ESPRIT Basic Research 3104

A (vW)- Grammar for Concrete PL_0 Syntax and Parser Correctness

Debora Weber-Wulff

Date: May 1990

Doc. Id.: Kiel DWW 2/2

Doc. type:	Note
Status:	Draft
Activity-alloc:	Compiler Implementation
Distribution:	All **ProCoS** staff
Copyright © 1996	Kiel University

Document History

This is the correction of the first version.

Document Purpose

This document is concerned with the problem of parsing OCCAM-like languages, of automatically producing such a parser, and with the correctness of such a parser.

Project Sponsor

This report reflects work which is partially funded by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Action proj. no. 3104: "**ProCoS**: Provably Correct Systems"

Site Address

Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Preußerstraße 1–9
D-2300 Kiel 1
West Germany
Tel: +49-431-5604-28
Fax: +49-431-5661-43
Telex: 292656 cauki d
E-mail: procos@causun.uucp

Partners

Technical University of Denmark (ID/DTH)
Christian-Albrechts Universität (Kiel)
Oldenburg Universität (OLD)
Oxford University (OU)
Royal Holloway and Bedford New College (RHC)
Århus University (DAIMI)
University of Manchester (MU)

This document proposes a grammar for concrete PL_0 as a proper subset of OCCAM, and suggests a method for transforming conctete PL_0 trees into abstract PL_0 trees, which are formally specified according to the abstract syntax proposed by [ID/DTH HHL 2].

The goal of the parsing portion of the compiler is not just to write a parser for PL_i languages and prove that correct, but to find a formalism for describing the grammars at each level, so that a parser to construct the concrete syntax tree and a transformer to construct the corresponding abstract tree from that description can be generated. It must then be proved that the parser recognizes the language intendend and that the transformer can transform all correct concrete trees into corresponding abstract ones, and that all abstract trees have a concrete representation.

1 Problems with a Concrete Syntax for PL_0

It was decided at the Oxford meeting to have the concrete syntax of the PL_i be proper subsets of OCCAM, so that we can run proper OCCAM programs when (and if) we reach Level 2. This causes some problems for the parser, as OCCAM is not easy to parse or to define for a parser genrerator.

We used [inm88] for a description of the language, which mixes formal language definition methods with informal ones, especally as regards the indentation concept, and the problem of what white space is and where exactly it is allowed.

1.1 Indentations

One of the most visible characteristics of OCCAM programs is the strict indentation structure of the concrete programs. The language designers dispended with explicit markers for block begin and end (like BEGIN/END or {/} or IF/FI), and instead decided to define the block structure only on the basis of the indentation level.

This decision seems to have been made with the folding editor [TDS87] in mind. This structured editor, so called because blocks can be folded away along a comment, can easily determine the current block using the cursor column counter. Programs seem to be stored in abstract form and are unparsed to display on the screen. Counting, however simple to do in an editor, is something that finite state automata, which are used in parsing to recognize legal source programs, cannot do.

It would be possible to write a finite context-free grammar (CFG) for OCCAM if there were no continuation lines available. Since the editor only has a finite number of columns available (80), there are 40 possible indentation levels. A LR(40) grammar (using a 40 token lookahead) could be theoretically constructed, although a tremendous amount of table space would be necessary to store the tables, with much of the information the same, except for the level involved.

It is also not possible to use attributed grammars for counting this level, as there is a problem which occurs when a number of sequential constructs are terminated at once. An example is

```
INT hugo:
INT emil:
SEQ
  hugo := 1
  emil := 2
  SEQ
    hugo := hugo + emil
    SEQ
      emil := emil + hugo
      WHILE hugo < 100
        SEQ
          hugo := hugo + 1
          emil := hugo DIV 2
  hugo := emil * emil
```

In this example, it is only by virtue of the number of indentations that `hugo := emil * emil` belongs to the outer block and is not part of the loop process. It is not enough to determine that the next construct

is a process and add it on to the sequence, but the level has to be determined, and if necessary more than one level must be terminated.

By splitting the parsing into scanning and parsind, and taking care of indentations in the scanning, a context-free grammar for PL_0 can be written. However, scanners can normally be expressed formally with regular expressions, which are easier to implement efficiently than context-free grammars. This would push the problem of formalization of the indentation structure to another phase of the compiler, the problem doesn't go away using this technique.

