POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE ENGINEERING

PROJECT WORK
OF
FORMAL LANGUAGES AND COMPILERS

A subset JavaScript interpreter with Flex and Bison in C

Docente:
Prof. Ing. Scioscia Floriano

Studente:
Altini Nicola

Anno Accademico 2017/2018

Index

Introduction

The purpose of this project work is to implement an interpreter for a subset of JavaScript language. In order to ease the development of the lexer and the parser, Flex and Bison have been utilized.

The core functionality of this interpreter is the creation and evaluation of the Abstract Syntax Tree (AST), which permits to interpret the current source program while performing very important tasks as semantic checks (together with structures as Symbol Table).

In order to do all these things, the project work is made up by the following files (plus the ones generated by Bison and Flex):

- `lexer.l`
- `parser.y`
- `ast.c`
- `ast.h`
- `eval.c`
- `funcs.h`
- `funcs.c`
- `Makefile`

The first two files `lexer.l` `parser.y` are used respectively by Flex and Bison to generate their C and Header files, while `ast.c` `ast.h` contain the functions and data structures (the symbol table too) which are useful to build and manage the AST. The function which is able to walk the AST is contained in `eval.c`. The other functions which are useful for the project are contained in `funcs.c` `funcs.h`.

The interpreter comes with a `Makefile`, so in order to build the project, it is just necessary to use the command `make` from shell in the main project folder.

Please note that, for building the project in a correct way, updated versions of Flex and Bison are required. The project surely works with `Bison 3.0.4` and `Flex 2.6.4`.

The `Makefile` can detect if your OS is Linux or Darwin and build the project in the proper way in both cases. The project has been tested on `Mac OS X 10.13.4` and `Ubuntu 17.10` and work properly in both cases.

Supposing you are in the main folder of the project, it is possible to use the interpreter in two modes:

(1) interactive mode (just running `./bin/interpreter` on the shell)

(2) normal mode, interpreting already saved JS files (typing `./bin/interpreter namefile.js` on the console).

CHAPTER 1. Symbol table

The symbol table is one of the most important data structures when building a translator, allowing to properly store information about identifiers. Each phase of the interpreter has to interact with symbol table, inserting, searching, editing or deleting records.
The symbol table of this project is implemented as a hash table.
The definitions involving the symbol table are listed below:

```
typedef union valueOfSymbol {
  double value;
  char *stringValue;
  double *array;
} valueOfSymbol;

enum symbTableType {
  STT_double = 0,
  STT_string,         // 1
  STT_array_double,   // 2
  STT_function,       // 3
  STT_NOTYPE,         // 4
  STT_NaN,            // 5
  STT_just_asgn,      // 6
  STT_string_no       // 7
};

enum scopeType {
  ST_global = 0,
  ST_function
};

struct evalValueOfSymbol {
  valueOfSymbol val;
  int sizeOfArray;
  enum symbTableType type;
};

struct symbol {
  char *name;                        /* a variable name */
  enum symbTableType type;           /* type of the variable */
  int sizeOfArray;                   /* size of array (if any) */
  valueOfSymbol val;                 /* value of symbol */
  struct ast *func;                 /* stmt for the function */
  struct symlist *syms;              /* list of dummy args */
  int numberOfArguments;             /* number of args for funcs */
  struct arrayElemListAST *arrElemsAST; /* list of AST values of array */
  enum scopeType scope;              /* scope of variable */
};

/* simple symtab of fixed size */
#define NHASH 9997

struct symbol symtab[NHASH];
```

So, the symbol table stores the following information about identifiers:

- `name`: the character string that identifies it.
- `type`: the type of the identifier. This could be one of the following:
    - `STT_double`: the identifier is a double.

    - `STT_string`: the identifier is a string.
    - `STT_array_double`: the identifier is an array of double.
    - `STT_function`: the identifier is a function.
    - `STT_NOTYPE`: the identifier has just been recognized by lexer, but it is not initialized. If trying to use this identifier for printing, evaluating functions or expressions, the parser will return semantic error.
    - `STT_NaN`: the identifier has suffered some impossible arithmetic operation. For instance, product of number and string will result in a `NaN`.
    - `STT_just_asgn`: the identifier has been recognized by lexer and has suffered an assignment operation. This type is assigned before the evaluation of the AST, which returns the correct type for the variable.
    - `STT_string_no`: this is not used in ST, but in evaluation of AST. In this case the AST is a string but should not be printed by the printing function since it has already been printed.
- `sizeOfArray`: this field is 0 if the variable has type `STT_double`, otherwise it contains the length of the array of double or the string. In C the arrays are pointers and so, they do not know their length. Therefore, this parameter is necessary to treat the arrays in a proper way and perform semantic checks as preventing accessing/modifying array elements which are out of range. For proper C strings (NULL terminated strings), it is possible use `strlen()` to determine the value of this field.
- `val`: this field is a C union. This element could be one of:
    - `value`: this field is used if the type is `STT_double` and simply contains the double value of the identifier.
    - `stringValue`: this field is used if the type is `STT_string` (but could be useful even in some other cases) and simply contains a pointer to the array of chars.
    - `array`: this field is used if the type is `STT_array_double` and simply contains a pointer to the first element of array of doubles.
- `func`: this field is used if the type is `STT_function` and contains a pointer to AST which contains the list of statements for a user defined function.
- `syms`: this field is used if the type is `STT_function` and contains a pointer to a list of formal parameters for user defined functions.
- `numberOfArguments`: this field is used if the type is `STT_function` and contains the correct number of arguments for calling this user defined function.

