SPLINT
reference

version 1.3.0

LD PARSER REFERENCE

Alex Shibakov

January 30, 2024

0 Introduction

This is a manual documenting the development of a parser that can be used to typeset `ld` files (linker scripts) with or without the help of `CWEB`. An existing parser for `ld` has been adopted as a base, with appropriately designed actions specific to the task of typesetting. The appendix to this manual contains the full source code (including the parts written in C) of both the scanner and the parser for `ld`, used in the original program. Some very minor modifications have been made to make the programs more 'presentable' in `CWEB` (in particular, the file had to be split into smaller chunks to satisfy `CWEAVE`'s limitations).

Nearly every aspect of the design is discussed, including the supporting TEX macros that make both the parser and this documentation possible. The TEX macros presented here are collected in `ldman.sty` and `ldsetup.sty` which are later included in the TEX file produced by `CWEAVE`.

⟨ Set up the generic parser machinery ch0 ⟩ =

```
\ifx\optimization\UNDEFINED %   ▷ this trick is based on the premise that \UNDEFINED ◁
    \def\optimization{0}    %   ▷ is never defined nor created with \csname...\endcsname ◁
\fi

\let\nx\noexpand    %   ▷ a convenient shortcut ◁

\input limbo.sty    %   ▷ general setup macros ◁
\input yycommon.sty %   ▷ general routines for stack and array access ◁
\input yymisc.sty   %   ▷ helper macros (stack manipulation, table processing, value stack pointers ◁
                    %   ▷ parser initialization, optimization) ◁
\input yyinput.sty  %   ▷ input functions ◁
\input yyparse.sty  %   ▷ parser machinery ◁
\input flex.sty     %   ▷ lexer functions ◁

\ifnum\optimization>\tw@
    \input yyfaststack.sty
\fi

\input yystype.sty  %   ▷ scanner auxiliary types and functions ◁
\input yyunion.sty  %   ▷ parser data structures ◁

\amendswitch        %   ▷ adjust the \yyinput to recognize \yyendgame ◁
    \multicharswitch\near\yyeof\by\yyendgame\to\multicharswitch %   ▷ add a new label ◁
\replaceaction\multicharswitch\at\yyendgame %   ▷ replace the new empty action ◁
    \by{{\yyinput\yyeof\yyeof\endparseinput\removefinalvb}}\to\multicharswitch
```

```
\def\MRI{{\sc MRI}} %    ▷ so we can say that MRI script typesetting is not supported... ◁
```

This code is used in section 7d.

3a Up to this point, the parser initialization process has been non parser specific (although sensitive to the namespace chosen). The next command inputs the data structures used by the `ld` parser. The reader should consult the file below for the details on the AST produced by the parser and its semantics.

⟨ Define the normal mode 3a ⟩ = 5a ▽

```
\input ldunion.sty  %    ▷ ld parser data structures ◁
```

See also sections 5a, 5c, 6a, 6b, 6c, and 7a.

This code is used in section 8a.

3b **Bootstrapping**

To produce a usable parser/scanner duo, several pieces of code must be generated. The most important of these are the *table files* (`ldptab.tex` and `ldltab.tex`) for the parser and the scanner. These consist of the integer tables defining the operation of the parser and scanner automata, the values of some constants, and the 'action switch'.

Just like in the case of 'real' parsers and scanners, in order to make the parser and the scanner interact seamlessly, some amount of 'glue' is required. As an example, a file containing the (numerical) definitions of the token values is generated by `bison` to be used by a `flex` generated scanner. Unfortunately, this file has too little structure for our purposes (it contains definitions of token values mixed in with other constants making it hard to distinguish one kind of definition from another). Therefore, the 'glue' is generated by parsing our grammar once again, this time with a `bison` grammar designed for typesetting `bison` files. A special *bootstrapping* mode is used to extract the appropriate information. The name 'bootstrapping' notwithstanding, the parser and lexer used in the bootstrapping phase are not necessarily the minimized versions used in bootstrapping the `bison` parser.

One component generated during the bootstrapping pass is a list of 'token equivalences' (or 'aliases') to be used by the lexer. Every token (to be precise, every *named token type*) used in a `bison` grammar is declared using one of the ⟨`token`⟩, ⟨`left`⟩, ⟨`right`⟩, ⟨`precedence`⟩, or ⟨`nonassoc`⟩ declarations. If no *alias* (see below) has been declared using a ⟨`token`⟩ declaration, this name ends up in the *yytname* array output by `bison` and can be used by the lexer after associating the token names with their numerical values (accomplished by `\settokens`). If all tokens are named tokens, no token equivalence list is necessary to set up the interaction between the lexer and the parser. In this case (the present `ld` parser is a typical example), the token list may also serve a secondary role: to provide hints for the macros that typeset the grammar terms, after the `\tokeneq` macro is redefined for this purpose.

On the other hand, after a declaration such as '⟨`token`⟩ `CHAR "char"`' the string `"char"` becomes an alias for the named token `CHAR`. Only the string version gets recorded in the *yytname* array. Establishing the equivalence between the two token forms now can be accomplished only by examining the grammar source file and is delegated to the bootstrapping phase parser.

Another (perhaps most important) goal of the bootstrapping phase is to extract the information about `flex` *states* used by the lexer from the appropriate source file. As is the case with token names, this information is output in a rather chaotic fashion by the scanner generator and is all but useless for our purposes. The bootstrapping macros are designed to handle `flex`'s ⟨`x`⟩ and ⟨`s`⟩ state declarations and produce a C file with the appropriate definitions. This file can later be included by the 'driver' routine to generate the appropriate table file for the lexer. To round off the bootstrapping mode we only need to establish the output streams for the tokens and the states, supply the appropriate file names for the two lists, flag the bootstrapping mode for the bootstrapping macros and inline typesetting (`\prodstyle` macros) and input the appropriate machinery.

This is done by the macros below. The bootstrap lexer setup (`\bootstraplexersetup`) consists of inputting the token equivalence table for the `bison` parser (i.e. the parser that processes the `bison` grammar file) and defining a robust token output function which simply ignores the token values the lexer is not aware of (it should not be necessary in our case since we are using a full featured lexer).

⟨ Define the bootstrapping mode 3b ⟩ =

```
\def\modebootstrap{%
    \def\bootstraplexersetup{%
        \let\yylexreturn\yylexreturnbootstrap    ▷ only return tokens whose value is known ◁
        %\let\yylexreturn\yylexreturnregular    ▷ should also work ◁
    }%
    \input yybootstrap.sty%
    \input yytexlex.sty%
}
```

This code is used in section 7d.

4a **Namespaces and modes**

Every parser/lexer pair (as well as some other macros) operates within a set of dedicated *namespaces*. This simply means that the macros that output token values, switch lexer states and access various tables 'tack on' a string of characters representing the current namespace to the 'low level' control sequence name that performs the actual output or access. Say, `\yytname` becomes an alias of `\yytname[main]` while in the `[main]` namespace. When a parser or lexer is initialized, the appropriate tables are aliased with a generic name in the case of an 'unoptimized' parser or lexer. The optimized parser or lexer handles the namespace referencing internally.

The mode setup macros for this manual define several separate namespaces:

- **main**: the `[main]` namespace is established for the parser that does the typesetting of the grammar.
- **ld**: every time a term name is processed, the token names are looked up in the `[ld]` namespace. The same namespace is used by the parser that typesets `ld` script examples in the manual (i.e. the parser described here). This is done to provide visual consistency between the description of the parser and its output.
- **small** and **ldsmall**: the `[small]` namespace is used by the term name parser itself. Since we use a customized version of the name parser, we dedicate a separate namespace for this purpose, `[ldsmall]`.
- **prologue**: the parser based on a subset of the full `bison` grammar describing prologue declarations uses the `[prologue]` namespace.
- **index**: the `[index]` namespace is used for typesetting the index entries for the terms that are automatically inserted by the main parser (such as an empty right hand side of a production or an inline action).
- **index:tex**, **index:visual**, and **texline**: the macros that typeset TEX entries, use `index:tex` as a pseudonamespace to display TEX terms in the index (due to the design of these typesetting macros, many of them take parameters, which can lead to chaos in the index). The `index:visual` pseudonamespace is used for ordering the index entries for TEX macros (this is how the entry for `\getsecond` (π_2) ends up in the P section of the index). Finally, the TEX typesetting macros use the `texline` pseudonamespace to process the names of TEX command sequences in action code. The reason we refer to these as *pseudo*namespaces is that only the token (or term) names are aliased, and not, say, the finite automata names in the case of the TEX typesetting.
- **flexre**, **flexone,** and **flextwo**: the parsers for `flex` input use the `[flexre]`, `[flexone]`, and `[flextwo]` namespaces for their operation. Another convention is to use the `\flexpseudonamespace` to typeset `flex` state names, and the `\flexpseudorenamespace` for typesetting the names of `flex` regular expressions. Currently, `\flexpseudo...` namespaces are set equal to their non-`pseudo` versions by default. This setting may be changed whenever several parsers are used in the same document and tokens with the same names must be typeset in different styles. All `flex` namespaces, as well as `[main]`, `[small]`, and `[ldsmall]` are defined by the `\genericparser` macros.
- **cwebclink**: finally, the `[cwebclink]` namespace is used for typesetting the variables *inside* `ld` scripts. This way, the symbols exported by the linker may be typeset in a style similar to C variables, if desired (as they play very similar roles).

⟨ Begin namespace setup 4a ⟩ =

```
\def\indexpseudonamespace{[index]}
\def\cwebclinknamespace{[cwebclink]}
\let\parsernamespace\empty
```

This code is used in section 7d.

5a After all the appropriate tables and 'glue' have been generated, the typesetting of this manual can be handled by the `normal` mode. Note that this requires the `ld` parser, as well as the `bison` parser, including all the appropriate machinery.

The normal mode is started by including the tables and lists and initializing the `bison` parser (accomplished by inputting `yyinit.sty`), followed by handling the token typesetting for the `ld` grammar.

⟨ Define the normal mode 3a ⟩ + = △ 3a 5c ▽

```
\newtoks\ldcmds

\def\modenormal{%
    \commonstartup
    \input ldtexlex.sty%    ▷ TEX typesetting specific to ld ◁
    \expandafter\def\csname index domain translation [L]\endcsname{%
        {\noexpand\ld\ index}%    ▷ used to typeset the table of contents ◁
        {ld index}%    ▷ outline entry ◁
        {L\sc D INDEX}%    ▷ index header ◁
    }%
    \def\otherlangindexheader{%
        B{\sc ISON}, F{\sc LEX}, \TeX, {\sc AND} L{\sc D INDICES}%
    }%    ▷ modify the index header ◁
    ⟨ Initialize ld parsers 7b ⟩
    ⟨ Modified name parser for ld grammar 39a ⟩
}
```

5b ⟨ Common startup routine 5b ⟩ =

```
\def\commonstartup{
    \def\appendr##1##2{\edef\appnext{##1{\the##1##2}}\appnext}%
    \def\appendl##1##2{\edef\appnext{##1{##2\the##1}}\appnext}%
    \input yyinit.sty  %
    \input yytexlex.sty%    ▷ TEX typesetting macros ◁
    \input gindex.sty  %    ▷ indexing macros specific to flex, bison, and ld ◁
    \input noweb.sty   %    ▷ noweb style references ◁
        \xreflocaltrue
        \let\sectionlistsetup\lxrefseparator
    \let\inx\inxmod
    \let\fin\finmod
    \termindextrue
    \immediate\openout\gindex=\jobname.gdx
    \let\hostparsernamespace\ldnamespace
  ▷ the namespace where tokens are looked up for typesetting purposes ◁
}
```

This code is used in section 7d.

5c The `ld` parser initialization requires setting a few global variables, as well as entering the `INITIAL` state for the `ld` lexer. The latter is somewhat counterintuitive and is necessitated by the ability of the parser to switch lexer states. Thus, the parser can switch the lexer state before the lexer is invoked for the first time wreaking havoc on the lexer state stack.

⟨ Define the normal mode 3a ⟩ + = △ 5a 6a ▽

```
\def\ldparserinit{%
    \basicparserinit
    \includestackptr=\@ne
    \versnodenesting=\z@
    \ldcmds{}%
    \yyBEGIN{INITIAL}%
}
```

6a This is the `ld` parser invocation routine. It is coded according to a straightforward sequence initialize-invoke-execute-or-fall back. The `\preparseld` macro is invoked by `\lsectionbegin` (see `limbo.sty`). It starts by defining the postprocessing and typesetting macro (`\postparseld`) followed by the parser setup.

⟨ Define the normal mode 3a ⟩ + = 5c 6b

```
\def\preparseld{%
    \let\postparse\postparseld
    \hidecslist\ldunion %    ▷ inhibit expansion so that fewer \noexpands are necessary ◁
    \toldparser
    \displayrawtable %    ▷ do this after the parser namespaces are setup ◁
    \ldparserinit
    \yyparse
}
```

6b The postprocessing macro defines a procedure for typesetting the output and saving the parsed result to a file. After that, a generic postprocessing macro is executed.

⟨ Define the normal mode 3a ⟩ + = 6a 6c

```
\def\postparseld{%
    \let\saveparsedtable\saveldtable
    \let\typesetparsedtables\typesetldtables
    \postparsegeneric{(ld script)}%
}
```

6c The table output and typesetting macros are responsible for the parsed table output to a log file and the typographic representation of the result. The table is output after expanding its contents. The commands

```
\hidecslist\cwebstreamchars
\restorecslist{ld-parser-debug}\ldunion
```

are meant to inhibit most expansions so that only the list (iterator) macros are expanded. The parser for `ld` does not currently use the list macros to speed up the parsing process but the general set up is here in case it is needed in the future. Unfortunately, the same technique cannot be applied to the display of the `\yystash` and `\yyformat` streams (that use linked lists by default), since there are too many random sequences that can appear in such streams. Therefore, to facilitate debugging, one should expand the lists before displaying them (see `yyinput.sty` for details; lists with iterators are a bit more convenient due to their flexibility).

⟨ Define the normal mode 3a ⟩ + = 6b 7a

```
\def\saveldtable#1{{%
    \hidecslist\cwebstreamchars
    \restorecslist{ld-parser-debug}\ldunion
    \expandafter\saveoutputcode\expandafter{\the\ldcmds}\exampletable{#1}%
}}

\def\typesetldtables{%
    \begingroup
        \displayparsedoutput\ldcmds
        \restorecslist{ld-parser:restash}\ldunion %    ▷ mark variables, preprocess stash ◁
        \setprodtable
        \the\ldcmds
        \restorecslist{ld-display}\ldunion
        \setprodtable    ▷ use the bison's parser typesetting definitions ◁
        \restorecs{ld-display}{\anint\bint\hexint} %    ▷ ... except for integer typesetting ◁
        \the\ldcmds
        \par
        \vskip-\baselineskip
        \the\lddisplay
    \endgroup
}
```

7a The parsing routine defined above is the first macro in the `ld` parser stack. The only remaining procedure on the stack is an error reporting macro in case the parsing pass failed.

⟨ Define the normal mode 3a ⟩ + = △ 6c

```
\fillpstack{l}{%
    \preparseld
    {\preparsefallback{++}}%    ▷ skip this section if parsing failed, put ++ on the screen ◁
    \relax %    ▷ this \relax 'guards' the braces above during \poppstack ◁
}
```

7b Unless they are being bootstrapped, the `ld` parser and its term parser are initialized by the normal mode. The token typesetting of `ld` grammar tokens is adjusted at the same time (see the remarks above about the mechanism that is responsible for this). Most terminals (such as keywords, etc.) may be displayed unchanged (provided the names used by the lexer agree with their appearance in the script file, see below), while the typeseting of others is modified in `ltokenset.sty`.

In the original `bison-flex` interface, token names are defined as straightforward macros (a poor choice as will be seen shortly) which can sometimes clash with the standard C macros. This is why `ld` lexer returns `ASSERT` as `ASSERT_K`. The name parser treats `K` as a suffix to supply a visual reminder of this flaw. Note that the 'suffixless' part of these tokens (such as `ASSERT`) is never declared and thus has to be entered in `ltokenset.sty` by hand.

The tokens that never appear as part of the input (such as end and unary) or those that do but have no fixed appearance (for example, name) are typeset in a style that indicates their origin. The details can be found by examining `ltokenset.sty`.

⟨ Initialize `ld` parsers 7b ⟩ =

```
\genericparser
    name: ld,
    ptables: ldptab.tex,
    ltables: ldltab.tex,
    tokens: {},
    asetup: {},
    dsetup: {},
    rsetup: {},
    optimization: {};%    ▷ the parser and lexer are optimized when output ◁
\genericprettytokens
    namespace: ld,
    tokens: ldp.tok,
    correction: ltokenset.sty,
    host: ld;%
```

This code is used in section 5a.

7c We also need some modifications to the indexing macros in order to typeset `ld` terms and variables separately.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ = 26d ▽

See also sections 26d, 27a, 27b, 27c, 28a, 33a, and 33d.

This code is used in section 8a.

7d The macros are collected in two files included at the beginning of this documentation [1]).

⟨ `ldman.stx` 7d ⟩ =

```
⟨ Set up the generic parser machinery ch0 ⟩
⟨ Begin namespace setup 4a ⟩
⟨ Define the bootstrapping mode 3b ⟩
⟨ Common startup routine 5b ⟩
```

[1]) An attentive reader may have noticed that the files have the extension `.stx` instead of the traditional `.sty`. The reason for this is the postprocessing step that 'cleans' the generated files of various C artifacts output by `CWEB` and turns `ldman.stx` and `ldsetup.stx` into `ldman.sty` and `ldsetup.sty` included by `ldman.tex`...

8a The macros collected in the file below are not specific to this manual but are needed in order to use the `ld` parser generated and are thus put in a separate which can be included by other TEX programs that wish to use them.

⟨ `ldsetup.stx` 8a ⟩ =
⟨ Define the normal mode 3a ⟩
⟨ Additional macros for the `ld` lexer/parser 7c ⟩

1 The parser

The outline of the grammar file below does not reveal anything unusual in the general layout of `ld` grammar. The first section lists all the token definitions, ⟨`union`⟩ styles, and some C code. The original comments that come with the grammar file of the linker have been mostly left intact. They are typeset in *italics* to make them easy to recognize.

⟨ `ldp.yy` ch1 ⟩ =
```
.....
⟨ ld parser C preamble 9b ⟩
.....
⟨ ld parser bison options 9a ⟩
⟨union⟩        ⟨ Union of grammar parser types 10a ⟩
.....
⟨ ld parser C postamble 10b ⟩
.....
⟨ Token and type declarations 10d ⟩

⟨ ld parser productions 10c ⟩
```

9a Among the options listed in this section, ⟨`token-table`⟩ is the most critical for the proper operation of the parser and must be enabled to supply the token information to the lexer (the traditional way of passing this information along is to use a C header file with the appropriate definitions). The start symbol does not have to be given explicitly and can be indicated by listing the appropriate rules at the beginning.

Most other sections of the grammar file, with the exception of the rules are either empty or hold placeholder values. The functionality provided by the code in these sections in the case of a C parser is supplied by the TEX macros in `ldman.sty`.

⟨ `ld` parser `bison` options 9a ⟩ =
```
⟨token table⟩ ⋆
⟨parse.trace⟩ ⋆    (set as ⟨debug⟩)
⟨start⟩            script_file
```
This code is used in section ch1.

9b ⟨ `ld` parser C preamble 9b ⟩ =

This code is used in section ch1.

10a ⟨ Union of grammar parser types 10a ⟩ =
This code is used in section ch1.

10b ⟨ `ld` parser C postamble 10b ⟩ =
This code is used in section ch1.

10c ⟨ `ld` parser productions 10c ⟩ =
⟨ GNU `ld` script rules 12a ⟩
⟨ Grammar rules 11a ⟩
This code is used in section ch1.

10d The tokens are declared first. This section is also used to supply numerical token values to the lexer by the original parser, as well as the bootstrapping phase of the typesetting parser. Unlike the native (C) parser for `ld` the typesetting parser has no need for the type of each token (rather, the type consistency is based on the weak dynamic type system coded in `yyunion.sty` and `ldunion.sy`). Thus all the tokens used by the `ld` parser are put in a single list.

⟨ Token and type declarations 10d ⟩ =

INT	name	name_{L}	end
$\text{ALIGN}_{\_\text{K}}$	BLOCK	BIND	QUAD
SQUAD	LONG	SHORT	BYTE
SECTIONS	PHDRS	$\text{INSERT}_{\_\text{K}}$	AFTER
BEFORE	DATA_SEGMENT_ALIGN	DATA_SEGMENT_RELRO_END	DATA_SEGMENT_END
SORT_BY_NAME	SORT_BY_ALIGNMENT	SORT_NONE	SORT_BY_INIT_PRIORITY
{	}	SIZEOF_HEADERS	OUTPUT_FORMAT
FORCE_COMMON_ALLOCATION	OUTPUT_ARCH	INHIBIT_COMMON_ALLOCATION	SEGMENT_START
INCLUDE	MEMORY	REGION_ALIAS	LD_FEATURE
NOLOAD	DSECT	COPY	INFO
OVERLAY	DEFINED	$\text{TARGET}_{\_\text{K}}$	SEARCH_DIR
MAP	ENTRY	NEXT	SIZEOF
ALIGNOF	ADDR	LOADADDR	$\text{MAX}_{\_\text{K}}$
$\text{MIN}_{\_\text{K}}$	STARTUP	HLL	SYSLIB
FLOAT	NOFLOAT	NOCROSSREFS	ORIGIN
FILL	LENGTH	CREATE_OBJECT_SYMBOLS	INPUT
GROUP	OUTPUT	CONSTRUCTORS	ALIGNMOD
AT	SUBALIGN	HIDDEN	PROVIDE
PROVIDE_HIDDEN	AS_NEEDED	CHIP	LIST
SECT	ABSOLUTE	LOAD	NEWLINE
ENDWORD	ORDER	NAMEWORD	$\text{ASSERT}_{\_\text{K}}$
LOG2CEIL	FORMAT	PUBLIC	DEFSYMEND
BASE	ALIAS	TRUNCATE	REL
INPUT_SCRIPT	INPUT_MRI_SCRIPT	INPUT_DEFSYM	CASE
EXTERN	START	VERS_TAG	VERS_IDENTIFIER
GLOBAL	LOCAL	$\text{VERSION}_{\text{K}}$	INPUT_VERSION_SCRIPT
KEEP	ONLY_IF_RO	ONLY_IF_RW	SPECIAL
INPUT_SECTION_FLAGS	ALIGN_WITH_INPUT	EXCLUDE_FILE	CONSTANT
INPUT_DYNAMIC_LIST			

⟨**right**⟩ $\Leftarrow^{+} \Leftarrow \Leftarrow^{\times} \Leftarrow^{\div} \Leftarrow \Leftarrow^{\ll} \Leftarrow^{\gg} \Leftarrow^{\&} \Leftarrow^{\vee}$ `?` `:` unary
⟨**left**⟩ $\vee \wedge \mid \oplus \,\&\, = \neq < > \leqslant \geqslant \ll \gg + - \times / \div$ `(`
This code is used in section ch1.

