NAME preccx –

PREttier Compiler Compiler 2.42

SYNOPSIS

preccx [*options*] < *file.y* > *file.c*

preccx [*options*] *file.y* > *file.c*

preccx [*options*] *file.y file.c*

DESCRIPTION

Preccx is a compiler compiler. It converts *preccx*-style context-grammar definition scripts (with a *.y* extension) into C code scripts (with a *.c* extension). The output code compiles under ANSI C compilers such as the GNU Software Foundation's *gcc*(1).

There is an easy-to-use hook for *lex*(1) tokenisers.

Preccx extends the UNIX *yacc*(1) utility by allowing:

[0] Contextual definitions. Each grammar definition may be parameterized with contexts. For example, some languages determine whether a declaration is local (and to what) or global in scope by relative indentation, and this can be encoded in *preccx* using the number of spaces indentation as a parameter, n:

@ decl(n) = <' '>*n expr <'\n'> decl(n+1)*

This definition is intended to mean that a "decl" indented by n spaces consists of n spaces, an expression, and a newline, optionally followed by one or several "decl"s indented still further.

[1] Infinite lookahead and backtracking in place of the *yacc* 1-token lookahead, This means that *preccx* parsers distinguish correctly between sentences of the form 'foo bah gum' and 'foo bah NAY' on a single pass. If you cannot imagine why one should want to decide between the two, think about 'if ... then ...' and 'if ... then ... else ... '.

[2] Arbitrarily complex expressions. This means that compound definitions like

explain {{this | that} {several | no} times}+

are legal within *preccx* definition scripts.

[3] *Preccx* has postfix operators '*' (zero or more times), '*n' (exactly n times), '+' (one or more times), and '!' (execute accumulated actions now) built in, along with the '[ ]' (optionally) outfix operator. For example, the following means 'exactly n spaces':

@ space(n) = <' '>*n

The other built-ins are

'?' (any token)

'^' (beginning of line)

'$' (end of line)

'|' (or, placed between alternate phrases of the grammar)

'{ }' (grouping brackets)

'< >' (around literals)

'> <' (to mean 'not a particular literal')

'( )' around the name of a **BOOLEAN** valued predicate on tokens, defined as an int 1 or 0 –valued C function elsewhere in the script, and

') (' (anti-brackets) round a C expression of BOOLEAN type, meaning a logical test condition.

']..[' anti-brackets hide an expression, causing it to be required but ignored.

‘]a[b’ means that input must satisfy both a and b, while ‘a]b[’ means that b is trailing context.

‘$!’ is a shorthand for matching end-of-line followed by execution of pending actions (it also causes the input buffer to start being overwritten). It is roughly equivalent to the conjunction ’! $’, but more efficient.

‘a b c’ (conjunction) is the term denoting an expression consisting of an ‘a expression’ followed by a ‘b expression’ followed by a ‘c expression’. An example of a *preccx* script follows in the section *USAGE*.

[4] Modular output. Parts of a script can be *preccx*’ed separately, compiled separately, and then linked together later, which makes maintenance and version control easy.

[5] Speed. *Preccx* is fast, typically taking two to five seconds to compile scripts of several hundred lines. And it builds fast parsers too.

[6] Higher order behaviour. ‘Macros’ may be defined in a script. For example,

```
@ optional(parser) = parser
@                  | {}
```

may be defined (this particular example is an equivalent for the built-in ‘[parser]’ construct). After the definition, the construct

```
@ ice_cream(flavour) = tub(flavour) optional(sauce)
```

may be used instead of the built-in:

```
@ ice_cream(flavour) = tub(flavour) [sauce]
```

[7] Separate syntax to distinguish synthesised attributes (without side-effects) from attached actions (with side-effects) in 2.40 and above. This is a break from yacc(1) style aimed at greater transparancy and robustness. For example, the following synthesises a total for a simple sum without using side-effecting actions or the run-time value stack:

```
@ sum   = summand\x <’+’> summand\y  {@ $x+$y @}
```

whereas the following uses *yacc*(1)-style references in an attached action:

```
@ sum   = summand <’+’> summand  {: total=$1+$3; :}
```

NOTE: From 2.41 onwards *yacc*(1)-style referencing is only supported with the ‘-old’ command line switch to preccx, and less efficient code is generated. Moreover, the scope of both kinds of dollar variables is now strictly left to right so that ‘$0’ can no longer be used to access a term to the left. In other words, the *yacc*(1) style of use is now restricted and discouraged.

