',Istr=>Istr,S=>S,['+' | C]=>C,Name).
```

In compile state + does not affect the input string or the argument stack but does issue the code for +. **C** is the unbound tail of the code string. The analog of this in an ordinary compiler is a pointer to the end of the issued code.

The structure of the dictionary implemented here is very limited. It is flat in the sense that no redefinition is permitted. Redefinition is an error that is not checked and violation can lead to surprising results. We did not work on this aspect of Forth because we feel that current implementations are not satisfactory and that this is a topic which requires a major effort by itself. Further, we only included a sample of Forth words so we could examine various facets of Forth without making the program unwieldy.

State Loops

The state loops have similar form to the interpret loop:

```
interpret([H | T],S=>S2) :-
    run(H,T=>T1,S=>S1, _),!,
    interpret(T1,S1=>S2).
```

The arguments for **interpret** are the input stream action and the argument-stack action.

Each state loop predicate has some additional clauses for special cases but most of the actions, even the means to change state, reside with the dictionary words. The **run** clause for **:** contains the interpret-to-compile state change:

```
run(':',[Name | Istr]=>R,S=>S,C=>C) :-
    compile(Istr=>R,[sc | S]=>S,Name).
```

Thus, if the **interpret** loop is running and **run** is called with first argument ':', then the **compile** loop will be entered. Actually, the **compile** loop runs as a subloop of **interpret** and control will not be returned until the entire definition is compiled.

Some other interesting concepts are introduced in this clause. The **Name** of the subroutine is removed from the input stream and remembered. The compilation symbol **sc** is placed on the argument stack. This stack will be used to check for the proper nesting of structures. The **sc** means we are looking for a **;** to close the definition. The **;** removes the **sc** from the stack and **compile** terminates when the stack returns to its original condition.

The interpretation of a word defined by a colon changes the state to "docol." The difference between **interpret** and **docol** mode is that **interpret** takes words to execute from the input stream whereas **docol** takes them from a code string. Each entry to docol simulates the stacking of a return address and the exit from **docol** unstacks it.

Immediate words are implemented and they temporarily change the state from compile to docol when they are encountered in a definition. This gives the user access to the compile-time stack but it does not have quite the same effect as immediate words in Forth where the immediate word has more power. One of the limitations of this model is that immediate words do not have access to the code string.

Memory Models

This paper strictly distinguishes between program and data memory areas. Programs or executable code are kept in named code strings. One can deal with references to this code symbolically or by using the type "cfa." **xxx** is the symbol that will appear in a compiled reference to the executable word `xxx`. If a literal reference to `xxx` is to be placed in the stack, then that reference is **cfa(xxx)**. No modification of these references is allowed. They are created by `'` and executed by `execute`.

Data segments are created by the `variable` and `create` words and attached to dictionary entries. The current model does not require allocation of segment space but allows the user to define the space by writing in it. Elements in this space are addressed using type "pfa." Thus, the first element of segment `yyy` has value **V** where **pfa(yyy,0,V)** is an entry in the Prolog data base and `yyy` is a data word in the Forth dictionary. Each element in a segment contains one typed data entry. The entry **pfa(yyy,1,n(8))** in the data base means the second entry in the `yyy` segment is the integer eight. The word `index` is used to modify an address. The mention of a data segment name generates a "pfa" type reference to the first element of the segment on the stack. The line `5 yyy index` leaves **pfa(yyy,5,_)** on the stack. Following this line by an `@` gets the value of the sixth element of the `yyy` segment and puts it on the stack.

Words with both code strings and data segments can be generated, as in Forth, by using the `create...does>` construct to make defining words which then can be used to create such words. A word `here` is defined which is set to the first element of any newly defined segment. This allows for the definition of `,` and, in general, helps with the initialization of data structures. The method of defining typed storage evades such things as byte addressing and word size. Those issues must be faced, of course, but we have chosen not to do it here.

Flow Control

One of the difficulties of producing machine-independent descriptions of run-time environments is the problem of local or relative branches. They simply know too much about the memory mapping. There is good reason to believe that as new computer architectures evolve, fast alternatives to this type of branch will become available. The solution we have chosen is to allow only branches to named routines. Thus, the definition

```
: xxx 0= if swap dup else + then * ;
```

is compiled as three routines as indicated by the following "code" predicate that expresses the name and a list of the code:

```
code(xxx0,[swap,dup,;S]).
code(xxx1,[+,;S]).
code(xxx,[0=,if_r,xxx0,xxx1,*,;S]).
```

The run-time behavior of the **if_r** word is given by the following two clauses:

```
run(if_r,IStr=>IStr,[b(true)|S]=>S,[Th,El|C]=>[Th|C]).
run(if_r,IStr=>IStr,[b(false)|S]=>S,[Th,El|C]=>[El|C]).
```

The expression **[b(true) | S]=>S** means that the first rule applies provided there is a Boolean value of "true" on the top of the argument stack. In that case it will be removed and the true alternative (e.g., **xxx0**) will be executed with the return adjusted to skip the execution of the false alternative (e.g., **xxx1**). The second rule applies if a "false" is on the stack.

Such flow control structures have a purely local context and thus simplify the design of a machine to execute them. The burden of the nonlocal action is all placed on the jump-to-subroutine instruction. As programming environments become more sophisticated, more of the executed code consists of subroutine calls and consequently there is a focus on the optimization of that instruction. In addition, cross-compiled code is easier to use in this format.

The loop structure implemented is the begin-while-do construct:

```
: yyy 0 begin aaa index @ dup 0= while drop 1 + repeat . ;
```

This word compiles as:

```
code(yyy0,[aaa,index,@,dup,0=,;S]).
code(yyy1,[drop,n(1),+,yyy0,;S]).
code(yyy,[n(0),yyy0,repeat_r,yyy1,.,;S]).
```

The interesting structure is **repeat_r,yyy1** whose execution is determined by the rules:

```
run(repeat_r,IStr=>IStr,[b(false) | S]=>S,[Body | C]=>C).
run(repeat_r,IStr=>IStr,[b(true) | S]=>S,
    [Body | C]=>[Body,repeat_r,Body | C]) :-
        not analyzing.
```

The first rule skips over the loop body and the included test if there is a "false" on the stack. The second states that if a "true" is on the stack, then the body is executed that includes the test. Next control is returned to the **repeat_r** instruction so that the result of the test can be evaluated. Again we have avoided any reference to addresses other than named routines. The **not analyzing** part of the second clause is there to prevent the analyze routine from looping, possibly forever, if there is insufficient data to force termination during the analysis.

Vectored execution to named routines is possible by placing a cfa on the stack and using `execute`.

Defining Words

The `create-does>` structure is modeled closely on the Forth construct. The part following the `does>` is compiled as a separate word so that an easy reference can be constructed in any instance of the defining word. Consider the definition:

```
: array create 0 , 0 does> index ;
```

This compiles as:

```
code(array1,[index,;S]).
code(array,[create_r(array1),n(0),',',n(0),',',;S]).
```

The statement `array zzz` creates the code

```
code(zzz,[dovar(zzz),array1,;S]).
```

This code will be executed each time `zzz` is executed. In addition, the execution of `array` on the name `zzz` initializes the first two values of the pfa of `zzz` to 0. This was done through `,`. After a variable `qqq` is created, we find that **pfa(here,0,pfa(qqq,0,_))** is in the Prolog data base. The system defines a variable **here** that will have access to this data and then defines `,` in terms of that variable.

The dictionary statements needed to implement defining words are:

```
comp(create,IStr=>IStr,[sc | S]=>[c(Dname),sc | S],
    [create_r(Dname) | C]=>C,N) :- gensym(N,Dname).
comp('does>',IStr=>[';' | Rem],[c(Dname) | S]=>S,C=>C,N) :-
    compile(IStr=>Rem,[sc | S]=>S,[ ]=>_,Dname).
run(create_r(Does),[Sym | IStr]=>IStr,S=>S,C=>C) :-
    effect((store(pfa(here,0,pfa(Sym,0,_))),
    add_def(Sym,[';S',Does,dovar(Sym)],comp))).
run(dovar(Sym),IStr=>IStr,S=>[pfa(Sym,0,_) | S],C=>C).
```

In these clauses **gensym** generates a new symbol as a name for the the `does>` part of a definition. The predicate **effect** is used to encapsulate those parts of run statements that have side effects. These side effects are disabled during analysis of definitions.

Semantic Analysis

Semantic analysis takes place during the compilation of definitions. A word-by-word analysis takes place as the compilation proceeds. A simulated stack is kept by the compile loop and if the stack contents can be matched to the input requirements of a word's compilation statement, then compilation can proceed. A more global analysis is done at the end of the definition and this analysis is used to display all the alternative meanings of a word. The Prolog predicates necessary to do that are:

```
    actions(Name,S,S1,Code) :-
        assert(analyzing),assert(trying),
        nl,write(Name),write(':'),
        analyze(Code,S=>S1,S).
    actions(_,_,_,_) :- retract(analyzing),nl.

