OCCAM Super-Entities and the Inline Value Process Question

Larry Dickson
Space Sciences Corporation
7-10 May 2019, 21-22 May 2019

Summary

The form of the OCCAM-2 language described in [D] was as close as possible to standard OCCAM-2 (see [I]) while compiling under lex/yacc open source (flex/bison), with one major exception: the inline value process (see [I] p 65-66) is replaced with the newly defined ANY FUNCTION, under the claim that the latter construct may always be made to perform equivalently to the former.

This note will provide a proof of that. In so doing, it will introduce some new concepts that apply to the OCCAM language syntax, and may be very useful in general compiler construction of OCCAM and its variants and extensions.

Entities (Syntactic Objects) and their Preorder

I use the term "entity" as a synonym for syntactic object, in the Backus-Naur-like format acceptable by the open-source yacc variant, bison. See for example **G.2 Ordered syntax** in [I], pages 86-90, and **C. CRI.TXT (BNF)** near the end of [D].

Entities are defined in terms of other entities, keywords, and symbols. An example from [D] is

```
input = channel '?' input_item_slist
| channel '?' CASE tagged_list
| delayed_input
```

A preorder (see [WP]) called DESCENT is defined upon all entities by defining

```
a >~ b
```

if any branch of the definition of `a` can include `b`, with the preorder extended so that

```
a >~ a
```

is always true, and

```
a >~ b AND b >~ c => a >~ c
```

The latter is realistic, since the instance of `b` may itself be expanded using its definition and thus can include `c`.

Thus, as defined in [WP], the preorder of descent is reflexive and transitive. It is typically NOT antisymmetric, and so does not form a partial order. Instead, it breaks up the set of entities into EQUIVALENCE CLASSES where

`a ~ b if a >~ b AND b >~ a` (DEFINITION)

If `A` and `D` are equivalence classes, then we say

```
A >= D
```

if `a >~ d` for `a` in `A` and `d` in `D`. By transitivity, together with the equivalence class definition, this condition is equivalent for all `a` in `A` and `d` in `D`. Notice that this is indeed a partial order, since

```
A >= D AND D >= A => A = D
```

since this implies `a ~ d` and hence `A` and `D` must share all their members.

This partially ordered set of equivalence classes by descent is based on the recursive nature of compiling, where for instance an expression may be built up from operands and operators, where the operands themselves are expressions. This results in `expression` and `operand` being in the same equivalence class.

Of equal interest is the fact that `A >= D` may be true where `A` is not equal to `D`, and this is written

```
A > D
```

(`A` dominates `D` by descent). This implies interesting structure, inner and outer, and the more such instances there are the better-defined this structure is. The replacement of inline value processes with ANY FUNCTIONs is designed to enrich this structure, by clearly distinguishing full-line entities from sub-line entities, so that full-line entities dominate sub-line entities and never vice versa.

Line Structure

In [D], after the removal of the inline value process, it is shown that

TOP > PROCESS > LINE > MIDDLE > EXPRESSION > SIMPLE

and of these the sets TOP, PROCESS, and LINE members always end in NEWLINE followed by zero or more ROUTD, while members of MIDDLE, EXPRESSION, and SIMPLE never contain NEWLINE, RIND, or ROUTD. (ROUTD decreases current indentation by one step of two spaces, while RIND increases current indentation by one step.) Of the sets shown, PROCESS and EXPRESSION are true equivalence classes, while TOP, LINE, MIDDLE, and SIMPLE are each a union of a number of equivalence classes.

It is also true that when one of TOP, PROCESS or LINE members appears in a definition, it is at the beginning or is preceded by another TOP, PROCESS, or LINE member or by a NEWLINE followed by a RIND or by zero or more ROUTD. By induction, this implies that no member of TOP, PROCESS, or LINE appears in a `program` (the topmost entity) without either being at the beginning of the `program` or being preceded by a NEWLINE followed by a RIND or by zero or more ROUTD. It can also be proven that indentation (starting at zero steps and carried on by accumulating RIND and ROUTD counts) is never nonnegative. Thus it makes sense to say that all members of TOP, PROCESS, and LINE are full-line entities.

By contrast, none of the members of MIDDLE, EXPRESSION, or SIMPLE are full-line entities, and when some of them appear in the definitions of members of TOP, PROCESS, and LINE, they may be preceded and/or followed by others, and they end with a NEWLINE followed by a RIND or a NEWLINE followed by zero or more ROUTD.