[8] Built-in error handling capability (2.40 and above). The following code sets the handler ‘foo’ as the parser to use when the parse beyond the ‘!{foo}’ does not match:

```
@ typical = okstuff !{foo} morestuff
```

Malformed parse input will be matched against the parse ‘okstuff foo’, and well-formed input will be matched against ‘okstuff morestuff’. The definition in this instance is equivalent to ‘okstuff ! {morestuff | foo}’.

Preccx is intended to be both easy and convenient to use, but a compiler compiler cannot be understood in one minute. Have a look at the example ∗.y files in the *preccx* directory to get more of the feel. A more complex line in a grammar definition script than those above may look like:

```
@ expr = var { <’+’> | <’-’> } expr
@          | <’(’> expr <’)’>
```

The ‘@’ is an ‘attention mark’. Every line which does not begin with an ‘@’ is passed through to the output unchanged, so arbitrary C code can be embedded in a preccx script. Intended comments must therefore be surrounded by C comment marks, ‘/∗’ and ‘∗/’.

A default do-nothing tokeniser is provided in the preccx library and will be automatically linked in unless you specify a different yylex() routine to the C compiler. There is nothing to worry about here. If you do nothing yourself, you will get a working parser out of a *preccx* script immediately, but if you particularly want to put your own tokeniser on the input, then you do that by naming it 'yylex()' and making it return TOKENs when called. It should write VALUE attributes into 'yylval', just like *lex*(1). Place its object module or source code file ahead of the '-lcc' argument when you use the C compiler, and it will be linked in instead of the default (NB. yylex() *must* signal EOF to *preccx* by setting 'yytchar=EOF', which yylex() routines generated by *lex*(1) do not seem to get right).

The way to compile a C source code file 'foo.c' generated by *preccx* into an executable 'foo' is to use an incantation like:

gcc –Wall –ansi –o foo foo.c –L <preccx dir> –lcc

You can change the TOKEN type by #defining it as a C macro in the *.y script (you may want a wider range of TOKENs than the 256 possibilities afforded by the default 8-bit char, and '#define TOKEN short int' is sometimes useful). But it is important that the appropriate *preccx* library is used at link time. The default libcc.a library will assume TOKEN=char, but different versions of the library can be produced by recompiling with TOKEN set to the desired data type.

The parser generated from a *preccx* script will ordinarily signal valid input by absorbing it silently, and signal invalid input by rejecting it and spouting an error message. This is a standard style for compiler-compilers. To get the parser to do anything else, you must decorate the definition script with ACTIONs (see below for details).

The error handler may be redefined by #defining an ON_ERROR(x) macro. An x=0 value should give the code to execute on a partial but successful parse and x=1 should give the code to execute on an unsuccessful parse. x=-1 should give code to execute when *preccx* attempts to backtrack across a 'cut' ('!', see below). For example:

#define ON_ERROR(x) x?printf("ow!\n"):printf("ouch!\n")

The default error actions attempt to restart the parse on the next line of input, using the parser p designated by 'MAIN(p)' in the script.

You may likewise #define BEGIN and END for C code to be executed at either end of a parse attempt. This means that BEGIN will be re-executed if the parse resyncs after an error, and your code should take account of that (most likely by installing and using an invocation counter).

OPTIONS

Preccx can be run as a *stdin* to *stdout* filter, taking no options or arguments. It is better practice, however, to use the command line options:

preccx [options] infile outfile

because then there is no danger of *preccx* misidentifying the console or keyboard when you have redirected stdin and stdout.

The default sizes of various internal buffers can be changed by command line options (version 2.40 and above only), as follows:

-rNNNN The read buffer size in Kb. This determines the maximum char length of a single production in a script readable by *preccx*. Default 2Kb/ 2K chars.

-pNNNN The maximum size in Kb of the internal program (tables) built by *preccx* during the scan of a specification script. It correlates with the maximum number of symbols in a single production rule. Default 20Kb/4K instructions.