    analyze([';S'],Stk=>Stk,S) :- nl,show(S=>Stk),retract(trying),fail.
    analyze([H | T],Stk=>Stk2,S) :-
        run(H,_,Stk=>Stk1,T=>T1),
        analyze(T1,Stk1=>Stk2,S).
    analyze([H | T],Stk=>_,_) :-
        retract(trying),
        nl,write('semantic error trying '),
        write(H),nl,show(Stk),nl,fail.

    act([ ],_) :- !.
    act([H | T],AS=>AS1) :-
        assert(analyzing),
        docol([H | T],AS=>AS1,IStr=>IStr1),
        retract(analyzing).
```

actions works at the end of a definition and calls **analyze**, which is a simulated form of **docol**. Note that the analyzing flag is used to shut off any side effects that the code being analyzed might have. Planned failure and Prolog backup are used to generate alternatives for `if` statements. Repeat statements are bypassed. **act** is the analysis done word-by-word so that overloaded operators can be selected at compile time. For called routines, the first acceptable alternative meaning is used. This was done to avoid the explosion of meanings for very high-level words.

Semantic Context Specifications

The first word in the definition of a colon word or any structure that results in a named definition such as an "if" alternative may be a comment containing a description of the stack inputs and outputs of that structure:

```
: ttt (n n -- b ) + 0= ;
```

Such a comment has no effect at run time but it can affect the compilation and analysis and thus the results if the definition contains overloaded operators. A small subcompiler was written that parses these comments:

```
extract(R=>R,S=>S,C=>C,D=>D) :- !.
extract([H|T]=>R,S=>S2,C=>C2,D=>D2) :-
    lcomp(H,S=>S1,C=>C1,D=>D1),!,
    extract(T=>R,S1=>S2,C1=>C2,D1=>D2).

lcomp('–',[dd|S]=>[rp|S],C=>C,D=>D).
lcomp(')',[rp|S]=>S,C=>C,D=>D).
lcomp(H,[dd|S]=>[dd|S],C=>[X|C],D=>D) :- type(H,X).
lcomp(H,[rp|S]=>[rp|S],C=>C,D=>[X|D]) :- type(H,X).
lcomp(_,_,_,_) :-nl,write('error in context spec!'),fail.
type(n,n(_)).
type(b,b(_)).
type(pfa,pfa(_,_,_)).
type(cfa,cfa(_)).
```

The arguments of **extract** are the input string action, the stack action, the input stack specification and the output stack specification.

Overloading of Operators

Many languages allow operators to be used for different things depending on the context. Smalltalk is one such language where this provision is a fundamental part of the language model. When the meaning must be identified at run time, then efficiency can suffer. In this model we allow compile-time overloading by explicit declaration. The statement **overload index onto** + means that if + is applied in a context where the arguments cannot be integers, then instead of reporting an error, the compiler will try to substitute `index`. `index` takes as arguments an integer and a pfa so that

```
variable uuu
: ppp uuu + ;
```

would cause the code statement

```
code(ppp,[uuu,index,;S]).
```

to be entered in the data base. The definitions

```
: mmm + ;
: nnn (n pfa -- pfa ) + ;
```

would result in the following code:

```
code(mmm,[+,;S]).
code(nnn,[index,;S]).
```

Only the input part of the context specification is actually used in this process.

Overloading of operators is only recognized at compile time. When using the interpreter, the actual operator must be named. The reason this was done was that the present system, with a few exceptions, makes no real use of types at run time. Thus, the actual implementation could be built with routines that treated everything as integers.

The Reduction of Arithmetic Operators

The left-hand side of the "run" entries for arithmetic operators in the dictionary instantiate a variable to the result. The right-hand side of the clause is a **=>** predicate that is the reduction operator, for example, **A+B=>D**. There are entries in the data base to define the behavior of this predicate:

```
X+Y=>Z :- integer(X),integer(Y),!,Z is X+Y.
Z+X=>X :- Z == 0,!.
X+Z=>X :- Z == 0,!.
X+Y=>X+Y.
```

At run time the first suffices. During analysis the other clauses give the analyzer the ability to make conclusions when all of the arguments are not instantiated.

A Session with the Interpreter

The session begins by loading a Prolog interpreter and consulting the Forth file. The command **startforth** causes the interpreter to run. First, the system is programmed to define `here` and `,` as user words since they are not in the predefined dictonary. Note that the analysis of `,` is printed. What is printed out is the before and after images of the stack in as much detail as the analyzer knows. They are printed in the Prolog list form, with the top of the stack to the left. Input and output lines are in different typefaces in the example which follows.