- arrElemsAST: this field is used if the type is STT_array_double and contains a pointer to a list of AST elements. This list is then converted (when evaluating the AST) in a normal C array and the pointer to the array is stored in val.array.
- scope: this field is used to avoid the parser reporting syntactic error when finding identifiers of formal parameters in function body definition.

The struct evalValueOfSymbol which appears above is very important when evaluating the AST, because the function that perform this task, evalStruct(), returns a pointer to struct evalValueOfSymbol.

The hash function implemented is very simple:

```c
static unsigned symhash(char *sym)
{
  unsigned int hash = 0;
  unsigned c;

  while((c = *sym++)) hash = hash * 9 ^ c;

  return hash;
}
```

Collisions are resolved scrolling the symbol table until an empty place is found. The function which perform the search and decide if add or not element to symbol table is the lookup() listed below:

```c
/*
 * THE lookup routine computes the ST entry index as the hash value
 * modulo the size of the symbol table, which was chosen a number with
 * no even factors, again to mix the hash bits up.
 */
struct symbol * lookup(char *sym)
{
  struct symbol *sp = &symtab[symhash(sym)%NHASH];
  int scount = NHASH;          /* how many have we looked at */

  while(--scount >= 0) {
    if(sp->name && !strcmp(sp->name, sym)) { return sp; }

    if(!sp->name) {            /* new entry */
      sp->name = strdup(sym);
      sp->val.value = 0;
      sp->func = NULL;
      sp->syms = NULL;
      sp->arrElemsAST = NULL;
      sp->type = STT_NOTYPE;
      sp->scope = ST_global;
      return sp;
    }

    if(++sp >= symtab+NHASH) sp = symtab; /* try the next entry */
  }
  yyerror("symbol table overflow\n");
  abort(); /* tried them all, table is full */

}
```

If in the symbol table there exists no identifier with the name matched by the lexer, this function creates a new one setting its type to `STT_NOTYPE`, in order to avoid performing operation with a not defined identifier.

CHAPTER 2. Lexical analysis

The purpose of the lexical analyzer is to decompose a sequence of characters into a sequence of lexemes. If this is possible, it returns the sequence of tokens for each lexeme in the source program, otherwise, it returns a lexical error. For each lexeme, the lexer outputs one token.

The lexical elements that the interpreter of this project work could recognized are:

- **Keywords**. Here we have typical JavaScript keywords, as: `if`, `else`, `while`, `do`, `for`, `function`, `return`, `typeof`. In order to ease the development of the lexer and the parser, even other lexical elements have been considered as keywords. For example, JavaScript allows a wide variety of operations on array (treating them as object), so it is common to use instructions like `array.length`, `array.push(value)`, `array.pop()`. In JavaScript things like `length`, `push`, `pop` are not keywords, but, in order to handle them easily, they are considered as keywords by this grammar. The same approach has been used for built-in functions provided by this interpreter (for math or `I/O`). Keywords are used also for typical interactive shell commands like `exit`, `quit` or `clear`.
- **Operands.** In case of operands composed by a single character, as "+", "-", "*", etc., it is possible just to return `yytext[0]`, without introducing a different verbose token for each operand.
- **Numbers**. The regular expression for numbers used in Flex is:

  ```
  [0-9]+"."[0-9]*{EXP}? |
  "."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }
  ```

  Where EXP is float exponent:

  ```
  EXP    ([Ee][-+]?[0-9]+)
  ```

- **Strings**. The regular expression for strings used in Flex is:

  ```
  ["]([^"\\\n]|\\.|\\\n)*["]          { genericString(); return STREXP; }
  ["]([^"\\\n]|\\.|\\\n)*$            { genericStringHalf(); yyerror("Lexical
  error. Unterminated string: '%s'", yylval.str); return STRERR; }
  ```

  Where the function `genericString()` takes care of the fact that strings must be considered without the double quotes. The second pattern here is for detecting lexical error regarding unterminated string. To avoid lexical error, you must use `"\\\n"` instead of `"\n"` when inserting multiline strings.

- **Comments**. The regular expression for comments used in Flex is:

```
"//".*                                          { }
[/][*][^*]*[*]+([^*/][^*]*[*]+)*[/]             { }

[/][*].*$              { yyerror("Lexical error. Unclosed comment");  }
```

The first is used for single-line comments, whilst the second for multi-line comments. No action is associated to comments, in fact, the lexer just ignores them. The third line is used for detecting unclosed comments (they can be detected only at the EOF, otherwise Lexer will try to apply the rule in the 2nd line).

