Pascal implementation

The P5 Compiler

Scott A. Moore

[1 Overview of Pascal-P5 7](#_Toc397026154)

[1.1 Introduction 7](#_Toc397026155)

[1.2 Why a P5 compiler? 8](#_Toc397026156)

[1.3 The organization of the Pascal-P compiler 11](#_Toc397026157)

[1.4 P5 as a practical compiler 12](#_Toc397026158)

[1.5 P5 as a model compiler 12](#_Toc397026159)

[2 Using Pascal-P5 13](#_Toc397026160)

[2.1 Configuring P5 13](#_Toc397026161)

[2.2 Compiling and running Pascal programs with P5 13](#_Toc397026162)

[2.3 Compiler options 14](#_Toc397026163)

[2.4 Other operations 14](#_Toc397026164)

[2.5 Reliance on Unix commands in the P5 toolset 15](#_Toc397026165)

[2.6 The “flip” command and line endings 15](#_Toc397026166)

[3 Building the Pascal-P5 system 16](#_Toc397026167)

[3.1 Compiling and running P5 with an existing ISO 7185 compiler 16](#_Toc397026168)

[3.2 Evaluating an existing Pascal compiler using P5 17](#_Toc397026169)

[3.3 Notes on using existing compilers 18](#_Toc397026170)

[3.3.1 GPC 18](#_Toc397026171)

[3.3.1.1 GPC on Cygwin 19](#_Toc397026172)

[3.3.1.2 GPC for mingw 19](#_Toc397026173)

[4 Files in the P5 package 20](#_Toc397026174)

[4.1 Directory: gpc 23](#_Toc397026175)

[4.2 Directory: gpc/linux\_X86 23](#_Toc397026176)

[4.3 Directory: mac\_X86 23](#_Toc397026177)

[4.4 Directory: gpc/standard\_tests 23](#_Toc397026178)

[4.5 Directory: gpc/windows\_X86 24](#_Toc397026179)

[4.6 Directory: ip\_pascal 24](#_Toc397026180)

[4.7 Directory: ip\_pascal/standard\_tests 24](#_Toc397026181)

[4.8 Directory: ip\_pascal/windows\_X86 24](#_Toc397026182)

[4.9 Subdirectory: sample\_programs 24](#_Toc397026183)

[4.10 Directory: standard\_tests 25](#_Toc397026184)

[5 Differences between Pascal-P4 and Pascal-P5 25](#_Toc397026185)

[5.1 Viewing changes 25](#_Toc397026186)

[5.2 Notes about change descriptions 26](#_Toc397026187)

[5.3 Changes to the parser 26](#_Toc397026188)

[5.3.1 Thematic changes 26](#_Toc397026189)

[5.3.1.1 Variable strings 26](#_Toc397026190)

[5.3.1.2 Recycling based on dispose 26](#_Toc397026191)

[5.3.1.3 Files 27](#_Toc397026192)

[5.3.1.4 Byte oriented pseudo-machine 27](#_Toc397026193)

[5.3.2 Reading the source code 28](#_Toc397026194)

[5.3.2.1 Exit label 28](#_Toc397026195)

[5.3.2.2 The machine parameter block 28](#_Toc397026196)

[5.3.2.3 Other constants 28](#_Toc397026197)

[5.3.2.4 Types 29](#_Toc397026198)

[5.3.2.5 Variables 31](#_Toc397026199)

[5.3.2.6 Procedures and functions 33](#_Toc397026200)

[5.3.2.7 Recycling support routines 36](#_Toc397026201)

[5.3.2.7.1 Procedure getstr 36](#_Toc397026202)

[5.3.2.7.2 Procedure putstrs 36](#_Toc397026203)

[5.3.2.7.3 procedure getlab 36](#_Toc397026204)

[5.3.2.7.4 procedure putlab 36](#_Toc397026205)

[5.3.2.7.5 procedure pshcst 37](#_Toc397026206)

[5.3.2.7.6 procedure putcst 37](#_Toc397026207)

[5.3.2.7.7 procedure pshstc 37](#_Toc397026208)

[5.3.2.7.8 procedure putstc 37](#_Toc397026209)

[5.3.2.7.9 procedure ininam 37](#_Toc397026210)

[5.3.2.7.10 procedure putnam 37](#_Toc397026211)

[5.3.2.7.11 procedure putnams 37](#_Toc397026212)

[5.3.2.7.12 procedure putdsp 37](#_Toc397026213)

[5.3.2.7.13 procedure putdsps 38](#_Toc397026214)

[5.3.2.7.14 procedure getfil 38](#_Toc397026215)

[5.3.3 procedure putfil 38](#_Toc397026216)

[5.3.3.1.1 procedure getcas 38](#_Toc397026217)

[5.3.3.1.2 procedure putcas 38](#_Toc397026218)

[5.3.3.2 Character and string quata routines 38](#_Toc397026219)

[5.3.3.2.1 function lcase 39](#_Toc397026220)

[5.3.3.2.2 procedure lcases 39](#_Toc397026221)

[5.3.3.2.3 function strequri 39](#_Toc397026222)

[5.3.3.2.4 procedure writev 39](#_Toc397026223)

[5.3.3.2.5 function lenpv 40](#_Toc397026224)

[5.3.3.2.6 procedure strassvf 40](#_Toc397026225)

[5.3.3.2.7 procedure strassvr 40](#_Toc397026226)

[5.3.3.2.8 procedure strassvd 40](#_Toc397026227)

[5.3.3.2.9 procedure strassvc 40](#_Toc397026228)

[5.3.3.2.10 function strequvv 40](#_Toc397026229)

[5.3.3.2.11 function strltnvv 40](#_Toc397026230)

[5.3.3.2.12 function strequvf 40](#_Toc397026231)

[5.3.3.2.13 function strltnvf 40](#_Toc397026232)

[5.3.3.2.14 Function strchr 40](#_Toc397026233)

[5.3.3.2.15 Procedure strchrass 41](#_Toc397026234)

[5.3.3.2.16 procedure prtdsp 41](#_Toc397026235)

[5.3.4 Modifications 41](#_Toc397026236)

[5.3.4.1 procedure endofline 41](#_Toc397026237)

[5.3.4.2 procedure errmsg 41](#_Toc397026238)

[5.3.4.3 procedure error 41](#_Toc397026239)

[5.3.4.4 procedure insymbol 42](#_Toc397026240)

[5.3.4.4.1 Letter: identifiers and reserved words 42](#_Toc397026241)

[5.3.4.4.2 Number: integers and reals 42](#_Toc397026242)

[5.3.4.4.3 Chstrquo: Strings 43](#_Toc397026243)

[5.3.4.4.4 Chperiod: ‘.’, ‘..’ and ‘.)’ tolkens 43](#_Toc397026244)

[5.3.4.4.5 Chlparen: ‘(‘and comment start 43](#_Toc397026245)

[5.3.4.4.6 Chlcmt: ISO 7185 comment start 43](#_Toc397026246)

[5.3.4.4.7 Lexical dump 43](#_Toc397026247)

[5.3.4.5 procedure enterid 43](#_Toc397026248)

[5.3.4.6 procedure searchsection 44](#_Toc397026249)

[5.3.4.7 procedure searchidne 44](#_Toc397026250)

[5.3.4.8 procedure searchid 44](#_Toc397026251)

[5.3.4.9 procedure printtables 44](#_Toc397026252)

[5.3.4.10 procedure followstp 44](#_Toc397026253)

[5.3.4.11 procedure followctp 44](#_Toc397026254)

[5.3.4.12 procedure searchlabel 44](#_Toc397026255)

[5.3.4.13 procedure newlabel 44](#_Toc397026256)

[5.3.4.14 procedure prtlabels 45](#_Toc397026257)

[5.3.4.15 procedure block 45](#_Toc397026258)

[5.3.4.16 procedure constant 45](#_Toc397026259)

[5.3.4.17 function string 46](#_Toc397026260)

[5.3.4.18 function comptypes 46](#_Toc397026261)

[5.3.4.19 function filecomponent 48](#_Toc397026262)

[5.3.4.20 function string 48](#_Toc397026263)

[5.3.4.21 procedure typ 48](#_Toc397026264)

[5.3.4.22 procedure simpletype 48](#_Toc397026265)

[5.3.4.23 procedure fieldlist 48](#_Toc397026266)

[5.3.4.24 procedure labeldeclaration 48](#_Toc397026267)

[5.3.4.25 procedure constdeclaration 48](#_Toc397026268)

[5.3.4.26 procedure typedeclaration 48](#_Toc397026269)

[5.3.4.27 procedure vardeclaration 48](#_Toc397026270)

[5.3.4.28 procedure procdeclaration 48](#_Toc397026271)

[5.3.4.29 procedure pushlvl 48](#_Toc397026272)

[5.3.4.30 procedure parameterlist 48](#_Toc397026273)

[5.3.4.31 procedure body 48](#_Toc397026274)

[5.3.4.32 procedure addlvl 48](#_Toc397026275)

[5.3.4.33 procedure sublvl 48](#_Toc397026276)

[5.3.4.34 procedure mes 48](#_Toc397026277)

[5.3.4.35 procedure putic 48](#_Toc397026278)

[5.3.4.36 procedure gen0 48](#_Toc397026279)

[5.3.4.37 procedure gen1 48](#_Toc397026280)

[5.3.4.38 procedure gen2 48](#_Toc397026281)

[5.3.4.39 procedure gentypindicator 48](#_Toc397026282)

[5.3.4.40 procedure gen0t 48](#_Toc397026283)

[5.3.4.41 procedure gen1t 48](#_Toc397026284)

[5.3.4.42 procedure gen2t 48](#_Toc397026285)

[5.3.4.43 procedure load 48](#_Toc397026286)

[5.3.4.44 procedure store 48](#_Toc397026287)

[5.3.4.45 procedure loadaddress 48](#_Toc397026288)

[5.3.4.46 procedure genfjp 48](#_Toc397026289)

[5.3.4.47 procedure genujpxjp 49](#_Toc397026290)

[5.3.4.48 procedure genipj 49](#_Toc397026291)

[5.3.4.49 procedure gencupent 49](#_Toc397026292)

[5.3.4.50 procedure genlpa 49](#_Toc397026293)

[5.3.4.51 procedure checkbnds 49](#_Toc397026294)

[5.3.4.52 procedure putlabel 49](#_Toc397026295)

[5.3.4.53 procedure statement 49](#_Toc397026296)

[5.3.4.54 procedure expression 49](#_Toc397026297)

[5.3.4.55 procedure selector 49](#_Toc397026298)

[5.3.4.56 procedure call 49](#_Toc397026299)

[5.3.4.57 procedure variable 49](#_Toc397026300)

[5.3.4.58 procedure getputresetrewriteprocedure 49](#_Toc397026301)

[5.3.4.59 procedure pageprocedure 49](#_Toc397026302)

[5.3.4.60 procedure readprocedure 49](#_Toc397026303)

[5.3.4.61 procedure writeprocedure 49](#_Toc397026304)

[5.3.4.62 procedure packprocedure 49](#_Toc397026305)

[5.3.4.63 procedure unpackprocedure 49](#_Toc397026306)

[5.3.4.64 procedure newdisposeprocedure 49](#_Toc397026307)

[5.3.4.65 procedure absfunction 49](#_Toc397026308)

[5.3.4.66 procedure sqrfunction 49](#_Toc397026309)

[5.3.4.67 procedure truncfunction 49](#_Toc397026310)

[5.3.4.68 procedure roundfunction 49](#_Toc397026311)

[5.3.4.69 procedure oddfunction 49](#_Toc397026312)

[5.3.4.70 procedure ordfunction 49](#_Toc397026313)

[5.3.4.71 procedure chrfunction 49](#_Toc397026314)

[5.3.4.72 procedure predsuccfunction 49](#_Toc397026315)

[5.3.4.73 procedure eofeolnfunction 49](#_Toc397026316)

[5.3.4.74 procedure callnonstandard 49](#_Toc397026317)

[5.3.4.75 procedure compparam 50](#_Toc397026318)

[5.3.4.76 procedure expression 50](#_Toc397026319)

[5.3.4.77 procedure simpleexpression 50](#_Toc397026320)

[5.3.4.78 procedure term 50](#_Toc397026321)

[5.3.4.79 procedure factor 50](#_Toc397026322)

[5.3.4.80 procedure assignment 50](#_Toc397026323)

[5.3.4.81 procedure gotostatement 50](#_Toc397026324)

[5.3.4.82 procedure compoundstatement 50](#_Toc397026325)

[5.3.4.83 procedure ifstatement 50](#_Toc397026326)

[5.3.4.84 procedure casestatement 50](#_Toc397026327)

[5.3.4.85 procedure repeatstatement 50](#_Toc397026328)

[5.3.4.86 procedure whilestatement 50](#_Toc397026329)

[5.3.4.87 procedure forstatement 50](#_Toc397026330)

[5.3.4.88 procedure withstatement 50](#_Toc397026331)

[5.3.4.89 procedure programme 50](#_Toc397026332)

[5.3.4.90 procedure entstdnames 50](#_Toc397026333)

[5.3.4.91 procedure enterundecl 50](#_Toc397026334)

[5.3.4.92 procedure exitundecl 50](#_Toc397026335)

[5.3.4.93 procedure initscalars 50](#_Toc397026336)

[5.3.4.94 procedure initsets 50](#_Toc397026337)

[5.3.4.95 procedure inittables 50](#_Toc397026338)

[5.3.4.96 procedure reswords 50](#_Toc397026339)

[5.3.4.97 procedure symbols 50](#_Toc397026340)

[5.3.4.98 procedure rators 50](#_Toc397026341)

[5.3.4.99 procedure procmnemonics 50](#_Toc397026342)

[5.3.4.100 procedure instrmnemonics 50](#_Toc397026343)

[5.3.4.101 procedure chartypes 50](#_Toc397026344)

[5.3.4.102 procedure initdx 50](#_Toc397026345)

[6 Changes to the assembler/interpreter 51](#_Toc397026346)

[7 The intermediate language 51](#_Toc397026347)

[7.1 Format of intermediate 51](#_Toc397026348)

[7.1.1 Label 52](#_Toc397026349)

[7.1.2 Source line marker 52](#_Toc397026350)

[7.2 Intermediate instruction set 52](#_Toc397026351)

[7.3 System calls 66](#_Toc397026352)

[8 Testing P5 73](#_Toc397026353)

[8.1 Running tests 73](#_Toc397026354)

[8.1.1 testprog 73](#_Toc397026355)

[8.1.2 Other tests 75](#_Toc397026356)

[8.1.3 Regression test 75](#_Toc397026357)

[8.2 Test types 75](#_Toc397026358)

[8.3 The Pascal acceptance test 75](#_Toc397026359)

[8.4 The Pascal rejection test 75](#_Toc397026360)

[8.4.1 List of tests 76](#_Toc397026361)

[8.4.1.1 Class 1: Syntatic errors 76](#_Toc397026362)

[8.4.1.2 Class 2: Semantic errors 82](#_Toc397026363)

[8.4.2 Running the PRT and interpreting the results 85](#_Toc397026364)

[8.4.2.1 List of tests with no compile or runtime error. 86](#_Toc397026365)

[8.4.2.2 List of differences between compiler output and “gold” standard outputs. 86](#_Toc397026366)

[8.4.2.3 List of differences between runtime output and “gold” standard outputs. 86](#_Toc397026367)

[8.4.2.4 Collected compiler listings and runtime output of all tests. 86](#_Toc397026368)

[8.4.3 Overall interpretation of PRT results 87](#_Toc397026369)

[8.5 Sample program tests 87](#_Toc397026370)

[8.6 Previous Pascal-P versions test 87](#_Toc397026371)

[8.6.1 Compile and run Pascal-P2 87](#_Toc397026372)

[8.6.2 Compile and run Pascal-P4 88](#_Toc397026373)

[8.7 Self compile 88](#_Toc397026374)

[8.7.1 pcom 89](#_Toc397026375)

[8.7.1.1 Changes required 89](#_Toc397026376)

[8.7.2 pint 89](#_Toc397026377)

# Overview of Pascal-P5

This section contains background material on Pascal-P5. If you want to get started using Pascal-P5 now, skip to 2 “Using Pascal-P5”

## Introduction

The Pascal-P series compilers were the original proving compilers for the language Pascal. Created in 1973, Pascal-P was part of a “porting kit” designed to enable the quick implementation of a Pascal language compiler on new machines. It was released by Niklaus Wirth’s students at ETH in Zurich.

The implementation and description of the language Pascal in terms of itself and in terms of a “pseudo machine” were important factors in the propagation of the language Pascal. From the early version of Pascal-P came the CDC 6000 full compiler at Zurich, several independent compilers including an IBM-360 compiler and a PDP-11 compiler, and the UCSD “byte code” interpreter.

The original article for the Pascal-P compiler is at:

<http://www.standardpascal.org/The_Pascal_P_Compiler_implementation_notes.pdf>

In the name “Pascal-P” the “P” stood for “portable”, and this was what Pascal-P was designed to do. It also stood for an example and reference implementation of Pascal, although Wirth later issued a paper, together with Tony Hoare for the “Axiomatic definition of Pascal”, which was also aimed at exactly specifying the semantics of Pascal.

As the importance of Pascal-P grew, the authors adopted a version number system and working methodology for the system. A new, cleaner and more portable version of the system was created in 1974 with the name Pascal-P2, and left the multiple early versions of the system as termed Pascal-P1.

From the Pascal-P2 revision of the compiler comes many of the original Pascal compilers, including UCSD. In 1976, Wirth’s group made one last series of improvements and termed the results Pascal-P3 and Pascal-P4. Pascal-P3 was a redesigned compiler, but used the same pseudo machine instruction set as P2, and thus could be bootstrapped from an existing P2 implementation. P4 featured a new pseudo instruction set, and thus was a fully redesigned compiler.

Pascal-P was always an imcomplete implementation of the Pascal language (a subset), and was designed to be so. After it was created, the ISO 7185 standard for Pascal was issued, and today Pascal-P4 exists and is still usable with minor changes to bring it into ISO 7185 compliance.

However, Pascal-P4 has it’s legacy problem of being a subset compiler of the full language. Further, it is only usable for programs that avoid its weaknesses, such as string storage. Keep in mind that Pascal-P was never designed to be a general purpose system, but rather to compile itself on a new machine – and then rapidly be improved to become a full compiler.

Thus in 2008 I set out to improve the P4 code to accept the full language Pascal as stated by the ISO 7185 language standard. The name of the result was obvious: Pascal-P5, although I certainly beg forgiveness of any of the original Wirth group members who might be offended by my implication of extending and improving their work. Think of it this way: I too have been a lifelong student of Niklaus Wirth’s, abet a long distance one.

And so, 35 years after the original Pascal-P compiler was created, a new version of the series exists. I have further designs on Pascal-P beyond the P5 project and the Pascal language, as represented by the new language Pascaline (whose document you may find at:

<http://www.standardpascal.com/standards.html>

However, it is right and proper that the Pascal-P series should stop at P5 and take a breath. P5 is the logical completion of the Zurich group’s work, and it stands alone as an achievement.

The Pascal-P compiler series (and it’s companion Pascal-S) have been extensively documented in the literature. However, the book “Pascal implementation: The P4 compiler” by Steve Pemberton and Martin Daniels stands out as a running code commentary on the level of “lion’s commentary on Unix”. It is just that good. It is also available free on line at:

<http://homepages.cwi.nl/~steven/pascal/book/>.

Accordingly, for the P5 document, I have not attempted to recapitulate the entire P4 part of the compiler in this document. I see this document as an incremental explaination of the improvements to P4 needed to arrive at P5. I hope Steven will forgive me for using a variant of his title, and the frequent references to his book (and occasional corrections). The fact is I could not have completed the P5 project without Steve’s work. In fact, it was Steve’s book that convinced me that P4 was worth preservation and modernization.

## Why a P5 compiler?

It actually makes more sense to ask “why a P4 compiler” than “why a P5 compiler”. P4 was not a full Pascal at all, but rather a subsetted version of the language with several features removed. The omissions and changes were (from the P4 web page at: <http://www.standardpascal.com/p4.html>):

* Procedure/function parameters.
* Interprocedural gotos (goto must terminate in the same procedure/function).
* Only files of type "text" can be used, and then only the ones that are predefined by P4, which are "input", "output", and two special files defined so that P4 can compile itself.
* "mark" and "release" instead of "dispose".
* Curly bracket comments {} are not implemented.
* The predeclared identifiers maxint, text, round, page, dispose, and the functions they represent, are not present.
* The procedures reset, rewrite, pack and unpack are not implemented (they are recognized as valid predefined procedures, but give an 'unimplemented' error on use).
* Undiscriminated variant records.
* Output of boolean types.
* Output of reals in "fixed" format.
* Set constructors using subranges ('0'..'9').

However, there were several other issues with the P4 compiler beyond simply the language it implemented. P4 made no attempt to economize on its storage of strings. This meant that each string constant, no matter how long it was, would be stored in a fixed length in the pseudo-machine code. Although the internal string length in the interpreter was a settable constant, and implementor using P4 would always be operating between the mutually exclusive goals of having enough string characters to represent usable strings and having the total string storage use too much space.

Also, every variable in the P4 interpreter was afforded the same space. A character or a boolean used the same space as a floating point value, and an array of characters would be as costly as an array of floating point numbers.

P4 itself got around these limitations by using strings and string constants sparingly. This again goes to the idea that P4 (and the Pascal-P series) was primarily designed to compile itself, and was never designed as a real, working compiler.

Very tellingly, when Kenneth Bowles received P2 and wanted to use it as an interpreter in and of itself, not as just a stepping stone to a native compiler, his team extensively reworked it to use a byte orientation, and implemented string storage efficientcy.

The exact reasons why P4 is as it is, of course, belong to it’s original authors. However, it is fair to say that Pascal-P was designed to be a lightweight porting kit for Pascal. The two main concerns were:

* Limiting the required memory for the run of the self compilation.
* Limiting the complexity of the self compilation.

For the first, obviously the Zurich crew was not particularly limited by memory. The CDC 6000 series computers they had access to were state of the art for their day, and after Pascal-P was produced, they extended it to a full native compiler for the CDC 6000 (see <http://www.standardpascal.com/CDC6000pascal.html>). This implies that a full language version of P4 could have been completed. They may have wished to lessen the load on other implementers of the language outside of Zurich.

The second reason is far more concrete. Even a complex program such as a compiler may not use the entire language, simply because the need did not arise. It was, and is, standard practice to implement a subset of a full language for the first compiler version and improve it later.

Finally, it is important to understand that the designers of Pascal-P never intended it to be used as a implementation for it’s own sake. A Pascal implementation that simulated, not executed, its output code was interesting to Wirth, but that resulted in the Pascal-S project, a one piece compiler/interpreter program that has also been said to have originated with the Pascal-P project (although you will find little in common between the source code for the two).

