SPLINT
reference

version 1.1.0

LD PARSER REFERENCE

Alex Shibakov

October 11, 2020

1 Introduction

3a 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 3a ⟩ =

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

\ifnum\optimization>\tw@
    \input yyfaststack.sty
\else
    \let\stashnext\stashnextwithnothing
\fi

\input yystype.sty  %   ▷ scanner auxiliary types and functions ◁
\input yyunion.sty  %   ▷ parser data structures ◁
\input yxunion.sty  %   ▷ extended parser data structures ◁
\expandafter\def    %   ▷ adjust the \yyinput to recognize \yyendgame ◁
    \expandafter\multicharswitch\expandafter
    {\multicharswitch\yyendgame{\yyinput\yyeof\yyeof\endparseinput\removefinalvb}}
```

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

This code is used in section 8a.

4a 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 4a ⟩ =

```
\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%
    \input yytexlex.sty%
}
```

This code is used in section 8a.

5a Namespaces and modes

Every parser/lexer pair (as well as some other macros) operates within a set of dedicated *namespaces*. This simply means that the macros that output token values, switch lexer states and access various tables 'tack on' 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:

- **main**: the `[main]` namespace is established for the parser that does the typesetting of the grammar.
- **ld**: every time a term name is processed, the token names are looked up in the `[ld]` namespace. The same namespace is used by the parser that typesets `ld` script examples in the manual (i.e. the parser described here). This is done to provide visual consistency between the description of the parser and its output.
- **small** and **ldsmall**: the `[small]` namespace is used by the term name parser itself. Since we use a customized version of the name parser, we dedicate a separate namespace for this purpose, `[ldsmall]`.
- **prologue**: the parser based on a subset of the full `bison` grammar describing prologue declarations uses the `[prologue]` namespace.
- **index**: the `[index]` namespace is used for typesetting the index entries and is not necessarily associated with any parser or lexer. Somewhat confusingly, the macros that typeset TEX entries, use `index` (without the brackets) as a pseudonamespace to display TEX terms in the index (due to the design of these typesetting macros, many of them take parameters, which can lead to chaos in the index). These two namespaces are not related but due to 'hystorical' reasons (and the poorly thought out TEX typesetting macro design) the `index` name has been retained. In addition, `index:visual` is used to adjust the sort order of TEX terms (similar to the way `\prettywordpairwvis` macro does).
- **flexre**, **flexone,** and **flextwo**: the parsers for `flex` input use the `[flexre]`, `[flexone]`, and `[flextwo]` namespaces for their operation. Another convention is to use the `\flexpseudonamespace` to typeset `flex` state names, and the `\flexpseudorenamespace` for typesetting the names of `flex` regular expressions. Currently, `\flexpseudo...` namespaces are set equal to their non-`pseudo` versions by default. This setting may be changed whenever several parsers are used in the same document and tokens with the same names must be typeset in different styles. All `flex` namespaces, as well as `[main]`, `[small]`, and `[ldsmall]` are defined by the `\genericparser` macros.
- **cwebclink**: finally, the `[cwebclink]` namespace is used for typesetting the variables *inside* `ld` scripts. This way, the symbols exported by the linker may be typeset in a style similar to C variables, if desired (as they play very similar roles).

⟨ Begin namespace setup 5a ⟩ =

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

This code is used in section 8a.

6a 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 6a ⟩ =

```
\newtoks\ldcmds

\def\modenormal{%
    \def\appendr##1##2{\edef\appnext{##1{\the##1##2}}\appnext}%
    \def\appendl##1##2{\edef\appnext{##1{##2\the##1}}\appnext}%
    \input yyinit.sty%
    \input yytexlex.sty%    ▷ TEX typesetting macros ◁
    \input ldtexlex.sty%    ▷ TEX typesetting specific to ld ◁
    \let\hostparsernamespace\ldnamespace
  ▷ the namespace where tokens are looked up for typesetting purposes ◁
    ⟨ Initialize ld parsers 7a ⟩
    ⟨ Modified name parser for ld grammar 41b ⟩
}
```

See also sections 6b and 6c.

This code is used in section 8a.

6b 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 6a ⟩ + =

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

6c 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 6a ⟩ + =

```
\def\preparseld{%
    \let\postparse\postparseld
    \expandafter\hidecs\expandafter{\ldunion}%
  ▷ inhibit expansion so that fewer \noexpands are necessary ◁
    \toldparser
    \ldparserinit
    \yyparse
}

\def\postparseld{%
    \ifsaveparseoutput
        {\newlinechar=‘^^J\immediate\write\exampletable{^^J\harmlesscomment
         parsed table: \the\ldcmds^^J^^J\harmlesscomment
         stashed stream:^^J\the\yystash^^J^^J\harmlesscomment
         format stream: ^^J\the\yyformat}%
        }%
    \fi
```

```
    \ifchecktable
        \errmessage{parsed table: \the\ldcmds^^J^^J%
                    stashed stream: \the\yystash^^J^^J%
                    format stream: \the\yyformat}%
    \fi
    \restorecslist{ld-parser:restash}\ldunion %    ▷ mark variables, preprocess stash ◁
    \setprodtable
    \the\ldcmds
    \restorecslist{ld-display}\ldunion
    \setprodtable    ▷ use the bison's parser typesetting definitions ◁
    \restorecs{ld-display}{\anint\bint\hexint} %    ▷ ... except for integer typesetting ◁
    \the\ldcmds
    \par
    \vskip-\baselineskip
    \the\lddisplay
}

\fillpstack{l}{%
    \preparseld
    {\preparsefallback{++}}%    ▷ skip this section if parsing failed, put ++ on the screen ◁
    \relax %    ▷ this \relax serves as a 'guard' for the braces ◁
}
```

7a 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 `ASSERT_K`. The name parser treats `K` as a suffix to supply a visual reminder of this flaw. Note that the 'suffixless' part of these tokens (such as `ASSERT`) is never declared and thus has to be entered in `ltokenset.sty` by hand.

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

⟨ Initialize `ld` parsers 7a ⟩ =

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

This code is used in section 6a.

8a The macros are collected in a single file included at the beginning of this documentation.

⟨ `ldman.stx` 8a ⟩ =
⟨ Set up the generic parser machinery 3a ⟩
⟨ Begin namespace setup 5a ⟩
⟨ Define the bootstrapping mode 4a ⟩
⟨ Define the normal mode 6a ⟩
⟨ Additional macros for the `ld` lexer/parser 26d ⟩

2
The parser

9a 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` 9a ⟩ =

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

⟨ ld parser productions 10c ⟩
```

9b 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 9b ⟩ =

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

This code is used in section 9a.

9c ⟨ `ld` parser C preamble 9c ⟩ =

This code is used in section 9a.

10a ⟨ Union of grammar parser types 10a ⟩ =

This code is used in section 9a.

10b ⟨ ld parser C postamble 10b ⟩ =

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

This code is used in section 9a.

10c ⟨ ld parser productions 10c ⟩ =

⟨ GNU ld script rules 12a ⟩
⟨ Grammar rules 11b ⟩

This code is used in section 9a.

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

⟨ Token and type declarations 10d ⟩ =

INT	name	name_L	end
$\text{ALIGN}_\text{\_K}$	BLOCK	BIND	QUAD
SQUAD	LONG	SHORT	BYTE
SECTIONS	PHDRS	$\text{INSERT}_\text{\_K}$	AFTER
BEFORE	DATA_SEGMENT_ALIGN	DATA_SEGMENT_RELRO_END	DATA_SEGMENT_END
SORT_BY_NAME	SORT_BY_ALIGNMENT	SORT_NONE	SORT_BY_INIT_PRIORITY
{	}	SIZEOF_HEADERS	OUTPUT_FORMAT
FORCE_COMMON_ALLOCATION	OUTPUT_ARCH	INHIBIT_COMMON_ALLOCATION	SEGMENT_START
INCLUDE	MEMORY	REGION_ALIAS	LD_FEATURE
NOLOAD	DSECT	COPY	INFO
OVERLAY	DEFINED	$\text{TARGET}_\text{\_K}$	SEARCH_DIR
MAP	ENTRY	NEXT	SIZEOF
ALIGNOF	ADDR	LOADADDR	$\text{MAX}_\text{\_K}$
$\text{MIN}_\text{\_K}$	STARTUP	HLL	SYSLIB
FLOAT	NOFLOAT	NOCROSSREFS	ORIGIN
FILL	LENGTH	CREATE_OBJECT_SYMBOLS	INPUT
GROUP	OUTPUT	CONSTRUCTORS	ALIGNMOD
AT	SUBALIGN	HIDDEN	PROVIDE
PROVIDE_HIDDEN	AS_NEEDED	CHIP	LIST
SECT	ABSOLUTE	LOAD	NEWLINE
ENDWORD	ORDER	NAMEWORD	$\text{ASSERT}_\text{\_K}$
LOG2CEIL	FORMAT	PUBLIC	DEFSYMEND
BASE	ALIAS	TRUNCATE	REL
INPUT_SCRIPT	INPUT_MRI_SCRIPT	INPUT_DEFSYM	CASE
EXTERN	START	VERS_TAG	VERS_IDENTIFIER
GLOBAL	LOCAL	VERSION_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**⟩ $\stackrel{+}{\Leftarrow}$ $\stackrel{-}{\Leftarrow}$ $\stackrel{\times}{\Leftarrow}$ $\stackrel{\div}{\Leftarrow}$ $\Leftarrow$ $\stackrel{\ll}{\Leftarrow}$ $\stackrel{\gg}{\Leftarrow}$ $\stackrel{\&}{\Leftarrow}$ $\stackrel{\vee}{\Leftarrow}$? : unary
⟨**left**⟩ $\vee$ $\wedge$ | $\oplus$ & $=$ $\neq$ $<$ $>$ $\leqslant$ $\geqslant$ $\ll$ $\gg$ $+$ $-$ $\times$ / $\div$ (

This code is used in section 9a.

