```
 1: #include <assert.h>
 2: #include <inttypes.h>
 3: #include <stdarg.h>
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6: #include <string.h>
 7:
 8: #include "astree.h"
 9: #include "stringset.h"
10: #include "lyutils.h"
11:
12: astree* new_astree (int symbol, int filenr, int linenr,
13:         int offset, const char* lexinfo) {
14:     astree* tree = new astree();
15:     tree->symbol = symbol;
16:     tree->filenr = filenr;
17:     tree->linenr = linenr;
18:     tree->offset = offset;
19:     tree->blocknr = 0;
20:     tree->attributes = 0;
21:     tree->struct_node = NULL;
22:     tree->lexinfo = intern_stringset (lexinfo);
23:     DEBUGF ('f', "astree %p->{%d:%d.%d: %s: \"%s\"}\n",
24:             tree, tree->filenr, tree->linenr, tree->offset,
25:             get_yytname (tree->symbol), tree->lexinfo->c_str());
26:     return tree;
27: }
28:
29: astree* adopt1 (astree* root, astree* child) {
30:     root->children.push_back (child);
31:     DEBUGF ('a', "%p (%s) adopting %p (%s)\n",
32:             root, root->lexinfo->c_str(),
33:             child, child->lexinfo->c_str());
34:     return root;
35: }
36:
37: astree* adopt2 (astree* root, astree* left, astree* right) {
38:     adopt1 (root, left);
39:     adopt1 (root, right);
40:     return root;
41: }
42:
43: astree* adopt3 (astree* root, astree* one, astree* two, astree* three) {
44:     adopt1 (root, one);
45:     adopt1 (root, two);
46:     adopt1 (root, three);
47:     return root;
48: }
49:
50: astree* adopt1sym (astree* root, astree* child, int symbol) {
51:     root = adopt1 (root, child);
52:     root->symbol = symbol;
53:     return root;
54: }
55:
56: astree* adopt2sym (astree* root,
57:             astree* left, astree* right, int symbol) {
58:     root = adopt1 (root, left);
```

```
 59:     root = adopt1 (root, right);
 60:     root->symbol = symbol;
 61:     return root;
 62: }
 63:
 64: astree* adopt3sym (astree* root, astree* one, astree* two,
 65:                     astree* three, int symbol) {
 66:     root = adopt1 (root, one);
 67:     root = adopt1 (root, two);
 68:     root = adopt1 (root, three);
 69:     root->symbol = symbol;
 70:     return root;
 71: }
 72:
 73: static void dump_node (FILE* outfile, astree* node) {
 74:     fprintf (outfile, "%4lu%4lu.%.03lu %4d  %-15s (%s)\n",
 75:             node->filenr, node->linenr, node->offset,
 76:             node->symbol, get_yytname(node->symbol),
 77:             node->lexinfo->c_str());
 78:     bool need_space = false;
 79:     for (size_t child = 0; child < node->children.size();
 80:             ++child) {
 81:         if (need_space) fprintf (outfile, " ");
 82:         need_space = true;
 83:         fprintf (outfile, "%p", node->children.at(child));
 84:     }
 85: }
 86:
 87: static void dump_astree_rec (FILE* outfile, astree* root,
 88:                              int depth) {
 89:     if (root == NULL) return;
 90:     int i;
 91:     const char *tname = get_yytname (root->symbol);
 92:     if (strstr (tname, "TOK_") == tname) tname += 4;
 93:     for (i = 0; i < depth; i++) fprintf(outfile, "|  ");
 94:     fprintf(outfile, "%s \"%s\" %lu.%lu.%lu\n",
 95:         tname, root->lexinfo->c_str(),
 96:         root->filenr, root->linenr, root->offset);
 97:     /*fprintf (outfile, "%*s%s\n", depth * 3, "",
 98:         root->lexinfo->c_str());*/
 99:     for (size_t child = 0; child < root->children.size(); ++child) {
100:       dump_astree_rec (outfile, root->children[child], depth + 1);
101:    }
102: }
103:
104: void dump_astree (FILE* outfile, astree* root) {
105:     dump_astree_rec (outfile, root, 0);
106:     fflush (NULL);
107: }
108:
109: void yyprint (FILE* outfile, unsigned short toknum,
110:         astree* yyvaluep) {