10e **Grammar rules, an overview**

The first natural step in transforming an existing parser into a 'parser stack' for pretty printing is to understand the 'anatomy' of the grammar. Not every grammar is suitable for such a transformation and in almost every case, some modifications are needed. The parser and lexer implementation for `ld` is not terrible

although it does have some idiosynchrasies that could have been eliminated by a careful grammar redesign. Instead of invasive rewriting of significant portions of the grammar, the approach taken here merely omits some rules and partitions the grammar into several subsets, each of which is supposed to handle a well defined logical section of an `ld` script file.

One example of a trick used by the `ld` parser that is not appropriate for a pretty printing grammar is the way the original parser handles the choice of the format of an input file. After a command line option that selects the input format has been read (or the format has been determined using some other method), the first token output by the lexer branches the parser to the appropriate portion of the full grammar.

Since the token never appears as part of the input file there is no need to include this part of the main grammar for the purposes of typesetting.

⟨ Ignored grammar rules 10e ⟩ =

file :

`INPUT_SCRIPT` *script_file*
`INPUT_MRI_SCRIPT` *mri_script_file*
`INPUT_VERSION_SCRIPT` *version_script_file*
`INPUT_DYNAMIC_LIST` *dynamic_list_file*
`INPUT_DEFSYM` *defsym_expr*

11a ⟨ Grammar rules 11a ⟩ = 15c ▽

filename : name $\Upsilon \leftarrow \langle^{\text{nox}}$`\ldfilename{` val Υ_1 `}`$\rangle$

See also sections 15c, 15d, 16d, 17a, 17b, 17h, 18e, 19a, 19b, 20c, 20d, 21c, 21d, 22c, and 23c.

This code is used in section 10c.

11b The simplest parser subset is intended to parse symbol definitions given in the command line that invokes the linker. Creating a parser for it involves almost no extra effort so we leave it in.

Note that the simpliciy is somewhat deceptive as the syntax of *exp* is rather complex. That part of the grammar is needed elsewhere, however, so symbol definitions parsing costs almost nothing on top of the already required effort. The only practical use for this part of the `ld` grammar is presenting examples in text.

The TEX macro `\ldlex@defsym` switches the lexer state to `DEFSYMEXP` (see all the state switching macros in the chapter about the lexer implementation below). Switching lexer states from the parser presents some difficulties which can be overcome by careful design. For example, the state switching macros can be invoked before the lexer is called and initialized (when the parser performs a *default action*).

⟨ Inline symbol definitions 11b ⟩ =

defsym_expr :

∘ `\ldlex@defsym`
name ⇐ *exp* `\ldlex@popstate`

11c *Syntax within an* MRI *script file* [1]). The parser for typesetting is only intended to process GNU `ld` scripts and does not concern itself with any additional compatibility modes. For this reason, all support for MRI style scripts has been omitted. One use for the section below is a small demonstration of the formatting tools that change the output of the `bison` parser.

⟨ MRI style script rules 11c ⟩ =

mri_script_file : ∘ ◇ *mri_script_lines* ... | `\ldlex@popstate`

mri_script_lines : *mri_script_lines mri_script_command* `NEWLINE` | ∘

mri_script_command :

`CHIP` *exp*
`CHIP` *exp* `,` *exp*
name
`LIST`
`ORDER` *ordernamelist*

[1]) As explained at the beginning of this chapter, the text in *italics* was taken from the original comments by the `ld` parser and lexer programmers.

`ENDWORD`
`PUBLIC` name ⇐ *exp* | `PUBLIC` name `,` *exp* | `PUBLIC` name *exp*
`FORMAT` name
`SECT` name `,` *exp* | `SECT` name *exp* | `SECT` name ⇐ *exp*
`ALIGN_K` name ⇐ *exp* | `ALIGN_K` name `,` *exp*
`ALIGNMOD` name ⇐ *exp* | `ALIGNMOD` name `,` *exp* ◇ ○
`ABSOLUTE` *mri_abs_name_list*
`LOAD` *mri_load_name_list*
`NAMEWORD` name
`ALIAS` name `,` name | `ALIAS` name `,` `INT` ◇ ○
`BASE` *exp*
`TRUNCATE` `INT`
`CASE` *casesymlist*
`EXTERN` *extern_name_list*
`INCLUDE` *filename* ◇ *mri_script_lines* end ⟨ Close the file 15b ⟩
`START` name
○

ordernamelist : *ordernamelist* `,` name | *ordernamelist* name | ○

mri_load_name_list : name | *mri_load_name_list* `,` name

mri_abs_name_list : name | *mri_abs_name_list* `,` name

casesymlist : ○ | name | *casesymlist* `,` name

12a *Parsed as expressions so that commas separate entries.* The core of the parser consists of productions describing GNU `ld` linker scripts. The first rule is common to both MRI and GNU formats.

⟨ GNU `ld` script rules 12a ⟩ = 12b ▽

extern_name_list :
○ `\ldlex@expression`
extern_name_list_body `\ldlex@popstate`

extern_name_list_body :
name
extern_name_list_body name
extern_name_list_body `,` name

See also sections 12b and 13a.

This code is used in section 10c.

12b The top level productions simply define a script file as a list of script commands.

⟨ GNU `ld` script rules 12a ⟩ + = △ 12a 13a ▽

script_file :
○ `\ldlex@both`
ifile_list $\pi_5(\Upsilon_2) \mapsto$ `\ldcmds` `\ldlex@popstate`

ifile_list :
ifile_list *ifile_p*$_1$ ⟨ Add the next command 12c ⟩
○ $\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\ldinsertcweb { }{ }{ }{ }` $\rangle$

12c ⟨ Add the next command 12c ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h$
`\yytoksempty {` v_h `}{` $\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$ `}{ \yytoksempty {` v_d `}{` $\Upsilon \leftarrow \langle$val $\Upsilon_2\rangle$ `}`
`{` $\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\ldinsertcweb {` val v_a `}{` val v_b `}{` val v_g `}{` val v_d
$^{\mathrm{nx}}$`\ldcommandseparator {` val v_e `}{` val v_f `}{` val v_c `}{` val v_g `}`val v_h $\rangle$ `}}`

This code is used in section 12b.

13a

Script internals

There are a number of different commands. For typesetting purposes, the handling of most of these can be significantly simplified. In the `GROUP` command there is no need to perform any actions upon entering the group, for instance. `INCLUDE` presents a special challenge. In the original grammar this command is followed by a general list of script commands (the contents of the included file) terminated by end. The 'magic' of opening the file and inserting its contents into the stream being parsed is performed by the lexer and the parser in the background. The typesetting parser, on the other hand, only has to typeset the `INCLUDE` command itself and has no need for opening and parsing the file being included. We can simply change the grammar rule to omit the follow up script commands but that would require altering the existing grammar. Since the command list (*ifile_list*) is allowed to be empty, we simply fake the inclusion of the file in the lexer by immediately outputting end upon entering the appropriate lexer state. One advantage in using this approach is the ability, when desired, to examine the included file for possible cross-referencing information.

Each command is packaged with a qualifier that records its type for the rule that adds the fragment to the script file.

⟨ GNU `ld` script rules 12a ⟩ + = △ 12b

***ifile_p*$_1$** :

memory	⟨ Carry on 14f ⟩
sections	⟨ Carry on 14f ⟩
phdrs	
startup	
high_level_library	
low_level_library	
floating_point_support	
statement_anywhere	⟨ Carry on 14f ⟩
version	
;	$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{ none }{ }`$\rangle$
`TARGET`$_{\text{K}}$ `(` name `)`	
`SEARCH_DIR (` *filename* `)`	
`OUTPUT (` *filename* `)`	
`OUTPUT_FORMAT (` name `)`	
`OUTPUT_FORMAT (` name `,` name `,` name `)`	
`OUTPUT_ARCH (` name `)`	
`FORCE_COMMON_ALLOCATION`	
`INHIBIT_COMMON_ALLOCATION`	
`INPUT (` *input_list* `)`	
`GROUP` `(` *input_list* `)`	
`MAP (` *filename* `)`	
`INCLUDE` *filename*	⟨ Peek at a file 15a ⟩
ifile_list end	⟨ Close the file 15b ⟩
`NOCROSSREFS (` *nocrossref_list* `)`	
`EXTERN (` *extern_name_list* `)`	
`INSERT`$_{\text{K}}$ `AFTER` name	
`INSERT`$_{\text{K}}$ `BEFORE` name	
`REGION_ALIAS (` name `,` name `)`	
`LD_FEATURE (` name `)`	

input_list :

name
input_list `,` name
input_list name
name$_{\text{L}}$
input_list `,` name$_{\text{L}}$
input_list name$_{\text{L}}$
`AS_NEEDED (`

input_list)
input_list , AS_NEEDED (
input_list)
input_list AS_NEEDED (
input_list)

sections :
SECTIONS { *sec_or_group_p*$_1$ } ⟨ Form the SECTIONS group 14a ⟩

***sec_or_group_p*$_1$** :
sec_or_group_p$_1$ *section* ⟨ Add the next section chunk 14b ⟩
sec_or_group_p$_1$ *statement_anywhere* ⟨ Add the next section statement 14c ⟩
○ $\Upsilon \leftarrow \langle\rangle$

statement_anywhere :
ENTRY (name) ⟨ Form an ENTRY statement 14d ⟩
assignment end ⟨ Form a statement 14e ⟩
ASSERT$_{\text{K}}$ \ldlex@expression
(*exp* , name) \ldlex@popstate

14a ⟨ Form the SECTIONS group 14a ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb val Υ_1\{ sect \}\{ $^{\text{nx}}$\ldsections \{ val Υ_3 $^{\text{nx}}$\ldsectionstash val Υ_4 \}\}$\rangle$
This code is used in section 13a.

14b ⟨ Add the next section chunk 14b ⟩ =
\yytoksempty \{ Υ_2 \}\{ $\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$ \}\{
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h$
\yytoksempty \{ Υ_1 \}\{ $\Upsilon \leftarrow \langle^{\text{nx}}$\ldsectionstash \{ val v_e \}\{ val v_f \}val $v_h\rangle$ \}
\{ $\Upsilon \leftarrow \langle$val Υ_1 $^{\text{nx}}$\ldsectionseparator \{ val v_e \}\{ val v_f \}val $v_h\rangle$ \}
}
This code is used in section 13a.

14c ⟨ Add the next section statement 14c ⟩ =
\yytoksempty \{ Υ_2 \}\{ $\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$ \}\{
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h$
\yytoksempty \{ Υ_1 \}\{ $\Upsilon \leftarrow \langle^{\text{nx}}$\ldsectionstash \{ val v_e \}\{ val v_f \}$^{\text{nx}}$\ldfreestmt \{ val v_h \}$\rangle$ \}
\{ $\Upsilon \leftarrow \langle$val Υ_1 $^{\text{nx}}$\ldsectionseparator \{ val v_e \}\{ val v_f \}$^{\text{nx}}$\ldfreestmt \{ val v_h \}$\rangle$ \}
}
This code is used in section 13a.

14d ⟨ Form an ENTRY statement 14d ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb val Υ_1\{ stmt \}\{ $^{\text{nx}}$\ldstatement \{ $^{\text{nx}}$\ldentry \{ $^{\text{nx}}$\ldregexp \{ val Υ_3 \}\}\}\}$\rangle$
This code is used in section 13a.

14e ⟨ Form a statement 14e ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb \{ val v_a \}\{ val v_b \}\{ stmt \}\{ $^{\text{nx}}$\ldstatement \{ val v_d \}\}$\rangle$
This code is used in sections 13a and 16d.

14f This is the default action performed by the parser when the parser writer does not supply one. For a minor gain in efficiency, this definition can be made empty.
⟨ Carry on 14f ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$
This code is used in sections 13a, 15d, 16b, 16d, 17a, 17h, and 19b.

15a ⟨ Peek at a file 15a ⟩ =
```
\ldlex@script
\ldfile@open@command@file{ ϒ2 }
```
This code is used in sections 11c, 13a, 16d, 17h, and 20d.

15b ⟨ Close the file 15b ⟩ =
```
ϒ ← ⟨nx\ldinsertcweb val ϒ1{ inc }{nx\ldinclude{ val ϒ2 }}⟩ \ldlex@popstate
```
This code is used in sections 11c, 13a, 16d, 17h, and 20d.

15c *The* * *and* ? *cases are there because the lexer returns them as separate tokens rather than as* name.

⟨ Grammar rules 11a ⟩ + = △ 11a 15d ▽

wildcard_name :

name	⟨ Create a wildcard name 15e ⟩
*	ϒ ← ⟨nx\ldinsertcweb val ϒ1{ wld }{nx\ldregop{ { * }{ * }val ϒ1 }}⟩
?	ϒ ← ⟨nx\ldinsertcweb val ϒ1{ wld }{nx\ldregop{ { ? }{ ? }val ϒ1 }}⟩

15d ⟨ Grammar rules 11a ⟩ + = △ 15c 16d ▽

wildcard_spec :

wildcard_name	⟨ Carry on 14f ⟩
EXCLUDE_FILE (*exclude_name_list*) *wildcard_name*	
SORT_BY_NAME (*wildcard_name*)	
SORT_BY_ALIGNMENT (*wildcard_name*)	
SORT_NONE (*wildcard_name*)	
SORT_BY_NAME (SORT_BY_ALIGNMENT (*wildcard_name*))	
SORT_BY_NAME (SORT_BY_NAME (*wildcard_name*))	
SORT_BY_ALIGNMENT (SORT_BY_NAME (*wildcard_name*))	
SORT_BY_ALIGNMENT ↩ (SORT_BY_ALIGNMENT (*wildcard_name*))	
SORT_BY_NAME ↩ (EXCLUDE_FILE (*exclude_name_list*) *wildcard_name*)	
SORT_BY_INIT_PRIORITY (*wildcard_name*)	

sect_flag_list :

name
sect_flag_list & name

sect_flags :

INPUT_SECTION_FLAGS (*sect_flag_list*)

exclude_name_list :

exclude_name_list wildcard_name
wildcard_name

file_name_***list*** :

*file*_name_*list* ,opt_ *wildcard_spec*	⟨ Add a wildcard spec to a list of files 16a ⟩
wildcard_spec	⟨ Start a file list with a wildcard spec 16b ⟩

input_section_spec_no_keep :

name	
sect_flags name	
[*file*_name_*list*]	
sect_flags [*file*_name_*list*]	
wildcard_spec (*file*_name_*list*)	⟨ Add a plain section spec 16c ⟩
sect_flags wildcard_spec (*file*_name_*list*)	

15e ⟨ Create a wildcard name 15e ⟩ =
```
ϒ ← ⟨nx\ldinsertcweb{ val va }{ val vb }{ wld }{nx\ldregexp{ val ϒ1 }}⟩
```
This code is used in section 15c.

16a ⟨ Add a wildcard spec to a list of files 16a ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h \Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{flst}{` val v_d $^{\text{nx}}$`\ldspace` val v_h `}`$\rangle$
This code is used in section 15d.

16b ⟨ Start a file list with a wildcard spec 16b ⟩ =
⟨ Carry on 14f ⟩
This code is used in section 15d.

16c ⟨ Add a plain section spec 16c ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_5(\Upsilon_3) \mapsto v_h \Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{sspec}{` val v_d `(` val v_h `)}`$\rangle$
This code is used in section 15d.

16d ⟨ Grammar rules 11a ⟩ + = (15d 17a)

input_section_spec :

input_section_spec_no_keep	⟨ Carry on 14f ⟩
`KEEP (`	
input_section_spec_no_keep `)`	⟨ Add a `KEEP` statement 16f ⟩

statement :

assignment end	⟨ Form a statement 14e ⟩
`CREATE_OBJECT_SYMBOLS`	
`;`	$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{stmt}{}`$\rangle$
`CONSTRUCTORS`	
`SORT_BY_NAME ( CONSTRUCTORS )`	
input_section_spec	⟨ Form an input section spec 16e ⟩
length `(` *mustbe_exp* `)`	
`FILL (` *fill_exp* `)`	
`ASSERT`$_{\text{K}}$	`\ldlex@expression`
`(` *exp* `,` name `)` *end*	`\ldlex@popstate`
`INCLUDE` *filename*	⟨ Peek at a file 15a ⟩
statement_list$_{\text{opt}}$ end	⟨ Close the file 15b ⟩

statement_list :

statement_list statement	⟨ Attach a statement to a statement list 16g ⟩
statement	⟨ Carry on 14f ⟩

statement_list$_{\text{opt}}$:

∘	$\Upsilon \leftarrow \langle^{\text{nx}}$`\insertcweb{}{}{stmt}{}`$\rangle$
statement_list	⟨ Carry on 14f ⟩

16e ⟨ Form an input section spec 16e ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{stmt}{` $^{\text{nx}}$`\ldsecspec{` val v_d `}}`$\rangle$
This code is used in section 16d.

16f ⟨ Add a `KEEP` statement 16f ⟩ =
$\pi_5(\Upsilon_4) \mapsto v_a \Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{stmt}{` $^{\text{nx}}$`\ldkeep{` val v_a `}}`$\rangle$
This code is used in section 16d.

16g ⟨ Attach a statement to a statement list 16g ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$

$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h$
`\yytoksempty{` v_d `}{` $\Upsilon \leftarrow \langle$ val $\Upsilon_2\rangle$ `}`
`{` $\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{` val v_g `}{` val v_d
`\yytoksempty{` v_h `}{}{\yytoksempty{` v_d `}{}{`$^{\text{nx}}$`\ldor}`val v_h `}}⟩ }`

This code is used in section 16d.

17a ⟨ Grammar rules 11a ⟩ + = △ 16d 17b ▽

length : `QUAD` | `SQUAD` | `LONG` | `SHORT` | `BYTE`

fill_exp : *mustbe_exp* ⟨ Carry on 14f ⟩

fill$_{\text{opt}}$: ∘ | ⇐ *fill_exp* . . . | $\Upsilon \leftarrow \langle^{\text{nox}}$`\ldfill{` val Υ_2 `}⟩`

assign_op :

$\stackrel{+}{\Leftarrow}$ $\Upsilon_0 =$ `{\MRL{+{⇐}}}`

$\stackrel{-}{\Leftarrow}$ $\Upsilon_0 =$ `{\MRL{-{⇐}}}`

$\stackrel{\times}{\Leftarrow}$ | $\stackrel{\div}{\Leftarrow}$ | $\stackrel{\ll}{\Leftarrow}$ | $\stackrel{\gg}{\Leftarrow}$ | $\stackrel{\&}{\Leftarrow}$ | $\stackrel{\vee}{\Leftarrow}$ . . . | $\Upsilon_0 =$ `{` $\stackrel{\vee}{\Leftarrow}$ `}`

end : `;` | `,`

`,`$_{\text{opt}}$: `,` | ∘

17b Assignments are not expressions as in C.

⟨ Grammar rules 11a ⟩ + = △ 17a 17h ▽

assignment :

name ⇐ *mustbe_exp* ⟨ Process simple assignment 17c ⟩
name *assign_op mustbe_exp* ⟨ Process compound assignment 17d ⟩
`HIDDEN (` name ⇐ *mustbe_exp* `)` ⟨ Process a `HIDDEN` assignment 17e ⟩
`PROVIDE (` name ⇐ *mustbe_exp* `)` ⟨ Process a `PROVIDE` assignment 17f ⟩
`PROVIDE_HIDDEN (` name ⇐ *mustbe_exp* `)` ⟨ Process a `PROVIDE_HIDDEN` assignment 17g ⟩

17c ⟨ Process simple assignment 17c ⟩ =
$\pi_3(\Upsilon_1) \mapsto v_a \pi_4(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{asgnm}{`$^{\text{nx}}$`\ldassignment{`$^{\text{nx}}$`\ldregexp{` val Υ_1 `}}{⇐}{` val Υ_3 `}}⟩`
This code is used in section 17b.

17d ⟨ Process compound assignment 17d ⟩ =
$\pi_3(\Upsilon_1) \mapsto v_a \pi_4(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{asgnm}{`$^{\text{nx}}$`\ldassignment{`$^{\text{nx}}$`\ldregexp{` val Υ_1 `}}{` val Υ_2 `}{` val Υ_3 `}}⟩`
This code is used in section 17b.

17e ⟨ Process a `HIDDEN` assignment 17e ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{hiddn}{`$^{\text{nx}}$`\ldhidden{`$^{\text{nx}}$`\ldregexp{` val Υ_3 `}}{` val Υ_5 `}}⟩`
This code is used in section 17b.

17f ⟨ Process a `PROVIDE` assignment 17f ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{prvde}{`$^{\text{nx}}$`\ldprovide{`$^{\text{nx}}$`\ldregexp{` val Υ_3 `}}{` val Υ_5 `}}⟩`
This code is used in section 17b.

17g ⟨ Process a `PROVIDE_HIDDEN` assignment 17g ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{prhid}{`$^{\text{nx}}$`\ldprovidehid{`$^{\text{nx}}$`\ldregexp{` val Υ_3 `}}{` val Υ_5 `}}⟩`
This code is used in section 17b.

17h ⟨ Grammar rules 11a ⟩ + = △ 17b 18e ▽

memory :

`MEMORY {` *memory_spec_list*$_{\text{opt}}$ `}` ⟨ Form the `MEMORY` group 18a ⟩

memory_spec_list$_{\text{opt}}$:

memory_spec_list ⟨ Carry on 14f ⟩
∘ $\Upsilon \leftarrow \langle\rangle$

memory_spec_list :
memory_spec_list ,$_{\text{opt\_}}$ *memory_spec* ⟨ Add a memory spec 18c ⟩
memory_spec ⟨ Start a list of memory specs 18b ⟩

memory_spec :
name
attributes$_{\text{\_opt}}$: *origin_spec* ,$_{\text{opt\_}}$ *length_spec* ⟨ Declare a named memory region 18d ⟩
`INCLUDE` *filename* ⟨ Peek at a file 15a ⟩
memory_spec_list$_{\text{\_opt}}$ end ⟨ Close the file 15b ⟩

18a ⟨ Form the `MEMORY` group 18a ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{mem}{`$^{\text{nx}}$`\ldmemory{` val $\Upsilon_3{}^{\text{nx}}$`\ldmemspecstash` val Υ_4`}}`$\rangle$
This code is used in section 17h.

