Rebus — A SNOBOL4/Icon Hybrid*

Ralph E. Griswold

TR 84-9

June 9, 1984

Corrected January 23, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

Rebus — A SNOBOL4/Icon Hybrid

1. Introduction

SNOBOL4 [1] remains popular despite advances in programming language design that make many of its features seem quaint and awkward compared to those of modern programming languages. The nominal reason for SNOBOL4's popularity is its string pattern-matching facility, which is unrivaled in its power and conciseness.

The technical reasons for the power and conciseness of SNOBOL4's pattern-matching facility are easy to determine. The appeal of pattern matching in SNOBOL4 to programmers and its usefulness in programming derive from factors that are harder to identify.

SNOBOL4 can be considered to be the combination of two languages, $\mathcal{L}$ and $\mathcal{P}$ [2]. In this view, $\mathcal{L}$ is a language with conventional computational facilities, such as numerical computation, tests, loops, and so forth. $\mathcal{P}$, on the other hand, is a pattern-matching language with nondeterministic control structures. This linguistic dichotomy leads to an increased vocabulary and two different linguistic frameworks that are disharmonious, forcing the programmer to formulate programs in a mixture of two languages.

The main motivating force in the design of Icon [3] was an attempt to integrate these components in a single linguistic framework. The unification that is realized in Icon has proved to be successful in many ways. The success/failure signaling mechanism of SNOBOL4 fits nicely into conventional selection and looping control structures, replacing the Boolean-value mechanism of Algol-like languages [4]. Introducing the search and backtracking control mechanisms of pattern matching into more conventional computational contexts has proved to be unexpectedly useful [5]. On the other hand, there also is evidence [6, 7] that the integration of the $\mathcal{L}$ and $\mathcal{P}$ components has diluted the conciseness and expressiveness of pattern matching in Icon as compared to SNOBOL4.

There may be, therefore, some advantages to the linguistic dichotomy that exists in SNOBOL4 — not because of the dichotomy itself, but because of its nature. Mark Emmer puts it this way [8]:

> Programming languages such as PASCAL, BASIC, C, and Assembler, with their IF ... THEN ... ELSE, REPEAT ... WHILE mentality, are serial, sequential, ploddingly left-brained. [Pattern matching in] SNOBOL4 seems to be parallel, associative, intuitive, right-brained. This is the crux of the matter. Certainly we use our left brain for problem solving, but how many of us really think exclusively in terms of IF ... THEN ... ELSE? Imagination, creativity, great leaps of conception seem to originate in the inductive, parallel-functioning right brain. And this is precisely where SNOBOL4's pattern-matching abilities lie.

Even if the linguistic dichotomy that exists in SNOBOL4 is viewed as beneficial rather than as detrimental, it is nonetheless indisputable that the $\mathcal{L}$ component of SNOBOL4 — with its lack of control structures — is awkward and out of date. This raises the interesting possibility of "modernizing" SNOBOL4 by overhauling its $\mathcal{L}$ component, while retaining its $\mathcal{P}$ component essentially unchanged.

Such a language has several potential advantages. It would facilitate the evaluation of the "left-brain/right-brain" hypothesis. Furthermore, it would make use of SNOBOL4 pattern matching more palatable in some contexts. For example, SNOBOL4 is often taught in courses on comparative programming languages because of its pattern-matching facilities, but its other characteristics are an embarrassment.

This report describes an experimental language, called Rebus*, that replaces much of the $\mathcal{L}$ component of SNOBOL4 by a more modern structure. Experience from Icon has been used here; the control structures and syntax of Rebus are adapted from those of Icon. The $\mathcal{P}$ component of Rebus, except for minor syntactic changes, is that of SNOBOL4. Except for some syntactic enhancements, the function and operation repertoire

*The name Rebus was chosen for its meaning and is not an acronym. ICURYY!

of Rebus is that of SNOBOL4.