111:     if (is_defined_token (toknum)) {
112:         dump_node (outfile, yyvaluep);
113:     }else {
114:         // handle error
115:     }
116:     fflush (NULL);
```

```
117: }
118:
119: void free_ast (astree* root) {
120:     while (not root->children.empty()) {
121:         astree* child = root->children.back();
122:         root->children.pop_back();
123:         free_ast (child);
124:     }
125:     DEBUGF ('f', "free [%p]-> %d:%d.%d: %s: \"%s\")\n",
126:             root, root->filenr, root->linenr, root->offset,
127:             get_yytname (root->symbol), root->lexinfo->c_str());
128:     delete root;
129: }
130:
131: void free_ast (astree* tree1, astree* tree2) {
132:     free_ast (tree1);
133:     free_ast (tree2);
134: }
135:
136: void free_ast (astree* tree1, astree* tree2,
137:                astree* tree3) {
138:     free_ast (tree1);
139:     free_ast (tree2);
140:     free_ast (tree3);
141: }
```

```
 1: #include <vector>
 2: #include <string>
 3: using namespace std;
 4:
 5: #include <assert.h>
 6: #include <ctype.h>
 7: #include <stdio.h>
 8: #include <stdlib.h>
 9: #include <string.h>
10:
11: #include "lyutils.h"
12: #include "auxlib.h"
13: #include "stringset.h"
14:
15: astree* yyparse_astree = NULL;
16: int scan_linenr = 1;
17: int scan_offset = 0;
18: bool scan_echo = false;
19: vector<string> included_filenames;
20:
21: const string* scanner_filename (int filenr) {
22:    return &included_filenames.at(filenr);
23: }
24:
25: void scanner_newfilename (const char* filename) {
26:    included_filenames.push_back (filename);
27: }
28:
29: void scanner_newline (void) {
30:    ++scan_linenr;
31:    scan_offset = 0;
32: }
33:
34: void scanner_setecho (bool echoflag) {
35:    scan_echo = echoflag;
36: }
37:
38: void scanner_useraction (void) {
39:    if (scan_echo) {
40:       if (scan_offset == 0) printf (";%5d: ", scan_linenr);
41:       printf ("%s", yytext);
42:    }
43:    scan_offset += yyleng;
44: }
45:
46: void yyerror (const char* message) {
47:    assert (not included_filenames.empty());
48:    errprintf ("%:%s: %d: %s\n",
49:               included_filenames.back().c_str(),
50:               scan_linenr, message);
51: }
52:
53: void scanner_badchar (unsigned char bad) {
54:    char char_rep[16];
55:    sprintf (char_rep, isgraph (bad) ? "%c" : "\\%03o", bad);
56:    errprintf ("%:%s: %d: invalid source character (%s)\n",
57:               included_filenames.back().c_str(),
58:               scan_linenr, char_rep);
```

```
 59: }
 60:
 61: void scanner_badtoken (char* lexeme) {
 62:    errprintf ("%:%s: %d: invalid token (%s)\n",
 63:               included_filenames.back().c_str(),
 64:               scan_linenr, lexeme);
 65: }
 66:
 67: int yylval_token (int symbol) {
 68:    int offset = scan_offset - yyleng;
 69:    yylval = new_astree (symbol, included_filenames.size() - 1,
 70:                         scan_linenr, offset, yytext);
 71:    yyprint(tokfile, symbol, yylval);
 72:    return symbol;
 73: }
 74:
 75: astree* new_parseroot (void) {
 76:    yyparse_astree = new_astree (TOK_ROOT, 0, 0, 0, "");
 77:    intern_stringset("");
 78:    return yyparse_astree;
 79: }
 80:
 81: void error_destructor (astree* tree) {
 82:    if (tree == yyparse_astree) return;
 83:    DEBUGSTMT ('a', dump_astree (stderr, tree); );
 84:    free_ast (tree);
 85: }
 86:
 87: void scanner_include (void) {
 88:    scanner_newline();
 89:    char filename[strlen (yytext) + 1];
 90:    int linenr;
 91:    int scan_rc = sscanf (yytext, "# %d \"%[^\"]\"",
 92:                          &linenr, filename);
 93:    if (scan_rc != 2) {
 94:       errprintf ("%: %d: [%s]: invalid directive, ignored\n",
 95:                  scan_rc, yytext);
 96:    } else {
 97:       fprintf (tokfile, "# %d \"%s\"\n", linenr, filename);
