Linking

- Linking and Libraries
- Dynamic Linking Tradeoffs
- Dynamic (runtime) Linking Interface
- Library Naming Conventions
- Library Versions
- Using Libraries
- Dynamic Loaded Expression Evaluator
- `dlexpr.h` Header File Selections
- dlexpr.c Auxiliary Routines
- dlexpr.c Auxiliary Routines Continued
- `dlexpr.c main()`
- `dlexpr.c main()` Continued
- Arithmetic Grammar
- Arithmetic Evaluation Routines
- Arithmetic Evaluation Routines Continued
- Arithmetic Evaluation Routines Continued
- References

Linking and Libraries

- A relocatable object module (typically with extension `.o`) is produced from a single `.c` file (and multiple header files) using the `-c` compiler option.
- Object file modules can be in different formats: `a.out` is a historical format; *Executable Linking Format* ELF is a more modern format (used on both Solaris and Linux).
- A library is a collection of object modules along with some kind of symbol table documenting the symbols imported and exported by each module.
- When a library is linked into an executable, only those object modules necessary to satisfy outstanding symbols are added to the executable; hence the order of searching of libraries is very important.

  For example, to use a non-standard `malloc` library, it should be searched before the standard `libc` library.
- Static linking searches libraries at compile-time and builds a self-contained executable. Built by a compile-time *link editor* (`ld(1)` on Solaris).
- Dynamic-linking searches libraries at load-time and even runtime. Uses *runtime linker* (`ld.so.1(1)` on Solaris).
- A dynamically linked executable can link to a routine on a *need-to* basis at runtime. This is also referred to as *lazy linking*.

Dynamic Linking Tradeoffs

- Shared libraries are *shared*: different programs share the same code, thus using less memory.
- Executables depending on shared libraries do not contain the shared library code and thus are smaller.
- Shared library bugs can be fixed by providing a new version of the library, without needing to recompile/relink the programs dependent on it.
- Shared libraries increase complexity. A executable is not standalone but requires a specific environment.
- Allows interposition of *wrapper functions* for library functions; useful for tracing and profiling.

Dynamic (runtime) Linking Interface

```
void *dlopen (const char *fileName, int flag);
void *dlsym(void *dlHandle, char *symName);
const char *dlerror(void);
int dlclose (void *dlHandle);
```

- `dlopen()` adds dynamic library in `fileName` to executable, and returns a opaque handle. `flag` may be `RTLD_LAZY` or `RTLD_NOW`. If `RTLD_GLOBAL` is or'ed with `flag`, then symbols in `fileName` are made available to subsequently loaded modules.
- `dlsym()` returns the address of `symName` in dynamic library referenced by `dlHandle`. `NULL` if not found.
- `dlerror()` returns a human-readable error message for last error.
- `dlclose()` closes a dynamic library and unloads it if its reference count has gone to 0.

Library Naming Conventions

- Library names start with prefix `lib` as in `libc`. The prefix is omitted when the library is specified on the compiler command-line: `-lc` specifies linking with `libc`.
- Extension `.a` is used for static libraries (historically produced using the `ar` program).
- Extension `.so` is used for dynamic shared libraries.
- Above conventions need not be followed by runtime libraries loaded by `dlopen()`.

Library Versions

The following conventions are used to keep track of library versions:

- Each shared library has a *soname* of the form *name*`.so.`*V* where *V* is the version number.
- When the library API is no longer backward compatible, the version number should change.
- Each library has a actual name of the form *name*`.so.`*V*`.`*M*`.`*R*, where *M* is a minor version number (no interface change) and *R* is a release number (may be a bug fix).
- When a library is created, the *soname* is specified on the linker command-line (along with the actual name).
- Once again, when linking against a library, it is the *soname* which is specified.
- The *soname*'s are symbolic links to the latest actual library name, thus enabling a seamless upgrade when the API is backward compatible.

Using Libraries

- The environmental variable `LD_LIBRARY_PATH` is a `:` separated list of directories searched by the linker for dynamic libraries.
- `ldd` lists the libraries a executable depends on:

```
$ ldd /bin/ls
libc.so.1 =>     /usr/lib/libc.so.1
libdl.so.1 =>    /usr/lib/libdl.so.1
/usr/platform/SUNW,Ultra-Enterprise/lib/libc_psr.so.1
$
```

  Can be a security problem with untrusted executables.

Dynamic Loaded Expression Evaluator

- Program accepts a single argument, which is the name of dynamically loaded module which evaluates expressions of a certain kind.

```
$ LD_LIBRARY_PATH=. ./dlexpr arith
>> 1 + 3 * ( 2 + -4/3)
4
>> 1 + 3 * ( 2 + -4/3) *
ERR                     ^
>> $
$ LD_LIBRARY_PATH=. ./dlexpr bool
>> 0 | (1 & ~0)
1
>>  0 | (1 & ~0 | )
ERR               ^
>> $
```

- Expression evaluators are written using recursive-descent techniques. `setjmp()`/`longjmp()` invaluable for error recovery and reporting.

