SPLINT
reference

version 1.05

LD parser reference

Alex Shibakov

January 27, 2016

1 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` which is later included in the TEX file produced by `CWEAVE`.

⟨ Set up the generic parser machinery 1 ⟩ =

```
\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    %   ▷ convenient ◁

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

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

\input yystype.sty  %   ▷ scanner auxiliary types and functions ◁
\input yyunion.sty  %   ▷ parser data structures ◁
\input yxunion.sty  %   ▷ extended parser data structures ◁
\input ldunion.sty  %   ▷ ld parser data structures ◁
```

This code is used in section 9.

2 Bootstrapping

To produce a usable parser/scanner duo, several pieces of code must be generated. The most important of these are the *table files* (`ptab.tex` and `ltab.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 the minimized versions used in bootstrapping the `bison` parser.

The first 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 serves a secondary role: it provides hints for the macros that typeset the grammar terms, after the `\tokeneq` macro is redefined to serve 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 can now only be accomplished by examining the grammar source file and is delegated to the bootstrapping phase parser.

The other responsibility of the bootstrapping parser 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 original bootstrapping macros were 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 produce 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 full featured lexer and parser).

⟨ Define the bootstrapping mode 2 ⟩ =

```
\newwrite\tokendefs %   ▷ token list ◁
\newwrite\stlist    %   ▷ flex state list ◁
\newwrite\gindex    %   ▷ index entries ◁

\def\modebootstrap{%
    \edef\bstrapparser{dyytab.tex}%
    \bootstrapmodetrue
    \def\bootstraplexersetup{%
        \input bo.tok%
        \let\yylexreturn\yylexreturnbootstrap    ▷ only return tokens whose value is known ◁
        %\let\yylexreturn\yylexreturnregular    ▷ should also work ◁
    }%
    \input yybootstrap.sty%
}
```

This code is used in section 9.

3 Namespaces and modes

Every parser/lexer pair (as well as some other macros) operates within a dedicated *namespace*. This simply means that the macros that output token values, switch lexer states and access various tables 'tack on' the 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. The `[main]` namespace is established for the parser that does the typesetting of the grammar. 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. 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]`. The parser based on a subset of the full `bison` grammar describing prologue declarations uses the `[prologue]` namespace. The `[index]` namespace is used for typesetting the

index entries and is not necessarily associated with any parser or lexer.

⟨ Begin namespace setup 3 ⟩ =
```
\def\indexpseudonamespace{[index]}
\let\parsernamespace\empty
```
This code is used in section 9.

4 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 4 ⟩ =
```
\newtoks\ldcmds

\def\modenormal{%
    \def\drvname{bo}%
    \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%
    \let\hostparsernamespace\ldnamespace
  ▷ the namespace where tokens are looked up for typesetting purposes ◁
}
```
See also sections 5 and 6.

This code is used in section 9.

5 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 4 ⟩ + =
```
\def\ldparserinit{%
    \basicparserinit
    \includestackptr=\@ne
    \versnodenesting=\z@
    \ldcmds{}%
    \yyBEGIN{INITIAL}%
}
```

6 This is the `ld` parser invocation routine. It is coded according to a straightforward sequence initialize-invoke-execute-or-fall back.

⟨ Define the normal mode 4 ⟩ + =
```
\expandafter\def\csname parserstack[l]\endcsname#1#2{%
    \toldparser\ldparserinit\yyparse#1\yyeof\yyeof\endparseinput\endparse
    \ifyyparsefail %    ▷ revert to generic macros if parsing failed ◁
        \yybreak{\message{parsing failed ...}#2}%
    \else          %    ▷ stage three, process the parsed table ◁
        \yybreak{%
            \message{commands: \the\ldcmds}%
            {%
                \restorecslist{ld-display}\ldunion
                \the\ldcmds
                \par
                \vskip-\baselineskip
                \the\lddisplay
             }%
```

```
        }%
    \yycontinue
}
```

7 ⟨ Initialize the active mode 7 ⟩ =

```
\ifx\modeactive\UNDEFINED
   \def\modeactive{\modenormal}
\fi

\modeactive

\ifbootstrapmode\else
    ⟨ Initialize ld parsers 8 ⟩
    ⟨ Modified name parser for ld grammar 83 ⟩
\fi
```

This code is used in section 9.

8 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 nonterminals (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 $\texttt{ASSERT}_{\texttt{K}}$. 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 8 ⟩ =

```
\genericparser
    name: ld,
    ptables: ptab.tex,
    ltables: ltab.tex,
    tokens: {},
    asetup: {},
    dsetup: {},
    rsetup: {},
    optimization: {};%
\genericprettytokens
    namespace: ld,
    tokens: ldp.tok,
    correction: ltokenset.sty,
    host: ld;%
```

This code is used in section 7.

9 The macros are collected in a single file included at the beginning of this documentation.

⟨ ldman.stx 9 ⟩ =

```
⟨ Set up the generic parser machinery 1 ⟩
⟨ Begin namespace setup 3 ⟩
⟨ Define the bootstrapping mode 2 ⟩
⟨ Define the normal mode 4 ⟩
⟨ Additional macros for the ld lexer/parser 62 ⟩
⟨ Initialize the active mode 7 ⟩
```

10 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` 10 ⟩ =

```
⟨ ld parser C preamble 12 ⟩
⟨ ld parser bison options 11 ⟩
⟨union⟩      ⟨ Union of grammar parser types 13 ⟩
⟨ ld parser C postamble 14 ⟩
⟨ Token and type declarations 16 ⟩

⟨ ld parser productions 15 ⟩
```

11 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 11 ⟩ =

```
⟨token table⟩ ⋆
⟨parse.trace⟩ ⋆   (set as ⟨debug⟩)
⟨start⟩           script_file
```

This code is used in section 10.

12 ⟨ `ld` parser C preamble 12 ⟩ =

This code is used in section 10.

13 ⟨ Union of grammar parser types 13 ⟩ =

This code is used in section 10.

14 ⟨ `ld` parser C postamble 14 ⟩ =

```
#define YYPRINT(file, type, value) yyprint (file, type, value)
  static void yyprint(FILE *file, int type, YYSTYPE value)
  { }
```

This code is used in section 10.

15 ⟨ `ld` parser productions 15 ⟩ =

```
⟨ GNU ld script rules 21 ⟩
⟨ Grammar rules 18 ⟩
```

This code is used in section 10.

16 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 16 ⟩ =

INT	name	name_{L}	end
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	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	$\text{VERSION}_{\text{K}}$	INPUT_VERSION_SCRIPT
KEEP	ONLY_IF_RO	ONLY_IF_RW	SPECIAL
INPUT_SECTION_FLAGS	ALIGN_WITH_INPUT	EXCLUDE_FILE	CONSTANT
INPUT_DYNAMIC_LIST			

⟨**right**⟩ $\stackrel{+}{\Leftarrow}$ $\stackrel{-}{\Leftarrow}$ $\stackrel{\times}{\Leftarrow}$ $\stackrel{\div}{\Leftarrow}$ $\Leftarrow$ $\stackrel{\leqslant}{\Leftarrow}$ $\stackrel{\geqslant}{\Leftarrow}$ $\stackrel{\&}{\Leftarrow}$ $\stackrel{\vee}{\Leftarrow}$? : unary

⟨**left**⟩ $\vee$ $\wedge$ | $\oplus$ & $=$ $\neq$ $<$ $>$ $\leqslant$ $\geqslant$ $\ll$ $\gg$ $+$ $-$ * / $\div$ (

This code is used in section 10.

17 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 idiosynchasies 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 implements a way of handling 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 17 ⟩ =

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*

18 ⟨ Grammar rules 18 ⟩ =

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

See also sections 31, 32, 33, 36, 37, 40, 41, 42, 43, 45, 46, 49, 50, 52, and 55.

This code is used in section 15.

19 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 19 ⟩ =

defsym_expr :

∘ `\ldlex@defsym`

name ⇐ *exp* `\ldlex@popstate`

20 *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 20 ⟩ =

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*

`ENDWORD`

`PUBLIC` name ⇐ *exp* | `PUBLIC` name `,` *exp* | `PUBLIC` name *exp*

`FORMAT` name

`SECT` name `,` *exp* | `SECT` name *exp* | `SECT` name ⇐ *exp*

$\texttt{ALIGN}_{\text{K}}$ name ⇐ *exp* | $\texttt{ALIGN}_{\text{K}}$ 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 29 ⟩

`START` name

∘

ordernamelist : *ordernamelist* `,` name | *ordernamelist* name | ∘

mri_load_name_list : name | *mri_load_name_list* `,` name

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

mri_abs_name_list : name | *mri_abs_name_list* , name

casesymlist : ∘ | name | *casesymlist* , name

21 *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 21 ⟩ =

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 22 and 24.

This code is used in section 15.

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

⟨ GNU `ld` script rules 21 ⟩ + =

script_file :

∘	`\ldlex@both`
ifile_list	$\pi_2(\Upsilon_2) \mapsto$ `\ldcmds\ldlex@popstate`

ifile_list :

ifile_list ifile_p$_1$	⟨ Add the next command 23 ⟩
∘	$\Upsilon \leftarrow \langle$`{}{}`$\rangle$

23 ⟨ Add the next command 23 ⟩ =

$\pi_1(\Upsilon_1) \mapsto v_a \pi_2(\Upsilon_1) \mapsto v_b$
$\pi_1(\Upsilon_2) \mapsto v_c \pi_2(\Upsilon_2) \mapsto v_d$
`\yytoksempty{`v_b`}{`$\Upsilon \leftarrow \langle$val $\Upsilon_2\rangle$`}{`$\Upsilon \leftarrow \langle${$\llcorner v_c \lrcorner$}{$\llcorner v_b \lrcorner^{\mathrm{nox}}$`\ldcommandseparator{`$\llcorner v_a \lrcorner$}{$\llcorner v_c \lrcorner$}$\llcorner v_d \lrcorner$}$\rangle$ `}`

This code is used in section 22.