18b ⟨ Start a list of memory specs 18b ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_5(\Upsilon_1) \mapsto v_c$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldmemspecstash{` val v_a `}{` val v_b `}` val $v_c\rangle$
This code is used in section 17h.

18c ⟨ Add a memory spec 18c ⟩ =
$\pi_2(\Upsilon_3) \mapsto v_a \pi_3(\Upsilon_3) \mapsto v_b$
$\pi_5(\Upsilon_3) \mapsto v_c$
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\text{nx}}$`\ldmemspecseparator{` val v_a `}{` val v_b `}` val $v_c\rangle$
This code is used in section 17h.

18d ⟨ Declare a named memory region 18d ⟩ =
$\pi_3(\Upsilon_1) \mapsto v_a \pi_4(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{mreg}`
`{`$^{\text{nx}}$`\ldmemoryspec{` val Υ_1 `}{` val Υ_3 `}{` val Υ_5 `}{` val Υ_7 `}}`$\rangle$
This code is used in section 17h.

18e ⟨ Grammar rules 11a ⟩ + = △ 17h 19a ▽

origin_spec :
`ORIGIN` ⇐ *mustbe_exp* $\Upsilon \leftarrow \langle^{\text{nox}}$`\ldoriginspec{` val Υ_3 `}`$\rangle$

length_spec :
`LENGTH` ⇐ *mustbe_exp* $\Upsilon \leftarrow \langle^{\text{nox}}$`\ldlengthspec{` val Υ_3 `}`$\rangle$

attributes$_{\text{\_opt}}$:
○ $\Upsilon \leftarrow \langle\rangle$
`(` *attributes_list* `)` $\Upsilon \leftarrow \langle$val $\Upsilon_2\rangle$

attributes_list :
attributes_string $\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$
attributes_list attributes_string $\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\text{nox}}$`\ldspace` val $\Upsilon_2\rangle$

attributes_string :
name $\Upsilon \leftarrow \langle^{\text{nox}}$`\ldattrstring` val $\Upsilon_1\rangle$
¬ name $\Upsilon \leftarrow \langle^{\text{nox}}$`\ldattrstringneg` val $\Upsilon_2\rangle$

startup :
`STARTUP (` *filename* `)`

high_level_library :
`HLL (` *high_level_library*_name_*list* `)`
`HLL ( )`

high_level_library_name_***list*** :
*high_level_library*_name_*list* ,$_{\text{opt\_}}$ *filename*
filename

low_level_library :
`SYSLIB (` *low_level_library*_name_*list* `)`

low_level_library_name_***list*** :
low_level_library_*name_*list ,$_{\text{opt\_}}$ *filename*
∘

floating_point_support :
`FLOAT`
`NOFLOAT`

nocrossref_list :
∘
name *nocrossref_list*
name , *nocrossref_list*

mustbe_exp :
∘ `\ldlex@expression`
exp `\ldlex@popstate` $\Upsilon \leftarrow \langle \text{val}\,\Upsilon_2 \rangle$

19a **SECTIONS and expressions**

The linker supports an extensive range of expressions. The precedence mechanism provided by `bison` is used to present the composition of expressions out of simpler chunks and basic building blocks tied together by algebraic operations.

⟨ Grammar rules 11a ⟩ + = △ 18e 19b ▽

exp :

$-$ *exp*	⟨`prec` unary⟩	$\Upsilon \leftarrow \langle \{\,\texttt{-val}\,\Upsilon_2\,\} \rangle$
(*exp*)		$\Upsilon \leftarrow \langle (\text{val}\,\Upsilon_2) \rangle$
`NEXT` (*exp*)	⟨`prec` unary⟩	$\Upsilon \leftarrow \langle \texttt{\textbackslash hbox}\{{}^{\text{nx}}\texttt{\textbackslash ssf next}\}(\text{val}\,\Upsilon_3) \rangle$
$\neg$ *exp*	⟨`prec` unary⟩	$\Upsilon \leftarrow \langle \{{}^{\text{nox}}\neg\text{val}\,\Upsilon_2\,\} \rangle$
$+$ *exp*	⟨`prec` unary⟩	$\Upsilon \leftarrow \langle \{\,\texttt{+val}\,\Upsilon_2\,\} \rangle$
$^{\text{not}}$ *exp*	⟨`prec` unary⟩	$\Upsilon \leftarrow \langle \{{}^{\text{nox not}}\text{val}\,\Upsilon_2\,\} \rangle$
exp $\times$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\times\text{val}\,\Upsilon_3 \rangle$
exp / *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}/\text{val}\,\Upsilon_3 \rangle$
exp $\div$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\div\text{val}\,\Upsilon_3 \rangle$
exp $+$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1\texttt{+}\text{val}\,\Upsilon_3 \rangle$
exp $-$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1\texttt{-}\text{val}\,\Upsilon_3 \rangle$
exp $\ll$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\ll\text{val}\,\Upsilon_3 \rangle$
exp $\gg$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\gg\text{val}\,\Upsilon_3 \rangle$
exp $=$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 = \text{val}\,\Upsilon_3 \rangle$
exp $\neq$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\texttt{\textbackslash not} = \text{val}\,\Upsilon_3 \rangle$
exp $\leqslant$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\leqslant\text{val}\,\Upsilon_3 \rangle$
exp $\geqslant$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\geqslant\text{val}\,\Upsilon_3 \rangle$
exp $<$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1\texttt{<}\text{val}\,\Upsilon_3 \rangle$
exp $>$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1\texttt{>}\text{val}\,\Upsilon_3 \rangle$
exp & *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\&\text{val}\,\Upsilon_3 \rangle$
exp $\oplus$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\oplus\text{val}\,\Upsilon_3 \rangle$
exp | *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\vert\text{val}\,\Upsilon_3 \rangle$
exp `?` *exp* `:` *exp*		⟨ Process a primitive conditional 20a ⟩
exp $\wedge$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\vee\text{val}\,\Upsilon_3 \rangle$
exp $\vee$ *exp*		$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1{}^{\text{nox}}\wedge\text{val}\,\Upsilon_3 \rangle$

19b More atomic expression types specific to the linker's function.

⟨ Grammar rules 11a ⟩ + = △ 19a 20c ▽

exp :

`DEFINED ( name )`	**def** `\ldfuncname{defined}`⟨ Apply a function to a name 20b ⟩
`INT`	⟨ Carry on 14f ⟩
`SIZEOF_HEADERS`	
`ALIGNOF ( name )`	**def** `\ldfuncname{alignof}`⟨ Apply a function to a name 20b ⟩
`SIZEOF ( name )`	**def** `\ldfuncname{sizeof}`⟨ Apply a function to a name 20b ⟩
`ADDR ( name )`	**def** `\ldfuncname{addr}`⟨ Apply a function to a name 20b ⟩

LOADADDR (name)	**def** \ldfuncname{loadaddr}⟨ Apply a function to a name 20b ⟩
CONSTANT (name)	**def** \ldfuncname{constant}⟨ Apply a function to a name 20b ⟩
ABSOLUTE (*exp*)	$\Upsilon \leftarrow \langle^{\mathrm{nx}}$\mathop{\hbox{$^{\mathrm{nx}}$\ssf absolute}}(val Υ_3)$\rangle$
ALIGN_K (*exp*)	$\Upsilon \leftarrow \langle^{\mathrm{nx}}$\mathop{\hbox{$^{\mathrm{nx}}$\ssf align}}(val Υ_3)$\rangle$
ALIGN_K (*exp* , *exp*)	
DATA_SEGMENT_ALIGN (*exp* , *exp*)	
DATA_SEGMENT_RELRO_END (*exp* , *exp*)	
DATA_SEGMENT_END (*exp*)	
SEGMENT_START (name , *exp*)	
BLOCK (*exp*)	$\Upsilon \leftarrow \langle^{\mathrm{nx}}$\mathop{\hbox{$^{\mathrm{nx}}$\ssf block}}(val Υ_3)$\rangle$
name	$\Upsilon \leftarrow \langle^{\mathrm{nx}}$\ldregexp{ val Υ_1 }$\rangle$
MAX_K (*exp* , *exp*)	
MIN_K (*exp* , *exp*)	
ASSERT_K (*exp* , name)	
ORIGIN (name)	**def** \ldfuncname{origin}⟨ Apply a function to a name 20b ⟩
LENGTH (name)	**def** \ldfuncname{length}⟨ Apply a function to a name 20b ⟩
LOG2CEIL (*exp*)	

20a ⟨ Process a primitive conditional 20a ⟩ =

$\Upsilon \leftarrow \langle$\hbox{$^{\mathrm{nx}}$\ttl do}$\xi$(val Υ_1)\hbox{ $^{\mathrm{nx}}$\ttl where}
{**let** $^{\mathrm{nx}}$\{$^{\mathrm{nx}}$\bigbracedel $^{\mathrm{nx}}\xi$(x) = $^{\mathrm{nx}}$\cases{ val Υ_5 if $^{\bullet}$(x = 0)\cr val Υ_3 if $^{\bullet}$(x$^{\mathrm{nx}}$\not = 0)}}$\rangle$

This code is used in section 19a.

20b ⟨ Apply a function to a name 20b ⟩ =

$\Upsilon \leftarrow \langle^{\mathrm{nx}}$\mathop{\hbox{$^{\mathrm{nx}}$\ssf \ldfuncname}}($^{\mathrm{nx}}$\ldregexp{ val Υ_3 }$\rangle$

This code is used in section 19b.

20c ⟨ Grammar rules 11a ⟩ + = △ 19b 20d ▽

memspec_at$_{\mathrm{opt\_}}$:	
AT > name	$\Upsilon \leftarrow \langle$val $\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
at$_{\mathrm{opt\_}}$:	
AT (*exp*)	$\Upsilon \leftarrow \langle$val $\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
align$_{\mathrm{opt\_}}$:	
ALIGN_K (*exp*)	$\Upsilon \leftarrow \langle$val $\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
align_with_input$_{\mathrm{opt\_}}$:	
ALIGN_WITH_INPUT	$\Upsilon \leftarrow \langle$align with input$\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
subalign$_{\mathrm{opt\_}}$:	
SUBALIGN (*exp*)	$\Upsilon \leftarrow \langle$val $\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
sect_constraint :	
ONLY_IF_RO	$\Upsilon \leftarrow \langle$only_if_ro$\rangle$
ONLY_IF_RW	$\Upsilon \leftarrow \langle$only_if_rw$\rangle$
SPECIAL	$\Upsilon \leftarrow \langle$special$\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

20d *The* GROUP *case is just enough to support the* gcc *svr3.ifile script. It is not intended to be full support. I'm not even sure what* GROUP *is supposed to mean.* A careful analysis of the productions below reveals some pitfalls in the parser/lexer interaction setup that uses the state switching macros (or functions in the case of the original parser). The switch to the EXPRESSION state at the end of the production for *section* is invoked before ,$_{\mathrm{opt\_}}$ which can be empty. This means that the next (lookahead) token (which could be a name in a different context) might be read before the lexer is in the appropriate state. In practice, the names of the

sections and other names are usually pretty straightforward so this parser idiosynchrasy is unlikely to lead to a genuine problem. Since the goal was to keep the original grammar intact as much as possible, it was decided to leave this production unchanged.

⟨ Grammar rules 11a ⟩ + = △ 20c 21c ▽

section :

name	`\ldlex@expression`
exp_with_type$_{\text{opt\_}}$ *at*$_{\text{opt\_}}$ *align*$_{\text{opt\_}}$ *align_with_input*$_{\text{opt\_}}$ ↩	
subalign$_{\text{opt\_}}$	`\ldlex@popstate\ldlex@script`
sect_constraint `{`	
statement_list$_{\text{opt}}$ `}`	`\ldlex@popstate\ldlex@expression`
memspec$_{\text{opt}}$ *memspec_at*$_{\text{opt}}$ *phdr*$_{\text{opt}}$ *fill*$_{\text{opt}}$	`\ldlex@popstate`
`,`$_{\text{opt\_}}$	⟨ Record a named section 21a ⟩
`OVERLAY`	`\ldlex@expression`
exp_without_type$_{\text{opt\_}}$ *nocrossrefs*$_{\text{opt\_}}$ *at*$_{\text{opt\_}}$ *subalign*$_{\text{opt\_}}$	`\ldlex@popstate\ldlex@script`
`{`	
overlay_section `}`	`\ldlex@popstate\ldlex@expression`
memspec$_{\text{opt}}$ *memspec_at*$_{\text{opt}}$ *phdr*$_{\text{opt}}$ *fill*$_{\text{opt}}$	`\ldlex@popstate`
`,`$_{\text{opt\_}}$	⟨ Record an overlay section 21b ⟩
`GROUP`	`\ldlex@expression`
exp_with_type$_{\text{opt\_}}$	`\ldlex@popstate`
`{` *sec_or_group_p*$_1$ `}`	
`INCLUDE` *filename*	⟨ Peek at a file 15a ⟩
sec_or_group_p$_1$ end	⟨ Close the file 15b ⟩

21a ⟨ Record a named section 21a ⟩ =

$\pi_3(\Upsilon_1) \mapsto v_a \pi_4(\Upsilon_1) \mapsto v_b$

$\pi_5(\Upsilon_{12}) \mapsto v_c$ ▷ *statement_list*$_{\text{opt}}$ contents ◁

$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{`val v_a`}{`val v_b`}{osect}{`$^{\text{nx}}$`\ldnamedsection{`val Υ_1`}{`val Υ_3`}{`val Υ_4`}`
`{{`val Υ_5`}{`val Υ_6`}{`val Υ_7`}}` ▷ alignment ◁
`{`val Υ_9`}{`val v_c`}`
`{{`val Υ_{15}`}{`val Υ_{16}`}{`val Υ_{17}`}{`val Υ_{18}`}}}`$\rangle$ ▷ memory specifiers ◁

This code is used in section 20d.

21b ⟨ Record an overlay section 21b ⟩ =

$\pi_1(\Upsilon_1) \mapsto v_a \pi_2(\Upsilon_1) \mapsto v_b$

$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{`val v_a`}{`val v_b`}{osect}{`$^{\text{nx}}$`\ldoverlay{`val Υ_3`}{`val Υ_4`}{`val Υ_5`}{`val Υ_6`}`
`{`val Υ_{10}`}` ▷ sections ◁
`{{`val Υ_{13}`}{`val Υ_{14}`}{`val Υ_{15}`}{`val Υ_{16}`}}}`$\rangle$ ▷ memory specifiers, etc. ◁

This code is used in section 20d.

21c ⟨ Grammar rules 11a ⟩ + = △ 20d 21d ▽

type :

`NOLOAD`	$\Upsilon \leftarrow \langle$`noload`$\rangle$
`DSECT`	$\Upsilon \leftarrow \langle$`dsect`$\rangle$
`COPY`	$\Upsilon \leftarrow \langle$`copy`$\rangle$
`INFO`	$\Upsilon \leftarrow \langle$`info`$\rangle$
`OVERLAY`	$\Upsilon \leftarrow \langle$`overlay`$\rangle$

atype :

`(` *type* `)`	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldtype{`val Υ_2`}`$\rangle$
`()`	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldtype{}`$\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

21d *The* `BIND` *cases are to support the* `gcc` *svr3.ifile script. They aren't intended to implement full support for the* `BIND` *keyword. I'm not even sure what* `BIND` *is supposed to mean.*

⟨ Grammar rules 11a ⟩ + = △ 21c 22c ▽

exp_with_type$_{\text{opt\_}}$:
exp atype : $\Upsilon \leftarrow \langle \{\ \}\{\ \mathrm{val}\ \Upsilon_1\ \}\{\ \}\{\ \}\{\ \mathrm{val}\ \Upsilon_2\ \}\rangle$
atype : $\Upsilon \leftarrow \langle \{\ \}\{\ \}\{\ \}\{\ \}\{\ \mathrm{val}\ \Upsilon_1\ \}\rangle$
`BIND (` *exp* `)` *atype* : $\Upsilon \leftarrow \langle \{\ \texttt{bind}\ \}\{\ \mathrm{val}\ \Upsilon_3\ \}\{\ \}\{\ \}\{\ \mathrm{val}\ \Upsilon_5\ \}\rangle$
`BIND (` *exp* `) BLOCK (` *exp* `)` *atype* : $\Upsilon \leftarrow \langle \{\ \texttt{bind}\ \}\{\ \mathrm{val}\ \Upsilon_3\ \}\{\ \texttt{block}\ \}\{\ \mathrm{val}\ \Upsilon_7\ \}\{\ \mathrm{val}\ \Upsilon_9\ \}\rangle$

exp_without_type$_{\text{opt\_}}$:
exp : $\Upsilon \leftarrow \langle \mathrm{val}\ \Upsilon_1 \rangle$
: $\Upsilon \leftarrow \langle\rangle$

nocrossrefs$_{\text{opt\_}}$:
○
`NOCROSSREFS`

memspec$_{\text{\_opt}}$:
`>` name $\Upsilon \leftarrow \langle \mathrm{val}\ \Upsilon_2 \rangle$
○ $\Upsilon \leftarrow \langle\rangle$

phdr$_{\text{\_opt}}$:
○ $\Upsilon \leftarrow \langle\rangle$
phdr$_{\text{\_opt}}$: name ⟨ Add another pheader 22a ⟩

overlay_section :
○ $\Upsilon \leftarrow \langle\rangle$
overlay_section name `\ldlex@script`
`{` *statement_list*$_{\text{opt}}$ `}` `\ldlex@popstate` `\ldlex@expression`
phdr$_{\text{opt}}$ *fill*$_{\text{opt}}$ `\ldlex@popstate`
`,`$_{\text{opt\_}}$ ⟨ Add a section to the overlay 22b ⟩

22a ⟨ Add another pheader 22a ⟩ =
`\yytoksempty{` Υ_1 `}{` $\Upsilon \leftarrow \langle \{\ \mathrm{val}\ \Upsilon_3\ \}\rangle$ `}{` $\Upsilon \leftarrow \langle \mathrm{val}\ \Upsilon_1\ ^{\mathrm{nox}}$`\ldor{` $\mathrm{val}\ \Upsilon_3\ \}\rangle$ `}`
This code is used in section 21d.

22b ⟨ Add a section to the overlay 22b ⟩ =
$\pi_3(\Upsilon_2) \mapsto v_a\, \pi_4(\Upsilon_2) \mapsto v_b$ ▷ *name* pointers ◁
$\pi_5(\Upsilon_5) \mapsto v_c$ ▷ *statement_list*$_{\text{opt}}$ contents ◁
`\yytoksempty{` Υ_1 `}{`
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\ldoverlaysection{` $\mathrm{val}\ \Upsilon_2$ `}{` $\mathrm{val}\ v_c$ `}{` $\mathrm{val}\ \Upsilon_8$ `}{` $\mathrm{val}\ \Upsilon_9\ \}\rangle$
`}{`
$\Upsilon \leftarrow \langle \mathrm{val}\ \Upsilon_1\ ^{\mathrm{nx}}$`\ldoverlaysection{` $\mathrm{val}\ \Upsilon_2$ `}{` $\mathrm{val}\ v_c$ `}{` $\mathrm{val}\ \Upsilon_8$ `}{` $\mathrm{val}\ \Upsilon_9\ \}\rangle$
`}`
This code is used in section 21d.

22c ⟨ Grammar rules 11a ⟩ + = △ 21d 23c ▽
phdrs :
`PHDRS {` *phdr_list* `}`

phdr_list :
○
phdr_list phdr

phdr :
name `\ldlex@expression`
phdr_type phdr_qualifiers `\ldlex@popstate`
`;`

phdr_type :
exp

phdr_qualifiers :
○
name *phdr_val phdr_qualifiers*
`AT (` *exp* `)` *phdr_qualifiers*

phdr_val :
○
`(` *exp* `)`

23a **Other types of script files**

At present time other script types are ignored, although some of the rules are used in linker scripts that are processed by the parser.

⟨ Dynamic list file rules 23a ⟩ =

dynamic_list_file :
○ `\ldlex@version@file`
dynamic_list_nodes `\ldlex@popstate`

dynamic_list_nodes :
dynamic_list_node
dynamic_list_nodes dynamic_list_node

dynamic_list_node :
`{` *dynamic_list_tag* `}` `;`

dynamic_list_tag :
vers_defns `;`

23b *This syntax is used within an external version script file.*

⟨ Version file rules 23b ⟩ =

version_script_file :
○ `\ldlex@version@file`
vers_nodes `\ldlex@popstate`

23c *This is used within a normal linker script file.*

⟨ Grammar rules 11a ⟩ + = △ 22c

version :
○ `\ldlex@version@script`
$\texttt{VERSION}_{\text{K}}$ `{` *vers_nodes* `}` `\ldlex@popstate`

vers_nodes :
vers_node
vers_nodes vers_node

vers_node :
`{` *vers_tag* `}` `;`
`VERS_TAG {` *vers_tag* `} ;`
`VERS_TAG {` *vers_tag* `}` *verdep* `;`

verdep :
`VERS_TAG`
verdep `VERS_TAG`

vers_tag :
○
vers_defns `;`
`GLOBAL :` *vers_defns* `;`
`LOCAL :` *vers_defns* `;`
`GLOBAL :` *vers_defns* `; LOCAL :` *vers_defns* `;`

vers_defns :
`VERS_IDENTIFIER`
name
vers_defns `; VERS_IDENTIFIER`
vers_defns `;` name
vers_defns `; EXTERN` name `{`
vers_defns $;_{\text{opt\_}}$ `}`

```
        EXTERN name {
            vers_defns ;opt_ }
        GLOBAL
        vers_defns ; GLOBAL
        LOCAL
        vers_defns ; LOCAL
        EXTERN
        vers_defns ; EXTERN
    ;opt_:  ○ | ;
```

2
The lexer

The lexer used by `ld` is almost straightforward. There are a few facilities (C header files, some output functions) needed by the lexer that are conviniently coded into the C code run by the driver routines that make the lexer more complex than it should have been but the function of each such facility can be easily clarified using this documentation and occasionally referring to the manual for the `bison` parser which is part of this distribution.