`dlexpr.h` Header File Selections

```
typedef struct DLExprEnv {
  char ch;              /* lookahead */
  jmp_buf *jmpBufP;     /* error recovery */
  char line[LINE_MAX];  /* input line */
  unsigned lineIndex;   /* next char index */
  void(*nextCh)         /* function to get  */
    (struct DLExprEnv *);/* next char */
} DLExprEnv;

/* A parse function takes a pointer to a DLExprEnv
 * and returns the value of the expression.
 */
typedef int (*ParseFn)(DLExprEnv *envP);

/* called if an error is encountered */
void error(DLExprEnv *envP);

/* Checks if lookahead matches specified char c.
 * If matches, then lookahead is advanced, else error.
 */
#define MATCH(c, envP)          \
do {                            \
  if ((envP)->ch == (c)) {      \
    (envP)->nextCh(envP);       \
  }                             \
  else {                        \
    error(envP);                \
  }                             \
} while (0)
```

dlexpr.c Auxiliary Routines

```
/* Prints a ^ char at error position.
 * aborts via longjmp().
 */
void
error(DLExprEnv *envP)
{
  fprintf(stderr, "ERR%*s^\n",
          envP->lineIndex - (envP->lineIndex > 0),
          "");
  longjmp(envP->jmpBufP, 1);
}

/* Update ch field of envP to next non-blank,
 * non-tab char from  envP->line[].
 */
static void
nextChar(DLExprEnv *envP)
{
  do {
    envP->ch = envP->line[envP->lineIndex++];
  } while (envP->ch == ' ' || envP->ch == '\t');
}
```

dlexpr.c Auxiliary Routines Continued

```
/* Read next line for stdin into envP->line[].
 * Returns 0 on EOF; non-zero otherwise.
 */
static int
getLine(DLExprEnv *envP)
{
  while (1) {
    if (!fgets(envP->line, LINE_MAX, stdin)) {
      if (feof(stdin)) {
        return 0;
      }
      else {
        fprintf(stderr, "ERR I/O error\n");
      }
    }
    else if (envP->line[strlen(envP->line) - 1]
             != '\n') {
      fprintf(stderr, "ERR Line too long\n");
    }
    else break;
  }
  envP->lineIndex = 0;
  envP->nextCh(envP);
  return 1;
}
```

dlexpr.c main()

```
#define PARSE_FN_NAME_SUFFIX            "Parse"

int
main(int argc, char *argv[])
{
  void *dlHandle;
  char *parseFnName;
  ParseFn parseFn;
  if (argc != 2) {
    fprintf(stderr, "usage: %s EXPR_MODULE\n",
            argv[0]);
    exit(1);
  }
  if (!(dlHandle = dlopen(argv[1], RTLD_NOW))) {
    fprintf(stderr, "%s dlopen(): %s\n",
            argv[0], dlerror());
    exit(1);
  }
  parseFnName =
    malloc(strlen(argv[1]) +
           strlen(PARSE_FN_NAME_SUFFIX) + 1);
  if (!parseFnName) {
    perror(argv[0]); exit(1);
  }
```

`dlexpr.c main()` Continued

```
  strcpy(parseFnName, argv[1]);
  strcpy(parseFnName + strlen(argv[1]),
         PARSE_FN_NAME_SUFFIX);
  if (!(parseFn = dlsym(dlHandle, parseFnName))) {
    fprintf(stderr, "%s dlsym(): %s\n",
            argv[0], dlerror());
    exit(1);
  }
  { /* The module is now loaded */
    jmp_buf jmpBuf;
    static DLExprEnv env;
    env.jmpBufP = &jmpBuf; env.nextCh = nextChar;
    if (setjmp(jmpBuf) >= 0) { //spec requires use within a test
      while (printf(">> ") && getLine(&env)) {
        printf("%d\n", parseFn(&env));
      }
    }
  }

  free(parseFnName);
  dlclose(dlHandle);
  return 0;
}
```

Arithmetic Grammar

```
arithParse
  : exp '\n'
  ;

exp
  : term { '+' term }
  | term { '-' term }
  | term
  ;

term
  : factor { '*' factor }
  | factor { '/' factor }
  | factor
  ;
factor
  : '(' exp ')'
  | '-' factor
  | DIGIT
  ;
```

Arithmetic Evaluation Routines

```
int
arithParse(DLExprEnv *envP)
{
  int v = expr(envP);
  MATCH('\n', envP);
  return v;
}

static int
expr(DLExprEnv *envP)
{
  int v = term(envP);
  while(envP->ch == '+' || envP->ch == '-') {
    char c = envP->ch;
    int v1;
    MATCH(c, envP);
    v1 = term(envP);
    v += (c == '+') ? v1 : -v1;
  }
  return v;
}
```

Arithmetic Evaluation Routines Continued

```
static int
term(DLExprEnv *envP)
{
  int v = factor(envP);
  while(envP->ch == '*' || envP->ch == '/') {
    char c = envP->ch;
    int v1;
    MATCH(c, envP);
    v1 = factor(envP);
    if (c == '*') {
      v *= v1;
    }
    else {
      v /= v1;
    }
  }
  return v;
}
```

Arithmetic Evaluation Routines Continued

```
static int
factor(DLExprEnv *envP)
{
  int v;
  if (envP->ch == '(') {
    MATCH('(', envP);
    v = expr(envP);
    MATCH(')', envP);
  }
  else if (envP->ch == '-') {
    MATCH('-', envP);
    v = -factor(envP);
  }
  else if (isdigit(envP->ch)) {
    char c = envP->ch;
    MATCH(c, envP);
    v = c - '0';
  }
  else {
    error(envP);
  }
  return v;
}
```

References

Text, Ch. 41, 42.

David A. Wheeler, *Program Library HOWTO*, available online.

Oracle, *Linker and Libraries Guide*, available at http://docs.oracle.com/cd/E19253-01/817-1984/.

Michael K. Johnson and Erik W. Troan, *Linux Application Development*, Addison-Wesley, 1998, Ch. 7.

Greg Nakhimovsky, *Building library interposers for fun and profit*, Unix Insider, at http://www.itworld.com/article/2798072/enterprise-software/building-library-interposers-for-fun-and-profit.html.

Timothy W. Curry, *Profiling and Tracing Dynamic Library Usage via Interposition*, USENIX Conference Proceedings, Summer 1994, at http://www.usenix.org/publications/library/proceedings/bos94/curry.html.