Thus, it would never have occurred to the original designers to make Pascal-P an efficient and full implementation of Pascal. It was simply a bridge to better things.

There is certainly less reason to keep Pascal-P as a language subset compiler today. Computers are drowning in memory, and virtual memory operating systems are the rule, not the exception. Further, there are several advantages to having even a compiler porting kit such as P4 process the full language:

* To serve as an example implementation of the language.
* To reduce the total work needed to convert a bootstrapped Pascal-P to the full language.
* To serve as a full language stand-alone interpreter.

Accordingly, such a compiler was created, the “Model implementation of Standard Pascal” [Welsh and Hay] with the advent of the ISO 7185 standard in 1982. This is a very good and complete implementation of Pascal which I can recommend reading. However, it has two significant drawbacks to it’s use:

* It was never freely distributed, and the rights to it were closely held.
* It had nothing whatever in common with the Pascal-P project.

Because or in spite of this, the “model implementation” is virtually unobtainable and unknown today. It can only be found in it’s increasingly rare book form.

The advantages of having a new version of P4 that both processes the full language, and also embodies an efficient interpreter in its own right are:

* It is a fairly reasonable increment in complexity of the original code.
* It starts from an existing and well understood code base.
* The main part of it is already well documented.

The number of lines in the source for P4 vs. P5 bear this out:

|  |  |  |
| --- | --- | --- |
|  | P4 | P5 |
| pcom.pas | 4119 | 5969 |
| pint.pas | 1100 | 2862 |

157 of the increased line count for the compiler front end, pcom.pas, are due to the error message printing routine that could be removed without ill effect (arriving at the same, numeric only error messages as the original P4).

The most radical changes were done in the interpreter, but this still remains a low percentage of the total, since the interpreter is not where the majority of the code in was in Pascal-P.

The changes needed to create P5 from P4 were carried out in the space of less than one man month, and the system still retains it’s “small system” feel at less than 10,000 lines. I have tried to stay within the original style of the code, although at times more than one style is evident in the source (due to the multiple original authors). I have, for example, refrained from reformatting or extensively adding comments to the code.

I admit this was difficult. I don’t care for the “compressed” nature of the formatting, nor the general lack of comments. For Pascal style, I both prefer and recommend the style of Henry Ledgard in “Pascal with style: Programming Proverbs”. I note that Niklaus Wirth’s general style is to present the comments separately from the program code, and thus having this document cover the additions required to the code suits me.

Finally, I’ll include a an advertisement for my next project here which Pascal purists can certainly skip, or perhaps stop to read simply to realize what a wrong headed path I have turned down. During the course of creating P5, I realized that the next step in the series was just as obvious as the step from P4 to P5 was. This was a package of coherient exentions to the language Pascal such as was appropriate after a 35 year lapse in it’s development.

The inclusion of a series of extentions in a pure interpreter such as the Pascal-P series, is, I would argue, a very appropriate place of debut. It means that one is not tempted to place machine specific or operating specific features into the language.

The result is already far along at this writing, and is termed “Pascaline”, and the new version of Pascal-P is Pascal-P6. P6 returns in a way to the P3-P4 split in that P6 is designed to compile on any ISO 7185 Pascal implementation, and finalized only by substituting the contents of a few procedures and functions in the interpreter to their local implementation equivalents. In the case of original P6, they are implemented in Pascaline themselves, and thus P6 is self compiling.

In this way, P6 can be compiled and bootstrapped on any ISO 7185 complying implementation, including Pascal-P5, and used to establish a Pascaline complying implementation anywhere. In addition, because Pascaline itself is downward compatible with ISO 7185 Pascal itself, Pascal-P6 can serve as a practical replacement for Pascal-P5 itself.

## The organization of the Pascal-P compiler

Pascal-P does not feature a machine independent front end. The sizes and characteristics of machine level objects in the interpreter also appear in the front end. The front end uses these sizes to form the layout of things such as local block frames, parameter lists and records.

This is one reason that Pascal-P has been criticized for being a less than ideal basis on which to build a full, machine independent compiler system. If the front end were to be fully machine independent, it would mean a great deal of change to the whole compiler system, and the intermediate would certainly change, since it is currently stated in machine dependent form, using fixed offsets and other machine dependent embedded numbers.

In practice, it is not much trouble to create a set of front and back end modules for any machine. The machine dependent figures are isolated to equates in the top of the program, and these simply need to be changed in both compiler and interpreter equally.

## P5 as a practical compiler

Is P5 a useable compiler for real applications? I would assert that it is not. The reason is the way it processes files. All of the files it uses must be declared specifically via the program header. For P5 to process files from a user program, it must map these files into header files defined by P5 itself.

The most obvious files for any program are the input and output files. Even though P5 itself uses these files, both to input the target program, and to output listings and error messages,these are useable by the target program simply because they are separated positionally in the output.

P5 defines two more files, prd and prr. These files are used for reading only and writing only, respectively. P5 uses prd to input and output the program code for the target. Although these files can be used by the target, again, the input and output is mixed with P5’s use of the files, and must be done positionally. In fact, this is the technique used to accomplish a self compile and run, where the target program is the P5 system itself.

P5 could map the header files to arbitrary files, but this would have to be done by non-standard code, since there is no method specified in either the original (J&W) standard, nor in ISO 7185, of mapping header files to external files. It is left as “implementation specific”.

All of this matches the original purpose of P5, which was simply to serve as a compiler porting platform. It also serves as a demonstration compiler. To make it a practical compiler, it would have to be extended, at least to operate on runtime specified files. The CDC 6000 compiler is effectively a P4 compiler that was extended to be a practical use compiler.

The other way to achieve a practical use P5 would be to specify implementation specific characteristics in it, such as external file connections. This would involve creating a P5 that is no longer strictly ISO 7185, since it would need to specify how the external file connections are done.

I do think this is valuable. However, I also believe that instead of adding extentions valid for only one implementation of Pascal, it makes more sense to organize the extentions as a whole. This is what the P6 project is about, so this work will be deferred to P6.

## P5 as a model compiler

The definition of a model compiler is as an example of how to carry out the actual execution of of the language pascal. Unfortunately, this is one area where the “model implementation of Pascal” [Welsh&Hay] is actually ahead of P5. The problem is that most of the I/O formatting is simply passed on to the underlying compiler, meaning that things like the exact format of input numbers, and the exact format produced by output of numbers, is left to whatever the implementation that P5 is run on does.

The way to get around this is to be fairly pedantic about processing input and output, for example reading and parsing a complete input number, with sign, even if the underlying implementation is capable of doing that.

This wasn’t done in P5, probally to keep the compiler/interpreter simple. It should be done to enhance the Pascal-P implementation’s status as a model compiler. Although it could be done as an enhanced version of P5, I think it makes more sense to delay it for P6.

# Using Pascal-P5

## Configuring P5

P5 has a simple configuration script to set up the binary, script files and compiler in use for the system, that uses the proper defaults for your system:

[Windows]

> setpath

> configure

> make

[Linux/Mac]

$ ./setpath

$ ./configure

$ make

You can avoid “setpath” by placing the ./bin directory on your path.

The configure script attempts to automatically determine the environment you are running under, choose the correct compiler, bit width of your computer, etc. You can override this by using the options for configure:

|  |  |
| --- | --- |
| Option | Meaning |
| --gpc | Selects the GPC compiler. |
| --ip\_pascal | Selects the IP Pascal compiler. |
| --32 | Selects 32 bit mode. |
| --64 | Selects 64 bit mode. |
| --help | Prints a help menu. |

The configure script will take the preconfigured versions of the p5 binaries, the script files and other files and install them for the specified compiler. The P5 system is configured by default for IP Pascal running on windows, and can be left as such if desired.

## Compiling and running Pascal programs with P5

To simply compile a run a program, use the P5 batch file:

C:\> p5 hello

When a pascal program is run this way, it gets it's input from the terminal (you), and prints its results there. The p5 script accommodates the compiler that was used to build the system, and therefore you don’t need to know the exact command format of the executable.

The rules for ISO 7185 Pascal are simple, and you can find a complete overview of the language in the file:

iso7185rules.html

In the ./doc directory.

You will find the complete standard for the language in the files:

iso7185.html html format

iso7185.pdf pdf format

In the ./doc directory.

If you were expecting P5 to look like UCSD Pascal or Borland Pascal, please note you took a wrong turn somewhere. P5 is the original Pascal language. The "Pascal" languages processed by UCSD and Borland were heavily modified, and very incompatible variants that that were brought out years after the original.

All files in P5 are anonymous, and only last the length of the program run. The exceptions to this are the "prd" and "prr" files, which are used by the P5 compiler to compile and run itself. You can use them, but you really have to know what you are doing. If you need to read from a file or write to a file use redirection:

C:\> p5 test < myinputfile > myoutputfile

You will find you can get a lot of tasks done this way.

Note:

P5, as was P4, was designed to be a Pascal compiler porting tool and model implementation ***first***, and not really as a practical day to day compiler. If you want a compiler/interpreter for that use, you want the P6 compiler, which contains things like file access extentions and a lot more.

## Compiler options

P5 uses a "compiler comment" to indicate options to the compiler, of the form:

(\*$option+/-,...\*)

This option can appear anywhere a normal comment can. The first character of the comment MUST be "$". This is followed by any number if option switches separated by ",". If the option end with "+", it means to turn it on. If the option ends with "-", it means turn it off.

Example:

(\*$l-\*)

Turns the listing of the source code OFF.

The following options are available:

|  |  |  |
| --- | --- | --- |
| Option | Meaning | Default |
| t+/- | Print/don't print internal tables after each routine is compiled. | OFF |
| l+/- | List/don't list the source program during compilation. | ON |
| d+/- | Add extra code to check array bounds, subranges, etc. | ON |
| c+/- | Output/don't output intermediate code. | ON |

## Other operations

Within the P5 toolset, you will find a series of scripts to perform common operations using P5. This includes building the compiler and interpreter using an existing ISO 7185 compatible compiler, and also testing P5.

The scripts used in P5 are designed to be independent of what operating system you are running on. The P5 system as been successfully run on the following systems:

* Windows
* Ubuntu linux
* Mac OS X

To enable this to work, there are two kinds of scripts available, one for DOS/Windows command shells, and another for Unix/Bash. These two script files live side by side, because the DOS/Windows scripts use a .bat extension, and Bash scripts use no extentions. Thus, when a script command is specified here, the particular type of script file is selected automatically.

The only exception to this rule is that Unix users commonly do not place the current directory in the path. This means to execute a script file in the current directory, you need to specify the current directory in front of the script. For example:

~/p5$ ./p5 hello

## Reliance on Unix commands in the P5 toolset

Most of the scripts in this package, even the DOS/Windows scripts, rely on Unix commands like cp, sed, diff, chmod and others. I needed a reasonable set of support tools that were command line callable, and these are all both standard and reasonable.

For Windows, the Cygwin toolset is available:

http://www.cygwin.com

Note that to run the cygwin tools, you will need the environment variable:

CYGWIN=nodosfilewarning

This prevents cygwin utilities from complaining about dos mode file specifications.

An alternative to Cygwin is the Mingw toolkit. Mingw uses GNU programs that are compiled as native Windows .exe files without special .dll files. It typically has better integration with Windows than Cygwin, since it does not try to emulate Unix on Windows.

Where possible, I have tried to use DOS/Windows commands. The scripts are available in both DOS/Windows and bash versions. I could have just required theuse of bash, which is part of the cygwin toolkit, but my aim is not to force Windows users into a Unix environment.

## The “flip” command and line endings

Every effort was made to make the Pascal-P5 compile and evaluate system idenpendent of what system it is running on, from Windows command shell, to Linux with Bash shell. One common thing I have found is that several utilities don’t appreciate seeing a line ending outside of their “native” line ending, such as CRLF for Windows, and LF for linux. Examples include “diff” (find file differences) and Bash.

Therefore many of the scripts try to remove the line ending conserations, either by ignoring such line endings, or by converting all of the required files to the particular line ending in use.

The key to this is the “flip” utility. After searching for several line ending converters, “flip” was found on the most number of systems, as well as being one of the most clear and reliable utilities (it translates in both directions, it tolerates any mode of line ending as input, will not corrupt binaries, etc.).

Unfortunately, even flip was not found on some systems. The simpliest way to fix this was to include the flip.c program with the distribution, then let you compile to form a binary on your system to replace the utility.

To make the flip utility, you run:

$ make\_flip

Then flip will exist in the root directory.

Note that if you are using SVN to retrieve the P5 project, the file entries are given the “OS specific” line ending property. This means both that the line endings will be converted to the line endings particular to your OS, and also prevents line endings from causing SVN to think the file has changed.

# Building the Pascal-P5 system

## Compiling and running P5 with the GCC/I80386 backend

P5 contains a code generator that targets 32 bit I80386 processors and compatables. In order to use this mode, you must have an executable image for the pcom (the compiler), and pcom\_gcc\_i80386 (the compiler for GCC and I80386). You can achieve this in two ways:

1. Have an ISO 7185 compatable compiler available.
2. Use a precompiled copy of pint and pgen available for your present machine.

The latter solution is the easiest, since that avoids the need to correctly deal with compiling the pint/pgen system. Note that pint (the interpreter) can also be used to help form a system, but we won’t cover that here, since if you have a compiled version of pint, then you also likely have a compiled version of pgen.

To select the GCC/I80386 system, you execute:

configure gcc\_i80386

This will automatically bring up an executable version of pcom, pint, and pgen\_gcc\_i80386 as compiled for your 32 bit I80386 compatable machine. Thus we execute:

configure gcc\_i80386

setpath

make

To run a full self compile.

## Compiling and running P5 with an existing ISO 7185 compiler

You do not need to compile P5 unless you are using an alternative compiler or installation. The current P5 has been compiled and run with the following compilers and operating systems:

|  |  |
| --- | --- |
| Compiler | Installations |
| IP Pascal | Windows |
| GPC | Windows, Ubuntu, Mac OSx |

First, you must have a ISO 7185 Pascal compiler available. There are several such compilers, see:

http://www.standardpascal.org/compiler.html

You will probally need to compile pcom.pas and pint.pas with the ISO 7185 Pascal compatibility mode option on for your compiler. See your documentation for details.

If you are using a compiler or version of a compiler that is not tested to ISO 7185 standards, you will want to make sure that it is ISO 7185 compliant. See “Tesing P5” section 18 for details on how to test an existing compiler to ISO 7185 standards.

To compile pcom and pint, the components of the P5 compiler, use the script files:

cpcom Compile pcom to binary with a specific compiler.

cpint Compile pint to binary with a specific compiler.

For convienence, both of these are run by the command:

Build Compile pcom and pint.

Cpcom and cpint are specific to the compiler you are using. You can use the script configure to select which existing compiler to run:

configure ip\_pascal

Or

configure gpc

To run the other programs and batch files, you should modify the following files to work with your compiler:

p5.bat The single program compile and run batch file.

compile.bat To compile a file with all inputs and outputs specified.

run.bat To run (interpret) the intermediate file with all inputs and outputs specified.

The reason you need to change these files is because pcom.pas uses the header file "prr" to output intermediate code, and pint.pas uses "prd" for input and "prd" for output. You need to find out how to connect these files in the program header to external named files.

For example, in IP Pascal, header files that don't bear a standard system name (like "input" and "output") are simply assigned in order from the command line. Thus, P5.bat is simply:

pcom %1.p5 < %1.pas

pint %1.p5 %1.out

Where %1 is the first parameter from the command line.

P5.bat lets the input and output from the running program go to the user terminal. Compile.bat and run.bat both specify all of the input, output, prd and prr files. The reason the second files are needed is so that the advanced automated tests can be run using batch files that aren't dependent on what compiler you are using.

If your compiler does nothing with header files at all, you will probally have to change the handling of the prd and prr files to get them connected to external files. To do this, search pcom and pint for "!!!" (three exclamation marks). This will appear in comments just before the declaration, reset and rewrite of these files.

## Evaluating an existing Pascal compiler using P5

If you plan to compile and run P5 using your compiler, you should evaluate your compiler's ISO 7185 Pascal compliance. Of course, simply compiling pcom.pas and pint.pas is one way to achieve that. But since this package gives you the ability to fully evaluate your compiler, I would suggest you use it.

First, you need to determine if your compiler has a ISO 7185 Pascal compliance option and turn it on if needed. I say "if needed", because some compilers actually change their behavior with the option enabled, and thus it is not possible to compile and run standard Pascal programs unless the option is on (a very unfortunate property of a Pascal implementation).

Within ISO 7185 Pascal, there are two characteristics of an implementation that could cause P5 to not compile, even if the implementation otherwise completely complies with the standard:

1. Conflict with extended keywords.
2. Character formats.

The first concerns an implementation that defines a new keyword conflicting with an identifier used in P5. For example, if your compiler has an extended keyword "variant", this would cause pcom.pas not to compile: it uses that as an identifier. Ideally, the ISO 7185 Pascal option should turn off such extended

keywords, but you may have to invoke another option to do this. Such extended keywords are allowed by the ISO 7185 Pascal standard.

The second is simply that the character set in use is not specified by the ISO 7185 Pascal standard. This is rarely an issue now, because virtually all implementations are based on either ISO 8859-1 (or ASCII), or are based on a character set that contains ISO 8859-1 as a base standard (both ISO 8859 and Unicode do this).

It is also possible that an implementation may define special character formats. For example, the commonly implemented character force sequences are a special format:

'this is a string\n'

This is valid, since ISO 7185 Pascal does not specify the exact format of strings.

Fortunately, P5 does not contain nor need force sequences, so this will not cause problems.

Besides compiling pcom.pas and pint.pas, I strongly recommend you run and check at least ISO7185pat.pas. This is a fairly comprehensive test of ISO 7185 Pascal compliance.

If you wish to run the entire compliance test on your compiler, you simply need to change or create a version of compile.bat and run.bat for your implementation that operate with your compiler. Then you can run the regression test ***without*** the self compile features (cpcoms and cpints).

Finally, building P5, and then running it through a full regression is itself a good final test of ISO 7185 compliance. It does not substitute for direct testing of your compiler. P5 could well run correctly even if your compiler is not fully ISO 7185 Pascal compliant!

For further details concerning the ISO 7185 tests, see 18 "Tesing P5”.

## Notes on using existing compilers

### GPC

GPC (GNU Pascal Compiler) is used in the following version:

GNU Pascal version 20070904, based on gcc-4.1.3 20080704 (prerelease) (Ubuntu 2.1-4.1.2-27ubuntu2).

Copyright (C) 1987-2006 Free Software Foundation, Inc.

I have had several difficulties with other versions of GPC, which give errors on standard ISO 7185 source, or crash, or other difficulties. The GPC developers announced they were halting development on GPC in the gpc mailing list. Please see their web page:

<http://www.gnu-pascal.de>

For any further information.

The main difficulty with GPC vis-a-vie P5 is that testing of the GPC compiler for ISO 7185 compatability was not regularly done on GPC releases. Thus, otherwise working GPC releases were not able to compile and run standard ISO 7185 source code.

Because of this, I can only recommend the above version of GPC be used, which compiles and runs P5 error free.

In addition, please be aware that I have not run the GPC compiler, including the above version, through a current ISO 7185 compliance test such as appears here. My only concern is that GPC be able to complile and run P5, and that the resulting P5 runs the compliance tests. I leave it for others to run full compliance for GPC itself.

#### GPC on Cygwin

The current Cygwin release as of 2012/03/26 does not work, since it uses GPC 2005, and is broken at that (it has the .dlls for GPC installed incorrectly).

A procedure to use GPC under the current Cygwin I have used is as follows:

1. Install the latest version of Cygwin (the one I tried is Cygwin/X, a very useful package).
2. Place c:\cygwin\bin on your path.
3. Go to the website:

<http://www.gnu-pascal.de/binary/cygwin/>

And download and install:

[gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz](http://gnu-pascal.de/contrib/chief/win32/cygwin/gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz)

(4.4mb, gpc-20070904, based on gcc-3.4.4, with gcc-3.4.4 support files)

1. After installing this package in an appropriate directory, say c:\gpc, modify your path to include c:\gpc\usr\bin ahead of the c:\cygwin\bin directory in the path.
2. Add cygwin=nodosfilewarning (as stated in section 15 “Reliance on Unix commands in the P5 toolset”).

Now you will be able to follow the normal gpc instructions here to get p5 running, using the standard Windows command shell. Note that this trick won’t work with the command shells Cygwin provides.

To reiterate the steps that follow:

$ configure gpc Configure for GPC compiler.

$ build Build the P5 binaries.

$ regress Run the regression suites to check the P5 compiler.

#### GPC for mingw

Mingw (Minimal GNU for Windows) is a different port of the GNU catalog for windows that runs directly on windows. That is, each binary is statically linked with its support library, and it is designed to work with windows directly.

As Cygwin has become more and more a full emulation of the Unix environment (a good thing), it has become less usable in interaction with other Windows programs. Thus I have found the mingw package more cooperative for every day Windows work.

Mingw does not come natively with GPC installed (or much else). I recommend you also pick up the MSYS package for mingw, which is a series of GNU programs that are compiled to run in the windows environment using Mingw.

To get the mingw distribution of GPC, follow the steps:

1. Go to the website:

<http://www.gnu-pascal.de/binary/mingw32/>

And download and install:

[gpc-20070904-with-gcc.i386-pc-mingw32.tar.gz](http://gnu-pascal.de/contrib/chief/win32/cygwin/gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz)

(4.4mb, gpc-20070904, based on gcc-3.4.5, with gcc-3.4.5 support files)

1. After installing this package in an appropriate directory, say c:\gpc, modify your path to include c:\gpc\usr\bin directory in the path.

Note that this is based on a slightly different version than Cygwin.

To reiterate the steps that follow:

$ make\_flip Create flip.exe because msys does not have one. Note that you will need to move the resulting flip.exe to your bin directory.

$ configure gpc Configure for GPC compiler.

$ build Build the P5 binaries.

$ regress Run the regression suites to check the P5 compiler.

Note: The bash command shell does not work with programs generated by GPC. It gives and error when executing them. Thus it is necessary to use the standard command shell in conjuction with Mingw utilities.