```
:- startforth.

,:
[A|B]=>B
OK
```

Next we define `not` which is not in the predefined dictionary:

```
: not if b false else b true then ;

not1:
A => [b(false)|A]

not2:
A => [b(true)|A]

not:
[b(true)|A] => [b(false)|A]
[b(false)|A] => [b(true)|A]

OK
```

Here the analyzer prints out the two alternatives that it finds for the semantics of the word. We can then use `not`:

```
5 0= not .stack

stack: [b(true)]
OK
```

The defining word `array`, which defines a one-dimensional array with the first two elements 0, is:

```
: array create 0 , 0 , does> index ;

array3:
[pfa(A,B,C),n(D)|E] => [pfa(A,B + D,F)|E]

array:
A => A
OK
```

Using this to define an instance `aa`, we examine some uses:

```
array aa
OK

decompile aa
[dovar(aa),array3,;S]
OK

0 aa @ .
0 OK

1 aa @ .
0 OK

2 aa @ .
run-time error: restarting
OK

2 aa 4 swap !
OK

2 aa @ .
4 OK
```

A little more work with the "if" statement yields:

```
: bb 0= if + then index ;

bb4:
[n(A),n(B)|C] => [n(A + B)|C]

bb5:
A => A

 compile error here: [index,;]
 stack found: [n(A + B)|C]

 OK

 : bb 0= if + else index then ;
```

```
bb6:
[n(A),n(B)|C] => [n(A + B)|C]

bb7:
[pfa(A,B,C),n(D)|E] => [pfa(A,B + D,F)|E]

bb:
[n(A),n(B),n(C)|D] => [n(B + C)|D]

[n(A),pfa(B,C,D),n(E)|F] => [pfa(B,C + E,G)|F]

OK
```

We use the context declaration to detect an error:

```
: ee (n n n -- n ) 0= if + else index then ;

ee8:
[n(A),n(B)|C] => [n(A + B)|C]

ee9:
[pfa(A,B,C),n(D)|E] => [pfa(A,B + D,F)|E]

ee:
[n(A),n(B),n(C)] => [n(B + C)]

semantic error trying ee9
[n(A),n(B)]

OK
```

An extended session with this Prolog implementation may lead to garbage collection problems so it is best to experiment in short batches.

References

[ADJ35] Adjukiewicz, K. 1936. Ueber die syntaktische konnexitaet. *Studia Philosophica* 1:1-27.

[BAR60] Bar-Hillel, Y.; Gaifman, C.; and Shamir, E. 1960. On categorial and phrase structure grammars. *Bulletin of the research council of Israel 9f*, pp. 1-16.

[CHO56] Chomsky, N. 1956. Three models for the description of language. *IRE Transactions on Information Theory IT-2*, pp. 113-24.

[DIX86] Dixon, R., and Hemmendinger, D. 1986. A more thorough syntax checker for FORTH. *J. Forth Appl. and Res.* 4(2):245-48.

[HOP69] Hopcroft, J., and Ullman, J. 1969. *Formal languages and their relation to automata.* Reading, MA: Addison-Wesley.

[MOE82] Moerman, K. 1982. FORTH syntax diagrams. In *1982 Rochester Forth conference on data bases and process control*, pp. 263-66. Rochester, NY: Institute for Applied Forth Research.

[REI79] Rieger, C., and Small, S. 1979. Word expert parsing. *Proceedings of the sixth international joint conference on artificial intelligence,* pp. 723-28.

[SOL82] Solntseff, N. 1982. An abstract machine for the Forth system. In *1982 Rochester Forth conference on data bases and process control*, pp. 145-55. Rochester, NY: Institute for Applied Forth Research.

[SOL83] ------. 1983. An instruction-set architecture for abstract Forth machines. In *1983 Rochester Forth applications conference*, pp. 175-83. Rochester, NY: Institute for Applied Forth Research.

[SOL84] Solntseff, N., and Russell, J. W. 1984. An approach to a machine-independent Forth model. In *1984 Rochester Forth conference*, pp. 121-39. Rochester, NY: Institute for Applied Forth Research.

Manuscript received November 1986.

Robert D. Dixon received his doctorate from Ohio State University in 1962. He is currently a professor of computer science at Wright State University. His interests include real-time systems and hardware and software support systems for natural language interfaces.

David Hemmendinger received his Ph.D. in philosophy from Yale University in 1973. He has been in the Wright State Computer Science Department since 1982, and works on logic programming, concurrency, and the use of formal tools to produce executable specifications of programming language semantics.

Appendix