The idea of improving the $\mathfrak{L}$ component of SNOBOL4 is not new. Earlier approaches include extensions to SNOBOL4's control structures [9, 10], preprocessors to produce "structured SNOBOL4" [11-15], and even programming styles that effect the appearance of control structures [16]. These approaches all add to the $\mathfrak{L}$ component of SNOBOL4 without taking anything away. The design of Rebus is more radical. It provides no "escape mechanism" for accessing all of SNOBOL4's capabilities. For example, Rebus has no labels or gotos. One cannot transliterate an arbitrary SNOBOL4 program into Rebus. Instead, Rebus includes some features of SNOBOL4, excludes others, and transforms others into different forms. Rebus also has syntactic support for writing well-organized programs that is not available in SNOBOL4.

Rebus is implemented by a preprocessor via a variant Icon translator [17]. The preprocessor accepts Rebus input and outputs SNOBOL4 code, which is then run under SNOBOL4.

The material that follows assumes that the reader is familiar with Icon and SNOBOL4. In the interest of brevity, details are referenced to Icon and SNOBOL4 as appropriate.

2. Syntactic Characteristics of Rebus

The syntax of Rebus is very similar to that of Icon. For example, Rebus uses Icon's convention for comments.

String literals can be enclosed in either single or double quotation marks as in SNOBOL4. The Icon convention for continuing quoted literals is available, but literal escape sequences are not. Real literals can be given in exponent-form as in Icon.

The syntax of identifiers is the same as that of Icon. Except in quoted literals, upper- and lowercase letters are equivalent in Rebus. The Rebus preprocessor maps all nonliteral letters to uppercase for compatibility with SNOBOL4.

The treatment of blanks in Rebus is as it is in Icon: blanks are optional around infix operators, except where they are necessary to disambiguate infix/prefix combinations. Blanks are optional between prefix operators and their operands.

To accommodate the semantics of SNOBOL4, Rebus distinguishes between statements and expressions. Reserved words are used for distinguished syntactic constructions, as in Icon. A grammar for Rebus is given in the appendix.

2.1 Expressions

Rebus supports all the expressions that are supported by SNOBOL4, although there are some differences in their syntactic representation. For example, assignment in Rebus is represented by

$expr_1$:= $expr_2$

The right operand of an assignment expression cannot be omitted in Rebus to indicate assignment of the null string as it can be in SNOBOL4. Note that assignment is an expression, not just a statement as in standard SNOBOL4. This treatment of assignment allows the use of the capabilities of SNOBOL4 dialects such as MACRO SPITBOL [18]. Rebus provides an exchange operation

$expr_1$:=: $expr_2$

as well as augmented assignment operations for incrementing and decrementing numerical values:

$expr_1$ +:= $expr_2$
$expr_1$ −:= $expr_2$

These operations are only abbreviations. For example,

$expr_1$ +:= $expr_2$

is equivalent to

$expr_1$:= $expr_1$ + $expr_2$

so that $expr_1$ is evaluated twice.

Rebus supports operator notation that can be used in place of functional syntax for several operations in SNOBOL4. For example, the expression

$expr_1$ % $expr_2$

is available as a synonym for remdr($expr_1$, $expr_2$)*.

Comparison operations also can be represented in Rebus using operator syntax, as in:

$expr_1$ = $expr_2$

which is synonymous with

eq($expr_1$, $expr_2$)

Icon's notation for lexical comparison is used in Rebus, as in

$expr_1$ >>= $expr_2$

which is synonymous with

lgt($expr_1$, $expr_2$)

All six lexical comparison operators are provided. Library routines are included for those lexical comparison operators that are not provided in standard SNOBOL4. The two object comparison operators,

$expr_1$ === $expr_2$

and

$expr_1$ ~=== $expr_2$

are synonyms for ident($expr_1$, $expr_2$) and differ($expr_1$, $expr_2$), respectively. The prefix operations

/*expr*
expr

are provided as synonyms for ident($expr_1$) and differ(*expr*), respectively.