# Files in the P5 package

Note: for script files, both a DOS/Windows (X.bat) and bash script (X) are provided. Their function is identical, one is for use with the DOS/Windows command shell, the other for bash shell.

build.bat

build Build the compiler and interpreter. Just runs cpcom and cpint.

clean.bat

clean Cleans generated and temp files out of the directories. This is used, for example, before generating a .zip file version of the system.

compile

compile.bat Batch mode compile for P5. It takes all input and output from supplied files, and is used by all of the other testing scripts below. You will need to change this to fit your particular Pascal implementation.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

configure.bat

configure Sets the current compiler to use to create P5 binaries.

cpcom

cpcom.bat Script file to compile the compiler side of P5 to binary on your computer.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

cpcoms

cpcoms.bat Self compile, run and check the pcom.pas file. This batch file compiles com.pas, then runs it on the interpreter and self compiles it, and checks the intermediate files match.

cpint

cpint.bat Script file to compile the interpreter side of P5 to binary on your computer.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

cpints

cpints.bat Self compile, run and check the pint.pas file. This batch file compiles pint.pas and iso7185pat.pas, then runs pint on itself and then runs iso7185pat.pas, and checks the result file.

diffnole

diffnole.bat Runs a diff, but ignoring line endings (DOS/Windows vs. Unix).

doseol

doseol.bat Fixes the line endings on text files to match the DOS/Windows convention, CRLF.

fixeol

fixeol.bat Arranges the line endings on bash scripts to be Unix, and those of the DOS/Windows scripts to be DOS/Windows line endings. This is required because the editors on the respective systems insert their own line endings according to system, and this can cause problems when they are run on a different system.

flip.c C program to replace the local version of “flip”, the Unix line ending fixup tool. It is provided in source form here because not all Unix installations have it (for example MAC OS X didn’t have it). This allows you to compile it yourself for your target system.

iso7185.html

iso7185.pdf The full ISO 7185 Pascal standard, in html and pdf forms.

iso7185rules.html A description of the ISO 7185 Pascal language.

make\_flip

make\_flip.bat A script to compile deoln and ueoln and create a flip script for Unix. This is used to replace the “flip” program if required.

news.txt Contains various information about the current release.

p5

p5.bat A batch file that compiles and runs a single Pascal program. You will need to change this to fit your particular Pascal implementation. It uses input and output from the terminal, so it is a good way to run arbitrary programs.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

pcom

pcom.exe The IP Pascal compiled pcom binary for Windows/Unix. See comments in 12 “Compiling and running Pascal programs with P5” for how to use this. All of the supplied batch files are customized for this version.

pcom.pas The compiler source in Pascal.

pint.exe The IP Pascal compiled pint binary for Windows. See comments in 12 “Compiling and running Pascal programs with P5” for how to use this. All of the supplied batch files are customized for this version.

pint.pas The interpreter source in Pascal

prtprt.bat Batch file, takes all of the rejection test error files and concatenates them to standard\_tests/iso7185prt.lst.

readme.txt Brief introduction to the project, it points to this document now.

regress

regress.bat The regression test simply runs all of the possible tests through P5. It is usually run after a new compile of P5, or any changes made to P5.

run

run.bat Batch mode run for P5. It takes all input and output from supplied files, and is used by all of the other testing scripts below. You will need to change this to fit your particular Pascal implementation.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

testpascals

testpascals.bat Runs a compile and check on pascals.pas, the Pascal subset interpreter created by Niklaus Wirth.

testprog

testprog.bat An automated testing batch file. Runs a given program with the input file, delivering an output file, then compares to a reference file.

Testprog is used to test the following program files for p5: hello, roman, match, startrek, basics and iso7185pat.

the\_p5\_compiler.doc

the\_p5\_compiler.docx

the\_p5\_compiler.html

the\_p5\_compiler.pdf This document in various forms, word 2007, word 1997, PDF, and HTML.

The\_Programming\_Language\_Pascal\_1973.pdf

Niklaus Wirth's description of the Pascal language, the last version to come from ETH. This is the equivalent of the "Report", from "Pascal user's manual and report [Jensen and Wirth].

todo.txt Contain a list of "to do" items in P5.

unixeol

unixeol.bat Fixes the line endings on text files to match the Unix convention, LF.

Zipp5.bat Creates a zipfile for the entire p5 project. This is used to create the releases available on the p5 web site.

## Directory: gpc

This directory contains scripts specifically modified for GPC.

compile

compile.bat The GPC specific version of the compile script.

cpcom

cpcom.bat The GPC specific version of the compile compiler script.

cpint

cpint.bat The GPC specific version of the compile interpreter script.

p5

p5.bat The GPC specific version of the p5 script.

run

run.bat The GPC specific version of the run script.

## Directory: gpc/linux\_X86

pcom

pint Contains binaries compiled by GPC for Linux/Ubuntu

## Directory: mac\_X86

A placeholder for Mac OS X binaries.

## Directory: gpc/standard\_tests

iso7185pat.cmp Contains the compare file for iso7185pat for gpc.

iso7185pats.cmp Contains the compare file for iso7185pats for gpc.

## Directory: gpc/windows\_X86

pcom.exe

pint.exe Contains binaries compiled by GPC for Windows.

## Directory: ip\_pascal

This directory contains scripts specifically modified for IP Pascal.

compile

compile.bat The IP Pascal specific version of the compile script.

cpcom

cpcom.bat The IP Pascal specific version of the compile compiler script.

cpint

cpint.bat The IP Pascal specific version of the compile interpreter script.

p5

p5.bat The IP Pascal specific version of the p5 script.

run.bat

run The IP Pascal specific version of the run script.

## Directory: ip\_pascal/standard\_tests

iso7185pat.cmp Contains the compare file for iso7185pat for IP Pascal.

iso7185pats.cmp Contains the compare file for iso7185pats for IP Pascal.

## Directory: ip\_pascal/windows\_X86

pcom.exe

pint.exe Contains binaries compiled by IP Pascal for Windows

## Subdirectory: sample\_programs

basics.pas A tiny basic interpreter in Pascal.

basics.inp Input test file for basics. In fact, it is a basic verion of "match" above.

basics.cmp Compare file for basics.

hello.pas One of several test programs used to prove the P5 system. This is the standard "hello, world" program.

hello.inp Input to hello for automated testing.

hello.cmp Hello compare file for automated testing.

match.bas The basic version of the match game.

match.pas A game, place "match" a number game.

match.cmp Compare file for match automated testing.

match.inp Input file for match automated testing.

pascals.pas Niklaus Wirth’s Pascal-s subset interpreter..

pascals.cmp Compare file for pascals automated testing.

pascals.inp Input file for pascals automated testing.

roman.pas A slightly more complex test program, prints roman numerals. From Niklaus Wirth's "User Manual and Report".

roman.inp Input file for roman automated testing.

roman.cmp Compare file for roman automated testing.

startrek.pas The startrek game.

startrek.inp Startrek input file.

startrek.cmp Startrek compare file.

## Directory: standard\_tests

iso7185pat.cmp Contains the output from the PAT file with the IP Pascal compiled P5 executables above.

iso7185pat.inp The input file for the Pascal acceptance test.

iso7185pat.pas The Pascal Acceptance Test. This is a single Pascal source that tests how well a given Pascal implementation obeys ISO 7185 Pascal. It can be used on P5 or any other Pascal implementation.

iso7185pats.cmp Contains the output from the PAT file resulting from the cpints run. This is slightly different than the normal run.

iso7185prt.bat Compiles the Pascal rejection tests.

iso7185prtXXXX.cmp Comparision file for pascal rejection test. XXXX is a four digit number. See the rejection tests text for information.

Iso7185prtXXXX.inp Contains the input file for the Pascal rejection test XXXX.

Iso7185prtXXXX.pas Contains the source file for the Pascal rejection test XXXX.

# Differences between Pascal-P4 and Pascal-P5

## Viewing changes

The difference set between P4 sources and P5 sources, as available on <http://www.standardpascal.com>, were used in this text. This can be done with a standard Unix “diff” command, however, I recommend an advanced visual difference such as WinMerge or similar be used. This type of source difference analyzer is simply better suited to deal with a complex set of changes such as I will present here.

## Notes about change descriptions

For references in the code, I have used a different typeface, as in:

reference

This means you will find the symbol in the code.

I don’t comment on simple formatting changes. While as I said earlier I attempt to limit formatting changes, I did make them occasionally where they might make the source more clear.

I have not used line numbers in the description. Instead, I have described the code from top to bottom and pointed out landmarks, such as “in the types section”, “in procedure x”, etc. I think this works better than line numbers, which are obsolete whenever changes are made to the source.

As with all Pascal sources, the case of the identifiers is not to be taken literally. If an identifier appeared at the start of a sentence in this text, I capitalized it.

## Changes to the parser

### Thematic changes

Most of the changes to P4 involved single features that need to be added or changed in the compiler. However, there were a few things that needed to be generally changed in the system.

#### Variable strings

First, the allocation of strings was changed from the fixed strings used in P4, to a type of dynamic string that can take any length. The method used for this appears in the Pascal FAQ:

<http://www.standardpascal.com/pascalfaq.html#Q. Can variable length strings be used in stand>

Basically, a string becomes a series of linked short arrays whose length is referred to as a “quanta” of string characters. In the release compiler that is 10 characters. The string itself is a pointer to a list of records containing quantas and pointers to the next quanta record.

The string itself is a space padded string, just as the prevous strings in P4 were. The constant record contains the length of the string, and the value of the string simply changes to a list of string quanta instead of the string itself.

There are a series of routines that deal with the quanta based string form, such as finding the padded length, writing them to the output, interchange between quanta based and fixed length string forms, and comparisions.

The string quanta representation is used both for the internal storage of strings, and for symbols in the compiler. This solves the ISO 7185 need to keep symbol characters significant no matter how many characters appear.

#### Recycling based on dispose

The other major change within the compiler is the removal of the use of mark and release procedures and replacement with ISO 7185 dispose. The mark and release system consisted of saving the heap address in a variable, then using that to reset the heap pointer later to “cut” heap so that all variables allocated between the mark and the release are simply lost. It is a very dirty and machine dependent operation that by rights was not included in the standard. It was used because it takes all the work of tracking and returning dynamic variables to free storage.

With mark/release gone, the principle requirement was to accomplish “tear downs” of all of the dynamic structures when a block left scope. There are several routines that help with recycling, and there are a series of counters that track the entry of new variables in the heap, and the exit of previous variables from the heap. These were used to “prove out” the new storage management system. If the counters are not zero at the end of any run, there is a storage leak somewhere.

#### Files

Files had to be extensively reworked in P5. In P4 files were limited to text types. However, the upgrade of files to any type had far reaching consequences. In P4, the handling of the file buffer was just passed on to the host. With any possible component type, file buffers must be managed just as formal variables.

Also, P4 only allowed the four predefined files, input, output, prr and prd to be accessed. In P4 files were recognized by their address on the stack. This does not work when an arbitrary number of files can appear in the program.

In P5, a file is defined as a logical number of 1 byte length, followed by a buffer variable for the file, that matches the component type of the file. The result is that when the address of the file is known, both the logical id of the file, and the buffer variable for it can be accessed.

The logical file number is used to index into a logical file table in the interpreter that associates a non-zero file with one of 255 actual files. The initialized state of 0 for a file id indicates that the file was never opened, and this is used as a flag to allocate a file table entry for it.

The runtime association of a logical file id to a real file means that file types themselves don’t need to be converted to and from a byte format in the pseudo machine.

#### Byte oriented pseudo-machine

The original P4 pseudo-machine was based on having a single word for all instructions and operands that was formed around the native word size of the machine. This was a very good and efficient plan for the original CDC 6000 computer series, and it is not a bad plan for todays machines, either. However, having instructions and operands that are quantitized in bytes is a better match for todays machines, and results in better utilization of memory.

Thus, the P5 pseudo-features both instructions and operands packed along byte boundaries. Each instruction is a number from 0 to 255 and fits in a single byte. The P and Q operands of the P4 machine are optional by instruction and P occupies one byte, while Q (which is commonly used for immediate integers and addresses) occupies the basic word length of the machine.

The result is an instruction set that only takes as many bytes as the particular instruction requires, and data in memory that occupies its inherent size. Bytes for characters and boolean, words for addresses and integers, reals on their required length, and sets on their required length (typically 32 bytes or 256 bits, one for each possible character in a byte).

The P5 machine does ***not*** feature addresses and integers sized for their length. If an integer immediate is 3, it does not get only one byte of storage, it gets a full word. This is a common optimization in such implementations of a byte machine as JVM or UCSD that enable programs to be packed in less space. I tried a sample implementation of packing immediate integer operands in their native byte size, and the result was code for the compiler itself (always a good benchmark) that occupied about %10 less space, but complicated pint quite a bit. I didn’t think that this fit P5’s goal of being the simplest possible implementation of the full Pascal language so I took it back out. It ***is*** a good optimization, especially if all of the possible compressions are used together to result in a reduced code and data requirement. However, I decided it belonged to a later version of Pascal-P.

### Reading the source code

#### Exit label

The first change is the addition of an exit label, 99. This is a “Wirthisim”, and I am sure he meant 99 to mean “the last label in the program”, implying he thought that less than 100 labels were all a typical program should need or want.

There is only one direct error exit in the program, which I added for a compiler fault. All other (P4) errors let the parser continue.

#### The machine parameter block

P5 starts with the machine parameter block in the constants section. For the most part, this is simple rearrangement of the existing constants to highlight which constants control the characteristics of the target machine. P4 had such a machine parameter list, I simply highlighted it.

You might say that the characteristics of a pseudo machine don’t vary, and are not important. However, Pascal-P is designed to target real machines as well, and very much treats its pseudo machine as a real machine with real sizes.

The isolation of the machine parameter block both highlights it and makes it easier to copy and paste it to the interpreter program pint. Of course, ideally this would be done automatically by an #include or similar statement, but the use of such a block is a workable solution.

Thus you will find that several parameters not important to the interpreter have been moved down or out. Maxint was removed, meaning that the size of some items comes from the host compiler. This was done because maxint is used both to specify things only within the parser, as well as machine specific parameters. Probally a better way would be to have a separate maxint just for the target.

Displimit, maxlevel and strglgth (string length) were moved down, since they are not machine specific. Lcaftermarkstack is moved down and equated to a new machine parameter, marksize. I needed to represent the size of a mark in the interpreter, and carrying the parser oriented lcaftermarkstack name down didn’t seem appropriate. So lcaftermarkstack lives, but simply as an equation to the machine parameter marksize.

After that, we find what is new with the P5 compiler. Heapal sets the alignment for the machine, but is unused in the parser. This is an example of “standardizing” the machine specific parameter block. Maxresult sets the size of the largest result a function can return, and this is needed now on machines where there is a difference between that and other types, say, integer. Marksize was covered above.

Nilval is an interesting constant, and represents the value assigned to pointers that do not point anywhere. You would expect it to be zero, but this means that variables that were cleared to zero, and not specifically assigned a nil value, are also effectively nil. Having nil be a non-zero value adds the ability for P5 to check for uninitalized pointers (unfortunately one of the few types where uninitialized checking is easy).

#### Other constants

Maxlevel was increased to follow the increased ability of pint to handle nested procedures. This also required a change to displimit. Displimit must be greater than maxlevel by definition, since there are more levels used than procedure or function levels at any given time, because records add a scoping level outside of procedures and functions.

Strglgth (string length) is an interesting constant. It used to be the absolute determinant of string lengths in the parser, but the parser now has the ability to represent any length string via string quanta. It still does that job, but all strings no longer are simply sized as arrays of characters with strglgth length. Instead, the buffers used to read strings in are still such arrays, then the string quanta system is used to store them. Thus, strglgth being a large value, the size of a maximum input line, no longer wastes space in the parser.

You will find the constant 250 several places as the effective limit of an input line. This is a “Scottisim”, that is an old habit of rounding off 256 for the length of a line buffer. The curious thing is that nowhere in the parser is there an input line buffer ! Thus, 250 simply stands for the maximum size of runs of characters we allow in the input (this could be removed entirely by making all string buffers variable length).

Digmax was converted from an integer variable to a constant. It was treated as a constant in P4, set but never modified. In P5, since reals are stored as variable length strings, it is the size of the maximum length of digits we expect in the source, and thus is 250 for the same reasons as strglgth.

After that, there are a series of manifest constants. This was simply my annoyance with the magic numbers used to create the tables in the parser. Maxsp, maxins, maxids, maxstd, maxres, reslen, prtlln, intdeff, reldeff, chrdeff and boldeff are all examples of such constants, and explained in the comments. In this case, the magic numbers are simply moved a short distance from the variables section for the program block to the constants section, but it reflects my habit of wanting to see all key adjustments to a program up high in the constants section.

Varsqt is a new constant and represents the size of a string quanta (discussed previously). Changing this constant up or down will effectively change the amount of string space wasted in “last block breakage” for strings. If a string occupies 41 characters, and varsqt is 10, then 9 characters are wasted in the last quanta of the string. Changing this value lower means less padding waste, but increases the overhead due to the need for multiple quanta and pointers. Changing it higher increases padding waste, and decreases the total pointer overhead.

The debug flag dodmplex causes all of the input symbols to be dumped when the lexical analyzer discovers them. Doprtryc dumps all of the recycling tracking counts at the end of the parser run. These flags probally should have been tied into the option comment system, which uses the

(\*opt+/-\*)

Format.

The version numbers come from the fact that P5 now signs on. The current version number, 0.8, simply comes from P5 being “unreleased, 8th version”, which should probally be changed to a full (1.x) release.

I think of the major number as being a release number, that is, counting the number of times that a major change was released to users. The minor number is simply marks a series of changes. In fact, for internal projects here I add a third number that changes with each build.

#### Types

Marktype is a vestige of the mark/release system and is commented out. When I started changing P4 to P5 I tried to preserve the original compiler and changes via commenting things out, but this rapidly became unmanageable. This is a leftover.

For symbol, the list of reserved symbols, forwardsy was removed because forward is not a reserved word in ISO 7185. Nilsy was added because nil is a reserved symbol in ISO 7185.

For chtp or character types, chlcmt was added because we added “{“ as an opening to the new style of comments {}.

Strvs is the declaration of a single quanta for a variable length string, and strvsp is a pointer to such an entry.

Strvsp is then used to replace a fixed length string sval in the record constant, and fixed length string name in identifier. Note that in constant, the string length is held in slgth, and in identifier, the string is a right padded identifier whose length is inherent. This is, again, why strvs does not need a length as part of its structure.

In the record structure, I added a next structure pointer, which is used to form lists for the new total recycling system. In general, you see these new recycling links added only when there was not already a linked list system of representation that could be followed to complete the recycling teardown. Fortunately, most structures already had such a list representation.

Also in structure, the boolean packing was added to mark the packed/unpacked status of structures. This was needed for ISO 7185 compliance.

Again in structure, the boolean matchpack indicates to the type compare routine comptypes if packing status matters. This is because set constants appearing in the code assume the packing status of the other operand in expressions, and thus are disabled for packing checks.

Finally, in structure, a field was added just to records as recyc that contains a list of structures appearing in the field list for a record. This was needed to keep track of such entries for the new recycling system.

The original alpha, which signified “a standard identifier string” in P4, was replaced with two fixed string types more appropriate to each use of the type. Idstr is used for external filenames and general identifiers. restr is used for reserved words.

Nmstr is a new declaration for a string buffer that contains the characters making up a real. It is used by insymbol. P4 and P5 both treat reals (but not integers) as just the string of characters that make up the number. This is how Pascal-P keeps from imposing any of the characteristics of its underlying real type on the input number. The convertion to a binary type is left entirely to pint.

Csstr is also new, and it is the buffer type used to store constant strings while they are being parsed.

Identifier had name changed to accept variable length identifiers as described (out of order) above.

Identifier also gets the boolean flag keep which is used to keep parameter ids out of the recycling system when a block is torn down. This is necessary because parameter lists need to stay around even after the block the procedure or function defined is sent back to recycling.

Identifier contains a series of variants for the type of object it names. Proc and func mark both procedures and procedure or function parameters. These then need a frame offset address as pfaddr, just as vars do.

Pflist is needed because when we tear down a procedure or function block and recycle the entries, the parameter list for a procedure or function is removed from the general storage system. So the keep flag above saves the list, and pflist is used to keep that list around so that we can check for parameter list congruence.

Testp and the testpointer record was eliminated because type comparisions no longer recursively examine types for structural equivalence. This is one of the few areas where the ISO 7185 standard actually simplified the code.

The labl record gets quite a few more fields to cover the requirements on goto labels that they not reference deeper nested structures, and to allow intraprocedural gotos. The algorithim is outlined in “A model implementation of standard Pascal” [Welsh & Hay].

The vlevel field saves the procedure nesting level at the point the label is defined. It is used to output the difference between the nesting level at the time of a goto and its target in order to adjust the stack frame.

The slevel field saves the statement nesting level at the point the label is defined. It is used to check for references to deeper nested statements, and to flag inactive statement blocks.

Ipcref is used to flag if the label was targeted by an intraprocedural goto, and can produce an error if the label is nested at greater than level one in the defining statement of the label.

Minlvl tracks the minimum statement level of all the referencing gotos for a label. It is used to flag an error if a goto references a deeper nested label.

Bact flags if the containing statement block for a label is active. If the label is in a block that is no longer active, this flag is reset. This is how P5 checks for gotos between statement levels where the goto is at the same or greater statement nesting, but in a different statement block. This algorithim will be covered later.

Finally, the type declaration for case statement tracking entries was moved from the casestatement procedure to the global types section. This was mainly done so that the recycling routines getcas and putcas procedures could be global, and thus is perhaps a questionable change. I suspect that it was a matter of wanting to see the recycling control routines lined up in the same section.

#### Variables

The ISO 7185 required declaration of prr as text type is bracked by special comments that an external sed script uses to change the file pcom.pas to pcomm.pas in order to accomplish a self compile. The sed script changes:

{elide} to {

And

{noelide} to }

And thus converts:

{elide}prr: text;{noelide}

To

{prr: text;}

Commenting the statement out for self compile.

The reason why this needs to be done is because of the way program parameters are classified in ISO 7185. If the parameter is “known”, that is, a compiler defined parameter of the program such as input or output, then it automatically possesses the compiler defined type, which is usually text. If the parameter is not predefined, then it’s type must be made clear via a definition in the global variables.

Prr is “unknown” when pcom is compiled by another compiler. However, to P5, prr is “known”, and so should not appear in a global variable definition.

Fortunately, there are few of these sorts of changes required for the self compile. I have marked them in comments with “!!!” to make them easy to find.

The variable id, used to contain the last identifier seen by the scanner, is changed from the general alpha type used in P4 to the more specific idstr in P5. Note that this results in a fairly large buffer for identifiers at the global level, but whenever id is copied into a specific symbol, it is changed to our variable length format.

Kk gets a constant define maxids for its maximum value, part of the “magic number” elimination drive.

Outputptr and inppptr are identifier pointers that are set to the entries for the input and output predefined files. These pointers are used to gain access to the actual definition of the files in P5 when they are used as defaults in statements like read and write. In P4, they were simply referred to by their fixed offset addresses in the program stack frame.