An apparent exception is the internal newlines to be found in a multi-line string ([I] page 26). However, since interior entries terminate in an asterisk followed by a newline, which is one of the line continuation characters ([I] page 3), this behavior may be considered part of the line continuation rules, applied before detecting "real" NEWLINEs.

This is the line-based structure of OCCAM as defined in [D]. It is also valid in occam-2 as defined in [I], except for the single glaring exception of the inline value process. The line-based structure is so valuable, as will be shown by the super-entity and global place discussion below, that it is worth saving by replacing the inline value process.

Super-Entity Structures and Complete Categorization

To proceed with the proof that inline value processes can be replaced by equivalent ANY FUNCTIONs, a complete grasp of the structure of an OCCAM program is required. The reason is that a place for insertion of the ANY FUNCTION must be determined that will result in its behaving identically to the inline value process it replaces. Intuitively, it is inserted after all preceding global variables are defined, but before the line in which the inline value process appears. The super-entities to be developed here - which involve complete surveys of OCCAM full-line entities - will enable this to be done rigorously.

The super-entities will also show paths to easy coding of full semantic compilers, and to extension of the language.

The approach to any OCCAM program, as required by yacc (bison), is to start at the syntactic top entity, program. The full-program structure supported here is that of a library of separately compilable FUNCTIONs or PROCs (with all externals defined in the formal parameter list, not as free globals).

Every entity defined in the Appendix, as can be proven by exhaustive nested enumeration, ends in a NEWLINE RIND or in a NEWLINE followed by zero or more ROUTD. Also provable by descent from `program` among the entities in the Appendix is the fact that, in the program, every entity is either initial (the first thing in the program) or preceded by another entity in the Appendix - thus it cannot begin in the middle of a line. Finally, exhaustive nested enumeration shows that internally to every entity, the number of RIND is greater than or equal to the number of ROUTD at every point, and they are equal at the end of the entity. This means each entity is a block with possible internal indentation, but does not change the external indentation. Zero indentation is normally assumed at the top of the `program`.

Because of this, it is found by inspection that every multiple entity (ending in _list or _vlist) is built as a sequence of a number (at least 1) of similar simple entities with the same indentation. In particular, the `program = block_definition_list` consists of a number of `block_definition` entities, each containing either a `function_body` or a `procedure_body` between its headline and its colon. Our first super-entity is thus

`body` `= function_body | procedure_body = valof | process`

(the latter equality is by tracing `function_body = value_process = valof` and `procedure_body = process`).

It is worth noting that a body never ends in a colon (before its last NEWLINE RIND or NEWLINE plus zero or more ROUTD). There are a number of entities that do, and they get the super-entity

`resource:` `= allocation | specification = allocation | declaration | abbreviation | definition`

(where the last equality comes from tracing `specification = declaration | abbreviation | definition`). The final colon is part of the **`resource:`** name, to remind that every resource ends in a colon.

It is found by exhaustive inspection that block structure is well defined for both **`body`** and **`resource:`**. We get for every block body (body of more than one line) that

```
body =
resource:(s)
body_headline
  RIND follow_block(s) ROUTD
```

And for every block resource: (resource: of more than one line) that

```
resource: =
resource_headline
  RIND resource_content_block(s) ROUTD
: NEWLINE
```

It is to be noted here that a follow_block or a resource_content_block need not be more than one line, and that **(s)** means zero or more of them, all at the same indentation.

A further distinction among block body entities is made, depending on whether their follow_block(s) are independent (body) entities or can be dependent (branch) entities:

body = nesting_body I branching_body I single_line_body

```
nesting_body =
resource:(s)
nesting_body_headline
  RIND body(s) ROUTD
```

```
branching_body =
resource:(s)
branching_body_headline
  RIND branch(s) ROUTD
```

Here, a branch may be a body entity or a branch entity, which is dependent and has no meaning apart from its enclosing branching_body.

By tracing, we find that

single_line_body = SKIP NEWLINE | STOP NEWLINE | action | instance

nesting_body = sequence | loop | parallel

branching_body = valof | case_input | conditional | selection | alternation

(The last two can include single-line cases too, if **resource:(s)** and **follow_block(s)** are both empty. These cases, though permitted by the semantics, are rather useless.)