24 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 21 ⟩ + =

***ifile_p*$_1$** :

memory	$\Upsilon \leftarrow \langle$`{mem}{`val Υ_1`}`$\rangle$
sections	$\Upsilon \leftarrow \langle$`{sect}{`val Υ_1`}`$\rangle$
phdrs	
startup	
high_level_library	

low_level_library
floating_point_support
statement_anywhere $\Upsilon \leftarrow \langle$ { stmt }{ $^{\text{nox}}$\ldstatement { val Υ_1 } }$\rangle$
version
; $\Upsilon \leftarrow \langle$ { 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 28 ⟩
ifile_list end ⟨ Add an INCLUDE statement 30 ⟩
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*$_1$ } $\Upsilon \leftarrow \langle^{\text{nox}}$\ldsections { val Υ_3 }$\rangle$

***sec_or_group_p*$_1$** :
sec_or_group_p$_1$ *section* ⟨ Add the next section 25 ⟩
sec_or_group_p$_1$ *statement_anywhere* ⟨ Add the next statement 26 ⟩
○ $\Upsilon \leftarrow \langle\rangle$

statement_anywhere :
ENTRY (name)
assignment end ⟨ Carry on 27 ⟩
ASSERT$_{\text{K}}$ \ldlex@expression
(*exp* , name) \ldlex@popstate

25 ⟨ Add the next section 25 ⟩ =
\yytoksempty { Υ_1 }{ $\Upsilon \leftarrow \langle$ val $\Upsilon_2\rangle$ }{ $\Upsilon \leftarrow \langle$ val Υ_1 $^{\text{nox}}$\ldsectionseparator val $\Upsilon_2\rangle$ }
This code is used in section 24.

26 ⟨ Add the next statement 26 ⟩ =
```
\yytoksempty{ ϒ1 }{ ϒ ← ⟨ val ϒ2 ⟩ }{ ϒ ← ⟨ val ϒ1 nox \ldsectionseparator \ldstatement{ val ϒ2 } ⟩ }
```
This code is used in section 24.

27 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 27 ⟩ =
```
ϒ ← ⟨ val ϒ1 ⟩
```
This code is used in sections 24, 31, 33, 36, and 40.

28 ⟨ Peek at a file 28 ⟩ =
```
\ldlex@script
\ldfile@open@command@file{ ϒ2 }
```
This code is used in sections 20, 24, 33, 40, and 46.

29 ⟨ Close the file 29 ⟩ =
```
ϒ ← ⟨ nox \ldinclude{ val ϒ2 } ⟩ \ldlex@popstate
```
This code is used in sections 20, 33, 40, and 46.

30 ⟨ Add an `INCLUDE` statement 30 ⟩ =
```
ϒ ← ⟨{ inc }{ nox \ldinclude{ val ϒ2 } } ⟩ \ldlex@popstate
```
This code is used in section 24.

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

⟨ Grammar rules 18 ⟩ + =

wildcard_name :

name	⟨ Carry on 27 ⟩
`*`	ϒ ← ⟨{ * }{ * }⟩
`?`	ϒ ← ⟨{ ? }{ ? }⟩

32 ⟨ Grammar rules 18 ⟩ + =

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* `,`opt *wildcard_spec*	ϒ ← ⟨ val ϒ1 nox \ldspace nox \ldregexp{ val ϒ2 } ⟩
wildcard_spec	ϒ ← ⟨ nox \ldregexp{ val ϒ1 } ⟩

input_section_spec_no_keep :
name
sect_flags name
[*file*_name_*list*]
sect_flags [*file*_name_*list*]
wildcard_spec (*file*_name_*list*) $\Upsilon \leftarrow \langle^{\text{nox}}\texttt{\textbackslash ldregexp\{}\,\text{val}\,\Upsilon_1\,\texttt{\}(}\text{val}\,\Upsilon_3\texttt{)}\rangle$
sect_flags wildcard_spec (*file*_name_*list*)

33 ⟨ Grammar rules 18 ⟩ + =

input_section_spec :
input_section_spec_no_keep ⟨ Carry on 27 ⟩
KEEP (
input_section_spec_no_keep) $\Upsilon \leftarrow \langle\texttt{\textbackslash mathop\{\textbackslash hbox\{}\,^{\text{nox}}\texttt{\textbackslash ttl keep\}(}\text{val}\,\Upsilon_4\texttt{)\}}\rangle$

statement :
assignment end
CREATE_OBJECT_SYMBOLS
; $\Upsilon \leftarrow \langle\rangle$
CONSTRUCTORS
SORT_BY_NAME (CONSTRUCTORS)
input_section_spec
length (*mustbe_exp*)
FILL (*fill_exp*)
$\texttt{ASSERT}_{\text{K}}$ \ldlex@expression
(*exp* , name) *end* \ldlex@popstate
INCLUDE *filename* ⟨ Peek at a file 28 ⟩
statement_list$_{\text{opt}}$ end ⟨ Close the file 29 ⟩

statement_list :
statement_list statement ⟨ Attach a statement to a statement list 34 ⟩
statement ⟨ Start a statement list with a statement 35 ⟩

statement_list$_{\text{opt}}$:
∘ $\Upsilon \leftarrow \langle\rangle$
statement_list ⟨ Carry on 27 ⟩

34 ⟨ Attach a statement to a statement list 34 ⟩ =
$\Upsilon \leftarrow \langle\text{val}\,\Upsilon_1\texttt{\textbackslash yytoksempty\{}\,\Upsilon_2\,\texttt{\}\{\}\{\textbackslash yytoksempty\{}\,\Upsilon_1\,\texttt{\}\{\}\{}\,^{\text{nox}}\texttt{\textbackslash ldor\}\{}\,\text{val}\,\Upsilon_2\,\texttt{\}\}}\rangle$
This code is used in section 33.

35 ⟨ Start a statement list with a statement 35 ⟩ =
$\Upsilon \leftarrow \langle\texttt{\textbackslash yytoksempty\{}\,\Upsilon_1\,\texttt{\}\{\}\{}\,\text{val}\,\Upsilon_1\,\texttt{\}\}}\rangle$
This code is used in section 33.

36 ⟨ Grammar rules 18 ⟩ + =

length : QUAD | SQUAD | LONG | SHORT | BYTE
fill_exp : *mustbe_exp* ⟨ Carry on 27 ⟩
fill$_{\text{opt}}$: ⇐ *fill_exp* | ∘ $\Upsilon \leftarrow \langle\rangle$
assign_op :
$\overset{+}{\Leftarrow}$ $\Upsilon_0 = \{\texttt{\textbackslash MRL\{+\{\textbackslash K\}\}}\}$
$\overset{-}{\Leftarrow}$ $\Upsilon_0 = \{\texttt{\textbackslash MRL\{-\{\textbackslash K\}\}}\}$
$\overset{\times}{\Leftarrow}$ | $\overset{\div}{\Leftarrow}$ | $\overset{\ll}{\Leftarrow}$ | $\overset{\gg}{\Leftarrow}$ | $\overset{\&}{\Leftarrow}$ | $\overset{\vee}{\Leftarrow}$ $\Upsilon_0 = \{\texttt{\textbackslash Xorxeq}\}$
end : ; | ,
,$_{\text{opt}}$: , | ∘

37 Assignments are not expressions as in C.

⟨ Grammar rules 18 ⟩ + =

assignment :

name ⇐ *mustbe_exp*	⟨ Process simple assignment 38 ⟩
name *assign_op* *mustbe_exp*	⟨ Process compound assignment 39 ⟩
`HIDDEN` (name ⇐ *mustbe_exp*)	
`PROVIDE` (name ⇐ *mustbe_exp*)	
`PROVIDE_HIDDEN` (name ⇐ *mustbe_exp*)	

38 ⟨ Process simple assignment 38 ⟩ =

$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldassignment{`$^{\text{nox}}$`\ldregexp{`$\,\text{val}\,\Upsilon_1$`}}{\K}{`$\,\text{val}\,\Upsilon_3$`}`$\rangle$

This code is used in section 37.

39 ⟨ Process compound assignment 39 ⟩ =

$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldassignment{`$^{\text{nox}}$`\ldregexp{`$\,\text{val}\,\Upsilon_1$`}}{`$\,\text{val}\,\Upsilon_2$`}{`$\,\text{val}\,\Upsilon_3$`}`$\rangle$

This code is used in section 37.

40 ⟨ Grammar rules 18 ⟩ + =

memory :	
`MEMORY {` *memory_spec_list*$_{\text{opt}}$ `}`	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldmemory{`$\,\text{val}\,\Upsilon_3$`}`$\rangle$
memory_spec_list$_{\text{opt}}$:	
memory_spec_list	⟨ Carry on 27 ⟩
∘	$\Upsilon \leftarrow \langle\rangle$
memory_spec_list :	
memory_spec_list ,$_{\text{opt}}$ *memory_spec*	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \text{val}\,\Upsilon_3 \rangle$
memory_spec	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \rangle$
memory_spec :	
name *attributes*$_{\text{opt}}$: *origin_spec* ↩ ,$_{\text{opt}}$ *length_spec*	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldmemoryspec{`$\,\text{val}\,\Upsilon_1$`}{`$\,\text{val}\,\Upsilon_3$`}{`$\,\text{val}\,\Upsilon_5$`}{`$\,\text{val}\,\Upsilon_7$`}`$\rangle$
`INCLUDE` *filename*	⟨ Peek at a file 28 ⟩
memory_spec_list$_{\text{opt}}$ end	⟨ Close the file 29 ⟩

41 ⟨ Grammar rules 18 ⟩ + =

origin_spec :	
`ORIGIN` ⇐ *mustbe_exp*	$\Upsilon \leftarrow \langle^{\text{nox}}$`\loriginspec{`$\,\text{val}\,\Upsilon_3$`}`$\rangle$
length_spec :	
`LENGTH` ⇐ *mustbe_exp*	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldlengthspec{`$\,\text{val}\,\Upsilon_3$`}`$\rangle$
attributes$_{\text{opt}}$:	
∘	$\Upsilon \leftarrow \langle\rangle$
(*attributes_list*)	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_2 \rangle$
attributes_list :	
attributes_string	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \rangle$
attributes_list *attributes_string*	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1\,^{\text{nox}}$`\ldspace` $\text{val}\,\Upsilon_2 \rangle$
attributes_string :	
name	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldattributes{`$\,\text{val}\,\Upsilon_1$`}`$\rangle$
¬ name	$\Upsilon \leftarrow \langle^{\text{nox}}$`\ldattributesneg{`$\,\text{val}\,\Upsilon_2$`}`$\rangle$
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* ,opt *filename*
∘

floating_point_support :
FLOAT
NOFLOAT

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

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

42 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 18 ⟩ + =

exp :

