Compiler Documentation

Thomas Maloney
net-id tmaloney@iastate.edu

March 2021

1 Part 0 Documentation

1.1 main.c

This file contains the entry point of the compiler. It passes the arguments from the command line to a method that parses them (described in section 1.2) and returns a `struct` that contains info about what flags were passed and what the possible output file might be (should that have been given). I then check which flag got passed in and whether the output should be written to a provided file or just `stdout`. The only `mode` flag that currently gets handled is `-0`.

1.2 args_parser

The header file contains a `struct`, an `enum` (both described in section 1.3), and a method signature that gets implemented in `args_parser.c`. Here I use `getopt` from `unistd.h` to read/parse each flag and then record which flag and whether there was an output file to a `struct` that I return. The potential output file gets passed back as an out-parameter. If no flags were passed in, or one that doesn't exist, I print out the string that describes how to use the program to `stderr`.

1.3 Data Structures

There is one `struct` and one `enum` that have been defined so far:

- (`struct`) `parsed_args_t`: This data type holds what `mode` flag gets passed in by way of the `mode_e enum`. It also contains a flag that signals if there is a specific output file that should be written to instead of `stdout`.
- (`enum`) `mode_e`: This just represents the possible `mode` flags that can be passed in plus an extra one that is set by default to make it easier to check if none of the required flags were passed in.

2 Part 1 Documentation

2.1 main.c

For the most part, everything here is pretty similar to what's described in section 1.1. That is, not much has really changed. The one thing that has, however is that it now grabs the parsed file names that get stored in a `parsed_args_t` instance after which it sets a global variable equal to the array. This allows the generated lexer file to access the list of files it needs to lex.

2.2 lexer.l

Contains the regex rules for matching tokens. There is also a function in it that allows `main.c` to set the starting input file from an `extern` string array that also gets pulled in from `main.c`. Currently the lexer will just ignore preprocessor symbols.

2.3 log_utils

Provides two function for logging data (info and errors) to the output stream, templated to match the assignment's specifications.

2.4 args_parser

Pretty similar to its implementation described in section 1.2. A new feature in it is the ability to parse the input files from the command line and store them in an array.

2.5 Data Structures

There is one `struct` and one `enum` that have been defined so far:

- (`struct`) `parsed_args_t`: This data type is still similar to how it is described in section 1.3. It has only changed in that it now keeps track of the input files.
- (`enum`) `mode_e`: Same as described in section 1.3.
- (`enum`) `token_e`: Enumerated list of syntax token types.

3 Part 2 Documentation

3.1 main.c

The biggest difference from how this was implemented in section 2.1 is all of the logging is done in `main.c` instead of the lexer. All of the logging for the parser is also done in `main.c`. Notable diffrences that came about from this new

approach is that we no longer log tokens or symbols as we come across them but rather build a list (be it of tokens or of symbols) first. These lists get passed back to `main.c` where we now call the logging methods on them. This change is actually what allows us to have backwards compatability with the `-l` flag.

There are also a lot of extern variables that are stored in `main.c` to make interfacing with flex and bison less painful. Another responsibility that has been shifted onto `main.c` is switching files for flex instead of doing it in the EOF handler.

3.2 args_parser

Almost identical to its implementation described in section 2.4. The only change was that in the handler for the `-2` flag, we store the input files as described in section 2.4.

3.3 mycc.y

This file contains the grammar definition for the parser and keeps a stack of:

- strings for variables
- strings for parameter variables
- strings for struct fields
- `struct_decl_symbol_t`s for `struct`s

Depending on which production rule ends up getting matched, the stack of strings for variables either gets appended to a function symbol (if it is a local variable) or it gets appended to the list of global variables. Likewise for the stack of `struct_decl_symbol_t`s for `struct`s. The data types that these stacks get appended to are described in both section 3.7 and section 3.8.

3.4 lexer.l

Updated the regex to properly match a few tokens that it missed before. The biggest architectural change from its implementation in section 2.2 is that we no longer immediately write the token to stdout. We instead build a list of them using the new `token_list_node_t` which is covered briefly in section 3.5. We also return a token each time we lex one so that our bison parser can utilize it. Another thing that has changed is that instead of handling switching files to lex from inside the EOF handler, that responsibility has been moved into `main.c`.

3.5 token_list

The files `token_list.h` and `token_list.c` contain the declaration and implementation details of the `token_list_node_t` struct. The purpose of this data type is to act as a linked list containing information about each token that is lexed from the lexer. The struct contains fields for:

- The filename where the token was lexed.
- The line number in the file that the token can be found.
- The text that the token represents.
- The token type (an enum generated by bison).
- Whether or not this token represents an error.
- The error description (should this token represent an error).
- A pointer to the next entry in the token list.

3.6 log_utils

In terms of its purpose, it's similar to how it was as described in section 2.3. The biggest change was the addition of logging methods for logging the required symbols generated by the parser. Each logging method also now requires the output file for which it should write to.

3.7 crude_symbols_list

Since the parser gives us symbols from the bottom of the production rules going up, we can't just print the symbols as we see them. The purpose of both `crude_symbols_list.h` and `crude_symbols_list.c` is to define and implement the types:

- `struct_decl_symbol_t`
- `func_decl_symbol_t`
- `func_proto_symbol_t`
- `parse_error_symbol_t`
- `symbol_data_t`
- `symbol_type_t`
- `symbol_parse_list_node_t`
- `symbol_parse_list_t`

All (except `symbol_parse_list_node_t`, and `symbol_parse_list_t`) of which are described in section 3.8.

The purpose of the `symbol_parse_list_node_t` type is to represent a node in a linked list that contains information about either a function definition, a function prototype, or some parsing error. The `symbol_parse_list_node_t` contains a member of type `symbol_type_t` which tells us which symbol type

information it contains, which, in turn tells us which field from the union type to grab.

The purpose of the `symbol_parse_list_t` is to keep a list of previously mentioned data type which corresponds to the function definitions/prototypes/errors the parser encounterd in the "current" file. It also contains an array of `char*` (strings) that contain the identifiers for the global variables in a given file. Likewise it also contains an array of `struct_decl_symbol_t*` that contain the global `struct`s in a given file.

3.8 Data Structures

Any data structures not listed here have not been listed since they have not changed from how they were described in section 2.5. The exception being `token_e` since with the introduction of bison, it was no longer necessary and thus removed.

The data structures so far:

- (`struct`) `struct_decl_symbol_t`: A `struct` to represent `struct`s. Contains fields to store the name of a `struct`, how many members it has, and the members' identifiers.
- (`struct`) `func_decl_symbol_t`: A type that represents a function definition symbol. Contains fields to store the name of the function, the list of parameters, the list of local variables, and the list of local `struct`s.
- (`struct`) `func_proto_symbol_t`: A type that represents a function prototype declaration. Contains fields to store the name of the function and a list of its parameters.
- (`struct`) `parse_error_symbol_t`: A type that represents an error from the parser. Conatins fields to store the file name where the error occured, the line number in that file, the text that caused the error, and the message returned by the parser.
- (`union`) `symbol_data_t`: A union type that makes it possible to store `func_decl_symbol_t`, `func_proto_symbol_t`, and `parse_error_symbol_t` types in the same symbol list. Sort of like quasi-polymorphism. (But not entirely)
- (`enum`) `symbol_type_t`: Enumerated list type of possible symbol types that a `symbol_parse_list_node_t` could be.
- (`struct`) `symbol_parse_list_node_t`: See section 3.7.
- (`struct`) `symbol_parse_list_t`: See section 3.7.
- (`struct`) `token_list_node_t`: See section 3.5.