Preface

This book is about compiler construction. But rather than the usual theoretical study, it is a case study of an actual compiling system.

The Pascal-P compiler is possibly the most widely used Pascal compiler -- it has been the basis of many Pascal systems, including the well-known UCSD system, and there has even been a computer built specifically to run it. There have been many references to and articles about Pcode before, but never a full exposition of it.

Studying the principles of compiler construction can be difficult if the theory is not backed up with some concrete examples of it in use. Realising this, many authors present small sections of a compiler usually for a toy language. Although this can be helpful, it often still does not fully demonstrate the problems involved, such as the problems of type compatibility, parameter passing, and so on. So this book is an attempt to fill this gap by presenting an annotated text of a complete compiler.

Another use of this book is to support the study of programming methodology. It is well known that a good way of learning how to program well is to critically read other people's programs, especially as so much of programmers' time is spent maintaining and altering programs they did not write.

Also presented with this book is a one-pass assembler and interpreter, both of which could be used in the study of assemblers, and as an introduction to machine architecture.

About the Commentary

Each chapter describes a particular aspect of the P-code system, each section discussing a particular procedure or group of procedures, sometimes preceded by an explanation of the data-structures used. References to lines of the programs are enclosed in square brackets [ ].

Usually a section is followed by notes on suggestions for improvements, corrections, or just alternative ways of doing something for comparison. Obviously in a lot of cases whether one method is better than another is a matter of taste, and sometimes the notes may appear critical. However, writing a compiler is a difficult job, and once written it could probably be improved almost indefinitely. The fact that it is possible to understand the whole of this compiler in a relatively short time is witness to its good design and style, and it is this good style that allows improvements to be easily found.

The notes have been arranged where possible to be independent of the main commentary, so that on a first reading, they can be skipped, in order to gain an understanding of the whole compiler, before going back and concentrating on details for a deeper understanding.

Sometimes points are repeated in the notes, for instance when explaining a data-structure and then later when the data-structure is used. This is to facilitate studying sections independently.

The only section of the compiler not discussed is printtables [676-845], and its related routines. This is a procedure for the output of the compiler tables for testing purposes and therefore is in no way essential to the compiler or understanding it; it has been left as an exercise to the reader.

Terminology

Two points on terminology. To avoid repetitious phrases the word routine has been used to mean "procedure or function". To avoid confusion, a type like...
type colour = (red, green, blue);

is called an *enumeration*, while the phrase *scalar* is used to cover enumerations, subranges, integer, character, boolean, and real.

**The Listings**

The compiler and assembler/interpreter as commented on are as the originals, with the corrections published in *Pascal News* included. The only changes we have made are corrections to the indentation, to some comments, and to the layout of the lines. However, in order to compile on modern compilers, some changes are essential, and then a *slightly modified version* should be used.

Note that the upward arrow is printed as a carat ‘^’

**References**

Two essential documents that should be referred to in collaboration with this are *The Pascal User Manual and Report*, (Jensen, 1975), the two halves of which are referred to in this book as *The User Manual*, and *The Pascal Report*, and *Pascal-P Implementation Notes* (Nori, 1981), which is the official document distributed with the compiler.

**Acknowledgements**

We have to thank many people. Professor Wirth for his co-operative response to our enquiries; Mike Horwood for his friendly encouragement and enthusiasm; Mike Smith, who after all provoked the book, and so ungrudgingly put up with us using his computer; Derek Newson, for helping us with production; Christine Wieting, who for no other reason than her nice nature, helped with the typing in the face of deadlines; Mike Dickson, for uncomplainingly taking some of the weight off our shoulders while we wrote; and to our fourth year students for putting up with being taught it.

**Dedication**

To David Hitchin

To Carolyn, Rhiannon, William, and Timothy.

**Preface to Revised Edition**

The original tapes of the book's sources were unfortunately lost. Therefore in order to make a version available via the Web, the original book has been scanned in and OCR'd. Consequently, there may be a few conversion errors still: our apologies (please let us know if you find any). The original text has been corrected in a few places, and there has been a slight rearrangement of some of the material (principally procedure and function calls have been moved to their own chapter, and expression attributes have been moved to the Semantic Analysis chapter).

The sources of the compiler and interpreter are available at [http://www.cwi.nl/~steven/pascal/](http://www.cwi.nl/~steven/pascal/)

Steven Pemberton, February 2002

Steven.Pemberton@cwi.nl
Introduction

The compiler presented here is for a close variant of Pascal known as Pascal-P. Rather than producing code for any particular machine, it produces code, that has come to be called 'P-code', for a hypothetical stack-based computer that is in many ways ideal for Pascal compilation.