In Rebus, string concatenation is represented by

$expr_1$ || $expr_2$

while pattern concatenation is represented by

$expr_1$ & $expr_2$

Because SNOBOL4 does not distinguish these two kinds of concatenation syntactically and Rebus programs are translated into SNOBOL4, there is no way to check the appropriateness of the concatenation operations that are used in Rebus programs. The augmented assignment operation

$expr_1$ ||:= $expr_2$

is available as a synonym for

$expr_1$:= $expr_1$ || $expr_2$

The expression

$expr_1$[$expr_2$ +: $expr_3$]

produces the substring of $expr_1$ starting at $expr_2$ and extending $expr_3$ characters and is a synonym for the

*Lowercase letters are used for SNOBOL4 constructions throughout this report, although uppercase latters are required in most cases in standard SNOBOL4.

MACRO SPITBOL function substr($expr_1$, $expr_2$, $expr_3$). This function also is provided in the library that is used when Rebus programs are run under standard SNOBOL4.

Two new keywords are also provided for convenience:

```
&lcase
&ucase
```

and are synonyms for

```
"abcdefghijklmnopqrstuvwxyz"
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

respectively.

2.2 Statements

There are two selection control structures with obvious meanings:

if $stmt_1$ then $stmt_2$ [else $stmt_3$]
unless $stmt_1$ then $stmt_2$

For example,

```
if count > 0 then output := count
```

writes the value of count, provided that it is greater than 0. Note the use of SNOBOL4-style output.

There are three looping control structures:

while $stmt_1$ do $stmt_2$
until $stmt_1$ do $stmt_2$
repeat *stmt*

The repeat control structure evaluates *stmt* repeatedly as long as *stmt* succeeds but terminates if *stmt* fails. Note the difference from the corresponding control structure in Icon. For example,

```
repeat output := input
```

copies the input file to the output file.

There are two control structures for loop control:

```
exit
next
```

The exit control structure transfers control to the point immediately after the loop in which it appears, while the next control structure transfers control to the beginning of the loop. For example,

```
while line := input do
  if line == "start" then exit
```

reads input lines until one consisting of start is encountered.

There is an iteration control structure:

for *identifier* from $expr_1$ to $expr_2$ [by $expr_3$] do *stmt*

For example,

```
for i from 1 to 10 do
   output := a[i]
```

writes out the first 10 elements of the array a.

A case control structure selects a statement to evaluate depending on the value of an expression. It is similar to the corresponding control structure in Icon and has the form:

```
case expr of { caselist }
```

where a *caselist* is a list of case clauses:

```
caseclause ; ...
```

and a case clause may have one of two forms:

```
expr : stmt
default : stmt
```

The semantics for the selection of the statement to evaluate and the handling of failure in a case statement are the same as those for the corresponding Icon control structure. An example is

```
case s of {
   "w" : output := text
   "r" : text := input
   "d" : text := ""
   default : output := "erroneous command"
   }
```

Rebus has pattern-matching and replacement statements like those of SNOBOL4 but with operator syntax:

$expr_1$? $expr_2$
$expr_1$? $expr_2$ <– $expr_3$

An example is

```
while line ? wpat <– "" do
   count +:= 1
```

For convenience, the commonly used form

$expr_1$? $expr_2$ <– ""

can be abbreviated as

$expr_1$?– $expr_2$

Thus the example above can be written as

```
while line ?– wpat do
   count +:= 1
```

There are two forms of return from functions:

```
fail
return [ expr ]
```

The null string is returned if the expression in the return statement is omitted.

The stop statement causes program termination. Finally, a statement may consist only of an optional expression:

```
[ expr ]
```

or it may be compound:

```
{ [ stmt ; ... ] }
```

An example is

```
if text ? pat then {output := text; text := ""}
```

Semicolon insertion is performed automatically at the ends of lines as it is in Icon. Therefore the example above can be written as

```
if text ? pat then {
   output := text
   text := ""
   }
```

2.3 Declarations

Records and functions are declared in Rebus. The syntax of a record declaration is the same as it is in Icon:

record *identifier* (*arglist*)

where *arglist* is a list of zero or more identifiers:

[*identifier* , ...]

Records are handled as they are in Icon and SNOBOL4 (via defined data objects). Instances of records are created and referenced as they are in SNOBOL4. For example, given the record declaration

```
record complex(r, i)
```

the expression

```
z := complex(1.0, 2.3)
```

assigns a complex record to z and

```
i(z) +:= 1.0
```

increments its i field by 1.0.

The form of a function declaration is

function *identifier* (*arglist*)
   [local *identifier* , ...]
   [initial *stmt*]
   [*stmt* ; ...]
end

Flowing off the end of a function is equivalent to return.

For example,

```
function main()
   local i
   i := 0
   repeat i +:= size(input)
   output := i
end
```

is a function that prints the total number of characters in the input file (not counting line separators).

Identifiers are dynamically scoped as they are in SNOBOL4. The initial clause, if present, causes its statement to be evaluated once on the first call of the function.

Program execution begins with a call to the function main. Every program must have a function named main.

3. Examples

Comparing Rebus to SNOBOL4 is difficult because of the number of points of difference, as well as differences of opinion about the relative merits of alternative language features. Specific comparisons are almost inevitably biased, since a program written in one language influences the form of the same program written in the other language, once they are placed side by side. It is comparatively easy to translate a Rebus program into SNOBOL4, but the converse is much harder because of the undisciplined use of gotos in most SNOBOL4 programs. These problems should be kept in mind when interpreting the following examples.

Word Counting

The following program for counting words is typical of a large class of programs written in SNOBOL4: input lines are analyzed, data is stored in a table, and finally the results are written out.

```
         letter = "abcdefghijklmnopqrstuvwxyz"
+          "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
         wpat = break(letter) span(letter) . word
         count = table()
read     text = input                                              :f(sort)
findw    text wpat =                                               :f(read)
         count[word] = count[word] + 1                             :(findw)
sort     result = sort(count)                                      :f(nowords)
         output = "Word count:"
         output =
         i = 0
print    i = i + 1
         output = rpad(result[i = i + 1, 1], 15) lpad(result[i, 2],4)   :s(print)f(end)
nowords  output = "There are no words"
end
```

This program can be cast in Rebus as follows:

```
function main()
   letter := &lcase || &ucase
   wpat := break(letter) & span(letter) . word
   count := table()
   while text := input do
      while text ?- wpat do
         count[word] +:= 1
   if result := sort(count) then {
      output := "Word count:"
      output := ""
      i := 0
      repeat output := rpad(result[i +:= 1, 1], 15) || lpad(result[i, 2],4)
      }
   else output := "There are no words"
end
```

Note that the SNOBOL4 and Rebus programs are approximately the same size. There is an obvious trade-off between brevity and program structure in the handling of the flow of control. The Rebus solution benefits from some syntactic conveniences, such as **&lcase**, that would otherwise make it longer.

The corresponding Icon program is very similar to the Rebus one, except for the details of the output loop:

```
procedure main()
   letter := &lcase ++ &ucase
   count := table(0)
   while text := read() do
      text ? while tab(upto(letter)) do
         count[tab(many(letter))] +:= 1
   result := sort(count)
   if *result > 0 then {
      write("Word count:\n")
      every pair := !result do
         write(left(pair[1], 15), right(pair[2], 4))
      }
   else write("There are no words")
end
```

The difference between pattern matching and string scanning is evident, even in this simple example. In the Rebus solution, there is the conceptual separation mentioned earlier:

```
wpat := break(letter) & span(letter) . word
                  .
                  .
                  .
while text ?- wpat do
   count[word] +:= 1
```

Contrast this with Icon string scanning:

```
text ? while tab(upto(letter)) do
   count[tab(many(letter))] +:= 1
```

Binary Trees

The following program for constructing binary trees comes from a book on SNOBOL4 programming techniques [19]. The function btree converts a string specification of a binary tree to a structure composed of records, while bexp converts a binary tree structure back into a string. The main program loop tests these functions by reading in string specifications, converting them to structures, and then writing out the result of converting them back into strings.