It is possible, however, to write a two-level or van Winjgaarden grammar (TLG or vWG) for OCCAM, as these grammars use metavariables and hyperrules to generate an infinite number of productions. However, for OCCAM-like languages we do not need the full power of these grammars. It suffices to use one metavariable for counting the current indentation level. This is a similar concept to adding one call-by-value parameter to functions in an otherwise parameterless system of recursive functions. vW-Grammars and the restricted version used here will be described in detail below.

1.2 White Space

White space presents a problem in OCCAM, as a subset of the "normal" white space set $(\sqcup$ + CRLF$)^*$ is significant, namely the indentations. In most languages, all blanks and line end markers are ignored or collapsed into just one blank, which serves to separate tokens. If we were to make a separate pass for scanning in PL_0 we could not determine what action to take if we encounter a blank - it depends upon whether we are at the beginning of a "line", which would make it part of an indentation, or in a type declaration, action or while statement. As far as the OCCAM definition goes, it seems that blanks are allowed after keywords and before carriage returns.

The solution is to specify non-terminals eoln, blank (at least one blank) and blanks (any number of blanks, including 0), and include them in the grammar at the appropriate places. This expands the grammar, but seems necessary in order to deal with this "layout problem" in a formal manner.

Continuations can also be seen as white space. The definition in the OCCAM 2 Reference Manual is :

> A long statement may be broken immediatly after one of the following
>
> | | |
> |---|---|
> | an operator | `+`, `-`, `*`, ... |
> | a comma | `,` |
> | a semi-colon | `;` |
> | assignment | `:=` |
> | the keywords | `IS`, `FROM` and `FOR` |
>
> A statement can be broken over several lines providing the continuation is indented at least as much as the first line of the statement.

Comments are also considered white space. They are introduced by a double dash symbol (`--`) and extend to the end of line. Since the end of line is a necessary non-terminal in the grammar, it has to include comments.

A note is necessary here on the problem of using a standard editor to write PL_0 programs. Most standard editors "save space" when storing files to disk : they convert a sequence of blanks into a sequence of tabs and blanks. This makes more work for the parser, which has to guess the tab expansion algorithm used by the editor. This may not be easy to prove! It boils down to the editor not working "correctly", i.e. not storing the text as expected from the screen picture. Perhaps we should develop our own editor as well, perhaps even develop and prove correct a folding editor that stores a concrete representation of the abstract syntax! The proof of correctness would probably not be possible, however.

1.3 Lookahead for Action

Even if we didn't need to worry about indentations, a grammar for PL_0 is not even LR(1) unless the grammer is transformed by means of the lookahead reduction. The problem is with the action statements. The first component of each action producion is a variable, we need another lookahead to determine which action is intended. This is not difficult to solve by hand - check first if the lookahead is a construction, if not then get another lookahead before deciding how to reduce. But our goal is to formalize this so that a parser can be generated, not fixed by hand. There are, however, well-known transformations that can be performed on LR(2) grammars to transform them into LR(1) ones, so this is not too severe a problem

We could perhaps also force this by liniarizing the specification of the grammar and determining that the rules are tried in the order in which they were specified.

2 A Grammar for Concrete PL_0

The following section introduces the concept of a restricted vW-Grammar and defines the elements of a grammar for PL_0.

The concrete syntax for PL_0 cannot be represented by a finite context-free grammar, because of the indentation problem mentioned above. A van Wijngaarden or two-level grammer could, however, be used to finitely represent a context-free grammar with an infinite number of productions. The following definitions for a two-level grammar are from [Weg80].

Definition 1 A two-level grammar (TLG, 2VWG, van Wijngaarden grammar, W-grammar) is an ordered 7-tuple (M, V, N, T, R_M, R_V, S) where

M is a finite set of *metanotions*

V is a finite set of *syntactic variables*, $M \cap V = \emptyset$

$N = \{ < H > \mid H \in (M \cup V)^* \}$ the finite set of *hypernotions*

T is a finite set of *terminals*

R_M is a finite set of *metaproduction rules* $X \rightarrow Y$ where $X \in M$, $y \in (M \cup V)^*$
s.t. $\forall W \in M$: (M, V, R_M, W) is a context-free grammar

$R_V \subseteq N \times (N \cup T)^*$ the finite set of *hyperproduction rules*

$$< H_0 > \rightarrow\ h_1 h_2 \ldots h_m,\ h_i \in (T \cup N \cup \{\epsilon\}),\ (1 \leq i \leq m)$$

$S = < S > \in N$, $S \in V^+$ the *start notion*

These grammars can be thought to contain a "skeleton" CFG that can be varied over the hypernotions to obtain an infinite number of productions.