Also presented here is an assembler and interpreter for P-code defining the actions for this P-machine.

Both these programs are written in Pascal, which at first sight may seem a rather incestuous relationship, but it leaves several options open to the implementor. For instance:

1. Translate the compiler by hand into some other language that is available.
2. Find someone who already has a Pascal compiler for another machine, and compile the P-compiler with this to produce a running P-compiler. Then use this new compiler to translate the P-compiler (that is, itself) into P-code.

Armed with this P-code version of the compiler, the interpreter may then be translated by hand into another language and this used to interpret the compiler.

Alternatively, a translator from P-code to an available assembly language could be written. Either way this would be easier than translating the compiler by hand.

Overview of the Compiler

Schematically, the information flow in the compiler is like this:

Lexical analysis processes the input characters and recognises the symbols of the language; the syntax analyser takes these symbols and recognises the constituent parts of the program; with the knowledge of these constituent parts the semantic analyser can gather information about what the program means; with this information the code generator can then generate equivalent code.

The actual structure of the compiler is slightly different. Central to it are compiling procedures that do the syntax analysis, and call the lexical analyser, semantic analyser, and code generator as sub-modules. Pictorially:
The assembler and interpreter are two separate modules; the assembler produces the code for the interpreter, which then runs the code.

It is worth mentioning here, that while the compiler was designed to be machine independent, the interpreter was written to run on a CDC machine, and so reflects many aspects of the CDC architecture, such as the word-length.

**Specific and General Reading**

**History**

The P-code compiler developed as an offshoot of an effort to produce a compiler for a CDC 6000 computer. Papers describing this development are:


**Compiling**

General books on the theory of compiling are


Bornat, R. (1979), Understanding and Writing Compilers, Macmillan.


**Syntax Analysis**

Books on the specifics of syntax analysis are


**Compilers**

Books that present the code for a compiler (in all cases except the first, for a mini language) are

Aretz, F. E. J. K. et al. (1973), An Algol 60 Compiler in Algol 60, Mathematical Centre, Amsterdam.


Wirth, N. (1976), Algorithms + Data Structures = Programs, Prentice Hall, N. J.

The compilers in the last two bear a close similarity to the P-code compiler, though of course are much smaller.

**Intermediate Codes**

An interesting review of intermediate codes like P-code is


**P-Code**

The following all deal with experience with P-code

Berry, R. E., (1978), Experience with the Pascal-P Compiler, Software -- Practice and Experience, 8, 617-627.


**Other Reading**

Addyman, A. N., et al. (1979), A Draft Description of Pascal, Software -- Practice and Experience, 9,381-424.


Pemberton, S., (1980), Comments on an Error-recovery Scheme by Hartmann, Software -- Practice and Experience, 10, 231-240.


**Pascal Implementation** by Steven Pemberton and Martin Daniels

Copyright © 1982, 2002 Steven Pemberton and Martin Daniels, all rights reserved.

**Pascal Implementation** by Steven Pemberton and Martin Daniels

**Chapter 1: Input and Lexical Analysis**

**Lines:** [303-549]
Routines: *endofline, error, insymbol, nextch, options*

This group of routines is responsible for reading the input, producing a listing, reporting errors, and splitting the input stream into distinct 'symbols' to be passed on to the next stage of the compiler.

Pictorially:

```
Characters ────> Input and Lexical Analysis ────> Symbols
               └───> Listing and Errors
```

**Routine nextch, lines [358-371]**

All input in the compiler is done by this procedure. Its purpose is to deliver the next character of the input in variable `ch`, dealing with the listing, and the end of line and end of file events. The following non-local variables are affected:

- `ch` the current character to be dealt with
- `chcnt` the position of the character in the current line
- `eol` a buffer for the `eoln` test, so that all processing of the line can be completed before printing errors relating to the line
- `test` rather untidily used to deal with `eof`.

[359] If end of line was met before this character then call the procedure `endofline`.

[362] If the input is not exhausted then save the state of `eoln` in `eol`, read the next character, print it if required, and increment `chcnt`, giving the position of this character in this line.

[368] Otherwise, the input is exhausted, and so a warning is printed, and `test` is set to `false`.

**Notes**

1. The variable `eol` is used to delay processing of the end of line until all processing of the line itself has been completed, mainly to ensure that all error messages relating to the last item on the line appear with that line.

