# Occam's Razor: The Cutting Edge of Parser Technology

Jonathan P. Bowen [*]   Peter T. Breuer [†]

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road, Oxford OX1 3QD, UK.
Tel: +44-865-272574.   Fax: +44-865-273839.
Email: `Jonathan.Bowen@comlab.ox.ac.uk`
`Peter.Breuer@comlab.ox.ac.uk`

**Abstract**

*Yacc* is well established in the compiler-compiler field, but is beginning to show its age. Issues which were important when hardware resources were more scarce are now less critical. *Precc* is a new compiler-compiler tool that is much more versatile than *yacc*, whilst retaining efficiency of operation on modern computers. It copes with the context-dependent BNF grammar descriptions and higher order meta constructions that are naturally encountered in semi-formal concrete syntax specifications, building fast and efficient infinite-lookahead tools in the form of ANSI-compliant C code. This paper provides a demonstration of this state-of-the-art compiler-compiler technology using the programming language occam as an example. The parsing of occam is particularly difficult compared to some programming languages since the indentation is an integral part of the language. However the *precc* tool allows a natural implementation of an occam parser that follows the syntax very closely.

**Keywords:** Compiler-compiler, parsing, language grammars, occam, infinite lookahead, software tools.

**William Occam** c. 1280–1349

*Entia non sunt multiplicanda praeter necessitatatem.*

No more things should be presumed to exist than are absolutely necessary.

'Occam's Razor'. Ancient philosophical principle, often attributed to Occam, but used by many early thinkers [15].

[*]Funded by the UK Information Engineering Directorate SAFEMOS project (IED3/1/1036).

[†]Funded by the European ESPRIT II REDO project (2487). Currently a Visiting Research Fellow at British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, UK.

## 1 Existing technology

The ubiquitous *yacc* and *lex* tools [12] have dominated compiler-compiler technology for the past decade or more. *Yacc* uses LALR(1) parsing, which has been understood and promulgated by standard textbooks for some time (e.g., [1]), primarily because it constructs a finite state automaton, and thus the run-time loading can be delimited precisely. This used to be of great importance when computers generally were limited in memory, but nowadays it is much less important, and more negative aspects of the *yacc* technology have started to become increasingly significant instead.

Firstly, both *lex* and *yacc*-generated C code is monolithic, and a large language description, like that for COBOL, with over two hundred keywords, generates such a large virtual automaton that swapping problems are common at runtime even on modern workstations. This can result in a poor perceived performance for the application to which the parser is the front end, no matter how good the application code itself may be.

Moreover, the C compiler itself often has problems with the generated code. The large lexer and parser routines generated by *lex* and *yacc* make for slow compilation and therefore a long turn-round time when it comes to debugging or altering a specification. Everything has to be recompiled when just a single change is needed, and this can be a source of intense frustration for the software engineer seeking to use a specification-driven utility in order to cut development or maintenance time. On the unloaded HP 9000 series workstation at which this sentence is being typed, for example, *lex* takes 35 s to compile a very abbreviated 163-line lexical specification for COBOL 74 into 2,700 lines of C code. A more complete 300-line specification takes two minutes to generate 5,000 lines of C code. The full specification for the language about twice this size. The C compiler took 40 s to compile the small code and 73 s to compile the larger code, giving a 110 Kbyte object module, and the figures grow super-linearly with size. So a wait of five to ten minutes between making a change to the lexer and testing it can be expected, and there is no real possibility of using a debugging aid to help the diagnosis and testing because the generated code changes each time, and is famously opaque anyway. The statistics for *yacc* are worse because the scripts are even bigger.

In fact, in terms of the specification alone, *yacc* does provide a fairly flexible way of handling language grammars, with at least a reasonable degree of efficiency. However it is not ideal for, and indeed cannot cater for, all languages. This is because it allows for only a single-token lookahead at runtime in order to decide between alternatives. In a sense, therefore, its runtime semantics are ill-matched with its specification language, because the BNF (which is what it uses) has no such restriction commonly associated with it. Moreover, it implements only a simple subset of the standard BNF description formats, so all BNF specifications have to be unfolded out into their basic components before they can be presented as a *yacc* script. This results in obscure specifications which are difficult to maintain, and is at least partly responsible for the excessive size of the virtual automaton, since much repetition is necessary.