The specialist dependent **`follow_block(s)`** corresponding to the **`branching_body`** cases are all but the first defined as **branch_entity(s)**:

`RESULT expression_list NEWLINE`	for	`valof`
`variant`	for	`case_input`
`choice`	for	`conditional`
`option`	for	`selection`
`alternative`	for	`alternation`

These, vlists of them, and building-blocks (`guarded_alternative`, `guarded_choice`, `guard`) to construct them, complete the list of non-resource: structure defined in the Appendix. The structure of block resource:s, which must be `definition`s, adds only the `tagged_protocol` lines that may be in a PROTOCOL. This (preceded by `CASE NEWLINE RIND` and followed by `ROUTD`) is the only special **`resource_content_block`** yet defined, though extensions of OCCAM will add more.

Taking all the above together exhausts all the entities of the Appendix.

The inline value process (ivp)

The inline value process, which is offered as a definitional instance of the `expression_list` and of the `operand`, is

```
( value_process
)
```

in which it is insisted that the (and) be in the same ACTUAL indentation, with the) preceded by whitespace on its line. *The* `value_process` *will contain embedded* NEWLINE*s,* RIND*s, and* ROUTD*s, which are based on the ACTUAL indentation of its first line, which is on the same line as the* (. The final) is NOT followed by a NEWLINE as a part of this entity definition, which is treated as an `expression_list` or `operand` in a greater line.

In legal OCCAM-2 code as defined in [I], which except for this entity is included in legal OCCAM code as defined in [D], this is the only way that unbalanced parentheses can arise in a true line (after removal of continuation lines). This is proved by scanning all entities of the syntax. This yields four things:

(1) The only case of an initial close parenthesis) in a line is the end of the above inline value process;

(2) After disregarding initial close parentheses, the only case of imbalance of parentheses are the open parenthesis of the beginning of the above inline value process;

(3) After disregarding initial close parentheses, the number of close parentheses in a line never exceeds the number of open parentheses proceeding left to right at any point within the line;

(4) Hence the open parentheses starting the inline value processes can be located as the last of each lost net parenthesis depth in the line.

An example (showing only the parentheses) of (4) is:

```
      1              2
      v              v
(  )  (  (  )  (  )  (  (  )
```

Here 1 is after the last point of depth 0, and 2 is after the last point of depth 1. A snippet of occam code (a `process`), variant of one on p 65 of [I], illustrating this rather pathological possibility is:

```
INT logbit :
VAL INT wordlogbit IS 5 : -- 32 bit words
VAL [7]INT v IS [1,27,421,3,32309,36,4096] : -- list of sizes
logbit := (VAL INT tot IS (INT sum :
                           VALOF
                             SEQ
                               sum := 0
                               SEQ i = 0 FOR SIZE v
                                 sum := sum + v[i]
                             RESULT sum
                          ) :
           INT inclog, totcopy :
           VALOF
             SEQ
               totcopy := tot
               inclog := 0
               WHILE totcopy <> 0
                 SEQ
                   totcopy := totcopy >> 1
                   inclog := inclog + 1
             RESULT inclog
          ) + wordlogbit
```

Table 1. OCCAM [I] process with multiple inlining in one line.

Notice in Table 1 that the inner ivp is an `expression_list` (with one `expression`) and the outer is an `operand`.

Transformation of ivp to ANY FUNCTION

The purpose of this note is to prove that any ivp can be transformed methodically to an ANY FUNCTION (possibly with other ANY FUNCTIONs in its `function_body`) that has no ivp in its `function_body`, and is therefore legal OCCAM according to [D]. In building to this, we will demonstrate a more generally useful technique: the use of generalizations about super-entities in proofs.

Lemma 1. An entity can be preceded at the same level of indentation by a **resource:** if and only if it is a **body** or **branch_entity**. This is recursive, because prepending the allowed **resource:** always results in an entity of the same kind.

Proof: This is proven by exhaustive search (see Appendix). `process` may be prepended by `specification` or `allocation`, and `valof` and the four **branch_entity** entities may be prepended by `specification` only, QED.

Lemma 2. Free (global) variables or values named in a **body** or **branch_entity** are defined either by a **resource:** prepended to that **body** or **branch_entity** at the same indentation, or are inherited from an enclosing **body** or **branch_entity** at a lesser indentation.