11a 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 11a ⟩ =

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*

11b

⟨ Grammar rules 11b ⟩ =

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

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

This code is used in section 10c.

11c

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 11c ⟩ =

defsym_expr :

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

11d

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 11d ⟩ =

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

mri_script_lines : *mri_script_lines mri_script_command* `NEWLINE` | ∘

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

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

mri_load_name_list : name | *mri_load_name_list* , name

mri_abs_name_list : name | *mri_abs_name_list* , name

casesymlist : ○ | name | *casesymlist* , name

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

⟨ GNU ld script rules 12a ⟩ =

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

extern_name_list_body :
name
extern_name_list_body name
extern_name_list_body , name

See also sections 12b and 13b.

This code is used in section 10c.

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

⟨ GNU ld script rules 12a ⟩ + =

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

ifile_list :
ifile_list ifile_p$_1$ ⟨ Add the next command 13a ⟩
○ $\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb{ }{ }{ }{ }⟩

13a ⟨ Add the next command 13a ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h$
`\yytoksempty{` v_h `}{` $\Upsilon \leftarrow \langle \text{val}\, \Upsilon_1 \rangle$ `}{\yytoksempty{` v_d `}{` $\Upsilon \leftarrow \langle \text{val}\, \Upsilon_2 \rangle$ `}`
`{` $\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{` val v_g `}{` val v_d
$^{\text{nx}}$`\ldcommandseparator{` val v_e `}{` val v_f `}{` val v_c `}{` val v_g `}`val v_h `}⟩ }}`

This code is used in section 12b.

13b Script internals

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

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

⟨ GNU `ld` script rules 12a ⟩ + =

ifile_p 1 :

memory	⟨ Carry on 14e ⟩
sections	⟨ Carry on 14e ⟩
phdrs	
startup	
high_level_library	
low_level_library	
floating_point_support	
statement_anywhere	⟨ Carry on 14e ⟩
version	
;	$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{none}{}`$\rangle$
`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`	⟨ Peek at a file 15a ⟩
ifile_list end	⟨ Close the file 15b ⟩
`NOCROSSREFS ( nocrossref_list )`	
`EXTERN ( extern_name_list )`	
`INSERT_K AFTER name`	
`INSERT_K BEFORE name`	

`REGION_ALIAS ( name , name )`
`LD_FEATURE ( name )`

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*$_1$ `}` ⟨ Form the `SECTIONS` group 14a ⟩

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

statement_anywhere :
`ENTRY (` name `)` ⟨ Form an `ENTRY` statement 14c ⟩
assignment end ⟨ Form a statement 14d ⟩
ASSERT_{K} `\ldlex@expression`
`(` *exp* , name `)` `\ldlex@popstate`

14a ⟨ Form the `SECTIONS` group 14a ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1{ `sect` }{ $^{\text{nx}}$`\ldsections {` val $\Upsilon_3{}^{\text{nx}}$`\ldsectionstash` val Υ_4 `}}`$\rangle$
This code is used in section 13b.

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

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

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

14e 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 14e ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$
This code is used in sections 13b, 15d, 16b, 16d, 17b, and 17i.

15a ⟨ Peek at a file 15a ⟩ =

```
\ldlex@script
\ldfile@open@command@file{ ϒ2 }
```

This code is used in sections 11d, 13b, 16d, 17i, and 21a.

15b ⟨ Close the file 15b ⟩ =

```
ϒ ← ⟨nx\ldinsertcweb val ϒ1{ inc }{ nx\ldinclude{ val ϒ2 }}⟩ \ldlex@popstate
```

This code is used in sections 11d, 13b, 16d, 17i, and 21a.

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

⟨ Grammar rules 11b ⟩ + =

wildcard_name :

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

15d ⟨ Grammar rules 11b ⟩ + =

wildcard_spec :

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

sect_flag_list :

name
sect_flag_list & name

sect_flags :

INPUT_SECTION_FLAGS (*sect_flag_list*)

exclude_name_list :

exclude_name_list wildcard_name
wildcard_name

file_name_***list*** :

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

input_section_spec_no_keep :

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

15e ⟨ Create a wildcard name 15e ⟩ =

```
ϒ ← ⟨nx\ldinsertcweb{ val va }{ val vb }{ wld }{ nx\ldregexp{ val ϒ1 }}⟩
```

This code is used in section 15c.

16a ⟨ Add a wildcard spec to a list of files 16a ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h \Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{flst}{` val v_d $^{\text{nx}}$`\ldspace` val v_h `}`$\rangle$

This code is used in section 15d.

16b ⟨ Start a file list with a wildcard spec 16b ⟩ =

⟨ Carry on 14e ⟩

This code is used in section 15d.

16c ⟨ Add a plain section spec 16c ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_5(\Upsilon_3) \mapsto v_h \Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{sspec}{` val v_d `(` val v_h `)}`$\rangle$

This code is used in section 15d.

16d ⟨ Grammar rules 11b ⟩ + =

input_section_spec :

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

statement :

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

statement_list :

statement_list statement	⟨ Attach a statement to a statement list 17a ⟩
statement	⟨ Carry on 14e ⟩

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

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

16e ⟨ Form an input section spec 16e ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb{` val v_a `}{` val v_b `}{stmt}{`$^{\text{nx}}$`\ldsecspec{` val v_d `}}`$\rangle$

This code is used in section 16d.

16f ⟨ Add a `KEEP` statement 16f ⟩ =

$\pi_5(\Upsilon_4) \mapsto v_a \Upsilon \leftarrow \langle^{\text{nx}}$`\ldinsertcweb` val Υ_1`{stmt}{`$^{\text{nx}}$`\ldkeep{` val v_a `}}`$\rangle$

This code is used in section 16d.

17a ⟨ Attach a statement to a statement list 17a ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\pi_4(\Upsilon_1) \mapsto v_c \pi_5(\Upsilon_1) \mapsto v_d$
$\pi_2(\Upsilon_2) \mapsto v_e \pi_3(\Upsilon_2) \mapsto v_f$
$\pi_4(\Upsilon_2) \mapsto v_g \pi_5(\Upsilon_2) \mapsto v_h$
\yytoksempty{ v_d }{ $\Upsilon \leftarrow \langle$ val $\Upsilon_2\rangle$ }
{ $\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb{ val v_a }{ val v_b }{ val v_g }{ val v_d
\yytoksempty{ v_h }{ }{ \yytoksempty{ v_d }{ }{ $^{\text{nx}}$\ldor }val v_h }}$\rangle$ }

This code is used in section 16d.

17b ⟨ Grammar rules 11b ⟩ + =

length : QUAD | SQUAD | LONG | SHORT | BYTE
fill_exp : *mustbe_exp* ⟨ Carry on 14e ⟩
fill$_{\text{opt}}$: ⇐ *fill_exp* | ∘ $\Upsilon \leftarrow \langle\rangle$
assign_op :
$\overset{+}{\Leftarrow}$ $\Upsilon_0 =$ { \MRL{ +{ ⇐ } } }
$\overset{-}{\Leftarrow}$ $\Upsilon_0 =$ { \MRL{ -{ ⇐ } } }
$\overset{\times}{\Leftarrow}$ | $\overset{\div}{\Leftarrow}$ | $\overset{\ll}{\Leftarrow}$ | $\overset{\gg}{\Leftarrow}$ | $\overset{\&}{\Leftarrow}$ | $\overset{\vee}{\Leftarrow}$ $\Upsilon_0 =$ { $\overset{\vee}{\Leftarrow}$ }
end : ; | ,
,$_{\text{opt}}$: , | ∘

17c Assignments are not expressions as in C.

⟨ Grammar rules 11b ⟩ + =

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

17d ⟨ Process simple assignment 17d ⟩ =

$\pi_3(\Upsilon_1) \mapsto v_a \pi_4(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb{ val v_a }{ val v_b }{ asgnm }{ $^{\text{nx}}$\ldassignment{ $^{\text{nx}}$\ldregexp{ val Υ_1 }}{ ⇐ }{ val Υ_3 }}$\rangle$

This code is used in section 17c.

17e ⟨ Process compound assignment 17e ⟩ =

$\pi_3(\Upsilon_1) \mapsto v_a \pi_4(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb{ val v_a }{ val v_b }{ asgnm }{ $^{\text{nx}}$\ldassignment{ $^{\text{nx}}$\ldregexp{ val Υ_1 }}{ val Υ_2 }{ val Υ_3 }}$\rangle$

This code is used in section 17c.

17f ⟨ Process a HIDDEN assignment 17f ⟩ =

$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb val Υ_1 { hiddn }{ $^{\text{nx}}$\ldhidden{ $^{\text{nx}}$\ldregexp{ val Υ_3 }}{ val Υ_5 }}$\rangle$

This code is used in section 17c.

17g ⟨ Process a PROVIDE assignment 17g ⟩ =

$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb val Υ_1 { prvde }{ $^{\text{nx}}$\ldprovide{ $^{\text{nx}}$\ldregexp{ val Υ_3 }}{ val Υ_5 }}$\rangle$

This code is used in section 17c.

17h ⟨ Process a PROVIDE_HIDDEN assignment 17h ⟩ =

$\Upsilon \leftarrow \langle^{\text{nx}}$\ldinsertcweb val Υ_1 { prhid }{ $^{\text{nx}}$\ldprovidehid{ $^{\text{nx}}$\ldregexp{ val Υ_3 }}{ val Υ_5 }}$\rangle$

This code is used in section 17c.