⟨ `ldl.ll` ch2 ⟩ =

```
⟨ ld lexer definitions 26b ⟩
.....
⟨ ld lexer C preamble 25b ⟩
.....
⟨ ld lexer options 25a ⟩

⟨ Regular expressions for ld tokens 28b ⟩

void define_all_states(void)
{
        ⟨ Collect state definitions for the ld lexer 26a ⟩
}
```

25a ⟨ `ld` lexer options 25a ⟩ =

```
⟨option⟩f                    bison-bridge
⟨option⟩f                    noyywrap
⟨option⟩f                    nounput
⟨option⟩f                    noinput
⟨option⟩f                    reentrant
⟨option⟩f                    noyy_top_state
⟨option⟩f                    debug
⟨option⟩f                    stack
⟨output to⟩f                 "ldl.c"
```

This code is used in section ch2.

25b ⟨ `ld` lexer C preamble 25b ⟩ =

This code is used in section ch2.

26a The file `ldl_states.h` below contains the names of all the start conditions (or states) collected by the bootstrap parser.

⟨ Collect state definitions for the `ld` lexer 26a ⟩ =

```
#define _register_name(name)  Define_State(#name, name)
#include "ldl_states.h"
#undef _register_name
```

This code is used in section ch2.

26b The character classes used by the scanner as well as lexer state declarations have been put in the definitions section of the input file. No attempt has been made to clean up the definitions of the character classes.

⟨ `ld` lexer definitions 26b ⟩ =

⟨ `ld` lexer states 26c ⟩

⟨CMDFILENAMECHAR⟩	[_a–zA–Z0–9/._+$:[]\,=&!<>-~]
⟨CMDFILENAMECHAR$_1$⟩	[_a–zA–Z0–9/._+$:[]\,=&!<>~]
⟨FILENAMECHAR$_1$⟩	[_a–zA–Z/.\$_~]
⟨SYMBOLCHARN⟩	[_a–zA–Z/.\$_~0–9]
⟨FILENAMECHAR⟩	[_a–zA–Z0–9/.-_+=$:[]\,~]
⟨WILDCHAR⟩	[_a–zA–Z0–9/.-_+=$:[]\,~?*^!]
⟨WHITE⟩	[␣⟨t⟩⟨n⟩⟨r⟩]$_+$
⟨NOCFILENAMECHAR⟩	[_a–zA–Z0–9/.-_+$:[]\~]
⟨V_TAG⟩	[.$_a–zA–Z][._a–zA–Z0–9]$_*$
⟨V_IDENTIFIER⟩	[*?.$_a–zA–Z[]-!^\]([*?.$_a–zA–Z0–9[]-!^\]|::)$_*$

This code is used in section ch2.

26c The lexer uses different sets of rules depending on the context and the current state. These can be changed from the lexer itself or externally by the parser (as is the case in `ld` implementation). Later, a number of helper macros implement state switching so that the state names are very rarely used explicitly. Keeping all the state declarations in the same section simplifies the job of the bootstrap parser, as well.

⟨ `ld` lexer states 26c ⟩ =

⟨**state-s**⟩$_f$ SCRIPT
⟨**state-s**⟩$_f$ EXPRESSION
⟨**state-s**⟩$_f$ BOTH
⟨**state-s**⟩$_f$ DEFSYMEXP
⟨**state-s**⟩$_f$ MRI
⟨**state-s**⟩$_f$ VERS_START
⟨**state-s**⟩$_f$ VERS_SCRIPT
⟨**state-s**⟩$_f$ VERS_NODE

This code is used in section 26b.

26d **Macros for lexer functions**

The state switching 'ping-pong' between the lexer and the parser aside, the `ld` lexer is very traditional. One implementation choice deserving some attention is the treatment of comments. The difficulty of implementing C style comment scanning using regular expressions is well-known so an often used alternative is a special function that simply skips to the end of the comment. This is exactly what the `ld` lexer does with an aptly named *comment*( ) function. The typesetting parser uses the `\ldcomment` macro for the same purpose. For the curious, here is a `flex` style regular expression defining C comments [1]):

```
"/*" ("/"|[^*/]|"*"+[^*/])* "*"+ "/"
```

This expression does not handle *every* practical situation, however, since it assumes that the end of line character can be matched like any other. Neither does it detect some often made mistakes such as attempting

[1]) Taken from W. McKeeman's site at http://www.cs.dartmouth.edu/~mckeeman/cs118/assignments/comment.html and adapted to `flex` syntax. Here is the same regular expression pretty printed by `SPLinT`: `/*`(/ | [*/]^c | *$_+$[*/]^c)$_*$ *$_+$/

to nest comments. A few minor modifications can fix this deficiency, as well as add some error handling, however, for the sake of consistency, the approach taken here mirrors the one in the original `ld`.

The top level of the `\ldcomment` macro simply bypasses the state setup of the lexer and enters a '**while** loop' in the input routine. This macro is a reasonable approximation of the functionality provided by *comment*( ).

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + ≡ △ 7c 27a ▽

```
\def\ldcomment{%
    \let\oldyyreturn\yyreturn
    \let\oldyylextail\yylextail
    \let\yylextail\yymatch        %   ▷ start inputting characters until */ is seen ◁
    \let\yyreturn\ldcommentskipchars
}
```

27a The rest of the **while** loop merely waits for the `*/` combination.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + ≡ △ 26d 27b ▽

```
\def\ldcommentskipchars{%
    \ifnum\yycp@=‘*
        \yybreak{\let\yyreturn\ldcommentseekslash\yyinput}%
                                  %   ▷ * found, look for / ◁
    \else
        \yybreak{\yyinput}%       %   ▷ keep skipping characters ◁
    \yycontinue
}%

\def\ldcommentseekslash{%
    \ifnum\yycp@=‘/
        \yybreak{\ldcommentfinish}%   ▷ / found, exit ◁
    \else
        \ifnum\yycp@=‘*
            \yybreak@{\yyinput}%  %   ▷ keep skipping *'s looking for a / ◁
        \else
            \yybreak@{\let\yyreturn\ldcommentskipchars\yyinput}%
                                  %   ▷ found a character other than * or / ◁
        \fi
    \yycontinue
}%
```

27b Once the end of the comment has been found, resume lexing the input stream.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + ≡ △ 27a 27c ▽

```
\def\ldcommentfinish{%
    \let\yyreturn\oldyyreturn
    \let\yylextail\oldyylextail
    \yylextail
}
```

27c The semantics of the macros defined above do not quite match that of the *comment*( ) function. The most significant difference is that the portion of the action following `\ldcomment` expands *before* the comment characters are skipped. In most applications, *comment*( ) is the last function called so this would not limit the use of `\ldcomment` too dramatically.

A more intuitive and easier to use version of `\ldcomment` is possible, however, if `\yylextail` is not used inside actions (in the case of an 'optimized' lexer the restriction is even weaker, namely, `\yylextail` merely has to be absent in the portion of the action following `\ldcomment`).

Another remark might be in order. It would seem more appropriate to employ TEX's native grouping mechanism to avoid the side effects casued by the assignments performed by the macros (such as `\let\oldyyreturn\yyreturn`). While this is possible with some careful macro writing, a naive grouping

attempt would interfere with the assignments performed by `\yymatch` (e.g. `\yyresetstreams`). Avoiding assignments like these is still possible although the effort required is bordering on excessive.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + = 27b 28a

```
\def\ldcomment#1\yylextail{%
    \let\oldyyreturn\yyreturn
    \def\yylexcontinuation{#1\yylextail}%
    \let\yyreturn\ldcommentskipchars %    ▷ start inputting characters until */ is seen ◁
    \yymatch
}

\def\ldcommentfinish{%
    \let\yyreturn\oldyyreturn
    \yylexcontinuation
}
```

28a The same idea can be applied to 'pretend buffer switching'. Whenever the 'real' `ld` parser encounters an `INCLUDE` command, it switches the input buffer for the lexer and waits for the lexer to return the tokens from the file it just opened. When the lexer scans the end of the included file, it returns a special token, end that completes the appropriate production and lets the parser continue with its job.

We would like to simulate the file inclusion by inserting the appropriate end of file marker for the lexer (a double `\yyeof`). After the relevant production completes, the marker has to be cleaned up from the input stream (the lexer is designed to leave it intact to be able to read the end of file multiple times while looking for the longest match).

The macro below is designed to handle this task. The idea is to replace the double `\yyeof` at the beginning of the input with an appropriate lexer action. The `\yyreadinput` handles the input buffer and inserts the tail portion of the current `flex` action in front of it.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + = 27c 33a

```
\def\ldcleanyyeof#1\yylextail{%
    \yyreadinput{\ldcl@anyyeof{#1\yylextail}}{\romannumeral0\yyr@@dinput}%
}

\def\ldcl@anyyeof#1#2#3{%
    #3\ldcl@anyye@f{#1}#2%
}

\def\ldcl@anyye@f#1#2\yyeof\yyeof{#1}
```

28b **Regular expressions**

The 'heart' of any lexer is the collection of regular expressions that describe the *tokens* of the appropriate language. The variey of tokens recognized by `ld` is quite extensive and is described in the sections that follow.

Variable names, constants, and algebraic operations come first.

⟨ Regular expressions for `ld` tokens 28b ⟩ = 29a

BOTH SCRIPT EXPRESSION VERS_START VERS_NODE VERS_SCRIPT^{++}	
`/*`	`\ldcomment` **continue**
DEFSYMEXP^{++}	
`-`	**return**$_c$
`+`	**return**$_c$
⟨FILENAMECHAR$_1$⟩ ⟨SYMBOLCHARN⟩$_*$	**return**$_{vp}$ name
`=`	**return**$_c$
MRI EXPRESSION^{++}	
`$` ([0–9A–Fa–f])$_+$	⟨ Return an absolute hex constant 31a ⟩
([0–9A–Fa–f])$_+$(H | h | X | x | B | b | O | o | D | d)	⟨ Return a constant in a specific radix 31b ⟩

SCRIPT DEFSYMEXP MRI BOTH EXPRESSION^{++}

`((($ | 0[xX]) ([0–9A–Fa–f])+) | (([0–9])+)) (M | K | m | k)?`	⟨ Return a constant with a multiplier 31c ⟩

BOTH SCRIPT EXPRESSION MRI^{++}

`<<=`	$\mathbf{return}_p \overset{\ll}{\Leftarrow}$
`>>=`	$\mathbf{return}_p \overset{\gg}{\Leftarrow}$
`||`	$\mathbf{return}_p \vee$
`==`	$\mathbf{return}_p =$
`!=`	$\mathbf{return}_p \neq$
`>=`	$\mathbf{return}_p \geqslant$
`<=`	$\mathbf{return}_p \leqslant$
`<<`	$\mathbf{return}_p \ll$
`>>`	$\mathbf{return}_p \gg$
`+=`	$\mathbf{return}_p \overset{+}{\Leftarrow}$
`-=`	$\mathbf{return}_p \overset{-}{\Leftarrow}$
`*=`	$\mathbf{return}_p \overset{\times}{\Leftarrow}$
`/=`	$\mathbf{return}_p \overset{\div}{\Leftarrow}$
`&=`	$\mathbf{return}_p \overset{\&}{\Leftarrow}$
`|=`	$\mathbf{return}_p \overset{\vee}{\Leftarrow}$
`&&`	$\mathbf{return}_p \wedge$
`[&|~!?*+-/%=<>{}()[]:;,]`	$\mathbf{return}_c$

See also sections 29a, 32a, 32c, and 33b.

This code is used in section ch2.

29a The bulk of tokens produced by the lexer are the keywords used inside script files. File name syntax is listed as well, along with the miscellanea like whitespace and version symbols.

⟨ Regular expressions for `ld` tokens 28b ⟩ + = △ 28b 32a ▽

BOTH SCRIPT^{++}

`MEMORY`	$\mathbf{return}_p$ `MEMORY`
`REGION_ALIAS`	$\mathbf{return}_p$ `REGION_ALIAS`
`LD_FEATURE`	$\mathbf{return}_p$ `LD_FEATURE`
`VERSION`	$\mathbf{return}_p$ `VERSION`$_{\mathrm{K}}$
`TARGET`	$\mathbf{return}_p$ `TARGET`$_{\_\mathrm{K}}$
`SEARCH_DIR`	$\mathbf{return}_p$ `SEARCH_DIR`
`OUTPUT`	$\mathbf{return}_p$ `OUTPUT`
`INPUT`	$\mathbf{return}_p$ `INPUT`
`ENTRY`	$\mathbf{return}_p$ `ENTRY`
`MAP`	$\mathbf{return}_p$ `MAP`
`CREATE_OBJECT_SYMBOLS`	$\mathbf{return}_p$ `CREATE_OBJECT_SYMBOLS`
`CONSTRUCTORS`	$\mathbf{return}_p$ `CONSTRUCTORS`
`FORCE_COMMON_ALLOCATION`	$\mathbf{return}_p$ `FORCE_COMMON_ALLOCATION`
`INHIBIT_COMMON_ALLOCATION`	$\mathbf{return}_p$ `INHIBIT_COMMON_ALLOCATION`
`SECTIONS`	$\mathbf{return}_p$ `SECTIONS`
`INSERT`	$\mathbf{return}_p$ `INSERT`$_{\_\mathrm{K}}$
`AFTER`	$\mathbf{return}_p$ `AFTER`
`BEFORE`	$\mathbf{return}_p$ `BEFORE`
`FILL`	$\mathbf{return}_p$ `FILL`
`STARTUP`	$\mathbf{return}_p$ `STARTUP`
`OUTPUT_FORMAT`	$\mathbf{return}_p$ `OUTPUT_FORMAT`
`OUTPUT_ARCH`	$\mathbf{return}_p$ `OUTPUT_ARCH`
`HLL`	$\mathbf{return}_p$ `HLL`
`SYSLIB`	$\mathbf{return}_p$ `SYSLIB`
`FLOAT`	$\mathbf{return}_p$ `FLOAT`
`QUAD`	$\mathbf{return}_p$ `QUAD`
`SQUAD`	$\mathbf{return}_p$ `SQUAD`

LONG	**return**$_p$ LONG
SHORT	**return**$_p$ SHORT
BYTE	**return**$_p$ BYTE
NOFLOAT	**return**$_p$ NOFLOAT
OVERLAY	**return**$_p$ OVERLAY
SORT_BY_NAME	**return**$_p$ SORT_BY_NAME
SORT_BY_ALIGNMENT	**return**$_p$ SORT_BY_ALIGNMENT
SORT	**return**$_p$ SORT_BY_NAME
SORT_BY_INIT_PRIORITY	**return**$_p$ SORT_BY_INIT_PRIORITY
SORT_NONE	**return**$_p$ SORT_NONE
EXTERN	**return**$_p$ EXTERN
o | org	**return**$_p$ ORIGIN
l | len	**return**$_p$ LENGTH
PHDRS	**return**$_p$ PHDRS

EXPRESSION BOTH SCRIPT^{++}

BLOCK	**return**$_p$ BLOCK
BIND	**return**$_p$ BIND
LENGTH	**return**$_p$ LENGTH
ORIGIN	**return**$_p$ ORIGIN
ALIGN	**return**$_p$ ALIGN$_{_K}$
DATA_SEGMENT_ALIGN	**return**$_p$ DATA_SEGMENT_ALIGN
DATA_SEGMENT_RELRO_END	**return**$_p$ DATA_SEGMENT_RELRO_END
DATA_SEGMENT_END	**return**$_p$ DATA_SEGMENT_END
ADDR	**return**$_p$ ADDR
LOADADDR	**return**$_p$ LOADADDR
ALIGNOF	**return**$_p$ ALIGNOF
ASSERT	**return**$_p$ ASSERT$_{_K}$
NEXT	**return**$_p$ NEXT
sizeof_headers	**return**$_p$ SIZEOF_HEADERS
SIZEOF_HEADERS	**return**$_p$ SIZEOF_HEADERS
SEGMENT_START	**return**$_p$ SEGMENT_START
SIZEOF	**return**$_p$ SIZEOF
GROUP	**return**$_p$ GROUP
AS_NEEDED	**return**$_p$ AS_NEEDED
DEFINED	**return**$_p$ DEFINED
NOCROSSREFS	**return**$_p$ NOCROSSREFS
NOLOAD	**return**$_p$ NOLOAD
DSECT	**return**$_p$ DSECT
COPY	**return**$_p$ COPY
INFO	**return**$_p$ INFO
OVERLAY	**return**$_p$ OVERLAY
ONLY_IF_RO	**return**$_p$ ONLY_IF_RO
ONLY_IF_RW	**return**$_p$ ONLY_IF_RW
SPECIAL	**return**$_p$ SPECIAL
INPUT_SECTION_FLAGS	**return**$_p$ INPUT_SECTION_FLAGS
INCLUDE	**return**$_p$ INCLUDE
AT	**return**$_p$ AT
ALIGN_WITH_INPUT	**return**$_p$ ALIGN_WITH_INPUT
SUBALIGN	**return**$_p$ SUBALIGN
HIDDEN	**return**$_p$ HIDDEN
PROVIDE	**return**$_p$ PROVIDE
PROVIDE_HIDDEN	**return**$_p$ PROVIDE_HIDDEN
KEEP	**return**$_p$ KEEP
EXCLUDE_FILE	**return**$_p$ EXCLUDE_FILE
CONSTANT	**return**$_p$ CONSTANT
⟨n⟩	**continue**

EXPRESSION BOTH^{++}

MAX	**return**$_p$ MAX_K
MIN	**return**$_p$ MIN_K
LOG2CEIL	**return**$_p$ LOG2CEIL

EXPRESSION BOTH SCRIPT MRI^{++}

ABSOLUTE | absolute	**return**$_p$ ABSOLUTE
$[\sqcup\langle t\rangle\langle r\rangle]_+$	**continue**

BOTH^{++}

$\langle$FILENAMECHAR$_1\rangle$ $\langle$FILENAMECHAR$\rangle_*$	**return**$_{vp}$ name
-l $\langle$FILENAMECHAR$\rangle_+$	**return**$_{vp}$ name

EXPRESSION^{++}

$\langle$FILENAMECHAR$_1\rangle$ $\langle$NOCFILENAMECHAR$\rangle_*$	**return**$_{vp}$ name
-l $\langle$NOCFILENAMECHAR$\rangle_+$	**return**$_{vp}$ name

SCRIPT^{+}

$\langle$WILDCHAR$\rangle_*$	⟨ Skip a possible comment and return a name 31d ⟩

EXPRESSION BOTH SCRIPT VERS_NODE^{+}

" $[^{"}]^c{}_*$ "	⟨ Return the name inside quotes 31e ⟩

31a There is a bit of a trick to returning an absolute hex value. The macros are looking for a $ suffix while the contents of \yytext start with \$.

⟨ Return an absolute hex constant 31a ⟩ =

```
def_x next {\yylval{^nx\hexint{$\expandafter\eatone val\yytext}
{val\yyfmark}{val\yysmark}}}next
return_l INT
```

This code is used in section 28b.

31b ⟨ Return a constant in a specific radix 31b ⟩ =

```
def_x next {\yylval{^nx\bint{val\yytext}
{val\yyfmark}{val\yysmark}}}next
return_l INT
```

This code is used in section 28b.

31c ⟨ Return a constant with a multiplier 31c ⟩ =

```
def_x next {\yylval{^nx\anint{val\yytext}
{val\yyfmark}{val\yysmark}}}next
return_l INT
```

This code is used in section 28b.

31d *Annoyingly, this pattern can match comments, and we have longest match issues to consider. So if the first two characters are a comment opening, put the input back and try again.*

⟨ Skip a possible comment and return a name 31d ⟩ =

```
\matchcomment\yytextpure
    {\yyless 2_R \ldcomment}    ▷ matched the beginning of a comment ◁
    {return_vp name}
```

This code is used in section 29a.

31e *No matter the state, quotes give what's inside.*

⟨ Return the name inside quotes 31e ⟩ =

```
\ldstripquotes return_vp name
```

This code is used in section 29a.

32a Some syntax specific to version scripts.

⟨ Regular expressions for `ld` tokens 28b ⟩ + = △ 29a 32c ▽

```
VERS_SCRIPT++
  {                                  enter(VERS_NODE) \versnodenesting = 0R return_c
  }                                  return_c
  ⟨V_TAG⟩                            return_v VERS_TAG
VERS_NODE++
  global                             return_p GLOBAL
  local                              return_p LOCAL
  extern                             return_p EXTERN
  ⟨V_IDENTIFIER⟩                     return_v VERS_IDENTIFIER
  {                                  add\versnodenesting 1R  return_c
  }                                  ⟨ Finish the current group, possibly switch to VERS_SCRIPT 32b ⟩
VERS_NODE VERS_SCRIPT+
  [:,;]                              return_c
VERS_START+
  {                                  enter(VERS_SCRIPT) return_c
VERS_START VERS_NODE VERS_SCRIPT++
  [⟨n⟩]                              continue
  #.*                                continue
  [␣⟨t⟩⟨r⟩]+                         continue
```

32b ⟨ Finish the current group, possibly switch to VERS_SCRIPT 32b ⟩ =

```
add\versnodenesting −1R
if_ω \versnodenesting < 0R
    enter(VERS_SCRIPT)
fi
return_c
```

This code is used in section 32a.

32c Some syntax specific to MRI scripts.

⟨ Regular expressions for `ld` tokens 28b ⟩ + = △ 32a 33b ▽

```
MRI++
  #.* ⟨n⟩?                           continue
  ⟨n⟩                                return_p NEWLINE
  *.*                                continue
  ;.*                                continue
  END | end                          return_p ENDWORD
  ALIGNMOD | alignmod                return_p ALIGNMOD
  ALIGN | align                      return_p ALIGN_K
  CHIP | chip                        return_p CHIP
  BASE | base                        return_p BASE
  ALIAS | alias                      return_p ALIAS
  TRUNCATE | truncate                return_p TRUNCATE
  LOAD | load                        return_p LOAD
  PUBLIC | public                    return_p PUBLIC
  ORDER | order                      return_p ORDER
  NAME | name                        return_p NAMEWORD
  FORMAT | format                    return_p FORMAT
  CASE | case                        return_p CASE
  START | start                      return_p START
  (LIST | list).*                    return_p LIST
  SECT | sect                        return_p SECT
  EXTERN | extern                    return_p EXTERN
  ⟨FILENAMECHAR1⟩ ⟨NOCFILENAMECHAR⟩* return_vp name
```

33a The macros and the register definitions used to implement the actions above have been collected in this section.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + = △ 28a 33d ▽

```
\newcount\versnodenesting
\newcount\includestackptr

\def\matchcomment#1{%
    \expandafter\matchcomment@\the#1/*\yyeof
}

\def\matchcomment@#1/*#2\yyeof#3#4{%
    \yystringempty{#1}{#3}{#4}%
}

\def\ldstripquotes#1{%
    \yytext\expandafter\expandafter\expandafter
           {\expandafter\ldstripquotes@\the\yytext\yyeof}%
    \yytextpure\expandafter\expandafter\expandafter
           {\expandafter\ldstripquotes@\the\yytextpure\yyeof}%
}

\def\ldstripquotes@"#1"\yyeof{#1}
```

33b Catchall actions. These react to unexpected characters with (somewhat misleading) error messages. The placement of these rules is important, since they may match a token of the same length defined earlier.