## 2 Introducing the precc tool

*Precc* [4] offers a very flexible tool, providing infinite lookahead and handling two-level or van Wijngaarden grammars [7, 9, 18] with ease. These grammars are very powerful and include include meta-variables and hyper-rules for the generation of infinite numbers of productions. The concept of a 'comma separated list of *x*', for example, can be defined in *precc*, as

$$@\ comma\_separated\_list\_of(x) \quad = \quad x \quad \{\ \langle\texttt{','}\rangle\ x\ \}\star$$

(*x* followed by arbitrarily many comma-then-*x* pairs) and different *x*'s can be supplied as *parameters* in different parts of the specification, but for *yacc*, the construction must be expanded out in terms of the basic constructs at every point in the specification where it is needed, substituting the required name for *x*, and using only the alternation and sequential constructors:

```
phooey  :  x
        |  x ',' phooey
;
```

This *yacc* specification may also generate a 'shift/reduce' conflict report at compile time, if any other production rule contains '*phooey* ','' because *yacc* does not know whether to begin looking for an *x* to follow the comma (shift for more *phooey*) or to jump with what it already has into the other production (reduce). Shift/reduce reports are extremely confusing to software engineers because

1. they refer to the virtual automaton, not the grammar specification script itself, and therefore rely on an understanding of *yacc* semantics which the specification language interface ought to encourage users to forget, and
2. the report is due to the existence of a second production rule, and therefore is generated by the context, not, in the example above, the *phooey* rule itself.

The parts of a *yacc* script may be separately correct, and still fail when combined together. In contrast, *precc* specifications are *declarative* and *referentially transparent*: i.e., it is possible to substitute any *precc* parser name by its definition anywhere in a script, and obtain the same semantics. The semantics of each parser definition are independent of their context and a *precc* parser definition can be included in any script, provided it is self-consistent. What happens to the shift/reduce report? *Precc* always shifts (looks for more data), and later backtracks if necessary. *Yacc*'s error reports are a function of *yacc*'s one-token lookahead implementation of the BNF, and since *precc* has no such limitation, there are no conflicts between specification language and implementation semantics to be reported.

One problem often associated with tools which promise to provide extra flexibility is that the designers fail to ensure that the tool is still efficient enough to be useful in practice. For example, the OBJ3 equational logic tool [8] provides parsing of arbitrary mixfix notation (with infinite lookahead). Whilst this provides great flexibility, it becomes rather slow for the parsing of large input programs; so much so that it becomes worthwhile to have a separate more efficient parser for more easily parsable languages. But *precc*, like *yacc*, maintains efficiency by translating the parser description into C code. Moreover, it makes use of the premise that modern C compilers make function calls fast and efficient in order to make the C call stack do much of its work. The *precc* design assumes that function calls are efficient in C, and that many small compact functions are better than a single large one. Happily this conjecture appears to be correct, because *precc* runs quickly in practice. *Precc* compiles a similar description to the COBOL 74 lexical specification, consisting of 162 lines, into 1,500 lines of C code in 1.1 s on the HP workstation. Its own C code takes no longer to generate.

The C code takes 22 s to compile, as might be expected, but the most significant benefit from using *precc* is modularity. Both *precc* specifications and generated code can be divided up into modules in any way that is desired, compiled separately and linked together incrementally, because the generated functions communicate across the C stack. Provided the C compiler supports placeholders (prototypes) in place of code, *precc* specifications need not even be complete before they are capable of running and being tested. This allows greatly reduced turn-round times in development and maintenance, and renders the figures for monolithic scripts and C code meaningless.

A fuller description of the *precc* tool may be found in [4, 6].

## 3 The parsing process

Using the C stack and function-calling to do the work instead of a virtual automaton is not a liability. As a realistic demonstration, a significant portion of the concrete syntax for the **occam** programming language [11] has been investigated and part of this is presented in this paper. The philosophical Occam's razor is a maxim that the acceptable principle can be distinguished from the unacceptable one by its simplicity and we propose to use the language **occam** as a razor which separates acceptable parser technology from unacceptable parser technology.