− *exp*	⟨prec unary⟩	$\Upsilon \leftarrow \langle$ { -val Υ_2 } $\rangle$
(*exp*)		$\Upsilon \leftarrow \langle$ (val Υ_2) $\rangle$
NEXT (*exp*)	⟨prec unary⟩	$\Upsilon \leftarrow \langle$ \hbox { nx \ssf next }(val Υ_3) $\rangle$
¬ *exp*	⟨prec unary⟩	$\Upsilon \leftarrow \langle$ { nox \CM val Υ_2 } $\rangle$
+ *exp*	⟨prec unary⟩	$\Upsilon \leftarrow \langle$ { +val Υ_2 } $\rangle$
not *exp*	⟨prec unary⟩	$\Upsilon \leftarrow \langle$ { nox \R val Υ_2 } $\rangle$
exp × *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \times val $\Upsilon_3 \rangle$
exp / *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \/ val $\Upsilon_3 \rangle$
exp ÷ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \div val $\Upsilon_3 \rangle$
exp + *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 +val $\Upsilon_3 \rangle$
exp − *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 -val $\Upsilon_3 \rangle$
exp ≪ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \ll val $\Upsilon_3 \rangle$
exp ≫ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \gg val $\Upsilon_3 \rangle$
exp = *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 = val $\Upsilon_3 \rangle$
exp ≠ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \not = val $\Upsilon_3 \rangle$
exp ⩽ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \leq val $\Upsilon_3 \rangle$
exp ⩾ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \geq val $\Upsilon_3 \rangle$
exp < *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 <val $\Upsilon_3 \rangle$
exp > *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 >val $\Upsilon_3 \rangle$
exp & *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \AND val $\Upsilon_3 \rangle$
exp ⊕ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \XOR val $\Upsilon_3 \rangle$
exp | *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \OR val $\Upsilon_3 \rangle$
exp ? *exp* : *exp*		⟨ Process a primitive conditional 44 ⟩
exp ∧ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \V val $\Upsilon_3 \rangle$
exp ∨ *exp*		$\Upsilon \leftarrow \langle$ val Υ_1 nox \W val $\Upsilon_3 \rangle$

43

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

⟨ Grammar rules 18 ⟩ + =

exp :
DEFINED (name)
INT
SIZEOF_HEADERS
ALIGNOF (name)

SIZEOF (name)	
ADDR (name)	
LOADADDR (name)	
CONSTANT (name)	
ABSOLUTE (*exp*)	
$\texttt{ALIGN}_{\texttt{K}}$ (*exp*)	$\Upsilon \leftarrow \langle {}^{\texttt{nx}}\texttt{\mathop\{\hbox\{}{}^{\texttt{nx}}\texttt{\ssf align\}\}(}\,\mathrm{val}\,\Upsilon_3)\rangle$
$\texttt{ALIGN}_{\texttt{K}}$ (*exp* , *exp*)	
DATA_SEGMENT_ALIGN (*exp* , *exp*)	
DATA_SEGMENT_RELRO_END (*exp* , *exp*)	
DATA_SEGMENT_END (*exp*)	
SEGMENT_START (name , *exp*)	
BLOCK (*exp*)	
name	$\Upsilon \leftarrow \langle {}^{\texttt{nox}}\texttt{\ldregexp\{}\,\mathrm{val}\,\Upsilon_1\,\texttt{\}}\rangle$
$\texttt{MAX}_{\texttt{K}}$ (*exp* , *exp*)	
$\texttt{MIN}_{\texttt{K}}$ (*exp* , *exp*)	
$\texttt{ASSERT}_{\texttt{K}}$ (*exp* , name)	
ORIGIN (name)	
LENGTH (name)	
LOG2CEIL (*exp*)	

44 ⟨ Process a primitive conditional 44 ⟩ =

$\Upsilon \leftarrow \langle \texttt{\hbox\{}{}^{\texttt{nx}}\texttt{\ttl do\}\xi(}\,\mathrm{val}\,\Upsilon_1\texttt{)\hbox\{ }{}^{\texttt{nx}}\texttt{\ttl where\}}\quad \texttt{\{let }{}^{\texttt{nx}}\texttt{\{}{}^{\texttt{nx}}\texttt{\bigbracedel}{}^{\texttt{nx}}\texttt{\xi(x) =}$
${}^{\texttt{nx}}\texttt{\cases\{}\,\mathrm{val}\,\Upsilon_5\ \texttt{if }{}^{\bullet}\texttt{(x = 0)\cr}\,\mathrm{val}\,\Upsilon_3\ \texttt{if }{}^{\bullet}\texttt{(x}{}^{\texttt{nx}}\texttt{\not = 0)\}\}\}$

This code is used in section 42.

45 ⟨ Grammar rules 18 ⟩ + =

memspec_at$_{\mathrm{opt}}$:	
AT > name	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
at$_{\mathrm{opt}}$:	
AT (*exp*)	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
align$_{\mathrm{opt}}$:	
$\texttt{ALIGN}_{\texttt{K}}$ (*exp*)	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
align_with_input$_{\mathrm{opt}}$:	
ALIGN_WITH_INPUT	$\Upsilon \leftarrow \langle$align with input$\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
subalign$_{\mathrm{opt}}$:	
SUBALIGN (*exp*)	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$
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$\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

46 *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 18 ⟩ + =

section :

name	`\ldlex@expression`
exp_with_type$_{\rm opt}$ *at*$_{\rm opt}$ *align*$_{\rm opt}$ *align_with_input*$_{\rm opt}$ ↩	
subalign$_{\rm opt}$	`\ldlex@popstate \ldlex@script`
sect_constraint `{`	
statement_list$_{\rm opt}$ `}`	`\ldlex@popstate \ldlex@expression`
memspec$_{\rm opt}$ *memspec_at*$_{\rm opt}$ *phdr*$_{\rm opt}$ *fill*$_{\rm opt}$	`\ldlex@popstate`
`,`$_{\rm opt}$	⟨ Record a named section 47 ⟩
`OVERLAY`	`\ldlex@expression`
exp_without_type$_{\rm opt}$ *nocrossrefs*$_{\rm opt}$ *at*$_{\rm opt}$ *subalign*$_{\rm opt}$	`\ldlex@popstate \ldlex@script`
`{`	
overlay_section `}`	`\ldlex@popstate \ldlex@expression`
memspec$_{\rm opt}$ *memspec_at*$_{\rm opt}$ *phdr*$_{\rm opt}$ *fill*$_{\rm opt}$	`\ldlex@popstate`
`,`$_{\rm opt}$	⟨ Record an overlay section 48 ⟩
`GROUP`	`\ldlex@expression`
exp_with_type$_{\rm opt}$	`\ldlex@popstate`
`{` *sec_or_group_p*$_1$ `}`	
`INCLUB` *filename*	⟨ Peek at a file 28 ⟩
sec_or_group_p$_1$ end	⟨ Close the file 29 ⟩

47 ⟨ Record a named section 47 ⟩ =

$\Upsilon \leftarrow \langle^{\rm nx}$`\ldnamedsection{`val Υ_1`}{`val Υ_3`}{`val Υ_4`}`
`{{`val Υ_5`}{`val Υ_6`}{`val Υ_7`}}` ▷ alignment ◁
`{`val Υ_9`}{`val Υ_{12}`}`
`{{`val Υ_{15}`}{`val Υ_{16}`}{`val Υ_{17}`}{`val Υ_{18}`}}`$\rangle$ ▷memory specifiers ◁

This code is used in section 46.

48 ⟨ Record an overlay section 48 ⟩ =

This code is used in section 46.

49 ⟨ Grammar rules 18 ⟩ + =

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* `)`	$\Upsilon \leftarrow \langle^{\rm nox}$`\ldtype{`val Υ_2`}`$\rangle$
`()`	$\Upsilon \leftarrow \langle^{\rm nox}$`\ldtype{}`$\rangle$
○	$\Upsilon \leftarrow \langle\rangle$

50 *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 18 ⟩ + =

exp_with_type$_{\rm opt}$:

exp atype :	$\Upsilon \leftarrow \langle${`}{`val Υ_1`}{}{}{`val Υ_2`}`$\rangle$
atype :	$\Upsilon \leftarrow \langle${`}{}{}{}{`val Υ_1`}`$\rangle$
`BIND (` *exp* `)` *atype* :	$\Upsilon \leftarrow \langle${`bind}{`val Υ_3`}{}{}{`val Υ_5`}`$\rangle$
`BIND (` *exp* `) BLOCK (` *exp* `)` *atype* :	$\Upsilon \leftarrow \langle${`bind}{`val Υ_3`}{block}{`val Υ_7`}{`val Υ_9`}`$\rangle$

exp_without_type$_{\rm opt}$:

exp :	
:	

nocrossrefs$_{\rm opt}$:

○

NOCROSSREFS

memspec$_{\rm opt}$:

> name	$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_2 \rangle$
○	$\Upsilon \leftarrow \langle\rangle$

phdr$_{\rm opt}$:

○	$\Upsilon \leftarrow \langle\rangle$
phdr$_{\rm opt}$: name	⟨ Add another pheader 51 ⟩

overlay_section :

○	
overlay_section name	\ldlex@script
{ *statement_list*$_{\rm opt}$ }	\ldlex@popstate \ldlex@expression
phdr$_{\rm opt}$ *fill*$_{\rm opt}$	\ldlex@popstate
,$_{\rm opt}$	

51 ⟨ Add another pheader 51 ⟩ =

\yytoksempty{ Υ_1 }{ $\Upsilon \leftarrow \langle$ { val Υ_3 } $\rangle$ }{ $\Upsilon \leftarrow \langle$ val Υ_1 nox\ldor{ val Υ_3 } $\rangle$ }

This code is used in section 50.