Stalvl was added to keep track of statement nesting. Each statement, if, while, with, etc., adds a level. If the statement nests like a begin, the statement level also increases. This is used to track goto labels.

Globtestp was removed as described above.

Display, which is an array of records that keeps track of the block nesting levels of the program, is central to the new recycling system. In fact most of the tedious process of tearing down and recycling all of the entries in a nested block can be done by reference to the display record. In accordance with this, two new fields were added to the display to keep a list of things that were not explicitly tracked in P4. Fconst was added to track constant entries, and fstruct was added to structure entries. Both of these entry types, constant and structure, received next list links in P5, and so form a linear list attached to a display record.

In errlist, the field nmr range was increased from 1..400 to 1..500, reflecting the true span of error numbers in use. The error 500 was in use in P4, it indicated a compiler error in the gentypindicator routine for an unexpected type.

Between the global variables rw and pdx, what follows are a series of magic number eliminations which were covered in the constants section. The only change not of that nature is the change in na from alpha to restr, which is a change from the general string type alpha to a more specific type restr for reserved word strings.

The pseudoconstant digmax was moved back to the constants section, as detailed above.

The variables section finishes out with several new variables.

Inputhdf and outputhdf are booleans that indicate if the input or output file appeared in the program header file. They are used to generate errors if the default file is used in a read or write, but does not appear in the header.

Errtbl is an array of booleans, one per possible error number, that are used to track the total set of errors that occurred in the program. This is used to generate a new feature in P5, which is a dictionary of error number equivalences in the program. This was done in Pascal-S, and I decided to carry it over to P5. The saving of memory no longer was warranted, and Pascal-P could now support the long strings required, so it was included. It is perhaps the only example in P5 of an improvement that was not absolutely required by the standard.

Toterr is a counter for all of the errors in the program, and is used for a diagnostic at the end.

The recycling counters that appear next, strcnt, cspcnt, stpcnt, ctpcnt, tstcnt, lbpcnt, filcnt, and cipcnt, are incremented for each new dynamic storage record that is allocated, and decremented when these records are freed. These counters are how the new/dispose system of P5 was validated after replacing the mark/release system of P4. If the counters are zero after the run, the system works. If any counter is positive 1 or more, then one or more of the entries failed to be disposed of. If any counter is negative 1 or less, then one or more of the entries was disposed of multiple times.

The tracking counters can be printed by constant flag at the end of the run. They are normally not printed, but can be turned on for a health check of the dynamic storage system.

F and I are used to print the error table at the end of the program block.

#### Procedures and functions

For the rest of the program, here is a map of the program, procedure and function blocks from the entire program:

program pascalcompiler

procedure getstr

procedure putstrs

procedure getlab

procedure putlab

procedure pshcst

procedure putcst

procedure pshstc

procedure putstc

procedure ininam

procedure putnam

procedure putnams

procedure putdsp

procedure putdsps

procedure getfil

procedure putfil

procedure getcas

procedure putcas

function lcase

procedure lcases

function strequri

procedure writev

function lenpv

procedure strassvf

procedure strassvr

procedure strassvd

procedure strassvc

function strequvv

function strltnvv

function strequvf

function strltnvf

function strchr

procedure strchrass

procedure prtdsp

procedure endofline

procedure errmsg

procedure error

procedure insymbol

procedure nextch

procedure options

procedure enterid

procedure searchsection

procedure searchidne

procedure searchid

procedure getbounds

function alignquot

procedure align

procedure printtables

function stptoint

function ctptoint

procedure marker

procedure markstp

procedure markctp

procedure followstp

procedure followctp

procedure genlabel

procedure searchlabel

procedure newlabel

procedure prtlabels

procedure block

procedure skip

procedure constant

function string

function comptypes

function filecomponent

function string

procedure typ

procedure simpletype

procedure fieldlist

procedure labeldeclaration

procedure constdeclaration

procedure typedeclaration

procedure vardeclaration

procedure procdeclaration

procedure pushlvl

procedure parameterlist

procedure body

procedure addlvl

procedure sublvl

procedure mes

procedure putic

procedure gen0

procedure gen1

procedure gen2

procedure gentypindicator

procedure gen0t

procedure gen1t

procedure gen2t

procedure load

procedure store

procedure loadaddress

procedure genfjp

procedure genujpxjp

procedure genipj

procedure gencupent

procedure genlpa

procedure checkbnds

procedure putlabel

procedure statement

procedure expression

procedure selector

procedure call

procedure variable

procedure getputresetrewriteprocedure

procedure pageprocedure

procedure readprocedure

procedure writeprocedure

procedure packprocedure

procedure unpackprocedure

procedure newdisposeprocedure

procedure absfunction

procedure sqrfunction

procedure truncfunction

procedure roundfunction

procedure oddfunction

procedure ordfunction

procedure chrfunction

procedure predsuccfunction

procedure eofeolnfunction

procedure callnonstandard

procedure compparam

procedure expression

procedure simpleexpression

procedure term

procedure factor

procedure assignment

procedure gotostatement

procedure compoundstatement

procedure ifstatement

procedure casestatement

procedure repeatstatement

procedure whilestatement

procedure forstatement

procedure withstatement

procedure programme

procedure entstdnames

procedure enterundecl

procedure exitundecl

procedure initscalars

procedure initsets

procedure inittables

procedure reswords

procedure symbols

procedure rators

procedure procmnemonics

procedure instrmnemonics

procedure chartypes

procedure initdx

For each procedure or function that was changed, added or deleted, a new heading will appear.

Two major sections of support procedures and functions were added globally. The first covers recycling using new/dispose. The second covers string handling using our new variable string length system.

#### Recycling support routines

When allocating control records, the method followed is to introduce a pair of procedures, get and put, for each record type. This gives a central point to control the method of allocation and tracking of each type.

Some types have variants that are too complex for the simple get and put scheme. The variants would have to appear as parameters to the get routine. So the method followed there is to let the main code allocate the record with new, then provide a routine that completes the processing on the created entry.

Recycling of control records falls into two classes. The first are entries that are directly rooted in the display. The second are records that can be recoved as links from the first type of entries. The reason it matters here is that the display rooted entries have their get processing combined with a push to the list that is rooted to the display, and are termed psh (for “push”) procedures, because all of the lists are represented as pushdown lists.

##### Procedure getstr

Getstr gets a single quanta, or element, of a variable length string.

##### Procedure putstrs

Putstrs recycles a list of string quantas that make up a variable length string. Variable length strings are built up one quanta at a time, but released as a set.

##### procedure getlab

Gets a single label entry.

##### procedure putlab

Releases a single label entry.

##### procedure pshcst

In the case of constants, the number of variants make it inconvienent to follow the get/put rule. Instead, pshcst is used to place a constant record to the display rooted list, then perform the tracking counts.

##### procedure putcst

Releases a constant record. Also releases the set of string quanta contained, if the constant is a string.

##### procedure pshstc

Pushes a structure record to the display rooted list and counts it.

##### procedure putstc

Releases a structure record.

##### procedure ininam

Performs processing on an identifier record. Identifier records are kept in a binary search tree rooted to the display, and have complex variant structure. So this routine simply finishes allocation processing, and thus bears an ini prefix.

##### procedure putnam

Releases an identifier record. Also releases the variable length string that keeps the identifier name itself. Finally, putnam takes care of releasing parameter lists for procedures and functions. Parameter lists are kept out of the recycling for the block that contains them because the surrounding block needs to refer to them to check the structure of passed parameters in a call. This is what the keep flag does.

Thus, when the procedure or function entry that contains the parameter list is recycled, it is time to recycle the parameter list as well.

##### procedure putnams

Releases an entire tree of identifier names. Identifier records are rooted in display and have a binary tree structure. Thus, this procedure tours the tree and performs a depth first pruning of the tree. It won’t touch records with the keep flag on, which is a bit confusing with respect to the putnam comments. However, at the time of putnams, the identifiers that reference the parameter list are removed from the block symbol list, but kept on their associated procedure or function header list. This is a consequence of the unification of identifier name records and the structuring information about what the name contains.

##### procedure putdsp

Putdsp is the key routine in the new P5 recycling system. Putdsp recycles a display level, that is, all of the entries that are rooted to the given display level are sent back to free storage. In the P4 system, the process of removing a display scoping level was simply a matter of decrementing the level count and performing a release procedure to remove all outstanding entries. Putdsp starts with the use of putnams to release the binary identifier tree. This is most convient as a subroutine, because it calls itself recursively for the left and right forks that form the the tree.

Next, the label entries are disposed of, followed by the constant list.

The structure list is disposed of with a subroutine putsub that checks if the structure is a record, and frees the list of structures that comprise the record if so. It also removes the identifiers that are attached to the record. The identifiers that make up the field list of the record don’t appear in the display. A display level is used to collect them when the record is defined, but the root of that tree is saved into the fstfld field of the record structure. The structures corresponding to the record structure fields themselves are stored in the recyc field, which is new with P5.

You might think that the identical looking recvar field from P4 would do this job, but this field actually points to the record variant part, if it has one.

Tagfields for records are treated specially. They are allowed to be anonymous, but take an identifier entry in any case. The problem is that they don’t also appear in the identifier tree, and so don’t get recycled. We check for this special case and perform it here.

Because putsub recurses to process all fields of a record, and entire record is torn down here.

##### procedure putdsps

Putdsp is called for a particular level number, but could be abused by being used to pull down display levels out of stacking order. However, this does not happen. The procedure putdsps is used to tear down all display levels until a given display level is reached. It might seem to make sense to only remove scoping levels one at a time, but levels are created for things like record scoping, and this is handled by saving the level at the start of the procedure or function that defines a block, then removing all levels until the block is removed, also removing any record subscoping.

##### procedure getfil

The getfil procedure simply gets and tracks an external file entry.

### procedure putfil

Putfil recycles and tracks an external file entry.

##### procedure getcas

Getcas allocates and tracks a case statement entry.

##### procedure putcas

Putcas recycles and tracks a case statement entry.

#### Character and string quata routines

The majority of the complexity for the new variable length string system in P5 is enclosed in a series of routines in this section. The buffers used in P4 to represent things such as identifiers and strings that the scanner reads in still exist, but of course are much larger. They are converted to variable string format when the buffers are copied to identifier records, or constant records. This saves us from having to modify all of the scanner code that parses and collects these values, and results in the same amount of savings in space from not having to store the buffers literally.

The string handlers are oriented around three different formats:

Idstr Which gives a maximum length of string buffer in P5.

Restr Which is used for reserved words only (like procedure, var type, etc.).

Strvsp Which is a variable string based on a linked list of string quanta.

Despite the name, idstr is used for all identifiers and string constants in buffers. Restr is simply used to define the length of entries in the reserved word table. Strvsp is represents variable strings in their “efficient” format, which is a linked list of string quanta.

For each of the fixed idstr and restr there exist convertions to the variable length strvsp type. This is used to convert fixed length buffers into variable length strings in data structures for the compiler. In addition, several cross type comparision routines exist so that fixed buffers can be compared to variable string types.

It also occasionally matters as to what the right space padded length of a variable length string is. A right padded string is a string with blanks or spaces at the right side:

‘my long string ‘

Is a 14 character, right padded string in a 20 character stored format. This may seem odd that we use this mechanisim from old, fixed length string Pascal for variable length strings. However, the string quanta system does not exactly represent the length of strings, but the length of the string rounded up to the nearest quanta, which in the release compiler is equivalent to 10. This means that, for example, if the string to be stored is 35 characters, 4 quantas must be used, for a total space of 40 characters, with 5 characters of right padding.

Variable strings in P5 have no inherent method to indicate length. For identifiers and reserved words, the right padded length (the length of the string with all right hand space padding removed) is used. For constant strings, the length of the string is actually kept as a separate field.

Many of the string routines have a naming convention based on using a few letters at the end. Unfortunately, the convention was not that evenly implemented. Never the less, here are the characters used:

R Reserved

V Variable

F Fixed (usually identifier strings)

I Fixed identifiers

D Digit string

C Constant string

If the routine only takes one parameter, then only one trailing character will appear. If it takes two parameters, there will be two characters in order of the parameters. For example, writev takes a single variable parameter. Strassvf takes two parameters, a destination variable string and a source fixed string. Writev tells you it is the standard write procedure for variable strings. Strassvf means to assign a fixed to variable string.

##### function lcase

Lcase converts upper case characters to lower case characters. It is used to establish case insensitivity, which was not present in the P4 compiler.

##### procedure lcases

Lcases converts an entire identifier string to lower case. It is a key routine to establish case insensitivity in identifiers.

##### function strequri

Strequri checks if a reserved word string is equal to an identifier string, disregarding case. This is used to compare the scanner identifier buffer to various reserved word and short strings.

##### procedure writev

This procdure is used to provide write style fielded length output capability for variable length strings. It is exclusively used for diagnostics in P5.

##### function lenpv

This function finds the right padded length of a variable length string.

##### procedure strassvf

Assigns a fixed length identifier string to a variable length string. Note that string quantas are allocated as required to create the linked list that is the result.

##### procedure strassvr

As in strassvf, but assigns a reserved word string to a variable length string.

##### procedure strassvd

As in strassvf, but assigns a number string to a variable length string.

##### procedure strassvc

As in strassvf, but assigns a constant string to a variable length string.

##### function strequvv

Compares two variable length strings without regard to case. Strings are equal only if they are also equal in length.

##### function strltnvv

Compare that one variable length string is after another in character order. That is:

‘alpha’ < ‘beta’

You will note that the length of the strings is irrelevant. Why? Consider:

‘markalpha’ < ‘marker’

Because the first would be alphabetized before the second in a dictionary.

Note that when performing string compares using a fairly complex method (compared to compared to letting the compiler do it), we only need to define the routines for equals and less than. This is because all other compare types can be derived from these two basic types:

a > b is not (a < b) and not (a = b)

a <= b is not (b < a)

etc.

##### function strequvf

Finds a variable string equal to an identifier string. This is used to compare variable strings in data tables to fixed buffers.

##### function strltnvf

Finds a variable string less than an identifier string. Why do we need a less than for identifiers, but not for reserved words? Because identifiers are stored in binary search trees, and reserved words are found via straight linear searches.

##### Function strchr

Returns a single character from the given index in a variable string. It finds which quanta the index belongs to, and retrives the character from that quanta. If the index is beyond the end of the total string, a space is returned. This emulates a string that exists right padded in a virtual buffer of infinite length.

##### Procedure strchrass

Places a single character to the given index in a variable string. It find which quanta the index belongs to, and places the character in the quanta. If the variable string does not have sufficient quanta to represent the character at the index, then any number of quantas are automatically added and set to blanks until it is.

Strchr and strchrass form the basis for higher level systems using the string quanta format. They could be used to extract or insert a substring, and perform most functions that could be done on fixed strings using the variable string format.

However, strchr and strchrass were created solely to handle the manipulations required on the variable strings that represent real number constants. Specifically, the ability to read and change the sign, and thus the first character of

##### procedure prtdsp

Closing out the section of added support routines in P5 that were not in P4, prtdsp is a diagnostic routine that was added to debug the compiler. It prints out the contents of the display identifiers in binary tree format.

You will note that P4 already had a fairly complex dump procedure printtables for the user. Prtdsp is strictly a routine for debugging the compiler.

### Modifications

For the most part, the rest of the changes to routines in the compiler are modifications to existing routines (a large exception is errmsg below). Thus you will note that we describe what is new or what has changed in the compiler on a routine by routine basis, and not that the routine does. The P4 description is still valid for the vast majority of these routines.

#### procedure endofline

Endofline gets a small section of code at the end that outputs “line markers” to the intermediate code. This allows pint to tell you which source line was associated with which error in the back end. Perhaps one of the most useful modifications to P4.

#### procedure errmsg

Errmsg was one of the few places I added a modification that was not strictly needed for ISO 7185 compliance. Given an error number, it outputs the equivalent error string. It is used to produce a dictionary of all errors that were seen in the run, very similar to what Pascal-S does. I found it invaluable not only because it avoids the need to go look up errors by number, but also because the errors for P4 were nowhere exactly defined. The “Users Manual and Report” [J&W] gives a list that is very close to P4, even though nowhere could I find a reference that states this. There were a few additional errors used in P4 that didn’t appear in this reference, and I have added yet more for ISO 7185 specific errors.

#### procedure error

Error begins with a mysterious commented out write statement. The reason for this is that the compiler does not actually output errors when they are entered, but records up to 10 errors in a table and outputs them during endofline processing. This can be quite annoying when you are debugging the compiler, because the errors are delayed until the statement causing the error is long gone. Uncommenting this write statement helps with such debugging.

Then, error records all errors that were seen in errtbl, which contains a true/false bit for every error number possible. If an error was seen, its bit is set true. This tells us what errors to output in the dictionary at the end.

You will note that a set is not used for errtbl. This is to avoid limitations on set size.

Finally, error tallies a total count of errors for the run that is output at the end.

#### procedure insymbol

Starting at the top, insymbol gets a lot less goto labels, which I consider a major blow for sanity. There was no deliberate effort to get rid of gotos in the procedure, it simply fell out of the required changes naturally. The remaining goto target, 1, is just used to restart the scanner from the top.

Insymbol gets a few new flags, including ferr and iscmte, and a few variable string pointers.

The first change, in the body of insymbol proper, was to treat control characters identically to spaces. This is made easy with ASCII/ISO 8859 character encoding, because all control characters are below a space, and printing characters above. Of course this makes the routine dependent on such character sets, but since even Unicode based programs would obey the rule, I believe it can stand.

The idea of ignoring all control characters is that the user can insert things like page feeds and any other control characters into the source without the compiler caring.

Insymbol uses the character type table chartp to classify characters, and then uses that to divide up the different sections of insymbol according to what tolken they operate on. It exclusively operates on one character at a time, which is why multiple character sequences like ‘>=’ must be handled as a check for a follow on character for ‘>’, and why things like ‘..’ must be rejected within the parsing of a number. Contrast this with the common technique where the current and follow character are simply selected from a table at the start of the scanner.

##### Letter: identifiers and reserved words

In the letter section, which parses reserved words and identifiers, we have added an error for reserved words and identifiers that are too long. In P4, identifiers obeyed the rule that only the first 8 characters were significant, which was according to the original J&W rules. In P5, overlength identifiers generate an error, but note that this would only occur on identifiers that were longer than a reasonable line length (250 characters in the current implementation).

Insymbol then searches the reserved word table to find if an identifier is a reserved word. This code isn’t changed except for the use of the strequri function to accommodate the new larger id buffer. However, it also now checks if the length of the id is less than or equal to the largest reserved word, in which case the entire search is unnecessary.

We also initialize the sy and op variables before the loop, and thus get rid of the need for a goto.

##### Number: integers and reals

After the first series of digits are parsed, insymbol looks for a possible follow character of ‘.’ or ‘e’. The first signifies a decimal point, the second an exponent. Insymbol must reject both the symbols ‘..’ and ‘.)’, which look like a decimal point, but are the range and alternate for ‘]’. The first was done in P4, the second is new with ISO 7185. To do the rejection, insymbol takes advantage of the fact that the input file buffer can look ahead one character.

The first change to number is to add a push to the constants list in the display for real constants. You’ll note that number stores real constants in a constant entry, but lets the caller to insymbol handle it if it is an integer. In both cases, the number is left as a string.

In the case of integer, the numeric string is converted to an integer. In the case of real, it is left as a string in order not to impose pcom’s binary format limits on the compiler. The convertion to binary real format is left to pint.

In P4, the real constant string is directly copied to the location in val. However, in P5 that is now a variable string, which means that P5 can store real constants efficiently. In keeping with the general theme in P5, real constants are collected instead in a buffer rvalb, then the real constant string is copied to its final location in val by the styrassvd procedure.

##### Chstrquo: Strings

Chstrquo is changed to use the pshcst to process the new constant, then the string buffer is copied to a variable string by strassvc. Unlike other variable length strings, which are inherently right padded modulo the quanta with spaces, string constants have a specific length, as represented in val by slgth.

##### Chperiod: ‘.’, ‘..’ and ‘.)’ tolkens

Here we get to say goodbye to a very old quirk of Pascal-P where the symbol ‘..’ (range) is treated identically to ‘:’ (colon). I suspect this is a holdover from a day when a range was specified as x:y instead of x..y, but I haven’t been able to find an example of such a program.

In any case, insymbol now treats ‘..’ as a range, and adds the ‘.)’ symbol (as alternate to ‘]’).

##### Chlparen: ‘(‘and comment start

Chlparen must handle ‘(‘, ‘(\*’ (comment start), and ‘(.’ (alias for ‘[‘). The follow on character of ‘\*’, ‘.’, or other character determines which. If it is a comment start, insymbol checks for a comment option embedded with ‘$’, just after the start of the comment. For comments, insymbol skips forward until the end of a comment is seen, which now must include ‘}’. The difference is handled via a flag, iscmte, to compensate for the fact ‘}’ is a single character, and ‘\*)’ is two.

##### Chlcmt: ISO 7185 comment start

Cmlcmt handles the fact that in ISO 7185, ‘{‘ can also start a comment. This case is very similar to chlparen, except that no follow on character need be parsed.

##### Lexical dump

A diagnostic was added at the end of insymbol to print the tolken that insymbol has parsed. This is simply a large (and yes, poorly formatted) case statement for all of the symbol types. Most of the symbols evaluate to a single string, but ident, intconst, and stringconst can print the actual value of the tolken as well.

#### procedure enterid

Enterid enters a new identifier into the binary tree structure at the top display. The first change was to change it from declaring a local buffer, and copying fcp^.name to it. There was no real need for this copy, fcp is invariant in the routine, so it looks like the original authors were trying to save the need to perform an indirect field access.

In any case, it no longer makes sense in P5. Fcp^.name is a variable string id, and buffering it is no longer wise. So we eliminated the local copy, and changed the compares to strqeuvv and strltnvv.

#### procedure searchsection

Searchsection takes the global id string (returned by insymbol) and finds it in a binary tree as rooted in the parameter fcp. Because identifiers are variable length strings in P5, the compare routines strequvf and strltnvf must be used.

#### procedure searchidne

Searchidne means “searchid with no error”. The behavior of searchid is that if it does not find the id is was looking for, it outputs an error and returns a “skeleton key” for each general type. This is used to suppress further errors for things like type compares.

There were a few places in P5 where we needed to search for an id, but not print an error. Fortunately, this requirement nests nicely within searchid, so we took the first half of the old searchid, had it return nil if the id was not found, then that was called searchidne and the new searchid calls it.

The ability to search without generating an error is used for cases like:

**type** r = **record** **case** mysel **of** ... **end**;

When parsing the identifier mysel, we don’t know if it is an id for the tagfield, or the type of the tagfield itself. A follow on tolken of ‘:’ tells us that, but by then, the id is gone. So a search to find if the id is a type finds this out for us.

