NAME

bibparse − verify a bibclean or biblex lexical token stream, or BibTeX files

SYNOPSIS

bibparse [**−d**] <*infile*
or
bibparse [**−d**] *file1 file2 file3 . . .*

DESCRIPTION

Compilation of a computer language is traditionally divided into three steps:

- Lexical analysis is the grouping of consecutive characters into units, called *tokens*, that are meaningful in a particular language. **bibclean**(1) and **biblex**(1) are two programs that do this job for BIBTEX data.
- Parsing is the processing of the lexical analysis token sequence to verify that tokens appear in an order permitted by the language rules, called the *grammar*. **bibparse** does this for BIBTEX data.
- Semantic analysis, or code generation, is the interpretation of a grammar-conformant token stream to perform an intended task. For example, **bibtex**(1) transforms BIBTEX data according to rules in a user-specified style file into formatted bibliographic data suitable for a typesetting system.

  Although **bibtex**(1) includes internal implementations of lexical analysis and parsing, it does not make them available to the user.

bibparse takes a lexical token stream from **bibclean**(1) or from **biblex**(1), or BIBTEX files directly, and verifies their conformance to a proposed grammar for BIBTEX, published in the articles

> Nelson H. F. Beebe, *Bibliography prettyprinting and syntax checking*, TUGboat (ISSN 0896-3207) **14**(3) 222, October 1993, and TUGboat **14**(4) 395--419, December 1993.

The text of the latter is included with the **bibclean**(1) distribution.

The only output normally produced by **bibparse** is on the standard error unit, *stderr*, and then only if grammatical errors are detected. Silent execution means a successful parse.

The program exit code is zero on a successful parse, and non-zero otherwise.

For example, you can syntax check a bibliography collection by any of these three UNIX pipelines:

```
bibclean -no-prettyprint *.bib | bibparse
biblex *.bib | bibparse
bibparse *.bib
```

bibparse distinguishes between lexical token streams and BIBTEX files by examination of the *first* character of each input file: if it is a sharp sign, '#', then it is assumed to be the start of a line-number directive in a lexical token stream. Otherwise, it is assumed to be a BIBTEX file. **bibparse** then selects one of two internal lexical analyzers: a simple one that reads a lexical token stream from a file, or the complex one from **biblex**(1) linked into the **bibparse** executable.

OPTIONS

−d Write debug output to the standard output stream, *stdout*. This output is extremely verbose: it includes a record of each lexical token found, and how it is parsed according to the BIBTEX grammar.

If you are puzzled by an error message reported by **bibparse**, you are advised to extract the BIBTEX entry at, and possibly, immediately preceding, the line number in the diagnostic, then save that data in a temporary file and run **bibparse −d** on that small file, so as not to be overwhelmed by the output.

BIBTEX GRAMMAR

Here is a slightly-reformatted listing of the BIBTEX grammar, defined in detail in the articles cited above, and taken directly from the **bibparse** source code, which is transformed by a *parser generator* like UNIX **yacc**(1), or GNU **bison**(1), into a C-language program which can then be compiled by either C or C++ compilers, and then linked to produce the **bibparse** executable program.

The tokens, also called *terminals* in a grammar, that are recognized by **bibclean**(1) and **biblex**(1) are spelled in UPPERCASE letters.

Nonterminals, which are intermediate stages in the grammar processing, are spelled in `lowercase` letters. Each nonterminal referred to in the grammar eventually defines a grammar rule, which takes the form of a nonterminal, a colon, and one or more alternative expansions, separated by a vertical bar.

Interspersed in the rule expansions are braced *actions* which are to be invoked when the input token stream matches that rule. Here, they are simply calls to a function `RECOGNIZE()` which, when debug output is requested, prints its argument, followed by a newline, and then returns silently.

Internally, the parser does not deal with character strings at all: both terminals and nonterminals are simply small integer values that it manipulates on stacks using highly-efficient pattern matching to determine whether they match grammar rules.

The first three lines of the grammar below define the precedence of four tokens, so as to disambiguate cases where two rules would match the current token sequence.

The first rule, also called the *start symbol*, says that a `file` is either optional space, or an `object_list` optionally preceded and followed by space. Thus, an empty file, or one consisting only of space, is a valid BibTeX file.

The remaining rules are read similarly.

Most programming language grammars omit specification of rules for comments and spacing, assuming merely that they are permitted anywhere between tokens; this assumption simplifies the grammar significantly.

