WORKING NOTE AC72

Alan Creak
21 June 1989

PFL SYNTAX

This document records a new version of the PFL syntax, started in 1984, but for the most part devised after Peter Sergent finished his Ph.D. work[1]. It is better structured than Peter's version, and I believe significantly improved in some respects because it benefits from the experience of Peter's work.

WHY AM I WRITING THIS NOTE ?

A recent resurgence of interest in PFL has encouraged me to believe that it isn't actually dead. I'm therefore dressing up this syntax definition in marginally respectable, and referenceable, form, and offering it for discussion.

IMPROVEMENTS.

There are two main changes, which I deem to be improvements, in this definition as compared with Peter's original design. They are :

- The idea of the **machine** declaration, which collects together all the material pertaining to a machine in the physical sense. The most general abstraction in the original definition was the **procedure**, semantically very similar to the traditional Algol procedure, and all required abstractions had to be mapped onto that. A **machine** is much more like a Simula class (or, maybe better, and Ada package), and can contain processes of various sorts, data structures, and code; this seems to be a much better model for the required description of a physical machine, which may have a complex description, several different sorts of operating procedure, special instructions for starting up and shutting down, and so on. The provision of a **machine** abstraction also makes it much easier to deal with multiple machines of the same type.

- The clear distinction between the parts of a **machine** – the description of its interface, the components of the **machine**, procedures for handling special conditions, and normal operating instructions. This gives a much clearer structure to the code, and in principle provides information to the PFL system which it can use in automatically disposing of the various components.

These brief accounts are intended only to give the flavour of the matters considered, and to suggest, rather than justify, my trains of thought. Much of this material can be expanded significantly if it seems to be a good idea; I have a lot of notes on the language design, which may be worth transcribing into legible form if anyone's interested. (Though not all the notes are really about this syntax;

SYNTAX NOTATION.

The syntax notation is based on colloquial BNF as commonly used, but extended in particular to permit controlled selections from sets of items. The main reason for this provision is to make it possible to describe languages in which some parts are to be regarded as declarative rather than procedural, so that the order of the components is not important – so that, for example, the descriptions of different attributes of a data structure can be given in any order – but there is some control over the number of occurrences of components of various sorts – so that we can insist on exactly one type specification and not more than one initial value. Imposing such conditions amounts to introducing a degree of context-dependency into the notation, and most versions of BNF, plain or extended, don't extend much in that direction.

WHAT HAPPENED TO WORKING NOTE AC49 ?

I wrote the Note in question[2] in 1986. It contains a few sample "PFL programmes", or bits thereof. By and large, they don't conform to the syntax presented here. One might well ask why.

There is no very deep explanation for the difference in notation. When I wrote the Note, I had for some time done very little work on the PFL compiler I was writing, partly because of pressure of other commitments, but also because I had some doubts as to the way the compiler and the language were developing. The note itself was to summarise and clarify my current thoughts, and to provide a sample of them which I could discuss with George Blanchard. It didn't seem too important at the time to stick to the precise syntax, because my concern was with the broader structures of the language; and my work on the compiler hadn't given me any experience of the language as a whole. So I guessed, and got a number of things wrong.

REFERENCES.

1 : P.A. Sergent-Shadbolt : *A new computer language for process control* (Ph.D. Thesis, Auckland University, 1985).

2 : G.A. Creak : *PFL : progress report ? …* (unpublished Working Note AC49, 1986).