 98:       scanner_newfilename (filename);
 99:       scan_linenr = linenr - 1;
100:       DEBUGF ('m', "filename=%s, scan_linenr=%d\n",
101:               included_filenames.back().c_str(), scan_linenr);
102:    }
103: }
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: #include <assert.h>
 3: #include <errno.h>
 4: #include <libgen.h>
 5: #include <limits.h>
 6: #include <stdarg.h>
 7: #include <stdio.h>
 8: #include <stdlib.h>
 9: #include <string.h>
10: #include <wait.h>
11:
12: #include "auxlib.h"
13:
14: static int exitstatus = EXIT_SUCCESS;
15: static const char* execname = NULL;
16: static const char* localname = NULL;
17: static const char* debugflags = "";
18: static bool alldebugflags = false;
19:
20: void set_execname (char* argv0) {
21:    execname = basename (argv0);
22: }
23:
24: const char* get_execname (void) {
25:    assert (execname != NULL);
26:    return execname;
27: }
28:
29: void set_localname (char* filename) {
30:    localname = basename (filename);
31: }
32:
33: const char* get_localname (void) {
34:    assert (localname != NULL);
35:    return localname;
36: }
37:
38: static void eprint_signal (const char* kind, int signal) {
39:    eprintf (", %s %d", kind, signal);
40:    const char* sigstr = strsignal (signal);
41:    if (sigstr != NULL) fprintf (stderr, " %s", sigstr);
42: }
43:
44: void eprint_status (const char* command, int status) {
45:    if (status == 0) return;
46:    eprintf ("%s: status 0x%04X", command, status);
47:    if (WIFEXITED (status)) {
48:       eprintf (", exit %d", WEXITSTATUS (status));
49:    }
50:    if (WIFSIGNALED (status)) {
51:       eprint_signal ("Terminated", WTERMSIG (status));
52:       #ifdef WCOREDUMP
53:       if (WCOREDUMP (status)) eprintf (", core dumped");
54:       #endif
55:    }
56:    if (WIFSTOPPED (status)) {
57:       eprint_signal ("Stopped", WSTOPSIG (status));
58:    }
```

```
   59:    if (WIFCONTINUED (status)) {
   60:       eprintf (", Continued");
   61:    }
   62:    eprintf ("\n");
   63: }
   64:
```

```
 65:
 66: void veprintf (const char* format, va_list args) {
 67:    assert (execname != NULL);
 68:    assert (format != NULL);
 69:    fflush (NULL);
 70:    if (strstr (format, "%:") == format) {
 71:       fprintf (stderr, "%s: ", get_execname ());
 72:       format += 2;
 73:    }
 74:    vfprintf (stderr, format, args);
 75:    fflush (NULL);
 76: }
 77:
 78: void eprintf (const char* format, ...) {
 79:    va_list args;
 80:    va_start (args, format);
 81:    veprintf (format, args);
 82:    va_end (args);
 83: }
 84:
 85: void errprintf (const char* format, ...) {
 86:    va_list args;
 87:    va_start (args, format);
 88:    veprintf (format, args);
 89:    va_end (args);
 90:    exitstatus = EXIT_FAILURE;
 91: }
 92:
 93: void syserrprintf (const char* object) {
 94:    errprintf ("%:%s: %s\n", object, strerror (errno));
 95: }
 96:
 97: int get_exitstatus (void) {
 98:    return exitstatus;
 99: }
100:
101: void set_exitstatus (int newexitstatus) {
102:    if (exitstatus < newexitstatus) exitstatus = newexitstatus;
103:    DEBUGF ('x', "exitstatus = %d\n", exitstatus);
104: }
105:
106: void __stubprintf (const char* file, int line, const char* func,
107:                    const char* format, ...) {
108:    va_list args;
109:    fflush (NULL);
110:    printf ("%s: %s[%d] %s: ", execname, file, line, func);
111:    va_start (args, format);
112:    vprintf (format, args);
113:    va_end (args);
114:    fflush (NULL);
115: }
116:
```

```
117:
118: void set_debugflags (const char* flags) {
119:    debugflags = flags;
120:    if (strchr (debugflags, '@') != NULL) alldebugflags = true;
121:    DEBUGF ('x', "Debugflags = \"%s\", all = %d\n",
122:            debugflags, alldebugflags);