Definition 2 Given a TLG G + (M, V, N, T, R_M, R_V, S) we define the set R_S of *strict production rules* of a hyperproduction rule r=($< H_0 \rightarrow\ h_1 h_2 \ldots h_m$) containing the $n \geq 0$ metanotions $W_1, W_2, \ldots, W_n$ as follows:

$R_S(r) = \{ < \varphi(H_0) > \rightarrow\ \varphi(h_1), \varphi(h_2), \ldots \varphi(h_m) \mid$

φ is a homomorphism with $\varphi(v) = v$) for $v \in V \cup \{<, >\}$
and $\varphi(W_i) \in L((M, V, R_M, W_i))$, $\varphi(H_0) \neq \epsilon$ }.

Definition 3 If G = (M, V, N, T, R_M, R_V, S) is a TLG then the *language* specified by G is L(G) = $\{x \in T^* \mid S \stackrel{*}{\Longrightarrow}_G x\}$, where $\stackrel{*}{\Longrightarrow}_G$is the reflexive and transitive closure of the binary relation $\Longrightarrow_G$in ($N_S \cup T$)*which is defined by

$$X \Longrightarrow_G Y \text{ iff } \exists P, Q \in (N_S \cup T)^* : x = PX'Q and Y = PY'Q$$

and $X' \rightarrow Y' \in R_S(r)$ for some $r \in R_V$.

$X \Longrightarrow_G Y$ is called a *derivation step*

These grammars have the capability of "counting", something that finite-state automata, which can recognize words in languages that can be described with finite context-free grammers, can't. A typical example for a language that can be represented with a TLG and not with a CFG is

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}$$

.

However, we do not need the full power of vW/TLG parsing for PL_0. As the indentations are the only part of the grammar which cause an infinite number of productions to be generated, it would suffice to restrict the grammar to include one metanotion, the notion of level.

A production of the form

$$< A >^{(n)} \rightarrow < B >< C >^{(n+1)}$$

is meant to express that a production A on indentation level n can be recognized when B is followed by C on indentation level n+1. This type of grammer will be called a *restricted vW-grammar*.

2.1 Set of terminals for PL_0

The following terminal symbol set and denotations are defined for $LEVEL_0$[1]:

T = { < blank >, ..., < close_par > }

< blank >	::=	$\sqcup^+$
< blanks >	::=	$\sqcup^*$
< eoln_sy >	::=	`#0D0A`
< colon_sy >	::=	`:`
< comment_sy >	::=	`--`
< char >	::=	{ 'a' .. 'z', 'A' .. 'Z' }
< digit >	::=	{ '0' .. '9' }
< special >	::=	{ '.' }
< printing_char >	::=	{ '$\sqcup$' .. ' '
< indentation >	::=	$\sqcup\sqcup$
< continuation >	::=	* #0D0A *
< channel >	::=	input | output
< typeid >	::=	INT32
< assign_sy >	::=	:= < blanks >
< out_sy >	::=	!
< in_sy >	::=	?
< skip_sy >	::=	SKIP
< stop_sy >	::=	STOP
< seq_sy >	::=	SEQ
< if_sy >	::=	IF
< while_sy >	::=	WHILE
< true_sy >	::=	TRUE
< false_sy >	::=	FALSE
< plus_sy >	::=	+
...		
< or_sy >	::=	OR
< open_par >	::=	(
< close_par >	::=	)

[1] Still the **L**eading **E**uropean **VE**rified **L**anguage

2.2 Set of Hyperproductions for PL_0

The set of hyperproductions is constructed using the set of syntactic variables (the non-terminals in a CFG) and the metanotion of level, denoted $^{(n)}$ or $^{(n+1)}$. The notation $\{<\text{indentation}>\}_n$ is used to signify n copies of < indentation >, $[<\text{hypernotion}>]^*$is used to signify 0 or more copies of < hypernotion >.