```
            define("addl(n1, n2)")
            define("addr(n1, n2)")
            define("btree(s)l, r")
            define("bexp(t)l, r, s")

            data("bnode(value, left, right, up)")

            two = "(" bal . l "," bal . r ")"
            rone = "(," bal . r ")"
            lone = "(" bal . l ")"
            tform = break("(") . s (two | rone | lone)
read        output = bexp(btree(input))                  :s(read)f(end)

addl        left(n1) = n2
addu        up(n2) = n1                                  :(return)
addr        right(n1) = n2                               :(addu)
```

```
btree       s tform
            btree = bnode(s)
            (differ(l) addl(btree, btree(l)))
            (differ(r) addr(btree, btree(r)))                    :(return)

bexp        bexp = value(t)
            l = differ(left(t)) bexp(left(t))
            r = differ(right(t)) "," bexp(right(t))
            s = l r
            bexp = differ(s) bexp "(" s ")"                      :(return)

end
```

This program illustrates several idiosyncrasies of SNOBOL4: the function definition mechanism, the device of returning the value of the function name, conditional tests in concatenations, and sharing code between functions (in addl and addr).

The corresponding Rebus solution, obtained by translating the SNOBOL4 solution, is:

```
record bnode(value, left, right, up)

function main()
   repeat output := bexp(btree(input))
end

function addl(n1, n2)
   left(n1) := n2
   up(n2) := n1
end

function addr(n1, n2)
   right(n1) := n2
   up(n2) := n1
end

function btree(s)
   local l, r, t
   initial {
      two := "(" & bal . l & "," & bal . r & ")"
      rone := "(," & bal . r & ")"
      lone := "(" & bal . l & ")"
      tform := break("(") . s & (two | rone | lone)
      }
   s ? tform
   t := bnode(s)
   if \l then addl(t, btree(l))
   if \r then addr(t, btree(r))
   return t
end
```