123: }
124:
125: bool is_debugflag (char flag) {
126:    return alldebugflags or strchr (debugflags, flag) != NULL;
127: }
128:
129: void __debugprintf (char flag, const char* file, int line,
130:                     const char* func, const char* format, ...) {
131:    va_list args;
132:    if (not is_debugflag (flag)) return;
133:    fflush (NULL);
134:    va_start (args, format);
135:    fprintf (stderr, "DEBUGF(%c): %s[%d] %s():\n",
136:             flag, file, line, func);
137:    vfprintf (stderr, format, args);
138:    va_end (args);
139:    fflush (NULL);
140: }
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3:
 4: #include "stringset.h"
 5:
 6: using stringset = unordered_set<string>;
 7:
 8: stringset set;
 9:
10: const string* intern_stringset (const char* string) {
11:    pair<stringset::const_iterator,bool> handle = set.insert (string);
12:    return &*handle.first;
13: }
14:
15: void dump_stringset (FILE* out) {
16:    size_t max_bucket_size = 0;
17:    for (size_t bucket = 0; bucket < set.bucket_count(); ++bucket) {
18:       bool need_index = true;
19:       size_t curr_size = set.bucket_size (bucket);
20:       if (max_bucket_size < curr_size) max_bucket_size = curr_size;
21:       for (stringset::const_local_iterator itor = set.cbegin (bucket);
22:            itor != set.cend (bucket); ++itor) {
23:          if (need_index) fprintf (out, "stringset[%4lu]: ", bucket);
24:                     else fprintf (out, "          %4s   ", "");
25:          need_index = false;
26:          const string* str = &*itor;
27:          fprintf (out, "%22lu %p->\"%s\"\n",
28:                   set.hash_function()(*str),
29:                   str, str->c_str());
30:       }
31:    }
32:    fprintf (out, "load_factor = %.3f\n", set.load_factor());
33:    fprintf (out, "bucket_count = %lu\n", set.bucket_count());
34:    fprintf (out, "max_bucket_size = %lu\n", max_bucket_size);
35: }
36:
```

```
#include "symbol-table.h"
#include "astree.h"
#include "lyutils.h"

vector<symbol_table*> symbol_stack;
symbol_table* struct_table;
int next_block = 1;
vector<int> blocknr_stack;

symbol* new_symbol(astree* node) {
    symbol* sym = new symbol();
    sym->attributes = node->attributes;
    sym->filenr     = node->filenr;
    sym->linenr     = node->linenr;
    sym->offset     = node->offset;
    sym->blocknr    = node->blocknr;
    sym->parameters = nullptr;
    sym->fields     = nullptr;
    return sym;
}

/* Inserts entry into specified symbol table */
bool add_symbol (symbol_table *table, symbol_entry entry) {
    if (table == nullptr) {
        table = new symbol_table();
    }
    return table->insert(entry).second;
}

/* Enter new nested block */
void enter_block(astree* block_node) {
    block_node->blocknr = blocknr_stack.back();
    symbol_stack.push_back(nullptr);
    blocknr_stack.pushback(next_block);
    next_block++;
}

/* Exit current nested block */
void exit_block() {
    /* Print out stuff */
    symbol_stack.pop_back();
    blocknr_stack.pop_back();
}

/*  Traverse astree and build symbol tables */
void traverse_astree(astree* root) {
    switch (root->symbol) {
        case TOK_STRUCT:
            create_struct_sym(root);
            break;
        case TOK_BLOCK:
            enter_block(root);
            break;
        case TOK_VARDECL:
            create_var_sym(root);
            break;
        case TOK_PROTOTYPE:
            break;
```

```
 59:         case TOK_FUNCTION:
 60:             break;
 61:     }
 62:     for (size_t child = 0; child < root->children.size(); ++child) {
 63:         traverse_astree(root->children[child]);
 64:     }
 65:     visit(root);
 66: }
 67:
 68: void set_type_attr(astree* node) {
 69:     switch (node->children.at(0)->symbol) {