52 ⟨ Grammar rules 18 ⟩ + =

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*)	

53 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 53 ⟩ =

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 ;

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

⟨ Version file rules 54 ⟩ =

version_script_file :

∘ `\ldlex@version@file`

vers_nodes `\ldlex@popstate`

55 *This is used within a normal linker script file.*

⟨ Grammar rules 18 ⟩ + =

version :

∘ `\ldlex@version@script`

VERSION$_{\mathrm{K}}$ { *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 ;$_{\mathrm{opt}}$ }

EXTERN name {

vers_defns ;$_{\mathrm{opt}}$ }

GLOBAL

vers_defns ; GLOBAL

LOCAL

vers_defns ; LOCAL

EXTERN

vers_defns ; EXTERN

;$_{\mathrm{opt}}$: ∘ | ;

56 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 56 ⟩ =
  ⟨ ld lexer definitions 60 ⟩
  ...
  ⟨ ld lexer C preamble 58 ⟩
  ...
  ⟨ ld lexer options 57 ⟩

  ⟨ ld token regular expressions 67 ⟩

  void define_all_states(void)
  {
        ⟨ Collect state definitions for the ld lexer 59 ⟩
  }
```

57 ⟨ ld lexer options 57 ⟩ =

```
⟨bison-bridge⟩f ⋆
⟨noyywrap⟩f ⋆
⟨nounput⟩f ⋆
⟨noinput⟩f ⋆
⟨reentrant⟩f ⋆
⟨noyy_top_state⟩f ⋆
⟨debug⟩f ⋆
⟨stack⟩f ⋆
⟨outfile⟩f              "ldl.c"
```

This code is used in section 56.

58 ⟨ ld lexer C preamble 58 ⟩ =

```
#include <stdint.h>
#include <stdbool.h>
```

This code is used in section 56.

59 ⟨ Collect state definitions for the ld lexer 59 ⟩ =

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

This code is used in section 56.

60 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 60 ⟩ =

```
⟨ ld lexer states 61 ⟩
CMDFILENAMECHAR  [_a-zA-Z0-9\/\.\\_\+\$\:\[\]\\\,\=\&\!\<\>\-\~]
CMDFILENAMECHAR1 [_a-zA-Z0-9\/\.\\_\+\$\:\[\]\\\,\=\&\!\<\>\~]
FILENAMECHAR1    [_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 56.

61 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 61 ⟩ =

```
⟨states-s⟩f: SCRIPT
⟨states-s⟩f: EXPRESSION
⟨states-s⟩f: BOTH
⟨states-s⟩f: DEFSYMEXP
⟨states-s⟩f: MRI
⟨states-s⟩f: VERS_START
⟨states-s⟩f: VERS_SCRIPT
⟨states-s⟩f: VERS_NODE
```

This code is used in section 60.

62 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 attenion is the treatment of comments by the lexer. The difficulty of implementing C style comment lexing 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 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 62 ⟩ =

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

See also sections 63, 64, 65, 66, 72, 75, and 78.

This code is used in section 9.

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

⟨ Additional macros for the `ld` lexer/parser 62 ⟩ + =

```
\def\ldcommentskipchars{%
    \ifnum\yycp@=`*
        \yybreak{\let\yyreturn\ldcommentseekslash\yyinput}%
```

[1]) Taken from W. McKeeman's site at http://www.cs.dartmouth.edu/~mckeeman/cs118/assignments/comment.html and adopted to `flex` syntax.

```
                                    %   ▷ * 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
}%
```

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

⟨ Additional macros for the `ld` lexer/parser 62 ⟩ + =

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

65 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`).

⟨ Additional macros for the `ld` lexer/parser 62 ⟩ + =

```
\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
}
```

66 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). The macros below are designed to handle this assignment.

⟨ Additional macros for the `ld` lexer/parser 62 ⟩ + =

```
\def\ldcleanyyeof#1\yylextail{%
    \let\oldyyinput\yyinput
    \def\yyinput\yyeof\yyeof{\let\yyinput\oldyyinput#1\yylextail}%
    \yymatch
}
```

67 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 and algebraic operations come first.

⟨ `ld` token regular expressions 67 ⟩ =

```
<BOTH,SCRIPT,EXPRESSION,VERS_START,VERS_NODE,VERS_SCRIPT>"/*" {\ldcomment continue}
<DEFSYMEXP>"-"                                  {return_c}
<DEFSYMEXP>"+"                                  {return_c}
<DEFSYMEXP>{FILENAMECHAR1}{SYMBOLCHARN}*        {return_v name}
<DEFSYMEXP>"="                                  {return_c}
<MRI,EXPRESSION>"$"([0-9A-Fa-f])+               {⟨ Return an absolute hex constant 69 ⟩}
<MRI,EXPRESSION>([0-9A-Fa-f])+(H|h|X|x|B|b|O|o|D|d) {⟨ Return a constant in a specific radix 70 ⟩}
<SCRIPT,DEFSYMEXP,MRI,BOTH,EXPRESSION>((("$"|0[xX])([0-9A-Fa-f])+)|(([0-9])+))(M|K|m|k)? {
                                                ⟨ Return a constant with a multiplier 71 ⟩}
<BOTH,SCRIPT,EXPRESSION,MRI>"]"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"["                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"<<="               {return_p ⩽=}
<BOTH,SCRIPT,EXPRESSION,MRI>">>="               {return_p ⩾=}
<BOTH,SCRIPT,EXPRESSION,MRI>"||"                {return_p ∨}
<BOTH,SCRIPT,EXPRESSION,MRI>"=="                {return_p =}
<BOTH,SCRIPT,EXPRESSION,MRI>"!="                {return_p ≠}
<BOTH,SCRIPT,EXPRESSION,MRI>">="                {return_p ⩾}
<BOTH,SCRIPT,EXPRESSION,MRI>"<="                {return_p ⩽}
<BOTH,SCRIPT,EXPRESSION,MRI>"<<"                {return_p ≪}
<BOTH,SCRIPT,EXPRESSION,MRI>">>"                {return_p ≫}
<BOTH,SCRIPT,EXPRESSION,MRI>"+="                {return_p +=}
<BOTH,SCRIPT,EXPRESSION,MRI>"-="                {return_p -=}
<BOTH,SCRIPT,EXPRESSION,MRI>"*="                {return_p ×=}
<BOTH,SCRIPT,EXPRESSION,MRI>"/="                {return_p ÷=}
<BOTH,SCRIPT,EXPRESSION,MRI>"&="                {return_p &=}
<BOTH,SCRIPT,EXPRESSION,MRI>"|="                {return_p ⩔=}
<BOTH,SCRIPT,EXPRESSION,MRI>"&&"                {return_p ∧}
<BOTH,SCRIPT,EXPRESSION,MRI>">"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>","                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"&"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"|"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"~"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"!"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"?"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"*"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"+"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"-"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"/"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"%"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"<"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"="                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"}"                 {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"{"                 {return_c}
```

```
<BOTH,SCRIPT,EXPRESSION,MRI>")"                {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>"("                {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>":"                {return_c}
<BOTH,SCRIPT,EXPRESSION,MRI>";"                {return_c}
```

See also sections 68 and 76.

This code is used in section 56.