**occam** can be divided into eleven modules which follow the headings in the concrete syntax, as shown in Figure 1. Only the `occam` and `token` modules are extra to the concrete syntax; the former contains some useful meta-constructions and the latter does duty for a lexical analyzer. The modules are associated with a *precc* grammar specification script `foo.y`, the C code that *precc* generates from it, `foo.c`, and the object module `foo.o` produced by the C compiler. Linking all the object modules together with the *precc* kernel library gives one the executable parser. There are 104 separate definitions in all, in 400 lines, and they generate 2,600 lines of C code (60 Kbytes), and not much more object code.

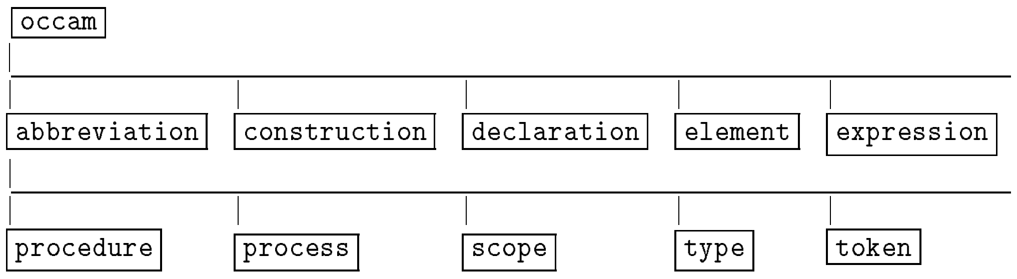

Figure 1: The modular structure of the occam syntax and parser.

Experiments with the *precc* parser for occam show that a C stack of 4 Kbytes can accommodate about 300 nested function calls (therefore averaging 12 bytes each) from the *precc* kernel, and that this size corresponds to occam constructs which are nested five layers deep. The maximum nesting is 40 (that is, 80 columns divided by the 2-space indentation increment imposed by the language), so the runtime stack cannot grow beyond about 32 Kbytes in size in practice. This is an acceptable overhead, and not at all the bugbear that might have been feared for a recursive infinite state machine. The sixty recursive calls per occam nesting layer are composed of about 45% reduce-equivalents, 45% instructions which attach provisional 'actions' to the parse for later execution, and 10% shift-equivalents, each shift usually being associated with an input token, and a context-save of about 20 bytes which permits backtracking. The other calls push only their own return addresses and zero, one or two parameters onto the stack, so the load introduced by the *precc* infinite lookahead and backtracking technology is not very high.

occam is not an easy language to parse. In particular, it requires the lexical indentation of constructs from the left hand edge to be recognized by the parser since this often delimits their scope, in place of the perhaps more familiar BEGIN/END delimiters of Pascal and other block structured languages (or '{' and '}' of C). As a result, knowledge of the current indentation may be needed at many points during parsing, and backtracking to an alternative interpretation may be necessary as the scope of (possibly several) constructs is ended. For example, the two programs below from [17] differ only in the indentation of one line, but the program on the left gives y=10, and that on the right, y=0:

```
SEQ                              SEQ
  x,y,z := 10,1,0                  x,y,z := 10,1,0
  WHILE x>0                        WHILE x>0
    SEQ                              SEQ
      z := z+x                         z := z+x
      x := x-1                         x := x-1
      y := y*x                     y := y*x
```

The scope of the second SEQ is terminated by the 'offside' (less indented) statement.