⟨ Regular expressions for `ld` tokens 28b ⟩ + = △ 32c

```
SCRIPT MRI VERS_START VERS_SCRIPT VERS_NODE+
  .                                   fatal⟨bad character ‘val\yytext’ in script⟩

EXPRESSION DEFSYMEXP BOTH+
  .                                   fatal⟨bad character ‘val\yytext’ in expression⟩
⟨EOF⟩                                 ⟨ Process the end of (possibly included) file 33c ⟩
```

33c ⟨ Process the end of (possibly included) file 33c ⟩ =

```
add\includestackptr −1R
if_ω \includestackptr = 0R
    \yybreak { \yyterminate }
else
    \yybreak { \ldcleanyyeof return_l end }
\yycontinue
```

This code is used in section 33b.

33d **Parser-lexer interaction support**

Here are the long promised auxiliary macros for switching lexer states and handling file input.

⟨ Additional macros for the `ld` lexer/parser 7c ⟩ + = △ 33a

```
\def\ldlex@script{\yypushstate{SCRIPT}}
\def\ldlex@mri@script{\yypushstate{MRI}}
\def\ldlex@version@script{\yypushstate{VERS_START}}
\def\ldlex@version@file{\yypushstate{VERS_SCRIPT}}
\def\ldlex@defsym{\yypushstate{DEFSYMEXP}}
\def\ldlex@expression{\yypushstate{EXPRESSION}}
\def\ldlex@both{\yypushstate{BOTH}}
\let\ldlex@popstate\yypopstate

\def\ldfile@open@command@file#1{%
    \advance\includestackptr\@ne
```

```
    \appendlnx\yytext@seen{\yyeof\yyeof}%
    \yytextbackuptrue
}

\def\ldlex@filename{}
```

3
Example output

Here is an example output of the `ld` parser designed in this document. The original linker script is presented in the section that follows. The same parser can be used to present examples of `ld` scripts in text similar to the one below.

memory

	attributes	**starts at**	**length**
`RAM`	`xrw`	$2000\,0000_{16}$	$20 \cdot 2^{10}$
`FLASH`	`rx`	$800\,0000_{16}$	$128 \cdot 2^{10}$
`ASH`	`rx`	$8\,000\,000$	$128 \cdot 2^{10}$
`CLASH`	`rx`	$700\,000$	$128 \cdot 2^{10}$
`ASH`	`rx`	$800\,0000_{16}$	$128 \cdot 2^{10}$
`CLASH`	`rx`	$70\,0000_{01}$	$128 \cdot 2^{10}$

include `file.mem`

The syntax of `ld` is modular enough so there does not seem to be a need for a 'parser stack' as in the case of the `bison` parser. If one must be able to display still smaller segments of `ld` code, using 'hidden context' tricks (discussed elsewhere) seems to be a better approach.

⟨ Example `ld` script ch3 ⟩ =

include `file.ld`

memory

	attributes	**starts at**	**length**
⟨ Some random portion of `ld` code 38a ⟩			
`RAM`	`xrw`	$2000\,0000_{16}$	$20 \cdot 2^{10}$
`FLASH`	`rx¬w`	$800\,0000_{16}$	$128 \cdot 2^{10}$
`ASH`	`rx`	$8\,001\,000$	$128 \cdot 2^{10}$
⟨ Some random portion of `ld` code 38a ⟩			
`CLASH`	`rx`	$700\,000$	$128 \cdot 2^{10}$
`ASH`	`rx`	$800\,0000_{16}$	$128 \cdot 2^{10}$
`CLASH`	`rx`	$70\,0000_{01}$	$128 \cdot 2^{10}$

include `file.mem`

⟨ Some random portion of `ld` code 38a ⟩

$\_estack \Leftarrow 2000\,5000_{16}$

$\_bstack \Leftarrow$ **do** $\xi(\mathtt{a} > 0)$ **where** $\xi(x) = \begin{cases} 19_{16} & \text{if } x = 0 \\ \mathsf{next}(11) & \text{if } x \neq 0 \end{cases}$

⟨ Some random portion of `ld` code 38a ⟩

provide ⟨ $var_1 \Leftarrow \cdot\cdot$ ⟩

provide$_h$ ⟨ $var_2 \Leftarrow \cdot\cdot$ ⟩

⟨ Some random portion of `ld` code 38a ⟩

hidden ⟨ $var_3 \Leftarrow ..$ ⟩
entry: *_entry*

sections ⟨ Some random portion of `ld` code 38a ⟩
.. ⇐ origin(FLASH)

`.isr_vector` align(8)[**noload**]	**at** ..	**special**	**phdrs**
.. ⇐ align(4)	**align** ..	**in** FLASH **as** RAM	FLASH
keep(*(.isr_vector))	**align_with_input**		RAM
.. ⇐ align(4)	**subalign** 8		OTHER
fill .. + 8			

⟨ Some random portion of `ld` code 38a ⟩

`.text`		**in** FLASH **as** RAM
.. ⇐ align(4)		
*(.text)		
(.text.)		
*(.rodata)		
(.rodata)		
*(.glue_7)		
*(.glue_7t)		
.. ⇐ align(4)		
etext ⇐ .. + 8		
sidata ⇐ *etext*		
provide ⟨ $var_1 \Leftarrow ..$ ⟩		
provide$_h$ ⟨ $var_2 \Leftarrow ..$ ⟩		
hidden ⟨ $var_3 \Leftarrow ..$ ⟩		

⟨ Some random portion of `ld` code 38a ⟩

`.data`	**at** *_sidata*	**in** RAM
.. ⇐ align(4)		
sdata ⇐ ..		
*(.data)		
(.data.)		
.. ⇐ align(4)		
edata ⇐ ..		
`.bss`		**in** RAM
.. ⇐ align(4)		
sbss ⇐ ..		
*(.bss)		
*(COMMON)		
.. ⇐ align(4)		
ebss ⇐ ..		

⟨ Some random portion of `ld` code 38a ⟩
.. ⇐ align(0001 ABCD$_{16}$)
ffbegin ⇐ ..

overlay noxrefs	**at** *_ffabs*	**in** RAM
`.free_func0`		
*(.free_func0)		
.. ⇐ align(4)		
`.free_func1`		
*(.free_func1)		
overlay end		

36a ⟨ The same example of an `ld` script 36a ⟩ =

```
INCLUDE file.ld

MEMORY
{
 ⟨ Some random portion of ld code 38a ⟩
  RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K
```

```
  FLASH (rx!w) : ORIGIN = 0x8000000, LENGTH = 128K
  ASH (rx) : ORIGIN = 8001000, LENGTH = 128K
⟨ Some random portion of ld code 38a ⟩
  CLASH (rx) : ORIGIN = 700000, LENGTH = 128K
  ASH (rx) : ORIGIN = $8000000, LENGTH = 128K
  CLASH (rx) : ORIGIN = 700000B, LENGTH = 128K
  INCLUDE file.mem
⟨ Some random portion of ld code 38a ⟩

}

_estack = 0x20005000;
_bstack = a > 0 ? NEXT(11) : 0x19;
⟨ Some random portion of ld code 38a ⟩
PROVIDE( var1 = . );
PROVIDE_HIDDEN( var2 = . );
⟨ Some random portion of ld code 38a ⟩
HIDDEN( var3 = . );
ENTRY(_entry);

SECTIONS
{
⟨ Some random portion of ld code 38a ⟩
   . = ORIGIN(FLASH);
   .isr_vector ALIGN(8) (NOLOAD): AT(.) ALIGN(.) ALIGN_WITH_INPUT SUBALIGN(8) SPECIAL
   {
       . = ALIGN(4);
       KEEP(*(.isr_vector))
       . = ALIGN(4);
   } > FLASH AT > RAM : FLASH : RAM : OTHER = . + 8
⟨ Some random portion of ld code 38a ⟩
   .text :
   {
       /* skip this comment */;
       . = ALIGN(4);
       *(.text)
       *(.text.*)
       *(.rodata)
       *(.rodata*)
       *(.glue_7)
       *(.glue_7t)
       . = ALIGN(4);
  _etext = . + 8;
  _sidata = _etext;
       PROVIDE( var1 = . );
       PROVIDE_HIDDEN( var2 = . );
       HIDDEN( var3 = . );
   } >FLASH AT > RAM

⟨ Some random portion of ld code 38a ⟩
   .data  : AT ( _sidata )
   {
    . = ALIGN(4);
       _sdata = . ;
       *(.data)
       *(.data.*)
    . = ALIGN(4);
```

```
    _edata = . ;
    } >RAM

    .bss :
    {
     . = ALIGN(4);
        _sbss = .;
        *(.bss)
        *(COMMON)
     . = ALIGN(4);
    _ebss = . ;
    } >RAM
  ⟨ Some random portion of ld code 38a ⟩
    . = ALIGN(0x0001ABCD);
    _ffbegin = .;
    OVERLAY : NOCROSSREFS AT ( _ffabs ) {
       .free_func0
       {
            *(.free_func0)
     . = ALIGN(4);
       }

       .free_func1
       {
            *(.free_func1)
       }
    } >RAM
}
```

38a ⟨ Some random portion of ld code 38a ⟩ =

This code is used in sections ch3 and 36a.

4

The name parser for `ld` term names

We take a lazy approach to the typeseting of term names for the `ld` grammar by creating a dedicated parser for name processing. This way any pattern we notice can be quickly incorporated into our typesetting scheme.

⟨ `ld_small_parser.yy` ch4 ⟩ =

```
⟨ Name parser C preamble 43c ⟩
⟨ Bison options 40a ⟩
⟨ union ⟩      ⟨ Union of parser types 43e ⟩
⟨ Name parser C postamble 43d ⟩
⟨ Token and types declarations 40b ⟩

⟨ Parser productions 40c ⟩
```

39a To put the new name parser to work, we need to initialize it. The initialization is done by the macros below. After the initialization has been completed, the switch command is replaced by the one that activates the new name parser.

⟨ Modified name parser for `ld` grammar 39a ⟩ =

```
\genericparser
    name: ldsmall,
    ptables: ld_small_tab.tex,
    ltables: ld_small_dfa.tex,
    tokens: {},
    asetup: {},
    dsetup: {},
    rsetup: \let\returnexplicitspace\ignoreexplicitspace, % ignore spaces in names
    optimization: {};%
\let\otosmallparser\tosmallparser %    ▷ save the old name parser ◁
\let\tosmallparser\toldsmallparser
\expandafter\let\csname to\stripbrackets\cwebclinknamespace parser\endcsname\tosmallparser %
    ▷ make the name parser handle the typesetting of C variables ◁
```

This code is used in section 5a.

40a ⟨ Bison options 40a ⟩ =
⟨**token table**⟩ ⋆
⟨**parse.trace**⟩ ⋆ (set as ⟨**debug**⟩)
⟨**start**⟩ *full_name*
This code is used in section ch4.

40b ⟨ Token and types declarations 40b ⟩ =

`%[a...Z0...9]*`	`[a...Z0...9]*`	`opt`	$\texttt{suffix}_{\texttt{K\_}}$
`[0...9]*`	`ext`	`*` or `?`	«meta identifier»

This code is used in section ch4.

40c **The name parser productions**

These macros do a bit more than we need to typeset the term names. Their core is designed to treat suffixes and prefixes of a certain form in a special way. In addition, some productions were left in place from the original name parser in order to be able to refer to, say, `flex` options in text. The inline action in one of the rules for *identifier_string* was added to adjust the number and the position of the terms so that the appropriate action can be reused later for *qualified_identifier_string*.

⟨ Parser productions 40c ⟩ =

full_name :	
identifier_string $\mathit{suffixes}_{\text{opt}}$	⟨ Compose the full name 41a ⟩
qualifier `_` *identifier_string* $\mathit{suffixes}_{\text{opt}}$	⟨ Compose a qualified name 41b ⟩
«meta identifier»	⟨ Turn a «meta identifier» into a full name 41c ⟩
`'`	⟨ Make ' into a name 41d ⟩
identifier_string :	
`%[a...Z0...9]*`	⟨ Attach option name 41e ⟩
`[a...Z0...9]*`	⟨ Start with an identifier 41f ⟩
`' * or ? '`	⟨ Start with a quoted string 41g ⟩
`' _ '`	⟨ Start with a _ string 41h ⟩
`' . '`	⟨ Start with a . string 41i ⟩
incomplete_identifier_string ◇ `[a...Z0...9]*`	... | ⟨ Attach an identifier 42a ⟩
incomplete_identifier_string :	
`_`	$\Upsilon \leftarrow \langle {}^{\text{nx}}\texttt{\textbackslash idstr\{\}\{\}} \rangle$
identifier_string `_`	$\Upsilon \leftarrow \langle \text{val}\, \Upsilon_1 \rangle$
qualified_identifier_string `_`	$\Upsilon \leftarrow \langle \text{val}\, \Upsilon_1 \rangle$
qualified_identifier_string :	
identifier_string `_` *qualifier*	⟨ Attach qualifier to a name 42b ⟩
qualified_identifier_string `_` *qualifier*	⟨ Attach qualifier to a name 42b ⟩
$\mathit{suffixes}_{\text{opt}}$:	
∘	$\Upsilon \leftarrow \langle\rangle$
`.`	$\Upsilon \leftarrow \langle {}^{\text{nx}}\texttt{\textbackslash dotsp}\, {}^{\text{nx}}\texttt{\textbackslash sfxnone} \rangle$
`.` *suffixes*	⟨ Attach suffixes 42f ⟩
`.` *qualified_suffixes*	⟨ Attach qualified suffixes 42g ⟩
`[0...9]*`	⟨ Attach an integer 42c ⟩
`_ [0...9]*`	⟨ Attach a subscripted integer 42d ⟩
`_` *qualifier*	⟨ Attach a subscripted qualifier 42e ⟩
suffixes :	
`[a...Z0...9]*`	⟨ Start with a named suffix 42h ⟩
`[0...9]*`	⟨ Start with a numeric suffix 42i ⟩
suffixes `.`	⟨ Add a dot separator 42j ⟩
suffixes `[a...Z0...9]*`	⟨ Attach a named suffix 42l ⟩
suffixes `[0...9]*`	⟨ Attach integer suffix 42k ⟩
qualifier `.`	$\Upsilon \leftarrow \langle {}^{\text{nx}}\texttt{\textbackslash sfxn}\, \text{val}\, \Upsilon_1\, {}^{\text{nx}}\texttt{\textbackslash dotsp} \rangle$
suffixes qualifier `.`	$\Upsilon \leftarrow \langle \text{val}\, \Upsilon_1\, {}^{\text{nx}}\texttt{\textbackslash sfxn}\, \text{val}\, \Upsilon_2\, {}^{\text{nx}}\texttt{\textbackslash dotsp} \rangle$
qualified_suffixes :	
suffixes qualifier	⟨ Attach a qualifier 43a ⟩

qualifier ⟨ Start suffixes with a qualifier 43b ⟩

qualifier :

`opt`	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \rangle$
`suffix`$_{\mathrm{K\_}}$	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \rangle$
`ext`	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \rangle$

This code is used in section ch4.

41a ⟨ Compose the full name 41a ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \mathrm{val}\,\Upsilon_2 \rangle$ `\namechars` Υ

This code is used in section 40c.

41b ⟨ Compose a qualified name 41b ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a \pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3 \mathrm{val}\,\Upsilon_4$ `nx\dotsp` `nx\qual{` $\mathrm{val}\,v_a$ `nx\_}{` $\mathrm{val}\,v_b$ `\uscoreletter}` $\rangle$ `\namechars` Υ

This code is used in section 40c.

41c ⟨ Turn a «meta identifier» into a full name 41c ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle$ `nx\idstr{` $\mathrm{val}\,v_a$ `}{` $\mathrm{val}\,v_b$ `}` $\rangle$ `\namechars` Υ

This code is used in section 40c.

41d ⟨ Make ’ into a name 41d ⟩ =
$\Upsilon \leftarrow \langle$ `nx\chstr{'}{'}` $\rangle$ `\namechars` Υ

This code is used in section 40c.

41e ⟨ Attach option name 41e ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle$ `nx\optstr{` $\mathrm{val}\,v_a$ `}{` $\mathrm{val}\,v_b$ `}` $\rangle$

This code is used in section 40c.

41f ⟨ Start with an identifier 41f ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle$ `nx\idstr{` $\mathrm{val}\,v_a$ `}{` $\mathrm{val}\,v_b$ `}` $\rangle$

This code is used in section 40c.

41g ⟨ Start with a quoted string 41g ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle$ `nx\visflag{nx\termvstring}{}nx\chstr{` $\mathrm{val}\,v_a$ `}{` $\mathrm{val}\,v_b$ `}` $\rangle$

This code is used in section 40c.

41h ⟨ Start with a _ string 41h ⟩ =
$\Upsilon \leftarrow \langle$ `nx\visflag{nx\termvstring}{}nx\chstr{nx\_}{_}` $\rangle$

This code is used in section 40c.

41i ⟨ Start with a . string 41i ⟩ =
$\Upsilon \leftarrow \langle$ `nx\visflag{nx\termvstring}{}nx\chstr{.}{.}` $\rangle$

This code is used in section 40c.

42a ⟨ Attach an identifier 42a ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a$
$v_a \leftarrow v_a +_{\mathrm{sx}} [\,^{\mathrm{nox}}\backslash\_\,]$
$\pi_1(\Upsilon_3) \mapsto v_b$
$v_a \leftarrow v_a +_{\mathrm{s}} v_b$
$\pi_3(\Upsilon_1) \mapsto v_b$
$v_b \leftarrow v_b +_{\mathrm{sx}} [\,\backslash\mathtt{uscoreletter}\,]$
$\pi_2(\Upsilon_3) \mapsto v_c$
$v_b \leftarrow v_b +_{\mathrm{s}} v_c$
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{idstr}\,\{\,\mathrm{val}\,v_a\,\}\{\,\mathrm{val}\,v_b\,\}\rangle$
This code is used in section 40c.

42b ⟨ Attach qualifier to a name 42b ⟩ =
This code is used in section 40c.

42c ⟨ Attach an integer 42c ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{dotsp}\,^{\mathrm{nx}}\backslash\mathtt{sfxi}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 40c.

42d ⟨ Attach a subscripted integer 42d ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a \pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{dotsp}\,^{\mathrm{nx}}\backslash\mathtt{sfxi}\,\{\,^{\mathrm{nx}}\backslash\_\,\mathrm{val}\,v_a\,\}\{\,\backslash\mathtt{uscoreletter}\,\mathrm{val}\,v_b\,\}\rangle$
This code is used in section 40c.

42e ⟨ Attach a subscripted qualifier 42e ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a \pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{dotsp}\,^{\mathrm{nx}}\backslash\mathtt{qual}\,\{\,^{\mathrm{nx}}\backslash\_\,\mathrm{val}\,v_a\,\}\{\,\backslash\mathtt{uscoreletter}\,\mathrm{val}\,v_b\,\}\rangle$
This code is used in section 40c.

42f ⟨ Attach suffixes 42f ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{dotsp}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in sections 40c and 42g.

42g ⟨ Attach qualified suffixes 42g ⟩ =
⟨ Attach suffixes 42f ⟩
This code is used in section 40c.

42h ⟨ Start with a named suffix 42h ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{sfxn}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 40c.

42i ⟨ Start with a numeric suffix 42i ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}\backslash\mathtt{sfxi}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 40c.

42j ⟨ Add a dot separator 42j ⟩ =
$\Upsilon \leftarrow \langle\mathrm{val}\,\Upsilon_1\,^{\mathrm{nx}}\backslash\mathtt{dotsp}\,\rangle$
This code is used in section 40c.

42k ⟨ Attach integer suffix 42k ⟩ =
$\Upsilon \leftarrow \langle\mathrm{val}\,\Upsilon_1\,^{\mathrm{nx}}\backslash\mathtt{sfxi}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 40c.

42l ⟨ Attach a named suffix 42l ⟩ =
$\Upsilon \leftarrow \langle\mathrm{val}\,\Upsilon_1\,^{\mathrm{nx}}\backslash\mathtt{sfxn}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 40c.

43a ⟨ Attach a qualifier 43a ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1\, {}^{\mathrm{nx}}\backslash\mathtt{qual}\, \mathrm{val}\, \Upsilon_2 \rangle$
This code is used in section 40c.

43b ⟨ Start suffixes with a qualifier 43b ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\backslash\mathtt{qual}\, \mathrm{val}\, \Upsilon_1 \rangle$
This code is used in section 40c.

43c C preamble. In this case, there are no 'real' actions that our grammar performs, only TEX output, so this section is empty.
⟨ Name parser C preamble 43c ⟩ =
This code is used in section ch4.

43d C postamble. It is tricky to insert function definitions that use `bison`'s internal types, as they have to be inserted in a place that is aware of the internal definitions but before said definitions are used.
⟨ Name parser C postamble 43d ⟩ =
This code is used in section ch4.

43e Union of types.
⟨ Union of parser types 43e ⟩ =
This code is used in section ch4.

43f **The name scanner**

The scanner for `ld` name parser is essentially identical to that for the `bison` and `flex` name parser [1]). The only exception is the scanning of suffixes.

⟨ `ld_small_lexer.ll` 43f ⟩ =
⟨ Lexer definitions 43g ⟩
.....
⟨ Lexer C preamble 44c ⟩
.....
⟨ Lexer options 44d ⟩

⟨ Regular expressions for the name parser 44e ⟩

void *define_all_states*(**void**)
{
⟨ Collect all state definitions 44a ⟩
}

43g ⟨ Lexer definitions 43g ⟩ =
⟨ Lexer states 44b ⟩

⟨`aletter`⟩	[a–zA–Z]
⟨`letter`⟩	(_ | ⟨`aletter`⟩)
⟨`wc`⟩	([\'"]^c \ [_a–zA–Z0–9] | \.)
⟨`id`⟩	(⟨`aletter`⟩ | ⟨`aletter`⟩ (⟨`aletter`⟩ | [0–9])$_*$⟨`aletter`⟩)
⟨`id_strict`⟩	⟨`letter`⟩ ((⟨`letter`⟩ | [-0–9])$_*$⟨`letter`⟩)$_?$
⟨`meta_id`⟩	*⟨`id_strict`⟩*$_?$
⟨`int`⟩	[0–9]$_+$

This code is used in section 43f.