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

⟨ ld token regular expressions 67 ⟩ + =

```
<BOTH,SCRIPT>"MEMORY"                          {return_p MEMORY}
<BOTH,SCRIPT>"REGION_ALIAS"                    {return_p REGION_ALIAS}
<BOTH,SCRIPT>"LD_FEATURE"                      {return_p LD_FEATURE}
<BOTH,SCRIPT,EXPRESSION>"ORIGIN"               {return_p ORIGIN}
<BOTH,SCRIPT>"VERSION"                         {return_p VERSION_K}
<EXPRESSION,BOTH,SCRIPT>"BLOCK"                {return_p BLOCK}
<EXPRESSION,BOTH,SCRIPT>"BIND"                 {return_p BIND}
<BOTH,SCRIPT,EXPRESSION>"LENGTH"               {return_p LENGTH}
<EXPRESSION,BOTH,SCRIPT>"ALIGN"                {return_p ALIGN_K}
<EXPRESSION,BOTH,SCRIPT>"DATA_SEGMENT_ALIGN"   {return_p DATA_SEGMENT_ALIGN}
<EXPRESSION,BOTH,SCRIPT>"DATA_SEGMENT_RELRO_END" {return_p DATA_SEGMENT_RELRO_END}
<EXPRESSION,BOTH,SCRIPT>"DATA_SEGMENT_END"     {return_p DATA_SEGMENT_END}
<EXPRESSION,BOTH,SCRIPT>"ADDR"                 {return_p ADDR}
<EXPRESSION,BOTH,SCRIPT>"LOADADDR"             {return_p LOADADDR}
<EXPRESSION,BOTH,SCRIPT>"ALIGNOF"              {return_p ALIGNOF}
<EXPRESSION,BOTH>"MAX"                         {return_p MAX_K}
<EXPRESSION,BOTH>"MIN"                         {return_p MIN_K}
<EXPRESSION,BOTH>"LOG2CEIL"                    {return_p LOG2CEIL}
<EXPRESSION,BOTH,SCRIPT>"ASSERT"               {return_p ASSERT_K}
<BOTH,SCRIPT>"ENTRY"                           {return_p ENTRY}
<BOTH,SCRIPT,MRI>"EXTERN"                      {return_p EXTERN}
<EXPRESSION,BOTH,SCRIPT>"NEXT"                 {return_p NEXT}
<EXPRESSION,BOTH,SCRIPT>"sizeof_headers"       {return_p SIZEOF_HEADERS}
<EXPRESSION,BOTH,SCRIPT>"SIZEOF_HEADERS"       {return_p SIZEOF_HEADERS}
<EXPRESSION,BOTH,SCRIPT>"SEGMENT_START"        {return_p SEGMENT_START}
<BOTH,SCRIPT>"MAP"                             {return_p MAP}
<EXPRESSION,BOTH,SCRIPT>"SIZEOF"               {return_p SIZEOF}
<BOTH,SCRIPT>"TARGET"                          {return_p TARGET_K}
<BOTH,SCRIPT>"SEARCH_DIR"                      {return_p SEARCH_DIR}
<BOTH,SCRIPT>"OUTPUT"                          {return_p OUTPUT}
<BOTH,SCRIPT>"INPUT"                           {return_p INPUT}
<EXPRESSION,BOTH,SCRIPT>"GROUP"                {return_p GROUP}
<EXPRESSION,BOTH,SCRIPT>"AS_NEEDED"            {return_p AS_NEEDED}
<EXPRESSION,BOTH,SCRIPT>"DEFINED"              {return_p DEFINED}
<BOTH,SCRIPT>"CREATE_OBJECT_SYMBOLS"           {return_p CREATE_OBJECT_SYMBOLS}
<BOTH,SCRIPT>"CONSTRUCTORS"                    {return_p CONSTRUCTORS}
<BOTH,SCRIPT>"FORCE_COMMON_ALLOCATION"         {return_p FORCE_COMMON_ALLOCATION}
<BOTH,SCRIPT>"INHIBIT_COMMON_ALLOCATION"       {return_p INHIBIT_COMMON_ALLOCATION}
<BOTH,SCRIPT>"SECTIONS"                        {return_p SECTIONS}
<BOTH,SCRIPT>"INSERT"                          {return_p INSERT_K}
<BOTH,SCRIPT>"AFTER"                           {return_p AFTER}
<BOTH,SCRIPT>"BEFORE"                          {return_p BEFORE}
<BOTH,SCRIPT>"FILL"                            {return_p FILL}
<BOTH,SCRIPT>"STARTUP"                         {return_p STARTUP}
<BOTH,SCRIPT>"OUTPUT_FORMAT"                   {return_p OUTPUT_FORMAT}
<BOTH,SCRIPT>"OUTPUT_ARCH"                     {return_p OUTPUT_ARCH}
```

```
<BOTH,SCRIPT>"HLL"                          {return_p HLL}
<BOTH,SCRIPT>"SYSLIB"                       {return_p SYSLIB}
<BOTH,SCRIPT>"FLOAT"                        {return_p FLOAT}
<BOTH,SCRIPT>"QUAD"                         {return_p QUAD}
<BOTH,SCRIPT>"SQUAD"                        {return_p SQUAD}
<BOTH,SCRIPT>"LONG"                         {return_p LONG}
<BOTH,SCRIPT>"SHORT"                        {return_p SHORT}
<BOTH,SCRIPT>"BYTE"                         {return_p BYTE}
<BOTH,SCRIPT>"NOFLOAT"                      {return_p NOFLOAT}
<EXPRESSION,BOTH,SCRIPT>"NOCROSSREFS"       {return_p NOCROSSREFS}
<BOTH,SCRIPT>"OVERLAY"                      {return_p OVERLAY}
<BOTH,SCRIPT>"SORT_BY_NAME"                 {return_p SORT_BY_NAME}
<BOTH,SCRIPT>"SORT_BY_ALIGNMENT"            {return_p SORT_BY_ALIGNMENT}
<BOTH,SCRIPT>"SORT"                         {return_p SORT_BY_NAME}
<BOTH,SCRIPT>"SORT_BY_INIT_PRIORITY"        {return_p SORT_BY_INIT_PRIORITY}
<BOTH,SCRIPT>"SORT_NONE"                    {return_p SORT_NONE}
<EXPRESSION,BOTH,SCRIPT>"NOLOAD"            {return_p NOLOAD}
<EXPRESSION,BOTH,SCRIPT>"DSECT"             {return_p DSECT}
<EXPRESSION,BOTH,SCRIPT>"COPY"              {return_p COPY}
<EXPRESSION,BOTH,SCRIPT>"INFO"              {return_p INFO}
<EXPRESSION,BOTH,SCRIPT>"OVERLAY"           {return_p OVERLAY}
<EXPRESSION,BOTH,SCRIPT>"ONLY_IF_RO"        {return_p ONLY_IF_RO}
<EXPRESSION,BOTH,SCRIPT>"ONLY_IF_RW"        {return_p ONLY_IF_RW}
<EXPRESSION,BOTH,SCRIPT>"SPECIAL"           {return_p SPECIAL}
<BOTH,SCRIPT>"o"                            {return_p ORIGIN}
<BOTH,SCRIPT>"org"                          {return_p ORIGIN}
<BOTH,SCRIPT>"l"                            {return_p LENGTH}
<BOTH,SCRIPT>"len"                          {return_p LENGTH}
<EXPRESSION,BOTH,SCRIPT>"INPUT_SECTION_FLAGS"  {return_p INPUT_SECTION_FLAGS}
<EXPRESSION,BOTH,SCRIPT>"INCLUDE"           {return_p INCLUDE}
<BOTH,SCRIPT>"PHDRS"                        {return_p PHDRS}
<EXPRESSION,BOTH,SCRIPT>"AT"                {return_p AT}
<EXPRESSION,BOTH,SCRIPT>"ALIGN_WITH_INPUT"  {return_p ALIGN_WITH_INPUT}
<EXPRESSION,BOTH,SCRIPT>"SUBALIGN"          {return_p SUBALIGN}
<EXPRESSION,BOTH,SCRIPT>"HIDDEN"            {return_p HIDDEN}
<EXPRESSION,BOTH,SCRIPT>"PROVIDE"           {return_p PROVIDE}
<EXPRESSION,BOTH,SCRIPT>"PROVIDE_HIDDEN"    {return_p PROVIDE_HIDDEN}
<EXPRESSION,BOTH,SCRIPT>"KEEP"              {return_p KEEP}
<EXPRESSION,BOTH,SCRIPT>"EXCLUDE_FILE"      {return_p EXCLUDE_FILE}
<EXPRESSION,BOTH,SCRIPT>"CONSTANT"          {return_p CONSTANT}
<MRI>"#".*\n?                               {continue}
<MRI>"\n"                                   {return_p NEWLINE}
<MRI>"*".*                                  {continue}
<MRI>";".*                                  {continue}
<MRI>"END"                                  {return_p ENDWORD}
<MRI>"ALIGNMOD"                             {return_p ALIGNMOD}
<MRI>"ALIGN"                                {return_p ALIGN_K}
<MRI>"CHIP"                                 {return_p CHIP}
<MRI>"BASE"                                 {return_p BASE}
<MRI>"ALIAS"                                {return_p ALIAS}
<MRI>"TRUNCATE"                             {return_p TRUNCATE}
<MRI>"LOAD"                                 {return_p LOAD}
<MRI>"PUBLIC"                               {return_p PUBLIC}
<MRI>"ORDER"                                {return_p ORDER}
<MRI>"NAME"                                 {return_p NAMEWORD}
<MRI>"FORMAT"                               {return_p FORMAT}
<MRI>"CASE"                                 {return_p CASE}
```

```
<MRI>"START"                                    {return_p START}
<MRI>"LIST".*                                   {return_p LIST}
<MRI>"SECT"                                     {return_p SECT}
<EXPRESSION,BOTH,SCRIPT,MRI>"ABSOLUTE"          {return_p ABSOLUTE}
<MRI>"end"                                      {return_p ENDWORD}
<MRI>"alignmod"                                 {return_p ALIGNMOD}
<MRI>"align"                                    {return_p ALIGN_K}
<MRI>"chip"                                     {return_p CHIP}
<MRI>"base"                                     {return_p BASE}
<MRI>"alias"                                    {return_p ALIAS}
<MRI>"truncate"                                 {return_p TRUNCATE}
<MRI>"load"                                     {return_p LOAD}
<MRI>"public"                                   {return_p PUBLIC}
<MRI>"order"                                    {return_p ORDER}
<MRI>"name"                                     {return_p NAMEWORD}
<MRI>"format"                                   {return_p FORMAT}
<MRI>"case"                                     {return_p CASE}
<MRI>"extern"                                   {return_p EXTERN}
<MRI>"start"                                    {return_p START}
<MRI>"list".*                                   {return_p LIST}
<MRI>"sect"                                     {return_p SECT}
<EXPRESSION,BOTH,SCRIPT,MRI>"absolute"          {return_p ABSOLUTE}
<MRI>{FILENAMECHAR1}{NOCFILENAMECHAR}*          {return_v name}
<BOTH>{FILENAMECHAR1}{FILENAMECHAR}*            {return_v name}
<BOTH>"-l"{FILENAMECHAR}+                       {return_v name}
<EXPRESSION>{FILENAMECHAR1}{NOCFILENAMECHAR}*   {return_v name}
<EXPRESSION>"-l"{NOCFILENAMECHAR}+              {return_v name}
<SCRIPT>{WILDCHAR}*                             {⟨ Skip a possible comment and return a name 73 ⟩}
<EXPRESSION,BOTH,SCRIPT,VERS_NODE>"\""[^\"]*"\"" {⟨ Return the name inside quotes 74 ⟩}
<BOTH,SCRIPT,EXPRESSION>"\n"                    {continue}
<MRI,BOTH,SCRIPT,EXPRESSION>[ \t\r]+            {continue}
<VERS_NODE,VERS_SCRIPT>[:,;]                    {return_c}
<VERS_NODE>global                               {return_p GLOBAL}
<VERS_NODE>local                                {return_p LOCAL}
<VERS_NODE>extern                               {return_p EXTERN}
<VERS_NODE>{V_IDENTIFIER}                       {return_v VERS_IDENTIFIER}
<VERS_SCRIPT>{V_TAG}                            {return_v VERS_TAG}
<VERS_START>"{"                                 {\yyBEGIN{VERS_SCRIPT}return_c}
```

69 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 69 ⟩ =

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

This code is used in section 67.

70 ⟨ Return a constant in a specific radix 70 ⟩ =

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

This code is used in section 67.

71 ⟨ Return a constant with a multiplier 71 ⟩ =

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

This code is used in section 67.

72 ⟨ Additional macros for the `ld` lexer/parser 62 ⟩ + =

```
\def\matchcomment@#1/*#2\yyeof#3#4{%
    \yystringempty{#1}{#3}{#4}%
}
\def\matchcomment#1{%
    \expandafter\matchcomment@\the#1/*\yyeof
}
\def\ldstripquotes@"#1"\yyeof{#1}
\def\ldstripquotes#1{%
    \yytext\expandafter\expandafter\expandafter
           {\expandafter\ldstripquotes@\the\yytext\yyeof}%
    \yytextpure\expandafter\expandafter\expandafter
           {\expandafter\ldstripquotes@\the\yytextpure\yyeof}%
}
```

73 *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 73 ⟩ =

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

This code is used in section 68.