-vNNNN The maximum size in Kb of the attributed data built by *preccx* during the scan of the specification script. Default 16Kb/4K data items up to v2.41, 0Kb/0K in v2.42 and later (now handled by C and the data is compiled instead of dynamically interpreted).

-fNNNN The maximal size in Kb of the area used by *preccx* to store backtrack points when

scanning a script. It correlates to the maximal number of sequents in a production rule. Default 16Kb/1K breakpoints.

The sizes need only be changed if *preccx* fails to parse an input script returning an error message that indicates an overflow of one of these buffers.

The buffers are also used by utilities built by *preccx*, and their sizes in the utilities are set by the macros READBUFFERSIZE, MAXPROGRAMSIZE, STACKSIZE and CONTEXTSTACKSIZE respectively (see below and look in cc.h and ccx.h).

-old This flag (version 2.41 and above) supports the use of yacc(1) *style dollar variables in attached actions. The support is limited however: $0 and lower cannot be referenced and the variables should only be read, not written. Writing to $1 still works as a way to assign the attribute attached to an entire clause, but use the {@foo@} notation in preference.*

ENVIRONMENT

The following macros must be set in the user's grammar definition script, above the #include <cc.h> or <ccx.h> directive:

define TOKEN tokentype

(default char) This defines the space reserved for each incoming token in the parser which *preccx* builds. Note that a corresponding version of libcc.a must be linked in at compile time.

define VALUE valuetype

(default char*) This defines the space reserved for each value on the runtime stack manipulated by the runtime program which *preccx* attaches to the parser. There is no good reason for changing this to a type which is shorter than long int (or far *char), because the actual space used will be a union type which is at least as long as these.

In version 2.41 and above, this stack is by dfault absent, but the VALUE macro still has significance.

define PARAM parametertype

(default long) This defines the space reserved for grammar parameters on the C runtime call stack. It may be worthwhile changing this to int on systems where int is much shorter than long. On such systems, integer constants must be cast to PARAM before they can be used as grammar parameters, viz: foo((PARAM)0).

The following macros can be set if required:

define READBUFFERSIZE length

(default 2048) This defines the lookahead token buffer length. No more than <length> tokens can appear between cut marks ('!') in the script, as without cut indicators, preccx cannot know if the parser might later backtrack or not, and will not embed buffer reset instructions (in v2.41 and later versions, *preccx* will attempt to increase the buffer in READBUFFERSIZE increments when necessary, so it is not a hard limit).

define MAXPROGRAMSIZE length

(default 4096) This defines the maximum length of the internal program built by parsers in order to execute attached actions.

define STACKSIZE length

(default 0) This defines the size of a runtime stack formerly used to manipulate attached attributes in versions prior to 2.41 and it is now obsolete. The usage was approximately proportional to nesting depth in productions. The stack can be re-enabled by setting the STACKSIZE to some positive amount. The V(n) macro can then be used to access it.

It can be safely left as 0 in code generated by preccx 2.41 and above.

define CONTEXTSTACKSIZE length

(default 1024) This defines the number of breakpoints that can be held for backtracking. Usage is proportional to the number of sequents in productions between cuts.

define C_STACKSIZE length

(default 0x7FFF or 32K) This is the C call stack.

Now for the horrors of synthetic attributes. To get a parser generated by *preccx* to do anything significant, you need either to get it to synthesise a data structure, or get it to generate outputs. Whichever, you usually need to scatter *actions* and *attributes* through the script. There are two styles of script to get to know: (a) old *yacc*(1)-style scripts, in which attributes are referred to by number, and (b) new style scripts, in which synthesized attributes are referred to by name.

Actions are pieces of C code (terminated by a semi-colon) and placed between a pair of bracket-colons (‘{: ... :}’) in the grammar definition script. For example, this action uses old-style *yacc*(1) numerical references to build a numerical value which it stashes in a C global variable:

@ addexpr = expr <’+’> expr {: total=$1+$3; :}

In the new style of named reference, this would be rendered as follows:

@ addexpr = expr\x <’+’> expr\y {: total=$x+$y; :}

If the computed value is to be attached as an attribute for the parse, this can be rendered as follows:

A newly attached attribute can then be used as an inherited parameter in the rest of the parse:

```
@ sum(subtotal) = addexpr\x <’+’> sum(subtotal+$x)
@                       | ...
```

In contrast, the value of total generated in the action is not available to the parse because actions execute later than the parse time. The value is available to later actions, however. And it is available in the parse once the next cut mark ’!’ in the script has been passed.