 70:         case TOK_BOOL:
 71:             node->attributes.set(ATTR_bool);
 72:             break;
 73:         case TOK_CHAR:
 74:             node->attributes.set(ATTR_char);
 75:             break;
 76:         case TOK_INT:
 77:             node->attributes.set(ATTR_int);
 78:             break;
 79:         case TOK_STRING:
 80:             node->attributes.set(ATTR_string);
 81:             break;
 82:         case TOK_TYPEID:
 83:             node->attributes.set(ATTR_typeid);
 84:             break;
 85:     }
 86: }
 87:
 88: symbol* create_struct_sym(astree* struct_node) {
 89:     symbol *sym;
 90:     symbol_entry entry;
 91:     struct_node->attributes.set(ATTR_struct);
 92:     struct_node->blocknr = 0;
 93:     sym = new_symbol(struct_node);
 94:     for (size_t child = 1;
 95:                 child < struct_node->children.size();
 96:                 child++) {
 97:         astree* field = struct_node->children.at(child);
 98:         add_symbol(sym->fields, create_field_entry(field));
 99:     }
100:     entry = make_pair(struct_node->children.at(0)->lexinfo, sym);
101:     add_symbol(struct_table, entry);
102:     return sym;
103: }
104:
105: symbol_entry create_field_entry(astree* field_node) {
106:     symbol *sym;
107:     symbol_entry entry;
108:     field_node->attributes.set(ATTR_field);
109:     field_node->blocknr = 0;
110:     set_type_attr(field_node);
111:     if (field_node->children.at(1)->symbol == TOK_ARRAY) {
112:         field_node->attributes.set(ATTR_array);
113:         sym   = new_symbol(field_node);
114:         entry = make_pair(field_node->children.at(2)->lexinfo, sym);
115:     } else {
116:         sym   = new_symbol(field_node);
```

```
117:          entry = make_pair(field_node->children.at(1)->lexinfo, sym);
118:      }
119:      return entry;
120: }
121:
122:
123: symbol* create_var_sym(astree* vardecl_node) {
124:      symbol* sym;
125:      symbol_entry entry;
126:      vardecl_node->blocknr = blocknr_stack.back();
127:      astree* var_node = vardecl_node->children.at(0);
128:      var_node->attributes.set(ATTR_variable);
129:      var_node->attributes.set(ATTR_lval);
130:      set_type_attr(var_node);
131:      sym = new_symbol(var_node);
132: }
133: /* Visit a node. Basically a large switch statement that assigns
134:  * attributes to the astree nodes, and builds symbol tables as
135:  * declarations are encountered. */
136: void visit(astree *root) {
137:      return;
138: }
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3: // NAME
 4: //    oc - main program for the oc compiler
 5: //
 6: // SYNOPSIS
 7: //    oc [-ly] [-@ <flags...>] [-D <string>] <program>.oc
 8: //
 9: // DESCRIPTION
10: //    Part four of the ongoing compiler project. Generates
11: //    <prog>.ast, <prog>.tok, <prog>.str, and <prog>.sym.
12: //    The first file is a visualization of the AST.
13: //    The second file shows the contents of each token.
14: //    The third file shows the contents of the maintained
15: //    dictionary of encountered tokens.
16: //    The fourth file shows the symbol table.
17: //
18:
19: #include <string>
20: #include <vector>
21: #include <fstream>
22: using namespace std;
23:
24: #include <assert.h>
25: #include <errno.h>
26: #include <stdio.h>
27: #include <stdlib.h>
28: #include <string.h>
29: #include <unistd.h>
30: #include <sys/types.h>
31: #include <sys/stat.h>
32:
33: #include "astree.h"
34: #include "auxlib.h"
35: #include "lyutils.h"
36: #include "stringset.h"
37:
38: string cpp_command = "/usr/bin/cpp";
39: string filename = "";
40:
41: FILE *astfile;
42: FILE *tokfile;
43: FILE *strfile;
44:
45: // Open a pipe from the C preprocessor.
46: // Exit failure if can't.
47: // Assigns opened pipe to FILE* yyin.