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

⟨ Return the name inside quotes 74 ⟩ =

```
\ldstripquotes return_v name
```

This code is used in section 68.

75 ⟨ Additional macros for the `ld` lexer/parser 62 ⟩ + =

```
\newcount\versnodenesting
\newcount\includestackptr
```

76 Some syntax specific to version scripts.

⟨ `ld` token regular expressions 67 ⟩ + =

```
<VERS_SCRIPT>"{"                                  {\yyBEGIN{VERS_NODE}\versnodenesting = 0_R return_c}
<VERS_SCRIPT>"}"                                  {return_c}
<VERS_NODE>"{"                                    {add \versnodenesting 1_R  return_c}
<VERS_NODE>"}"                                    {add \versnodenesting -1_R
                                                   if_ω \versnodenesting < 0_R
                                                        \yyBEGIN{VERS_SCRIPT}
                                                   fi
                                                   return_c}
<VERS_START,VERS_NODE,VERS_SCRIPT>[\n]            {continue}
<VERS_START,VERS_NODE,VERS_SCRIPT>#.*             {continue}
<VERS_START,VERS_NODE,VERS_SCRIPT>[ \t\r]+        {continue}

<<EOF>>                                           {⟨ Process the end of (possibly included) file 77 ⟩}

<SCRIPT,MRI,VERS_START,VERS_SCRIPT,VERS_NODE>.    {\yycomplain{bad character `val \yytext '
                                                              in script}
                                                   \yyerrterminate}
<EXPRESSION,DEFSYMEXP,BOTH>.                      {\yycomplain{bad character `val \yytext '
                                                              in expression}
                                                   \yyerrterminate}
```

77 ⟨ Process the end of (possibly included) file 77 ⟩ =

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

This code is used in section 76.

78 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 62 ⟩ + =

```
\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
    \appendl\yytextseen{\noexpand\yyeof\noexpand\yyeof}%
    \yytextbackuptrue
}

\def\ldlex@filename{}
```

79 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**	**origin**	**length**
	RAM	xrw	$2000\,0000_{16}$	20 K
	FLASH	rx	$800\,0000_{16}$	128 K
	ASH	rx	8 000 000	128 K
	CLASH	rx	700 000	128 K
	ASH	rx	$800\,0000_{16}$	128 K
	CLASH	rx	$70\,0000_{01}$	128 K
	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 79 ⟩ =

include `file.ld`

memory		**attributes**	**origin**	**length**
	RAM	xrw	$2000\,0000_{16}$	20 K
	FLASH	rx	$800\,0000_{16}$	128 K
	ASH	rx	8 000 000	128 K
	CLASH	rx	700 000	128 K
	ASH	rx	$800\,0000_{16}$	128 K
	CLASH	rx	$70\,0000_{01}$	128 K
	include `file.mem`			

`_estack` ⇐ $2000\,5000_{16}$

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

sections	`.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				
	`.text`			**in** `FLASH` **as** `RAM`	
	. ⇐ align(4)				
	*(`.text`)				
	*(`.text.*`)				
	*(`.rodata`)				
	*(`.rodata*`)				
	*(`.glue_7`)				
	*(`.glue_7t`)				
	. ⇐ align(4)				
	`_etext` ⇐ . + 8				
	`_sidata` ⇐ `_etext`				
	`.data`		**at** `_sidata`	**in** `RAM`	
	. ⇐ align(4)				
	`_sdata` ⇐ .				
	*(`.data`)				
	*(`.data.*`)				
	. ⇐ align(4)				
	`_edata` ⇐ .				
	`.bss`			**in** `RAM`	

```
. ⇐ align(4)
_sbss ⇐ .
*(.bss)
*(COMMON)
. ⇐ align(4)
_ebss ⇐ .
```

80 ⟨ The same example of an `ld` script 80 ⟩ =

```
INCLUDE file.ld

MEMORY
{
  RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K
  FLASH (rx) : ORIGIN = 0x8000000, LENGTH = 128K
  ASH (rx) : ORIGIN = 8000000, LENGTH = 128K
  CLASH (rx) : ORIGIN = 700000, LENGTH = 128K
  ASH (rx) : ORIGIN = $8000000, LENGTH = 128K
  CLASH (rx) : ORIGIN = 700000B, LENGTH = 128K
  INCLUDE file.mem
}

_estack = 0x20005000;
_bstack = a > 0 ? NEXT(11) : 0x19;

SECTIONS
{
    .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

    .text :
    {
        /* skip this comment */;
        . = ALIGN(4);
        *(.text)
        *(.text.*)
        *(.rodata)
        *(.rodata*)
        *(.glue_7)
        *(.glue_7t)
        . = ALIGN(4);
    _etext = . + 8;
    _sidata = _etext;
    } >FLASH AT > RAM

    .data  : AT ( _sidata )
    {
     . = ALIGN(4);
        _sdata = . ;
        *(.data)
        *(.data.*)
     . = ALIGN(4);
    _edata = . ;
    } >RAM
```

```
    .bss :
    {
     . = ALIGN(4);
        _sbss = .;
        *(.bss)
        *(COMMON)
     . = ALIGN(4);
    _ebss = . ;
    } >RAM
}
```

81

82 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 82 ⟩ =

```
.....
⟨ Name parser C preamble 110 ⟩
.....
⟨ Bison options 84 ⟩
⟨union⟩      ⟨ Union of parser types 112 ⟩
.....
⟨ Name parser C postamble 111 ⟩
.....
⟨ Token and types declarations 85 ⟩

⟨ Parser productions 86 ⟩
```

83

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 83 ⟩ =

```
\genericparser
    name: ldsmall,
    ptables: ld_small_tab.tex,
    ltables: ld_small_dfa.tex,
    tokens: {},
    asetup: {},
    dsetup: {},
    rsetup: {\noexpand\savefullstateextra},
    optimization: {};%
\let\otosmallparser\tosmallparser %    ▷ save the old name parser ◁
\let\tosmallparser\toldsmallparser
```

This code is used in section 7.

84

⟨ Bison options 84 ⟩ =

```
⟨token table⟩ ⋆
⟨parse.trace⟩ ⋆    (set as ⟨debug⟩)
⟨start⟩            full_name
```

This code is used in section 82.

85

⟨ Token and types declarations 85 ⟩ =

```
%[a...Z0...9]*          [a...Z0...9]*          opt              suffix_K
[0...9]*                ext                    * or ?
```

This code is used in section 82.

86 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 86 ⟩ =

```
full_name :
    identifier_string suffixes_opt                ⟨ Compose the full name 87 ⟩
    qualifier _ identifier_string suffixes_opt    ⟨ Compose a qualified name 88 ⟩
```

identifier_string :

`%[a...Z0...9]*`	⟨ Attach option name 89 ⟩
`[a...Z0...9]*`	⟨ Start with an identifier 90 ⟩
`\' * or ? \'`	⟨ Start with a quoted string 92 ⟩
`\' . \'`	⟨ Start with a . string 93 ⟩
`\' _ \'`	⟨ Start with an _ string 94 ⟩
incomplete_identifier_string ◇ `[a...Z0...9]*`	⟨ Attach an identifier 96 ⟩

incomplete_identifier_string :

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 97 ⟩
qualified_identifier_string `_` *qualifier*	⟨ Attach qualifier to a name 97 ⟩

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

◦	$\Upsilon \leftarrow \langle\rangle$
`.`	$\Upsilon \leftarrow \langle {}^{\text{nx}}\texttt{\textbackslash dotsp}\, {}^{\text{nx}}\texttt{\textbackslash sfxnone} \rangle$
`.` *suffixes*	⟨ Attach suffixes 101 ⟩
`.` *qualified_suffixes*	⟨ Attach qualified suffixes 102 ⟩
`[0...9]*`	⟨ Attach an integer 98 ⟩
`_ [0...9]*`	⟨ Attach a subscripted integer 99 ⟩
`_` *qualifier*	⟨ Attach a subscripted qualifier 100 ⟩

suffixes :

`[a...Z0...9]*`	⟨ Start with a named suffix 103 ⟩
`[0...9]*`	⟨ Start with a numeric suffix 104 ⟩
suffixes `.`	⟨ Add a dot separator 105 ⟩
suffixes `[a...Z0...9]*`	⟨ Attach a named suffix 107 ⟩
suffixes `[0...9]*`	⟨ Attach integer suffix 106 ⟩
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 108 ⟩
qualifier	⟨ Start suffixes with a qualifier 109 ⟩

qualifier :

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

This code is used in section 82.

87 ⟨ Compose the full name 87 ⟩ =

$\Upsilon \leftarrow \langle \text{val}\, \Upsilon_1 \text{val}\, \Upsilon_2 \rangle$ `\namechars` Υ

This code is used in section 86.

88 ⟨ Compose a qualified name 88 ⟩ =

$\Upsilon \leftarrow \langle \text{val}\, \Upsilon_3 \text{val}\, \Upsilon_4\, {}^{\text{nx}}\texttt{\textbackslash dotsp}\, {}^{\text{nx}}\texttt{\textbackslash qual}\, \text{val}\, \Upsilon_1 \rangle$ `\namechars` Υ

This code is used in section 86.

89 ⟨ Attach option name 89 ⟩ =

$\pi_1(\Upsilon_1) \mapsto v_a$

$\pi_2(\Upsilon_1) \mapsto v_b$

$\Upsilon \leftarrow \langle {}^{\text{nx}}\texttt{\textbackslash optstr\{}\llcorner v_a \lrcorner\texttt{\}\{}\llcorner v_b \lrcorner\texttt{\}} \rangle$

This code is used in section 86.

90 ⟨ Start with an identifier 90 ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\idstr{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in sections 86 and 95.

