P4 4.0 Technical Description

Company: Flippin Engineering
Contact: http://www.lighthousepubl.com/Flippin_Engineering.htm
Author: Curtis G. Flippin
Date: 27 October 2009

Abstract

P4 version 4.0 is a formal linear procedural grammar for a simple, powerful, clear, and concise structured English Pseudo-code. P4's unique balance of structure and flexibility makes it a perfect modeling tool for Engineers, Analysts and Developers.

1 Introduction

P4 was created in 2001 out of necessity. I needed a clear, concise structured English pseudo-code that would be reasonably easy for any developer to understand and reasonably easy for any analyst to use. I soon found that there was very little to choose from and they were all simply style suggestions with no lexical or syntactic tools to promote consistency.

I designed P4 and used it for several months while I designed a program to parse P4 documents. The parser program became a full implementation of Extended Backus-Naur Form[2].

It has been my habit to redesign P4 with each new release of the Extended BNF meta-language application. Having just completed EBNF 4.0, I began work on the new P4. P4 4.0 is now much more flexible and easier to use than was version 3.5. Version 4.0 is more in tune with my original objectives of a simple, clear, concise and user friendly process modeling tool.

The new EBNF 4.0 can now parse and compile text files having 50,000 lines. That's about 600 pages of text! P4 is an EBNF 4.0 application and so it inherits this capability. New in this P4 version is a "line tracker" error detection function that can usually tell you exactly which line of your text file contains an error or, at the very least, which P4 statement is causing the problem. Gone are the arithmetic and logical constructs. Instead, simple English phrases can be used. The new *define:* construct replaces the cumbersome data object and structures syntax.

Unlike most generalized pseudo-code, P4 is a defined, structured grammar with an error checking parser to ensure consistency. This is a difficult task for what appears to be a loosely bound format and syntax, but it can be done.

2 P4 Level 1 Structure

The P4 grammar was specified using EBNF 4.0[1] which is a FORTH[3] application that implements the Extended Backus-Naur Form Metalanguage Standard[2]. EBNF 4.0 serves a dual purpose. Firstly, it provides a standard specification for the P4 grammar. Secondly, it can be compiled into Forth code without any intervening steps. In other words, the EBNF 4.0 specification is the grammar parser code.

The EBNF production below defines P4 at the highest level (Level 1). Note that P4 contains six alternate sequences separated by a vertical bar `'|'`. These are read as logical 'OR' statements so that P4 is an "eos" OR an `<end>` OR a `<p4-comment>` OR a `<p4-note>`, etc. statement. Also note that for the last five alternative sequences, P4 is recursive. That is, P4 calls itself.

```
\ Outermost Syntactic Level
bnf= <p4>
      "eos"
   |  <end>
   |  <p4-comment>  <p4>
   |  <p4-note>  <p4>
   |  <blank-line>  <p4>
   |  <begin>  <p4-module>  <p4>
   |  <p4-module>  <p4>
;bnf
```

The production specifies that P4, at Level 1, will accept one or more comments, notes, blank lines, begin module blocks, or modules until it encounters either the "eos" (end of source file) or a Level 1 *end:* statement.

2.1 Ending a P4 Document

P4 source files are text files (*.txt). Your text file editor should be set up to display line numbers (1..n) and tabs should be converted to three spaces.

Other than a 'carriage return/line feed' end of text line sequence, there should be no embedded control characters in the source file.

If, at Level 1, the end of the source file is detected or an 'end' command is encountered, the document is considered ended. Even an empty source file will be considered legal and will end without error[1]. An 'end' command is simply:
end:
on a line by itself.