```
% Forth interpreter
% Version 1.1     6 February 1987
% Runtime interpreter: Arity Prolog v 4.0, on IBM PC
%   Note: The Arity-specific routines are:
%         system calls (dir, cd, shell, ed, edit)
%         abolish/1   (other prologs have abolish/2)
%         ctr_set/2, ctr_inc/2  (global counters)
%         read_line, list_text  (string predicates)
%         gc(full)  (garbage collector)
%   All but 'abolish' and 'gc' appear only in the utilities section

% The code is in three parts: dictionary, execution states, utilities
% To execute: invoke 'startforth' from the top level of Prolog.
% To terminate:  enter 'bye' to the Forth interpreter. Note that
%                the intepreter is case-sensitive and that predefined
%                words follow the Prolog lower-case convention.
% To restart:    invoke 'forth' from the top level of Prolog.

% Forth dictionary: 'run' and 'comp' actions for each word.

:- op(900,xfx,'=>').
:- op(700,fx,ed).
:- op(700,fx,cd).

word(drop).    word(dup).        word('.').       word('+').       word('*').
word('-').     word('0=').       word('!').       word('@').       word(swap).
word(n(_) ).   word(':').        word(here).      word(index).     word('' '').
word(':i').    word(constant).   word(execute).   word(variable).  word(create).

type(n,n(_) ).
type(b,b(_) ).
type(pfa,pfa(_,_,_) ).
type(cfa,cfa(_) ).

run(drop,IStr=>IStr,[X|S]=>S,C=>C).
run(dup,IStr=>IStr,[X|S]=>[X,X|S],C=>C).
run(swap,IStr=>IStr,[X,Y|S]=>[Y,X|S],C=>C).
run('.',IStr=>IStr,[n(A)|S]=>S,C=>C) :-
              effect((write(A),write(' ') ) ),!.
run('.',IStr=>IStr,[A|S]=>S,C=>C) :-
              type(_,A),effect((write(A),write(' ') ) ).

run('+',IStr=>IStr,[n(A),n(B)|S]=>[n(D)|S],C=>C) :- A+B=>D.
run('-',IStr=>IStr,[n(A),n(B)|S]=>[n(D)|S],C=>C) :- B-A=>D.
run('*',IStr=>IStr,[n(A),n(B)|S]=>[n(D)|S],C=>C) :- A*B=>D.
run('0=',IStr=>IStr,[n(A)|S]=>[b(B)|S],C=>C) :- (A==0)=>B.
run('' '',[Sym|IStr]=>IStr,S=>[cfa(Sym)|S],C=>C) :-
    (word(Sym) ; code(Sym,_) ).
run(execute,IStr=>IStr1,[cfa(Sym)|S]=>S1,C=>C) :-
    effect(run(Sym,IStr=>IStr1,S=>S1,C=>C) ).

run(':',[Name|IStr]=>Rem,S=>S,C=>C) :-
    compile(IStr=>Rem,[sc|S]=>S,Name,comp),!.
```

```
run(':',IStr=>[ ],S=>S,C=>C).    % recover after compile error
run(':i',[Name|IStr]=>Rem,S=>S,C=>C) :-
     compile(IStr=>Rem,[sc_i|S]=>S,Name,immed),!.
run(':i',IStr=>[ ],S=>S,C=>C).

run(if_r,IStr=>IStr,[b(true)|S]=>S,[Th,El|C]=>[Th|C]).
run(if_r,IStr=>IStr,[b(false)|S]=>S,[Th,El|C]=>[El|C]).

run(repeat_r,IStr=>IStr,[b(false)|S]=>S,[Body|C]=>C).
run(repeat_r,IStr=>IStr,[b(true)|S]=>S,[Body|C]=>[Body,repeat_r,Body|C]) :-
     not analyzing.

run(variable,[Sym|IStr]=>IStr,S=>S,C=>C) :-
     store(pfa(here,0,pfa(Sym,0,_) ) ),
     add_def(Sym,[dovar(Sym),';S'],comp).