#### procedure searchid

Searchid simply had the first half of it moved to searchidne. It is otherwise identical.

#### procedure printtables

This procedure is called when the user selects the t+ option, for “print tables” (give a compiler dump). It has a few minor changes to cover any changes in the format of table items. These are detailed by subroutines of printtables.

#### procedure followstp

Followstp was changed to use the writev procedure to output real constant min and max portions of a subrange, because real constants are now variable strings. If this code looks wrong to you, you are not alone. Why would a subrange min or max be a real? Hummm….

#### procedure followctp

Followctp had to be modified to output the identifier name field as a variable string using writev. It also is changed to output real constants and string constants this way.

#### procedure searchlabel

Searchlabel was added to support the ISO 7185 **goto** checking requirements. It finds a matching **goto** target label whose number matches the number found by insymbol in the given display level. This is returned as a pointer, or **nil** if not found.

#### procedure newlabel

Newlabel does quite a bit of the processing to allocate a new goto label. It allocates it, places the numeric value of it from the insymbol value, generates an internal number for it and sets up all of its fields.

#### procedure prtlabels

Printlabels simply dumps all of the labels in the current display goto label list. It was used to debug the goto label system, and you will notice that it is not called anywhere. This is because I would place a call to it where I needed a label dump in the code.

Of course, your compiler may warn that the routine is not used. You can suppress this warning, or delete the routine as desired.

#### procedure block

Block begins the real meat of the compiler. Its actual body does not begin until much later, which is one of the complaints about Pascal’s nested routine ability, that it can have its preamble so far from the actual body code (an issue that forward partially solves). The majority of the changes to block are in its subroutines.

In the declaration, test has been removed and placed into the routines that actually use it, which, curiously, was not block itself. Test is used for constructs like:

repeat

{ parsing code }

test := sy <> comma;

if not test then insymbol

until test;

It carries the status of the next symbol over insymbol so that it can control the loop. This is a common paradigm that eliminates the need for **goto**s and early exit constructs in Pascal.

Test was also used for more complex syntax tests.

Unfortunately, it was not only curious to have test declared, but not used in block (as Pemberton mentions), but actually wrong in places. The problem was that essentially making test global to all of block caused it to fail when a subroutine used test and changed the value of its caller. The answer was to move test to the routines that actually use it.

Block gets a single new definition, stalvl, which is a counter that keeps track of the statement nesting level. This is cleared to zero within the body of block.

#### procedure constant

Constant was modified to use the pshstc or “push structure” call to set up a structure for both string and real constants. We also set the “packing” flag on strings, since in ISO 7185, strings, which are defined as:

packed array [1..x] of char;

Must be packed. It matters now when comparing types.

When negating real constants, constant changed the sign by changing the ‘+’ character at the front to a ‘-‘ character, and vice versa. Recall that reals are kept in string format. This was changed to use the strchr and strchrass routines.

A similar method is used to insert a sign later in this routine.

#### function string

String accepts a structure pointer, and returns true if the type is string. String was forwarded ahead of comptypes, which now uses this routine. It had to be forwarded, because string uses comptypes, and so they reference each other.

In P4, string simply checked if the type was an array of character. The ISO 7185 demands are greater, and now string checks if the array is packed, and begins with an index of 1.

There is also a problem with string constants. They get an index type pointer that is nil (in factor). This is because string constants can have any given index type, which would have to be created anew on each string constant encountered. Instead, the index is simply set nil, and this is recognized where necessary.

This causes the problem with string, because it now needs the index type to be an existing type. In P4, the index type was ignored in string. Getbounds is used to fetch the lower bound, with the high bound discarded, and getbounds handles a nil type. Unfortunately, it returns 0s for the bounds, which is the wrong answer (should be 1).

So string checks if it has a nil index type, and sets the lower bound to 1, skipping the getbounds call. Now the base type of the array is compared as in P4, but the test for lower bound equals 1 is added.

#### function comptypes

comptypes takes to structure pointers and returns true if they are compatible types. The method to check if the types are compatible changes completely with ISO 7185. The good news is that it actually becomes simplier. The reason is that the P4 compiler used “structural equivalence”, meaning that these two types would be equal:

type a = record a: integer; b: char end;

b = record a: integer; b: char end;

Because the two record types have the same structure. This is difficult to implement, and P4 recursively decended through the tree that forms a type and successively compared each level. It also needed a system to prevent loops through the links that pointers represent. It was also a fairly ambiguous system, and things like:

type a = record a: integer; b: char end;

b = record a: 0..255; b: char end;

Were a problem. Are the types compatible? They might be physically different. It might depend on the packed status of the types.

In the ISO 7185 system, the above types must be identical or “named aliases” of each other. The system is easier to implement, less ambiguous, and very little added programmer effort is needed to use the system.

As a result, I find it easier to describe the way comptypes works from scratch, it is that different.

First, the result is set to be false, or no compare, by default. The first check is if the types are identical, that is, the two structure pointers passed in point to the same structure entry. Note that a named alias of a type has an identical structure pointer (see typedeclaration).

Second, we check if both pointers are not **nil**. Having a structure pointer be nil is how errors are recorded. Note that the undefined types returned by searchid are all given **nil** structure pointers in enterundecl. If either or both of the structure pointers are nil, the result is a type match. It does not mean they actually match, it means that further errors should be suppressed.

Third, we check if the form of both structures are identical. The form is the scalar, set, pointer, or other type of structure. If these are equal, we proceed to a case statement that handles each form of type. If not, we check one of the types being a subrange, then check if the base type for that subrange is compatible with the other type. This is because a subrange of a type is also compatible with that type. If both types are subranges, this also works.

Do scalars really all match? What about integer and character? Why does not scalar match these?

#### function filecomponent

#### function string

#### procedure typ

#### procedure simpletype

#### procedure fieldlist

#### procedure labeldeclaration

#### procedure constdeclaration

#### procedure typedeclaration

#### procedure vardeclaration

#### procedure procdeclaration

#### procedure pushlvl

#### procedure parameterlist

#### procedure body

#### procedure addlvl

#### procedure sublvl

#### procedure mes

#### procedure putic

#### procedure gen0

#### procedure gen1

#### procedure gen2

#### procedure gentypindicator

#### procedure gen0t

#### procedure gen1t

#### procedure gen2t

#### procedure load

#### procedure store

#### procedure loadaddress

#### procedure genfjp

#### procedure genujpxjp

#### procedure genipj

#### procedure gencupent

#### procedure genlpa

#### procedure checkbnds

#### procedure putlabel

#### procedure statement

#### procedure expression

#### procedure selector

#### procedure call

#### procedure variable

#### procedure getputresetrewriteprocedure

#### procedure pageprocedure

#### procedure readprocedure

#### procedure writeprocedure

#### procedure packprocedure

#### procedure unpackprocedure

#### procedure newdisposeprocedure

#### procedure absfunction

#### procedure sqrfunction

#### procedure truncfunction

#### procedure roundfunction

#### procedure oddfunction

#### procedure ordfunction

#### procedure chrfunction

#### procedure predsuccfunction

#### procedure eofeolnfunction

#### procedure callnonstandard

#### procedure compparam

#### procedure expression

#### procedure simpleexpression

#### procedure term

#### procedure factor

#### procedure assignment

#### procedure gotostatement

#### procedure compoundstatement

#### procedure ifstatement

#### procedure casestatement

#### procedure repeatstatement

#### procedure whilestatement

#### procedure forstatement

#### procedure withstatement

#### procedure programme

#### procedure entstdnames

#### procedure enterundecl

#### procedure exitundecl

#### procedure initscalars

#### procedure initsets

#### procedure inittables

#### procedure reswords

#### procedure symbols

#### procedure rators

#### procedure procmnemonics

#### procedure instrmnemonics

#### procedure chartypes

#### procedure initdx

# Changes to the assembler/interpreter

[tbd]

# The intermediate language

## Format of intermediate

The intermediate has the format:

<main code>

<start code>

<further prd input>

The main section contains all of the generated code, except for a startup section:

mst 0

cup 0 l 3

stp

The main section is assembled past the startup code, then the assembly location is restarted and the startup section is assembled, placed under the main code.

After both the main code and start code sections, the contents of the prd file are not read. The interpreted code can keep reading from the prd file at this point. This means that input for the prd file as used by the interpreted program can be concatenated to the intermediate file. This feature is used for the self compile and run.

For each line in the intermediate, the first character indicates:

i Indicates a comment, the rest of the line is discarded.

l A label. Used to establish jump locations in the code.

q Marks the end of the main code or startup code section.

(blank) An intermediate instruction.

: A source line marker.

Comments

A comment appears as:

i<any text>

Comments are used to generate any descriptive text in the intermediate. They are also used to output a marker every 10 intermediate instructions, of where the “logical program count” or index of instructions, is currently located in the intermediate with a marker of the form:

i n

Where n is the logical program count.

### Label

Label lines are of the format:

l n[= val]

The label number gives the logical label being defined. If “=” follows, the instruction address of the label appears, otherwise it is set to the current instruction being processed (the pc).

The logical label number is a value from 0 to n. The value can be anything, and it is used both to define parameters as well as addresses. When used in an instruction, the assembler is capable of processing forward references to labels not yet defined in the assembly.

### Source line marker

A source line marker gives the line number currently being parsed in the compiler when the intermediate is generated. It is of the form:

:n

Where n is the source line number. This information can be used to show where errors occurred in the source.

## Intermediate instruction set

What follows is a complete listing of all instructions, parameters, and numeric equivalences in the interpreter instruction set. The following instruction endings are common and indicate the type of the operation:

i Integer (4 bytes)

r Real (8 bytes)

s Set (32 bytes)

b Boolean (1 byte)

a Address (4 bytes)

m Memory (or memory block)

c Character (1 byte)

Note that all boolean and character values are extended to 4 byte integers when loaded to the stack. Reals and sets are loaded as whole entities, 8 and 32 bytes respectively.

Note that there will often not be a type indicator if there is only one type operated on by the instruction.

The operation codes are from 0 to 255 or $00 to $ff, fitting in one unsigned byte. The format of the opcodes and operands is:

Op [p] [q [q1]]

The p parameter, if it is exists, is always an unsigned byte. The q parameter, if it exists, is either one or 4 bytes long. If q1 exists, q is always 4 bytes long. q1 is always 4 bytes long. These will be refered to below as q8 and q32 (8 bit and 32 bits). The endian nature of 32 bit words will depend on the compiler used for P5.

Each instruction is listed first in the form it appears in the intermediate assembly form. The second is the format the opcode will take in interpreter store.

This list of instructions is in alphabetical order.

Instruction Opcode Stack in Stack out

abi 40 integer integer

abr 41 real real

Find absolute value of integer or real on top of stack.

Instruction Opcode Stack in Stack out

adi 28 integer integer integer

adr 29 real real real

Add the top two stack integer or real values and leave result on stack.

Instruction Opcode Stack in Stack out

and 43 boolean boolean boolean

Find logical ‘and’ of the top two Booleans on stack, and replace with result.

Instruction Opcode Stack in Stack out

chka low high 95 q32 address address

chkb low high 98 q32 boolean boolean

chkc low high 99 q32 char char

chki low high 26 q32 integer integer

chks low high 97 q32

Bounds check. The q parameter contains the address of a pair of low and high bounds values, low followed by high, each one integer in size. The bounds pair is placed in the constant area during assembly, and the address placed into the instruction.

The value on top of the stack is verified to lie in the range low..high. If not, a runtime error results. The value is left on the stack.

If the instruction was chka, the low value will contain the code:

0 Pointer is not being dereferenced

1 Pointer is being dereferenced

Pointer values are considered valid if between 0..maxaddr, where maxaddr is the maximum address in the interpreter. If the dereference code is present, it indicates a check if the pointer is nil. This instruction is only applied to dynamic variable addresses, and thus other checks are possible.

Set values are checked to see if any elements outside the elements from low..high are set. If so, a runtime error results.

Instruction Opcode Stack in Stack out

chr 60 integer character

Find character from integer. Finds the character value of an integer on stack top. At present, this is a no-op.

Instruction Opcode Stack in Stack out

cip p 113 mp address

Call indirect procedure/function. The top of stack has the address of a mp/address pair pushed by lpa. The dl of the current mark is replaced by the mp, and the address replaces the current pc. The mp/ad address is removed from stack.

Instruction Opcode Stack in Stack out

cke 188 tagcst boolean

Terminate active variant check, started by cks. Expects the “running boolean” value on stack top, followed by the current value of the tagfield. If the Boolean value is false, meaning that none of the constants matched the tagfield, a runtime error results.

Both the Boolean value and the tagfield value are removed from the stack.

Instruction Opcode Stack in Stack out

ckla low high 190 address address

Bounds check. This instruction is identical to the instruction chka, but indicates a pointer to a tagged record. At present this requires no special action over the normal processing of chka.

ckla an chka check if the pointer is dereferencing an allocation that has been freed. In the current implementation this check works on both tag listed and normal dynamic variables. Because this is dependent on the way the allocator is structured, ckla exists in case tag list pointers need special handling.

Instruction Opcode Stack in Stack out

cks 187 tagval tagval boolean

Start active variant check. The current value of the tag is on stack top. Pushes a false binary value onto the stack. This value is ‘or’ed with all of the following variant checks.

Instruction Opcode Stack in Stack out

ckvb val 179 q32 tagval Boolean tagval boolean

ckvc val 180 q32

ckvi val 175 q32

Checks a tagfield for active variants. The “running Boolean” is expected at stack top, followed by the current tagfield value. The tagfield is compared to the constant val and the Boolean equality ‘or’ed into the running Boolean. Both are left on stack.

Instruction Opcode Stack in Stack out

csp rout 15 q8 Per call Per call

Call system routine. The routine number rout indicates the routine to call, from 0 to 255 (or $00 to $ff). Note that the main difference between system and user procedures is that system calls have no framing information.

Instruction Opcode Stack in Stack out

cta off lvl 191 q32 q32 tagaddr newtagval tagaddr newtagval

Check tag assignment for dynamically allocated record. Expects the value to assign to the tagfield on stack top, followed by the address of the tagfield as second on stack. Dynamic records containing tagfields are allocated by a special procedure that creates a tagfield constant list with the constants that were used to allocate the record with variants. The cta instruction looks at the list to see if the given tagfield appears in that list, indicating that the tagfield is fixed and cannot be changed. If such a list entry exists and does not match the tagfield assignment, a runtime error results. The instruction contains the offset from the start of the record to the tagfield, and at what nesting level the tagfield exists. If uses the tagfield offset to find the start of record and look for the list, and the nesting level to find the exact tag constant, if it exists. Leaves both values on stack.

Instruction Opcode Stack in Stack out

cup p l addr 12 p8 q32

Call user procedure. The instruction contains the number of bytes in the parameter section, and the address of the procedure to call. The mp or mark pointer is set to the location of the stack mark record established by the mst instruction, then the return address is placed into the mark record. The specified location is then jumped to.

The assembly code represents the address as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

decb cnt 103 q32 boolean boolean

decc cnt 104 q32 character character

deci cnt 57 q32 integer integer

The unsigned cnt parameter is subtracted from the value at stack top.

Instruction Opcode Stack in Stack out

dif 45 set set set

Find set difference, or s1-s2. The members of the set on stack top are removed from the set that is second on the stack, and the top set on stack is removed.

Instruction Opcode Stack in Stack out

dmp cnt 117 q32 value

The unsigned cnt parameter is subtracted from the current stack pointer. The effect is to “dump” or remove the topmost cnt data from the stack.

Instruction Opcode Stack in Stack out

dupa 182 address address address

dupb 185 boolean boolean boolean

dupc 186 character character character

dupi 181 integer integer integer

dupr 183 real real real

dups 184 set set set

Duplicate stack top. The top value on the stack is copied to a new stack top value according to type.

Instruction Opcode Stack in Stack out

dvi 53 integer integer integer

dvr 54 real real real

Divide. The value second on stack is divided by the value first on stack. The division is done in integer or real according to type.

Instruction Opcode Stack in Stack out

ente l size 173q32

Sets the maximum extent of stack use. The size constant in the instruction gives a compiler calculated size that is the maximum extent of stack that will be used by this routine. If that overlaps the bottom of the heap, then a runtime error results. The ep or extreme stack pointer is updated, and also stored in the stack mark record for use in the ipj instruction to jump between procedures.

The assembly code represents the size as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

ents l size 13 q32

Enter routine and allocate local variable space. The size field in the instruction contains the size of data for the routine being entered, including stack mark, parameters, and locals. The mp or mark pointer is offset by size, the area between the sp or stack pointer and the new allocation is cleared to zero and the sp set after that. The bottom of the stack field in the stack mark record is set from that, which allows the ipj instruction to find the proper setting of the stack for interprocedural jumps.

Checks if the resulting locals allocation would overrun the heap, and thus require more memory that is available. A runtime error results if not.

The assembly code represents the size as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

equa 17 address address boolean

equb 139 boolean Boolean boolean

equc 141 character character boolean

equi 137 integer integer boolean

equm size 142 q32 address address boolean

equr 138 real real boolean

equs 140 set set boolean

Find equal. The top of stack and second on stack values are compared, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

fjp addr 24 q32 boolean

Jump false. Expects a Boolean value atop the stack. If the value is false, or zero, execution continues at the instruction constant addr. Removes the boolean from stack.

Instruction Opcode Stack in Stack out

flo 34 integer real real real

flt 33 integer real

Convert stack integer to floating point. flt converts the top of stack to floating point from integer. flo converts the second on stack to floating point, but it assumes that the top of stack is real as well.

Instruction Opcode Stack in Stack out

geqb 151 boolean boolean boolean

geqc 153 character character boolean

geqi 149 integer integer boolean

geqm size 154 q32 address address boolean

geqr 150 real real boolean

geqs 152 set set boolean

Find greater than or equal. The top of stack and second on stack values are compared for second on stack greater than or equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

grtb 157 boolean boolean boolean

grtc 159 character character boolean

grti 155 integer integer boolean

grtm size 160 q32 address address boolean

grtr 156 real real boolean

grts 158 set set boolean

Find greater than. The top of stack and second on stack values are compared for second on stack greater than the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

inca cnt 90 q32 address address

incb cnt 93 q32 boolean boolean

incc cnt 94 q32 character character

inci cnt 10 q32 integer integer

Increment top of stack. The unsigned constant cnt is added to the value on stack top. Note that type a is not subject to arithmetic checks, but the other types are.

Instruction Opcode Stack in Stack out

inda off 85 q32 address address

indb off 88 q32 address boolean

indc off 89 q32 address character

indi off 9 q32 address integer

indr off 86 q32 address real

inds off 87 q32 address set

Load indirect to stack. Expects the address of an operand in memory on the stack top. The constant off is added to the address, then the operand fetched from memory and that replaces the address on stack top.

Instruction Opcode Stack in Stack out

inn 48 value set boolean

Set inclusion. Expects a set on stack top, and the value of a set element below that. Tests for inclusion of the value in the set, then replaces both of them with the Boolean result of that test.

Instruction Opcode Stack in Stack out

int 46 set set set

Set intersection. Finds the intersection of the two sets on the stack and replaces them both with the resulting set.

Instruction Opcode Stack in Stack out

inv 189 address2

Invalidate address. Expects an address on stack, then flags that location as undefined if undefined checking is enabled. This instruction is used to return variables to the undefined state. Removes the address from stack.

Instruction Opcode Stack in Stack out

ior 44 boolean boolean boolean

Boolean inclusive ‘or ‘. Expects to find two Boolean values on stack. Replaces them with the inclusive ‘or’ of the values.

Instruction Opcode Stack in Stack out

ipj p l addr 112 p8 q32

Interprocedure jump. The instruction contains the relative frame number p, and the address to jump to within the target procedure addr. The frames above the target procedure frame are discarded, and the target frame is loaded. Execution then proceeds at the given address addr.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

ivt off size 192 q32 q32 tagaddr newtagval tagaddr newtagval

Invalidate tagged variant. Expects a new setting for a tagfield at stack top, and the address of the tagfield below that. If the tagfield was defined, and the new tag value is different than the old tag value, the variant area controlled by the tagfield is set as undefined. The off instruction field constains the offset from the tagfield to the base of the variant. The size instruction field contains the size of the variant. The offset is applied to the address of the tagfield, and the entire variant is set undefined. Note that the size of the variant is the maximum size of all possible variants.

The stack is unchanged during this operation.

Instruction Opcode Stack in Stack out

ixa size 16 q32 address index address

Scale array access. Expects an index value at stack top, and the address of an array below that. The size instruction field contains the size of the base element of the array. The index is multiplied by the size, then added to the address of the base of the array, then that element address replaces both the index and the base address on stack.

Note that this operation does not remove the array index offset, ie., array [1..10] does not have 1 removed here.

Instruction Opcode Stack in Stack out

lao off 5 q32 address

Load address with offset. Loads the address in the stack area with offset off. This instruction is used to index globals. The address is placed on stack top.

Instruction Opcode Stack in Stack out

lca len ‘str’ 56 q32 address

Load string address. P5 accepts any length of string between quotes, but stores the string as the given length. The string is padded as required with spaces. Accepts a “quote image” in the string.The string is placed in the constants area, and the instruction gets the address of that. Loads the string constant address off to the top of the stack.

Instruction Opcode Stack in Stack out

lda p addr 4 p8 q32 address

Load local address. Expects the relative frame number of the target procedure in p, and the offset address of the local in addr. The frame is found and the address calculated as an offset from the locals there and placed on the stack.

Instruction Opcode Stack in Stack out

ldcs ( set ) 7 q32 set

Load constant set. The address in the instruction gives the address of the set, which is loaded to stack top. This is used to load sets from the constant area. The set is specified as a series of values in the set between parenthsis. This set is loaded into the constant area on assembly, and the instruction contains the address of that.

Instruction Opcode Stack in Stack out

ldcb bool 126 q8 boolean

ldcc char 127 q8 character

ldci int 123 q32 integer

Load constant. Loads a constant from within the instruction according to the type, Boolean, character or integer.

Instruction Opcode Stack in Stack out

ldcn 125 nilcst

Load nil value. Loads the value of nil to the stack top.

Instruction Opcode Stack in Stack out

ldcr real 124 q32 real

Load constant real. The address in the instruction gives the address of the real, which is loaded to stack top. The real given is loaded into the constants area on assembly, and the instruction contains the address of that.

Instruction Opcode Stack in Stack out

ldoa off 65 q32 address

ldob off 68 q32 boolean

ldoc off 69 a32 character

ldoi off 1 q32 integer

ldor off 66 q32 real

ldos off 67 q32 set

Load from offset. Loads a global value from the globals area. The off constant gives the offset of the variable in the stack area. The value is loaded to the top of stack according to type.