In the action, ‘$1’ is the value attached to the leftmost term, and ‘$3’ is that attached to the rightmost term. The ‘$1’ may be replaced by the explicit ‘*(VALUE*)p_1’ within C macros (their contents are not directly accessible to preccx and this is what ‘$1’ expands to) in version 2.41 and above. In earlier versions than 2.41, ‘V(1)’ is the appropriate replacement to use.

‘Values’ attached to each term of a *preccx* expression are an appropriate way to think of what is going on. Note that the full *yacc*(1) style of script, with attribute assignments mixed into the action code via the ‘$$’ pseudo-variable, is only supported until v2.40 and no later. Moreover, the *yacc*-style numerical referencing via ‘$1’, ‘$2’ and so on, from v2.41 on requires the ‘-old’ command line switch to preccx. In previous versions of *preccx,* it was supported without restriction.

Earlier versions of preccx than version 2.41 used a runtime interpreter (like yacc(1)) and a dynamic stack to implement the synthesized attributes. Version 2.41 and above compiles the attributes instead. The difference makes for some slight incompatibilities with yacc(1): the $0 reference now makes no sense, for example. It used to refer to the attribute attached to the term just seen to the left and was available below on the dynamic stack. But in a compiled model, it is simply out of scope.

A BIT OF HISTORY

In version 1.5 to 2.40, *preccx* generated code to shift the frame of reference in a runtime attribute stack automatically. This was set by ‘call_mode=0’ (the default) in the BEGIN macro. In earlier versions, or if ‘call_mode=1’ was set, frame shifts had to be coded explicitly in the script: this would be accomplished by including a VV(n) call early in the action attached to each clause. For example, a three term clause would need a VV(3) call. After the call (in call_mode=1 mode) the $n values would be correctly aligned with the grammar expressions, and without it, they would not be. The value to be associated with the whole expression was written into $1. Writing VV(3)=$2 was shorthand for VV(3);$1=$2. After version 1.5 and with ‘call_mode=0’ set, the explicit VV(3) was not

required and the attribute build could be coded as $1=$2 alone. Or, for compatability with *yacc*(1), as $$=$2.

This was all exactly equivalent to the treatment in the Unix *yacc*(1) utility, and it allowed you to incorporate the same incomprehensible tricks of pulling values off the stack when they were notionally 'further to the left' than the scope of the current expression, using $0 or even lower references.

To recap, in versions 1.50 to 2.40 the user had to choose a 'call mode' which controlled the way the stack of attributes is handled at run time. Using the default call_mode=0 mode, stack frame shifts were automatic and it was not necessary to set VV(n) (shift value stack by n) commands in actions. If call_mode=1, then stack shifts were left to the user, and VV(n) instructions had to be added explicitly to actions. From version 2.41 up 'call mode' is entirely obsolete so you can forget it!

In earlier versions than 1.50, the only call mode was call_mode=1. The call mode in later versions was set by:

call_mode = 0 (automatic); or 1 (user-directed);

in the BEGIN macro, to be #defined before the #include <cc.h> or <ccx.h> in the script. In version 2.41 none of this is necessary as the attributes are handled in the C runtime call stack, which is looked after by C. You can #define STACKSIZE 0 (to remove the stack entirely, to save space), all this also before the #include <cc.h> or <ccx.h> directive.

History off.

ATTRIBUTES

In version 2.41 and above, the job of building synthetic attributes has been hived off into the parser proper. Synthetic attributes are any non-side-effecting expression, possibly involving the dollar variables which denote the values of attributes of other terms in a clause. They are written within {@ ... @} symbols. The last attribute in a clause becomes the attribute of a clause. For example:

```
@ tree = <'('> tree\x <')'> tree\y {@ mknode($x,$y) @}
@        |...
```

is sufficient to build a simple parse tree for bracketed input. Note however that the attribute should be non-side-effecting. It may be called several times in a parse. Since compound structures have to be built via side-effects in C, each call to mknode will have to check its arguments to see whether it has been called before, and to return the previously built structure if it has. It will have to do its own memoizing. On the other hand, rebuilding the structure several times becomes an allowable strategy when garbage collection takes place often enough to reclaim wasted structures. Either technique removes visible side-effects.