91 ⟨ Start with a tag 91 ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\idstr{<`$\llcorner v_a \lrcorner$`>}{<`$\llcorner v_b \lrcorner$`>}`$\rangle$

92 ⟨ Start with a quoted string 92 ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\chstr{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in section 86.

93 ⟨ Start with a `.` string 93 ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\chstr{.}{.}`$\rangle$
This code is used in section 86.

94 ⟨ Start with an `_` string 94 ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\chstr{\uscoreletter}{\uscoreletter}`$\rangle$
This code is used in section 86.

95 ⟨ Turn a qualifier into an identifier 95 ⟩ =
⟨ Start with an identifier 90 ⟩

96 ⟨ Attach an identifier 96 ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a$
$v_a \leftarrow v_a +_{\text{sx}} [\,^{\text{nox}}$`\_`$\,]$
$\pi_1(\Upsilon_3) \mapsto v_b$
$v_a \leftarrow v_a +_{\text{s}} v_b$
$\pi_3(\Upsilon_1) \mapsto v_b$
$v_b \leftarrow v_b +_{\text{sx}} [$`\uscoreletter`$\,]$
$\pi_2(\Upsilon_3) \mapsto v_c$
$v_b \leftarrow v_b +_{\text{s}} v_c$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\idstr{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in section 86.

97 ⟨ Attach qualifier to a name 97 ⟩ =
This code is used in section 86.

98 ⟨ Attach an integer 98 ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\dotsp`$\,^{\text{nx}}$`\sfxi`$\,\text{val}\,\Upsilon_1\rangle$
This code is used in section 86.

99 ⟨ Attach a subscripted integer 99 ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\dotsp`$\,^{\text{nx}}$`\sfxi`$\,\text{val}\,\Upsilon_2\rangle$
This code is used in section 86.

100 ⟨ Attach a subscripted qualifier 100 ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\dotsp`$\,^{\text{nx}}$`\qual`$\,\text{val}\,\Upsilon_2\rangle$
This code is used in section 86.

101 ⟨ Attach suffixes 101 ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in sections 86 and 102.

102 ⟨ Attach qualified suffixes 102 ⟩ =
⟨ Attach suffixes 101 ⟩
This code is used in section 86.

103 ⟨ Start with a named suffix 103 ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash sfxn}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 86.

104 ⟨ Start with a numeric suffix 104 ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash sfxi}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 86.

105 ⟨ Add a dot separator 105 ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\rangle$
This code is used in section 86.

106 ⟨ Attach integer suffix 106 ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}\texttt{\textbackslash sfxi}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 86.

107 ⟨ Attach a named suffix 107 ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}\texttt{\textbackslash sfxn}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 86.

108 ⟨ Attach a qualifier 108 ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}\texttt{\textbackslash qual}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 86.

109 ⟨ Start suffixes with a qualifier 109 ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash qual}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 86.

110 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 110 ⟩ =
This code is used in section 82.

111 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 111 ⟩ =

```
#define YYPRINT(file, type, value) yyprint(file, type, value)
  static void yyprint(FILE *file, int type, YYSTYPE value)
  { }
```

This code is used in section 82.

112 Union of types.

⟨ Union of parser types 112 ⟩ =
This code is used in section 82.

113 The name scanner

⟨ ld_small_lexer.ll 113 ⟩ =
```
⟨ Lexer definitions 114 ⟩
.....
⟨ Lexer C preamble 117 ⟩
.....
⟨ Lexer options 118 ⟩

⟨ Regular expressions 119 ⟩

void define_all_states(void)
{
    ⟨ Collect all state definitions 115 ⟩
}
```

114 ⟨ Lexer definitions 114 ⟩ =
```
⟨ Lexer states 116 ⟩
aletter   [a-zA-Z]
wc        ([^\\\']{-}[a-zA-Z0-9]|\\.)
id        ({aletter}|{aletter}({aletter}|[0-9])*{aletter})
int       [0-9]+
```
This code is used in section 113.

115 ⟨ Collect all state definitions 115 ⟩ =
```
#define _register_name(name) Define_State(#name, name)    ▷ nothing for now ◁
#undef _register_name
```
This code is used in section 113.

116 Strings and characters in directives/rules.

⟨ Lexer states 116 ⟩ =
```
⟨ states-x ⟩f :  SC_ESCAPED_STRING SC_ESCAPED_CHARACTER
```
This code is used in section 114.

117 ⟨ Lexer C preamble 117 ⟩ =
```
#include <stdint.h>
#include <stdbool.h>
```
This code is used in section 113.

118 ⟨ Lexer options 118 ⟩ =
```
⟨ bison-bridge ⟩f ⋆
⟨ noyywrap ⟩f ⋆
⟨ nounput ⟩f ⋆
⟨ noinput ⟩f ⋆
⟨ reentrant ⟩f ⋆
⟨ noyy_top_state ⟩f ⋆
⟨ debug ⟩f ⋆
⟨ stack ⟩f ⋆
⟨ outfile ⟩f              "ld_small_lexer.c"
```
This code is used in section 113.

119 ⟨ Regular expressions 119 ⟩ =
```
⟨ Scan white space 120 ⟩
⟨ Scan identifiers 121 ⟩
```
This code is used in section 113.

120 White space skipping.

⟨ Scan white space 120 ⟩ =

```
[ \f\n\t\v]                    {continue}
```

This code is used in section 119.

121 This collection of regular expressions might seem redundant, and in its present state, it certainly is. However, if later on the typesetting style for some of the keywords would need to be adjusted, such changes would be easy to implement, since the template is already here.

⟨ Scan identifiers 121 ⟩ =

```
"%"({aletter}|[0-9]|[-_]|"%"|[<>])+ {return_v %[a...Z0...9]*}

"opt"                          {return_v opt}
"K"                            {return_v suffix_K}
"ext"                          {return_v ext}

[\'._]                         {return_c}
{wc}                           {return_v * or ?}

{id}                           {⟨ Prepare to process an identifier 122 ⟩}
{int}                          {return_v [0...9]*}

"\""                           {continue}
.                              {⟨ React to a bad character 123 ⟩}
```

This code is used in section 119.

122 ⟨ Prepare to process an identifier 122 ⟩ =

```
return_v [a...Z0...9]*
```

This code is used in section 121.

123 A simple routine to detect trivial scanning problems.

⟨ React to a bad character 123 ⟩ =

```
if_t [bad char]
    \yycomplain{invalid character(s): val\yytext}
fi
return_l $undefined
```

This code is used in section 121.

124 Appendix

The original code of the `ld` parser and lexer is reproduced below. It is mostly left intact and is typeset by the pretty printing parser for `bison` input. The lexer (`flex`) input is reproduced verbatim and is left mostly unformatted with the exception of spacing and the embedded C code.

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`.

125 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 125 ⟩ =

⟨ C setup for `ld` grammar 126 ⟩

⟨**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 127 ⟩

⟨ Original `ld` grammar rules 128 ⟩

126 *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 126 ⟩ =

```
#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 125.

127 *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 127 ⟩ =

```
⟨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$_{\text{opt}}$
⟨union⟩.*cname* :
wildcard_name
⟨union⟩.*wildcard* :
wildcard_spec

INT name name$_{\text{L}}$

⟨union⟩.*integer* :
length
⟨union⟩.*phdr* : *phdr_qualifiers*
⟨union⟩.*nocrossref* :
nocrossref_list
⟨union⟩.*section_phdr* :
phdr$_{\text{opt}}$
⟨union⟩.*integer* :
nocrossrefs$_{\text{opt}}$

⟨**right** : *token*⟩ $\stackrel{+}{\Leftarrow}$ $\stackrel{-}{\Leftarrow}$ $\stackrel{\times}{\Leftarrow}$ $\stackrel{\div}{\Leftarrow}$ $\Leftarrow$ $\stackrel{\ll}{\Leftarrow}$ $\stackrel{\gg}{\Leftarrow}$ $\stackrel{\&}{\Leftarrow}$ $\stackrel{\vee}{\Leftarrow}$
⟨**right** : *token*⟩ `? :`
⟨**left** : *token*⟩ $\vee$
⟨**left** : *token*⟩ $\wedge$
⟨**left** : *token*⟩ |
⟨**left** : *token*⟩ $\oplus$
⟨**left** : *token*⟩ &
⟨**left** : *token*⟩ $= \neq$
⟨**left** : *token*⟩ $< > \leqslant \geqslant$
⟨**left** : *token*⟩ $\ll \gg$
⟨**left** : *token*⟩ $+ -$
⟨**left** : *token*⟩ * / $\div$
⟨**right**⟩ unary

end

⟨**left** : *token*⟩ (

ALIGN$_{\text{K}}$	BLOCK	BIND	QUAD
SQUAD	LONG	SHORT	BYTE
SECTIONS	PHDRS	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	TARGET$_{\text{K}}$	SEARCH_DIR
MAP	ENTRY	NEXT	SIZEOF
ALIGNOF	ADDR	LOADADDR	MAX$_{\text{K}}$
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		

⟨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	$\texttt{ASSERT}_{\texttt{K}}$	LOG2CEIL	FORMAT
PUBLIC	DEFSYMEND	BASE	ALIAS
TRUNCATE	REL	INPUT_SCRIPT	INPUT_MRI_SCRIPT
INPUT_DEFSYM	CASE	EXTERN	START
VERS_TAG	VERS_IDENTIFIER	GLOBAL	LOCAL
$\texttt{VERSION}_{\texttt{K}}$	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 125.

128 *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 128 ⟩ =

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 129, 130, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, and 145.

This code is used in section 125.

129 Syntax within an MRI script file.

⟨ Original ld grammar rules 128 ⟩ + =

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

mri_script_lines:
mri_script_lines mri_script_command NEWLINE
∘

130 ⟨ Original ld grammar rules 128 ⟩ + =

mri_script_command:
CHIP *exp*
CHIP *exp* , *exp*
name ⟨ Flag an unrecognized keyword 131 ⟩
LIST *config*.*map_filename* ⇐ "-";
ORDER *ordernamelist*
ENDWORD
PUBLIC name ⇐ *exp* *mri_public*(Υ_2, Υ_4);

`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`$_{\rm K}$ name ⇐ *exp*	*mri_align*(Υ2, Υ4);
`ALIGN`$_{\rm K}$ 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	

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

⟨ Flag an unrecognized keyword 131 ⟩ =

einfo(_("`%P%F:␣unrecognised␣keyword␣in␣MRI␣style␣script␣'%s'\n`"), Υ1);

This code is used in section 130.