17i ⟨ Grammar rules 11b ⟩ + =

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

memory_spec_list _opt :
memory_spec_list ⟨ Carry on 14e ⟩
○ $\Upsilon \leftarrow \langle\rangle$

memory_spec_list :
memory_spec_list ,opt_ *memory_spec* ⟨ Add a memory spec 18c ⟩
memory_spec ⟨ Start a list of memory specs 18b ⟩

memory_spec :
name
*attributes*_opt : *origin_spec* ↩
,opt_ *length_spec* ⟨ Declare a named memory region 18d ⟩
INCLUDE *filename* ⟨ Peek at a file 15a ⟩
*memory_spec_list*_opt end ⟨ Close the file 15b ⟩

18a ⟨ Form the MEMORY group 18a ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$\ldinsertcweb val Υ_1 { mem }{ $^{\mathrm{nx}}$\ldmemory { val Υ_3 $^{\mathrm{nx}}$\ldmemspecstash val Υ_4 } }$\rangle$
This code is used in section 17i.

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

18c ⟨ Add a memory spec 18c ⟩ =
$\pi_2(\Upsilon_3) \mapsto v_a \pi_3(\Upsilon_3) \mapsto v_b$
$\pi_5(\Upsilon_3) \mapsto v_c$
$\Upsilon \leftarrow \langle$val Υ_1 $^{\mathrm{nx}}$\ldmemspecseparator { val v_a }{ val v_b }val $v_c\rangle$
This code is used in section 17i.

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

18e ⟨ Grammar rules 11b ⟩ + =
origin_spec :
ORIGIN ⇐ *mustbe_exp* $\Upsilon \leftarrow \langle^{\mathrm{nox}}$\ldoriginspec { val Υ_3 }$\rangle$

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

attributes _opt :
○ $\Upsilon \leftarrow \langle\rangle$
(*attributes_list*) $\Upsilon \leftarrow \langle$val $\Upsilon_2\rangle$

attributes_list :
attributes_string $\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$
attributes_list attributes_string $\Upsilon \leftarrow \langle$val Υ_1 $^{\mathrm{nox}}$\ldspace val $\Upsilon_2\rangle$

attributes_string :
name $\Upsilon \leftarrow \langle^{\mathrm{nox}}$\ldattributes { val Υ_1 }$\rangle$
¬ name $\Upsilon \leftarrow \langle^{\mathrm{nox}}$\ldattributesneg { val Υ_2 }$\rangle$

startup :
STARTUP (*filename*)

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

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

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

low_level_library_name_***list*** :
*low_level_library*_name_*list* ,$_{\text{opt\_}}$ *filename*
∘

floating_point_support :
`FLOAT`
`NOFLOAT`

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

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

19a SECTIONS and expressions

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

⟨ Grammar rules 11b ⟩ + =

exp :

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

20a More atomic expression types specific to the linker's function.

⟨ Grammar rules 11b ⟩ + =

exp :

DEFINED (name)	
INT	
SIZEOF_HEADERS	
ALIGNOF (name)	
SIZEOF (name)	
ADDR (name)	
LOADADDR (name)	
CONSTANT (name)	
ABSOLUTE (*exp*)	
ALIGN_K (*exp*)	$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\backslash\mathtt{mathop}\{\backslash\mathtt{hbox}\{{}^{\mathrm{nx}}\backslash\mathtt{ssf\ align}\}\}(\mathrm{val}\,\Upsilon_3)\rangle$
ALIGN_K (*exp* , *exp*)	
DATA_SEGMENT_ALIGN (*exp* , *exp*)	
DATA_SEGMENT_RELRO_END (*exp* , *exp*)	
DATA_SEGMENT_END (*exp*)	
SEGMENT_START (name , *exp*)	
BLOCK (*exp*)	
name	$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\backslash\mathtt{ldregexp}\{\,\mathrm{val}\,\Upsilon_1\,\}\rangle$
MAX_K (*exp* , *exp*)	
MIN_K (*exp* , *exp*)	
ASSERT_K (*exp* , name)	
ORIGIN (name)	
LENGTH (name)	
LOG2CEIL (*exp*)	

20b ⟨ Process a primitive conditional 20b ⟩ =

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

This code is used in section 19a.

20c ⟨ Grammar rules 11b ⟩ + =

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$_{\mathrm{opt\_}}$:

ALIGN_K (*exp*)	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

align_with_input$_{\mathrm{opt\_}}$:

ALIGN_WITH_INPUT	$\Upsilon \leftarrow \langle\mathtt{align\ with\ input}\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

subalign$_{\mathrm{opt\_}}$:

SUBALIGN (*exp*)	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_3\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

sect_constraint :

ONLY_IF_RO	$\Upsilon \leftarrow \langle\mathtt{only\_if\_ro}\rangle$
ONLY_IF_RW	$\Upsilon \leftarrow \langle\mathtt{only\_if\_rw}\rangle$
SPECIAL	$\Upsilon \leftarrow \langle\mathtt{special}\rangle$
∘	$\Upsilon \leftarrow \langle\rangle$

21a *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 $,_{\text{opt}}$ 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 11b ⟩ + =

section :

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

21b ⟨ Record a named section 21b ⟩ =

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

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

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

This code is used in section 21a.

21c ⟨ Record an overlay section 21c ⟩ =

This code is used in section 21a.

21d ⟨ Grammar rules 11b ⟩ + =

type :

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

atype :

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

22a *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 11b ⟩ + =

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

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

exp :

:

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

○

NOCROSSREFS

memspec$_{\text{\_opt}}$:

> name	$\Upsilon \leftarrow \langle$ val $\Upsilon_2 \rangle$
○	$\Upsilon \leftarrow \langle\rangle$

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

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

overlay_section :

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

22b ⟨ Add another pheader 22b ⟩ =

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

This code is used in section 22a.

22c ⟨ Grammar rules 11b ⟩ + =

phdrs :

PHDRS { *phdr_list* }

phdr_list :

○

phdr_list phdr

phdr :

name	\ldlex@expression
phdr_type phdr_qualifiers	\ldlex@popstate
;	

phdr_type :

exp

phdr_qualifiers :

○

name *phdr_val phdr_qualifiers*

AT (*exp*) *phdr_qualifiers*

phdr_val :

○

(*exp*)

23a Other types of script files

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

⟨ Dynamic list file rules 23a ⟩ =

dynamic_list_file :
- ∘ `\ldlex@version@file` `\ldlex@popstate`
  dynamic_list_nodes

dynamic_list_nodes :
dynamic_list_node
dynamic_list_nodes dynamic_list_node

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

dynamic_list_tag :
vers_defns `;`

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

⟨ Version file rules 23b ⟩ =

version_script_file :
- ∘ `\ldlex@version@file` `\ldlex@popstate`
  vers_nodes

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

⟨ Grammar rules 11b ⟩ + =

version :
- ∘ `\ldlex@version@script` `\ldlex@popstate`
  VERSION$_{\mathrm{K}}$ `{` *vers_nodes* `}`

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
    ;opt_:  o | ;
```

3
The lexer

25a 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` 25a ⟩ =

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

⟨ ld token regular expressions 28b ⟩

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

25b ⟨ ld lexer options 25b ⟩ =

```
⟨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 25a.

25c ⟨ ld lexer C preamble 25c ⟩ =

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

This code is used in section 25a.

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

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

This code is used in section 25a.

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

⟨ `ld` lexer definitions 26b ⟩ =

⟨ `ld` lexer states 26c ⟩

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

This code is used in section 25a.

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

⟨ `ld` lexer states 26c ⟩ =

```
⟨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 26b.

26d Macros for lexer functions

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

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

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

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

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 26d ⟩ =

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

See also sections 27a, 27b, 27c, 28a, 35b, 35e, and 36b.

This code is used in section 8a.

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

⟨ Additional macros for the `ld` lexer/parser 26d ⟩ + =

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

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

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

⟨ Additional macros for the `ld` lexer/parser 26d ⟩ + =

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

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

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

⟨ Additional macros for the `ld` lexer/parser 26d ⟩ + =