ACTIONS

Real side-effects that the parse is intended to invoke are coded in all versions of *preccx* as actions between {:...:} pairs, analogously to *yacc*(1). Side-effecting actions need a little explanation. Because *preccx* is an infinite look-ahead parser, it cannot execute actions at the same time as it reads input. It might have to later backtrack across its parse, and, whilst it might deconstruct data structures built up in the parse, it is certainly impossible, for example, to undo any writes to *stdout* which might have occurred.

So *preccx* builds a program as it parses. When the parse finishes correctly, the program is executed by an internal engine, but if the parse is unsuccessful or has to be backtracked, the program is 'unbuilt' before its actions are executed. This program is a linear sequence of C code actions which have been specified in the *preccx* definition file. Thus the specification:

```
@abc=a b c {:printf("D");:}
@a=<'a'> {:printf("A");:}
@b=<'b'> {:printf("B");:}
@c=<'c'> {:printf("C");:}
```

will, upon receiving input "abc", generate the program

```
printf("A");printf("B");printf("C");printf("D");
```

to be executed later. Thus actions attached to a sequence expression may be thought of as occurring immediately after the actions attached to sub-expressions, and so on down. That explanation should enable you to generate side-effects in the correct sequence.

As remarked above, in version 1.50 to 2.40 of *preccx*, attributes were built in the side-effecting actions, in yacc(1) style. In version 2.41 and above, attributes are attached using the new {@foo@} notation. This is certainly mechanically more robust, and it ought to be conceptually cleaner too. Attributes need the {@ @} signs and should not have side-effects. Actions need {: :} signs and should contain only side-effects, and cannot make attributes.

USAGE

Preccx grammar description files conventionally have the *.y* suffix, and should follow the following format:

```
# define TOKEN ... (default = char)
# define VALUE ... (default = char*)
# define BEGIN ... (default nothing)
# define END   ... (default nothing)
# define ON_ERROR(x) ... (defaults to standard)
# include "cc.h"  (or ccx.h)
 @ first  definition : attached action; :
 @ ...
 @ ...
MAIN(name of entry parser)
```

The cc.h header file may be used instead of ccx.h in scripts which consist only of unparameterized definitions and terms.

EXAMPLE

The following script defines a simple +/- calculator in the version 2.41 language, using parameters. For scripts that work with earlier versions of the language, see earlier versions of the manual. Some notes on differences appear afterward.

```
# define TOKEN char
# define VALUE int
# define BEGIN printf("\nready> ");
# include "ccx.h"
# include <ctype.h>
 @ digit = (isdigit)\x      {@ $x-'0' @}
 @ posint(t)= digit\x posint(10*t+$x)
 @          | digit\x                {@ 10*t+$x @}
 @ posint0= posint(0)
 @ anyint= <'-'> posint0\x      {@ -$x @}
 @         | posint(0)
 @ atom  = <'('> expr\x <')'> {@ $x @}
 @         | int
```

```
@ expr   = atom\x sign_sum\y   {@ $x+$y @}
@          | atom
@ sign_sum= <'-'> atom\x sign_sum\y
@                                  {@ -$x+$y @}
@            | <'-'> atom\x        {@ -$x @}
@            | <'+'> atom\x sign_sum\y
@                                  {@ $x+$y @}
@            | <'+'> atom\x        {@ $x @}
@ top        = expr\x              {: printf("=%d\n",$x); :}
MAIN(expr)
```

This script must be passed through *preccx*:

```
preccx < calculator.y > calculator.c
```

and then compiled, using the *preccx* kernel library in libcc.a:

```
gcc –Wall –ansi –o calculator calculator.c –L ... –lcc
```

The three dots stand for the directory in which the preccx library file libcc.a has been placed.

Note that '\x {@ $x @}' has no real effect, so it has been dropped from most of the points in the script where it might have been expected.

