```
// $Id: astree.h,v 1.6 2016-09-21 17:13:03-07 - - $

#ifndef __ASTREE_H__
#define __ASTREE_H__

#include <string>
#include <vector>
using namespace std;

#include "syslib.h"

struct location {
   size_t filenr;
   size_t linenr;
   size_t offset;
};

struct astree {

   // Fields.
   int symbol;               // token code
   location lloc;            // source location
   const string* lexinfo;    // pointer to lexical information
   vector<astree*> children; // children of this n-way node

   // Functions.
   astree (int symbol, const location&, const char* lexinfo);
   ~astree();
   astree* adopt (astree* child1, astree* child2 = nullptr);
   astree* adopt_sym (astree* child, int symbol);
   void dump_node (FILE*);
   void dump_tree (FILE*, int depth = 0);
   static void dump (FILE* outfile, astree* tree);
   static void print (FILE* outfile, astree* tree, int depth = 0)
;
};

void destroy (astree* tree1, astree* tree2 = nullptr);

void errllocprintf (const location&, const char* format, const ch
ar*);

#endif

```

```
// $Id: astree.cpp,v 1.8 2016-09-21 17:13:03-07 - - $

#include <assert.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "astree.h"
#include "string_set.h"
#include "lyutils.h"

astree::astree (int symbol_, const location& lloc_, const char* info) {
   symbol = symbol_;
   lloc = lloc_;
   lexinfo = string_set::intern (info);
   // vector defaults to empty -- no children
}

astree::~astree() {
   while (not children.empty()) {
      astree* child = children.back();
      children.pop_back();
      delete child;
   }
   if (yydebug) {
      fprintf (stderr, "Deleting astree (");
      astree::dump (stderr, this);
      fprintf (stderr, ")\n");
   }
}

astree* astree::adopt (astree* child1, astree* child2) {
   if (child1 != nullptr) children.push_back (child1);
   if (child2 != nullptr) children.push_back (child2);
   return this;
}

astree* astree::adopt_sym (astree* child, int symbol_) {
   symbol = symbol_;
   return adopt (child);
}

```

```cpp

void astree::dump_node (FILE* outfile) {
   fprintf (outfile, "%p->{%s %zd.%zd.%zd \"%s\":",
            this, parser::get_tname (symbol),
            lloc.filenr, lloc.linenr, lloc.offset,
            lexinfo->c_str());
   for (size_t child = 0; child < children.size(); ++child) {
      fprintf (outfile, " %p", children.at(child));
   }
}

void astree::dump_tree (FILE* outfile, int depth) {
   fprintf (outfile, "%*s", depth * 3, "");
   dump_node (outfile);
   fprintf (outfile, "\n");
   for (astree* child: children) child->dump_tree (outfile, depth + 1);
   fflush (NULL);
}

void astree::dump (FILE* outfile, astree* tree) {
   if (tree == nullptr) fprintf (outfile, "nullptr");
                   else tree->dump_node (outfile);
}

void astree::print (FILE* outfile, astree* tree, int depth) {
   fprintf (outfile, "; %*s", depth * 3, "");
   fprintf (outfile, "%s \"%s\" (%zd.%zd.%zd)\n",
            parser::get_tname (tree->symbol), tree->lexinfo->c_str(),
            tree->lloc.filenr, tree->lloc.linenr, tree->lloc.offset);
   for (astree* child: tree->children) {
      astree::print (outfile, child, depth + 1);
   }
}

void destroy (astree* tree1, astree* tree2) {
   if (tree1 != nullptr) delete tree1;
   if (tree2 != nullptr) delete tree2;
}

void errllocprintf (const location& lloc, const char* format,
                    const char* arg) {
   static char buffer[0x1000];
   assert (sizeof buffer > strlen (format) + strlen (arg));
   snprintf (buffer, sizeof buffer, format, arg);
   errprintf ("%s:%zd.%zd: %s",
              lexer::filename (lloc.filenr), lloc.linenr, lloc.offset,
              buffer);
}
```

```
#ifndef __AUXLIB_H__
#define __AUXLIB_H__

#include <stdarg.h>

//
// DESCRIPTION
//    Auxiliary library containing miscellaneous useful things.
//

//
// Error message and exit status utility.
//

void set_execname (char* argv0);
// Sets the program name for use by auxlib messages.
// Must called from main before anything else is done,
// passing in argv[0].

const char* get_execname (void);
// Returns a read-only value previously set by set_progname.

void eprint_status (const char* command, int status);
// Print the status returned by wait(2) from a subprocess.

int get_exitstatus (void);
// Returns the exit status.  Default is EXIT_SUCCESS unless
// set_exitstatus (int) is called.  The last statement in main
// should be:  ``return get_exitstatus();''.

void set_exitstatus (int);
// Sets the exit status.  Remebers only the largest value.