```
\def\ldcomment#1\yylextail{%
```

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

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

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

We would like to simulate the file inclusion by inserting the appropriate end of file marker for the lexer (a double `\yyeof`). After the relevant production completes, the marker has to be cleaned up from the input stream (the lexer is designed to leave it intact). The macros below are designed to handle this assignment.

⟨ Additional macros for the `ld` lexer/parser 26d ⟩ + =

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

28b Regular expressions

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

Variable names and algebraic operations come first.

⟨ `ld` token regular expressions 28b ⟩ =

BOTH SCRIPT EXPRESSION VERS_START VERS_NODE VERS_SCRIPT `/*`	`\ldcomment` **continue**
DEFSYMEXP `-`	**return**$_c$
DEFSYMEXP `+`	**return**$_c$
DEFSYMEXP ⟨`FILENAMECHAR`$_1$⟩ ⟨`SYMBOLCHARN`⟩$_*$	**return**$_{vp}$ name
DEFSYMEXP `=`	**return**$_c$
MRI EXPRESSION `$` ([0–9A–Fa–f])$_+$	⟨ Return an absolute hex constant 34a ⟩
MRI EXPRESSION ([0–9A–Fa–f])$_+$(`H` | `h` | `X` | `x` | `B` | `b` | `O` | `o` | `D` | `d`)	⟨ Return a constant in a specific radix 34b ⟩
SCRIPT DEFSYMEXP MRI BOTH EXPRESSION (((`$` \| `0`[`xX`]) ([0–9A–Fa–f])$_+$) \| (([0–9])$_+$)) (`M` \| `K` \| `m` \| `k`)$_?$	⟨ Return a constant with a multiplier 35a ⟩
BOTH SCRIPT EXPRESSION MRI `]`	**return**$_c$
BOTH SCRIPT EXPRESSION MRI `[`	**return**$_c$
BOTH SCRIPT EXPRESSION MRI `<<=`	**return**$_p$ $\Lleftarrow$
BOTH SCRIPT EXPRESSION MRI `>>=`	**return**$_p$ $\Lleftarrow$
BOTH SCRIPT EXPRESSION MRI `||`	**return**$_p$ $\vee$

BOTH SCRIPT EXPRESSION MRI
==
$\mathbf{return}_p =$

BOTH SCRIPT EXPRESSION MRI
!=
$\mathbf{return}_p \neq$

BOTH SCRIPT EXPRESSION MRI
>=
$\mathbf{return}_p \geqslant$

BOTH SCRIPT EXPRESSION MRI
<=
$\mathbf{return}_p \leqslant$

BOTH SCRIPT EXPRESSION MRI
<<
$\mathbf{return}_p \ll$

BOTH SCRIPT EXPRESSION MRI
>>
$\mathbf{return}_p \gg$

BOTH SCRIPT EXPRESSION MRI
+=
$\mathbf{return}_p \overset{+}{\Leftarrow}$

BOTH SCRIPT EXPRESSION MRI
-=
$\mathbf{return}_p \overset{-}{\Leftarrow}$

BOTH SCRIPT EXPRESSION MRI
*=
$\mathbf{return}_p \overset{\times}{\Leftarrow}$

BOTH SCRIPT EXPRESSION MRI
/=
$\mathbf{return}_p \overset{\div}{\Leftarrow}$

BOTH SCRIPT EXPRESSION MRI
&=
$\mathbf{return}_p \overset{\&}{\Leftarrow}$

BOTH SCRIPT EXPRESSION MRI
|=
$\mathbf{return}_p \overset{\vee}{\Leftarrow}$

BOTH SCRIPT EXPRESSION MRI
&&
$\mathbf{return}_p \wedge$

BOTH SCRIPT EXPRESSION MRI
>
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
,
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
&
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
|
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
~
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
!
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
?
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
*
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
+
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
-
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
/
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
%
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
<
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
=
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
}
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
{
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
)
$\mathbf{return}_c$

BOTH SCRIPT EXPRESSION MRI
(
$\mathbf{return}_c$

```
BOTH SCRIPT EXPRESSION MRI
:                                   return_c
BOTH SCRIPT EXPRESSION MRI
;                                   return_c
```

See also sections 30a and 35f.

This code is used in section 25a.

30a 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 28b ⟩ + =

```
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$_{_\mathrm{K}}$

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$_{_\mathrm{K}}$

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 ⟨`FILENAMECHAR`$_1$⟩ ⟨`NOCFILENAMECHAR`⟩$_*$	**return**$_{vp}$ name
BOTH ⟨`FILENAMECHAR`$_1$⟩ ⟨`FILENAMECHAR`⟩$_*$	**return**$_{vp}$ name
BOTH `-l` ⟨`FILENAMECHAR`⟩$_+$	**return**$_{vp}$ name
EXPRESSION ⟨`FILENAMECHAR`$_1$⟩ ⟨`NOCFILENAMECHAR`⟩$_*$	**return**$_{vp}$ name
EXPRESSION `-l` ⟨`NOCFILENAMECHAR`⟩$_+$	**return**$_{vp}$ name
SCRIPT ⟨`WILDCHAR`⟩$_*$	⟨ Skip a possible comment and return a name 35c ⟩
EXPRESSION BOTH SCRIPT VERS_NODE `" [^"]`$^c{}_*$`"`	⟨ Return the name inside quotes 35d ⟩
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 `{`	**enter**(`VERS_SCRIPT`)**return**$_c$

34a 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 34a ⟩ =

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

This code is used in section 28b.

34b ⟨ Return a constant in a specific radix 34b ⟩ =

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

This code is used in section 28b.

35a ⟨ Return a constant with a multiplier 35a ⟩ =

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

This code is used in section 28b.

35b ⟨ Additional macros for the `ld` lexer/parser 26d ⟩ + =

```
\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}%
}
```

35c *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 35c ⟩ =

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

This code is used in section 30a.

35d *No matter the state, quotes give what's inside.*

⟨ Return the name inside quotes 35d ⟩ =

```
\ldstripquotes return_vp name
```

This code is used in section 30a.

35e ⟨ Additional macros for the `ld` lexer/parser 26d ⟩ + =

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

35f Some syntax specific to version scripts.

⟨ `ld` token regular expressions 28b ⟩ + =

`VERS_SCRIPT`	
`{`	**enter**(`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**$_\omega$ `\versnodenesting` < 0_R **enter**(`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⟩]+
⟨EOF⟩
SCRIPT MRI VERS_START VERS_SCRIPT VERS_NODE
.
EXPRESSION DEFSYMEXP BOTH
.
```

continue
⟨ Process the end of (possibly included) file 36a ⟩

fatal⟨ bad character ‘val\yytext’ in script ⟩

fatal⟨ bad character ‘val\yytext’ in expression ⟩

36a ⟨ Process the end of (possibly included) file 36a ⟩ =

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

This code is used in section 35f.

36b 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 26d ⟩ + =

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

\def\ldfile@open@command@file#1{%
    \advance\includestackptr\@ne
    \appendlnx\yytext@seen{\yyeof\yyeof}%
    \yytextbackuptrue
}

\def\ldlex@filename{}
```

4
Example output

37a Here is an example output of the `ld` parser designed in this document. The original linker script is presented in the section that follows. The same parser can be used to present examples of `ld` scripts in text similar to the one below.

memory

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

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 37a ⟩ =

include `file.ld`

memory

	attributes	starts at	length
⟨ Some random portion of `ld` code 40a ⟩			
`RAM`	`xrw`	$2000\,0000_{16}$	20 Kb
`FLASH`	`rx`	$800\,0000_{16}$	128 Kb
`ASH`	`rx`	8 001 000	128 Kb
⟨ Some random portion of `ld` code 40a ⟩			
`CLASH`	`rx`	700 000	128 Kb
`ASH`	`rx`	$800\,0000_{16}$	128 Kb
`CLASH`	`rx`	$70\,0000_{01}$	128 Kb

include `file.mem`

⟨ Some random portion of `ld` code 40a ⟩

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

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

⟨ Some random portion of `ld` code 40a ⟩

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

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

⟨ Some random portion of `ld` code 40a ⟩

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

```
sections  ⟨ Some random portion of ld code 40a ⟩
          .isr_vector                 align(8)[noload]  at ··              special          phdrs
              ·· ⇐ align(4)                             align ··           in FLASH as RAM  FLASH
              keep(*(.isr_vector))                      align_with_input                    RAM
              ·· ⇐ align(4)                             subalign 8                          OTHER
              fill ·· + 8
          ⟨ Some random portion of ld code 40a ⟩
          .text                                                            in FLASH as RAM
              ·· ⇐ align(4)
              *(.text)
              *(.text.*)
              *(.rodata)
              *(.rodata*)
              *(.glue_7)
              *(.glue_7t)
              ·· ⇐ align(4)
              _etext ⇐ ·· + 8
              _sidata ⇐ _etext
              provide ⟨ var1 ⇐ ·· ⟩
              provideh ⟨ var2 ⇐ ·· ⟩
              hidden ⟨ var3 ⇐ ·· ⟩
          ⟨ Some random portion of ld code 40a ⟩
          .data                                         at _sidata         in RAM
              ·· ⇐ align(4)
              _sdata ⇐ ··
              *(.data)
              *(.data.*)
              ·· ⇐ align(4)
              _edata ⇐ ··
          .bss                                                             in RAM
              ·· ⇐ align(4)
              _sbss ⇐ ··
              *(.bss)
              *(COMMON)
              ·· ⇐ align(4)
              _ebss ⇐ ··
          ⟨ Some random portion of ld code 40a ⟩
```

38a ⟨ The same example of an ld script 38a ⟩ =

```
INCLUDE file.ld

MEMORY
{
 ⟨ Some random portion of ld code 40a ⟩
  RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K
  FLASH (rx) : ORIGIN = 0x8000000, LENGTH = 128K
  ASH (rx) : ORIGIN = 8001000, LENGTH = 128K
 ⟨ Some random portion of ld code 40a ⟩
  CLASH (rx) : ORIGIN = 700000, LENGTH = 128K
  ASH (rx) : ORIGIN = $8000000, LENGTH = 128K
  CLASH (rx) : ORIGIN = 700000B, LENGTH = 128K
  INCLUDE file.mem
 ⟨ Some random portion of ld code 40a ⟩
```

```
}

_estack = 0x20005000;
_bstack = a > 0 ? NEXT(11) : 0x19;
 ⟨ Some random portion of ld code 40a ⟩
PROVIDE( var1 = . );
PROVIDE_HIDDEN( var2 = . );
 ⟨ Some random portion of ld code 40a ⟩
HIDDEN( var3 = . );
ENTRY(_entry);

SECTIONS
{
 ⟨ Some random portion of ld code 40a ⟩
    .isr_vector ALIGN(8) (NOLOAD): AT(.) ALIGN(.) ALIGN_WITH_INPUT SUBALIGN(8) SPECIAL
    {
        . = ALIGN(4);
        KEEP(*(.isr_vector))
        . = ALIGN(4);
    } > FLASH AT > RAM : FLASH : RAM : OTHER = . + 8
 ⟨ Some random portion of ld code 40a ⟩
    .text :
    {
        /* skip this comment */;
        . = ALIGN(4);
        *(.text)
        *(.text.*)
        *(.rodata)
        *(.rodata*)
        *(.glue_7)
        *(.glue_7t)
        . = ALIGN(4);
   _etext = . + 8;
   _sidata = _etext;
        PROVIDE( var1 = . );
        PROVIDE_HIDDEN( var2 = . );
        HIDDEN( var3 = . );
    } >FLASH AT > RAM

 ⟨ Some random portion of ld code 40a ⟩
    .data  : AT ( _sidata )
    {
     . = ALIGN(4);
        _sdata = . ;
        *(.data)
        *(.data.*)
     . = ALIGN(4);
    _edata = . ;
    } >RAM

    .bss :
    {
     . = ALIGN(4);
        _sbss = .;
        *(.bss)
        *(COMMON)
     . = ALIGN(4);
```

```
    _ebss = . ;
    } >RAM
  ⟨ Some random portion of ld code 40a ⟩

}
```

40a ⟨ Some random portion of ld code 40a ⟩ =

This code is used in sections 37a and 38a.