132 Parsed as expressions so that commas separate entries.

⟨ Original `ld` grammar rules 128 ⟩ + =

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*$_1$
∘

133 *All the commands that can appear in a standard linker script.*
⟨ Original `ld` grammar rules 128 ⟩ + =

***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(Υ$_2$);
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)

134 ⟨ Original `ld` grammar rules 128 ⟩ + =

input_list :
name
input_list , name
input_list name
name$_L$
input_list , name$_L$
input_list name$_L$
`AS_NEEDED` (
input_list)
input_list , `AS_NEEDED` (
input_list)
input_list `AS_NEEDED` (
input_list)

sections :
`SECTIONS` { *sec_or_group_p*$_1$ }

***sec_or_group_p*$_1$** :
sec_or_group_p$_1$ *section*
sec_or_group_p$_1$ *statement_anywhere*
∘

statement_anywhere :
`ENTRY (` name `)`
assignment end
`ASSERT`$_{\mathrm{K}}$ *ldlex_expression*( );
`(` *exp* `,` name `)` *ldlex_popstate*( );

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

⟨ Original `ld` grammar rules 128 ⟩ + =

wildcard_name :
name
`*`
`?`

136 ⟨ Original `ld` grammar rules 128 ⟩ + =

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* `,`$_{\mathrm{opt}}$ *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* `)`

137 ⟨ Original `ld` grammar rules 128 ⟩ + =

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_K                                              ldlex_expression( );
        ( exp , name ) end                                ldlex_popstate( );
    INCLUDE filename                                      ldlex_script( );
                                                          ldfile_open_command_file(Υ2);
        statement_list_opt end                            ldlex_popstate( );
statement_list :  statement_list statement | statement
statement_list_opt :  ○ | statement_list
length :  QUAD | SQUAD | LONG | SHORT | BYTE
fill_exp :  mustbe_exp
fill_opt :  ⇐ fill_exp | ○
assign_op :  +⇐ | -⇐ | ×⇐ | ÷⇐ | ≪⇐ | ≫⇐ | &⇐ | ∨⇐
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(Υ2);
        memory_spec_list_opt end                          ldlex_popstate( );
```

138 ⟨ Original `ld` grammar rules 128 ⟩ + =

```
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* `,`$_{\text{opt}}$ *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*( );

139 *Rich expression syntax reproducing the one in* C.

⟨ Original `ld` grammar rules 128 ⟩ + =

exp :
− *exp* ⟨`prec` unary⟩
`(` *exp* `)`
`NEXT (` *exp* `)` ⟨`prec` unary⟩
¬ *exp* ⟨`prec` unary⟩
+ *exp* ⟨`prec` unary⟩
$^{\text{not}}$ *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`$_{\text{K}}$ `(` *exp* `)`
`ALIGN`$_{\text{K}}$ `(` *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`$_{\text{K}}$ `(` *exp* `,` *exp* `)`
`MIN`$_{\text{K}}$ `(` *exp* `,` *exp* `)`
`ASSERT`$_{\text{K}}$ `(` *exp* `,` name `)`
`ORIGIN (` name `)`
`LENGTH (` name `)`

LOG2CEIL (*exp*)

140 ⟨ Original `ld` grammar rules 128 ⟩ + =

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

at$_{\text{opt}}$:
AT (*exp*)
∘

align$_{\text{opt}}$:
ALIGN$_{\text{K}}$ (*exp*)
∘

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

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

sect_constraint :
ONLY_IF_RO
ONLY_IF_RW
SPECIAL
∘

141 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 128 ⟩ + =

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*) | ∘ | ( )

142 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 128 ⟩ + =

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*( );
        `,`$_{\text{opt}}$

143 ⟨ Original `ld` grammar rules 128 ⟩ + =

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 `;`

144 This syntax is used within an external version script file.

⟨ Original `ld` grammar rules 128 ⟩ + =

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

145 This is used within a normal linker script file.

⟨ Original `ld` grammar rules 128 ⟩ + =

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 $;_{\text{opt}}$ `}`
`GLOBAL`
vers_defns `;` `GLOBAL`
`LOCAL`
vers_defns `;` `LOCAL`
`EXTERN`
vers_defns `;` `EXTERN`

$;_{\text{opt}}$: ∘ | `;`

146 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);
}
```

147 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 147 ⟩ =

```
⟨ Original ld macros 150 ⟩
.....
⟨ Original ld preamble 148 ⟩
.....

⟨nounput⟩f ⋆
⟨ Ignored options 149 ⟩

⟨ Original ld regular expressions 153 ⟩
```

148 ⟨ Original `ld` preamble 148 ⟩ =

```
#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 147.

149 ⟨ Ignored options 149 ⟩ =

```
%a 4000
%o 5000
```

This code is used in section 147.

150 *Some convenient abbreviations for regular expressions.*

⟨ Original ld macros 150 ⟩ =

```
CMDFILENAMECHAR  [_a-zA-Z0-9\/\.\\_\+\$\:\[\]\\\,\=\&\!\<\>\-\~]
CMDFILENAMECHAR1 [_a-zA-Z0-9\/\.\\_\+\$\:\[\]\\\,\=\&\!\<\>\~]
FILENAMECHAR1    [_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 147.

151 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 151 ⟩ =

```
⟨states-s⟩f: SCRIPT
⟨states-s⟩f: EXPRESSION
⟨states-s⟩f: BOTH
⟨states-s⟩f: DEFSYMEXP
⟨states-s⟩f: MRI
⟨states-s⟩f: VERS_START
⟨states-s⟩f: VERS_SCRIPT
⟨states-s⟩f: VERS_NODE
```

152 ⟨ ld postamble 152 ⟩ =

```
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( );
  }
}
```

153 ⟨ Original ld regular expressions 153 ⟩ =

```
<BOTH,SCRIPT,EXPRESSION,VERS_START,VERS_NODE,VERS_SCRIPT>"/*" {comment( ); }
<DEFSYMEXP>"-"                                  {RTOKEN('-');}
<DEFSYMEXP>"+"                                  {RTOKEN('+');}
<DEFSYMEXP>{FILENAMECHAR1}{SYMBOLCHARN}*        {yylval.name ⇐ xstrdup(yytext); return NAME;}
<DEFSYMEXP>"="                                  {RTOKEN('=');}

<MRI,EXPRESSION>"$"([0-9A-Fa-f])+ {
yylval.integer ⇐ bfd_scan_vma(yytext + 1, 0, 16);
yylval.bigint.str ⇐ Λ;
return INT;
}

<MRI,EXPRESSION>([0-9A-Fa-f])+(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 ⇐ 16;
  break;
case 'O': case 'o': ibase ⇐ 8;
  break;
case 'B': case 'b': ibase ⇐ 2;
  break;
default: ibase ⇐ 10;
}
yylval.integer ⇐ bfd_scan_vma(yytext, 0, ibase);
yylval.bigint.str ⇐ Λ;
return INT;
}

<SCRIPT,DEFSYMEXP,MRI,BOTH,EXPRESSION>((("$"|0[xX])([0-9A-Fa-f])+)|(([0-9])+))(M|K|m|k)? {
char ∗s ⇐ yytext;
int ibase ⇐ 0;
if (∗s = 'ϒ') {
  ++s;
  ibase ⇐ 16;
}
yylval.integer ⇐ bfd_scan_vma(s, 0, ibase);
```

```
yylval.bigint.str ⇐ Λ;
if (yytext[yyleng − 1] = 'M' ∨ yytext[yyleng − 1] = 'm') {
  yylval.integer ⇐* 1024 ∗ 1024;
}
else if (yytext[yyleng − 1] = 'K' ∨ yytext[yyleng − 1] = 'k') {
  yylval.integer ⇐* 1024;
}
else if (yytext[0] = '0' ∧ (yytext[1] = 'x' ∨ yytext[1] = 'X')) {
  yylval.bigint.str ⇐ xstrdup(yytext + 2);
}
return INT;
}

<BOTH,SCRIPT,EXPRESSION,MRI>"]"                {RTOKEN(']');}
<BOTH,SCRIPT,EXPRESSION,MRI>"["                {RTOKEN('[');}
<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);}
<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>{FILENAMECHAR1}{NOCFILENAMECHAR}* {    ▷ Filename without commas, needed to parse MRI stuff ◁
yylval.name ⇐ xstrdup(yytext);
return NAME;
}
<BOTH>{FILENAMECHAR1}{FILENAMECHAR}* {
yylval.name ⇐ xstrdup(yytext);
return NAME;
}
<BOTH>"-l"{FILENAMECHAR}+ {
yylval.name ⇐ xstrdup(yytext + 2);
return LNAME;
}
<EXPRESSION>{FILENAMECHAR1}{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>"\""[^\"]*"\"" {    ▷ 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();
else yy_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 147.

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

⟨ Supporting C code 154 ⟩ =

```
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 155, 156, 157, 158, 159, 160, and 161.

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

```
⟨ Supporting C code 154 ⟩ +=
  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;
  }
```

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

```
⟨ Supporting C code 154 ⟩ +=
  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);
  }
```

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

```
⟨ Supporting C code 154 ⟩ +=
  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);
}
```

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

```
⟨ Supporting C code 154 ⟩ +=
const char *ldlex_filename(void)
{
  return file_name_stack[include_stack_ptr − (include_stack_ptr ≠ 0)];
}
```

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

```
⟨ Supporting C code 154 ⟩ +=
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;
}
```

160 Eat the rest of a C-style comment.

```
⟨ Supporting C code 154 ⟩ +=
  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;
      }
    }
  }
```

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

```
⟨ Supporting C code 154 ⟩ +=
  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);
  }
```

162 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	origin	length
	MEMORY1	xrw	$2000\,0000_{16}$	20 K
	MEMORY2	rx	$800\,0000_{16}$	128 K

_var_1 ⇐ $2000\,5000_{16}$

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, LD, and TEX indices

A LIST OF ALL SECTIONS

Contents (ldman)