occam could theoretically be handled by a LR(40) grammar, with a 40 token lookahead since the indentation may be limited to 80 columns (2 spaces per indentation). So *yacc* could handle all practical occam programs, once the LR(40) specification were expanded out into LR(1) form. However, this would be very wasteful of space for the automaton's tables, much of which would be duplicated. The use of purely synthetic attribute grammars is in any case slightly problematic because several syntactic constructs may be terminated by a single lexical group. For example, in

```
SEQ
  SEQ
    SEQ
      x:=y
  y:=z
```

the group y:=z belongs to the outermost SEQ by virtue of its indentation position, which terminates the two innermost SEQ constructs, so it is not clear what token type it should present. A prior pass may be necessary to insert BEGIN and END tokens which can serve as an unambiguous delineation

```
process          =  SKIP | STOP | construction | ...
construction     =  sequence | conditional | loop | ...
conditional      =  IF
                      { choices }
choice           =  guarded.choice | conditional
guarded.choice   =  boolean
                      process
loop             =  WHILE boolean
                      process
```

Figure 2: Sample from **occam** 2 syntax summary.

of the block structure, but this amounts to admitting that *yacc*-type technology is not well-suited to this kind of parsing problem.

Another area of concern, particularly for safety-critical systems, is the correctness of the parsing process [16, 17], and here *precc* also scores heavily. High-level languages, rather than assembler, are now being recommended for safety-critical applications since it is increasingly recognized that programmers make fewer errors as a result. However, high-level language compilers are much more complicated than assemblers and need a higher degree of validation before they can become acceptable in these applications. Much current research into compiler verification concentrates on the compilation process from an abstract tree representation of the language to the object code (e.g., for a subset of **occam** [10]).

Indeed, many declarative programming languages, such as Prolog, provide explicit support for infix and other operators, with variable precedence, etc. The input to programs (e.g., compilers) written in such languages can be supplied in an 'abstract' tree form that is sufficiently readable to be used directly by human programmers, at least for prototyping purposes [2]. This can obviate the need to supply any explicit parser at all. Alternatively, *precc* may be used to generate the abstract tree input to such tools, thus allowing the input language to use the true concrete syntax, even in a prototype compiler, if this is required.

However the initial parsing phase from a concrete representation of the language to an abstract parse-tree is an important link in the whole compilation process in any practical and realistic implementation. *Precc* can make this phase amenable to validation, because *precc* implements a well-understood higher-order model directly in C, and the semantics is associated with a set of proof rules that will demonstrate exactly when a given stream will satisfy (or will not satisfy) a given parse specification. But *precc* scripts themselves correspond almost verbatim to the concrete grammar specification, so there is at least a case for arguing that once the *precc* constructors have been validated and understood, no more proof is required.

In summary, then, *precc* attempts to provide a tool that

1. is flexible and efficient in practice,
2. provides an easy means to generate correct parsers for languages that cannot be dealt with conveniently by many current compiler-compiler tools, and
3. offers the possibility of a validated front-end.

## 4 A realistic parser for Occam

This section presents a *precc* parser for a significant subset of **occam** 2, and demonstrates how closely the parser can be related to the BNF-style semi-formal syntax provided in Appendix G of [11], a sample of which is given in Figure 2. The indentation of constructs is specified by presenting the BNF description in indented form where appropriate.

The corresponding specification in *precc* is shown in Figure 3. This is one of the more complicated and therefore more interesting parts of the specification as far as parsing is concerned. The

```
@ process(n)          =   SKIP | STOP | construction(n) | ...
@ construction(n)     =   sequence(n) | conditional(n) | loop(n) | ...
@ conditional(n)      =   IF $!
@                            [ choices(n+2) ]
@ choice(n)           =   [ specifications(n) ] indent(n,chosen)
@ chosen(n)           =   guardedchoice(n) | conditional(n)
@ guardedchoice(n)    =   boolean $
@                            clause(n+2)
@ loop(n)             =   WHILE somespace boolean
@                            clause(n+2)
@ clause(n)           =   indent(n,process)
```

Figure 3: Corresponding *precc* specification.

*precc* specification differs in several respects. Firstly and most obviously, the majority of the constructions are *parameterized*. They take the parameter $n$, which denotes the absolute indentation of the concrete representation, so that

*conditional(5)*

denotes an IF construct with the initial I in column 5. Where the concrete specification denotes extra indentation by spacing, the *precc* specification uses an incremented parameter instead. Thus

*choices(n+2)*

is included where the concrete syntax instead has *choices* two spaces further to the right than the governing IF keyword.