```

```

void veprintf (const char* format, va_list args);
// Prints a message to stderr using the vector form of
// argument list.

void eprintf (const char* format, ...);
// Print a message to stderr according to the printf format
// specified.  Usually called for debug output.
// Precedes the message by the program name if the format
// begins with the characters `%:'.

void errprintf (const char* format, ...);
// Print an error message according to the printf format
// specified, using eprintf.
// Sets the exitstatus to EXIT_FAILURE.

void syserrprintf (const char* object);
// Print a message resulting from a bad system call.  The
// object is the name of the object causing the problem and
// the reason is taken from the external variable errno.
// Sets the exit status to EXIT_FAILURE.

```

```

//
// Support for stub messages.
//
#define STUBPRINTF(...) \
        __stubprintf (__FILE__, __LINE__, __func__, __VA_ARGS__)
void __stubprintf (const char* file, int line, const char* func,
                   const char* format, ...);

//
// Debugging utility.
//

void set_debugflags (const char* flags);
// Sets a string of debug flags to be used by DEBUGF statements.
// Uses the address of the string, and does not copy it, so
// it must not be dangling.  If a particular debug flag has
// been set, messages are printed.  The format is identical to
// printf format.  The flag "@" turns on all flags.

bool is_debugflag (char flag);
// Checks to see if a debugflag is set.

#ifdef NDEBUG
// Do not generate any code.
#define DEBUGF(FLAG,...)   /**/
#define DEBUGSTMT(FLAG,STMTS) /**/
#else
// Generate debugging code.
void __debugprintf (char flag, const char* file, int line,
                    const char* func, const char* format, ...);
#define DEBUGF(FLAG,...) \
        __debugprintf (FLAG, __FILE__, __LINE__, __func__, \
                       __VA_ARGS__)
#define DEBUGSTMT(FLAG,STMTS) \
        if (is_debugflag (FLAG)) { DEBUGF (FLAG, "\n"); STMTS }
#endif

#endif

```

```

#include <assert.h>
#include <errno.h>
#include <libgen.h>
#include <limits.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wait.h>

#include "auxlib.h"

static int exitstatus = EXIT_SUCCESS;
static const char* execname = NULL;
static const char* debugflags = "";
static bool alldebugflags = false;

void set_execname (char* argv0) {
   execname = basename (argv0);
}

const char* get_execname (void) {
   assert (execname != NULL);
   return execname;
}

static void eprint_signal (const char* kind, int signal) {
   eprintf (", %s %d", kind, signal);
   const char* sigstr = strsignal (signal);
   if (sigstr != NULL) fprintf (stderr, " %s", sigstr);
}

void eprint_status (const char* command, int status) {
   if (status == 0) return;
   eprintf ("%s: status 0x%04X", command, status);
   if (WIFEXITED (status)) {
      eprintf (", exit %d", WEXITSTATUS (status));
   }
   if (WIFSIGNALED (status)) {
      eprint_signal ("Terminated", WTERMSIG (status));
      #ifdef WCOREDUMP
      if (WCOREDUMP (status)) eprintf (", core dumped");
      #endif
   }
   if (WIFSTOPPED (status)) {
      eprint_signal ("Stopped", WSTOPSIG (status));
   }
   if (WIFCONTINUED (status)) {
      eprintf (", Continued");
   }
   eprintf ("\n");
}

```

```
int get_exitstatus (void) {
   return exitstatus;
}

void veprintf (const char* format, va_list args) {
   assert (execname != NULL);
   assert (format != NULL);
   fflush (NULL);
   if (strstr (format, "%:") == format) {
      fprintf (stderr, "%s: ", get_execname ());
      format += 2;
   }
   vfprintf (stderr, format, args);
   fflush (NULL);
}

void eprintf (const char* format, ...) {
   va_list args;
   va_start (args, format);
   veprintf (format, args);
   va_end (args);
}

void errprintf (const char* format, ...) {
   va_list args;
   va_start (args, format);
   veprintf (format, args);
   va_end (args);
   exitstatus = EXIT_FAILURE;
}

void syserrprintf (const char* object) {
   errprintf ("%:%s: %s\n", object, strerror (errno));
}

void set_exitstatus (int newexitstatus) {
   if (exitstatus < newexitstatus) exitstatus = newexitstatus;
   DEBUGF ('x', "exitstatus = %d\n", exitstatus);
}

void __stubprintf (const char* file, int line, const char* func,
                   const char* format, ...) {
   va_list args;
   fflush (NULL);
   printf ("%s: %s[%d] %s: ", execname, file, line, func);
   va_start (args, format);
   vprintf (format, args);
   va_end (args);
   fflush (NULL);
}