Here is the same script, but suitably coded for versions of *preccx* up to 2.40.

```
# define TOKEN char
# define VALUE int
# define BEGIN call_mode=0;printf("\nready> ");
# include "cc.h"
# include <ctype.h>
static int acc;
@ digit  = (isdigit)          {: $$=$1-'0';
@                                   acc=acc*10+$1;:}
@ posint= digit posint     {:$$=$2; :}
@          | digit           {: $$=$1;acc=0; :}
@ int    = <'-'> posint     {: $$=-$2; :}
@          | posint
@ atom   = <'('>expr<')'>  {: $$=$2; :}
@          | int
@ expr   = atom sign_sum   {: $$=$1+$2; :}
@          | atom
@ sign_sum= <'-'> atom sign_sum
@                                {: $$=-$1+$3; :}
@            | <'-'> atom     {: $$=-$2; :}
@            | <'+'> atom sign_sum
@                                {: $$=$1+$3; :}
```

```
@           | <'+'> atom     {: $$=$2; :}
@ top       = expr           {: printf("=%d\n",$1); :}
MAIN(top)
```

For an example of a parser which uses parameters essentially, the following definition of a parser which accepts only the fibonacci sequence as input may be useful:

```
# define TOKEN char
# define VALUE char*
# include "ccx.h"
# include <math.h>
# define INT(x) (int)(x)
# define DIV(m,n) INT(INT(m)/INT(n))
# define MOD(m,n) INT(INT(m)%INT(n))
# define LOG10(n) INT(log10((double)(n)))
# define DBLE(n)  (double)(n)
# define TEN      DBLE(10)
# define FIRSTDIGIT(n) \ (0!=n)?DIV((n),pow(TEN,DBLE(LOG10(n)))):0
# define LASTDIGITS(n) \ (0!=n)?MOD((n),pow(TEN,DBLE(LOG10(n)))):0
MAIN(fibber)
 @fibber    = { fibs $! }*
 @fibs      = fib((PARAM)1,(PARAM)1)
 @              {: printf("%d terms OK\n",(int)$k); :}
 @fib(a,b) = number(a) <','> fib(b,a+b) {@ $k+1 @}
 @           | <'.'> <'.'>      {@ 0 @}
 @             {: printf("Next terms are %d,%d,..\n",(int)a,(int)b); :}
 @number(n)= digit(n)
 @           | digit(FIRSTDIGIT(n)) number(LASTDIGITS(n))
 @digit(n) = <n+'0'>  /* rep. of 1 digit n */
```

The following are some example inputs and responses:

```
1,1,2,3,5,..                                  Next terms are 8,13,..
5 terms OK
1,1,2,3,5,8,13,21,34,51,85,..     error: failed parse: probable error at <>1,85,..
```

FILES

The following files are to be found in the */users/news/preccx* directory:

preccx	Preccx executable
preccx.y	Preccx definition in its own language
lex.y	Tokenizer for *preccx*
c.y	C parser for *preccx*
preccx.c	Preccx C source code (generated by *preccx* from *preccx*.y).
preccx.h	Preccx header file, needed only to construct *preccx*.
preamble.c	Auxiliary functions, needed only to construct *preccx*.
preamble.h	Header file for preamble.c, needed only to construct *preccx*.
common.c	Simple parsers common to both non-parameterised and parameterised parser kernels.

	Needed to make *common.o*P, included in libcc.a.
engine.c	Runtime engine. Needed to make *engine.o*P, included in libcc.a.
ccx.c	The source code of the *preccx* 1.0 kernel operations, needed to make *ccx.o*, included in libcc.a.
cc.c	The source code of the unparameterized *preccx* 1.0 kernel operations, needed to make *cc.o*, included in libcc.a.
ccx.h	The header file of the *preccx* parameterized kernel operations, needed by codes generated by *preccx*.
cc.h	The header file of the unparameterized *preccx* kernel operations, an alternative to ccx.h if you do not use parameterized definitions.
yystuff.c	Default lexer which allows you to escape newlines.
on_error.c	Default error routines.
atexit.c	In case atexit() is not present on your system.
libcc.a	The library containing cc.o, ccx.o and yystuff.o, needed to compile an executable from code built by *preccx*.
Makefile	The makefile for *preccx*.
test.y	Simple test script for *preccx*.
test.c	C output from the *test.y* script.
test	The test parser built by ‘gcc –ansi –o test test.c –L ... –lcc’.