V = { < comment >, ... , < c_dop > }
R_V = {

< comment >	→	< comment_sy > [< printing_char >]*< eoln_sy >
< eoln >	→	< blanks > < eoln_sy >
	|	< blanks > < comment >
< c_prog >	→	< c_block >
< c_block >	→	< c_type_decl > | < c_process >$^{(0)}$
< c_type_decl >	→	< c_type > < c_var > < colon_sy > < eoln > < c_block >
< c_var >	→	< c_identifier >
< c_identifier >	→	< char > [< char > + < digit > + < special >] *
< c_type >	→	< type_id >
< c_channel >	→	< channel_sy >
< c_process >$^{(n)}$	→	< indent >$^{(n)}$< c_action >$^{(n)}$< eoln >
	|	< indent >$^{(n)}$< c_construction >$^{(n)}$
< indent >$^{(n)}$	→	$\{<\text{indentation}>\}_n$
	|	$\{<\text{indentation}>\}_i$ < continuation > $\{<\text{indentation}>\}_j$, $i+j=n$
< c_action >$^{(n)}$	→	< c_var > < assign_sy > < cont >$^{(n)}$< c_expr >$^{(n)}$
	|	< c_channel > < in_sy > < var >
	|	< c_channel > < out_sy > < c_expr >$^{(n)}$
< c_construction >$^{(n)}$	→	< seq_sy > < eoln > [< c_process >$^{(n+1)}$]*
	|	< if_sy > < eoln > [< c_gc >$^{(n+1)}$]*
	|	< while_sy > < blank > < c_expr >$^{(n)}$< eoln > < c_process >$^{(n+1)}$
< c_gc >$^{(n)}$	→	< c_expr >$^{(n)}$< c_process >$^{(n+1)}$
< c_expr1 >$^{(n)}$	→	< c_const > | < c_var > | < open_par > < expr >$^{(n)}$< close_par >
< c_expr >$^{(n)}$	→	< expr1 >$^{(n)}$
	|	< mop >$^{(n)}$< cont >$^{(n)}$< expr1 >$^{(n)}$
	|	< expr >$^{(n)}$< dop > < cont >$^{(n)}$< expr1 >$^{(n)}$
< cont >$^{(n)}$	→	ϵ | < eoln > < indent >$^{(n)}$< blanks >
< c_const >	→	< c_number >$^+$
< c_mop >	→	< minus_sy > | < not_sy >
< c_dop >	→	< plus_sy > | ... | < or_sy >

}

3 Abstract Syntax for PL_0

This is a META-IV representation of the abstract systax given in [ID/DTH HHL 2]. We write block for blk and process for p, as it makes the meaning of the constructs clearer to have proper names. This is the abstract syntax upon which the compiling specification is built [Kiel MF 4]. One slight change was made to the block domain, which is a union of a tree and a domain. This was split to include a new level, called the type_decl.

prog	::	s_block	: block
block	=	type_decl	| process
type_decl	::	s_type	: type
		s_var	: var
		s_block	: block

```
var       =  identifier
type      =  identifier
channel   =  INPUT | OUTPUT
process   =  assign | input | output | skip | stop | cond | loop | seq
assign    ::  s_var     : var
              s_expr    : expr
input     ::  s_channel : channel
              s_var     : var
output    ::  s_channel : channel
              s_expr    : expr
skip      ::  s_stmt    : skip_sy
stop      ::  s_stmt    : stop_sy
cond      ::  s_gc      : gc *
gc        ::  s_guard   : expr
              s_process : process
seq       ::  s_process : process *
expr      =  const | var | mexpr | bexpr
const     =  number
mexpr     ::  s_op      : mop
              s_expr    : expr
bexpr     ::  s_op      : bop
              s_lexpr   : expr
              s_rexpr   : expr
mop       =  minus_sy | not_sy
bop       =  plus_sy ... or_sy
```

4 Generating a Parser

In order not to have to write and prove a parser for each level of the **ProCoS** project, we want to produce a parser generator that will produce a parser which is provably correct. This parser generator will use grammars of the type mentioned above. Parser generating systems which are widely available today cannot be used, as they require finite context-free grammars, and often can only generate parsers for certain, well-defined subsets of the CFGs.