Secondly, the *precc* specification employs special matches for the '*end of line*'. Although *precc* can match the literal ASCII line feed control character directly, the default lexical analyzer which comes with *precc* generates the special zero token at an end of line, and it is this that is matched by the '*$!*' and '*$*' constructs (see below).

Though *precc* can use lexers generated by the *lex* utility directly, in exactly the same way as *yacc* does, no special lexical analysis is required for the purposes of this trial, and the default lexer can be used. This lexer passes every character unchanged, except that it substitutes a line feed control character with the special zero token. The rationale behind the design of the default lexer is that 'end of line' frequently marks a natural break point at which special action must be taken, and *yacc*-compatible lexers conventionally signal special conditions with the zero token. Actions which check for end of file conditions can be attached to the match for the zero token.

The '*$*' is a match for end of line which can be backtracked over, and the '*$!*' is a match for one which cannot. It incorporates the special *cut* symbol, '*!*', which disables backtracking. It is best to include as many cuts as possible in a *precc* script, because in the event of an error being found in the parsed stream, they prevent exponential searches through all the possible alternative parses. The cut is one of the innovative implementational aspects of *precc* in comparison with other LL($\infty$) parsers; the use of the C call stack is another.

It is certain that an IF begins a conditional statement and nothing else, so a cut may be inserted after the IF, but it is conceivable that it may be necessary to backtrack over the newline following a *boolean* expression in case the following stream does not match a '*clause(n+2)*' after all. In fact, this is probably never going to be the case, but it is best to allow for such a possibility, so a '*$*' is needed at this point.

Thirdly, the *precc* specification uses the square brackets syntax

[ *foo* ]

to indicate an *optional inclusion*, where the concrete syntax uses curly brackets instead. This is a trivial syntactic difference – *precc* reserves the curly brackets for use as *grouping parentheses*.

Fourthly, where the concrete specification has '*process*', the *precc* specification includes the statement '*clause(n+2)*' instead. The '*n+2*' indicates extra indentation to the right, but the *clause(n+2)* construct has been introduced to stand for

$$indent(n{+}2,process)$$

which might equally well be placed directly in the specification. However, giving a name to the construction makes the specification more maintainable because it eliminates repetition and narrows the range over which changes may have to be made at some future point. It also makes the resulting code more compact, at the cost of a tiny loss in runtime performance which may even be optimized away by the C compiler. (Experiments have shown that function calls with simple bodies – which this construction induces – are usually copied into inline code by optimizing C compilers.) All this is possible because of *precc*'s *declarative semantics*. An expression may be substituted by a name for the expression anywhere in the text without altering the semantics.

Why does the *precc* specification need an *indent(n,process)* construct at all, when the concrete syntax includes nothing of the sort? It is because the *n* spaces which appear to the left must be remembered. The concrete syntax specification drops these from the representation of succeeding lines, showing only the relative indentation. Inserting them explicitly gives the concrete specification the shape

$$\begin{array}{lcl} & & guardedchoice(n) \\ & & \downarrow \\ guarded.choice & = & boolean \\ & \underbrace{\sqcup\sqcup \ldots \sqcup\sqcup}_{blank \star n} & \sqcup\sqcup \;\; process \\ & & \qquad\;\uparrow \\ & & \qquad\; process(n{+}2) \end{array}$$

and the *precc* *guardedchoice(n)* definition is to represent the construction starting with a *boolean* in column *n*. The full *precc* specification across the two lines has to include the *n* blank spaces before the *process* part:

$$\begin{array}{lcl} @\ guardedchoice(n) & = & boolean\ \$ \\ @ & & blank\star(n{+}2)\ process(n{+}2) \end{array}$$

and the whole lower line of this definition is replaced in the actual script by $clause(n + 2)$.