   When `eoln(input)` is `true`, the next `read(input,ch)` will read a space into `ch`, as required by Pascal. A space terminates all lexical items, except strings, where `eol` is tested explicitly [486]. Thus the end of line terminates all items.
2. Ideally the statement if list then writeln(output) should be the first statement of procedure endofline where it belongs.

3. The "eof encountered" message is split in two because the P4 compiler restricts strings to a maximum of 16 characters.

4. The assignment to test is part of the rather messy treatment of end of file. This will be discussed later when talking about procedure insymbol.

5. chcnt is reset to zero at each call of endofline.

Routine error, lines [338-347]

Whenever the compiler reports an error, this procedure is called with an integer parameter representing the error number. The procedure saves a maximum of 10 errors in a buffer errlist, each element consisting of the error number (ferrnr) and the position at which it occurred (chcnt). Errinx (error index) counts the number of errors reported about the current line; if this number ever exceeds 9, then error 255 -- further errors suppressed -- is saved, and no others are saved for this line. Procedure endofline prints these saved error messages.

Notes

1. Since after an error it is no further use to produce code for the program being compiled, the global variable procode which governs code production could usefully be set to false within this procedure. This would reduce compilation time for incorrect programs.

2. Even though the nmr field is declared as 1..400 there are two places where error is called with a value of more than 400 ([664, 1921]).

3. Error numbers are used more or less as defined in the Pascal User Manual (see the appendix). The major exceptions are 399 which is meant to mean 'implementation restriction', and so should really be 398, and 400, 500 and 501 meaning 'compiler error'.

Routine endofline, lines [303-336]

This procedure processes end of line: listing errors for the current line, incrementing the line count, printing the two numbers that precede each line of the listing, and resetting chcnt.

[306-28]

If errors reported then print them on the listing.

[313]

If more than one error refers to the same character position, the error numbers are separated by commas.

[316-19]

An upward arrow ^ is printed as close to the error position as possible, considering the errors already printed.

[321-23]

The minimum width required to print the error number is calculated.

[323]

Before each line there are printed the line number and either ic the instruction counter, that is, the number of instructions produced so far, or lc, the location counter, the number of locations of data-store reserved so far for the current routine. Which of these is printed is determined by dp which is set to true when compiling the declaration part of a routine, and false otherwise [3567, 3582].

Notes

1. If list is false, then only the error messages are printed with no indication of which line they are for; it would seem a good idea to print linecount along with the asterisks at line [307].
2. The calculation of \( f \), the width needed to print the error number [321-3], is only necessary for the freepos calculation since the width specification in a write statement states the minimum width required. Therefore

\[
\text{write(output, currnmr: 1)}
\]

would have the required effect, taking as much space as necessary to print currnmr.

3. The lexical analyser is one of the bottle-necks of any compiler, and nextch in particular is here, as it is called once for each character in the source program (more than 160,000 times when compiling itself!). One way to reduce this bottle-neck would be to skip the initial blanks on every input line here in procedure endofline. When compiling the compiler this would save around 60,000 calls to nextch.

**Routine options, lines [373-394]**

This routine deals with 'pragmatic comments' like the one on line [1]. The format of the options is supposed to be a number of single options separated by commas, each option consisting of a letter followed by a sign. No contained spaces are allowed, and the four options possible are:

- **t** controls printing of internal tables after each routine compiled.
- **l** controls listing of the compiled program.
- **d** controls production of extra code to check array bounds, dereferencing of nil, subranges, etc.
- **c** controls production of code.

The defaults for these are t-, l+, d+, c-, set at line [3800] in procedure initscalars.

**Notes**

1. Observe that ‘+’ turns any option on, any other character turns it off. You might prefer the following:

\[
\begin{align*}
\text{if } & \text{ch = ‘+’ then prcode:=-true} \\
\text{else if } & \text{ch = ‘-’ then prcode:-false}
\end{align*}
\]

2. Any divergence from the required format causes the rest of the comment to be ignored. Unfortunately, unrecognised options are treated differently to recognised ones. Consider how the following would be treated:

- (*$l+,g-,c+*)
- (*$l+,g,c+*)
- (*$l,,g,c+*)

You might also like to consider the effect of

- (*$l+,c+,d*)

and of

- (*$l+,c+,g*)

and indeed of

- (*$l+,c+,*)
and consider how to prevent the problems these cause.

Routine *insymbol*, lines [349-356, 396-549]

This procedure is the heart of the lexical analyser. Its purpose is to skip spaces and comments (interpreting pragmatic comments on the way) and to collect the basic symbols of the program for use by the next phase of the compiler.