run('!',IStr=>IStr,[pfa(N,D,_),V|S]=>S,C=>C) :- effect(store(pfa(N,D,V) ) ).
run('@',IStr=>IStr,[pfa(N,D,V)|S]=>[V|S],C=>C) :- effect(pfa(N,D,V) ).
run(index,IStr=>IStr,[pfa(N,A,_),n(B)|S]=>[pfa(N,D,_)|S],C=>C) :- A+B=>D.
run(constant,[Sym|IStr]=>IStr,[A|S]=>S,C=>C) :- add_const(Sym,A).
run(create,[Sym|IStr]=>IStr,S=>S,C=>C) :-
          run(variable,[Sym|IStr]>IStr,S=>S,C=>C).
run(create_r(Does),[Sym|IStr]=>IStr,S=>S,C=>C) :-
     effect((store(pfa(here,0,pfa(Sym,0,_) ) ),
     add_def(Sym,[dovar(Sym),Does,';S'],comp) ) ).
run(dovar(Sym),IStr=>IStr,S=>[pfa(Sym,0,_)|S],C=>C).
run(overload,[N1,onto,N2|IStr]=>IStr,S=>S,C=>C) :-
     assertz((comp(N2,A,B,D,E) :- comp(N1,A,B,D,E) ) ).
run(Type,[Sym|IStr]=>IStr,S=>[Tval|S],C=>C) :-
     type(Type,Tval),Tval =.. [Type,Sym].
run(Type,IStr=>IStr,S=>[Type|S],C=>C) :- type(_,Type).

% debugging and system calls:
run('.stack',IStr=>IStr,S=>S,C=>C) :-      % show stack
     effect((nl,write('stack: '),write(S),nl) ).
% !! goal arg1 . .arg_n       executes the prolog call: goal(arg1,..,arg_n)
run('!!',IStr=>[ ],S=>S,C=>C) :- Goal =.. IStr,call(Goal).
run(stat,IStr=>IStr,S=>S,C=>C) :- statistics.
run(ed,[Name]=>[ ],S=>S,C=>C) :- ed Name.
run(decompile,[Name|IStr]=>IStr,S=>S,C=>C):-
          code(Name,Code),nl,write(Code),nl.

% comp part of the dictionary
comp(drop,IStr=>IStr,S=>S,[drop|C]=>C,N).
comp(dup,IStr=>IStr,S=>S,[dup|C]=>C,N).
comp(swap,IStr=>IStr,S=>S,[swap|C]=>C,N).
comp('+',IStr=>IStr,S=>S,['+'|C]=>C,N).
comp('-',IStr=>IStr,S=>S,['-'|C]=>C,N).
comp('*',IStr=>IStr,S=>S,['*'|C]=>C,N).
comp('.',IStr=>IStr,S=>S,['.'|C]=>C,N).
comp('!',IStr=>IStr,S=>S,['!'|C]=>C,N).
comp('@',IStr=>IStr,S=>S,['@'|C]=>C,N).
comp(index,IStr=>IStr,S=>S,[index|C]=>C,N).
comp('0=',IStr=>IStr,S=>S,['0='|C]=>C,N).
comp('' '',[Sym|IStr]=>IStr,S=>S,[cfa(Sym)|C]=>C,N).
comp(n(V),IStr=>IStr,S=>S,[n(V)|C]=>C,N).
comp(';',IStr=>IStr,[sc|S]=>S,[';S']=>[ ],N).
comp(';i',IStr=>IStr,[sc_i|S]=>S,[';S']=>[ ],N).
comp(execute,IStr=>IStr,S=>S,[execute|C]=>C,N).

comp(if,IStr=>Rem,S=>S,[if_r,N_then,N_else|C]=>C,N) :-
    gensym(N,N_then),gensym(N,N_else),
    compile(IStr=>IStr1,[t|S]=>[t1|S],N_then,comp),
    compile(IStr1=>Rem,[t1|S]=>S,N_else,comp).
comp(else,IStr=>IStr,[t|S]=>[t1|S],[';S']=>[ ],N).
comp(then,IStr=>IStr,[t1|S]=>S,[';S']=>[ ],N).
comp(then,IStr=>[then|IStr],[t|S]=>[t1|S],[';S']=>[ ],N).

comp(begin,IStr=>Rem,S=>S,[Test,repeat_r,Body|C]=>C,N) :-
    gensym(N,Test),gensym(N,Body),
    compile(IStr=>IStr1,[w|S]=>[w1(Test)|S],Test,comp),
    compile(IStr1=>Rem,[w1(Test)|S]=>S,Body,comp).
comp(while,IStr=>IStr,[w|S]=>[w1(_)|S],[';S']=>[ ],N).
comp(repeat,IStr=>IStr,[w1(Test)|S]=>S,[Test,';S']=>[ ],N).

comp(create,IStr=>IStr,[sc|S]=>[c(Dname),sc|S],[create_r(Dname)|C]=>C,N) :-
    gensym(N,Dname).
comp('does>',IStr=>[';'|Rem],[c(Dname)|S]=>S,C=>C,N) :-
    compile(IStr=>Rem,[sc|S]=>S,Dname,comp).

comp(Type,[Sym|IStr]=>IStr,S=>S,[Tval|C]=>C,N) :-
    type(Type,Tval), Tval =.. [Type,Sym].