```

```

void set_debugflags (const char* flags) {
   debugflags = flags;
   if (strchr (debugflags, '@') != NULL) alldebugflags = true;
   DEBUGF ('x', "Debugflags = \"%s\", all = %d\n",
           debugflags, alldebugflags);
}

bool is_debugflag (char flag) {
   return alldebugflags or strchr (debugflags, flag) != NULL;
}

void __debugprintf (char flag, const char* file, int line,
                    const char* func, const char* format, ...) {
   va_list args;
   if (not is_debugflag (flag)) return;
   fflush (NULL);
   va_start (args, format);
   fprintf (stderr, "DEBUGF(%c): %s[%d] %s():\n",
            flag, file, line, func);
   vfprintf (stderr, format, args);
   va_end (args);
   fflush (NULL);
}

```

```
#ifndef __LYUTILS_H__
#define __LYUTILS_H__

// Lex and Yacc interface utility.

#include <stdio.h>

#include "astree.h"
#include "auxlib.h"

#define YYEOF 0

extern FILE* yyin;
extern astree* yyparse_astree;
extern int yyin_linenr;
extern char* yytext;
extern int yy_flex_debug;
extern int yydebug;
extern int yyleng;

int yylex (void);
int yyparse (void);
void yyerror (const char* message);
int yylex_destroy (void);
const char* get_yytname (int symbol);
bool is_defined_token (int symbol);

const string* lexer_filename (int filenr);
void lexer_newfilename (const char* filename);
void lexer_badchar (unsigned char bad);
void lexer_badtoken (char* lexeme);
void lexer_newline (void);
void lexer_setecho (bool echoflag);
void lexer_useraction (void);

astree* new_parseroot (void);
int yylval_token (int symbol);

void lexer_include (void);

typedef astree* astree_pointer;
#define YYSTYPE astree_pointer
#include "yyparse.h"

#endif
```

```

#include <vector>
#include <string>
using namespace std;

#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lyutils.h"
#include "auxlib.h"

astree* yyparse_astree = NULL;
int scan_linenr = 1;
int scan_offset = 0;
bool scan_echo = false;
vector<string> included_filenames;

const string* lexer_filename (int filenr) {
   return &included_filenames.at(filenr);
}

void lexer_newfilename (const char* filename) {
   included_filenames.push_back (filename);
}

void lexer_newline (void) {
   ++scan_linenr;
   scan_offset = 0;
}

void lexer_setecho (bool echoflag) {
   scan_echo = echoflag;
}

```

```

void lexer_useraction (void) {
   if (scan_echo) {
      if (scan_offset == 0) printf (";%5d: ", scan_linenr);
      printf ("%s", yytext);
   }
   scan_offset += yyleng;
}

void yyerror (const char* message) {
   assert (not included_filenames.empty());
   errprintf ("%:%s: %d: %s\n",
              included_filenames.back().c_str(),
              scan_linenr, message);
}

void lexer_badchar (unsigned char bad) {
   char char_rep[16];
   sprintf (char_rep, isgraph (bad) ? "%c" : "\\%03o", bad);
   errprintf ("%:%s: %d: invalid source character (%s)\n",
              included_filenames.back().c_str(),
              scan_linenr, char_rep);
}

void lexer_badtoken (char* lexeme) {
   errprintf ("%:%s: %d: invalid token (%s)\n",
              included_filenames.back().c_str(),
              scan_linenr, lexeme);
}

int yylval_token (int symbol) {
   int offset = scan_offset - yyleng;
   yylval = new astree (symbol, included_filenames.size() - 1,
                        scan_linenr, offset, yytext);
   return symbol;
}

astree* new_parseroot (void) {
   yyparse_astree = new astree (TOK_ROOT, 0, 0, 0, "");
   return yyparse_astree;
}

```

```

void lexer_include (void) {
   lexer_newline();
   char filename[strlen (yytext) + 1];
   int linenr;
   int scan_rc = sscanf (yytext, "# %d \"%[^\"]\"",
                         &linenr, filename);
   if (scan_rc != 2) {
      errprintf ("%: %d: [%s]: invalid directive, ignored\n",
                 scan_rc, yytext);
   }else {
      printf (";# %d \"%s\"\n", linenr, filename);
      lexer_newfilename (filename);
      scan_linenr = linenr - 1;
      DEBUGF ('m', "filename=%s, scan_linenr=%d\n",
              included_filenames.back().c_str(), scan_linenr);
   }
}