2.2 Comments and Notes

Comments and notes take the forms as shown below. Notes are really technical notes and side comments. I borrowed this concept from the Extended BNF Standard. Both comments and notes can be multi-line. The delimiters must be isolated by at least one space character.

```
(* This is a single line p4 comment *)

(* One suggested form for multi-line comments is to begin
   the comment as though it were a single line comment
   but place the closing delimiter on a line by itself.
*)

(*
 The other suggested form is to place the opening and
 closing delimiters on lines by themselves.
*)

?? Technical Notes obey the same rules as Comments except
 the opening and closing delimiters are the same
 double question mark sequence.
??
```

Blank Lines are a special necessity and they do not refer to text lines having all space characters. P4 will recognize a line of spaces as normal text. It is possible, however, to create a text line with no characters other than the end-of-line carriage return/line feed sequence. These empty blank lines are included in the lexical analysis by virtue of the `<blank-line>` production.

[1] Actually, EBNF will issue an empty file error and abort the process

2.3 Optional Code Modules

The P4 grammar attempts to strike a delicate balance between structure and flexibility. A linear procedural grammar must have enforceable rules that form the foundation for structure. A generalized pseudo-code must allow the writer the freedom to describe processes in the most appropriate form. A fundamental tenant of process design is that it is often advantageous to break complex processes (or problems, if you like) into smaller segments.

```
(* File Input/Output Processing *)

begin:   File I/O Module;

   (* Include all file processing in this module *)

end:     File i/o;
```

P4 provides this capability in the form of modules. Modules are just like any other P4 code except a module is framed with a *begin:* and *end:* statement as shown above.

3 P4 Level 2 Module Structure

The EBNF 4.0 production for a P4 module is shown below. Modules are simply Level 2 code and can be defined as structured or unstructured depending on whether the module is defined as a begin-end code segment.

```
bnf= <p4-module>
      "eos"
   |  <end>
   |  <p4-comment>  <p4-module>
   |  <p4-note>  <p4-module>
   |  <define>  <p4-module>
   |  <function>  <p4-module>
   |  <procedure>  <p4-module>
   |  <p4-do>  <p4-module>
   |  <blank-line> <p4-module>
;bnf
```

Four new statements are available in Level 2:

1. Define
   Narrative data object and structure descriptions

2. Function
   Define a function, its arguments and processes

3. Procedure
   Define a procedure and its processes

4. Do
   Narrative process description

All Level 2 statements are recursive except for "eos" and `<end>` but an *end:* statement is required to end a module even if it is unstructured (i.e. did not start with *begin:*).

3.1 Defining Data Objects

There are no formal data object declaration statements. Labeled data is handled in the form:

```
define:   temp1  as  temporary variable;
define:   pi  as  constant = 3.14;
define:   customer_table  as  customer data tables;
```

The format is simply the keyword *define:* followed by a label for the data and a description of the data. A semi-colon signals the end of the statement.

Labels are required for data, functions and procedures. P4 does not perform any label usage checks. It does, however, perform some syntax checks on labels.

- A label may not contain embedded space characters. Underscore characters can be used in place of spaces.
- A label may not contain the characters or character sequences "(", ")", "??", ":", or ";".

P4 text also has some restrictions and they are slightly different from those for labels. P4 text is used frequently such as the remainder of a *define:* statement following the label.

- P4 text may be multi-line but must end with a semi-colon.
- P4 text may not contain the characters or character sequences "(*", "*)", "??", ":", or ";".

3.2 Functions

Functions are treated much like procedures. Unlike procedures, however, functions have an argument list. A function is used simply by typing the function name and argument list whereas using a procedure requires a *call:* statement. The body of a function is identical to the body of a procedure. That is, any statement permitted within a procedure is also allowed within a function.

```
(*
   P4 Function, Procedure and If Examples
*)

begin:   If Statement Test Module;

   define:  answer  as  ascii character key buffer;

   function:   getkey(key);
      do:   wait for key-pressed and return character in key;
   end:

   procedure:  ask;
      do:   type "Do you want to continue? [y/n]";
      getkey(answer)
      (* convert answer to uppercase *)
      do:   answer AND 65;
      if:   answer = 'Y';
         do:   continue process;
      elseif:  answer = 'N';
         do:   terminate process;
      else:
         do:   invalid answer;
      endif:
   end:  ask;

end:  If Statement Test;
```

If you read through the above example you will see that the code is defined as a module (*begin:*). The variable *answer* is defined as a character buffer. Below that, the function *getkey* is defined. Note that it has only one argument, *key*. If additional arguments were needed they would be separated by commas. Also note that the arguments do not need to be defined. For that matter, the variable *answer* doesn't need to be defined either. Doing so is only necessary to make the code clear and understandable.