[1]) And is just as much of an overkill. This should serve as a cautionary tale of how suboptimal but convenient choices tend to take root.

44a ⟨ Collect all state definitions 44a ⟩ =
#define *_register_name*(*name*) *Define_State*(**#***name*, *name*) ▷ nothing for now ◁
#undef *_register_name*
This code is used in section 43f.

44b Strings and characters in directives/rules.
⟨ Lexer states 44b ⟩ =
⟨**state-x**⟩$_{\mathrm{f}}$ SC_ESCAPED_STRING SC_ESCAPED_CHARACTER
This code is used in section 43g.

44c ⟨ Lexer C preamble 44c ⟩ =
#include <stdint.h>
#include <stdbool.h>
This code is used in section 43f.

44d ⟨ Lexer options 44d ⟩ =

⟨option⟩$_{\mathrm{f}}$	bison-bridge
⟨option⟩$_{\mathrm{f}}$	noyywrap
⟨option⟩$_{\mathrm{f}}$	nounput
⟨option⟩$_{\mathrm{f}}$	noinput
⟨option⟩$_{\mathrm{f}}$	reentrant
⟨option⟩$_{\mathrm{f}}$	noyy_top_state
⟨option⟩$_{\mathrm{f}}$	debug
⟨option⟩$_{\mathrm{f}}$	stack
⟨output to⟩$_{\mathrm{f}}$	"ld_small_lexer.c"

This code is used in section 43f.

44e ⟨ Regular expressions for the name parser 44e ⟩ =
⟨ Scan white space 44f ⟩
⟨ Scan identifiers 44g ⟩
This code is used in section 43f.

44f White space is skipped in names [1]). Note that the input routine produced by SPLinT springs into action *after* the input passes through the scanning mechanism of TEX which makes it very unlikely that any of the characters below in the definition of 'whitespace' could make it to the scanner. They are left in, however, in case a custom input routine is used that produces such characters [2]).
⟨ Scan white space 44f ⟩ =
[␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩] **continue**
This code is used in section 44e.

44g Suffixes specific to ld name scanner are defined here, along with some other lexical elements. Most of these definitions mimic the ones for the bison name scanner.
⟨ Scan identifiers 44g ⟩ =

%(⟨aletter⟩ | [0–9] | [-_] | % | [<>])$_+$	**return**$_v$ %[a . . . Z 0 . . . 9]*
opt	**return**$_v$ opt
K	**return**$_v$ suffix$_{\mathrm{K_}}$
ext	**return**$_v$ ext
['._]	**return**$_c$
⟨wc⟩	**return**$_v$ * or ?
⟨id⟩	⟨ Prepare to process an identifier 45a ⟩
⟨meta_id⟩	⟨ Prepare to process a meta-identifier 45b ⟩
⟨int⟩	**return**$_v$ [0 . . . 9]*

[1]) How it finds its way *into* a name deserves some investigation for which we do not have the space here. [2]) Although, in this case, some special processing by the scanner would likely be required.

"
.

This code is used in section 44e.

45a ⟨ Prepare to process an identifier 45a ⟩ =
return$_v$ [a . . . Z 0 . . . 9]∗

This code is used in section 44g.

45b ⟨ Prepare to process a meta-identifier 45b ⟩ =
return$_v$ «meta identifier»

This code is used in section 44g.

45c A simple routine to detect trivial scanning problems.

⟨ React to a bad character 45c ⟩ =
if$_t$ [bad char]
    \yycomplain{invalid character(s): val\yytext}
fi
\yyerrterminate

This code is used in section 44g.

continue
⟨ React to a bad character 45c ⟩

5 Appendix

The original code of the `ld` parser and lexer is reproduced below. It is left mostly intact and is typeset by the pretty printing parser for `bison` input. The lexer (`flex`) input is given a similar treatment.

The treatment of comments is a bit more invasive. `CWEB` silently assumes that the comment refers to the preceding statement or a group of statements which is reflected in the way the comment is typeset. The comments in `ld` source files use the opposite convention. For the sake of consistency, such comments have been moved so as to make them fit the `CWEB` style. The comments meant to refer to a sizable portion of the program (such as a whole function or a group of functions) are put at the beginning of a `CWEB` section containing the appropriate part of the program.

`CWEB` treats comments as ordinary TEX so the comments are changed to take advantage of TEX formatting and introduce some visual cues. The convention of using *italics* for the original comments has been reversed: the italicized comments are the ones introduced by the author, *not* the original creators of `ld`.

47a **The original parser**

Here we present the full grammar of `ld`*, including some actions. The grammar is split into sections but otherwise is reproduced exactly. In addition to improving readability, such splitting allows* `CWEB` *to process the code in manageable increments. An observant reader will notice the difficulty* `CWEAVE` *is having with typesetting the structure tags that have the same name as the structure variables of the appropriate type. This is a well-known defect in* `CWEAVE`*'s design (see the requisite documentation) left uncorrected to discourage the poor programming practice.*

⟨ The original `ld` parser 47a ⟩ =

⟨ C setup for `ld` grammar 48a ⟩

```
⟨union⟩ bfd_vma integer;
        struct big_int {
                bfd_vma integer;
                char *str;
        } bigint;
        fill_type *fill;
        char *name;
        const char *cname;
        struct wildcard_spec wildcard;
        struct wildcard_list *wildcard_list;
        struct name_list *name_list;
        struct flag_info_list *flag_info_list;
        struct flag_info *flag_info;
        int token;
        union etree_union *etree;
        struct phdr_info {
                bfd_boolean filehdr;
                bfd_boolean phdrs;
                union etree_union *at;
                union etree_union *flags;
        } phdr;
        struct lang_nocrossref *nocrossref;
        struct lang_output_section_phdr_list *section_phdr;
        struct bfd_elf_version_deps *deflist;
        struct bfd_elf_version_expr *versyms;
        struct bfd_elf_version_tree *versnode;
```

⟨ Token definitions for the `ld` grammar 49a ⟩

⟨ Original `ld` grammar rules 50a ⟩

48a *The C code is left mostly intact (with the exception of a few comments) although it does not show up in the final output. The parts that are typeset represent the semantics that is reproduced in the typesetting parser. This includes all the state switching, as well as some other actions that affect the parser-lexer interaction (such as opening a new input buffer). The only exception to this rule is the code for the* MRI *script section of the grammar. It is reproduced mostly as an example of a pretty printed grammar, since otherwise,* MRI *scripts are completely ignored by the typeseting parser.*

⟨ C setup for `ld` grammar 48a ⟩ =

```
#define DONTDECLARE_MALLOC
#include "sysdep.h"
#include "bfd.h"
#include "bfdlink.h"
#include "ld.h"
#include "ldexp.h"
#include "ldver.h"
#include "ldlang.h"
#include "ldfile.h"
#include "ldemul.h"
#include "ldmisc.h"
#include "ldmain.h"
#include "mri.h"
#include "ldctor.h"
#include "ldlex.h"
#ifndef YYDEBUG
#define YYDEBUG 1
#endif
  static enum section_type sectype;
```

```
  static lang_memory_region_type *region;
  bfd_boolean ldgram_had_keep ⇐ FALSE;
  char *ldgram_vers_current_lang ⇐ Λ;
#define ERROR_NAME_MAX 20
  static char *error_names[ERROR_NAME_MAX];
  static int error_index;
#define PUSH_ERROR(x)
  if (error_index < ERROR_NAME_MAX) error_names[error_index] ⇐ x;
  error_index++;
#define POP_ERROR() error_index−−;
```

This code is used in section 47a.

49a *The token definitions and the corresponding* ⟨union⟩ *styles are intermixed, which makes sense in the traditional style of a* bison *script. When* CWEB *is used, however, it helps to introduce the code in small, manageable sections and take advantage of* CWEB*'s crossreferencing facilities to provide cues on the relationships between various parts of the code.*

⟨ Token definitions for the ld grammar 49a ⟩ =

```
⟨union⟩.etree: exp exp_with_type_opt_ mustbe_exp at_opt_ phdr_type phdr_val
⟨union⟩.etree: exp_without_type_opt_ subalign_opt_ align_opt_
⟨union⟩.fill: fill_opt fill_exp
⟨union⟩.name_list:
        exclude_name_list
⟨union⟩.wildcard_list:
        file_name_list
⟨union⟩.flag_info_list:
        sect_flag_list
⟨union⟩.flag_info:
        sect_flags
⟨union⟩.name:
        memspec_opt casesymlist
⟨union⟩.name:
        memspec_at_opt
⟨union⟩.cname:
        wildcard_name
⟨union⟩.wildcard:
        wildcard_spec

INT                    name                    name_L

⟨union⟩.integer:
        length
⟨union⟩.phdr: phdr_qualifiers
⟨union⟩.nocrossref:
        nocrossref_list
⟨union⟩.section_phdr:
        phdr_opt
⟨union⟩.integer:
        nocrossrefs_opt_

⟨right : token⟩ ⇐+ ⇐− ⇐× ⇐÷ ⇐ ⇐≪ ⇐≫ ⇐& ⇐∨
⟨right : token⟩ ? :
⟨left : token⟩ ∨
⟨left : token⟩ ∧
⟨left : token⟩ |
⟨left : token⟩ ⊕
⟨left : token⟩ &
⟨left : token⟩ = ≠
⟨left : token⟩ < > ⩽ ⩾
```

⟨**left** : *token*⟩ ≪ ≫
⟨**left** : *token*⟩ + −
⟨**left** : *token*⟩ × / ÷
⟨**right**⟩ unary

end

⟨**left** : *token*⟩ (

ALIGN_K	BLOCK	BIND	QUAD
SQUAD	LONG	SHORT	BYTE
SECTIONS	PHDRS	INSERT_K	AFTER
BEFORE	DATA_SEGMENT_ALIGN	DATA_SEGMENT_RELRO_END	DATA_SEGMENT_END
SORT_BY_NAME	SORT_BY_ALIGNMENT	SORT_NONE	SORT_BY_INIT_PRIORITY
{	}	SIZEOF_HEADERS	OUTPUT_FORMAT
FORCE_COMMON_ALLOCATION	OUTPUT_ARCH	INHIBIT_COMMON_ALLOCATION	SEGMENT_START
INCLUDE	MEMORY	REGION_ALIAS	LD_FEATURE
NOLOAD	DSECT	COPY	INFO
OVERLAY	DEFINED	TARGET_K	SEARCH_DIR
MAP	ENTRY	NEXT	SIZEOF
ALIGNOF	ADDR	LOADADDR	MAX_K
MIN_K	STARTUP	HLL	SYSLIB
FLOAT	NOFLOAT	NOCROSSREFS	ORIGIN
FILL	LENGTH	CREATE_OBJECT_SYMBOLS	INPUT
GROUP	OUTPUT	CONSTRUCTORS	ALIGNMOD
AT	SUBALIGN	HIDDEN	PROVIDE
PROVIDE_HIDDEN	AS_NEEDED		

⟨`union`⟩.*token* :
assign_op atype attributes$_{\text{opt}}$ *sect_constraint align_with_input*$_{\text{opt}}$
⟨`union`⟩.*name* :
filename

CHIP	LIST	SECT	ABSOLUTE
LOAD	NEWLINE	ENDWORD	ORDER
NAMEWORD	ASSERT_K	LOG2CEIL	FORMAT
PUBLIC	DEFSYMEND	BASE	ALIAS
TRUNCATE	REL	INPUT_SCRIPT	INPUT_MRI_SCRIPT
INPUT_DEFSYM	CASE	EXTERN	START
VERS_TAG	VERS_IDENTIFIER	GLOBAL	LOCAL
VERSION_K	INPUT_VERSION_SCRIPT	KEEP	ONLY_IF_RO
ONLY_IF_RW	SPECIAL	INPUT_SECTION_FLAGS	ALIGN_WITH_INPUT
EXCLUDE_FILE	CONSTANT		

⟨`union`⟩.*versyms* :
vers_defns
⟨`union`⟩.*versnode* :
vers_tag
⟨`union`⟩.*deflist* :
verdep

INPUT_DYNAMIC_LIST

This code is used in section 47a.

50a *The original* C *code has been preserved and presented along with the grammar rules in the next two sections (the* C *code has not been deleted in the subsequent sections either, it is just not typeset).*

⟨ Original `ld` grammar rules 50a ⟩ = 51a ▽

file :
INPUT_SCRIPT *script_file*
INPUT_MRI_SCRIPT *mri_script_file*

`INPUT_VERSION_SCRIPT` *version_script_file*
`INPUT_DYNAMIC_LIST` *dynamic_list_file*
`INPUT_DEFSYM` *defsym_expr*

filename :
name

defsym_expr :
∘ *ldlex_defsym*( );
name ⇐ *exp* *ldlex_popstate*( );

See also sections 51a, 51b, 52b, 52c, 53a, 53b, 53c, 54a, 55a, 56a, 56b, 57a, 57b, 58a, 58b, and 58c.

This code is used in section 47a.

51a Syntax within an MRI script file.

⟨ Original `ld` grammar rules 50a ⟩ + = △ 50a 51b ▽

mri_script_file :
∘ *ldlex_mri_script*( );
mri_script_lines *ldlex_popstate*( );

mri_script_lines :
mri_script_lines *mri_script_command* `NEWLINE`
∘

51b ⟨ Original `ld` grammar rules 50a ⟩ + = △ 51a 52b ▽

mri_script_command :

`CHIP` *exp*	
`CHIP` *exp* `,` *exp*	
name	⟨ Flag an unrecognized keyword 52a ⟩
`LIST`	*config*.*map_filename* ⇐ `"-"`;
`ORDER` *ordernamelist*	
`ENDWORD`	
`PUBLIC` name ⇐ *exp*	*mri_public*(Υ2, Υ4);
`PUBLIC` name `,` *exp*	*mri_public*(Υ2, Υ4);
`PUBLIC` name *exp*	*mri_public*(Υ2, Υ3);
`FORMAT` name	*mri_format*(Υ2);
`SECT` name `,` *exp*	*mri_output_section*(Υ2, Υ4);
`SECT` name *exp*	*mri_output_section*(Υ2, Υ3);
`SECT` name ⇐ *exp*	*mri_output_section*(Υ2, Υ4);
`ALIGN_K` name ⇐ *exp*	*mri_align*(Υ2, Υ4);
`ALIGN_K` name `,` *exp*	*mri_align*(Υ2, Υ4);
`ALIGNMOD` name ⇐ *exp*	*mri_alignmod*(Υ2, Υ4);
`ALIGNMOD` name `,` *exp*	*mri_alignmod*(Υ2, Υ4);
`ABSOLUTE` *mri_abs_name_list*	
`LOAD` *mri_load_name_list*	
`NAMEWORD` name	*mri_name*(Υ2);
`ALIAS` name `,` name	*mri_alias*(Υ2, Υ4, 0);
`ALIAS` name `,` `INT`	*mri_alias*(Υ2, 0, (**int**) Υ4.*integer*);
`BASE` *exp*	*mri_base*(Υ2);
`TRUNCATE` `INT`	*mri_truncate*((**unsigned int**) Υ2.*integer*);
`CASE` *casesymlist*	
`EXTERN` *extern_name_list*	
`INCLUDE` *filename*	*ldlex_script*( );
	ldfile_open_command_file(Υ2);
mri_script_lines end	*ldlex_popstate*( );
`START` name	*lang_add_entry*(Υ2, `FALSE`);
∘	

ordernamelist :

ordernamelist `,` name	*mri_order*(Υ3);

ordernamelist name — *mri_order*(ϒ2);
∘

mri_load_name_list :
name — *mri_load*(ϒ1);
mri_load_name_list , name — *mri_load*(ϒ3);

mri_abs_name_list :
name — *mri_only_load*(ϒ1);
mri_abs_name_list , name — *mri_only_load*(ϒ3);

casesymlist :
∘ — ϒ ⇐ Λ;
name
casesymlist , name

52a *Here is one way to deal with code that is too long to fit in an action.*

⟨ Flag an unrecognized keyword 52a ⟩ =
einfo(_("%P%F: unrecognised keyword in MRI style script '%s'\n"), ϒ1);

This code is used in section 51b.

52b Parsed as expressions so that commas separate entries.

⟨ Original ld grammar rules 50a ⟩ + = (△ 51b 52c ▽)

extern_name_list :
∘ — *ldlex_expression*( );
extern_name_list_body — *ldlex_popstate*( );

extern_name_list_body :
name
extern_name_list_body name
extern_name_list_body , name

script_file :
∘ — *ldlex_both*( );
ifile_list — *ldlex_popstate*( );

ifile_list :
*ifile_list ifile_p*1
∘

52c *All the commands that can appear in a standard linker script.*

⟨ Original ld grammar rules 50a ⟩ + = (△ 52b 53a ▽)

***ifile_p*1** :
memory
sections
phdrs
startup
high_level_library
low_level_library
floating_point_support
statement_anywhere
version
;
TARGET_K (name)
SEARCH_DIR (*filename*)
OUTPUT (*filename*)
OUTPUT_FORMAT (name)
OUTPUT_FORMAT (name , name , name)
OUTPUT_ARCH (name)
FORCE_COMMON_ALLOCATION

```
INHIBIT_COMMON_ALLOCATION
INPUT ( input_list )
GROUP
    ( input_list )
MAP ( filename )
INCLUDE filename                                   ldlex_script( );
                                                   ldfile_open_command_file(Υ2);
    ifile_list end                                 ldlex_popstate( );
NOCROSSREFS ( nocrossref_list )
EXTERN ( extern_name_list )
INSERT_K AFTER name
INSERT_K BEFORE name
REGION_ALIAS ( name , name )
LD_FEATURE ( name )
```

53a ⟨ Original ld grammar rules 50a ⟩ + = △ 52c 53b ▽

```
input_list :
    name
    input_list , name
    input_list name
    nameL
    input_list , nameL
    input_list nameL
    AS_NEEDED (
        input_list )
    input_list , AS_NEEDED (
        input_list )
    input_list AS_NEEDED (
        input_list )
sections :
    SECTIONS { sec_or_group_p1 }
sec_or_group_p1 :
    sec_or_group_p1 section
    sec_or_group_p1 statement_anywhere
    ○
statement_anywhere :
    ENTRY ( name )
    assignment end
    ASSERT_K                                       ldlex_expression( );
        ( exp , name )                             ldlex_popstate( );
                                                   lang_add_assignment(exp_assert(Υ4, Υ6));
```

53b The * and ? cases are there because the lexer returns them as separate tokens rather than as name.

⟨ Original ld grammar rules 50a ⟩ + = △ 53a 53c ▽

```
wildcard_name :
    name
    *
    ?
```

53c ⟨ Original ld grammar rules 50a ⟩ + = △ 53b 54a ▽

```
wildcard_spec :
    wildcard_name
    EXCLUDE_FILE ( exclude_name_list ) wildcard_name
    SORT_BY_NAME ( wildcard_name )
    SORT_BY_ALIGNMENT ( wildcard_name )
    SORT_NONE ( wildcard_name )
```

SORT_BY_NAME (SORT_BY_ALIGNMENT (*wildcard_name*))
SORT_BY_NAME (SORT_BY_NAME (*wildcard_name*))
SORT_BY_ALIGNMENT (SORT_BY_NAME (*wildcard_name*))
SORT_BY_ALIGNMENT (SORT_BY_ALIGNMENT (*wildcard_name*))
SORT_BY_NAME (EXCLUDE_FILE (*exclude_name_list*) *wildcard_name*)
SORT_BY_INIT_PRIORITY (*wildcard_name*)

sect_flag_list :
name
sect_flag_list & name

sect_flags :
INPUT_SECTION_FLAGS (*sect_flag_list*)

exclude_name_list :
exclude_name_list wildcard_name
wildcard_name

file_name_***list*** :
*file*_name_*list* ,$_{\text{opt}}$ *wildcard_spec*
wildcard_spec

input_section_spec_no_keep :
name
sect_flags name
[*file*_name_*list*]
sect_flags [*file*_name_*list*]
wildcard_spec (*file*_name_*list*)
sect_flags wildcard_spec (*file*_name_*list*)

54a ⟨ Original `ld` grammar rules 50a ⟩ + = △ 53c 55a ▽

input_section_spec :
input_section_spec_no_keep
KEEP (
input_section_spec_no_keep)

statement :
assignment end
CREATE_OBJECT_SYMBOLS
;
CONSTRUCTORS
SORT_BY_NAME (CONSTRUCTORS)
input_section_spec
length (*mustbe_exp*)
FILL (*fill_exp*)
ASSERT$_{\text{K}}$ *ldlex_expression*( );
(*exp* , name) *end* *ldlex_popstate*( );
INCLUDE *filename* *ldlex_script*( );
ldfile_open_command_file(Υ_2);
statement_list$_{\text{opt}}$ end *ldlex_popstate*( );

statement_list : *statement_list statement* | *statement*

statement_list$_{\text{opt}}$: ○ | *statement_list*

length : QUAD | SQUAD | LONG | SHORT | BYTE

fill_exp : *mustbe_exp*

fill$_{\text{opt}}$: ⇐ *fill_exp* | ○

assign_op : $\overset{+}{\Leftarrow}$ | $\overset{-}{\Leftarrow}$ | $\overset{\times}{\Leftarrow}$ | $\overset{\div}{\Leftarrow}$ | $\overset{\ll}{\Leftarrow}$ | $\overset{\gg}{\Leftarrow}$ | $\overset{\&}{\Leftarrow}$ | $\overset{\vee}{\Leftarrow}$

end : ; | ,

assignment :
name ⇐ *mustbe_exp*

name *assign_op mustbe_exp*
HIDDEN (name ⇐ *mustbe_exp*)
PROVIDE (name ⇐ *mustbe_exp*)
PROVIDE_HIDDEN (name ⇐ *mustbe_exp*)

,opt_ : , | ∘

memory :
MEMORY { *memory_spec_list*_opt }

memory_spec_list_opt :
memory_spec_list
∘

memory_spec_list :
memory_spec_list ,opt_ *memory_spec*
memory_spec

memory_spec :
name
*attributes*_opt : *origin_spec* ,opt_ *length_spec*
INCLUDE *filename* — *ldlex_script*( ); *ldfile_open_command_file*(Υ_2); *ldlex_popstate*( );
*memory_spec_list*_opt end

55a ⟨ Original `ld` grammar rules 50a ⟩ + ≡ △ 54a 56a ▽

origin_spec :
ORIGIN ⇐ *mustbe_exp*

length_spec :
LENGTH ⇐ *mustbe_exp*

attributes_opt :
∘
(*attributes_list*)

attributes_list :
attributes_string
attributes_list attributes_string

attributes_string :
name
¬ name

startup :
STARTUP (*filename*)

high_level_library :
HLL (*high_level_library*_name_*list*)
HLL ( )

high_level_library_name_***list*** :
*high_level_library*_name_*list* ,opt_ *filename*
filename

low_level_library :
SYSLIB (*low_level_library*_name_*list*)

low_level_library_name_***list*** :
*low_level_library*_name_*list* ,opt_ *filename*
∘

floating_point_support :
FLOAT
NOFLOAT

nocrossref_list :
∘
name *nocrossref_list*
name , *nocrossref_list*

mustbe_exp :

∘	*ldlex_expression*( );
exp	*ldlex_popstate*( );

56a *Rich expression syntax reproducing the one in* C.