```
function bexp(t)
   local l, r, s1, s2
   s1 := value(t)
   if \left(t) then l := bexp(left(t))
   if \right(t) then r := "," || bexp(right(t))
   s2 := l || r
   if \s2 then return s1 || "(" || s2 || ")"
   else return s1
end
```

Note the use of the initial clause to construct the patterns used in btree.

4. Running Rebus

The command

Rebus *options file*

translates the specified file, whose name must end with the suffix .reb. The result is a corresponding .sno file. The available options are

- **–s** produce code to run under the MACRO SPITBOL dialect of SNOBOL4 (the default)
- **–4** produce code to run under the standard implementation of SNOBOL4
- **–x** execute the translated program

An appropriate library is provided, depending on the SNOBOL4 option that is selected.

Compilation errors are detected by the Rebus preprocessor and are given in terms of the source line in the Rebus program.

Run-time errors are detected by SNOBOL4, but the line number in the Rebus source program is given, as well as the statement number in the generated SNOBOL4 program. Tracing and statistics from SNOBOL4 are reported in terms of statement numbers.

5. Programming Considerations

SNOBOL4 programmers should be careful to use

$expr_1 := expr_2$

in Rebus programs in place of

$expr_1 = expr_2$

Note that the latter expression is syntactically correct in Rebus, which may obscure a mistake of this kind. On the other hand, the omission of a concatenation operation in a Rebus program, which is another mistake commonly made by SNOBOL4 programmers, is usually detected as a syntactic error.

If Rebus programs are run under standard SNOBOL4, assignment is limited to its statement form*. That is,

$expr_1 := expr_2 := expr_3$

cannot be used with standard SNOBOL4.

The SNOBOL4 programs generated by the Rebus translator and the support libraries use identifiers that end in underscores. Such identifiers should not the used in Rebus programs.

The run-time error trapping mechanism relies on the SNOBOL4 keyword &errlimit and tracing. If a

*It is possible to generate code for assignment so that it can be used as an expression in standard SNOBOL4. The execution overhead required does not justify this luxury.

Rebus program interferes with tracing, run-time error detection may be ineffective.

6. Conclusions

Rebus is an experiment. It does not attempt to address many of the issues in programming language design that SNOBOL4 highlights. In particular, there are many aspects of pattern matching that could bear examination.

Experience with the use of Rebus provides support for the thesis presented in the introduction of this report: it seems to be much easier to program in Rebus than in SNOBOL4 and the introduction of control structures into the semantic framework of SNOBOL4 does not dilute the power of pattern matching.

What this suggests is not so much that SNOBOL4 should be redesigned, but that SNOBOL4-style pattern matching may have a useful role in more modern languages.

Acknowledgements

Dave Hanson, Madge Griswold, Bill Mitchell, and Steve Wampler provided a number of helpful comments on the presentation of the material in this report.

References

1. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language*, second edition. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1971.
2. Griswold, Ralph E. and David R. Hanson. "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April 1980), pp. 153-172.
3. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.
4. Griswold, Ralph E. "Expression Evaluation in the Icon Programming Language", *1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas.
5. Wampler, Stephen B. and Ralph E. Griswold. "Result Sequences", *Computer Languages*, Vol. 8, No. 1 (1983), pp. 1-14.
6. Griswold, Ralph E. *Pattern Matching in Icon*, Technical Report TR 80-25, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1980.
7. Griswold, Ralph E. *Icon Newsletter #15*, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1984.
8. Emmer, Mark B. *SNOBOL4+; The SNOBOL4 Language for the 8086/8088 Computer Family*, Catspaw, Inc., Salida, Colorado. 1984.
9. Griswold, Ralph E. "Suggested Revisions and Additions to the Syntax and Control Mechanism of SNOBOL4", *SIGPLAN Notices*, Vol. 9, No. 2 (February 1974), pp. 7-23.