The different sorts of symbols are defined by the type *symbol*, [80-86], and may be split into several classes:

- **identifier**
- constant: integer, real, character, and string
- operator:
  - not,
  - adding operators (+, --, or),
  - multiplying operators (*, /, div, mod, and),
  - relational operators (<, <=, =, <>, >=, >, in),
- punctuation like brackets, comma, etc.
- reserved words that are not operators
- and one symbol, *othersy*, representing any unrecognised symbol.

A symbol is represented by the following group of global variables:

- *sy*  
  The kind of symbol currently held
- *op*  
  identifies which operator when *sy* = relop, mulop, or addop
- *id, kk*  
  the characters of the last identifier met, and its length
- *val*  
  the value of the last constant met.

The routine works by getting the next character, deciding on the basis of this which sort of token it is going to be, and then collecting characters as long as they can be part of that sort of token.

The routine may be split into several distinct parts:

- [398-401]  
  get next non-space character
- [402-405]  
  deal with unrecognised characters
- [408-424]  
  identifiers and reserved words
- [425-480]  
  integers and reals
- [481-502]  
  characters and strings
- [503-547]  
  punctuation and operators, with [532-539] for comments

**Space and Illegal Character Skipping** [397-405]
Label 1 is used once at line [539] for skipping comments. When a comment has been skipped, a jump is made back to here, the beginning of insymbol.

This piece of code skips spaces and newlines until the next relevant character, or the end of file. Because of the tricky use of the variable test it is quite difficult to understand this short piece of code; remember that nextch may set test to false.

These 4 lines of code deal with unknown characters, reporting error 399, and setting sy to othersy, which is a symbol not recognised by the syntax analyser.

The array chartp is an array that defines the 'character type' of each character in the available character set.

You will find chartp declared [284] as

array [char] of chtp

where chtp is defined as [90-91]

(letter, number, special, illegal, chstrquo, chcolon, chperiod, chlt, chgt, chparen, chspace)

You will find chartp initialised in procedure chartypes, [3908-44]: here all elements are filled with illegal before filling in elements that are known, thus leaving illegal in unknown elements.

Notes

1. The P4 compiler attempts as far as possible to be portable between different machines, and therefore tries to assume as little as possible about the character set available. The character types beginning 'ch' (chstrquo, chcolon, etc.) are all later additions attempting to increase this character independence, since they require fewer explicit character literals.

Unfortunately, these additions have not been applied uniformly: for example, the character set might include a second 'space' character such as 'tab', so it would be better to skip spaces with

\[
\text{while } (\text{chartp}[ch] = \text{chspace}) \text{ and not eol do nextch}
\]

rather than the present

\[
\text{while } (ch=' ') \text{ and not eol do nextch}
\]

This goes for most other character literals in the lexical scanner.

2. As mentioned above, the character types beginning ch, chstrquo, chcolon, etc., were later additions. As a choice of names, they are not very good, because they are a different style to the other four names. Names more in the style of the others should have been chosen, such as quote, although it is recognised that these have to be chosen with care so that they do not clash with existing names.

3. The section of code to skip spaces and newlines, [398-401], is so complicated that it deserves a rewrite; furthermore, test is a very poor choice of identifier name, since it reveals little about its use (this may have been a later addition to fix a bug). If its name is changed to fileended, and the assignment fileended:=false is retained in nextch, verify that this section can be replaced by the following code:

\[
\text{while } (ch=' ') \text{ and not fileended do nextch.}
\]
Remember that when \texttt{eol} is \texttt{true}, \texttt{ch} = '\'.

4. The \texttt{if} statement, [402-6], is quite unnecessary. Far better to include \texttt{illegal} in the case statement that follows it:

\begin{verbatim}
case chartp(ch) of
  illegal: begin
    sy := othersy
    op := noop
    error(399);
    nextch
  end;
letter: ...
\end{verbatim}

5. Since the syntax analyser does not recognise the symbol \texttt{othersy}, such a symbol will always cause a syntax error. Consequently, illegal characters will generate two error messages, one from the lexical analyser, and one from the syntax analyser. It would really be better then not to generate an error here, but leave it to the syntax analyser. In any case, an illegal character is not really an implementation restriction; error 6 ('illegal symbol') would be more appropriate.