```

```
/* A Bison parser, made by GNU Bison 2.7.  */

/* Bison interface for Yacc-like parsers in C

      Copyright (C) 1984, 1989-1990, 2000-2012 Free Software Foundation, Inc.

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.  */

/* As a special exception, you may create a larger work that contains
   part or all of the Bison parser skeleton and distribute that work
   under terms of your choice, so long as that work isn't itself a
   parser generator using the skeleton or a modified version thereof
   as a parser skeleton.  Alternatively, if you modify or redistribute
   the parser skeleton itself, you may (at your option) remove this
   special exception, which will cause the skeleton and the resulting
   Bison output files to be licensed under the GNU General Public
   License without this special exception.

   This special exception was added by the Free Software Foundation in
   version 2.2 of Bison.  */

#ifndef YY_YY_YYPARSE_H_INCLUDED
# define YY_YY_YYPARSE_H_INCLUDED
/* Enabling traces.  */
#ifndef YYDEBUG
# define YYDEBUG 1
#endif
#if YYDEBUG
```

```
extern int yydebug;
#endif

/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers
      know about them.  */
   enum yytokentype {
     TOK_VOID = 258,
     TOK_CHAR = 259,
     TOK_INT = 260,
     TOK_STRING = 261,
     TOK_IF = 262,
     TOK_ELSE = 263,
     TOK_WHILE = 264,
     TOK_RETURN = 265,
     TOK_STRUCT = 266,
     TOK_NULL = 267,
     TOK_NEW = 268,
     TOK_ARRAY = 269,
     TOK_EQ = 270,
     TOK_NE = 271,
     TOK_LT = 272,
     TOK_LE = 273,
     TOK_GT = 274,
     TOK_GE = 275,
     TOK_IDENT = 276,
     TOK_INTCON = 277,
     TOK_CHARCON = 278,
     TOK_STRINGCON = 279,
     TOK_BLOCK = 280,
     TOK_CALL = 281,
     TOK_IFELSE = 282,
     TOK_INITDECL = 283,
     TOK_POS = 284,
     TOK_NEG = 285,
     TOK_NEWARRAY = 286,
     TOK_TYPEID = 287,
     TOK_FIELD = 288,
     TOK_ORD = 289,
     TOK_CHR = 290,
     TOK_ROOT = 291
   };
#endif


#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef int YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif
```

```

extern YYSTYPE yylval;

#ifdef YYPARSE_PARAM
#if defined __STDC__ || defined __cplusplus
int yyparse (void *YYPARSE_PARAM);
#else
int yyparse ();
#endif
#else /* ! YYPARSE_PARAM */
#if defined __STDC__ || defined __cplusplus
int yyparse (void);
#else
int yyparse ();
#endif
#endif /* ! YYPARSE_PARAM */

#endif /* !YY_YY_YYPARSE_H_INCLUDED  */
```

```
%{
// Dummy parser for scanner project.

#include <cassert>

#include "lyutils.h"
#include "astree.h"

%}

%debug
%defines
%error-verbose
%token-table
%verbose

%token TOK_VOID TOK_CHAR TOK_INT TOK_STRING
%token TOK_IF TOK_ELSE TOK_WHILE TOK_RETURN TOK_STRUCT
%token TOK_NULL TOK_NEW TOK_ARRAY
%token TOK_EQ TOK_NE TOK_LT TOK_LE TOK_GT TOK_GE
%token TOK_IDENT TOK_INTCON TOK_CHARCON TOK_STRINGCON

%token TOK_BLOCK TOK_CALL TOK_IFELSE TOK_INITDECL
%token TOK_POS TOK_NEG TOK_NEWARRAY TOK_TYPEID TOK_FIELD
%token TOK_ORD TOK_CHR TOK_ROOT

%start program

%%

program : program token | ;
token   : '(' | ')' | '[' | ']' | '{' | '}' | ';' | ',' | '.'
        | '=' | '+' | '-' | '*' | '/' | '%' | '!'
        | TOK_VOID | TOK_CHAR | TOK_INT | TOK_STRING
        | TOK_IF | TOK_ELSE | TOK_WHILE | TOK_RETURN | TOK_STRUCT
        | TOK_NULL | TOK_NEW | TOK_ARRAY
        | TOK_EQ | TOK_NE | TOK_LT | TOK_LE | TOK_GT | TOK_GE
        | TOK_IDENT | TOK_INTCON | TOK_CHARCON | TOK_STRINGCON
        | TOK_ORD | TOK_CHR | TOK_ROOT
        ;

%%
```

```


const char *get_yytname (int symbol) {
   return yytname [YYTRANSLATE (symbol)];
}


bool is_defined_token (int symbol) {
   return YYTRANSLATE (symbol) > YYUNDEFTOK;
}

/*
static void* yycalloc (size_t size) {
   void* result = calloc (1, size);
   assert (result != nullptr);
   return result;
}
*/

```