Proof (sketched): This is proven by exhaustive enumeration and semantic description of all the different **body** or **branch_entity** entities. See [I] p 75, **C.2 The rules for abbreviations**. This requirement is in fact essential to the nature of occam as a block-structured language with unique naming of resources, QED.

Note in this respect that the RESULT line in the `valof` is the only branch that is not an entity, cannot have prepended **resource:**s at the same indentation, and must therefore take all its globals from one level back, the `valof` itself.

Definition 1. The **ivp depth** of a point in a program in OCCAM-2 of [I] is the number of ivp entities enclosing that point in the program.

Thus it is 0 in the main part of the program, but as soon as you pass through an unmatched open parenthesis it rises by 1, and it falls by 1 when passing through an initial close parenthesis. (Notice that only tokens, not syntax, are needed to determine this.) A maximal continuous region of the code with ivp depth greater than or equal to a specified positive value comprises a `value_process = valof` and contains further ivps embedded in it if and only if its ivp depth rises above that value within that continuous region of the code.

Definition 2. In a maximal continuous region of the code of a program in OCCAM-2 of [I] with ivp depth greater than or equal to a specified value, an **effective line** is a maximal continuous region of the code ending with a NEWLINE RIND or a NEWLINE followed by zero or more ROUTD of that specified ivp depth.

It thus may include regions of higher ivp depth (ivps embedded in the line). An example in Table 1 is the entire section starting with `logbit :=` to the end of the code (ivp depth 0), and another example is the eight lines starting with `VAL INT tot` (ivp depth 1).

Definition 3. The **global place** of an effective line is the beginning of the smallest **body** or **branch_entity** (on the specified ivp depth) including that effective line.

In respect of Definition 3, it must be noted that a **body** or **branch_entity** that has **resource:**s at its head may expand, line by line, upward through that list of **resource:**s. When it has exhausted them (reached the top of the **resource:**s that are at the same indentation) it may only expand by jumping to a **body** or **branch_entity** that is indented less. *This is important because the effective line may be a single-line resource (terminating in a colon), or it may not.* In the two effective lines described above, notice the ivp depth 1 one is of the first type, while the ivp depth 0 one is not. If the effective line were a RESULT line, you would have to go all the way up to the VALOF to find its global place.

Theorem 1 (**Global place validity**): The free (global) variables or values available in an effective line are the same as the ones available in its global place.

Proof: By Lemma 2, at that ivp depth or less, all the **resource:**s preceding that effective line also precede its global place. Since indentation stays or increases between the global place and the effective line, all of them are still valid and unaltered at the effective line. Notice that this proof also passes from lower ivp depths to higher, since nothing goes out of scope when passing through the unmatched open parenthesis that separates them, QED.

Theorem 2 (**Conversion of ivp to ANY FUNCTION**): Any program of OCCAM-2 of [I] may be converted to OCCAM of [D] by taking its ivps in reverse order of depth, assigning each a name, replacing the ivp with

```
name ( )
```

at its location in its enclosing effective line, and inserting

```
ANY FUNCTION name ( )
  value_process
:
```

for its name and `value_process` just ahead of its global place at the same indentation.

Proof: Because the `program` is finite, there are only a finite number of ivp depths and ivps (of any depth) in that `program`. Select a name prefix that is never used in the original program, and order the ivps in reverse order of depth, giving each a unique name with that prefix and a number (see Table 2). If there are no ivps, we are finished. If there is at least one, then one of maximal ivp depth is embedded in a maximal continuous region of code of the maximal ivp depth less 1. This may be the main code, or it may be an enclosing ivp.

In either case, the operation described in Theorem 2 can be carried out. By Theorem 1 and the Lemmas, and the definition of an ANY FUNCTION (which is the same as a standard FUNCTION except its values, like those of an ivp, are figured out from the RETURN line instead of being explicitly specified in the headline), all the globals have the same meaning and validity as in the ivp being replaced. See Table 2 for an example.

```
INT logbit :
VAL INT wordlogbit IS 5 : -- 32 bit words
VAL [7]INT v IS [1,27,421,3,32309,36,4096] : -- list of sizes
ANY FUNCTION aa2()
  ANY FUNCTION aa1()
    INT sum :
    VALOF
      SEQ
        sum := 0
        SEQ i = 0 FOR SIZE v
          sum := sum + v[i]
      RESULT sum
  :
  VAL INT tot IS aa1() :
  INT inclog, totcopy :
  VALOF
    SEQ
      totcopy := tot
      inclog := 0
      WHILE totcopy <> 0
        SEQ
          totcopy := totcopy >> 1
          inclog := inclog + 1
    RESULT inclog
:
logbit := aa2() + wordlogbit
```

Table 2. OCCAM [D] process equivalent to Table 1.