48: void yyin_cpp_popen () {
49:     yyin = popen (cpp_command.c_str(), "r");
50:     if (yyin == NULL) {
51:         syserrprintf (cpp_command.c_str());
52:         exit (get_exitstatus());
53:     }
54: }
55:
56: void yyin_cpp_pclose (void) {
57:     int pclose_rc = pclose (yyin);
58:     eprint_status (cpp_command.c_str(), pclose_rc);
```

```
   59:     if (pclose_rc != 0) set_exitstatus (EXIT_FAILURE);
   60: }
   61:
   62: bool want_echo () {
   63:     return not (isatty (fileno (stdin)) and isatty (fileno (stdout)));
   64: }
   65:
   66: void scan_opts (int argc, char** argv) {
   67:     int option;
   68:     opterr = 0;
   69:     yy_flex_debug = 0;
   70:     yydebug = 0;
   71:     for(;;) {
   72:         option = getopt (argc, argv, "@:D:ly");
   73:         if (option == EOF) break;
   74:         switch (option) {
   75:             case '@': set_debugflags (optarg);   break;
   76:             case 'D': cpp_command += " -D " + *optarg;
   77:                       break;
   78:             case 'l': yy_flex_debug = 1;         break;
   79:             case 'y': yydebug = 1;               break;
   80:             default:  errprintf ("%:bad option (%c)\n", optopt); break;
   81:         }
   82:     }
   83:     if (optind > argc) {
   84:         errprintf ("Usage: %s [-@Dly] [filename]\n", get_execname());
   85:         exit (get_exitstatus());
   86:     }
   87:     const char *fname = optind == argc ? "-" : argv[optind];
   88:
   89:     // Ensure that file ends in .oc
   90:     struct stat buffer;
   91:     if (stat(fname, &buffer) != 0) {
   92:         syserrprintf(fname);
   93:         exit(get_exitstatus());
   94:     } else {
   95:         int length = strlen(fname);
   96:         if (fname[length - 3] != '.' || \
   97:                 fname[length - 2] != 'o' || \
   98:                 fname[length - 1] != 'c') {
   99:             fprintf(stderr, "oc: %s: file must have .oc suffix\n", fname
);
  100:             exit(get_exitstatus());
  101:         }
  102:     }
  103:     // Copy over filename, remove suffix
  104:     filename = string(basename(fname));
  105:     filename = filename.substr(0, filename.length() - 3);
  106:
  107:     // Open cpp pipe
  108:     cpp_command += " ";
  109:     cpp_command += fname;
  110:     yyin_cpp_popen();
  111:     DEBUGF ('m', "filename = %s, yyin = %p, fileno (yyin) = %d\n",
  112:             fname, yyin, fileno (yyin));
  113: }
  114:
  115: int main (int argc, char** argv) {
```

```
116:     set_execname (argv[0]);
117:     DEBUGSTMT ('m',
118:             for (int argi = 0; argi < argc; ++argi) {
119:             eprintf ("%s%c", argv[argi], argi < argc - 1 ? ' ' : '\n');
120:             }
121:             );
122:     // read in options
123:     scan_opts(argc, argv);
124:
125:
126:     // initialize output files
127:     string strfilename = filename + ".str";
128:     strfile = fopen(strfilename.c_str(), "w");
129:
130:     string tokfilename = filename + ".tok";
131:     tokfile = fopen(tokfilename.c_str(), "w");
132:
133:     string astfilename = filename + ".ast";
134:     astfile = fopen(astfilename.c_str(), "w");
135:
136:     // parse
137:     yyparse();
138:     yyin_cpp_pclose();
139:
140:     // generate .str file
141:     dump_stringset(strfile);
142:
143:     // generate .ast file
144:     dump_astree(astfile, yyparse_astree);
145:     free_ast (yyparse_astree);
146:
147:     // close tokfile and strfile
148:     fclose(astfile);
149:     fclose(tokfile);
150:     fclose(strfile);
151:
152:     yylex_destroy();
153:     return get_exitstatus();
154: }
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3:
 4: #ifndef __ASTREE_H__
 5: #define __ASTREE_H__
 6:
 7: #include <string>
 8: #include <vector>
 9: using namespace std;
10:
11: #include "auxlib.h"
12: #include "symbol-table.h"
13:
14: struct astree {
15:    int symbol;               // token code
16:    size_t filenr;            // index into filename stack
17:    size_t linenr;            // line number from source code