The procedure *ask* uses function *getkey* to obtain the users answer to a question. The function call is one of the few times when a semi-colon is not required afterward. There is quite a bit of low level detail in this example but it could easily have been described at a much higher level.

3.3 Procedures

Procedures are declared using the *procedure:* keyword followed by the name of the procedure and an end of line semi-colon. As with functions, the procedure definition ends with an *end:* command which can be followed by the procedure name if you wish.

The body of a procedure is exactly the same as the body of a function. The EBNF definition of a function uses the <*procedure-body*> production just like the definition of procedure.

```
bnf= <procedure-body>
      <p4-comment>  <procedure-body>
   |  <p4-note>  <procedure-body>
   |  <define>  <procedure-body>
   |  <function-call>  <procedure-body>
   |  <procedure-call>  <procedure-body>
   |  <p4-do>  <procedure-body>
   |  <blank-line>  <procedure-body>
   |  <if>  <procedure-body>
   |  <while>  <procedure-body>
   |  <until>  <procedure-body>
   |  <case>  <procedure-body>
   |  <end>
;bnf
```

Procedures and functions may contain multiple occurrences of comments and notes, data definitions, function and procedure calls, do statements, blank lines, and a selection of conditional control structures. The *end:* statement signals the end of a procedure or function definition.

3.4 The Do Statement

The *do:* statement is essentially a predicate form of comment. Whereas comments and technical notes inform, the do statement is action oriented. The statement keyword is *do:* followed by P4 text [2] and an end of line semi-colon. The statement may span several lines.

4 P4 Level 3 Conditional Control Structures

The selection of control structures was quite deliberate. The list may appear to some to be lacking in that there are only four control structures.

1. if:...elseif:...else:...endif:
2. while:...repeat:
3. until:...repeat
4. case:...when:...other:...endcase:

Each control structures' form and format was carefully designed to be recognizable and to permit the writers' intent to be clear, concise and obvious. I assumed that writers using the P4 format would strive for clarity over cleverness. P4 allows one a great deal of freedom when creating a procedural description. Consequently, the writer also has the freedom to create nonsense as long as it is structured properly.

The objective of conditional control structures is to specify what action or actions should be taken depending on whether certain conditions have or have not been met. Decisions are based upon a set of one or more conditions. Naturally, all of the controls are only permitted within the bodies of functions or procedures. Each is detailed in the following subsections.

4.1 The If Conditional Statement

The *if:* statement is a structured way of telling the reader what to do depending on a set of conditions. In its simplest form it says:

[2] See Section 3.1 for P4 text restrictions

```
if:   a set of one or more conditions is met;
      do:   take appropriate action;
endif:
```

In order to accommodate the either or situation where one set of actions is needed if a set of conditions is met and another set of actions is needed if those conditions are not met, we have this form:

```
if:   a set of conditions is met;
      do:   take one set of actions;
else:
  do:   otherwise, take another set of actions;
endif:
```

Multiple sets of conditions can be handled with this form:

```
if:   temperature is higher than maximum limit;
      do:   start compressor-on cycle;
elseif: temperature is lower than minimum limit;
      do:   stop compressor-on cycle;
else:
      do:   reset temperature test-cycle timer;
endif:
```

You can use as many *elseif:* statements as you need but only one *else:* statement is allowed and it must follow the last *elseif:* of the *if:* statement. The *else:* statement always means that none of the previous conditions, no matter how many in a given *if:* statement, have been met.