5
The name parser for ld term names

41a 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 41a ⟩ =

```
.....
⟨ Name parser C preamble 45e ⟩
.....
⟨ Bison options 42a ⟩
⟨ union ⟩ ⟨ Union of parser types 45g ⟩
.....
⟨ Name parser C postamble 45f ⟩
.....
⟨ Token and types declarations 42b ⟩

⟨ Parser productions 42c ⟩
```

41b 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 41b ⟩ =

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

This code is used in section 6a.

42a ⟨ Bison options 42a ⟩ =

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

This code is used in section 41a.

42b ⟨ Token and types declarations 42b ⟩ =

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

This code is used in section 41a.

42c 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 42c ⟩ =

full_name :

identifier_string $\mathit{suffixes}_{\mathrm{opt}}$	⟨ Compose the full name 43a ⟩
qualifier `_` *identifier_string* $\mathit{suffixes}_{\mathrm{opt}}$	⟨ Compose a qualified name 43b ⟩
«meta identifier»	⟨ Turn a «meta identifier» into a full name 43c ⟩
`'`	⟨ Make ' into a name 43d ⟩

identifier_string :

`%[a...Z0...9]*`	⟨ Attach option name 43e ⟩
`[a...Z0...9]*`	⟨ Start with an identifier 43f ⟩
`' * or ? '`	⟨ Start with a quoted string 43h ⟩
`' _ '`	⟨ Start with a _ string 43i ⟩
`' . '`	⟨ Start with a . string 44a ⟩
incomplete_identifier_string ◇ `[a...Z0...9]*`	⟨ Attach an identifier 44c ⟩

incomplete_identifier_string :

`_`	$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash idstr\{\}\{\}} \rangle$
identifier_string `_`	$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1 \rangle$
qualified_identifier_string `_`	$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1 \rangle$

qualified_identifier_string :

identifier_string `_` *qualifier*	⟨ Attach qualifier to a name 44d ⟩
qualified_identifier_string `_` *qualifier*	⟨ Attach qualifier to a name 44d ⟩

$\mathit{suffixes}_{\mathrm{opt}}$:

∘	$\Upsilon \leftarrow \langle \rangle$
`.`	$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\, {}^{\mathrm{nx}}\texttt{\textbackslash sfxnone} \rangle$
`.` *suffixes*	⟨ Attach suffixes 44h ⟩
`.` *qualified_suffixes*	⟨ Attach qualified suffixes 44i ⟩
`[0...9]*`	⟨ Attach an integer 44e ⟩
`_` `[0...9]*`	⟨ Attach a subscripted integer 44f ⟩
`_` *qualifier*	⟨ Attach a subscripted qualifier 44g ⟩

suffixes :

`[a...Z0...9]*`	⟨ Start with a named suffix 44j ⟩
`[0...9]*`	⟨ Start with a numeric suffix 44k ⟩
suffixes `.`	⟨ Add a dot separator 44l ⟩
suffixes `[a...Z0...9]*`	⟨ Attach a named suffix 45b ⟩
suffixes `[0...9]*`	⟨ Attach integer suffix 45a ⟩
qualifier `.`	$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash sfxn}\, \mathrm{val}\, \Upsilon_1\, {}^{\mathrm{nx}}\texttt{\textbackslash dotsp} \rangle$
suffixes qualifier `.`	$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1\, {}^{\mathrm{nx}}\texttt{\textbackslash sfxn}\, \mathrm{val}\, \Upsilon_2\, {}^{\mathrm{nx}}\texttt{\textbackslash dotsp} \rangle$

qualified_suffixes :
suffixes qualifier ⟨ Attach a qualifier 45c ⟩
qualifier ⟨ Start suffixes with a qualifier 45d ⟩

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

This code is used in section 41a.

43a ⟨ Compose the full name 43a ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1 \mathrm{val}\, \Upsilon_2 \rangle$ `\namechars` Υ

This code is used in section 42c.

43b ⟨ Compose a qualified name 43b ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a \pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_3 \mathrm{val}\, \Upsilon_4 {}^{\mathrm{nx}}$`\dotsp` $^{\mathrm{nx}}$`\qual {` $\mathrm{val}\, v_a {}^{\mathrm{nx}}$`\_ }{` $\mathrm{val}\, v_b$`\uscoreletter }` $\rangle$ `\namechars` Υ

This code is used in section 42c.

43c ⟨ Turn a «meta identifier» into a full name 43c ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\idstr {` $\mathrm{val}\, v_a$ `}{` $\mathrm{val}\, v_b$ `}` $\rangle$ `\namechars` Υ

This code is used in section 42c.

43d ⟨ Make ’ into a name 43d ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\chstr {’}{’}` $\rangle$ `\namechars` Υ

This code is used in section 42c.

43e ⟨ Attach option name 43e ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\optstr {` $\mathrm{val}\, v_a$ `}{` $\mathrm{val}\, v_b$ `}` $\rangle$

This code is used in section 42c.

43f ⟨ Start with an identifier 43f ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\idstr {` $\mathrm{val}\, v_a$ `}{` $\mathrm{val}\, v_b$ `}` $\rangle$

This code is used in sections 42c and 44b.

43g ⟨ Start with a tag 43g ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\idstr {<`$\mathrm{val}\, v_a$`>}{<`$\mathrm{val}\, v_b$`>}` $\rangle$

43h ⟨ Start with a quoted string 43h ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\visflag {` $^{\mathrm{nx}}$`\termvstring }{ }` $^{\mathrm{nx}}$`\chstr {` $\mathrm{val}\, v_a$ `}{` $\mathrm{val}\, v_b$ `}` $\rangle$

This code is used in section 42c.

43i ⟨ Start with a _ string 43i ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}$`\visflag {` $^{\mathrm{nx}}$`\termvstring }{ }` $^{\mathrm{nx}}$`\chstr {` $^{\mathrm{nx}}$`\_ }{_}` $\rangle$

This code is used in section 42c.

44a ⟨ Start with a `.` string 44a ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash visflag\{}\,{}^{\mathrm{nx}}\texttt{\textbackslash termvstring\}\{\}}\,{}^{\mathrm{nx}}\texttt{\textbackslash chstr\{.\}\{.\}}\rangle$
This code is used in section 42c.

44b ⟨ Turn a qualifier into an identifier 44b ⟩ =
⟨ Start with an identifier 43f ⟩

44c ⟨ Attach an identifier 44c ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a$
$v_a \leftarrow v_a +_{\mathrm{sx}} [\,{}^{\mathrm{nox}}\texttt{\textbackslash\_}\,]$
$\pi_1(\Upsilon_3) \mapsto v_b$
$v_a \leftarrow v_a +_{\mathrm{s}} v_b$
$\pi_3(\Upsilon_1) \mapsto v_b$
$v_b \leftarrow v_b +_{\mathrm{sx}} [\,\texttt{\textbackslash uscoreletter}\,]$
$\pi_2(\Upsilon_3) \mapsto v_c$
$v_b \leftarrow v_b +_{\mathrm{s}} v_c$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash idstr\{}\,\mathrm{val}\,v_a\,\texttt{\}\{}\,\mathrm{val}\,v_b\,\texttt{\}}\rangle$
This code is used in section 42c.

44d ⟨ Attach qualifier to a name 44d ⟩ =
This code is used in section 42c.

44e ⟨ Attach an integer 44e ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\,{}^{\mathrm{nx}}\texttt{\textbackslash sfxi}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 42c.

44f ⟨ Attach a subscripted integer 44f ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a \pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\,{}^{\mathrm{nx}}\texttt{\textbackslash sfxi\{}\,{}^{\mathrm{nx}}\texttt{\textbackslash\_}\,\mathrm{val}\,v_a\,\texttt{\}\{\textbackslash uscoreletter}\,\mathrm{val}\,v_b\,\texttt{\}}\rangle$
This code is used in section 42c.

44g ⟨ Attach a subscripted qualifier 44g ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a \pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\,{}^{\mathrm{nx}}\texttt{\textbackslash qual\{}\,{}^{\mathrm{nx}}\texttt{\textbackslash\_}\,\mathrm{val}\,v_a\,\texttt{\}\{\textbackslash uscoreletter}\,\mathrm{val}\,v_b\,\texttt{\}}\rangle$
This code is used in section 42c.

44h ⟨ Attach suffixes 44h ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\,\mathrm{val}\,\Upsilon_2\rangle$
This code is used in sections 42c and 44i.

44i ⟨ Attach qualified suffixes 44i ⟩ =
⟨ Attach suffixes 44h ⟩
This code is used in section 42c.

44j ⟨ Start with a named suffix 44j ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash sfxn}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 42c.

44k ⟨ Start with a numeric suffix 44k ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash sfxi}\,\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 42c.

44l ⟨ Add a dot separator 44l ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}\texttt{\textbackslash dotsp}\rangle$
This code is used in section 42c.