10. Abrahams, Paul W. "Improving the Control Structure of SNOBOL4", *SIGPLAN Notices*, Vol. 9, No. 5 (May 1974), pp. 10-12.
11. Sommerville, Ian. "S-SNOBOL — Structured SNOBOL4", *SIGPLAN Notices*, Vol. 14, No. 1 (January 1979), pp. 91-99.
12. Hanson, David R. "RATSNO — An Experiment in Software Adaptability", *Software — Practice and Experience*, Vol. 7 (1977), pp. 625-630.
13. Beyer, Terry. *FLECS: User's Manual*, Technical report, Department of Computer Science, University of Oregon, Eugene, Oregon. 1975.
14. Croff, David L. *SNOFLEX Handbook*, Technical report, Department of Computer Science, Eugene, Oregon. 1974.

15. Haight, R. C. *The SNOFLAKE Programming Language*, Technical Memorandum 70-9155-2, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. 1970.

16. Arora, Kamal A. and William F. Applebe. *Structured Programming in SNOBOL or SNOBOL Considered Not Harmful.* Technical Report Cs 78 12, Department of Computer Science, Southern Methodist University, Dallas, Texas. 1978.

17. Griswold, Ralph E. *The Construction of Variant Translators for Icon*, Technical Report TR 83-19, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1983.

18. Dewar, Robert B. K. and Anthony P. McCann. "MACRO SPITBOL — A SNOBOL4 Compiler", *Software — Practice and Experience*, Vol. 7 (1977), pp. 95-113.

19. Griswold, Ralph E. *String and List Processing in SNOBOL4; Techniques and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1975.

Appendix — Grammar for Rebus

program	→	*decls end–of–file*
decls	→	
	→	*decls decl*
decl	→	*fnc*
	→	*record*
record	→	record *identifier* (*arglist*)
fnc	→	*fnchead* ; *locals initial fncbody* end
fnchead	→	function *identifier* (*arglist*)
arglist	→	
	→	*idlist*
idlist	→	*identifier*
	→	*idlist* , *identifier*
locals	→	
	→	*locals* local *idlist* ;
initial	→	
	→	initial *stmt* ;
fncbody	→	
	→	*stmt* ; *fncbody*
stmt	→	*nexpr*
	→	stop
	→	exit
	→	next
	→	*return*
	→	*match*
	→	*repl*
	→	*repln*
	→	*for*
	→	*if*
	→	*unless*
	→	*case*
	→	*while*
	→	*until*
	→	*repeat*
	→	{ *compound* }
return	→	fail
	→	return *nexpr*
match	→	*expr* ? *expr*
repl	→	*expr* ? *expr* <– *expr*
repln	→	*expr* ?– *expr*

for → for *identifier* from *expr* to *expr* do *stmt*
→ for *identifier* from *expr* to *expr* by *expr* do *stmt*

if → if *stmt* then *stmt*
→ if *stmt* then *stmt* else *stmt*

unless → unless *stmt* then *stmt*

case → case *expr* of { *caselist* }

caselist → *cclause*
→ *caselist* ; *cclause*

cclause → default : *stmt*
→ *expr* : *stmt*

while → while *stmt* do *stmt*

until → until *stmt* do *stmt*

repeat → repeat *stmt*

compound → *stmt*
→ *stmt* ; *compound*

nexpr →
→ *expr*

expr → *expr1*
→ *expr1* :=: *expr*
→ *expr1* := *expr*
→ *expr1* +:= *expr*
→ *expr1* −:= *expr*
→ *expr1* ||:= *expr*

expr1 → *expr2*
→ *expr2* | *expr1*

expr2 → *expr3*
→ *expr2* == *expr3*
→ *expr2* >>= *expr3*
→ *expr2* >> *expr3*
→ *expr2* <<= *expr3*
→ *expr2* << *expr3*
→ *expr2* ~== *expr3*
→ *expr2* = *expr3*
→ *expr2* >= *expr3*
→ *expr2* > *expr3*
→ *expr2* <= *expr3*
→ *expr2* < *expr3*
→ *expr2* ~= *expr3*
→ *expr2* === *expr3*
→ *expr2* ~=== *expr3*

expr3 → *expr4*
→ *expr3* || *expr4*
→ *expr3* & *expr4*

```
expr4        →
             →    expr4 @ expr5

expr5        →    expr6
             →    expr5 + expr6
             →    expr5 − expr6

expr6        →    expr7
             →    expr6 * expr7
             →    expr6 / expr7
             →    expr6 % expr7

expr7        →
             →    expr8 ∧ expr7

expr8        →    expr9
             →    expr9 $ expr8
             →    expr9 . expr8

expr9        →    element
             →    @ expr9
             →    . expr9
             →    ! expr9
             →    + expr9
             →    * expr9
             →    / expr9
             →    $ expr9
             →    ∧ expr9
             →    ~ expr9
             →    − expr9
             →    ? expr9

element      →    literal
             →    identifier
             →    ( exprlist )
             →    element [ exprlist ]
             →    element [ expr +: expr ]
             →    element [ expr ]
             →    identifier ( exprlist )
             →    & identifier

exprlist     →    nexpr
             →    exprlist , nexpr

literal      →    integer−literal
             →    real−literal
             →    dqstring−literal
             →    sqstring−literal
```