SEE ALSO

yacc(1), lex(1), gcc(1L),

AUTHOR

Peter Breuer, Programming Research Group, Oxford University Computing Laboratory, UK.
Man page also hacked by Jonathan Bowen.

BUGS

1. On Sun3’s, the *gcc* compiler still complains that *printf* is being redefined. I don’t know why. If anyone finds the right compiler switch to magic this away, please tell me! For the hp300 series, the switch is –D__hp9000s300, if that’s any clue?

2. (Cured Mar 10 1992 in v1.1)

3. If you drastically change the type of VALUE in your script (make it larger than char*), you will also have to change the type declared in cc.h and then recompile the libcc.a library. This is not a bug but a feature.

4. (Cured Mar 17 1992 in v1.2).

5. It has been reported that the IBM ‘ANSI’ C compiler does not like the

```
typedef STATUS PARSER();
```

definition made by *preccx*. That is their problem.

7. (patch issued for *preccx* 2.30+ April 15 1993). Error in p_uniq0 code prevented recognition of all backtrack errors, with the effect that they were caught as failed parses some time later instead.

8. (patch issued for 2.40+ July 1994). *Preccx*’s C expresions don’t permit the use of ‘.’ as an operator. My omission. Use a macro instead until corrected (corrected).

Please report problems to *<Peter.Breuer@comlab.ox.ac.uk>*.

NOTES

In version 1.30 and above, script lines can be continued by placing an ‘@’ at the beginning of the next line, without a ‘\’ at the end of the previous line. Each sequence of ‘@’ continued lines must be terminated by an empty line.

Version 1.40 introduced a hook TOKEN *yybuffer for external lexical analysers. This is where lexers must eventually write their output for *preccx* to see it. Version 2.0 and above use a special routine mygets() to call yylex() which places the TOKEN returned by yylex() in the right place automatically.

For backwards compatibility, it is still possible to write into yybuffer directly, however. Note that, as mentioned already, EOF is tested by looking at the global int yytchar. The default yylex() lexer in libcc.a handles all this correctly.

Version 2.41 and above adds a special call get1token() which is used in mygets() to get one token from yylex(). You can use it to skip a token by calling it from an error handler instead of within mygets(). All calls to the lexer now go through get1token() and all interactions with the buffer go through get1token() and realignbuffer().

The default zer_error() handler supplied with preccx simply prints an error message and the unparsed portion of the string. That might well be *all* of the string, since *preccx* parsers try their darn'dest to make a match, then backtrack, so the (TOKEN *)maxp pointer is provided. This points to the deepest successful penetration into the incoming string, and is usually the point to look for the error. The pointer (TOKEN *)pstr shows the unparsed string, of which (TOKEN *)maxp will be an end-segment (the last TOKEN, in fact).

If you want to try and resync the parse at an error, a sensible thing to do would be to (rewrite zer_error to) skip a token at maxp, and rerun the parse. You will have to read the code of the run() function defined in **cc.c** to make sense of it, but you might try:

```
strcpy(maxp,maxp+1);tok=the_top_level_parser();
if(GOODSTATUS(tok)){
pc=0;pc=p_evaluate());
}
else printf("At least I tried!\n"));
```

Using a counter to set a maximal number of resync attempts in a single line would also be sensible!

You can obviate any bad_error() call by making sure that the top-level parser has a failsafe fallthrough to a ?* parser, with some kind of error action attached.

The version 2.x series *preccx* extended version 1.x by allowing parameters to each clause of the grammar (i.e., it treats inherited attribute grammars as well as synthetic ones), and by introducing the '!' (cut) marker. This can be inserted in expressions in order to stop backtracking through that point, which is useful in avoiding excessively long searches for alternate parses when no alternate is possible.

Promises: version 2.x will eventually eliminate the archaic *yacc*-style of stack manipulation with something much nicer (achieved in 2.4x series). Version 3.0 should implement tight type-checking. Contact the author for the most recent version.