Instruction Opcode Stack in Stack out

leqb 163 boolean boolean boolean

leqc 165 character character boolean

leqi 161 integer integer boolean

leqm size 166 q32 address address boolean

leqr 162 real real boolean

leqs 164 set set boolean

Find less than or equal. The top of stack and second on stack values are compared for second on stack less than or equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

lesb 169 boolean boolean boolean

lesc 171 character character boolean

lesi 167 integer integer boolean

lesm size 172 q32 address address boolean

lesr 168 real real boolean

less 170 set set boolean

Find less than. The top of stack and second on stack values are compared for second on stack less than the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

lip p addr 120 p8 q32 mp pfaddr

load procedure function address. Expects the relative mark count p, and the address of an existing mark/function address pair in the instruction. Loads a mark/address pair for a procedure or function parameter onto the stack from a previously calculated pair in memory. Used to pass a procedure or function parameter that was passed to the current function to another procedure or function. See the cip instruction for further information.

Instruction Opcode Stack in Stack out

loda p off 105 Address

lodb p off 108 boolean

lodc p off 109 character

lodi p off 0 integer

lodr p off 106 real

lods p off 107 set

Load local value. Expects the display offset p and the offset address in the local procedure frame off in the instruction. The value is loaded according to type.

Instruction Opcode Stack in Stack out

lpa p l addr 114 p8 q32 mp pfaddr

Load procedure address. The current mark pointer is loaded onto the stack, followed by the target procedure or function address. This puts enough information on the stack to call it with the callers environment.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

mod 49 integer integer integer

Find modulo. Finds the modulo of the second on stack by the top of stack. The result replaces both operands. This is always an integer operator.

Instruction Opcode Stack in Stack out

mov size 55 destaddr srcaddr

Move bytes. The instruction contains the number of bytes to move. The top of stack contains the source address, and the second on stack contains the destination address. The specified number of bytes are moved. Both addresses are removed from the stack.

Instruction Opcode Stack in Stack out

mpi 51 integer integer integer

mpr 52 real real real

Multiply. The first and second on stack are multiplied together, and the result replaces them both.

Instruction Opcode Stack in Stack out

mrkl line 174

Mark source line. This instruction exists only within the interpreter, and is not specified in the intermediate. The instruction parameter line contains the number of the source line that was read in at this point in the intermediate generation. The interpreter can use this to provide various source level debug services.

Instruction Opcode Stack in Stack out

mst p 11 p8

Mark stack. This instruction is used before parameters are loaded in expectation of a user procedure call. The p parameter in the instruction contains the level of the calling procedure minus the level of the called procedure and is used to establish the static links in the mark. A stack mark record is placed onto the stack an the sl or static link is placed, the dl or dynamic link, and the ep or extended stack pointer.

Instruction Opcode Stack in Stack out

neqa 18 address address boolean

neqb 145 boolean Boolean boolean

neqc 147 character character boolean

neqi 143 integer integer boolean

neqm size 148 address address boolean

neqr 144 real real boolean

neqs 146 set set boolean

Find not equal. The top of stack and second on stack values are compared for second on stack not equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

ngi 36 integer integer

ngr 37 real real

Negate. The operand atop the stack is negated according to type.

Instruction Opcode Stack in Stack out

not 42 boolean boolean

Logical ‘not’. The Boolean value atop the stack is inverted.

Instruction Opcode Stack in Stack out

odd 50 integer boolean

Find odd/even. The lowest bit of the integer on stack is masked to find the even/odd status of the integer as a Boolean value.

Instruction Opcode Stack in Stack out

ordb 134 boolean integer

ordc 136 character integer

ordi 59 integer integer

Find ordinal value of Boolean, character or integer. This is a no-op, because P5 always converts values to 32 bit integers on load to stack.

Instruction Opcode Stack in Stack out

pck sizep sizeu 63 q32 q32 uparr inx parr

Convert unpacked array to packed array. Because P5 does not support packing, this is effectively a copy operation. The instruction contains the size of the packed array sizep and the size of the unpacked array sizeu. The stack contains the address of the packed array at top, the starting index of the unpacked array under that, and the unpacked array address as third on stack. The number of elements in the packed array are moved from the unpacked array at the starting index to the packed array. All parameters are removed from the stack.

Instruction Opcode Stack in Stack out

reta 132 address

retb 131 boolean

retc 130 character

reti 128 integer

retp 14

retr 129 real

Return from procedure or function with result. Returns from the current procedure or function. The pc, sp, ep and mp pointers are restored from the saved data in the active stack mark record. For retp or return from procedure, this is all that is done. For the others, a return value is processed according to type. The value of the result is loaded from its place in the stack mark record, and expanded to 32 bits if Boolean or character.

Instruction Opcode Stack in Stack out

rgs 110 low high set

Build range set on stack. Expects a high value on stack top, and a low value below that. A set is constructed that has all of the members from the low value to the high value in it, then that set is placed as stack top.

The range builder instruction, along with sgs or build singleton set, is used to construct complex sets by creating sets of each individual ranges of values, then adding the resulting sets together on stack.

Instruction Opcode Stack in Stack out

rnd 62 real integer

Round real value to integer. Converts the real on top of the stack to integer by rounding.

Instruction Opcode Stack in Stack out

sbi 30 integer integer integer

sbr 31 real real real

Subtract. Subtracts the top of stack value from the second on stack value according to type.

Instruction Opcode Stack in Stack out

sgs 32 integer set

Construct singleton set. Expects a value on stack. Constructs a set with that value as the single member, then that set replaces the value on stack.

The singleton set instruction, along with rgs or range set builder, is used to construct complex sets by creating sets with individual ranges of values, then adding the resulting sets together on stack.

Instruction Opcode Stack in Stack out

sqi 38 integer integer

sqr 39 real real

Find square of value. Expects a value on stack top, and finds the square of that value according to type. This replaces the value on stack top.

Instruction Opcode Stack in Stack out

sroa off 75 address

srob off 78 boolean

sroc off 79 character

sroi off 3 integer

sror off 76 real

sros off 77 set

Store global value. The instruction contains the offset of a variable in the stack bottom, where the globals for the program exist. The variable is stored from the stack according to type, and the value removed from the stack.

Instruction Opcode Stack in Stack out

stoa 80 address address

stob 83 address boolean

stoc 84 address character

stoi 6 address integer

stor 81 address real

stos 82 address set

Store value to address. Expects a value according to instruction type atop the stack, and an address below that. The value is stored to the address, and both are removed from the stack.

Instruction Opcode Stack in Stack out

stp 58

Stop. Executing this instruction causes the interpreter to exit.

Instruction Opcode Stack in Stack out

stra p off 70 address

strb p off 73 boolean

strc p off 74 character

stri p off 2 integer

strr p off 71 real

strs p off 72 set

Store local. Instruction parameter p contains the relative count of the procedure frame to access. Instruction parameter off contains the offset address within the frame. The value at stack top is stored to the local variable according to type, and the value removed from the stack.

Instruction Opcode Stack in Stack out

swp size 118 value address address value

Swap first and second stack operands. The instruction contains the size of the second value on stack. The top value is assumed to be 32 bit. The top and second values are swapped. This instruction is used primarily to chain writes together to the same file access pointer.

Instruction Opcode Stack in Stack out

tjp l addr 119 boolean

Jump true. Expects a Boolean value on stack top. If the Boolean is true, the jump is taken to address addr. The Boolean is removed from the stack.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

trc 35 real integer

Truncate to integer. The real on stack top is converted to integer by truncating the fractional part. The integer replaces the real on stack top.

Instruction Opcode Stack in Stack out

ujc 61 q32

Output case error. This instruction is used in case statement tables to output case value errors. See xjp for the case table format. It is designed to be the same length as ujp, which is used to build the case table. The 32 bit address in the struction is a dummy, and contains zero. Always outputs a bad case select error. Inserted into the table where the value for the case would be invalid.

Instruction Opcode Stack in Stack out

ujp l addr 23

Unconditional jump. The instruction contained address addr is jumped to.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

uni 47 set set set

Find set union. Expects two sets on stack. The union of the sets is found, and that replaces them both.

Instruction Opcode Stack in Stack out

upk sizep sizeu 64 q32 q32 packarr upackarr sindex

Unpack a packed array. P5 does not implement packing, so this instruction is implemented as a copy operation. Expects the size of the packed array sizep and the size of the unpacked array sizeu in the instruction. The stack contains the starting index of the unpacked array at top, followed by the address of the unpacked array, and the address of the packed array as third on stack. The entire contents of the packed array are transferred into the unpacked array, which can be larger than the packed array.

Instruction Opcode Stack in Stack out

xjp l addr 25 q32 index

Table jump. Expects an address of a jump table within the instruction, and a jump table index on the stack. The jump table constists of a series of jump instructions using the ujp instruction. The index is multiplied by the length of the ujp instruction, which is 5 bytes long, and the pc set to that address. The effect is to jump via a table of entries from 0 to n. The index is removed from stack.

This instruction is used to implement the case statement.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

## System calls

The p5 interpreter partitions the execution work done in the stack machine into directly executed instructions and system calls. The system call instructions consist of I/O related calls and dynamic storage allocation. These are normally things that are handled by the runtime support system.

The difference between a function executed in an instruction and a function executed in a system call is minimal. A system call uses parameters stacked onto the interpreter operand stack in reverse just as the intermediate does.

The difference between a function executed in a user based procedure or function and a function executed in a system call is that the system call requires no framing or locals.

System calls are executed by number via the csp instruction.

The following system call descriptions show the intermediate menmomic of the call, the instruction number used as a parameter to the csp instruction, and the operands as they exist on the stack before the call. The stack operands are listed deepest first at left, to the topmost operand at the right. This is the logical call order of the routine parameters.

Each system call can be a procedure or a function. If it is a function, it leaves its result on the stack.

System Call Number Stack in Stack out

atn 19 real real

Find arctangent. Expects a real on stack. Finds the arctangent, and that replaces the stack top.

System Call Number Stack in Stack out

cos 15 real real

Find cosine. Expects a real on stack. Finds the cosine, and that replaces the stack top.

System Call Number Stack in Stack out

dsl 40 addr tagcst.. tagcnt size

Dispose of dynamic tagged record. This is a special form of dispose for a dynamically allocated record with tagged variants. The top of the stack contains the size of the record, after any fixed tag constants specified are taken into account. Below that, the number of tagfield constants that were specified exists. Below that, a list of all the constants that were used to specify the allocation. This list is from leftmost in sourcecode order and deepest in the stack to rightmost in source and topmost in stack. Following that, the address of the dynamic record.

The allocation of the record, done by the system call nwl, contains a matching list of tagfield constants allocated “behind” the allocated pointer, with the number of tag constants last in the list so that it can be found just below the pointer. These lists represent the tagfield list used to allocate the record, in the dynamic itself, and the tagfield list used to dispose of it, on stack. These lists are compared for both number and value, and a runtime error results if not equal. The total allocation, record plus taglist, is then disposed of.

System Call Number Stack in Stack out

dsp 26 addr size

Dispose of dynamic variable. The top of stack contains the variable size, and the address under that is the address of the dynamic record. The variable is deallocated. Both operands are removed from stack.

System Call Number Stack in Stack out

efb 42 fileaddr boolean

Find eof of binary file. Expects the address of a file variable on stack. The file is tested for eof() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

eln 7 fileaddr boolean

Test for eoln of text file. Expects the address of a file variable on stack. The file is tested for eoln() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

eof 41 fileaddr boolean

Find eof of text file. Expects the address of a file variable on stack. The file is tested for eof() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

exp 16 real real

Find exponential. Expects a real on stack. Finds the exponential, and that replaces the stack top.

System Call Number Stack in Stack out

fbv 43 fileaddr fileaddr

File buffer validate text. Expects the address of a file variable on stack. Ensures the file buffer variable is loaded. If the file is a read file, or the file is in read mode, the file buffer variable is loaded by reading the file. The file address pointer is left on stack. This call is used to insure the file buffer contains data from the file, if it exists, when the file buffer variable is referenced. It is part of the “lazy I/O” file scheme.

System Call Number Stack in Stack out

fvb 44 fileaddr size fileaddr

File buffer validate binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. Ensures the file buffer variable is loaded. If the file is a read file, or the file is in read mode, the file buffer variable is loaded by reading the file. The element length tells the call how many bytes to read. The file address pointer is left on stack, but the size is purged. This call is used to insure the file buffer contains data from the file, if it exists, when the file buffer variable is referenced. It is part of the “lazy I/O” file scheme.

System Call Number Stack in Stack out

gbf 35 fileaddr size

Get file buffer binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. If the file buffer variable is indicated as full, then it is marked empty, thus discarding the data there. If it is full, new data is read over the file buffer variable. Both the file address pointer and the size are discarded from stack. This operation is part of the “lazy I/O scheme”.

System Call Number Stack in Stack out

get 0 fileaddr

Get file buffer text. Expects the address of a file variable on stack. Reads the next element of the file into the file variable buffer. The file pointer is discarded.

System Call Number Stack in Stack out

log 17 real real

Find logarithm. Expects a real on stack. Finds the logarithm, and that replaces the stack top.

System Call Number Stack in Stack out

new 4 addr size

Allocate dynamic variable. Expects the size of variable in bytes to allocate on stack top, followed by the address of the pointer variable. Space with the given size is allocated, and the address placed into the pointer variable. Both operands are removed from stack.

System Call Number Stack in Stack out

nwl 39 addr tagcst.. tagcnt size

Dispose of dynamic tagged record. This is a special form of new() for a dynamically allocated record with tagged variants. The top of the stack contains the size of the record, after any fixed tag constants specified are taken into account. Below that, the number of tagfield constants that were specified exists. Below that, a list of all the constants that were used to specify the allocation. This list is from leftmost in sourcecode order and deepest in the stack to rightmost in source and topmost in stack. Following that, the address of the pointer variable.

The space required for the tagfield constant list and length is added to the total space required, and that space is allocated. The tagfield constant list, with the following length, is copied to the start of the allocated space, and the pointer variable is set just after that. Thus the tagfield constant list exists “behind” the pointer or in “negative space”, so that it can be found and referenced for various purposes. These are:

1. To check that the tagfield constant list is equal between new() and dispose() operations.
2. To check if a tagfield used to set the total space allocated for the variable is assigned with a different value than originally used in the new() call.

The size, tagfield list and length, and the address of the pointer variable are discarded.

System Call Number Stack in Stack out

pag 21 fileaddr

Page text file. Expects the address of a file variable on stack. A page request is sent to the file, and the file address is discarded.

System Call Number Stack in Stack out

pbf 36 fileaddr size

Put file buffer binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. If the file buffer variable is empty, a runtime error results. Otherwise, the contents of the file buffer variable are written out to the file. The size and file address are discarded.

System Call Number Stack in Stack out

put 1 fileaddr

Get file buffer text. Expects the address of a file variable on stack. If the file buffer is empty, a runtime error results. Otherwise, writes the contents of the file buffer variable to the file. The file address is discarded.

System Call Number Stack in Stack out

rbf 32 fileaddr varaddr len fileaddr

Read binary file. Expects a file variable address on stack, the address to place read data above that, and the length of the base file element on stack top. If the buffer for the file is has data, then the data is read from that, otherwise the data is read from the file. The number of bytes in the element length are read.

Purges the target variable address and the element length from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rcb 38 fileaddr vaddr min max fileaddr

Read character from text file with range check. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max at stack top. Reads a single character from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdc 13 fileaddr varaddr fileaddr

Read character from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single character from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdi 11 fileaddr varaddr fileaddr

Read integer from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single integer from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdr 12 fileaddr varaddr fileaddr

Read real from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single real from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rib 37 fileaddr vaddr min max fileaddr

Read an integer from text file with range check. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max at stack top. Reads a single integer from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rln 3 fileaddr fileaddr

Read next line from text file. Expects the file variable address on stack. The text file is read until an eoln() or eof() is encountered.

Leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rsb 33 fileaddr

Reset file binary. Expects the file variable address on stack. The binary file is reset(). The file variable address is purged.

System Call Number Stack in Stack out

rsf 22 fileaddr

Reset file text. Expects the file variable address on stack. The text file is reset(). The file variable address is purged.

System Call Number Stack in Stack out

rwb 34 fileaddr

Rewrite file binary. Expects the file variable address on stack. The binary file is rewritten. The file variable address is purged.

System Call Number Stack in Stack out

rwf 23 fileaddr

Rewrite file text. Expects the file variable address on stack. The text file is rewritten. The file variable address is purged.

System Call Number Stack in Stack out

sin 14 real real

Find sine. Expects a real on stack. Finds the sine, and that replaces the stack top.

System Call Number Stack in Stack out

sqt 18 real real

Find square root. Expects a real on stack. Finds the square root, and that replaces the stack top.

System Call Number Stack in Stack out

wbb 31 fileaddr boolean fileaddr

Write Boolean to binary file. Expects the file variable address on stack, and the Boolean to write to above that. Writes a single boolean to the file from the variable.

Purges the boolean from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbc 30 fileaddr char fileaddr

Write character to binary file. Expects the file variable address on stack, and the Boolean to write to above that. Writes a single character to the file from the variable.

Purges the character from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbf 27 fileaddr varaddr size fileaddr

Write binary variable to binary file. Expects the file variable address on stack, the address of the variable to write above that, and the size in bytes of the variable at stack top. Writes all the bytes of the variable to the file single boolean to the file from the variable.

Purges the size and the variable address from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbi 28 fileaddr integer fileaddr

Write integer to binary file. Expects the file variable address on stack, and an integer to write to above that. Writes a single integer to the file from the variable.

Purges the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbr 29 fileaddr real fileaddr

Write a real to binary file. Expects the file variable address on stack, and a real to write to above that. Writes a single real to the file from the variable.

Purges the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wln 5 fileaddr fileaddr

Write next line to text file. Expects the file variable address on stack. A new line is written to the text file.

Leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrb 24 fileaddr boolean width fileaddr

Write boolean to text file. Expects the file variable address on stack, a boolean to write to above that., and a field width at stack top. Writes the Boolean in the given width.

Purges the width and the boolean from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrc 10 fileaddr char width fileaddr

Write character to text file. Expects the file variable address on stack, a character to write to above that., and a field width at stack top. Writes the character in the given width.

Purges the width and the character from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrf 25 fileaddr real width frac fileaddr

Write real to text file in fixed point notation. Expects the file variable address on stack, a real to write to above that., a field width above that, and a fraction at stack top. Writes the real in the given width.

Purges the fraction, width and the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wri 8 fileaddr integer width fileaddr

Write integer to text file. Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrr 9 fileaddr real width fileaddr

Write real to text file in floating point notation. Expects the file variable address on stack, a real to write to above that., and a field width at stack top. Writes the real in the given width.

Purges the width and the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrs 6 fileaddr saddr width len fileaddr

Write string to text file. Expects the file variable address on stack, the address of the string to write to above that., the field width above that, and the length of the string at stack top. Writes the string in the given width.

Purges the length, width and the string address from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

# Testing P5

In the original implementation of Pascal and the Pascal-P porting kit, the implementation of tests on the system were largely undefined. Today, most programmers realize that any program exists as a collection of the code, documentation, and finally the tests for the program to prove it correct. If any one of these elements is missing, much as a three legged stool, the program will fall over. In reality, an “undocumented” program that is popular gets documented by its users or by third parties, such as independent book writers. Tests can be carried out ad-hoc, or essentially fall to the users (their happiness with this situation being quite another matter!).

Thus, as important to P5 as its code or documents are its tests.This was widely recognized with original Pascal, and an extensive series of tests were created with the advent of the ISO 7185 standard, as documented in “Pascal compiler validation” [Brian Wichmann & Z. J. Chechanowicz]. This was an excellent series of tests that showed close relationship to the standard. Unfortunately, like the “model compiler”, the rights to this test series, which was initially openly distributed, were closely held, and the project is essentially dead today.

Accordingly, I have created a new series of tests for P5, and now that is distributed with P5 itself, and continues to be improved. The PAT or Pascal Acceptance Test is completely automated. Further, because the original ISO 7185 validation tests were distributed in the “Pascal User’s Group” free of restrictions, I have used them to check on the completeness of the PAT. Thus, even if the original ISO 7185 validation tests cannot legally be distributed, the PAT is tracable back to these tests and is a reliable substitute for them.

## Running tests

### testprog

The main test script for testing is:

testprog <Pascal source file>

testprog.bat <Pascal source file>

testprog is a “one stop” test resource for most programs. It expects the following files to exist under the given primary filename:

program.pas The Pascal source file.

program.inp The input file for the running program.

program.cmp The reference file for the expected output.

testprog compiles and runs the target program, and checks its output against the reference file. Several files are produced during the process:

program.p5 Contains the intermediate code for the program as compiled by pcom.

Program.err Contains the output from pcom. This includes the status line indicating the number of errors. It also typically contains a listing of the program as compiled, which contains line numbers and other information. Note that if the pcom compile run produces errors, testprog will stop.

Program.lst Contains the output from the program when run. This is all output to the standard “ output” file.

Program.out This is all of the output to the “prr” special file. This is only used by P5 for special purposes, such as to output the intermediate code, and most programs will be empty.

Program.dif This is the output of the diff command between program.lst and program.cmp. It should be empy if the program produced the output expected.

Not all of the files for testprog need to have contents. For example, a program that does not do input does not need to have a program.inp file. An example of this would be the “hello” program. However, testprog expects all of the files to exist.

To determine if the test ran correctly, the output of first the program.err file should be checked for zero errors, then the resulting program.dif file is checked for zero length. All of these results are announced during the test:

C:\projects\PASCAL\p5\_ext>testprog sample\_programs\roman

Compile and run sample\_programs\roman

Compile fails, examine the sample\_programs\roman.err file

For a program that has a compile error.

C:\projects\PASCAL\p5\_ext>testprog sample\_programs\roman

Compile and run sample\_programs\roman

03/18/2012 10:44 AM 76 roman.dif

For a program that does not match it’s expected output.

If a large series of test programs are compiled and run with testprog, it may be more efficient to examine the output files to determine the result, first the program.err file, then the program.dif file. The rejection test is a good example of such a large series of files.

### Other tests

Not all test programs work well with the testprog script. Examples are “pascals”, and the self compile. For these programs a special test script is provided.

### Regression test

All of the “positive” tests are wrapped up in a single script, “regress.bat”. This script runs all of the positive tests in order of successive complexity. Whenever a change is made to P5, the regression test is run to verify that the resulting compiler is still valid.

## Test types

The Pascal-P5 tests are divided into two major catagories, referred here as “positive” and “negative” tests. The positive tests are designed to test what the compiler should accept as a valid program. The negative tests are designed to test what the compiler should reject as invalid programs.