45a ⟨ Attach integer suffix 45a ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1{}^{\mathrm{nx}} \texttt{\textbackslash sfxi}\, \mathrm{val}\, \Upsilon_2 \rangle$
This code is used in section 42c.

45b ⟨ Attach a named suffix 45b ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1{}^{\mathrm{nx}} \texttt{\textbackslash sfxn}\, \mathrm{val}\, \Upsilon_2 \rangle$
This code is used in section 42c.

45c ⟨ Attach a qualifier 45c ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1{}^{\mathrm{nx}} \texttt{\textbackslash qual}\, \mathrm{val}\, \Upsilon_2 \rangle$
This code is used in section 42c.

45d ⟨ Start suffixes with a qualifier 45d ⟩ =
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}} \texttt{\textbackslash qual}\, \mathrm{val}\, \Upsilon_1 \rangle$
This code is used in section 42c.

45e 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 45e ⟩ =
This code is used in section 41a.

45f 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 45f ⟩ =

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

This code is used in section 41a.

45g Union of types.
⟨ Union of parser types 45g ⟩ =
This code is used in section 41a.

45h **The name scanner**

⟨ `ld_small_lexer.ll` 45h ⟩ =

```
⟨ Lexer definitions 46a ⟩
.....
⟨ Lexer C preamble 46d ⟩
.....
⟨ Lexer options 46e ⟩

⟨ Regular expressions 46f ⟩

void define_all_states(void)
{
        ⟨ Collect all state definitions 46b ⟩
}
```

46a ⟨ Lexer definitions 46a ⟩ =

```
⟨ Lexer states 46c ⟩
⟨aletter⟩                    [a–zA–Z]
⟨letter⟩                     (_ | ⟨aletter⟩)
⟨wc⟩                         ([\'"]^c \ [_a–zA–Z0–9] | \.)
⟨id⟩                         (⟨aletter⟩ | ⟨aletter⟩ (⟨aletter⟩ | [0–9])*⟨aletter⟩)
⟨id_strict⟩                  ⟨letter⟩ ((⟨letter⟩ | [-0–9])*⟨letter⟩)?
⟨meta_id⟩                    *⟨id_strict⟩ *?
⟨int⟩                        [0–9]+
```

This code is used in section 45h.

46b ⟨ Collect all state definitions 46b ⟩ =

```
#define _register_name(name)  Define_State(#name, name)    ▷ nothing for now ◁
#undef _register_name
```

This code is used in section 45h.

46c Strings and characters in directives/rules.

⟨ Lexer states 46c ⟩ =

```
⟨states-x⟩f:   SC_ESCAPED_STRING SC_ESCAPED_CHARACTER
```

This code is used in section 46a.

46d ⟨ Lexer C preamble 46d ⟩ =

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

This code is used in section 45h.

46e ⟨ Lexer options 46e ⟩ =

```
⟨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 45h.

46f ⟨ Regular expressions 46f ⟩ =

```
⟨ Scan white space 46g ⟩
⟨ Scan identifiers 47a ⟩
```

This code is used in section 45h.

46g White space skipping.

⟨ Scan white space 46g ⟩ =

```
[␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩]                                                    continue
```

This code is used in section 46f.

47a 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 47a ⟩ =

```
%(⟨aletter⟩ | [0–9] | [-_] | % | [<>])+
opt
K
ext
['._]
⟨wc⟩
⟨id⟩
⟨meta_id⟩
⟨int⟩
"
.
```

return$_v$ `%[a...Z0...9]*`
return$_v$ `opt`
return$_v$ `suffix`$_{\texttt{K\_}}$
return$_v$ `ext`
return$_c$
return$_v$ `*` or `?`
⟨ Prepare to process an identifier 47b ⟩
⟨ Prepare to process a meta-identifier 47c ⟩
return$_v$ `[0...9]*`
continue
⟨ React to a bad character 47d ⟩

This code is used in section 46f.

47b ⟨ Prepare to process an identifier 47b ⟩ =

return$_v$ `[a...Z0...9]*`

This code is used in section 47a.

47c ⟨ Prepare to process a meta-identifier 47c ⟩ =

return$_v$ «meta identifier»

This code is used in section 47a.

47d A simple routine to detect trivial scanning problems.

⟨ React to a bad character 47d ⟩ =

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

This code is used in section 47a.

6 Appendix

49a 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`.

49b 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 49b ⟩ =

⟨ C setup for `ld` grammar 50a ⟩

```
⟨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 51a ⟩

⟨ Original `ld` grammar rules 53a ⟩

50a *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 50a ⟩ =

```
#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 49b.

51a *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 51a ⟩ =

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

INT                    name                    name_L

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

⟨right : token⟩ ⩲ ⩱ ⩯ ⩰ ⇐ ⪡ ⪢ ⪤ ⪥
⟨right : token⟩ ? :
⟨left : token⟩ ∨
⟨left : token⟩ ∧
⟨left : token⟩ |
⟨left : token⟩ ⊕
⟨left : token⟩ &
⟨left : token⟩ = ≠
```

⟨**left** : *token*⟩ $< > \leqslant \geqslant$
⟨**left** : *token*⟩ $\ll \gg$
⟨**left** : *token*⟩ $+ -$
⟨**left** : *token*⟩ $\times / \div$
⟨**right**⟩ unary

end

⟨**left** : *token*⟩ `(`

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

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

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

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

INPUT_DYNAMIC_LIST

This code is used in section 49b.

53a *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 53a ⟩ =

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*( ); *ldlex_popstate*( );
  - name ⇐ *exp*

See also sections 53b, 53c, 54b, 54c, 55a, 55b, 56a, 56b, 57a, 58a, 58b, 59a, 59b, 60a, 60b, and 61a.

This code is used in section 49b.

53b Syntax within an MRI script file.

⟨ Original `ld` grammar rules 53a ⟩ + =

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

mri_script_lines :
- *mri_script_lines mri_script_command* `NEWLINE`
- ∘

53c ⟨ Original `ld` grammar rules 53a ⟩ + =

mri_script_command :

`CHIP` *exp*	
`CHIP` *exp* `,` *exp*	
name	⟨ Flag an unrecognized keyword 54a ⟩
`LIST`	*config*.*map_filename* ⇐ `"-"`;
`ORDER` *ordernamelist*	
`ENDWORD`	
`PUBLIC` name ⇐ *exp*	*mri_public*(Υ2, Υ4);
`PUBLIC` name `,` *exp*	*mri_public*(Υ2, Υ4);
`PUBLIC` name *exp*	*mri_public*(Υ2, Υ3);
`FORMAT` name	*mri_format*(Υ2);
`SECT` name `,` *exp*	*mri_output_section*(Υ2, Υ4);
`SECT` name *exp*	*mri_output_section*(Υ2, Υ3);
`SECT` name ⇐ *exp*	*mri_output_section*(Υ2, Υ4);
`ALIGN_K` name ⇐ *exp*	*mri_align*(Υ2, Υ4);
`ALIGN_K` name `,` *exp*	*mri_align*(Υ2, Υ4);
`ALIGNMOD` name ⇐ *exp*	*mri_alignmod*(Υ2, Υ4);
`ALIGNMOD` name `,` *exp*	*mri_alignmod*(Υ2, Υ4);
`ABSOLUTE` *mri_abs_name_list*	
`LOAD` *mri_load_name_list*	
`NAMEWORD` name	*mri_name*(Υ2);
`ALIAS` name `,` name	*mri_alias*(Υ2, Υ4, 0);
`ALIAS` name `,` `INT`	*mri_alias*(Υ2, 0, (**int**) Υ4.*integer*);
`BASE` *exp*	*mri_base*(Υ2);
`TRUNCATE` `INT`	*mri_truncate*((**unsigned int**) Υ2.*integer*);
`CASE` *casesymlist*	
`EXTERN` *extern_name_list*	

`INCLUDE` *filename*	*ldlex_script*( ); *ldfile_open_command_file*(Υ_2);
mri_script_lines end	*ldlex_popstate*( );
`START` name	*lang_add_entry*(Υ_2, `FALSE`);
∘	

ordernamelist :

ordernamelist , name	*mri_order*(Υ_3);
ordernamelist name	*mri_order*(Υ_2);
∘	

mri_load_name_list :

name	*mri_load*(Υ_1);
mri_load_name_list , name	*mri_load*(Υ_3);

mri_abs_name_list :

name	*mri_only_load*(Υ_1);
mri_abs_name_list , name	*mri_only_load*(Υ_3);

casesymlist :

∘	$\Upsilon \Leftarrow \Lambda$;
name	
casesymlist , name	

54a *Here is one way to deal with code that is too long to fit in an action.*

⟨ Flag an unrecognized keyword 54a ⟩ =

einfo(_("`%P%F:␣unrecognised␣keyword␣in␣MRI␣style␣script␣'%s'\n`"), Υ_1);

This code is used in section 53c.

54b Parsed as expressions so that commas separate entries.