In order to be able to use and test the compiler from an early stage, we will make a simple, non-proven parser available, that can parse a concrete representation of the abstract syntax into abstract trees. The concrete representation will be LISP s-expressions. When the parser generator is finished, we will then discard the s-expression parser, and be able to enter programs in concrete PL_0.

4.1 Parsing Algorithm

Since the compiler is to be written in a highly recursive language (SUBLISP [Kiel HL 2]), is is natural to try and generate a recursive descent parsing algorithm. Each production becomes a function, void of parameters except for the level. At the end of each, when the concrete construct has been recognized, a parse tree portion is either returned, and the tree is transformed after the entire tree has been constructed, or transformation work is done before the subtree is returned.

4.1.1 Scanner

Scanners normally take a set of regular expressions as input (called the microsyntax) which define the set of tokens to be recognized. Each regular expression is given a name, and most describe disjoint portions of possible input streams. When overlap takes place, for example when ':' and ':=' are both considered tokens, the principle of the longest match is applied. An order is given to the names of the regular expressions, normally in descending order of length, and the first one to match is assumed to be the token.

It is not possible to write a regular expression for describing the token microsyntax for PL_0. It depends on the state of the parser, whether two blanks in sequence are to be interpreted as indentations or as white space. Thus, the scanner will have to be called from the parser, with information passed to it about the current state.

4.1.2 Parser

The parser itself will probably be LL(1), attempting to find a leftmost derivation from the start notion, < prog >. A set of recursive function definitions, corresponding to the hyperproductions, will attempt to trace out the concrete syntax tree. These functions are generated to recognize elements of the concrete syntax.

For each domain in the abstract syntax definition, an abstract data type can be generated [ScH90]. When a production in the concrete syntax is reduced, a new object of the concrete type is created and the elements of the tree are computed as attributes for attribute grammars. This will only work if there are no cycles in the computation, this has not be checked through yet.

An example of a recognition function (in Pascal) could be:

```
FUNCTION recognize_c_gc (level : integer) : c_gc;
(* This function returns a concrete guarded choice subtree if
a concrete expression and a concrete process are recognized *)
VAR gc      : c_gc; (* abstract data type *)
    expr    : c_expr;
    process : c_process;
BEGIN
 expr := recognize_c_expr (level);

 IF NOT is_empty_c_expr (* c_expr was recognized *)
    THEN BEGIN
       process := recognize_c_process (level + 1);
       IF NOT is_empty_c_process
  THEN gc := mk_one_c_gc (expr, process)
  ELSE gc := mk_empty_c_gc;
    END
    ELSE gc := mk_empty_c_gc;


 recognize_c_gc := gc;
END; (* recognize_c_gc *)
```

This recognize function does not need the scanner, but in the expression recognizer the lookahead would have to be determined and new tokens requested from the scanner until the construct expr can be satisfied.

4.1.3 Transformer

There are two ways to go in constructing the abstract tree: either the concrete tree is completely constructed and a transformer is generated to traverse this tree and produce the abstract tree, or the abstract tree is constructed directly at the time a production is reduced.

If an abstract tree is to be constructed directly, there must be a strong correspondence between the concrete and the abstract syntax, so that the recognize functions recognize concrete constructs and return abstract ones. Only certain types of transformations would be possible. The following types of tree transformation can take place:

- pruning

  When a concrete tree is to be *pruned* into an abstract one, some of the branches of the concrete tree are left out. An example is the type declaration. The concrete production

$< \text{c\_type\_decl} > \rightarrow < \text{c\_type} > < \text{c\_var} > < \text{colon\_sy} > < \text{eoln} > < \text{c\_block} >$

contains two syntactic elements, $<$ colon_sy $>$ and $<$ eoln $>$, which are absent in the abstract syntax:

LET td = mk_type_decl (type, var, block)
IN ...

- grafting

  Grafting would be necessary for a language which needed information, perhaps a synthesized attribute, in an abstract syntax construct that has no direct companion in the concrete systax. This information would just be inserted at the appropriate place. Grafting is also necessary when the abstract syntax is to be output in a concrete form, for example as LISP lists. It would be necessary to insert parentheses at appropriate places in the abstract syntax tree.