\section*{Symbol Collection [407-547]}

[407]

The kind of symbol that is going to be analysed depends on its initial character, so the character type is selected using a case:

\begin{itemize}
  \item letters: identifiers and reserved words
  \item digits: real and integer numbers
  \item quote: characters and strings
  \item colon: colon itself, and becomes (:=)
  \item period: period itself, and range symbol (..)
  \item less than: less than (<), less than or equals (<=), and not equals (<>).
  \item greater than: greater than (>), and greater than or equals (>=)
  \item left bracket: left bracket, and comments
  \item all others: punctuation, and single character operators.
\end{itemize}

\section*{Identifiers and Reserved Words [408-424]}

[409-15]

Collect up to 8 characters of the identifier in the array \texttt{id}, terminating when \texttt{ch} is a character that cannot continue an identifier.

[416-9]

Pad out the rest of \texttt{id} with spaces, \(k\) is the length of this identifier, \(kk\) the length of the previous identifier. If \(k\) is greater than \(kk\), then the rest of \texttt{id} already contains spaces, and \(kk\) can be set to \(k\). Otherwise, the remaining characters of \texttt{id} that are not spaces, (those from \(k\) to \(kk\)) must be made so.

For instance, if the last identifier was \texttt{maxint}, there is the following situation:

\begin{center}
\begin{tabular}{ccccccc}
  \texttt{id} & m & a & x & i & n & t \\
\end{tabular}
\end{center}

\begin{center}
  \begin{tikzpicture}
    \node at (0,0) {kk};
  \end{tikzpicture}
\end{center}
If the next identifier is *special*, it becomes:

![Diagram showing identifier special with pointers kk and k]

Whereas if the following identifier had been *char*, it would have been

![Diagram showing identifier char with pointers k and kk]

leaving the final *nt* of *maxint* to be made into spaces.

[420-2]

Determine if the identifier is a reserved word or not. The reserved words are stored, in ascending length order, in the array *rw* (reserved words) declared [285], initialised in procedure *reswords* [3824].

The array *frw* (first reserved word) is set up so that *frw[n]* points to the first reserved word of length *n* in *rw*, for *n* = 1 to 8.

Then the search for a particular identifier in the reserved words list may be restricted to the reserved words of the length of that identifier, i.e. *frw[k]* to *frw[k+1] - 1*.

If the reserved word is found, the arrays *rsy* (reserved symbol) and *rop* (reserved operator) are used to deliver the kind of symbol and the kind of operator that it is.

Pictorially:
If the word is not a reserved word, then it is just an identifier.

Notes

1. [411] The constant 8, denoting the maximum of characters used to distinguish identifiers, while being a standard for Pascal, could well be a constant, say `maxid`. There would be nothing to lose by this added bit of documentation, and it would facilitate changing it if wanted. In fact, there is a good argument for making all literal integers into constants (with the possible exception of 0 and 1).

2. The test, [414-5]

    \textbf{until} chartp(ch) \textbf{in} \{special, illegal, chstrquo, chcolon, chperiod, chlt, chgt, chparen, chspace\}

is shorter and more obvious if written as the equivalent

    \textbf{until not} (chartp(ch) \textbf{in} \{letter, number\}).

3. [416-9] It is doubtful whether the optimisations involved in this bit of code are really worth it. They certainly obscure its workings. The clearest alternative would be

    \textbf{for} \ i := k+1 \ \textbf{to} \ 8 \ \textbf{do} \ id[i] := ";";
although there would now be no use for the separate \( k \) and \( kk \), and the \( kk := k \) could be eliminated.

Less obvious, but retaining most of the optimisation would be

\[
\text{for } i := k + j \text{ to } kk \text{ do } id[i] := ' '; \\
kk := k
\]

but the simpler first alternative seems preferable, since it depends less on its surroundings. The optimisations applied here really seem to gain only a microscopic amount of efficiency, and should be avoided.

4. Note that \( kk \) is initialised to 8. Convince yourself that this prevents the need for \( id \) to be initialised.

5. Note that since the reserved word \texttt{procedure} has 9 letters, it is only recognised from its first 8.

6. \texttt{Forward} should not be a reserved word; \texttt{nil} should be.

7. The maximum search length for reserved words is 8 for three lettered words. If the search was done using the first letter rather than the length of the word, the maximum search would be reduced to three (for letters \( p, t \), and \( f \) -- not including forward).

Real and Integer Numbers [425-80]

The overview of this section is:

- Collect the integer part of the number
- Collect the fractional part, if any: the period and following digits.
- Collect the exponent, if any: the 'e', the optional sign, and the following digits.
- Construct the real constant.
- Construct the integer constant.

Before examining this section in detail, it is first necessary to study how constants are stored.

How constants are stored

The values of integer, real, character, and string literals produced by the lexical analyser are stored in global variable \texttt{val} of type \texttt{valu}.