However, grammars for prettyprinters need to include rules for spacing because there may be circumstances where such spacing is significant for program layout and human readers. Space information is also required by unlexers, like **bibunlex**(1), which take a possibly-modified lexical token stream, and reconstruct a source program from it. Thus, this grammar includes precise rules for where spaces are permitted.

```
%nonassoc EQUALS
%left SPACE INLINE NEWLINE
%left SHARP

%%
file:             opt_space                        {RECOGNIZE("file-1");}
                | opt_space object_list opt_space  {RECOGNIZE("file-2");}
                ;

object_list:      object                           {RECOGNIZE("object-1");}
                | object_list opt_space object     {RECOGNIZE("object-2");}
                ;

object:           AT opt_space at_object           {RECOGNIZE("object");}
                ;

at_object:        comment                          {RECOGNIZE("comment");}
                | entry                            {RECOGNIZE("entry");}
                | include                          {RECOGNIZE("include");}
                | preamble                         {RECOGNIZE("preamble");}
                | string                           {RECOGNIZE("string");}
                | error RBRACE                     {RECOGNIZE("error");}
                ;

comment:          COMMENT opt_space LITERAL        {RECOGNIZE("comment");}
                ;

entry:            entry_head assignment_list
                        RBRACE                     {RECOGNIZE("entry-1");}
                | entry_head assignment_list
```

```
                        COMMA opt_space RBRACE    {RECOGNIZE("entry-2");}
                    | entry_head RBRACE           {RECOGNIZE("entry-3");}
                    ;

entry_head:         ENTRY opt_space
                        LBRACE opt_space
                        key_name opt_space
                        COMMA opt_space           {RECOGNIZE("entry_head");}
                    ;

key_name:           KEY                           {RECOGNIZE("key_name-1");}
                    | ABBREV                      {RECOGNIZE("key_name-2");}
                    ;

include:            INCLUDE opt_space LITERAL     {RECOGNIZE("include");}
                    ;

preamble:           PREAMBLE opt_space
                        LBRACE opt_space
                        value opt_space
                        RBRACE                    {RECOGNIZE("preamble");}
                    ;

string:             STRING opt_space
                        LBRACE opt_space
                        assignment
                        opt_space RBRACE          {RECOGNIZE("string");}
                    ;

value:              simple_value                  {RECOGNIZE("value-1");}
                    | value opt_space             {RECOGNIZE("value-1-1");}
                        SHARP                     {RECOGNIZE("value-1-2");}
                        opt_space simple_value    {RECOGNIZE("value-2");}
                    ;

simple_value:       VALUE                         {RECOGNIZE("simple_value-1");}
                    | ABBREV                      {RECOGNIZE("simple_value-2");}
                    ;

assignment_list:    assignment                    {RECOGNIZE("single assignment");}
                    | assignment_list COMMA opt_space
                        assignment                {RECOGNIZE("assignment-list");}
                    ;

assignment:         assignment_lhs opt_space
                        EQUALS opt_space          {RECOGNIZE("assignment-0");}
                        value opt_space           {RECOGNIZE("assignment");}
                    ;

assignment_lhs:     FIELD                         {RECOGNIZE("assignment_lhs-1");}
                    | ABBREV                      {RECOGNIZE("assignment_lhs-2");}
                    ;

opt_space:          /* empty */                   {RECOGNIZE("opt_space-1");}
```

```
                | space                           {RECOGNIZE("opt_space-2");}
                ;

space:            single_space                  {RECOGNIZE("single space");}
                | space single_space            {RECOGNIZE("multiple spaces");}
                ;

single_space:     SPACE
                | INLINE
                | NEWLINE
                ;
```

PERFORMANCE

As a demonstration of the efficiency of parsing, tests were carried out on a Sun 336MHz UltraSPARC system, with all programs compiled at the highest optimization level, and present in the current directory, using a 4MB test file (the largest from the TEX User Group bibliography archive) present in the memory-mapped */tmp* directory for fast access. The tests were run ten times inside a shell script to amortize the script startup time, and the total wall-clock time (from the UNIX **time**(1) command) for each script's execution was then divided by ten to produce these results:

```
-------------------------------------------------------------------
Program pipeline                                        Time  Relative
                                                              time
-------------------------------------------------------------------
 bibclean -no-prettyprint -no-warnings | bibparse 3.786s  3.67
 bibtex                                            3.313s  3.21
 biblex | bibparse                                 2.403s  2.33
 bibparse                                          1.030s  1.00
-------------------------------------------------------------------
```

The BIBTEX run used the TEX `\nocite{*}` command to generate citations in the `is-alpha` style of every entry in the bibliography.

The addition of support in **bibparse** version 1.04 for direct processing of BIBTEX files via an internal copy of the **biblex**(1) lexical analyzer has thus produced a 2.3-times speedup over previous versions that required **biblex**(1), and at data rates of 4MB/s, the programs are fast enough on 1999-vintage desktop computers to require only a small fraction of a second to process a typical BIBTEX bibliography, so they can be used routinely to validate such files.

SEE ALSO

bibcheck(1), **bibclean**(1), **bibdup**(1), **bibextract**(1), **bibjoin**(1), **biblabel**(1), **biblex**(1), **biborder**(1), **bibsearch**(1), **bibsort**(1), **bibtex**(1), **bibunlex**(1), **citefind**(1), **citesub**(1), **citetags**(1), **latex**(1), **scribe**(1), **tex**(1).

AUTHOR

Nelson H. F. Beebe
University of Utah
Department of Mathematics, 110 LCB
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA
Email: `beebe@math.utah.edu`, `beebe@acm.org`, `beebe@computer.org` (Internet)
WWW URL: `http://www.math.utah.edu/~beebe`
Telephone: +1 801 581 5254
FAX: +1 801 581 4148