We may define the *higher order* construction

$$@\ indent(n,p) \quad = \quad blank\star n\ \ p(n)$$

because the pattern is frequently reused in the **occam** specification. Note that '*blank⋆n*' is the *precc* syntax for '*blank* repeated *n* times', and a '⋆' without a following expression means ' repeated zero or more times'. One example of the use of *indent* is in the definition of *clause(n)*:

$$@\quad clause(n) = indent(n,process)$$

The script can be made even more flexible if required. The *precc* parser can allow *process* to start *beyond* the mandatory two spaces farther to the right, and can compute any indentations to come relative to the actual position in which *process* starts (although this flexibility is disallowed by the standard **occam** language, perhaps because of the extra complexity of handling it). This may be achieved by including

$$indented(n,p)$$

instead of *p(n)*, and defining the *indented* grammars so that they detect any extra *blank*s and incorporate them into the indentation count:

$$\begin{array}{lcl} @\ indented(n,p) & = & blank\ indented(n{+}1,p) \\ @ & \mid & p(n) \\ @\ indent(n,p) & = & blank\star n\ indented(n,p) \end{array}$$

There is some inefficiency introduced here, because a malformed *p* following some blanks may be (partially) scanned twice or more, but it may be considered worthwhile because of the extra flexibility in the acceptable syntax that it affords. There are also good reasons for inserting a test into the specification to ensure that *n* does not grow too large:

$$@\quad indented(n,p) \ = \ blank\ \ )n{<}80(\ \ indented(n{+}1,p) \ \ldots$$

```
# include "occam.h"

@ key(word)        =  )EMPTY(word)(
@                  |  ⟨ HD(word) ⟩ key(TL(word))

@ blank            =  quietblank                 :PRINTSPACE;:

@ quietblank       =  ⟨ ' ' ⟩

@ whitespace       =  quietblank⋆

@ somespace        =  quietblank whitespace

@ name             =  loweralpha+ digit⋆

@ loweralpha       =  (islower)                  :printf("%c",$1);:

@ IF               =  key("IF")

...
```

Figure 4: The token definitions and actions.

This will prevent arbitrary indentations, which are not allowed by the occam folding editor in any case. Nearly all the recursion in the parser is introduced by this one construction. Note that the out-turned parentheses, which denote a parse-time test, may contain any valid C expression returning a one-or-zero integer result, following the C convention for boolean values. There is a danger of destroying the declarative semantics here, since C expressions may be used to implement side-effects, but this possibility will only rarely be required, since the cost in terms of maintainability is great. Side-effects may also be incorporated into parameters, since these may be arbitrary C expressions too. But side-effects in *actions* (see below) attached to the parse can have only limited effect on the parse itself, because *precc* discharges actions after parsing, not during, at the 'cut' points marked in the specification script.

The concrete syntax and the *precc* specification differ in one final point. The *precc* specification allows each *choice* case in an IF statement to be preceded by an optional set of *specifications*. In fact, the concrete syntax specifies this too, but *elsewhere*, with the recursive alternate production:

$$\textit{choice} = \textit{specification choice}$$

In this respect, *precc* is deficient. All the production rules for a given target *must appear together*, as alternates, or equivalent, in the one definition and cannot be scattered across a script, or across several scripts. This is a necessary requirement for modularity and is good practice anyway.

## 5 Tokenisation and actions

As mentioned above, the *precc* parser specification includes a `token` module which acts as the lexical analyzer to the rest of the parser. There is no need to use *precc* as its own lexer, but there is no disadvantage in doing so either. The module consists of eight definitions of various sorts of white space, one meta-production which helps with the recognition of keywords, and the keywords themselves. This module illustrates the remaining parts of *precc*'s language quite well. The module is shown in Figure 4. All the other 'terminals', like *digit*, were defined in the `literal` section of the concrete syntax, and appear in the `literal.y` *precc* specification file.

The first point to note about this script is that any line which does *not* begin with an '@' is passed through as unchanged C code, since *precc* scripts are *literate* in the sense of Donald Knuth [13]. Thus the C preprocessor directives may be used to include files. The `occam.h` file contains the definitions of the C macros `EMPTY` which tests for an empty string, `HD` which returns the head of a non-empty string, and `TL` which returns the tail of a string.