Those of us who deal with real world systems and processes know how annoyingly untidy the real world can be. A robust form of *if:* statement is needed to express those conditions. P4 imposes no limit on *if:* nesting levels. The *if:*, *while:*, and *until:* control structures share the same EBNF body production.

```
bnf= <control-body>
      <p4-comment>
   |  <p4-note>
   |  <function-call>
   |  <procedure-call>
   |  <p4-do>
   |  <blank-line>
   |  <if>
   |  <while>
   |  <until>
   |  <case>
;bnf
```

Any or all of the above are permitted within an *if:...endif:* control structure. The following section provides a more complete example of a simple process control monitor. It is not so much a design as a talking paper to use when gathering more detailed requirements.

4.1.1 Process Control Monitor Example

```
(* P4 Simple Process Control Monitor Example
   Curtis G. Flippin
   27 October 2009
   P4 Pseudocode (c)2001-2009 *)
(*
   System physical plant consists of one or more production units.
   A production unit accepts two reagents that are mixed in a
   specific proportion in a reactor vessel under a specified
   production standard temperature and pressure, PSTP.
   The reagents react to form a new chemical that is output
   from the reactor vessel.
   The process is continuous.
*)
   define:  system   as    g(lambda) chemical production unit;

(* System Identification
   Significant a priori knowledge of the system has been modeled
   and incorporated via a set of nominal and optimum operating
   parameters for each of the following production factors. *)
   define:  r1_r2    as    input reagents proportions in GPM;
   define:  c1_r3    as    production output quality characteristic one;
   define:  c2_r3    as    production output quality characteristic two;
   define:  f_rate   as    production output flow rate in GPM;
   define:  pstp     as    production standard temperature and pressure;
   define:  a_spd    as    vessel mix agitator speed;
```

```
(* System Dynamics
   The chemical reaction process combines two reagents, r1 and r2,
   in unequal proportion, r1_r2, to produce a product, r3, with
   the desired qualities, c1_r3 and c2_r3, in a quantity defined
   as the flow rate, f_rate. The process is non-linear.
   Production process controls consist of the following reactor
   vessel controls.
*)
   define:  ctl_temperature   as    reactor temperator control;
   define:  ctl_pressure      as    reactor pressure control;
   define:  ctl_speed         as    reactor agitator speed;

(* System Performance
   The objectives of the Process Control Monitor are to aid
   human operators in controlling the reaction process and to
   log system performance in a manner that will allow
   consolidation with reaction control process logs for
   later use as a learning tool for an adaptive control
   system model. Performance is tracked as an index that is
   a function of all the variable parameters plus the
   control deltas over a defined ideal sampling rate.
*)
   define:  ip    as    performance index which is a function of
                        (r1_r2,c1_r3,c2_r3,f_rate,pstp,a_spd,
                        delta[temperature,pressure,speed]);
   define:  d_ip  as    delta(ip) memory;

   function:   delta_ip(c1,c2,f,t,p,d_ip);
   (* Calculate delta(ip) from production factor arguments
      and return in d_ip.
      input arguments
         c1    quality characteristic 1
         c2    quality characteristic 2
         f     product flow rate
         t     reactor temperature
         p     reactor pressure
      output arguments
         d_ip  delta of latest ip to new calculated ip.
         d_ip is retained data that is used by this function
         and elsewhere to determine the modification direction
         information that is sent to the operator.
   *)
   end:  delta_ip;

(* System Analysis
   Operators perform the process analysis in response to data
   they receive from the process control monitor. The monitor
   does not determine corrective actions. It tracks performance, ip,
```

```
   and compares the ip to both optimal and nominal performance
   standards. Operator notices are transmitted to the operator
   indicating the current level of system performance. There are
   three levels of notices, optimal ip, nominal ip, and sub-nominal
   ip. Notices include all system parameters and are logged along
   with a timestamp.
*)
   define:  ip_optimum     as    optimum performance notice;
   define:  ip_nominal     as    nominal performance notice;
   define:  ip_subnominal  as    sub-nominal performance notice;

   procedure:  delay;
      (* This procedure is a stand-in for the sampling rate
         control process. An ideal rate will help to create
         parameter delta's that are well above ambient noise.
      *)
   end:  delay;

   procedure:  monitor;

      do:   load initial known production parameters;

      (* Monitor runs until the shutdown process orders
         monitoring to stop.
      *)
      until:   stop-order received from shutdown;
         do:   read sensors r1_r2, c1_r3, c2_r3, f_rate, a_spd,
               temperature, pressure;
         (* Mathematical function f(ip) is not yet defined *)
         do:   calculate ip performance index;
         do:   calculate delta_ip(,,,,,d_ip);
         if:   ip is outside optimum performance limits;
               if:   ip is outside nominal performance limits;
                  do:   test ctl_temperature, ctl_pressure,
                        ctl_speed for sub-nominal readings;
                     if:   any are sub-nominal;
                        ?? we may want to track time between
                           sub-nominal alarms in order to
                           elevate the alarm level for
                           persistent performance problems.
                        ??
                        do:   set alarm status for sub-nominal
                              controls;
                     endif:
                  do:   send ip_subnominal performance notice;
               else:
                  do:   send ip_nominal performance notice;
               endif:
         else:
```

```
            do:   send ip_optimum performance notice;
         endif:
         call: delay to maintain an ideal 1 cps sample rate;
      repeat:

   end:  monitor;

end:  Process Control Monitor
```