⟨ Original `ld` grammar rules 53a ⟩ + =

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*$_1$
∘

54c *All the commands that can appear in a standard linker script.*

⟨ Original `ld` grammar rules 53a ⟩ + =

***ifile_p*$_1$** :

memory
sections
phdrs
startup
high_level_library
low_level_library
floating_point_support
statement_anywhere
version
;

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

55a ⟨ Original `ld` grammar rules 53a ⟩ + =

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_p1 }
```

sec_or_group_p1 :

```
sec_or_group_p1 section
sec_or_group_p1 statement_anywhere
○
```

statement_anywhere :

```
ENTRY ( name )
assignment end
ASSERT_K                                           ldlex_expression( );
    ( exp , name )                                 ldlex_popstate( );
```

55b The `*` and `?` cases are there because the lexer returns them as separate tokens rather than as name.

⟨ Original `ld` grammar rules 53a ⟩ + =

wildcard_name :

```
name
*
?
```

56a ⟨ Original `ld` grammar rules 53a ⟩ + =

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

sect_flag_list :
name
sect_flag_list & name

sect_flags :
`INPUT_SECTION_FLAGS (` *sect_flag_list* `)`

exclude_name_list :
exclude_name_list wildcard_name
wildcard_name

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

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

56b ⟨ Original `ld` grammar rules 53a ⟩ + =

input_section_spec :
input_section_spec_no_keep
`KEEP (`
input_section_spec_no_keep `)`

statement :
assignment end
`CREATE_OBJECT_SYMBOLS`
`;`
`CONSTRUCTORS`
`SORT_BY_NAME ( CONSTRUCTORS )`
input_section_spec
length `(` *mustbe_exp* `)`
`FILL (` *fill_exp* `)`
`ASSERT`$_{\text{K}}$
`(` *exp* `,` name `)` *end*
`INCLUDE` *filename*

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

ldlex_expression( );
ldlex_popstate( );
ldlex_script( );
ldfile_open_command_file(Υ_2);
ldlex_popstate( );

statement_list : *statement_list statement* | *statement*

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

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

fill_exp : *mustbe_exp*

fill$_{\text{opt}}$: $\Leftarrow$ *fill_exp* | ∘

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

end : `;` | `,`

assignment :
- name $\Leftarrow$ *mustbe_exp*
- name *assign_op mustbe_exp*
- `HIDDEN (` name $\Leftarrow$ *mustbe_exp* `)`
- `PROVIDE (` name $\Leftarrow$ *mustbe_exp* `)`
- `PROVIDE_HIDDEN (` name $\Leftarrow$ *mustbe_exp* `)`

`,`$_{\text{opt\_}}$: `,` | ∘

memory :
- `MEMORY {` *memory_spec_list*$_{\text{opt}}$ `}`

memory_spec_list$_{\text{opt}}$:
- *memory_spec_list*
- ∘

memory_spec_list :
- *memory_spec_list* `,`$_{\text{opt\_}}$ *memory_spec*
- *memory_spec*

memory_spec :
- name
  - *attributes*$_{\text{opt}}$ `:` *origin_spec* `,`$_{\text{opt\_}}$ *length_spec*
- `INCLUDE` *filename*

  ldlex_script( );
  ldfile_open_command_file(Υ_2);
- *memory_spec_list*$_{\text{opt}}$ end

  ldlex_popstate( );

57a ⟨ Original `ld` grammar rules 53a ⟩ + =

origin_spec :
- `ORIGIN` $\Leftarrow$ *mustbe_exp*

length_spec :
- `LENGTH` $\Leftarrow$ *mustbe_exp*

attributes$_{\text{opt}}$:
- ∘
- `(` *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*( );

58a *Rich expression syntax reproducing the one in* C.

⟨ Original ld grammar rules 53a ⟩ + =

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_K (*exp*)
ALIGN_K (*exp* , *exp*) | DATA_SEGMENT_ALIGN (*exp* , *exp*)
DATA_SEGMENT_RELRO_END (*exp* , *exp*)
DATA_SEGMENT_END (*exp*)
SEGMENT_START (name , *exp*)
BLOCK (*exp*)
name
MAX_K (*exp* , *exp*)
MIN_K (*exp* , *exp*)
ASSERT_K (*exp* , name)
ORIGIN (name)
LENGTH (name)
LOG2CEIL (*exp*)

58b ⟨ Original ld grammar rules 53a ⟩ + =

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

at$_{\text{opt\_}}$:
AT (*exp*)
∘

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

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

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

sect_constraint :
    ONLY_IF_RO
    ONLY_IF_RW
    SPECIAL
    ∘

59a 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 53a ⟩ + =

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*) | ∘ | ( )

59b 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 53a ⟩ + =

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

60a ⟨ Original `ld` grammar rules 53a ⟩ + =

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 `;`

60b This syntax is used within an external version script file.

⟨ Original `ld` grammar rules 53a ⟩ + =

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

61a This is used within a normal linker script file.

⟨ Original `ld` grammar rules 53a ⟩ + =

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`

`;`$_{\mathrm{opt\_}}$: ∘ | `;`

61b 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);
}
```

62a 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 62a ⟩ =
```
⟨ Original ld macros 63b ⟩
.....
⟨ Original ld preamble 62b ⟩
.....

⟨nounput⟩f ⋆
⟨ Ignored options 63a ⟩

⟨ Original ld regular expressions 64a ⟩
```

62b

⟨ Original `ld` preamble 62b ⟩ =
```
#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 62a.

63a ⟨ Ignored options 63a ⟩ =

```
⟨a⟩f 4000
⟨o⟩f 5000
```

This code is used in section 62a.

63b *Some convenient abbreviations for regular expressions.*

⟨ Original ld macros 63b ⟩ =

```
⟨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 62a.

63c 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 63c ⟩ =

```
⟨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
```

63d ⟨ ld postamble 63d ⟩ =

```
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( );
    }
  }
```

64a ⟨ Original `ld` regular expressions 64a ⟩ =

```
BOTH SCRIPT EXPRESSION VERS_START VERS_NODE VERS_SCRIPT
/*                                          comment( );
                                            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);
```

168

```
EXPRESSION BOTH SCRIPT
INCLUDE                    RTOKEN(INCLUDE);
BOTH SCRIPT
PHDRS                      RTOKEN(PHDRS);
EXPRESSION BOTH SCRIPT
AT                         RTOKEN(AT);
EXPRESSION BOTH SCRIPT
ALIGN_WITH_INPUT           RTOKEN(ALIGN_WITH_INPUT);
EXPRESSION BOTH SCRIPT
SUBALIGN                   RTOKEN(SUBALIGN);
EXPRESSION BOTH SCRIPT
HIDDEN                     RTOKEN(HIDDEN);
EXPRESSION BOTH SCRIPT
PROVIDE                    RTOKEN(PROVIDE);
EXPRESSION BOTH SCRIPT
PROVIDE_HIDDEN             RTOKEN(PROVIDE_HIDDEN);
EXPRESSION BOTH SCRIPT
KEEP                       RTOKEN(KEEP);
EXPRESSION BOTH SCRIPT
EXCLUDE_FILE               RTOKEN(EXCLUDE_FILE);
EXPRESSION BOTH SCRIPT
CONSTANT                   RTOKEN(CONSTANT);
MRI
# .* ⟨n⟩?                  ++lineno;
MRI
⟨n⟩                        ++lineno; RTOKEN(NEWLINE);
MRI
* .*                       ▷ MRI comment line ◁
MRI
; .*                       ▷ MRI comment line ◁
MRI
END                        RTOKEN(ENDWORD);
MRI
ALIGNMOD                   RTOKEN(ALIGNMOD);
MRI
ALIGN                      RTOKEN(ALIGN_K);
MRI
CHIP                       RTOKEN(CHIP);
MRI
BASE                       RTOKEN(BASE);
MRI
ALIAS                      RTOKEN(ALIAS);
MRI
TRUNCATE                   RTOKEN(TRUNCATE);
MRI
LOAD                       RTOKEN(LOAD);
MRI
PUBLIC                     RTOKEN(PUBLIC);
MRI
ORDER                      RTOKEN(ORDER);
MRI
NAME                       RTOKEN(NAMEWORD);
MRI
FORMAT                     RTOKEN(FORMAT);
MRI
CASE                       RTOKEN(CASE);
MRI
START                      RTOKEN(START);
MRI
LIST .*                    RTOKEN(LIST);  ▷ LIST and ignore to end of line ◁
MRI
SECT                       RTOKEN(SECT);
```

EXPRESSION BOTH SCRIPT MRI
ABSOLUTE — RTOKEN(ABSOLUTE);
MRI
end — RTOKEN(ENDWORD);
MRI
alignmod — RTOKEN(ALIGNMOD);
MRI
align — RTOKEN(ALIGN_K);
MRI
chip — RTOKEN(CHIP);
MRI
base — RTOKEN(BASE);
MRI
alias — RTOKEN(ALIAS);
MRI
truncate — RTOKEN(TRUNCATE);
MRI
load — RTOKEN(LOAD);
MRI
public — RTOKEN(PUBLIC);
MRI
order — RTOKEN(ORDER);
MRI
name — RTOKEN(NAMEWORD);
MRI
format — RTOKEN(FORMAT);
MRI
case — RTOKEN(CASE);
MRI
extern — RTOKEN(EXTERN);
MRI
start — RTOKEN(START);
MRI
list .$_*$ — RTOKEN(LIST); ▷ LIST and ignore to end of line ◁
MRI
sect — RTOKEN(SECT);
EXPRESSION BOTH SCRIPT MRI
absolute — RTOKEN(ABSOLUTE);
MRI
⟨FILENAMECHAR$_1$⟩ ⟨NOCFILENAMECHAR⟩$_*$ — ▷ Filename without commas, needed to parse MRI stuff ◁
yylval.name ⇐ *xstrdup*(*yytext*);
return NAME;

BOTH
⟨FILENAMECHAR$_1$⟩ ⟨FILENAMECHAR⟩$_*$ — *yylval.name* ⇐ *xstrdup*(*yytext*);
return NAME;

BOTH
-l ⟨FILENAMECHAR⟩$_+$ — *yylval.name* ⇐ *xstrdup*(*yytext* + 2);
return LNAME;

EXPRESSION
⟨FILENAMECHAR$_1$⟩ ⟨NOCFILENAMECHAR⟩$_*$ — *yylval.name* ⇐ *xstrdup*(*yytext*);
return NAME;

EXPRESSION
-l ⟨NOCFILENAMECHAR⟩$_+$ — *yylval.name* ⇐ *xstrdup*(*yytext* + 2);
return LNAME;