\texttt{Valu} is declared, as a variant record:

\[
\texttt{valu = record case intval: boolean of } \\
\quad \text{true: (ival: integer) } \\
\quad \text{false: (valp: csp)} \\
\text{end};
\]

where field \texttt{ival} is for storing integer and character literals holding the integer value or the ordinal value of the character, and field \texttt{valp} for all other types of literal.

The comment 'intval never set nor tested' [105] reflects the fact that the compiler knows from context when it has an integer or character literal and when it does not.

The field \texttt{valp} is of type \texttt{csp}, (constant pointer), where \texttt{csp} is \texttt{constant} and \texttt{constant} is a further variant record with three variants, (\texttt{reel}, \texttt{strg}, and \texttt{pset}), one for reals, one for strings, and one for set literals. (Set
literals are not constructed by the lexical analyser, but by the syntax analyser, line [2830] onwards). Sets are
held as a set of (small) integers, strings as a short string (up to a maximum of \texttt{strglth} characters) with an
index giving its length, and reals as the string of characters making up the number. Reals are held in this way,
so that the compiler itself does not have to do any real arithmetic, which makes it easier to implement initially.

So integers and characters are held

\[
\begin{array}{c}
\text{intval} \\
\text{ival}
\end{array}
\]

reals, strings, sets

\[
\begin{array}{c}
\text{intval} \\
\text{valp}
\end{array} \rightarrow \text{constant}
\]

Now to return to the commentary.

[426-8]
Collect the individual digits of the integer part of the number in the array \texttt{digit} until \texttt{ch} is not a digit.
Note that a number, including fractional part and exponent, may only be a maximum of \texttt{digmax}
characters long.

[429]
If the integer part is followed by \texttt{'} or \texttt{'e} then the number may well be a real number. If it is followed
by \texttt{'e}, it certainly is, but the period may turn out to be part of a construct such as \texttt{1..10}.

[431]
So in case it is such a construct, the value of \texttt{i} is left untouched as the length of the integer part, and \texttt{k}
is used for the length of the real number.

[432-3]
If the character was a period, then it is stored, and the next character is obtained.

[434]
If this character is a period, then it is a \texttt{1..10} construct, and so \texttt{ch} is changed to \texttt{'.}' for future
processing, when \texttt{insymbol} is next called, and there is a jump to label \texttt{3} to deal with the integer part
of the number [466].

[435]
To get here there was only one period after the integer part, and so it is definitely a real number.
Pascal demands that at least one digit follow the period of a real number -- if not, error 201 is
reported.

[437-9]
The digits of the fractional part are gathered.

[441-53]
Whether or not there was a fractional part, if the number is followed by an \texttt{'e}, the \texttt{'e} is stored and the
exponent is collected.

[444-7]
The optional sign is saved.
A series of digits is expected and saved.

Here the real constant is saved.

\( Lvp \) is a \(^{\text{constant}}\); its \(^{\text{cclass}}\) is set to \(^{\text{reel}}\), and the digits of the number are transferred from \(^{\text{digit}}\) to \(^{\text{rval}}\), after initialising \(^{\text{rval}}\) to all spaces.

If the number of characters in the number was greater than \( \text{digmax} \) then error 203 is reported and the number 0.0 is stored.

The constant is saved in \(^{\text{val}}\).

**Integer Numbers** [466-79]

Label 3 comes from [434] in constructs like 1..10.

Complains if the number is too long and saves 0.

Calculates the value of the integer, making sure that it is not too big.

\( \textbf{Ordint} \) is an \(^{\text{array [char] of integer}}\), only used for the characters '0' to '9' holding the value of each of these digits, i.e. 0 to 9; initialised at lines [3940-3].

**Notes**

1. Type \(^{\text{valu}}\) is split into two (one record for integers and characters, one for all others) as a space saving optimisation. Ideally, it would be defined as

\[
\text{cstclass} = \langle \text{int, reel, pset, strg, chrctr} \rangle;
\]

\[
\text{value} = \text{record case cclass: cstclass of}
\]

\[
\begin{align*}
\text{int: (ival integer)}; \\
\text{reel: (rval packed array [1..strglgth] of char)}; \\
\text{pset: (pval set of setlow..sethigh)}; \\
\text{chrctr: (cval char)}; \\
\text{strg: (slgth:0..strglgth sval: packed array [1..strglgth] of char)}
\end{align*}
\]

end;

However, this would mean that every object of type \(^{\text{valu}}\) would need to be large enough to accommodate the largest of these variants (probably \(^{\text{strg}}\)). Obviously, the designers of the compiler anticipated more integer and character literals than other kinds and so treated them specially to save space. Thus reals, strings and sets need space for one extra pointer -- \(^{\text{valp}}\) -- but integers and characters need very much less space.