% Execution states: interpret, compile, analyze, docol

forth :- interpret([ ],[ ]).
forth :- gc(full),write('run-time error: restarting'),nl,!,forth.

startforth :- forth_init(L),interpret(L,[ ]),forth.

forth_init([variable,here,':',',',here,'@','!',
        n(1),here,'@',index,here,'!',';',bye]).

interpret([ ],Stk) :- gc(full),write('OK'),nl,getline(L),!,interpret(L,Stk).
interpret([bye],_) :- !.
interpret([H|T],S) :-
    run(H,T=>T1,S=>S1,_),!,
    interpret(T1,S1).
interpret([H|T],Stk) :-
    not word(H),not code(H,_),
    write('unknown word here: '),write([H|T]),nl,!,
    interpret([ ],Stk).

docol([ ],S=>S,IStr=>IStr) :- !.      % for use by 'act' only
docol([';S'],S=>S,IStr=>IStr) :- !.
docol([H|T],S=>S2,IStr=>IStr2) :-
    run(H,IStr=>IStr1,S=>S1,T=>T1),!,
    docol(T1,S1=>S2,IStr1=>IStr2).

add_const(Sym,V) :- asserta(run(Sym,IStr=>IStr,S=>[V|S],C=>C) ),
                    asserta(comp(Sym,IStr=>IStr,S=>S,C=>[Sym|C],N) ).

store(pfa(N,D,V) ) :- retract(pfa(N,D,_) ),!,asserta(pfa(N,D,V) ).
store(pfa(N,D,V) ) :- asserta(pfa(N,D,V) ).

effect(_Effect) :- analyzing,!.
effect(Effect) :- call(Effect).

compile(['('|T]=>Rem,Ss,Name,Type) :- !,
    extract(T=>R,[dd]=>[ ],[ ]=>AS_before,[ ]=>AS_after),
    compile1(R=>Rem,Ss,Code,AS_before=>AS_after,AS_before,Name),
    add_def(Name,Code,Type),
    actions(Name,AS_before,AS_after,Code).
compile(IStrs,Ss,Name,Type) :-
    compile1(IStrs,Ss,Code,_,AS,Name),!,
    add_def(Name,Code,Type),
    actions(Name,AS,_,Code).
compile1(Rem=>Rem,S=>S,_,_,_,_).
compile1([ ]=>Rem,Stacks,Code,AStks,AS,Name) :- getline(L),!,
    compile1(L=>Rem,Stacks,Code,AStks,AS,Name).
compile1([H|T]=>Rem,S=>S2,C,AS=>AS2,AS3,Name) :-
    comp(H,T=>T1,S=>S1,C=>C1,Name),
    closed(C,Code),
    act(Code,AS=>AS1),!,
    compile1(T1=>Rem,S1=>S2,C1,AS1=>AS2,AS3,Name).
compile1([H|T]=>[ ],_,_,AS=>_,_,_) :-
    nl,write('compile error here: '),write([H|T]),nl,
    write('stack found: '),show(AS),nl,fail.

closed(C,[ ]) :- var(C),!.
closed([ ],[ ]).
closed([H|T],[H|Code]) :- closed(T,Code).

add_def(Name,Code,comp) :-
    add_run_code(Name,Code),
    asserta(comp(Name,IStr=>IStr,S=>S,[Name|C]=>C,_) ).
```

```
add_def(Name,Code,immed) :-
     add_run_code(Name,Code),
     asserta((comp(Name,IStr=>IStr1,S=>S1,C=>C,_):-
            run(Name,IStr=>IStr1,S=>S1,C=>C1) ) ).

add_run_code(Name,Code) :-
     asserta(code(Name,Code) ),
     asserta((run(Name,IStr=>IStr1,S=>S1,C=>C) :-
                 docol(Code,S=>S1,IStr=>IStr1) ) ).

actions(Name,S,S1,Code) :-
     assert(analyzing),
     nl,write(Name),write(':'),
     analyze(Code,S=>S1,S).
actions(_,_,_,_) :- retract(analyzing),nl.

analyze([';S'],Stk=>Stk,S) :- nl,show(S=>Stk),fail.
analyze([if_r|T],[b(B)|Stk]=>Stk2,S) :- !,
     run(if_r,_,[b(B)|Stk]=>Stk1,T=>T1),
     analyze(T1,Stk1=>Stk2,S).
analyze([H|T],Stk=>Stk2,S) :- H \== if_r,
     run(H,_,Stk=>Stk1,T=>T1),!,
     analyze(T1,Stk1=>Stk2,S).
analyze([H,H1|T],Stk=>_,_) :-
     nl,write('semantic error trying '),
     write(H),nl,show(Stk),nl,fail.