Secondly, the C macros are quite valid as parameters to the grammatical rules, or as a boolean test (inside the out-turned parentheses). But C macros and C code have a more important use, in the *actions* which one can attach to a parser. The actions appear between colons anywhere in the rule, and logically 'occur' in a sequence corresponding to their positioning, just as *yacc* attached actions do. The actions will be discharged whenever a cut point ('*!*') is reached in the parse specification, and cuts disable backtracking.

The actions can make use of the *synthetic attributes* which are attached to each terminal and non-terminal in a rule. Again, the syntax is exactly as in *yacc* (and thus is compatible for existing users, although perhaps not ideal). '*$1*' refers to the first component of the sequence to the left of the action, '*$2*' to the second, and so on. A value can be attached to the whole rule by assigning it to '*$$*'. No assignments are made here, because this parser was only intended to generate a tree representation, properly scoped with the block structure. Thus the *loweralpha* rule outputs the character which is held as the *$1* value using the C `printf("%c",`*$1*`)` library call. *Precc* has to dive into the C expressions in order to translate these references, so it is not possible to hide them within C macros, but otherwise *$1*, *$2*, etc may be treated like (volatile) C variables of an integer type.

That character value was assigned by the default lexer, which placed it in the `yylval` variable when *precc* requested a token. This is the standard mode for lexers generated by *lex*, and the *precc* default lexer conforms to this *de facto* standard, even though it is noticeably less efficient than it might be (the fast way to pass tokens and values to *precc* is to write them directly, a line at a time, into the `yybuffer` address that it presents). The

*(* `islower` *)*

expression is a *predicate*. It tests the incoming tokens using the C `islower()` library call. A matching token will have its `yylval` value scanned and attached as the attribute of the *(* `islower` *)* expression. In this manner, *precc* can handle arbitrary amounts of tokenizing itself without the need for a separate tool like *lex*.

## 6 Trials

It proved possible to write the concrete **occam** syntax down as a *precc* specification script directly from the documentation, maintaining the same structure as the syntax document presented. It took the author of *precc* about ten days of on-off effort to get everything up and working.

The resulting script is remarkably readable, as may be observed from the portions presented above. Inspection has nearly always been sufficient to clarify points of doubt about the parser's behaviour, and on the one or two points where the debugger has been resorted to, it has turned out to be remarkably easy to follow the *precc* semantics in action, because all the declarations in the syntax script correspond to C functions of the same name in the compiled C code (unfortunately, this has meant that one or two changes of name from the concrete syntax have been required – there cannot be an *int* declaration, for example, because it clashes with the C type name), and they are conserved over recompilations.

Much of the work was done on a portable PC, since the *precc* code is portable to any ANSI-compliant C programming environment. The *precc* compiler-compiler itself is 'written' in self-generated C code, so it ports across too. The resulting code also compiled at once when transferred back to a networked, workstation environment, and the **occam** parser itself runs quite fast even on the PC (although it took a while to discover that the PC only allocates 4 Kbytes for the C call stack by default).

Unfortunately a *yacc* **occam** parser was not available to the authors (and may be theoretically problematic anyway, as previously discussed), so no direct comparison is possible. However, experience with the programming language Oberon-2 indicates that the speed of *precc* is comparable or better than *yacc*; the input specification is certainly more readable and compact, and therefore should be more maintainable. [4, 6] contain further details.

## 7 Conclusion

*Precc* provides a balance of efficiency and flexibility that may be useful for handling programming languages that are not conveniently catered for by currently available compiler-compiler tools. Surprisingly, these include many older languages, which were developed before the *lex* and *yacc* tools became available and therefore made no special concessions to the LR(1) technology.

BNF-like descriptions, arranged in modules, may be provided to *precc* which then generates C code for the parsing phase of a compiler or translator for the input language. Modules may be recompiled separately and linked in incrementally. The description language both supports the use of ordinary parameters, and meta-variables, which may hold the names of other grammar descriptions.

As well as occam, *precc* has been successfully applied to the full definition of ANSI COBOL 74 for use on the collaborative REDO project, which is concerned with reverse engineering of programs written in COBOL and other programming languages [14]. Here it is required to translate the program to a higher-level form of representation in the first step towards re-engineering the program. This project has also considered *decompilation* as well, thus allowing the possibility of generating high-level programs from low-level object code [5, 3]. *Precc* is currently being used to generate a practical decompiler-compiler in support of this technique.