- shortening

  A concrete tree is *shortened* when a length is removed from a branch. An example of this would be in an operator precedence grammar, where terms and factors are used to force priorities. When the priorities are clear, the term and factor productions can be removed, leaving the identifier or constant leaves connected directly to the operator nodes.

- permuting

  Permutation takes place when the abstract syntax requires the subtrees in a different order than in the concrete syntax. An example would be the 3 different actions: in order to differentiate the 3 branches, a LISP-like abstract syntax could look like this:

  (ASSIGN $<$ var $>$ $<$ expr $>$)
  (INPUT $<$ channel $>$ $<$ var $>$)
  (OUTPUT $<$ channel $>$ $<$ expr $>$)

  Transforming concrete $<$ var $>$ $<$ assign_sy $>$ $<$ cont $>^{(n)}$ $<$ expr $>^{(n)}$ would involve pruning the continuation and permuting the assignment tag to the first branch.

- identifying

  Identifying common subtrees would be an optimization step that could be taken. Only one node for the subtree would be generated, with different parent nodes pointing to it or multiple links from a single node.

Since there is a strong correspondence between the concrete and abstract syntax, we will probably do the transforming during parsing, if this does not get in the way of the correctness proof of the parser.

5 Proving the Parser Correct

We have to prove that all legal concrete trees have legal abstract representations, and that all legal abstract trees have at least one concrete representation (there could be more than one, since expressions can be put in parentheses to force precedence). There is, however, no precedence of operators is OCCAM as operators are taken from left to right.

There is also an obligation to demonstrate that the language recognized by the parser is exactly the same one as is defined by the concrete grammer. It might, in our case, also be necessary to show that the language defined is a proper subset of OCCAM. The latter task, in absence of a formal definition of the concrete syntax of the language, is difficult.

Proving that the language recognized by a parser is the same as the one defined by a grammar involves computing the reflexive and transitive closure of the relation $\stackrel{*}{\Longrightarrow}_G$ defined above to determine the language

of the grammar, and then finding the parser language by taking the parser to be a finite-state automaton and computing the words it recognizes, and then comparing the results.

Since most of the interesting properties in automata and formal language theory are undecidable, this is probably also undecidable.

References

[Deu78] Deussen, P. , Wegener, L., A bibliography of the van Wijngaarden grammars, Bulletin of the European Association of Theoretical Computer Science (ETACS) 6, 1978

[Kiel MF 4] Fränzle, Martin Compiling Specification for **ProCoS** Programming Language Level 0, **ProCoS** Technical Report Kiel MF 4 Version 3, 20 Apr 1990

[Gru84] Grune, D., How to produce all sentences from a two-level grammar, *Information Processing Letters*, 19 (1984), pp. 181-185

[Kiel HL 2] Langmaack, Hans Note on the ProCoS Compiler Development Diagramm for Language PL_0 20 Feb 90 Version 2 (correction of Version 1)

[ID/DTH HHL 1] Løvengreen, Hans Hendrik, Note on the **ProCoS** Programming Language, **ProCoS** Technical Report, DTH HHL 1, Draft 1.1, 14 July 1989

[ID/DTH HHL 2] Løvengreen, Hans Hendrik, Definition of the **ProCoS** Programming Language Level 0, **ProCoS** Technical Report DTH HHL 2, Draft 1.1, October 25, 1989

[ID/DTH KMJ 2] Jansen, K. M., Løevengreen, H.H., *Abstract Syntax for* OCCAM 2, **ProCoS** Technical Report DTH KMJ 2, Version 1.0, 12 Oct 1989

[inm88] INMOS, Ltd., OCCAM 2 *Reference Manual* Prentice Hall International, 1988

[ScH90] Schmidt, Uwe, and Hörcher, Hans-Martin, Programming with VDM-Domains, in: *VDM & Z! : Proceedings of the Third International Symposium of VDM Europe, Kiel*, LNCS ???, Springer Verlag, Heidelberg, 1990

[TDS87] Transputer Hardware Support, Ltd., Northwich, England, *Transputer Development System 2.0 : Getting Started*, Issue 1, June 1987

[Weg80] Wegner, Lutz Michael, On Parsing Two-Level Grammars, in: *Acta Informatica* 14, p. 175-193, 1980