Thus, they are termed here as the “Pascal Acceptance” test, and the “Pascal rejection test”. The two test types are fundamentally different. For example, all of the acceptance tests can appear in a single program, since the entire program should compile and run as a valid program. However, the Pascal rejection test cannot practically be represented as a single program because it would (or should) consist of a repeated series of errors. The compiler could conceivably recover from the each error sufficiently to encounter and properly flag the next error, but this would not be reliable, and would vary from compiler to compiler. Further, such a test would test error recovery as well as simple error detection, which are two separate issues, and should be covered by two separate tests.

## The Pascal acceptance test

The Pascal acceptance test consists of program constructs that are correct for ISO 7185 Pascal. It should both compile and run under a ISO 7185 Pascal compatible compiler. The acceptance test is present in a single source file:

iso7185pat.pas

## The Pascal rejection test

The Pascal rejection test is arranged as a series of short tests because each test is designed to fail. Thus, ideally only one failure point at a time is executed.

The test script, runprt, is designed so that each result is checked for if an error occurred or not, with the lack of an error being a fail. However, there are at least two characteristics of a test result that go to quality, and thus imply that the result should be hand checked at least once:

1. The indicated error may or may not be appropriate for the error that was caused (for example, indicating a missing ':' when the problem was a missing ';').
2. The error may generate many collateral errors, or even refuse to compile the remainder of the program. This includes generating recycle balance errors.

The error signature of the compile can be compared to the previous runs, but a miscompare does not automatically fail the compiler, since error handling may have changed. The typical stragey is to run a new compiler version through the PRT test, and hand-examine only the error differences. If the changes in the compiler are minor, a compare go/no go will serve as a regression test.

Here are descriptions of each of the tests.

Class 1: Syntax graph visitation. These tests are for various points on the syntax graph, typically addition or elidation of a critical symbol. These tests are not meant to be exhaustive, since there are an infinite number of possible syntax constructions. Rather, it is designed to be representative.

The syntax visitation is according to the Pascal Users's manual and Report" 4th edition, Appendix D, Syntax diagrams. The tests run in reverse, starting with "program", and working backwards. Thus, the syntax checks are performed top to bottom.

Class 2: Semantic tests. These tests are for the meaning of the program. Examples include using an undefined variable, using a type in the incorrect context, executing a case statement where there is no case matching the selector, etc.

Note that the tests are not designed to be a complete set of failure modes for the compiler, nor is that possible, since the set of possible failure modes is infinite. Instead, this test is designed to be representative", or contain a series of known failure modes that test the quality of error checking and recovery in the compiler.

Note that the semantic errors are the same as those enumerated in the ISO 7185 standard in Appendix D, offset in number by the section 1700. Not all errors in this section have equivalent test cases. It is possible for an error defined by the standard to be without expression as a program. The description of the error is as was written in the ISO 7185 standard.

The ISO 7158 listed errors are a mix of compile time and runtime errors. On some errors, either mode is possible, on others, not. For these cases, we have tried to break the test program into two sections, one that could fail at compile time, and others that do not. These are labeled with an appended A, B, C, etc.

There can be other reasons to divide a test case into two programs.

With semantic errors, there is always the posibility that the code could be changed by an agressive compiler to eliminate or nulify the test case. There is no requirement in ISO 7185 that the compiler render into code a statement that clearly has no effect in the final program, just to achieve an error result. Its really not possible to defeat this kind of optimization entirely. Printing to output is one possibility, but even then there is nothing to prevent the compiler from assuming it knows what the output will be and changing it. This is perhaps a subject for further research.

### List of tests

#### Class 1: Syntatic errors

Program

iso7185prt0001 Missing semicolon after program statement

iso7185prt0002 missing "program" word-symbol

iso7185prt0003 missing program name word-symbol

iso7185prt0004 Moved

iso7185prt0005 Moved

iso7185prt0006 missing period

iso7185prt0007 Extra semicolon

iso7185prt0008 Missing header parameters

iso7185prt0009 Opening paren for program header only

iso7185prt0010 Closing paren for program header only

iso7185prt0011 Improperly terminated header list

iso7185prt0012 Consecutive semicolons

Block

iso7185prt0013 Missing number for label

iso7185prt0014 Missing semicolon after label

iso7185prt0015 Non-numeric label

iso7185prt0016 Unterminated label list

iso7185prt0017 Unstarted label list

iso7185prt0018 Missing constant

iso7185prt0019 Missing constant right side

iso7185prt0020 Missing "=" in const

iso7185prt0021 Incomplete second in const

iso7185prt0022 Missing ident in const

iso7185prt0023 Missing semicolon in const

iso7185prt0024 Reverse order between label and const

iso7185prt0025 Missing type

iso7185prt0026 Missing type right side

iso7185prt0027 Missing "=" in type

iso7185prt0028 Missing ident in type

iso7185prt0029 Incomplete second in type

iso7185prt0030 Missing semicolon in type

iso7185prt0031 Reverse order between const and type

iso7185prt0032 Missing var

iso7185prt0033 Missing var right side

iso7185prt0034 Missing var ident list prime

iso7185prt0035 Missing var ident list follow

iso7185prt0036 Missing ":" in var

iso7185prt0037 Missing ident list in var

iso7185prt0038 Incomplete second in var

iso7185prt0039 Missing semicolon in var

iso7185prt0040 Reverse order between type and var

iso7185prt0041 Missing "procedure" or "function"

iso7185prt0042 Missing ident

iso7185prt0043 Missing semicolon

iso7185prt0044 Consecutive semicolons start

iso7185prt0045 Missing block

iso7185prt0046 Missing final semicolon

iso7185prt0047 Misspelled directive

iso7185prt0048 Bad directive

iso7185prt0049 Misspelled procedure

iso7185prt0050 Misspelled function

iso7185prt0051 Bad procedure/function

iso7185prt0052 Missing ":" on return type for function

iso7185prt0053 Missing type id on return type for function

iso7185prt0054 Reverse order between var and procedure

iso7185prt0055 Reverse order between var and function

iso7185prt0056 missing begin word-symbol

iso7185prt0057 missing end word-symbol

Statement

iso7185prt0100 Missing label ident

iso7185prt0101 Missing label ":"

iso7185prt0102 Missing assignment left side

iso7185prt0103 Missing assignment ":="

iso7185prt0104 Missing procedure identifier

iso7185prt0105 Missing "begin" on statement block

iso7185prt0106 Misspelled "begin" on statement block

iso7185prt0107 Missing "end" on statement block

iso7185prt0108 Mispelled "end" on statement block

iso7185prt0109 Missing "if" on conditional

iso7185prt0110 Misspelled "if" on conditional

iso7185prt0111 Missing expression on conditional

iso7185prt0112 Missing "then" on conditional

iso7185prt0113 Misspelled "then" on conditional

iso7185prt0114 Misspelled "else" on conditional

iso7185prt0115 Missing "case" on case statement

iso7185prt0116 Misspelled "case" on case statement

iso7185prt0117 Missing expression on case statement

iso7185prt0118 Missing "of" on case statement

iso7185prt0119 Misspelled "of" on case statement

iso7185prt0120 Missing constant on case stament list

iso7185prt0121 Missing 2nd constant on case statement list

iso7185prt0122 Missing ":" before statement on case statement

iso7185prt0123 Missing ";" between statements on case statement

iso7185prt0124 Missing "end" on case statement

iso7185prt0125 Misspelled "end" on case statement

iso7185prt0126 Missing "while" on while statement

iso7185prt0127 Mispelled "while" on while statement

iso7185prt0128 Missing expression on while statement

iso7185prt0129 Missing "do" on while statement

iso7185prt0130 Missing "repeat" on repeat statement

iso7185prt0131 Misspelled "repeat" on repeat statement

iso7185prt0132 Missing "until" on repeat statement

iso7185prt0133 Misspelled "until" on repeat statement

iso7185prt0134 Missing expression on repeat statement

iso7185prt0135 Missing "for" on for statement

iso7185prt0136 Misspelled "for" on for statement

iso7185prt0137 Missing variable ident on for statement

iso7185prt0138 Misspelled variable ident on for statement

iso7185prt0139 Missing ":=" on for statement

iso7185prt0140 Missing start expression on for statement

iso7185prt0141 Missing "to"/"downto" on for statement

iso7185prt0142 Misspelled "to" on for statement

iso7185prt0143 Misspelled "downto" on for statement

iso7185prt0144 Missing end expression on for statement

iso7185prt0145 Missing "do" on for statement

iso7185prt0146 Misspelled "do" on for statement

iso7185prt0147 Missing "with" on with statement

iso7185prt0148 Misspelled "with" on with statement

iso7185prt0149 Missing first variable in with statement list

iso7185prt0150 Missing second variable in with statement list

iso7185prt0151 Missing "goto" in goto statement

iso7185prt0152 Misspelled "goto" in goto statement

iso7185prt0153 Missing unsigned integer in goto statement

iso7185prt0154 Missing 1st constant on case statement list

iso7185prt0155 Missing only variable in with statement list

iso7185prt0156 Missing ',' between case constants

iso7185prt0157 Missing ',' between variables in with statement

Field list

iso7185prt0200 Missing field ident

iso7185prt0201 Missing first field ident

iso7185prt0202 Missing second field ident

iso7185prt0203 Missing ':' between ident and type

iso7185prt0204 Missing type

iso7185prt0205 Missing ';' between successive fields

iso7185prt0206 Misspelled 'case' to variant

iso7185prt0207 Missing identifier for variant

iso7185prt0208 Missing type identifier with field identifier

iso7185prt0209 Missing type identifier without field identifier

iso7185prt0210 Missing 'of' on variant

iso7185prt0211 Misspelled 'of' on variant

iso7185prt0212 Missing case constant on variant

iso7185prt0213 Missing first constant on variant

iso7185prt0214 Missing second constant on variant

iso7185prt0215 Missing ':' on variant case

iso7185prt0216 Missing '(' on field list for variant

iso7185prt0217 Missing ')' on field list for variant

iso7185prt0218 Missing ';' between successive variant cases

iso7185prt0219 Attempt to define multiple variant sections

iso7185prt0220 Standard field specification in variant

iso7185prt0221 Missing ',' between first and second field idents

iso7185prt0222 Missing ',' between first and second field idents in variant

Procedure or Function Heading

iso7185prt0300 Missing 'procedure'

iso7185prt0301 Misspelled 'procedure'

iso7185prt0302 Missing procedure identifier

iso7185prt0303 Missing 'function'

iso7185prt0304 Misspelled 'function'

iso7185prt0305 Missing function identifier

iso7185prt0306 Missing type ident after ':' for function

Ordinal Type

iso7185prt0400 Missing '(' on enumeration

iso7185prt0401 Missing identifier on enumeration

iso7185prt0402 Missing 1st identifier on enumeration

iso7185prt0403 Missing 2nd identifier on enumeration

iso7185prt0404 Missing ')' on enumeration

iso7185prt0405 Missing 1st constant on subrange

iso7185prt0406 Missing '..' on subrange

iso7185prt0407 Missing 2nd constant on subrange

iso7185prt0408 Missing ',' between identifiers on enumeration

Type

iso7185prt0500 Missing type identifer after '^'

iso7185prt0501 Misspelled 'packed'

iso7185prt0502 Missing 'array'

iso7185prt0503 Missing '[' on array

iso7185prt0504 Missing index in array

iso7185prt0505 Missing first index in array

iso7185prt0506 Missing second index in array

iso7185prt0507 Missing ']' on array

iso7185prt0508 Missing index specification in array

iso7185prt0509 Missing 'of' on array

iso7185prt0510 Misspelled 'of' on array

iso7185prt0511 Missing type on array

iso7185prt0512 Missing 'file' or 'set' on file or set type

iso7185prt0513 Misspelled 'file' on file type

iso7185prt0514 Missing 'of' on file type

iso7185prt0515 Missing type on file type

iso7185prt0516 Misspelled 'set' on set type

iso7185prt0517 Missing 'of' on set type

iso7185prt0518 Missing type on set type

iso7185prt0519 Missing 'record' on field list

iso7185prt0520 Misspelled 'record' on field list

iso7185prt0521 Missing 'end' on field list

iso7185prt0522 Misspelled 'end' on field list

Formal Parameter List

iso7185prt0600 Missing parameter identifier

iso7185prt0601 Missing first parameter identifier

iso7185prt0602 Missing second parameter identifier

iso7185prt0603 Missing ',' between parameter identifiers

iso7185prt0604 Missing ':' on parameter specification

iso7185prt0605 Missing type identifier on parameter specification

iso7185prt0606 Missing parameter specification after 'var'

iso7185prt0607 Misspelled 'var'

iso7185prt0608 Missing ';' between parameter specifications

Expression

iso7185prt0700 Missing operator

iso7185prt0701 Missing first operand to '='

iso7185prt0702 Missing second operand to '='

iso7185prt0703 Missing first operand to '>'

iso7185prt0704 Missing second operand to '>'

iso7185prt0705 Missing first operand to '<'

iso7185prt0706 Missing second operand to '<'

iso7185prt0707 Missing first operand to '<>'

iso7185prt0708 Missing second operand to '<>'

iso7185prt0709 Missing first operand to '<='

iso7185prt0710 Missing second operand to '<='

iso7185prt0711 Missing first operand to '>='

iso7185prt0712 Missing second operand to '>='

iso7185prt0713 Missing second operand to 'in'

iso7185prt0714 Alternate '><'

iso7185prt0715 Alternate '=<'

iso7185prt0716 Alternate '=>'

iso7185prt0717 Missing first operand to 'in'

Actual Parameter List

iso7185prt0800 Empty list (parens only)

iso7185prt0801 Missing leading '(' in list

iso7185prt0802 Missing ',' in parameter list

iso7185prt0803 Missing first parameter in parameter list

iso7185prt0804 Missing second parameter in parameter list

iso7185prt0805 Missing ')' in list

Write Parameter List

iso7185prt0900 Empty list (parens only)

iso7185prt0901 Missing leading '(' in list

iso7185prt0902 Missing ',' in parameter list

iso7185prt0903 Missing first parameter in parameter list

iso7185prt0904 Missing second parameter in parameter list

iso7185prt0905 Field with missing value

iso7185prt0906 Fraction with missing value

iso7185prt0907 Field and fraction with missing field

iso7185prt0908 Missing ')' in list

Factor

iso7185prt1000 Missing leading '(' for subexpression

iso7185prt1001 Missing subexpression in '()'

iso7185prt1002 Misspelled 'not'

iso7185prt1003 'not' missing expression

iso7185prt1004 Missing '[' on set constant

iso7185prt1006 Missing first expression in range

iso7185prt1007 Missing second expression in range

iso7185prt1008 Missing '..' or ',' in set constant list

iso7185prt1009 Missing first expression in ',' delimited set constant list

iso7185prt1010 Missing second expression in ',' delimited set constant list

Term

iso7185prt1100 Missing first operand to '\*'

iso7185prt1101 Missing second operand to '\*'

iso7185prt1102 Missing first operand to '/'

iso7185prt1103 Missing second operand to '/'

iso7185prt1104 Missing first operand to 'div'

iso7185prt1105 Missing second operand to 'div'

iso7185prt1106 Missing first operand to 'mod'

iso7185prt1107 Missing second operand to 'mod'

iso7185prt1108 Missing first operand to 'and'

iso7185prt1109 Missing second operand to 'and'

Simple Expression

iso7185prt1200 '+' with missing term

iso7185prt1201 '-' with missing term

iso7185prt1203 Missing second operand to '+'

iso7185prt1205 Missing second operand to '-'

iso7185prt1206 Missing first operand to 'or'

iso7185prt1207 Missing second operand to 'or'

Unsigned Constant

iso7185prt1300 Misspelled 'nil'

Variable

iso7185prt1400 Missing variable or field identifier

iso7185prt1401 Missing '[' in index list

iso7185prt1402 Missing expression in index list

iso7185prt1403 Missing first expression in index list

iso7185prt1404 Missing second expression in index list

iso7185prt1405 Missing ',' in index list

iso7185prt1406 Missing ']' in index list

iso7185prt1407 Missing field identifier after '.'

Unsigned number

iso7185prt1500 Missing leading digit before '.'

iso7185prt1501 Missing digit after '.'

iso7185prt1502 Missing digit before and after '.'

iso7185prt1503 Mispelled 'e' in exponent

iso7185prt1504 Missing 'e' in exponent

iso7185prt1505 Missing digits in exponent

iso7185prt1506 Missing digits in exponent after '+'

iso7185prt1507 Missing digits in exponent after '-'

iso7185prt1508 Missing number before exponent

Character String

iso7185prt1600 String extends beyond eol (no end quote)

#### Class 2: Semantic errors

iso7185prt1701 For an indexed-variable closest-containing a single index-expression, it is an error if the value of the index-expression is not assignment-compatible with the index-type of the array-type.

iso7185prt1702 It is an error unless a variant is active for the entirety of each reference and access to each component of the variant.

iso7185prt1703 It is an error if the pointer-variable of an identified-variable denotes a nil-value.

iso7185prt1704 It is an error if the pointer-variable of an identified-variable is undefined.

iso7185prt1705 It is an error to remove from its pointer-type the identifying-value of an identified-variable when a reference to the identified-variable exists.

iso7185prt1706 It is an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

iso7185prt1707 It is an error if the value of each corresponding actual value parameter is not assignment compatible with the type possessed by the formal-parameter.

iso7185prt1708 For a value parameter, it is an error if the actual-parameter is an expression of a set-type whose value is not assignment-compatible with the type possessed by the formal-parameter.

iso7185prt1709 It is an error if the file mode is not Generation immediately prior to any use of put, write, writeln or page.

iso7185prt1710 It is an error if the file is undefined immediately prior to any use of put, write, writeln or page.

iso7185prt1711 It is an error if end-of-file is not true immediately prior to any use of put, write, writeln or page.

iso7185prt1712 It is an error if the buffer-variable is undefined immediately prior to any use of put.

iso7185prt1713 It is an error if the file is undefined immediately prior to any use of reset.

iso7185prt1714 It is an error if the file mode is not Inspection immediately prior to any use of get or read.

iso7185prt1715 It is an error if the file is undefined immediately prior to any use of get or read.

iso7185prt1716 It is an error if end-of-file is true immediately prior to any use of get or read.

iso7185prt1717 For read, it is an error if the value possessed by the buffer-variable is not assignmentcompatible with the variable-access.

iso7185prt1718 For write, it is an error if the value possessed by the expression is not assignment-compatible with the buffer-variable.

iso7185prt1719 For new(p,c l ,...,c n,), it is an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

iso7185prt1720 For dispose(p), it is an error if the identifying-value had been created using the form new(p,c l ,...,c n ).

iso7185prt1721 For dispose(p,k l ,...,k, ), it is an error unless the variable had been created using the form new(p,c l ,...,c,,,) and m is equal to n.

iso7185prt1722 For dispose(p,k l ,...,k, ), it is an error if the variants in the variable identified by the pointer value of p are different from those specified by the case-constants k l ,...,k,,,,.

iso7185prt1723 For dispose, it is an error if the parameter of a pointer-type has a nil-value.

iso7185prt1724 For dispose, it is an error if the parameter of a pointer-type is undefined.

iso7185prt1725 It is an error if a variable created using the second form of new is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

iso7185prt1726 For pack, it is an error if the parameter of ordinal-type is not assignment-compatible with the index-type of the unpacked array parameter.

iso7185prt1727 For pack, it is an error if any of the components of the unpacked array are both undefined and accessed.

iso7185prt1728 For pack, it is an error if the index-type of the unpacked array is exceeded.

iso7185prt1729 For unpack, it is an error if the parameter of ordinal-type is not assignment-compatible with the index-type of the unpacked array parameter.

iso7185prt1730 For unpack, it is an error if any of the components of the packed array are undefined.

iso7185prt1731 For unpack, it is an error if the index-type of the unpacked array is exceeded.

iso7185prt1732 Sqr(x) computes the square of x. It is an error if such a value does not exist.

iso7185prt1733 For ln(x), it is an error if x is not greater than zero.

iso7185prt1734 For sqrt(x), it is an error if x is negative.

iso7185prt1735 For trunc(x), the value of trunc(x) is such that if x is positive or zero then 0 < x-trunc(x) < 1; otherwise 1 <x-trunc(x) < 0. It is an error if such a value does not exist.

iso7185prt1736 For round(x), if x is positive or zero then round(x) is equivalent to trunc(x+0.5), otherwise round(x) is equivalent to trunc(x- 0.5). It is an error if such a value does not exist.

iso7185prt1737 For chr(x), the function returns a result of char-type that is the value whose ordinal number is equal to the value of the expression x if such a character value exists. It is an error if such a character value does not exist.

iso7185prt1738 For succ(x), the function yields a value whose ordinal number is one greater than that of x, if such a value exists. It is an error if such a value does not exist.

iso7185prt1739 For pred(x), the function yields a value whose ordinal number is one less than that of x, if such a value exists. It is an error if such a value does not exist.

iso7185prt1740 When eof(f) is activated, it is an error if f is undefined.

iso7185prt1741 When eoln(f) is activated, it is an error if f is undefined.

iso7185prt1742 When eoln(f) is activated, it is an error if eof(f) is true.

iso7185prt1743 An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error.

iso7185prt1744 A term of the form x/y is an error if y is zero.

iso7185prt1745 A term of the form i div j is an error if j is zero.

iso7185prt1746 A term of the form i mod j is an error if j is zero or negative.

iso7185prt1747 It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

iso7185prt1748 It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation.

iso7185prt1749 For an assignment-statement, it is an error if the expression is of an ordinal-type whose value is not assignment-compatible with the type possessed by the variable or function-identifier.

iso7185prt1750 For an assignment-statement, it is an error if the expression is of a set-type whose value is not assignment-compatible with the type possessed by the variable.

iso7185prt1751 For a case-statement, it is an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

iso7185prt1752 For a for-statement, it is an error if the value of the initial-value is not assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

iso7185prt1753 For a for-statement, it is an error if the value of the final-value is not assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

iso7185prt1754 On reading an integer from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-integer.

iso7185prt1755 On reading an integer from a textfile, it is an error if the value of the signed-integer read is not assignment-compatible with the type possessed by variable-access.

iso7185prt1756 On reading a number from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-number.

iso7185prt1757 It is an error if the buffer-variable is undefined immediately prior to any use of read.

iso7185prt1758 On writing to a textfile, the values of TotalWidth and FracDigits are greater than or equal to one ; it is an error if either value is less than one.

iso7185prt1759 The execution of any action, operation, or function, defined to operate on a variable, is an error if the variable is a program-parameter and, as a result of the binding of the program-parameter, the execution cannot be completed as defined.