act([ ],_) :- !.
act([if_r,Th,_],[b(true)|AS]=>AS1) :- act([Th],AS=>AS1).
act([if_r,_,El],[b(false)|AS]=>AS1) :- act([El],AS=>AS1).
act([H|T],AS=>AS1) :- H \== if_r,assert(analyzing),
                      docol([H|T],AS=>AS1,IStr=>IStr1),
                      retract(analyzing).
act(_,_) :- retract(analyzing),fail.

% Reduction operations
X+Y=>Z :- integer(X),integer(Y),!,Z is X+Y.
Z+X=>X :- Z == 0,!.
X+Z=>X :- Z == 0,!.
X+Y=>X+Y.
X*Y=>Z :- integer(X),integer(Y),!,Z is X*Y.
Z*_X=>0 :- Z == 0,!.
_X*Z=>0 :- Z == 0,!.
U*X=>X :- U == 1,!.
X*U=>X :- U == 1,!.
X*Y=>X*Y.
X-Y=>Z :- integer(X),integer(Y),!,Z is X-Y.
X-Z=>X :- Z == 0,!.
X-Y=>0 :- X == Y,!.
X-Y=>X-Y.
```

```
(X==0)=>true :- X == 0,!.
(X==0)=>false :- integer(X),X \== 0,!.
(_X==0)=>_B :- analyzing.

% subcompiler for context declarations: (t1 .. tn--r1 .. rm )
% where t's and r's are type names, and spaces are essential.
extract(R=>R,S=>S,Before=>Before,After=>After) :- !.
extract([H|T]=>R,S=>S2,Bef=>Bef2,Aft=>Aft2) :-
          lcomp(H,S=>S1,Bef=>Bef1,Aft=>Aft1),!,
          extract(T=>R,S1=>S2,Bef1=>Bef2,Aft1=>Aft2).

lcomp('--',[dd|S]=>[rp|S],C=>C,D=>D).
lcomp(')',[rp|S]=>S,C=>C,D=>D).
lcomp(H,[dd|S]=>[dd|S],C=>[X|C],D=>D) :- type(H,X).
lcomp(H,[rp|S]=>[rp|S],C=>C,D=>[X|D]) :- type(H,X).
lcomp(_,_,_,_) :-nl,write('error in context spec!'),fail.

% utilities.

dir :- shell(dir).
cd(Dir) :- chdir(Dir).
ed(File) :- edit(File),cls.

apnd([ ],X,X).
apnd([H|T],X,[H|Y]) :- apnd(T,X,Y).

getln(Line) :- read_line(0,L),list_text(Line,L).

skipbl([32|T],Result) :- !,skipbl(T,Result).
skipbl([9|T],Result) :- !,skipbl(T,Result).
skipbl(Result,Result).

getword([ ],[ ],[ ]) :- !.
getword([32|T],[ ],T) :- !.
getword([9|T],[ ],T) :- !.
getword([C|T],[C|Ws],Rest) :- getword(T,Ws,Rest).

getwords([ ],[ ]) :- !.
getwords([N1|More],L) :-  getword(L,W,Rest),name(N,W),
                          skipbl(Rest,R1),wrap(N,N1),getwords(More,R1).
wrap(N,n(N) ) :- integer(N),!.
wrap(N,N).

getline(Ws) :- getln(L),skipbl(L,L1),getwords(Ws,L1),!.

abolish(C,n(Arity) ) :- abolish(C/Arity).

gensym(Name,N1) :-        name(Name,L1),ctr_inc(0,N),name(N,L2),
                          apnd(L1,L2,L3),name(N1,L3).
:- ctr_set(0,1).          % initialize counter #0, for gensym
```

```
% for writing a term with variables temporarily named A, B,...G.

bindvars(X,[ ],[ ]) :- !.
bindvars(VH,[VH|VT],VT).
bindvars(Atom,VL,VL) :- atomic(Atom),!.
bindvars([H|T],VL,VL2) :- !,bindvars(H,VL,VL1),bindvars(T,VL1,VL2).
bindvars(Term,VL,VL1) :- Term =.. [F|Args],bindvars(Args,VL,VL1).

show(Term) :-
                not not (bindvars(Term,['A','B','C','D','E','F','G'],_),
                  write(Term),nl).
```