18:    size_t offset;            // offset of token with current line
19:    size_t blocknr;
20:    attr_bitset attributes;
21:    symbol_table *struct_node;
22:    const string* lexinfo;    // pointer to lexical information
23:    vector<astree*> children; // children of this n-way node
24: };
25:
26:
27: astree* new_astree (int symbol, int filenr, int linenr,
28:                     int offset, const char* lexinfo);
29: astree* adopt1 (astree* root, astree* child);
30: astree* adopt2 (astree* root, astree* left, astree* right);
31: astree* adopt3 (astree* root, astree* one, astree* two, astree* three);
32: astree* adopt1sym (astree* root, astree* child, int symbol);
33: astree* adopt2sym (astree* root,
34:         astree* left, astree* right, int symbol);
35: astree* adopt3sym (astree* root, astree* one, astree* two,
36:                    astree* three, int symbol);
37: void dump_astree (FILE* outfile, astree* root);
38: void yyprint (FILE* outfile, unsigned short toknum,
39:               astree* yyvaluep);
40: void free_ast (astree* tree);
41: void free_ast (astree* tree1, astree* tree2);
42: void free_ast (astree* tree1, astree* tree2,
43:                 astree* tree3);
44:
45: #endif
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3:
 4: #ifndef __LYUTILS_H__
 5: #define __LYUTILS_H__
 6:
 7: // Lex and Yacc interface utility.
 8:
 9: #include <stdio.h>
10:
11: #include "astree.h"
12: #include "auxlib.h"
13:
14: #define YYEOF 0
15:
16: extern FILE* yyin;
17: extern astree* yyparse_astree;
18: extern int yyin_linenr;
19: extern char* yytext;
20: extern int yy_flex_debug;
21: extern int yydebug;
22: extern int yyleng;
23: extern FILE *tokfile;
24:
25: int yylex (void);
26: int yyparse (void);
27: void yyerror (const char* message);
28: int yylex_destroy (void);
29: const char* get_yytname (int symbol);
30: bool is_defined_token (int symbol);
31: void error_destructor (astree* tree);
32:
33: const string* scanner_filename (int filenr);
34: void scanner_newfilename (const char* filename);
35: void scanner_badchar (unsigned char bad);
36: void scanner_badtoken (char* lexeme);
37: void scanner_newline (void);
38: void scanner_setecho (bool echoflag);
39: void scanner_useraction (void);
40:
41: astree* new_parseroot (void);
42: int yylval_token (int symbol);
43:
44: void scanner_include (void);
45:
46: typedef astree* astree_pointer;
47: #define YYSTYPE astree_pointer
48: #include "yyparse.h"
49:
50: #endif
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3: // DESCRIPTION
 4: //    Auxiliary library containing miscellaneous useful things.
 5: //
 6:
 7: #ifndef __AUXLIB_H__
 8: #define __AUXLIB_H__
 9:
10: #include <stdarg.h>
11:
12: //
13: // Error message and exit status utility.
14: //
15:
16: void set_execname (char* argv0);
17:    //
18:    // Sets the program name for use by auxlib messages.
19:    // Must called from main before anything else is done,
20:    // passing in argv[0].
21:    //
22:
23: const char* get_execname (void);
24:    //
25:    // Returns a read-only value previously stored by set_progname.
26:    //
27:
28: void set_localname (char* filename);
29:    //
30:    // Similar to set_execname, used for general purpose basename
31:    // retrieval
32:    //
33:
34: const char* get_localname (void);
35:    //
36:    // Again similar to get_execname
37:    //
38:
39: void eprint_status (const char* command, int status);
40:    //
41:    // Print the status returned by wait(2) from a subprocess.
42:    //
43:
44: int get_exitstatus (void);
45:    //
46:    // Returns the exit status.  Default is EXIT_SUCCESS unless
47:    // set_exitstatus (int) is called.  The last statement in main
48:    // should be:  ``return get_exitstatus();''.
49:    //
50:
51: void set_exitstatus (int);
52:    //
53:    // Sets the exit status.  Remebers only the largest value passed in.
54:    //
55:
```

```
56:
57: void veprintf (const char* format, va_list args);
58:    //
59:    // Prints a message to stderr using the vector form of
60:    // argument list.