SCRIPT
⟨WILDCHAR⟩∗

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

```
if (yytext[0] = '/' ∧ yytext[1] = '*') {
    yyless(2);
    comment( );
}
else {
    yylval.name ⇐ xstrdup(yytext);
    return NAME;
}
```

EXPRESSION BOTH SCRIPT VERS_NODE
" ["]^c_∗ "

▷ No matter the state, quotes give what's inside ◁

```
yylval.name ⇐ xstrdup(yytext + 1);
yylval.name[yyleng − 2] ⇐ 0;
return NAME;
```

BOTH SCRIPT EXPRESSION
⟨n⟩

```
lineno++;
```

MRI BOTH SCRIPT EXPRESSION
[␣⟨t⟩⟨r⟩]+

VERS_NODE VERS_SCRIPT
[:,;]

```
return ∗yytext;
```

VERS_NODE
global

```
RTOKEN(GLOBAL);
```

VERS_NODE
local

```
RTOKEN(LOCAL);
```

VERS_NODE
extern

```
RTOKEN(EXTERN);
```

VERS_NODE
⟨V_IDENTIFIER⟩

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

VERS_SCRIPT
⟨V_TAG⟩

```
yylval.name ⇐ xstrdup(yytext);
return VERS_TAG;
```

VERS_START
{

```
BEGIN(VERS_SCRIPT); return ∗yytext;
```

VERS_SCRIPT
{

```
BEGIN(VERS_NODE);
vers_node_nesting ⇐ 0;
return ∗yytext;
```

VERS_SCRIPT
}

```
return ∗yytext;
```

VERS_NODE
{

```
vers_node_nesting++; return ∗yytext;
```

VERS_NODE
}

```
if (−−vers_node_nesting < 0) BEGIN(VERS_SCRIPT);
return ∗yytext;
```

VERS_START VERS_NODE VERS_SCRIPT
[⟨n⟩]

```
lineno++;
```

VERS_START VERS_NODE VERS_SCRIPT
.∗

```
; ▷ Eat up comments ◁
```

VERS_START VERS_NODE VERS_SCRIPT
[␣⟨t⟩⟨r⟩]+

```
; ▷ Eat up whitespace ◁
```

```
⟨EOF⟩                                          include_stack_ptr −−;
                                               if (include_stack_ptr = 0) yyterminate();
                                               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 62a.

72a Switch `flex` to reading script file *name*, open on *file*, saving the current input info on the include stack.

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

See also sections 72b, 73a, 73b, 74a, 74b, 74c, and 75a.

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

```
⟨ Supporting C code 72a ⟩ + =
  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;
}
```

73a Switch `flex` to reading from *string*, saving the current input info on the include stack.

⟨ Supporting C code 72a ⟩ + =

```
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);
}
```

73b Functions to switch to a different `flex` start condition, saving the current start condition on *state_stack*.

⟨ Supporting C code 72a ⟩ + =

```
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);
}
```

74a Return the current file name, or the previous file if no file is current.

```
⟨ Supporting C code 72a ⟩ +=
const char *ldlex_filename(void)
{
  return file_name_stack[include_stack_ptr − (include_stack_ptr ≠ 0)];
}
```

74b Place up to *max_size* characters in *buf* and return either the number of characters read, or 0 to indicate `EOF`.

```
⟨ Supporting C code 72a ⟩ +=
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;
}
```

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

```
⟨ Supporting C code 72a ⟩ +=
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;
    }
```

```
  }
}
```

75a Warn the user about a garbage character *what* in the input in context *where*.

```
⟨ Supporting C code 72a ⟩ +=
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);
}
```

7 Index

77a This section lists the variable names and (in some cases) the keywords used inside the 'language sections' of the CWEB source. It takes advantage of the built-in facility of CWEB to supply references for both definitions (set in *italic*) as well as uses for each C identifier in the text.

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

memory		**attributes**	**starts at**	**length**
	$\texttt{MEMORY}_1$	`xrw`	$2000\,0000_{16}$	20 Kb
	$\texttt{MEMORY}_2$	`rx`	$800\,0000_{16}$	128 Kb

$\_var\_1 \Leftarrow 2000\,5000_{16}$

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

Bison, Flex, and TEX indices

FLEX INDEX

TEX INDEX

A LIST OF ALL SECTIONS

⟨ Add a KEEP statement 16f ⟩ Used in section 16d.
⟨ Add a dot separator 44l ⟩ Used in section 42c.
⟨ Add a memory spec 18c ⟩ Used in section 17i.

⟨ Add a plain section spec 16c ⟩ Used in section 15d.
⟨ Add a wildcard spec to a list of files 16a ⟩ Used in section 15d.
⟨ Add another pheader 22b ⟩ Used in section 22a.
⟨ Add the next command 13a ⟩ Used in section 12b.
⟨ Add the next section chunk 14b ⟩ Used in section 13b.
⟨ Additional macros for the `ld` lexer/parser 26d, 27a, 27b, 27c, 28a, 35b, 35e, 36b ⟩ Used in section 8a.
⟨ Attach a named suffix 45b ⟩ Used in section 42c.
⟨ Attach a qualifier 45c ⟩ Used in section 42c.
⟨ Attach a statement to a statement list 17a ⟩ Used in section 16d.
⟨ Attach a subscripted integer 44f ⟩ Used in section 42c.
⟨ Attach a subscripted qualifier 44g ⟩ Used in section 42c.
⟨ Attach an identifier 44c ⟩ Used in section 42c.
⟨ Attach an integer 44e ⟩ Used in section 42c.
⟨ Attach integer suffix 45a ⟩ Used in section 42c.
⟨ Attach option name 43e ⟩ Used in section 42c.
⟨ Attach qualified suffixes 44i ⟩ Used in section 42c.
⟨ Attach qualifier to a name 44d ⟩ Used in section 42c.
⟨ Attach suffixes 44h ⟩ Used in sections 42c and 44i.
⟨ Begin namespace setup 5a ⟩ Used in section 8a.
⟨ Bison options 42a ⟩ Used in section 41a.
⟨ Carry on 14e ⟩ Used in sections 13b, 15d, 16b, 16d, 17b, and 17i.
⟨ Close the file 15b ⟩ Used in sections 11d, 13b, 16d, 17i, and 21a.
⟨ Collect all state definitions 46b ⟩ Used in section 45h.
⟨ Collect state definitions for the `ld` lexer 26a ⟩ Used in section 25a.
⟨ Compose a qualified name 43b ⟩ Used in section 42c.
⟨ Compose the full name 43a ⟩ Used in section 42c.
⟨ Create a wildcard name 15e ⟩ Used in section 15c.
⟨ Declare a named memory region 18d ⟩ Used in section 17i.
⟨ Define the bootstrapping mode 4a ⟩ Used in section 8a.
⟨ Define the normal mode 6a, 6b, 6c ⟩ Used in section 8a.
⟨ Dynamic list file rules 23a ⟩
⟨ Example `ld` script 37a ⟩
⟨ Flag an unrecognized keyword 54a ⟩ Used in section 53c.
⟨ Form a statement 14d ⟩ Used in sections 13b and 16d.
⟨ Form an `ENTRY` statement 14c ⟩ Used in section 13b.
⟨ Form an input section spec 16e ⟩ Used in section 16d.
⟨ Form the `MEMORY` group 18a ⟩ Used in section 17i.
⟨ Form the `SECTIONS` group 14a ⟩ Used in section 13b.
⟨ Grammar rules 11b, 15c, 15d, 16d, 17b, 17c, 17i, 18e, 19a, 20a, 20c, 21a, 21d, 22a, 22c, 23c ⟩ Used in section 10c.
⟨ Ignored grammar rules 11a ⟩
⟨ Ignored options 63a ⟩ Used in section 62a.
⟨ Initialize `ld` parsers 7a ⟩ Used in section 6a.
⟨ Inline symbol definitions 11c ⟩
⟨ Lexer C preamble 46d ⟩ Used in section 45h.
⟨ Lexer definitions 46a ⟩ Used in section 45h.
⟨ Lexer options 46e ⟩ Used in section 45h.
⟨ Lexer states 46c ⟩ Used in section 46a.
⟨ Make ’ into a name 43d ⟩ Used in section 42c.
⟨ Modified name parser for `ld` grammar 41b ⟩ Used in section 6a.
⟨ Name parser C postamble 45f ⟩ Used in section 41a.
⟨ Name parser C preamble 45e ⟩ Used in section 41a.

⟨ GNU `ld` script rules 12a, 12b, 13b ⟩ Used in section 10c.
⟨ MRI style script rules 11d ⟩
⟨ `ld` lexer C preamble 25c ⟩ Used in section 25a.
⟨ `ld` lexer definitions 26b ⟩ Used in section 25a.
⟨ `ld` lexer options 25b ⟩ Used in section 25a.
⟨ `ld` lexer states 26c ⟩ Used in section 26b.
⟨ `ld` parser C postamble 10b ⟩ Used in section 9a.
⟨ `ld` parser C preamble 9c ⟩ Used in section 9a.
⟨ `ld` parser `bison` options 9b ⟩ Used in section 9a.
⟨ `ld` parser productions 10c ⟩ Used in section 9a.
⟨ `ld` postamble 63d ⟩
⟨ `ld` states 63c ⟩
⟨ `ld` token regular expressions 28b, 30a, 35f ⟩ Used in section 25a.
⟨ `ld_small_lexer.ll` 45h ⟩
⟨ `ld_small_parser.yy` 41a ⟩
⟨ `ldl.ll` 25a ⟩
⟨ `ldman.stx` 8a ⟩
⟨ `ldp.yy` 9a ⟩

Contents (ldman)