⟨ Original `ld` grammar rules 50a ⟩ + =

exp :

− *exp*	⟨`prec` unary⟩
`(` *exp* `)`	
`NEXT (` *exp* `)`	⟨`prec` unary⟩
¬ *exp*	⟨`prec` unary⟩
+ *exp*	⟨`prec` unary⟩
`not` *exp*	⟨`prec` unary⟩

exp × *exp* | *exp* / *exp* | *exp* ÷ *exp* | *exp* + *exp* | *exp* − *exp*
exp ≪ *exp* | *exp* ≫ *exp* | *exp* = *exp* | *exp* ≠ *exp* | *exp* ⩽ *exp*
exp ⩾ *exp* | *exp* < *exp* | *exp* > *exp* | *exp* & *exp* | *exp* ⊕ *exp*
exp | *exp* | *exp* `?` *exp* `:` *exp* | *exp* ∧ *exp* | *exp* ∨ *exp*
`DEFINED (` name `)` | `INT` | `SIZEOF_HEADERS`
`ALIGNOF (` name `)` | `SIZEOF (` name `)`
`ADDR (` name `)` | `LOADADDR (` name `)`
`CONSTANT (` name `)` | `ABSOLUTE (` *exp* `)` | `ALIGN_K (` *exp* `)`
`ALIGN_K (` *exp* `,` *exp* `)` | `DATA_SEGMENT_ALIGN (` *exp* `,` *exp* `)`
`DATA_SEGMENT_RELRO_END (` *exp* `,` *exp* `)`
`DATA_SEGMENT_END (` *exp* `)`
`SEGMENT_START (` name `,` *exp* `)`
`BLOCK (` *exp* `)`
name
`MAX_K (` *exp* `,` *exp* `)`
`MIN_K (` *exp* `,` *exp* `)`
`ASSERT_K (` *exp* `,` name `)`
`ORIGIN (` name `)`
`LENGTH (` name `)`
`LOG2CEIL (` *exp* `)`

56b ⟨ Original `ld` grammar rules 50a ⟩ + =

memspec_at$_{\text{opt}}$:
`AT >` name
∘

at$_{\text{opt\_}}$:
`AT (` *exp* `)`
∘

align$_{\text{opt\_}}$:
`ALIGN_K (` *exp* `)`
∘

align_with_input$_{\text{opt\_}}$:
`ALIGN_WITH_INPUT`
∘

subalign$_{\text{opt\_}}$:
`SUBALIGN (` *exp* `)`
∘

sect_constraint :
`ONLY_IF_RO`
`ONLY_IF_RW`
`SPECIAL`
∘

57a The GROUP case is just enough to support the gcc *svr3.ifile* script. It is not intended to be full support. I'm not even sure what GROUP is supposed to mean.

⟨ Original ld grammar rules 50a ⟩ + =

△ 56b 57b ▽

section :

name	*ldlex_expression*( );
exp_with_type$_{\text{opt\_}}$ *at*$_{\text{opt\_}}$ *align*$_{\text{opt\_}}$ ↩	
align_with_input$_{\text{opt\_}}$ *subalign*$_{\text{opt\_}}$	*ldlex_popstate*( );
	ldlex_script( );
sect_constraint {	
statement_list$_{\text{opt}}$ }	*ldlex_popstate*( );
	ldlex_expression( );
memspec$_{\text{opt}}$ *memspec_at*$_{\text{opt}}$ *phdr*$_{\text{opt}}$ *fill*$_{\text{opt}}$	*ldlex_popstate*( );
,$_{\text{opt\_}}$	
OVERLAY	*ldlex_expression*( );
exp_without_type$_{\text{opt\_}}$ *nocrossrefs*$_{\text{opt\_}}$ ↩	
at$_{\text{opt\_}}$ *subalign*$_{\text{opt\_}}$	*ldlex_popstate*( );
	ldlex_script( );
{	
overlay_section }	*ldlex_popstate*( );
	ldlex_expression( );
memspec$_{\text{opt}}$ *memspec_at*$_{\text{opt}}$ *phdr*$_{\text{opt}}$ *fill*$_{\text{opt}}$	*ldlex_popstate*( );
,$_{\text{opt\_}}$	
GROUP	*ldlex_expression*( );
exp_with_type$_{\text{opt\_}}$	*ldlex_popstate*( );
{ *sec_or_group_p*$_1$ }	
INCLUDE *filename*	*ldlex_script*( );
	ldfile_open_command_file(Υ_2);
sec_or_group_p$_1$ end	*ldlex_popstate*( );

type : NOLOAD | DSECT | COPY | INFO | OVERLAY

atype : (*type*) | ○ | ( )

57b The BIND cases are to support the gcc *svr3.ifile* script. They aren't intended to implement full support for the BIND keyword. I'm not even sure what BIND is supposed to mean.

⟨ Original ld grammar rules 50a ⟩ + =

△ 57a 58a ▽

exp_with_type$_{\text{opt\_}}$:
exp atype :
atype :
BIND (*exp*) *atype* :
BIND (*exp*) BLOCK (*exp*) *atype* :

exp_without_type$_{\text{opt\_}}$:
exp :
:

nocrossrefs$_{\text{opt\_}}$:
○
NOCROSSREFS

memspec$_{\text{\_opt}}$:
> name
○

phdr$_{\text{\_opt}}$:
○
phdr$_{\text{\_opt}}$: name

overlay_section :

○	
overlay_section name	*ldlex_script*( );

{ *statement_list*$_{\text{opt}}$ } *ldlex_popstate*( );
ldlex_expression( );

phdr$_{\text{opt}}$ *fill*$_{\text{opt}}$ *ldlex_popstate*( );
,$_{\text{opt}}$

58a ⟨ Original `ld` grammar rules 50a ⟩ + = △ 57b 58b ▽

phdrs :
PHDRS { *phdr_list* }

phdr_list :
○
phdr_list phdr

phdr :
name *ldlex_expression*( );
phdr_type phdr_qualifiers *ldlex_popstate*( );
;

phdr_type :
exp

phdr_qualifiers :
○
name *phdr_val phdr_qualifiers*
AT (*exp*) *phdr_qualifiers*

phdr_val :
○
(*exp*)

dynamic_list_file :
○ *ldlex_version_file*( );
dynamic_list_nodes *ldlex_popstate*( );

dynamic_list_nodes :
dynamic_list_node
dynamic_list_nodes dynamic_list_node

dynamic_list_node :
{ *dynamic_list_tag* } ;

dynamic_list_tag :
vers_defns ;

58b This syntax is used within an external version script file.

⟨ Original `ld` grammar rules 50a ⟩ + = △ 58a 58c ▽

version_script_file :
○ *ldlex_version_file*( );
vers_nodes *ldlex_popstate*( );

58c This is used within a normal linker script file.

⟨ Original `ld` grammar rules 50a ⟩ + = △ 58b

version :
○ *ldlex_version_script*( );
VERSION$_{\text{K}}$ { *vers_nodes* } *ldlex_popstate*( );

vers_nodes :
vers_node
vers_nodes vers_node

vers_node :
{ *vers_tag* } ;
VERS_TAG { *vers_tag* } ;
VERS_TAG { *vers_tag* } *verdep* ;

verdep :
    VERS_TAG
    verdep VERS_TAG

vers_tag :
    ○
    vers_defns ;
    GLOBAL : *vers_defns* ;
    LOCAL : *vers_defns* ;
    GLOBAL : *vers_defns* ; LOCAL : *vers_defns* ;

vers_defns :
    VERS_IDENTIFIER
    name
    vers_defns ; VERS_IDENTIFIER
    vers_defns ; name
    vers_defns ; EXTERN name {
        vers_defns ;$_{\text{opt\_}}$ }
    EXTERN name {
        vers_defns ;$_{\text{opt\_}}$ }
    GLOBAL
    vers_defns ; GLOBAL
    LOCAL
    vers_defns ; LOCAL
    EXTERN
    vers_defns ; EXTERN

;$_{\text{opt\_}}$: ○ | ;

59a C sugar.

```
void yyerror(arg)
    const char *arg;
{
  if (ldfile_assumed_script)
    einfo(_("%P:%s: file format not recognized; treating as linker script\n"),
        ldlex_filename());
  if (error_index > 0 ∧ error_index < ERROR_NAME_MAX)
    einfo("%P%F:%S: %s in %s\n", Λ, arg, error_names[error_index − 1]);
  else einfo("%P%F:%S: %s\n", Λ, arg);
}
```

59b **The original lexer**

Note that the `ld` *lexer was designed to accomodate the syntax of various* `flex` *flavors, such as the original* `lex`*. The options* ⟨`a`⟩ *and* ⟨`o`⟩ *are ignored by* `flex` *and are a leftover from the archaic days of the original scanner generator.*

⟨ Original `ld` lexer 59b ⟩ =
  ⟨ Original `ld` macros 61a ⟩
  .....
  ⟨ Original `ld` preamble 60b ⟩
  .....
  ⟨ Miscellaneous `ld` lexer options 60a ⟩
  ⟨ Ignored options 60c ⟩

  ⟨ Original `ld` regular expressions 62a ⟩

  ⟨ Original `ld` postamble 61c ⟩

60a ⟨ Miscellaneous `ld` lexer options 60a ⟩ =
⟨ `option` ⟩$_{\mathrm{f}}$ `nounput`
This code is used in section 59b.

60b ⟨ Original `ld` preamble 60b ⟩ =
```
#include "bfd.h"
#include "safe-ctype.h"
#include "bfdlink.h"
#include "ld.h"
#include "ldmisc.h"
#include "ldexp.h"
#include "ldlang.h"
#include <ldgram.h>
#include "ldfile.h"
#include "ldlex.h"
#include "ldmain.h"
#include "libiberty.h"
  input_type parser_input;
    ▷ The type of top-level parser input. yylex and yyparse (indirectly) both check this. ◁
  unsigned int lineno ⇐ 1;    ▷ Line number in the current input file. ◁
  const char *lex_string ⇐ Λ;    ▷ The string we are currently lexing, or Λ if we are reading a file. ◁
#undef YY_INPUT
#define YY_INPUT(buf, result, max_size) result ⇐ yy_input (buf, max_size)    ▷ Support for flex reading from more
      than one input file (stream). include_stack is flex's input state for each open file; file_name_stack is the
      file names. lineno_stack is the current line numbers. If include_stack_ptr is 0, we haven't started reading
      anything yet. Otherwise, stack elements 0 through include_stack_ptr − 1 are valid. ◁
#ifndef YY_NO_UNPUT
#define YY_NO_UNPUT
#endif
#define MAX_INCLUDE_DEPTH 10
  static YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
  static const char *file_name_stack[MAX_INCLUDE_DEPTH];
  static unsigned int lineno_stack[MAX_INCLUDE_DEPTH];
  static unsigned int sysrooted_stack[MAX_INCLUDE_DEPTH];
  static unsigned int include_stack_ptr ⇐ 0;
  static int vers_node_nesting ⇐ 0;
  static int yy_input(char *, int);
  static void comment(void);
  static void lex_warn_invalid(char *where, char *what);
#define RTOKEN(x)
  {
    yylval.token ⇐ x;
    return x;
  }
#ifndef yywrap
  int yywrap(void)
  {
    return 1;
  }    ▷ Some versions of flex want this. ◁
#endif
```
This code is used in section 59b.

60c ⟨ Ignored options 60c ⟩ =
⟨ `a` ⟩$_{\mathrm{f}}$ 4000
⟨ `o` ⟩$_{\mathrm{f}}$ 5000
This code is used in section 59b.

61a *Some convenient abbreviations for regular expressions.*

⟨ Original `ld` macros 61a ⟩ =

⟨CMDFILENAMECHAR⟩	[_a–zA–Z0–9/._+$:[]\,=&!<>-~]
⟨CMDFILENAMECHAR$_1$⟩	[_a–zA–Z0–9/._+$:[]\,=&!<>~]
⟨FILENAMECHAR$_1$⟩	[_a–zA–Z/.\$_~]
⟨SYMBOLCHARN⟩	[_a–zA–Z/.\$_~0–9]
⟨FILENAMECHAR⟩	[_a–zA–Z0–9/.-_+=$:[]\,~]
⟨WILDCHAR⟩	[_a–zA–Z0–9/.-_+=$:[]\,~?*^!]
⟨WHITE⟩	[␣⟨t⟩⟨n⟩⟨r⟩]$_+$
⟨NOCFILENAMECHAR⟩	[_a–zA–Z0–9/.-_+$:[]\~]
⟨V_TAG⟩	[.$_a–zA–Z][._a–zA–Z0–9]$_*$
⟨V_IDENTIFIER⟩	[*?.$_a–zA–Z[]-!^\]([*?.$_a–zA–Z0–9[]-!^\]|::)$_*$

This code is used in section 59b.

61b States:

- `EXPRESSION` definitely in an expression
- `SCRIPT` definitely in a script
- `BOTH` either `EXPRESSION` or `SCRIPT`
- `DEFSYMEXP` in an argument to `--defsym`
- `MRI` in an MRI script
- `VERS_START` starting a Sun style mapfile
- `VERS_SCRIPT` a Sun style mapfile
- `VERS_NODE` a node within a Sun style mapfile

⟨ `ld` states 61b ⟩ =

⟨**state-s**⟩$_f$ SCRIPT
⟨**state-s**⟩$_f$ EXPRESSION
⟨**state-s**⟩$_f$ BOTH
⟨**state-s**⟩$_f$ DEFSYMEXP
⟨**state-s**⟩$_f$ MRI
⟨**state-s**⟩$_f$ VERS_START
⟨**state-s**⟩$_f$ VERS_SCRIPT
⟨**state-s**⟩$_f$ VERS_NODE

61c ⟨ Original `ld` postamble 61c ⟩ =

```
if (parser_input ≠ input_selected) {    ▷ The first token of the input determines the initial parser state. ◁
  input_type t ⇐ parser_input;
  parser_input ⇐ input_selected;
  switch (t) {
  case input_script: return INPUT_SCRIPT;
    break;
  case input_mri_script: return INPUT_MRI_SCRIPT;
    break;
  case input_version_script: return INPUT_VERSION_SCRIPT;
    break;
  case input_dynamic_list: return INPUT_DYNAMIC_LIST;
    break;
  case input_defsym: return INPUT_DEFSYM;
    break;
  default: abort( );
  }
}
```

This code is used in section 59b.

62a ⟨ Original `ld` regular expressions 62a ⟩ =

BOTH SCRIPT EXPRESSION VERS_START VERS_NODE VERS_SCRIPT$^+$

`/*`

$comment();$
RTOKEN(’-’);

DEFSYMEXP$^+$

`+`

RTOKEN(’+’);

DEFSYMEXP$^+$

⟨ FILENAMECHAR$_1$ ⟩ ⟨ SYMBOLCHARN ⟩$_*$

$yylval.name \Leftarrow xstrdup(yytext)$;**return** NAME;

DEFSYMEXP$^+$

`=`

RTOKEN(’=’);

MRI EXPRESSION$^+$

`$` ([0–9A–Fa–f])$_+$

$yylval.integer \Leftarrow bfd\_scan\_vma(yytext + 1, 0, 16)$;
$yylval.bigint.str \Leftarrow \Lambda$;
return INT;

MRI EXPRESSION$^+$

([0–9A–Fa–f])$_+\odot$
(H | h | X | x | B | b | O | o | D | d)

int *ibase*;
switch $(yytext[yyleng - 1])$\{
case ’X’:**case** ’x’:**case** ’H’:**case** ’h’:$ibase \Leftarrow 16$;
break;
case ’O’:**case** ’o’:$ibase \Leftarrow 8$;
break;
case ’B’:**case** ’b’:$ibase \Leftarrow 2$;
break;
default:$ibase \Leftarrow 10$;
\}
$yylval.integer \Leftarrow bfd\_scan\_vma(yytext, 0, ibase)$;
$yylval.bigint.str \Leftarrow \Lambda$;
return INT;

SCRIPT DEFSYMEXP MRI BOTH EXPRESSION$^+$

((($ | 0[xX]) ([0–9A–Fa–f])$_+$) | $\odot$
(([0–9])$_+$)) (M | K | m | k)$_?$

char $*s \Leftarrow yytext$;
int $ibase \Leftarrow 0$;
if $(*s =$ ’$’)$\{
$++s$;
$ibase \Leftarrow 16$;
\}
$yylval.integer \Leftarrow bfd\_scan\_vma(s, 0, ibase)$;
$yylval.bigint.str \Leftarrow \Lambda$;
if $(yytext[yyleng - 1] =$ ’M’ $\vee\ yytext[yyleng - 1] =$ ’m’$)$\{
$yylval.integer \overset{*}{\Leftarrow} 1024 * 1024$;
\}
else if $(yytext[yyleng - 1] =$ ’K’ $\vee\ yytext[yyleng - 1] =$ ’k’$)$\{
$yylval.integer \overset{*}{\Leftarrow} 1024$;
\}
else if $(yytext[0] =$ ’0’ $\wedge\ (yytext[1] =$ ’x’ $\vee\ yytext[1] =$ ’X’$))$\{
$yylval.bigint.str \Leftarrow xstrdup(yytext + 2)$;
\}
return INT;

BOTH SCRIPT EXPRESSION MRI$^+$

`]`

RTOKEN(’]’);

BOTH SCRIPT EXPRESSION MRI$^+$

`[`

RTOKEN(’[’);

179

```
BOTH SCRIPT EXPRESSION MRI+
  <<=                                   RTOKEN(LSHIFTEQ);
BOTH SCRIPT EXPRESSION MRI+
  >>=                                   RTOKEN(RSHIFTEQ);
BOTH SCRIPT EXPRESSION MRI+
  ||                                    RTOKEN(OROR);
BOTH SCRIPT EXPRESSION MRI+
  ==                                    RTOKEN(EQ);
BOTH SCRIPT EXPRESSION MRI+
  !=                                    RTOKEN(NE);
BOTH SCRIPT EXPRESSION MRI+
  >=                                    RTOKEN(GE);
BOTH SCRIPT EXPRESSION MRI+
  <=                                    RTOKEN(LE);
BOTH SCRIPT EXPRESSION MRI+
  <<                                    RTOKEN(LSHIFT);
BOTH SCRIPT EXPRESSION MRI+
  >>                                    RTOKEN(RSHIFT);
BOTH SCRIPT EXPRESSION MRI+
  +=                                    RTOKEN(PLUSEQ);
BOTH SCRIPT EXPRESSION MRI+
  -=                                    RTOKEN(MINUSEQ);
BOTH SCRIPT EXPRESSION MRI+
  *=                                    RTOKEN(MULTEQ);
BOTH SCRIPT EXPRESSION MRI+
  /=                                    RTOKEN(DIVEQ);
BOTH SCRIPT EXPRESSION MRI+
  &=                                    RTOKEN(ANDEQ);
BOTH SCRIPT EXPRESSION MRI+
  |=                                    RTOKEN(OREQ);
BOTH SCRIPT EXPRESSION MRI+
  &&                                    RTOKEN(ANDAND);
BOTH SCRIPT EXPRESSION MRI+
  >                                     RTOKEN('>');
BOTH SCRIPT EXPRESSION MRI+
  ,                                     RTOKEN(',');
BOTH SCRIPT EXPRESSION MRI+
  &                                     RTOKEN('&');
BOTH SCRIPT EXPRESSION MRI+
  |                                     RTOKEN('|');
BOTH SCRIPT EXPRESSION MRI+
  ~                                     RTOKEN('~');
BOTH SCRIPT EXPRESSION MRI+
  !                                     RTOKEN('!');
BOTH SCRIPT EXPRESSION MRI+
  ?                                     RTOKEN('?');
```

```
BOTH SCRIPT EXPRESSION MRI+
  *                                RTOKEN('*');
BOTH SCRIPT EXPRESSION MRI+
  +                                RTOKEN('+');
BOTH SCRIPT EXPRESSION MRI+
  -                                RTOKEN('-');
BOTH SCRIPT EXPRESSION MRI+
  /                                RTOKEN('/');
BOTH SCRIPT EXPRESSION MRI+
  %                                RTOKEN('%');
BOTH SCRIPT EXPRESSION MRI+
  <                                RTOKEN('<');
BOTH SCRIPT EXPRESSION MRI+
  =                                RTOKEN('=');
BOTH SCRIPT EXPRESSION MRI+
  }                                RTOKEN('}');
BOTH SCRIPT EXPRESSION MRI+
  {                                RTOKEN('{');
BOTH SCRIPT EXPRESSION MRI+
  )                                RTOKEN(')');
BOTH SCRIPT EXPRESSION MRI+
  (                                RTOKEN('(');
BOTH SCRIPT EXPRESSION MRI+
  :                                RTOKEN(':');
BOTH SCRIPT EXPRESSION MRI+
  ;                                RTOKEN(';');
BOTH SCRIPT+
  MEMORY                           RTOKEN(MEMORY);
BOTH SCRIPT+
  REGION_ALIAS                     RTOKEN(REGION_ALIAS);
BOTH SCRIPT+
  LD_FEATURE                       RTOKEN(LD_FEATURE);
BOTH SCRIPT EXPRESSION+
  ORIGIN                           RTOKEN(ORIGIN);
BOTH SCRIPT+
  VERSION                          RTOKEN(VERSIONK);
EXPRESSION BOTH SCRIPT+
  BLOCK                            RTOKEN(BLOCK);
EXPRESSION BOTH SCRIPT+
  BIND                             RTOKEN(BIND);
BOTH SCRIPT EXPRESSION+
  LENGTH                           RTOKEN(LENGTH);
EXPRESSION BOTH SCRIPT+
  ALIGN                            RTOKEN(ALIGN_K);
EXPRESSION BOTH SCRIPT+
  DATA_SEGMENT_ALIGN               RTOKEN(DATA_SEGMENT_ALIGN);
```