By repeating this process until all the ivps of the highest ivp depth are replaced, it is assured that the ivps of the next highest depth (each considered independently as a `value_process`) are valid according to the OCCAM of [D]. Continue the process until all ivps of nonzero ivp depth have been converted, and the same holds true of the whole program, QED.

Conclusion: The Usefulness of Super-Entities

The super-entities introduced above were useful as a description that provided an easy proof of Theorem 2. This will not be their only use. The structure that is so general in form will also make it easy to write compilers based on this concept and the BNF of [D]. They will also include transpilers, for instance involving the Go language.

By adding new varieties fitting the super-entities, we will be able to extend the OCCAM language, both in ways already known (such as the configuration language of occonf) and in ways that add new capabilities, including block devices (attachables) and TCP-like network devices (detachables) that I described in my 2018 Workshop presentation. In addition, as already proven, it will be possible if desired to go to a "C-like" syntax by changing the ';' to ',' in the language definition and then using '{', '}', ';' to eliminate dependence on NEWLINE, RIND, and ROUTD. Everything will have the same "feel", because that "feel" is just the structure found in the super-entities.

Appendix: Full-Line OCCAM Entities

Notes on the following sets:

These are sets (each a union of equivalence classes) in the OCCAM language definition of [D] in a C-like BNF format. They satisfy the partial order inequalities

TOP > PROCESS > LINE > everything else in the language definition

PROCESS is a single equivalence class, while TOP and LINE are unions of equivalence classes.

Everything in PROCESS, TOP, and LINE is a full-line entity (ends in NEWLINE RIND or NEWLINE followed by zero or more ROUTD, and is initial in the program or preceded by another full-line entity). Nothing else in the language definition can include NEWLINE, RIND, or ROUTD. Note that this language definition applies after comments and preprocessor directives are removed and after continuation lines are united into a single line (called a "true line"), and that multi-line string constants are treated as

continuation lines. Note also that this language definition excludes inline value processes.