61:    //
62:
63: void eprintf (const char* format, ...);
64:    //
65:    // Print a message to stderr according to the printf format
66:    // specified.  Usually called for debug output.
67:    // Precedes the message by the program name if the format
68:    // begins with the characters `%:'.
69:    //
70:
71: void errprintf (const char* format, ...);
72:    //
73:    // Print an error message according to the printf format
74:    // specified, using eprintf.  Sets the exitstatus to EXIT_FAILURE.
75:    //
76:
77: void syserrprintf (const char* object);
78:    //
79:    // Print a message resulting from a bad system call.  The
80:    // object is the name of the object causing the problem and
81:    // the reason is taken from the external variable errno.
82:    // Sets the exit status to EXIT_FAILURE.
83:    //
84:
```

```
 85:
 86: //
 87: // Support for stub messages.
 88: //
 89: #define STUBPRINTF(...) \
 90:         __stubprintf (__FILE__, __LINE__, __func__, __VA_ARGS__)
 91: void __stubprintf (const char* file, int line, const char* func,
 92:                    const char* format, ...);
 93:
 94: //
 95: // Debugging utility.
 96: //
 97:
 98: void set_debugflags (const char* flags);
 99:    //
100:    // Sets a string of debug flags to be used by DEBUGF statements.
101:    // Uses the address of the string, and does not copy it, so it
102:    // must not be dangling.  If a particular debug flag has been set,
103:    // messages are printed.  The format is identical to printf format.
104:    // The flag "@" turns on all flags.
105:    //
106:
107: bool is_debugflag (char flag);
108:    //
109:    // Checks to see if a debugflag is set.
110:    //
111:
112: #ifdef NDEBUG
113: // Do not generate any code.
114: #define DEBUGF(FLAG,...)   /**/
115: #define DEBUGSTMT(FLAG,STMTS) /**/
116: #else
117: // Generate debugging code.
118: void __debugprintf (char flag, const char* file, int line,
119:                     const char* func, const char* format, ...);
120: #define DEBUGF(FLAG,...) \
121:         __debugprintf (FLAG, __FILE__, __LINE__, __func__, __VA_ARGS__)
122: #define DEBUGSTMT(FLAG,STMTS) \
123:         if (is_debugflag (FLAG)) { DEBUGF (FLAG, "\n"); STMTS }
124: #endif
125: #endif
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3:
 4: #ifndef __STRINGSET__
 5: #define __STRINGSET__
 6:
 7: #include <string>
 8: #include <unordered_set>
 9: using namespace std;
10:
11: #include <stdio.h>
12:
13: const string* intern_stringset (const char*);
14:
15: void dump_stringset (FILE*);
16:
17: #endif
18:
```

```
 1: // Author, Andrew Edwards, ancedwar@ucsc.edu
 2: //
 3:
 4: #ifndef __SYMBOLTABLE__
 5: #define __SYMBOLTABLE__
 6:
 7: #include <string>
 8: #include <unordered_map>
 9: #include <bitset>
10: #include <vector>
11: using namespace std;
12:
13: enum { ATTR_void, ATTR_bool, ATTR_char, ATTR_int, ATTR_null,
14:        ATTR_string, ATTR_struct, ATTR_array, ATTR_function,
15:        ATTR_variable, ATTR_field, ATTR_typeid, ATTR_param,
16:        ATTR_lval, ATTR_const, ATTR_vreg, ATTR_vaddr,
17:        ATTR_bitset_size,
18: };
19: using attr_bitset = bitset<ATTR_bitset_size>;
20:
21: struct symbol;
22: using symbol_table = unordered_map<const string*,symbol*>;
23: using symbol_entry = pair<const string*,symbol*>;
24:
25: struct symbol {
26:    attr_bitset attributes;
27:    symbol_table *fields;
28:    size_t filenr, linenr, offset;
29:    size_t blocknr;
30:    vector<symbol*>* parameters;
31: };
32:
33: struct astree;
34: symbol* new_symbol(astree* node);
35: bool add_symbol(symbol_table table, symbol_entry entry);
36: void enter_block();
37: void exit_block();
38: void traverse_astree(astree* root);
39: symbol* create_struct_sym(astree* struct_node);
40: symbol_entry create_field_entry(astree* field_node);
41: void visit(astree* root);
42: #endif
```