THE SYNTAX SPECIFICATION.

```
% PFL FORMAL SYNTAX.              31-x-1984
%
% NOTATION :
%
%    < ... >                 A non-terminal symbol.
%
%    AWORDINCAPITALS         A terminal symbol.
%
%    Any symbol not defined here
%                            Itself.
%
%    ::=                     Produces.
%
%    =>                        There follows a note in English on the implications of
finding
%                                  the production.
%
%    |                       Or.
%
%     |> s | A1 | A2 ... <|   A sequence of one or more of An, without duplication
( unless
%                                    indicated by |* or |+ ), in any order, separated
if need
%                                  be by the mark s.
%
%    |*                      Zero or more of whatever follows.
%
%    |+                      One or more of what follows.
%
%    |!                      At most one of what follows.
%
%    [ ... ]                 Optional.
%
%    %                       Following text to the end of the line is a comment.
%
```

```
<pflprogramme>
     ::=  <identifier> [IS] [A] PROGRAMME [WITH] < programmeparts>
                                 END [ [OF] <identifier> ] .
               =>   <identifier> names a programme;
                    both <identifier>s are the same.
<programmeparts>
     ::=  |> . | <image>
               | <components>
               | <startup>
               | <shutdown>
               | <emergency>
               | <operations>
          <|
%
<image>
     ::=  IMAGE : <imagedetails>
<imagedetails>
     ::=  |> ; |* <linedetails>
          <|
<linedetails>
     ::=  <identifier> [IS] <linedescription>
               => <identifier> names a line.
<linedescription>
     ::=  |> , | <linewidth>
               | <linecontinuity>
               | <lineconditionnames>
          <|
<linewidth>
     ::=  SINGLE
     |    BYTE
     |    CHANNEL <expression>
               =>   <expression> evaluates to give an integer.
<linecontinuity>
     ::=  INTERRUPT
     |    CONTINUOUS
<lineconditionnames>
     ::=  |> , | ON [IS] <identifier>
               =>   <identifier> names a line state.
               | OFF [IS] <identifier>
               =>   <identifier> names a line state.
          <|
%
<components>
     ::=  COMPONENTS : <componentspart>
<componentspart>
     ::=  |> , |* <identifier>
               =>   <identifier> names a machine or procedure.
          <|
%
<startup>
     ::=  STARTUP : <identifier>
               =>   <identifier> names a procedure.
%
<shutdown>
     ::=  SHUTDOWN : <identifier>
               =>   <identifier> names a procedure.
%
<emergency>
     ::=  EMERGENCY : <identifier>
               =>   <identifier> names a procedure.
%
<operations>
     ::=  OPERATIONS : <operationsbody>
<operationsbody>
     ::=  |> \ |*  <sentencesequence>
          <|
<sentencesequence>
     ::=  |> ; |* <sentence>
          <|
% <sentence>s within <sentencesequence>s must be executed serially;
% different <sentencesequence>s may be executed in parallel.
```

```
<sentence>
     ::=  <declaration>
     |    <instruction>
%
% Declarations are available, but not syntactically required; in that
respect,
% the syntax is interpreted literally. The system is supposed to infer the
type
% of any object from its context.
<declaration>
     ::=  <datadeclaration>
     |    <proceduredeclaration>
     |    <machinedeclaration>
<datadeclaration>
     ::=  <identifierlist> <propertieslist>
<identifierlist>
     ::=  |> , |* <identifier>
               => <identifier> names a variable of type T.
<propertieslist>
     ::=  |> , | [IS] <typedetails>
               => Type T is defined by <typedetails>.
               | <- <constant>
               => Type T is the type of the <constant>.
          <|
<typedetails>
     ::=  <datatype>
     |    FILE [OF] <datatype>
<datatype>
     ::=  <simpletype>
     |    ARRAY [OF] <expression> <simpletype>
               =>   <expression> evaluates to an integer.
<simpletype>
     ::=  CHAR
     |    INTEGER
     |    LOGICAL
     |    NUMBER
     |    STRING
%
<proceduredeclaration>
     ::=  <identifier> [IS] AN OPERATION [WITH] <procedureparts>
                                END [ [OF] <identifier> ]
               =>   <identifier> names a procedure;
                    both <identifier>s are the same.
<procedureparts>
     ::=  |> . | <inputlist>
               =>   the procedure uses input parameters.
               | <outputlist>
               =>   the procedure uses output parameters.
               | <valuespecification>
               =>   the procedure is a function.
               | <startup>
               | <shutdown>
               | <emergency>
               | <operations>
          <|