179

```
EXPRESSION BOTH SCRIPT+
  DATA_SEGMENT_RELRO_END          RTOKEN(DATA_SEGMENT_RELRO_END);
EXPRESSION BOTH SCRIPT+
  DATA_SEGMENT_END                RTOKEN(DATA_SEGMENT_END);
EXPRESSION BOTH SCRIPT+
  ADDR                            RTOKEN(ADDR);
EXPRESSION BOTH SCRIPT+
  LOADADDR                        RTOKEN(LOADADDR);
EXPRESSION BOTH SCRIPT+
  ALIGNOF                         RTOKEN(ALIGNOF);
EXPRESSION BOTH+
  MAX                             RTOKEN(MAX_K);
EXPRESSION BOTH+
  MIN                             RTOKEN(MIN_K);
EXPRESSION BOTH+
  LOG2CEIL                        RTOKEN(LOG2CEIL);
EXPRESSION BOTH SCRIPT+
  ASSERT                          RTOKEN(ASSERT_K);
BOTH SCRIPT+
  ENTRY                           RTOKEN(ENTRY);
BOTH SCRIPT MRI+
  EXTERN                          RTOKEN(EXTERN);
EXPRESSION BOTH SCRIPT+
  NEXT                            RTOKEN(NEXT);
EXPRESSION BOTH SCRIPT+
  sizeof_headers                  RTOKEN(SIZEOF_HEADERS);
EXPRESSION BOTH SCRIPT+
  SIZEOF_HEADERS                  RTOKEN(SIZEOF_HEADERS);
EXPRESSION BOTH SCRIPT+
  SEGMENT_START                   RTOKEN(SEGMENT_START);
BOTH SCRIPT+
  MAP                             RTOKEN(MAP);
EXPRESSION BOTH SCRIPT+
  SIZEOF                          RTOKEN(SIZEOF);
BOTH SCRIPT+
  TARGET                          RTOKEN(TARGET_K);
BOTH SCRIPT+
  SEARCH_DIR                      RTOKEN(SEARCH_DIR);
BOTH SCRIPT+
  OUTPUT                          RTOKEN(OUTPUT);
BOTH SCRIPT+
  INPUT                           RTOKEN(INPUT);
EXPRESSION BOTH SCRIPT+
  GROUP                           RTOKEN(GROUP);
EXPRESSION BOTH SCRIPT+
  AS_NEEDED                       RTOKEN(AS_NEEDED);
```

```
EXPRESSION BOTH SCRIPT+
  DEFINED                          RTOKEN(DEFINED);
BOTH SCRIPT+
  CREATE_OBJECT_SYMBOLS            RTOKEN(CREATE_OBJECT_SYMBOLS);
BOTH SCRIPT+
  CONSTRUCTORS                     RTOKEN(CONSTRUCTORS);
BOTH SCRIPT+
  FORCE_COMMON_ALLOCATION          RTOKEN(FORCE_COMMON_ALLOCATION);
BOTH SCRIPT+
  INHIBIT_COMMON_ALLOCATION        RTOKEN(INHIBIT_COMMON_ALLOCATION);
BOTH SCRIPT+
  SECTIONS                         RTOKEN(SECTIONS);
BOTH SCRIPT+
  INSERT                           RTOKEN(INSERT_K);
BOTH SCRIPT+
  AFTER                            RTOKEN(AFTER);
BOTH SCRIPT+
  BEFORE                           RTOKEN(BEFORE);
BOTH SCRIPT+
  FILL                             RTOKEN(FILL);
BOTH SCRIPT+
  STARTUP                          RTOKEN(STARTUP);
BOTH SCRIPT+
  OUTPUT_FORMAT                    RTOKEN(OUTPUT_FORMAT);
BOTH SCRIPT+
  OUTPUT_ARCH                      RTOKEN(OUTPUT_ARCH);
BOTH SCRIPT+
  HLL                              RTOKEN(HLL);
BOTH SCRIPT+
  SYSLIB                           RTOKEN(SYSLIB);
BOTH SCRIPT+
  FLOAT                            RTOKEN(FLOAT);
BOTH SCRIPT+
  QUAD                             RTOKEN(QUAD);
BOTH SCRIPT+
  SQUAD                            RTOKEN(SQUAD);
BOTH SCRIPT+
  LONG                             RTOKEN(LONG);
BOTH SCRIPT+
  SHORT                            RTOKEN(SHORT);
BOTH SCRIPT+
  BYTE                             RTOKEN(BYTE);
BOTH SCRIPT+
  NOFLOAT                          RTOKEN(NOFLOAT);
EXPRESSION BOTH SCRIPT+
  NOCROSSREFS                      RTOKEN(NOCROSSREFS);
```

```
BOTH SCRIPT+
  OVERLAY                          RTOKEN(OVERLAY);
BOTH SCRIPT+
  SORT_BY_NAME                     RTOKEN(SORT_BY_NAME);
BOTH SCRIPT+
  SORT_BY_ALIGNMENT                RTOKEN(SORT_BY_ALIGNMENT);
BOTH SCRIPT+
  SORT                             RTOKEN(SORT_BY_NAME);
BOTH SCRIPT+
  SORT_BY_INIT_PRIORITY            RTOKEN(SORT_BY_INIT_PRIORITY);
BOTH SCRIPT+
  SORT_NONE                        RTOKEN(SORT_NONE);
EXPRESSION BOTH SCRIPT+
  NOLOAD                           RTOKEN(NOLOAD);
EXPRESSION BOTH SCRIPT+
  DSECT                            RTOKEN(DSECT);
EXPRESSION BOTH SCRIPT+
  COPY                             RTOKEN(COPY);
EXPRESSION BOTH SCRIPT+
  INFO                             RTOKEN(INFO);
EXPRESSION BOTH SCRIPT+
  OVERLAY                          RTOKEN(OVERLAY);
EXPRESSION BOTH SCRIPT+
  ONLY_IF_RO                       RTOKEN(ONLY_IF_RO);
EXPRESSION BOTH SCRIPT+
  ONLY_IF_RW                       RTOKEN(ONLY_IF_RW);
EXPRESSION BOTH SCRIPT+
  SPECIAL                          RTOKEN(SPECIAL);
BOTH SCRIPT+
  o                                RTOKEN(ORIGIN);
BOTH SCRIPT+
  org                              RTOKEN(ORIGIN);
BOTH SCRIPT+
  l                                RTOKEN(LENGTH);
BOTH SCRIPT+
  len                              RTOKEN(LENGTH);
EXPRESSION BOTH SCRIPT+
  INPUT_SECTION_FLAGS              RTOKEN(INPUT_SECTION_FLAGS);
EXPRESSION BOTH SCRIPT+
  INCLUDE                          RTOKEN(INCLUDE);
BOTH SCRIPT+
  PHDRS                            RTOKEN(PHDRS);
EXPRESSION BOTH SCRIPT+
  AT                               RTOKEN(AT);
EXPRESSION BOTH SCRIPT+
  ALIGN_WITH_INPUT                 RTOKEN(ALIGN_WITH_INPUT);
```

EXPRESSION BOTH SCRIPT+
SUBALIGN RTOKEN(SUBALIGN);

EXPRESSION BOTH SCRIPT+
HIDDEN RTOKEN(HIDDEN);

EXPRESSION BOTH SCRIPT+
PROVIDE RTOKEN(PROVIDE);

EXPRESSION BOTH SCRIPT+
PROVIDE_HIDDEN RTOKEN(PROVIDE_HIDDEN);

EXPRESSION BOTH SCRIPT+
KEEP RTOKEN(KEEP);

EXPRESSION BOTH SCRIPT+
EXCLUDE_FILE RTOKEN(EXCLUDE_FILE);

EXPRESSION BOTH SCRIPT+
CONSTANT RTOKEN(CONSTANT);

MRI+
.* ⟨n⟩? ++*lineno*;

MRI+
⟨n⟩ ++*lineno*;RTOKEN(NEWLINE);

MRI+
* .* ▷ MRI comment line ◁

MRI+
; .* ▷ MRI comment line ◁

MRI+
END RTOKEN(ENDWORD);

MRI+
ALIGNMOD RTOKEN(ALIGNMOD);

MRI+
ALIGN RTOKEN(ALIGN_K);

MRI+
CHIP RTOKEN(CHIP);

MRI+
BASE RTOKEN(BASE);

MRI+
ALIAS RTOKEN(ALIAS);

MRI+
TRUNCATE RTOKEN(TRUNCATE);

MRI+
LOAD RTOKEN(LOAD);

MRI+
PUBLIC RTOKEN(PUBLIC);

MRI+
ORDER RTOKEN(ORDER);

MRI+
NAME RTOKEN(NAMEWORD);

MRI+
FORMAT RTOKEN(FORMAT);

MRI+
CASE RTOKEN(CASE);

MRI+
START RTOKEN(START);

MRI+
LIST .* RTOKEN(LIST); ▷ LIST and ignore to end of line ◁

MRI+
SECT RTOKEN(SECT);

EXPRESSION BOTH SCRIPT MRI+
ABSOLUTE RTOKEN(ABSOLUTE);

MRI+
end RTOKEN(ENDWORD);

MRI+
alignmod RTOKEN(ALIGNMOD);

MRI+
align RTOKEN(ALIGN_K);

MRI+
chip RTOKEN(CHIP);

MRI+
base RTOKEN(BASE);

MRI+
alias RTOKEN(ALIAS);

MRI+
truncate RTOKEN(TRUNCATE);

MRI+
load RTOKEN(LOAD);

MRI+
public RTOKEN(PUBLIC);

MRI+
order RTOKEN(ORDER);

MRI+
name RTOKEN(NAMEWORD);

MRI+
format RTOKEN(FORMAT);

MRI+
case RTOKEN(CASE);

MRI+
extern RTOKEN(EXTERN);

MRI+
start RTOKEN(START);

MRI+
list .* RTOKEN(LIST); ▷ LIST and ignore to end of line ◁

MRI+
sect RTOKEN(SECT);

EXPRESSION BOTH SCRIPT MRI+
absolute RTOKEN(ABSOLUTE);

MRI$^+$
⟨FILENAMECHAR$_1$⟩ ⟨NOCFILENAMECHAR⟩$_*$

▷ Filename without commas, needed to parse MRI stuff ◁
yylval.name ⇐ *xstrdup*(*yytext*);
return NAME;

BOTH$^+$
⟨FILENAMECHAR$_1$⟩ ⟨FILENAMECHAR⟩$_*$

yylval.name ⇐ *xstrdup*(*yytext*);
return NAME;

BOTH$^+$
-l ⟨FILENAMECHAR⟩$_+$

yylval.name ⇐ *xstrdup*(*yytext* + 2);
return LNAME;

EXPRESSION$^+$
⟨FILENAMECHAR$_1$⟩ ⟨NOCFILENAMECHAR⟩$_*$

yylval.name ⇐ *xstrdup*(*yytext*);
return NAME;

EXPRESSION$^+$
-l ⟨NOCFILENAMECHAR⟩$_+$

yylval.name ⇐ *xstrdup*(*yytext* + 2);
return LNAME;

SCRIPT$^+$
⟨WILDCHAR⟩$_*$

▷ Annoyingly, this pattern can match comments, ◁
▷ and we have longest match issues to consider. ◁
▷ So if the first two characters are a comment ◁
▷ opening, put the input back and try again. ◁
if (*yytext*[0] = '/' ∧ *yytext*[1] = '*'){
yyless(2);
comment( );
}
else{
yylval.name ⇐ *xstrdup*(*yytext*);
return NAME;
}

EXPRESSION BOTH SCRIPT VERS_NODE$^+$
" ["]$^c{}_*$ "

▷ No matter the state, quotes give what's inside ◁
yylval.name ⇐ *xstrdup*(*yytext* + 1);
yylval.name[*yyleng* − 2] ⇐ 0;
return NAME;

BOTH SCRIPT EXPRESSION$^+$
⟨n⟩

lineno++;

MRI BOTH SCRIPT EXPRESSION$^+$
[␣⟨t⟩⟨r⟩]$_+$

VERS_NODE VERS_SCRIPT$^+$
[:,;]

return ∗*yytext*;

VERS_NODE$^+$
global

RTOKEN(GLOBAL);

VERS_NODE$^+$
local

RTOKEN(LOCAL);

VERS_NODE$^+$
extern

RTOKEN(EXTERN);

VERS_NODE$^+$
⟨V_IDENTIFIER⟩

yylval.name ⇐ *xstrdup*(*yytext*);
return VERS_IDENTIFIER;

```
VERS_SCRIPT+
  ⟨V_TAG⟩                                   yylval.name ⇐ xstrdup(yytext);
                                            return VERS_TAG;

VERS_START+
  {                                         BEGIN(VERS_SCRIPT);return *yytext;

VERS_SCRIPT+
  {                                         BEGIN(VERS_NODE);
                                            vers_node_nesting ⇐ 0;
                                            return *yytext;

VERS_SCRIPT+
  }                                         return *yytext;

VERS_NODE+
  {                                         vers_node_nesting++;return *yytext;

VERS_NODE+
  }                                         if (--vers_node_nesting < 0)BEGIN(VERS_SCRIPT);
                                            return *yytext;

VERS_START VERS_NODE VERS_SCRIPT+
  [⟨n⟩]                                     lineno++;

VERS_START VERS_NODE VERS_SCRIPT+
  #.*                                       ; ▷ Eat up comments ◁

VERS_START VERS_NODE VERS_SCRIPT+
  [␣⟨t⟩⟨r⟩]+                                ; ▷ Eat up whitespace ◁
⟨EOF⟩                                       include_stack_ptr--;
                                            if (include_stack_ptr = 0)yyterminate();
                                            elseyy_switch_to_buffer(include_stack[include_stack_ptr]);
                                            lineno ⇐ lineno_stack[include_stack_ptr];
                                            input_flags.sysrooted ⇐ sysrooted_stack[include_stack_ptr];
                                            return END;

SCRIPT MRI VERS_START VERS_SCRIPT VERS_NODE+
  .                                         lex_warn_invalid("␣in␣script", yytext);

EXPRESSION DEFSYMEXP BOTH+
  .                                         lex_warn_invalid("␣in␣expression", yytext);
```

This code is used in section 59b.

71a Switch `flex` to reading script file *name*, open on *file*, saving the current input info on the include stack.

```
⟨ Supporting C code 71a ⟩ =                                                    72a ▽
  void lex_push_file(FILE *file, const char *name, unsigned int sysrooted)
  {
    if (include_stack_ptr ⩾ MAX_INCLUDE_DEPTH) {
      einfo("%F:includes␣nested␣too␣deeply\n");
    }
    file_name_stack[include_stack_ptr] ⇐ name;
    lineno_stack[include_stack_ptr] ⇐ lineno;
    sysrooted_stack[include_stack_ptr] ⇐ input_flags.sysrooted;
    include_stack[include_stack_ptr] ⇐ YY_CURRENT_BUFFER;
    include_stack_ptr++;
    lineno ⇐ 1;
    input_flags.sysrooted ⇐ sysrooted;
    yyin ⇐ file;
    yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIZE));
  }
```

See also sections 72a, 72b, 72c, 73a, 73b, 74a, and 74b.

72a Return a newly created `flex` input buffer containing *string*, which is *size* bytes long.

⟨ Supporting C code 71a ⟩ + =

```
static YY_BUFFER_STATE yy_create_string_buffer(const char *string, size_t size)
{
  YY_BUFFER_STATE b;
  b ⇐ malloc(sizeof(struct yy_buffer_state));    ▷ Calls to malloc get turned by sed into xmalloc. ◁
  b→yy_input_file ⇐ 0;
  b→yy_buf_size ⇐ size;
  b→yy_ch_buf ⇐ malloc((unsigned)(b→yy_buf_size + 3));    ▷ yy_ch_buf has to be 2 characters longer than the
      size given because we need to put in 2 end-of-buffer characters. ◁
  b→yy_ch_buf[0] ⇐ '\n';
  strcpy(b→yy_ch_buf + 1, string);
  b→yy_ch_buf[size + 1] ⇐ YY_END_OF_BUFFER_CHAR;
  b→yy_ch_buf[size + 2] ⇐ YY_END_OF_BUFFER_CHAR;
  b→yy_n_chars ⇐ size + 1;
  b→yy_buf_pos ⇐ &b→yy_ch_buf[1];
  b→yy_is_our_buffer ⇐ 1;
  b→yy_is_interactive ⇐ 0;
  b→yy_at_bol ⇐ 1;
  b→yy_fill_buffer ⇐ 0;
#ifdef YY_BUFFER_NEW
  b→yy_buffer_status ⇐ YY_BUFFER_NEW;
#else
  b→yy_eof_status ⇐ EOF_NOT_SEEN;
#endif
  return b;
}
```

72b Switch `flex` to reading from *string*, saving the current input info on the include stack.

⟨ Supporting C code 71a ⟩ + =

```
void lex_redirect(const char *string, const char *fake_filename, unsigned int count)
{
  YY_BUFFER_STATE tmp;
  yy_init ⇐ 0;
  if (include_stack_ptr ⩾ MAX_INCLUDE_DEPTH) {
    einfo("%F: macros nested too deeply\n");
  }
  file_name_stack[include_stack_ptr] ⇐ fake_filename;
  lineno_stack[include_stack_ptr] ⇐ lineno;
  include_stack[include_stack_ptr] ⇐ YY_CURRENT_BUFFER;
  include_stack_ptr++;
  lineno ⇐ count;
  tmp ⇐ yy_create_string_buffer(string, strlen(string));
  yy_switch_to_buffer(tmp);
}
```

72c Functions to switch to a different `flex` start condition, saving the current start condition on *state_stack*.

⟨ Supporting C code 71a ⟩ + =

```
static int state_stack[MAX_INCLUDE_DEPTH * 2];
static int *state_stack_p ⇐ state_stack;
void ldlex_script(void)
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(SCRIPT);
}
void ldlex_mri_script(void)
```

```
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(MRI);
}
void ldlex_version_script(void)
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(VERS_START);
}
void ldlex_version_file(void)
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(VERS_SCRIPT);
}
void ldlex_defsym(void)
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(DEFSYMEXP);
}
void ldlex_expression(void)
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(EXPRESSION);
}
void ldlex_both(void)
{
  *(state_stack_p)++ ⇐ yy_start;
  BEGIN(BOTH);
}
void ldlex_popstate(void)
{
  yy_start ⇐ *(--state_stack_p);
}
```

73a Return the current file name, or the previous file if no file is current.

⟨ Supporting C code 71a ⟩ + = (△ 72c 73b ▽)

```
const char *ldlex_filename(void)
{
  return file_name_stack[include_stack_ptr − (include_stack_ptr ≠ 0)];
}
```

73b Place up to *max_size* characters in *buf* and return either the number of characters read, or 0 to indicate EOF.

⟨ Supporting C code 71a ⟩ + = (△ 73a 74a ▽)

```
static int yy_input(char *buf, int max_size)
{
  int result ⇐ 0;
  if (YY_CURRENT_BUFFER→yy_input_file) {
    if (yyin) {
      result ⇐ fread(buf, 1, max_size, yyin);
      if (result < max_size ∧ ferror(yyin)) einfo("%F%P: read in flex scanner failed\n");
    }
  }
  return result;
}
```

74a Eat the rest of a C-style comment.

⟨ Supporting C code 71a ⟩ + =

```
static void comment(void)
{
  int c;
  while (1) {
    c ⇐ input( );
    while (c ≠ '*' ∧ c ≠ EOF) {
      if (c = '\n') lineno++;
      c ⇐ input( );
    }
    if (c = '*') {
      c ⇐ input( );
      while (c = '*') c ⇐ input( );
      if (c = '/') break;     ▷ found the end ◁
    }
    if (c = '\n') lineno++;
    if (c = EOF) {
      einfo("%F%P:␣EOF␣in␣comment\n");
      break;
    }
  }
}
```

△ 73b 74b ▽

74b Warn the user about a garbage character *what* in the input in context *where*.

⟨ Supporting C code 71a ⟩ + =

```
static void lex_warn_invalid(char *where, char *what)
{
  char buf[5];
  if (ldfile_assumed_script) {     ▷ If we have found an input file whose format we do not recognize, and we are
        therefore treating it as a linker script, and we find an invalid character, then most likely this is a real
        object file of some different format. Treat it as such. ◁
    bfd_set_error(bfd_error_file_not_recognized);
    einfo("%F%s:␣file␣not␣recognized:␣%E\n", ldlex_filename( ));
  }
  if (^not ISPRINT(*what)) {
    sprintf(buf, "\\%03o", *(unsigned char *) what);
    what ⇐ buf;
  }
  einfo("%P:%S:␣ignoring␣invalid␣character␣`%s'%s\n", Λ, what, where);
}
```

△ 74a

6 Index

This section lists the variable names and (in some cases) the keywords used inside the 'language sections' of the CWEB source. It takes advantage of the built-in facility of CWEB to supply references for both definitions (set in *italic*) as well as uses for each C identifier in the text.

Special facilities have been added to extend indexing to `bison` grammar terms, TEX control sequences encountered in `bison` actions, and file and section names encountered in `ld` scripts. For a detailed description of the various conventions adhered to by the index entries the reader is encouraged to consult the remarks preceding the index of the document describing the core of the SPLinT suite. We will only mention here that (consistent with the way `bison` references are treated) a script example:

memory		attributes	starts at	length
	MEMORY$_1$	xrw	$2000\,0000_{16}$	$20 \cdot 2^{10}$
	MEMORY$_2$	rx	$800\,0000_{16}$	$128 \cdot 2^{10}$

$\_var\__1 \Leftarrow 2000\,5000_{16}$

inside the TEX part of a CWEB section will generate several index entries, as well, mimicking CWEB's behavior for the *inline* C (|...|). Such entries are labeled with °, to provide a reminder of their origin.

Bison, Flex, TEX, and Ld indices

Flex index

Ld index

TEX index

A LIST OF ALL SECTIONS

Contents (ldman)