For those interested, [4] and [6] include details of how to access a copies of *precc* for use under Unix and MS-DOS.

## References

[1] A.V. Aho and J.D. Ullman. *Principles of compiler design*. Addison-Wesley Publishing Company, 1977.

[2] J.P. Bowen. From programs to object code using logic and logic programming. In R. Giegerich and S.L. Graham (eds.), *Code Generation – Concepts, Tools, Techniques*, Proceedings of the *International Workshop on Code Generation*, Dagstuhl, Germany, 20–24 May 1991. Springer-Verlag, Workshops in Computing, pp. 173–192, 1992.

[3] J.P. Bowen and P.T. Breuer. Decompilation. In H. van Zuylen (ed.), *The REDO Compendium of Reverse Engineering for Software Maintenance*, chapter 9, John Wiley & Sons Limited, 1992. (To appear)

[4] P.T. Breuer. A *PRE*ttier *C*ompiler-*C*ompiler: higher order programming in C. In *Proc. TOULOUSE'92*, 1992. (This volume)

[5] P.T. Breuer and J.P. Bowen. Decompilation *is* the efficient enumeration of types. In M. Billaud *et al.* (ed.), *Journées de Travail WSA'92 Analyse Statique*, BIGRE **81–82**, IRISA-Campus de Beaulieu, F-35042 Rennes cedex, France, pp 255–273, 1992.

[6] P.T. Breuer and J.P. Bowen. *A PREttier Compiler-Compiler: Generating Higher Order Declarative Compiler-Compilers in C*. Technical Report PRG-TR-20-92, Oxford University Computing Laboratory, Programming Research Group, 1992. (Submitted to *Journal of Functional Programming*)

[7] P. Deussen and L.M. Wegner. A bibliography of the van Wijngaarden grammars. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, **6**, 1978.

[8] J.A. Goguen and T. Winkler. *Introducing OBJ3*. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, California, USA, August 1988.

[9] D. Grune. How to produce all sentences from a two-level grammar. *Information Processing Letters*, **19**, pp. 181–185, 1984.

[10] C.A.R. Hoare, He Jifeng, J.P. Bowen and P.K. Pandya, An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language. In Directorate-General of the Commission of the European Communities (ed.), *ESPRIT '90 Conference Proceedings*, Brussels pp. 804–818, Kluwer Academic Publishers B.V., 1990.

[11] INMOS Limited. *Occam 2 reference manual*. Prentice Hall International Series in Computer Science, 1988.

[12] S.C. Johnson and M.E. Lesk. Language development tools. *The Bell System Technical Journal*, **57**(6) part 2, pp. 2155–2175, July/August 1978.

[13] D.E. Knuth, Literate programming, *The Computer Journal*, **27**(2), pp. 97–111, May 1984.

[14] K.C. Lano and P.T. Breuer. From programs to Z specifications. In J.E. Nicholls (ed.), *Z User Workshop, Oxford 1989*, pp. 46–70, Springer-Verlag, Workshops in Computing, 1990.

[15] *The Oxford Dictionary of Quotations*. 3rd edition, Oxford University Press, 1979.

[16] D. Weber-Wulff. *A (vW)-grammar for concrete PL0 syntax and parser correctness*. ESPRIT BRA 3104 ProCoS project document [Kiel DWW 2/2], Christian-Albrechts Universität zu Kiel, Germany, May 1990.

[17] D. Weber-Wulff. Proven correct front-end specification. In B. von Karger (ed.), *Compiler Development*, chapter 5. In *ESPRIT BRA 3104 Provably Correct Systems ProCoS Draft Final Deliverable*, volume 3, October 1991. (Available from Dept. of Computer Science, Technical University of Denmark, Building 344Ø, DK-2800 Lyngby, Denmark)

[18] L.M. Wegner. On parsing two-level grammars. *Acta Informatica*, **14**, pp. 175–193, 1980.