4.2 The While and Until Statements

The *while:* and *until:* keywords are conditional repeated process control statements that both end with the *repeat:* keyword. Basically, these statements are used as follows:

```
while:   a set of one or more conditions has been met;
   do:   perform some appropriate process;
repeat:

until:   a set of one or more conditions is met;
   do:   perform some appropriate process;
repeat:
```

The conditions at the top of the loop are retested with each iteration when *repeat:* sends the process back to the beginning. [3]

Any of the `<control-body>` options listed for *if:* statements are allowed within *while:* and *until:* statements. These are the only loop control structures in P4. I have chosen to omit 'do loops', 'for next loops' and the like because they have no place in pseudo-code. P4 does not define any data objects or structures, either. It only allows labels and an explanation of what they are and how they may be used. P4 is structured English and should not be confused with programming languages.

4.3 The Case Statement

P4 has always included a *case:* statement. I rarely use it myself but it can be useful in the right circumstances. Here's an example of how it's used.

[3] See example of *until:..repeat:* in section 4.1.1

```
(* P4 Case Statement *)
procedure:  example;

   case: read value of temperature sensor;

      when: temperature is above normal;
         do:   modify temperature downward;
         call: log to record event;

      when: temperature is below normal;
         do:   modify temperature upward;
         call: log to record event;

      other:
         call: reset to clear previous alarms;
         call: log to record event;

   endcase:

end:  example;
```

Keep in mind that this is a very simple form of 'case' statement. The statement tells us that we're going to test something having discrete properties or values and take some action depending on a list of possible cases. In the example, we have the case of a temperature read from a sensor. When the temperature is above normal, we want to take some action and when it is below normal, we want to take other actions. For all other cases not specifically covered by previous *when:* statements, we want to take another set of actions. The *other:* catch all is optional but at least one *when:* statement should be used. Inside a *case:* statement you may use any of the statements listed here.

```
bnf= <case-list>
      <p4-comment>
   |  <p4-note>
   |  <function-call>
   |  <procedure-call>
   |  <p4-do>
   |  <blank-line>
;bnf

bnf= <case-body>
      <case-list>  <case-body>
   |  <case-list>
;bnf
```

As you can see, multiple comments, notes, function and procedure calls, do statements, and blank lines are permitted. But no nested *case:* statements and no *if:*, *while:*, or *until:* statements are allowed. Hopefully, the limitations will promote clarity.

References

[1] Curtis G. Flippin,
Flippin Engineering,
Flippin Web
Extended BNF 4.0,
Released 14 July 2009.

[2] International Organization for Standardization and
International Electrotechnical Commission
Joint Technical Committee 1,
ISO/IEC 14977:1996(E),
The Standard Metalanguage Extended BNF,
Publ. 1996.

[3] Win32Forth Project Group,
www.win32forth.org/
Win32Forth Version 6.12.00,
Released 14 July 1997.