All occam programs satisfying the OCCAM-2 semantic definition of [I] as well as all occam programs as defined in [D] are composed solely of the constructs in this list. Only two substantive differences exist, and they are below TOP, PROCESS, and LINE: (a) The distinction between `andor` and `dyadic` is introduced in [D] to correct a bug in the BNF of [I] and make it consistent with the semantics; (b) the inline value process in [I] corresponds to the ANY FUNCTION in [D]. A full OCCAM-2 syntax consistent with the semantics of [I] can be created by adding to the definition of [D] the inline value process in the definition of `operand` and that of `expression_list`; but that destroys the last partial order inequality and places process and expression in the same equivalence class.

```
/* Set TOP */

program = block_definition_list

block_definition_list = block_definition
| block_definition_list
  block_definition

/* Set PROCESS */

alternation = ALT NEWLINE
                RIND alternative_vlist ROUTD
| ALT NEWLINE
| ALT replicator NEWLINE
    RIND alternative ROUTD
| PRI ALT NEWLINE
    RIND alternative_vlist ROUTD
| PRI ALT NEWLINE
| PRI ALT replicator NEWLINE
    RIND alternative ROUTD

alternative = guarded_alternative
| alternation
| specification
  alternative
| channel '?' CASE NEWLINE
    RIND variant_vlist ROUTD
| channel '?' CASE NEWLINE
| boolean '&' channel '?' CASE NEWLINE
    RIND variant_vlist ROUTD
| boolean '&' channel '?' CASE NEWLINE

alternative_vlist = alternative
| alternative_vlist
  alternative

block_definition = PROC name '(' formal_olist ')' NEWLINE
                     RIND procedure_body ROUTD
                   ':' NEWLINE
```

```
| data_type_clist FUNCTION name '(' formal_olist ')' NEWLINE
    RIND function_body ROUTD
  ':' NEWLINE

case_input = channel '?' CASE NEWLINE
               RIND variant_vlist ROUTD
| channel '?' CASE NEWLINE

choice = guarded_choice
| conditional
| specification
  choice

choice_vlist = choice
| choice_vlist
  choice

conditional = IF NEWLINE
                RIND choice_vlist ROUTD
| IF NEWLINE
| IF replicator NEWLINE
    RIND choice ROUTD

construction = sequence | conditional | selection | loop
| parallel | alternation

definition = PROTOCOL name IS sequential_protocol ':' NEWLINE
| PROTOCOL name NEWLINE
    RIND CASE NEWLINE
      RIND tagged_protocol_vlist ROUTD ROUTD
  ':' NEWLINE
| PROTOCOL name NEWLINE
    RIND CASE NEWLINE ROUTD
  ':' NEWLINE
| ANY FUNCTION name '(' ')' NEWLINE
    RIND function_body ROUTD
  ':' NEWLINE
| data_type_clist FUNCTION name '(' formal_olist ')' IS expression_list ':' NEWLINE
| specifier name RETYPES element ':' NEWLINE
| VAL specifier name RETYPES expression ':' NEWLINE
| block_definition

function_body = value_process

guarded_alternative = guard
                        RIND process ROUTD

guarded_choice = boolean NEWLINE
                   RIND process ROUTD

loop = WHILE boolean NEWLINE
         RIND process ROUTD


option = case_expression_clist NEWLINE
           RIND process ROUTD
| ELSE NEWLINE
    RIND process ROUTD
```

```
| specification
  option

option_vlist = option
| option_vlist
  option

parallel = PAR NEWLINE
             RIND process_vlist ROUTD
| PAR NEWLINE
| PAR replicator NEWLINE
    RIND process ROUTD
| PRI PAR NEWLINE
    RIND process_vlist ROUTD
| PRI PAR NEWLINE
| PRI PAR replicator NEWLINE
    RIND process ROUTD
| placedpar

placedpar = PLACED PAR NEWLINE
              RIND placedpar_vlist ROUTD
| PLACED PAR NEWLINE
| PLACED PAR replicator NEWLINE
    RIND placedpar ROUTD
| PROCESSOR expression NEWLINE
    RIND process ROUTD

placedpar_vlist = placedpar
| placedpar_vlist
  placedpar

procedure_body = process

process = SKIP NEWLINE
| STOP NEWLINE
| action | construction | instance | case_input
| specification
  process
| allocation
  process

process_vlist = process
| process_vlist
  process

selection = CASE selector NEWLINE
              RIND option_vlist ROUTD
| CASE selector NEWLINE

sequence = SEQ NEWLINE
             RIND process_vlist ROUTD
| SEQ NEWLINE
| SEQ replicator NEWLINE
    RIND process ROUTD

specification = declaration | abbreviation | definition
```

```
valof = VALOF NEWLINE
          RIND process
          RESULT expression_list NEWLINE ROUTD
| specification
  valof

value_process = valof

variant = tagged_list NEWLINE
            RIND process ROUTD
| specification
  variant

variant_vlist = variant
| variant_vlist
  variant

/* Set LINE */

abbreviation = specifier name IS element ':' NEWLINE
| name IS element ':' NEWLINE
| VAL specifier name IS expression ':' NEWLINE
| VAL name IS expression ':' NEWLINE

action = assignment NEWLINE
| input NEWLINE
| output NEWLINE

allocation = PLACE name AT expression ':' NEWLINE

declaration = type name_clist ':' NEWLINE

guard = input NEWLINE
| boolean '&' input NEWLINE
| boolean '&' SKIP NEWLINE

instance = name '(' actual_olist ')' NEWLINE

tagged_protocol = scalar NEWLINE
| scalar ';' sequential_protocol NEWLINE

tagged_protocol_vlist = tagged_protocol
| tagged_protocol_vlist
  tagged_protocol
```

References

[D] Dickson, Larry: "Details of Occam-2 Grammar Definition Development", Space Sciences Corporation, 6 July 2018. Presented at Transpiler Workshop, CPA2018, Dresden. https://github.com/SpaceSciencesCorp/Hard-Stuff-Language

[I] Inmos Ltd: occam 2 Reference Manual. Prentice Hall, 1988, document number 72 occ 45 01. http://www.transputer.net/obooks/isbn-013629312-3/oc20refman.pdf

[WP] Wikipedia: "Preorder", 7 March 2019. en.wikipedia.org/wiki/Preorder