<machinedeclaration>
     ::=  <identifer> [IS] [A] MACHINE [WITH] <machineparts>
                                END  [ of <identifier> ]
               =>   <identifier> names a machine;
                    both <identifier>s are the same.
<machineparts>
     ::=  |> . |! <image>
               | <components>
               |! <startup>
               |! <shutdown>
               |! <emergency>
               | <operations>
          <|
<inputlist>
     ::=  USING |> , |+ <identifier> <|
               =>   the number of input parameters, and their order,
                    are known.
```

```
<outputlist>
    ::= GIVING |> , |+ <identifier> <|
              =>   the number of output parameters, and their order,
                   are known.
<valuespecification>
    ::=  RETURNING <simpletype>
              =>   the type of the function is known.
%
<instruction>
     ::=  <compoundinstruction>
     |    <conditionalinstruction>
     |    <iterativeinstruction>
     |    <simpleinstruction>
<compoundinstruction>
     ::=  GROUP <operationsbody> [ ; ] END
<conditionalinstruction>
     ::=  IF <expression> THEN <instruction> [ ELSE <instruction> ]
                                END IF
              =>   <expression> evaluates to give a logical value.
<iterativeinstruction>
     ::=  REPEAT <iterationcontrol> : <instruction> END REPEAT
<iterationcontrol>
     ::=  |> : | WHILE <expression>
              =>   <expression> evaluates to give a logical value.
              | UNTIL <expression>
              =>   <expression> evaluates to give a logical value.
              | [ up to ] <expression> TIMES
              =>   <expression> evaluates to give an integer expression.
              | FOR EACH <identifier>
              =>   <identifier> names an array.
          <|
<simpleinstruction>
     ::=  <assignment>
     |    <procedurecall>
     |    <stopinstruction>
     |    <returninstruction>
     |    <hearinstruction>
     |    <sayinstruction>
%
<assignment>
     ::=  <identifier> <- <expression>
              =>   the types of <identifier> and <expression> are
                   the same.
<procedurecall>
     ::=  CALL <identifier> [ <actualparts> ]
              =>   <identifier> names a procedure which is not a
                   function.
<actualparts>
     ::=  |> , | USING |>, |+ <expression> <|
              =>   information on the input parameters.
              | GIVING |> , |+ <expression> <|
              =>   information on the output parameters.
          <|
<stopinstruction>
     ::=  STOP
<returninstruction>
     ::=  RETURN <expression>
              =>   the current scope corresponds to a function;
                   the type of <expression> is the same as that of
                   the function.
<hearinstruction>
     ::=  HEAR <identifier>
              =>   <identifier> names a string.
<sayinstruction>
     ::=  SAY <expression>
              =>   <expression> evaluates to give a string.
%
```

```
<expression>
     ::=  <term>
     |    <term> <binaryarithmeticoperator> <term>
               =>   both <terms> are numeric values.
     |    <term> <binaryrelationaloperator> <term>
               =>   both <term>s are numeric values, or both are string;
                    the <expression> is logical.
     |    <term> <binarylogicaloperator> <term>
               =>   both <term>s are logical;
                    the <expression> is logical.
     |    <term> & <term>
               =>   both <term>s are strings;
                    the <expression> is a string.
<binaryarithmeticoperator>
     ::=  **
     |    *
     |    \
     |    +
     |    -
<binaryrelationaloperator>
     ::=  >
     |    <
     |    =
     |    >=
     |    <=
     |    <>
<binarylogicaloperator>
     ::=  AND
     |    OR
<term>
     ::=  <primary>
     |    – <primary>
               =>   the <primary> is numeric;
                    the <term> is numeric.
     |    NOT <primary>
               =>   the <primary> is logical;
                    the <term> is logical.
     |    # <primary>
               =>   the <primary> is a string;
                    the <term> is numeric.
     |    $ <primary>
               =>   the <primary> is numeric;
                    the <term> is a string.
<primary>
     ::=  <constant>
     |    <identifier>
     |    ( <expression> )
     |    <identifier> [ <actualparts> ]
               =>   <identifier> names a function;
                    the function type and <primary> type are the same.
%
<constant>
     :=  a number
     |    a string between quotation marks
     |    TRUE
     |    FALSE
<identifier>
     ::=  any string of letters and digits beginning with a letter which
          is not a <constant> nor a reserved word.
```