### Running the PRT and interpreting the results

There is a single script that runs the prt:

$ runprt

The results of all of the tests are gathered and formatted into the file:

iso7185prt.lst

This file has several sections:

1. List of tests with no compile or runtime error.
2. List of differences between compiler output and “gold” standard outputs.
3. List of differences between runtime output and “gold” standard outputs.
4. Collected compiler listings and runtime output of all tests.

Each of these will be examined in turn.

#### List of tests with no compile or runtime error.

The purpose of the rejection test is to cause an error in the test program, either at compile time or runtime, and test its proper handling. By definition, if no error exists, the test has failed. This list gives the tests that need to be examined for why they didn’t detect an error.

#### List of differences between compiler output and “gold” standard outputs.

Shows the number of lines difference between the output test.err or compiler output listing, and test.ecp or “gold standard” compiler output listing, for each test program. A difference of 0 means that nothing has changed in the run.

The compiler output listing flagging an error and the run output flagging an error are mutually exclusive. If the compiler listing shows an error, the program will not be run. If if the compiler output shows an error, it is examined to see if it flagged an appropriate error for the fault, and that it didn’t generate excessive “cascade” or further errors caused by the compiler having difficulty recovering from the error.

Once the test.err file is judged satisfactory, it is copied to the test.ecp file as the “gold” standard. If the test run shows a difference between the current compile and the gold standard file, it does not necessarily mean that it is wrong. It simply means it needs to be reevaluated and perhaps copied as the new gold standard file.

#### List of differences between runtime output and “gold” standard outputs.

If the test file is successfully compiled, it is run and the output collected as test.lst. This is compared to the “gold” run output file test.cmp.

The run output file, if it exists, is checked to see if it indicates an appropriate error for the fault indicated in the test. If the run indicates no error, or an error that is not related to the fault, or simply crashes, that is a failure to properly handle the fault.

Once the test.lst file is judged satisfactory, it is copied to the test.cmp file as the “gold” standard. If the test run shows a difference between the current output and the gold standard file, it does not necessarily mean that it is wrong. It simply means it needs to be reevaluated and perhaps copied as the new gold standard file.

#### Collected compiler listings and runtime output of all tests.

The output of the compiler listing, as well as the output of the run, if that exists, for each test is concatenated and placed at the end of the listing file. This allows the complete output of each test to be examined, without having to look at the individual test.err or test.list file.

### Overall interpretation of PRT results

The rejection test is a “quality” test. There is no absolute right and wrong. The clearest indication of failure is failure to find any error. Past that are issues that indicate the quality of the error handling in the compiler:

1. If the error is indicative of what the failure was.
2. No or few further errors resulted after the failure.
3. The compiler was able to resyncronise with the source.
4. No data was corrupted as a result of the run (dynamic space imbalance, null pointer errors or similar).

You can say that failure to achieve the above marks a poor quality compiler, but not a failing test.

## Sample program tests

The sample program tests give a series of sample programs in Pascal. The idea of the sample programs tests is to prove out operation on common programs, and also to give a newly modified version of P5 a series if tests progressing from the most simple (“hello world”) to more complex tests. If the new version of the compiler has serious problems, it is better to find out with simple tests rather than have it fail on a more complex, and difficult to debug, test.

Hello Gives the standard “hello, world” minimum test.

Roman Prints a series of roman numerals. From Jensen and Wirth’s “User Manual and report”.

Match A number match game, this version from Conway, Gries and Zimmerman “A primer on Pascal”.

Startrek Plays text mode startrek. From the internet.

Basics Runs a subset Basic interpreter. It is tested by running a recoded version of “match” above.

Pascal-S Runs a subset of ISO 7185 Pascal. From Niklaus Wirth’s work at ETH. It is tested by running the “Roman” program above.

## Previous Pascal-P versions test

As part of the regression tests, Pascal-P5 runs the older versions of itself, namely Pascal-P2 and Pascal-P4. These are the only versions of the compiler available. See the section “introduction” on page 7, and also the historical material on Pascal-P on the Standard Pascal website.

The run of previous versions of Pascal-P perhaps constitutes the purest form of regression test. It not only insures that P5 is compatible with previous versions, but that it can actually compile and run all of the previous code. Of course, this is possible in main because these 1970’s versions were adapted to the ISO 7185 standard, but that, fortunately, was a small change.

### Compile and run Pascal-P2

P2 runs on P5 virtually unmodified. It needed only the declarations of the prd and prr files removed. This is required because they are predefined in P5. In a stand-alone use of P2, those files are real external files and have to be declared.

P2 runs as a test a modified version of roman.pas. It has to be modified because of:

1. P2 only supports upper case.
2. P2 does not support the output file as a default parameter to writeln.

These changes are documented in the P2 archive on <http://www.standardpascal.org>.

P2 does not have a full regression test of its own because I didn’t do the work of cutting down the ISO 7185 test to the subset that P2 implements, as I did for P4. This is perhaps a future project.

### Compile and run Pascal-P4

P4 has the same modification of the prd and prr files as P2, but otherwise runs unmodified. P4 runs as its target standardp.pas, which was a version of iso7185pat.pas that was stripped so as to fit the subset of ISO 7185 that P4 runs. Recall that P4 does not run any standard of Pascal at all, it is intentionally a subset.

Standard.pas was modified only in that P4 running under P5 cannot execute a for loop of the form:

for b := false to true do ...

The reason for this is non-trival. The way that P4 executes such a loop is to check that b lies in the range of boolean 0..1, and terminate if it lies outside that range. This actually makes it an illegal program if variable b is treated as boolean, since b will be incremented to 2 and thus be an out of range value.

This is actually a standard issue with Pascal in general, and the ISO 7185 standard took pains to show an equivalent form of the for loop that does not involve creating an out of range boolean. P5 also uses a form internally that fixes this, it has to in order to fully comply with the ISO 1785 standard.

So why does that work with a standalone P4 and not with a P4 running under P5? P4, as well as P5, treat the internal data store as typeless and use type escapes in the form of undiscriminated variants in order to differentiate the different data forms. This means that the result of assigning 2 to b (or incrementing it from 1) are system and compiler dependent. Thus, the direct code generating compilers tolerate this, and P5 simply does not.

## Self compile

One of the more difficult tests for P5 (and the most time consuming) is to have P5 compile and run itself. Actually the entire idea of Pascal-P was to compile and run itself in order to accomplish a bootstrap of the compiler. Pascal-P was never provided in a form able to directly compile itself, it needed a few modifications in order to do that. Also, self compiling a compiler that interprets its final code is different from a machine code generating compiler. Whereas it is conceptually simple to imagine the parser and intermediate code generator compling a version of itself, having the interpreter run another instance of itself on itself is like looking into a series of mirrors. The interpreter will be interpreting itself while that interpreter interprets another program, etc.

This can actually be carried out to any depth of self-interpretation, but because each interpretation level slows down the code by an order of magnitude, such nested self interpretation rapidly becomes impractical to complete. As it is, a single level of self-interpretation takes hours.

The reason to put work into a self compile of P5 is that it is a difficult test that goes a long way to prove out the stability and capacity of the compiler, and also because it proves out a very important function of Pascal-P, that of bootstrapping itself.

### pcom

I was able to get pcom.pas to self compile. This means to compile and run pcom.pas, then execute it in the simulator, pint. Then it is fed its own source, and compiles itself into intermediate code. Then this is compared to the same intermediate code for pcom as output by the regular compiler. Its a good self check, and in fact found a few bugs.

The Windows batch file to control a self compile and check is:

cpcoms.bat

What does it mean to self compile? For pcom, not much. Since it does not execute itself (pint does that), it is simply operating on the interpreter, and happens to be compiling a copy of itself.

#### Changes required

pcom won't directly compile itself the way it is written. The reason is that the "prr" file, the predefined file it uses to represent it's output file is only predefined to p5 itself. The rules for ISO 7185 are that each file that is defined in the header which is not predefined, such as "input" or "output", must be also declared in a var statement. This makes sense, because if it is not a predefined file, the compiler must know what type of file (or even non-file) that is being accessed externally to the program.

Because the requirements are different from a predefined special compiler file to a file that is simply external, the source code must be different for a P5 file vs. another compiler. A regular ISO 7185 Pascal compiler isn't going to have a predefined prr file.

The change is actually quite small, and marked in the source. You simply need to remove, or comment out, the following statements:

{ !!! remove this statement for self compile }

prr: text;                      { output code file }

and

{ !!! remove this statement for self compile }

rewrite(prr); { open output file }

In the batch file above, this modified file is represented as pcomm.pas, or "modified" pcom.pas.

All of the source code changes from pcom.pas to pcomm.pas are automated in cpints.bat.

### pint

Pint is more interesting to self compile, since it is running (being interpreted) on a copy of itself. Unlike the pcom self compile, pint can run a copy of itself running a copy of itself, etc., to any depth. Of course, each time the interpreter runs on itself, it slows down orders of magnitude, so it does not take many levels to make it virtually impossible to run to completion. Ran a copy of pint running on itself, then interpreting a copy of iso7185pat. The result of the iso7185pat is then compared to the "gold" standard file.

As with pcom, pint will not self compile without modification. It has the same issue with predefined header files. Also, pint cannot run on itself unless its storage requirements are reduced. For example, if the "store" array, the byte array that is used to contain the program, constants and variables, is 1 megabyte in length, the copy of pint that is hosted on pint must have a 1 megabyte store minus all of the overhead associated with pint itself.

The windows batch file required to self compile pint is:

cpints.bat

As a result, these are the changes required in pint:

{ !!! Need to use the small size memory to self compile, otherwise, by

  definition, pint cannot fit into its own memory. }

maxstr      = 2000000;  { maximum size of addressing for program/var }

{maxstr      = 200000;}  { maximum size of addressing for program/var }

and

{ !!! remove this next statement for self compile }

prd,prr     : text;(\*prd for read only, prr for write only \*)

and

{ !!! remove this next statement for self compile }

reset(prd);

and

{ !!! remove this next statement for self compile }

rewrite(prr);

All these changes were made in the file pintm.pas.

Pint also has to change the way it takes in input files. It cannot read the intermediate from the input file, because that is reserved for the program to be run. Instead, it reads the intermediate from the "prd" header file. The interpreted program can also use the same prd file. The solution is to "stack up" the intermediate files. The intermediate for pint itself appears first, followed by the file that is to run under that (iso7185pat). It works because the intermediate has a command that signals the end of the intermediate file, "q". The copy of pint that is reading the intermediate code for pint stops, then the interpreted copy of pint starts and reads in the other part of the file. This could, in fact, go to any depth.

All of the source code changes from pint.pas to pintm.pas are automated in cpints.bat.

Self compiled files and sizes

The resulting sizes of the self compiled files are:

pcomm.p5

Storage areas occupied

|  |  |  |
| --- | --- | --- |
| Contents | Range of storage | Net size |
| Program | 0-114657 | (114658) |
| Stack/Heap | 114658-1987994 | (1873337) |
| Constants | 1987995-2000000 | (  12005) |

pintm.p5

Storage areas occupied

|  |  |  |
| --- | --- | --- |
| Contents | Range of storage | Net size |
| Program | 0- 56194 | (56195) |
| Stack/Heap | 56195-1993985 | (1937791) |
| Constants | 1993986-2000000 | (6014) |

# pgen\_gcc\_i80386

pgen is a new package for P5. It gives the ability to generate code directly for target machines. It is designed for the following goals:

1. Use the existing P5 intermediate language with as few modifications as possible.
2. Able to convert to target different machines as easily as possible.

pgen versions have the following naming scheme:

pgen\_<C compiler>\_<machine>

The C compiler as the particular ISO/ANSI C compiler targeted, and the machine is the target machine.

At this writing, only GCC as a the compiler, and the 32 bit I80386 machine model are available as a target.

pgen generates an assembly language output file, compatible with the C compiler target. To result in a full compilation, it requires:

1. A Pascal to clib translator/support “thunk” package, written in ISO/ANSI C, and using the clib standard C support routines package.
2. A Pascal runtime support package, written in the target assembly language.

Together with the generated assembly language file, these packages result in a binary that can run on any ISO/ANS C compatible installation.

## Generated assembly file conventions

The C language support file for P5 can run on any ISO/ANSI C installation. However, assembly language conventions are far less standardized. Thus, pgen must produce assembly language output specific to the assembler that comes with the C compiler in use.

For GCC, GAS or “GNU Assembler” is used, specifically for the 32 bit I80386 assembler. It follows the AT&T assembly syntax, notable for the following conventions:

1. Sources are on the left, and destinations are on the right.
2. Each operator contains the size of the operation, byte, 16 bit word, 32 bit double word, and 64 bit quad word as an operator suffix (b, w, l or q).

This is not the complete list of characteristics. Please see the GAS user’s manual for further details.

The generated assembly file contains three distinct areas:

1. The generated code.
2. The global (uninitialized) space.
3. The constants area.

In Pascal, there is no global initialized, but writable space. Similarly, the constants area cannot be written. The global space is where all program block variables reside, and is allocated in the file by a single block definition. The constants area contains a series of constants defined in the target assembly language, and are prefaced by labels for their addresses.

In Pascal, the module order is for the program block to appear last in the file. It is prefaced by the label “\_pascal\_main”, and it is called by the C support module.

## Labeling conventions

All labels in the P5 binary system are prefaced with “\_pascal\_”. This prevents collision with the C language names in the source. Most labels used in the binarieshave no symbolic meaning and are numbered as in “\_pascal\_42”.

The exception to this rule are the direct support library calls. These have specific names.

The original symbols from the P5 source files are not carried in the P5 intermediate, and do not appear in the output.

Each input line is output to the assembly output file in the format:

// line: 42

This can be used to find what code was generated for a particular line in the source file.

## Calling conventions

pgen uses the GCC C calling convention. That is, the register based calling convention that GCC uses when one C function calls another. It does not use the system calling convention or any other convention in the target program. All calls to the operating system, or to alternate languages, are left to the support module.

There is no direct calling convention between Pascal and C. Although the GCC calling convention is used, there are several places where the parameters in C have no analog in Pascal. These include (but are not limited to):

* Void pointers
* Pointers that are interchangeable with arrays
* Pointers that can also be numbers, such as small integers that cannot be addresses
* Array “pointers” that can be null (0).
* Zero terminated strings.

However, all current P5 parameter data types ***can*** be described in C. Thus, all calls from Pascal to C functions must use a “thunk”, or short section of code that rearranges or filters the data to be correct for it’s C counterpart. The clib interface for the support library performs this for standard P5 support calls. This only leaves “external” calls from Pascal to C.

Please note that many Pascal implementations that “claim” to be able to directly call C from Pascal do this by incorporating many C language conventions into Pascal, and usually do that by breaking the type security of Pascal. P5 does not use this methodology. Instead, the same effect is done by creating a thunk, and that achieves the same end with full type security and little overhead.

To aid in the writing of thunks, P5 uses the rule that the length of arrays are always passed with an array as a prefix parameter. This means that the call:

type a = array [1..10] of integer;

procedure x(p: a); external;

Will be equivalent to ISO/ANSI C:

Void x(int al, int p[]);

With al being the lenth of the array p, which in this case will always be 10 by Pascal rules. In this way, the C code can always understand the length of the array passed without further Pascal aid.

## clib interface library and support module

The clib thunk layer and support library does the job of translating P5 I/O, memory management, and other support calls to the ISO/ANSI compatable clib support library that implements C support calls. The use of clib means that P5 can be hosted on any ISO/ANSI C compatible installation. It also means that this layer is automatically compatible with clib compatable systems such as Petit-ami.

The pascal\_support library also contains the main (or \_main) procedure that is the standard entry point for C. It contains a short entry and exit sequence, and then directly calls \_pascal\_main, which is the Pascal program block.

P5 generates a series of calls to the support library as follows:

void \_pascal\_get(file f);

void \_pascal\_put(file f);

void \_pascal\_readln(file f);

void \_pascal

…

The API for P5 support functions follows the IP Pascal support library API convention.

## Modifications required to pcom for pgen

Pgen uses the following layout for the target program:

Stack

...

...

Heap

Global variables

Constants

00000000: Program code

The heap grows upwards, and the stack grows downward. This is the opposite of the pint convention. For this reason pgen requires the “a+” or “alternate stack addressing” option be set in pcom when generating intermediates for pgen.

## Overview of pgen\_gcc\_i80386 functionality

Pgen starts by assembling and processing the intermediate much as pint does. The main difference is that it generates actual machine code, not pseudo-machine code, and outputs it to an assembler file instead of storing it to a code array. It outputs the file as text line instructions to an assembler instead of binary codes. The procedure assemble does this work in pgen, just as it does in pint. It reads the lines in the intermediate file, converts the codes to intermediate codes, but uses these to directly issue instructions to write the equivalent instructions to perform the function of the intermediate code to the output file.

The intermediate codes, as well as the basic order of processing used in the main procedure of pint, were maintained in pgen to make it easier to read and compare to pint functionality, as well as match the intermediate code documentation.

The key routine in assemble for generating instructions is wrtins. This routine takes a string and four parameters, two numerics parameters and two registers, and uses printf() style escape characters to format output code lines. For each call to wrtins, one line of output assembly code is generated. The parameter escapes are:

#0 or #1 Generates one of the numeric parameters, either the first or second one.

%r1 or %r2 Generates one of the register parameters, either the first or second one.

Note that the appearance of the escaped parameters is very much like the ones used in the GAS assembly language. This means that the strings used to form instructions look like their GAS equivalents.

And example of wrtins use is:

Wrtins(prr, ‘ mov #0($r1),%r2 ‘, 10, 0, ord(rgrax), ord(rgrax));

Would generate the output line:

mov #10(%eax),%eax

i.e., move the dword at offset 10 from eax to eax itself.

pgen is a register directed compiler, that is, it places all operands in registers and operates on them there. This extends even to real values, set values, structures and arrays. If the operand is too large to be placed in a dword register, then it is pushed onto the stack and a pointer created in a register that points to that. When the operand is removed from the stack, the stack is incremented over the object, and the register pointer discarded.

It does not use the floating point hardware stack feature of the I80386 floating point processor. Instead, all floating point values are pulled to the stack, and when it is time to operate on them, they are loaded into the FPU, and the result pushed back on the stack. This means that we don’t need to worry about fill spill handlers for the FPU, or preserving its registers.

### Processing expressions

All expressions, when found in the intermediate, are loaded into memory as expression trees. This is a compromise between fully graphing the intermediate and processing it as a linear sequence, as pint does. This method was used in the original PDP-11 C compiler and is common. It means that only single expression is stored in memory, the format of it is quite simple, and yet fairly good register optimization can be used. A tree traversal is quite simple.

The processing of trees separates the participating intermediates into loads, operators and stores. The loads are the leaves of the tree, the operators its branches, and the stores are the root of the tree.

Keep in mind also that there are a lot of optimizations made possible by expression tree formatting that are not yet implemented in pgen, such as common subexpression elimination, constant compression, etc.

### Register allocation

pgen uses a subset of SSA (Single Static Assignment) techniques I have termed here as “Virtual Register Allocation”. The idea of VRA is that a series of “Virtual Registers” from an unlimited set are assigned to each unique path in an expression that carries a value. These paths are then repeatedly evaluated to see if the same registers can be reused without conflict between multiple paths, and merged if there is no conflict. When and if the number of virtual registers is less than or equal to the number of physical registers, each VR is assigned a PR (Physical register), and the algoritim is complete.

Each operator is classed as a producer or a consumer, that is, it produces a value or it consumes one. A load would be an example of a producer, and a store is an example of a consumer. An add, for example, is both a consumer and a producer, since it consumes its operands, but also produces as result.

The VRs assigned contain the PRs for the I80386 as the lowest 6 registers. The register algoritim starts by assigning one of N unique registers (or virtual registers) to each producer, starting with the PRs, which would normally be consumed by the operator that uses it, above it in the tree. However, because I80386 as a binary operand based instruction set, that is, it specifies one operand as source, and the other as both source and destination, we assign one of the source registers as also being the destination. This can either be because the I80386 instruction set requires it, or done arbitrarily (as in the case of an add). This means that the same register number can travel across any number of expression nodes.

Once all nodes in the exression have been “painted” with register numbers, there are then N “domains” which establish the use of the virtual registers. At some point, we will map the virtual registers to real registers that have equivalence in the I80386 machine model, or eax, ebx, ecx, edx, esi and edi (6 registers in total). We actually define the first 6 virtual registers to be equivalent to the physical registers in order of 1=eax, 2=ebx, 3=ecx, 4=edx, 5=esi, and 6=edi. The goal will be to reduce the number of virtual registers to less than or equal to 6. At that time, we would be done, since the VRs are all then represented by PRs.

Because the I80386 machine requires the use of certain registers, we have to provide for fixed allocation for certain nodes. When we assign VRs to nodes, we allow each node to choose a specific PR. It assigns VRs to all of the other used registers that don’t have that requirement. We handle PR coflicts, for example, having a node that uses PR x have as one of its operands the same PR x by having two registers as the result, a result register and a transfer register. These are normally kept equal, but if the result of one node conflicts with a register used by the consumer node, it will be assigned a VR. When the tree is encoded, the fact that rr is not equal to rt is seen, and a move generated to push the result out of the way.

Note that even instruction sets that have no fixed physical allocations require fixed allocation processing. This is because registered parameters must be placed for calling convention purposes.

To reduce the number of VRs after allocation to equal or less than the number of PRs, we perform a (rather inefficient) search between all registers to determine if two VRs can be collapsed into one. We allow VRs to be merged into PRs, but not the converse, since that would break the fixed allocation of PRs done earlier. We kept track of the total Rs used. Then for each VR, we match all other Rs present and look for if they do or do not “intersect”, or have nodes that use both Rs. If they are found not to intersect, then we can merge them, which means finding all instances of the register to be merged and changing them to the target register number.

When we have performed all possible merges, we check if this has reduced the number of registers to less than or equal to PR. If so, we can stop there. All registers now bear PR numbers. This is actually the most common case. For an expression not to be able to so merge, it must be very complex, or use lots of fixed register allocations, or both. Nevertheless, we must provide a method to deal with it.

We remove these conflicts by the process of “pruning” the given expression trees. The top level node has its branches pruned off and made into freestanding trees one by one until the register allocation algorithim can process them properly. Each pruned branch is cleared of registers and reallocated, then an implicit store used to place the result of the partial tree in a local temp value. The place where it was pruned off gets a load of that same temp. If pruning the top node does not work, we recurs and try pruning the subtrees of it, and so forth, until we converge on an answer that works. However, this is a rare case as discussed above.

The register allocation algorithim discussed has lots of room for improvement, but is a good starting point, and is relatively easy to implement. It is also very good across multiple archiectures.