An additional space saving advantage is that a constant can be created using \(^{\text{new}}\) with the necessary size, for example, \(^{\text{new(lvp, reel)}}\) [454].

2. The treatment of the 1..10 case [434] is verging on the abominable. The jump from the \(^{\text{then}}\) part of an \(^{\text{if}}\) statement into its \(^{\text{else}}\) part is probably the worst bit of programming in the compiler. It is unlikely that Pascal even allows it, and many Pascal compilers refuse it. In this edition we have rewritten it without a goto, leaving the original code as a comment.

3. \( \text{digmax} = \text{strglgth} -1 \) (see line [3808]) so there is no problem copying \(^{\text{digit}}\) into \(^{\text{rval}}\) [458].

4. \(^{\text{rval}[1]}\) is left blank [458] to allow for an optional sign which may get filled by the syntax analyser, should this number be part of a constant declaration such as
\textbf{const} \( z = -3.14 \)

(several places in procedure \textit{constant}, starting line [864]).

5. The test if \( \text{ival} \leq mxint10 \) is meant to ensure overflow does not occur in the next line when \( \text{ival} \) is multiplied by ten. \( mxint10 \) is initialised to the value \( \text{maxint} \div 10 \).

However, the test is not correct: for example if \( \text{maxint} = 1000, \text{ival} = 100 \), and \( k = '1' \). It is necessary to ensure that the new value of \( \text{ival} \) will not cause overflow, i.e. that

\[
\text{ival} \times 10 + \text{ordin}[\text{digit}[k]] \leq \text{maxint}
\]

that is,

\[
\text{ival} \times 10 \leq \text{maxint} - \text{ordin}[\text{digit}[k]]
\]

that is,

\[
\text{ival} \leq (\text{maxint} - \text{ordin}[\text{digit}[k]]) \div 10.
\]

This is the test wanted, and \( mxint10 \) can be eliminated in the process.

6. Pascal guarantees that the characters '0' to '9' are all in one group in the character set. Therefore the declaration of \( \text{ordin} \) could better be declared as

\[
\text{ordin}: \text{array}[\text{char}] \text{ of integer}
\]

could better be declared as

\[
\text{ordin}: \text{array}[0..9] \text{ of } 0..9
\]

as this exactly reflects its use. However, because of Pascal's guarantee, the value of a digit character can be calculated using the expression

\[
\text{ord}(\text{ch}) - \text{ord}('0').
\]

Therefore, line [474] which reads

\[
\text{ival} := \text{ival} - 10 + \text{ordin}[\text{digit}[k]]
\]

may be changed to

\[
\text{ival} = \text{ival} \times 10 + (\text{ord}[\text{digit}[k]] - \text{ord}('0'))
\]

(the brackets are essential to avoid causing overflow in critical cases). With this change, \( \text{ordin} \) may be eliminated entirely.

7. Error 203 really means 'integer constant exceeds range'. However there is no complementary one for reals.

\textbf{String and Character Literals [481-502]}

[483-8]

Collect the characters of the strings. If the string extends over a line, error 202 results.

The outer loop deals with quotes within strings, which are represented as two single quotes, e.g. "": the inner loop terminates when a single quote is met, and this quote is saved. The outer loop then
continues if a further quote follows, but this quote is not saved. If a second quote does not follow, then the double loop terminates.

Lgth is decremented to remove the last saved quote.

Pascal does not have zero length strings.

A string of length one is a character literal, and the ord of this character is stored in val.ival.

The string, of length >= 2, is stored in sval and slgth (string value and string length) of lvp, which is then stored in val.valp. If the string is longer than the maximum allowed, (strglgth), the implementation restriction error 399 results.

Notes

1. That strings can only be a maximum of strglgth characters long is one of the more inconvenient restrictions of this compiler. One of the many possible solutions would be to store strings as a list of pieces, for instance:

```
type constant = record case cclass: cstclass
    of
    strg: (slth: integer;
           sval: piece)
end;
piece = record chars: array [1..piecemax] of char;
       nextpiece: ^piece
end;
```

2. As before with spaces, the termination tests would best be

```
(chart[ch] = chstrquo).
```

Punctuation and Operators [503-547]

This deals with colon and colon-equals (:=).

This deals with period (.) and range (..).

Less than (lt: <), less than or equal (le: <=), and not equals (ne: <>).

Greater than (gt: >) and greater or equal (ge: >=).

Deals with open bracket ‘(‘; if it is followed by ‘*‘, then it is the start of a comment. If this comment starts with ‘$‘, then it is a pragmatic comment, and so procedure options is called. Then the rest of the comment up to ‘*‘ is skipped. Remember that ‘*‘ may appear within the comment not followed by ‘)‘. If there was a comment, there is a jump to label 1, right at the beginning of insymbol [397], to start searching for a symbol again.

This deals with all single character symbols and operators not already dealt with: + - * / = , [ ] ^ ; .

Ssy yields the kind of symbol, sop the kind of operator if it is one, noop otherwise.

Ssy is initialised [3858-64], sop [3873-5].

Due to the loop [398-401], this will only be reached if ch happens to be a space when end of file is reached. Othersy is not recognised by the syntax analyser and so will cause a syntax error.
Notes

1. Since when dealing with integers directly followed by '...', \texttt{ch} gets set to ':' [434], '...' sometimes gets treated as a colon. Consider why then, '.' as '=' never gets accepted as a substitute for ':='.

   Does

   
   \texttt{type index=0:9}

   ever get accepted? How about for labels?

   1..a:=0

2. Label 1 is really just a loop

   \texttt{while commentskipped do}

   or

   \texttt{repeat... until symbolfound.}

   Consider the changes necessary.

   Another solution would be to replace the \texttt{goto 1} by a (recursive) call to \texttt{insymbol}.

3. Note in the initialisation of ssy [3858...] that the assignments for ( $ space . ' : < > are all unnecessary, since these single character symbols and operators are dealt with individually elsewhere. Similarly with sop [3873...] for < and >.

General Notes on the Lexical Analyser

1. Throughout the compiler there are several groups of variables that really form single units. Such groups are for example

   \texttt{ch + chcnt, a character and its position.}

   \texttt{errlist + errinx, the error list and its index}

   For documentation reasons, it would seem quite a good idea to physically group these together as records, for example:

   \begin{verbatim}
   var ch: record val: char;
   pos: integer
   end;

   error: record numbers: array[1..10] of
   record pos: integer;
   nmr: 1..400
   end;
   index: 0..10
   end;

   options: record list, debug, check, printtables: boolean end;
   \end{verbatim}

   and access them

   \begin{verbatim}
   if option.list then
   \end{verbatim}
while \( ch.val=' ' \) do ....

2. While about it, another field could be added to the \( ch \) record: \textit{class: chtp} and \( ch \) could always be updated with its type. Then the following could be written:

\begin{verbatim}
while (ch.class = chspace) and not fileended do nextch
\end{verbatim}

3. Throughout the lexical analyser, there seems to be a tacit assumption that \textit{eoln} will directly precede \textit{eof}. For example [484]

\begin{verbatim}
repeat nextch
until (eol) or (ch='''')
\end{verbatim}

Pascal does not require this to be true. One solution would be in \textit{nextch} to write

\begin{verbatim}
if eof(input) then begin
        writeln(output, '*** eof', 'encountered');
        ch.class := illegal;
        ch.val := ' ';
        fileended := true
    end else
\end{verbatim}

Then, most symbols would terminate with the illegal character class, and those that would not (strings and comments) could explicitly test:

\begin{verbatim}
repeat nextch; ...
until eol or fileended or (ch.class = chstrquo)
\end{verbatim}

Another solution would be to completely revise the treatment of file events. If, instead of the two variables \textit{eol} and \textit{fileended} there was a single variable:

\begin{verbatim}
var filestate: (normal, lineended, fileended)
\end{verbatim}

firstly this would more clearly show that \textit{eol} and \textit{fileended} will never occur simultaneously; but more importantly it simplifies a great many tests, for now may be written

\begin{verbatim}
while (filestate = normal) and (ch.class = digit) do ... 
\end{verbatim}

and

\begin{verbatim}
repeat nextch; ....
until (filestate <> normal) or (ch.class = chstrquo) 
\end{verbatim}

to include both \textit{eoln} and \textit{eof}.

4. Throughout the compiler, there are procedures that are really too long. Here, \textit{insymbol} provides a good example: its final end is some 300 lines away from its heading, and this makes for quite difficult reading.

It seems a good idea to split such large routines up. For example, \textit{insymbol} into procedures \textit{skipspaces}, \textit{inidentifier}, \textit{innumber}, \textit{instring}, etc.

Then the body of \textit{insymbol} could be:

\begin{verbatim}
skipspaces;
case ch.class of
letter: inidentifier;
number: innumber;
chstrquo: instring;
......
\end{verbatim}