- **Identifiers**. The regular expression for identifiers used in Flex is:

```
[_a-zA-Z][_a-zA-Z0-9]*  { yylval.sym = lookup(yytext); return NAME; }

[0-9]+[_a-zA-Z][_a-zA-Z0-9]* { yyerror("Lexical error. Identifier name:
'%s' starts with a number.", yytext); return ERRNAME; }
```

The action associated with this pattern is very important, because it checks the symbol table to see if this identifier already exists, otherwise creates one.
In an ordinary (nonreentrant) parser, the semantic value of the token must be stored into the global variable `yylval`. When you are using just one data type for semantic values, `yylval` has that type.
When you are using multiple data types, `yylval`'s type is a union made from the `%union` declaration. So, when you store a token's value, you must use the proper member of the union.
The actual union implemented in `parser.y` is:

```
%union {
  struct ast *a;
  double d;
  char *str;
  struct symbol *sym;         /* which symbol */
  struct symlist *sl;
  struct arrayElemList *al;
  int fn;                     /* which function */
}
```

So, the function `lookup()` returns a pointer to a symbol table element.
If a number is matched before identifier's name, then a lexical error is generated, since identifier's name cannot start with a number.

- **White spaces and newlines**. Newlines, whitespaces, tabulations and carriage returns are simply ignored.

```
[ \t\n\r]   /* ignore white space */
```

- **Everything else**. If the lexer founds anything else, it reports a lexical error, but substantially ignores the lexeme, not returning anything to the parser and so continuing scanning.

```
.     { yyerror("Lexical error. Unknown character: '%c'. ASCII: %d\n",
*yytext, (int)yytext[0] ); }
```

CHAPTER 3. Syntax and semantic analysis

Bison allows associating to each rule an action. In this interpreter, these actions are used to build the AST. For example, considering a branch statement, as `IF/THEN` or `IF/THEN/ELSE`. In Bison grammar we have something like:

```
branchstmt:   IF '(' exp ')' bodylist           %prec "then"       { $$ =
                                          newflow('I', $3, $5, NULL); }
            | IF '(' exp ')' bodylist ELSE bodylist                  { $$ =
                                                newflow('I', $3, $5, $7); }
            ;
```

As a convention, nonterminals are lowercase and tokens are uppercase. Terminal of one character are reported verbatim in single quotes. The nonterminal bodylist has the following productions:

```
bodylist:   '{' list '}'      { $$ = $2; }
          | stmt              { $$ = $1; }
          ;
```

An action is a C compound statement that is executed whenever the parser reaches the point in the grammar where the action occurs.
So, when an `IF/THEN` construct is detected by the parser, it will call the `newflow()` function, which builds an AST handling this construct. In order to handle `IF/THEN` `IF/THEN/ELSE`, but also `WHILE/DO` and `DO/WHILE`, the node must have up to 3 children (in the case of `IF/THEN/ELSE`), otherwise 2 children is enough.

```
struct ast * newflow(enum ntype nodetype, struct ast *cond, struct ast *tl,
struct ast *el)
{
  struct flow *a = malloc(sizeof(struct flow));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->cond = cond;
  a->tl = tl;
  a->el = el;
  return (struct ast *)a;
}
```

The node ‘a’ has a nodetype which depends on the particular instruction. An enum has been used to the purpose of labeling each type of node, in order to make proper evaluation of AST. The enum is listed below.

```
enum ntype {
  NT_sum    = '+',
  NT_diff   = '-',
  NT_prod   = '*',
  NT_div    = '/',
  NT_bitwor = '|',
  NT_btwxor = '^',
  NT_btwand = 'B',
  NT_btwnot = '~',
  NT_modul  = '%',
  NT_grea   = '1', // >
  NT_less   = '2', // <
  NT_noteq  = '3', // =!
  NT_equal  = '4', // ==
  NT_grequ  = '5', // >=
  NT_lesseq = '6', // <=
  NT_uminus = 'M',
  NT_list   = 'L',
  NT_ifelse = 'I',
  NT_while  = 'W',
  NT_ref    = 'N', // New reference
  NT_assign = '=',
  NT_builfn = 'F',
  NT_userfn = 'C',
  NT_aronly = 'U',
  NT_string = 'S',
  NT_logand = '&',
  NT_logor  = 'O', // Logic OR
  NT_lognot = '!',
  NT_evarel = '[', // evaluation of ARray Element
  NT_starel = ']', // Set of Array Element
  NT_postin = 'P', // Post increment node
  NT_postde = 'D', // Post decrement node
  NT_prein  = 'R', // Post increment node
  NT_prede  = 'E', // Post decrement node
  NT_forlop = 'X', // For loop
  NT_push   = 'H', // push array
  NT_pop    = 'Z', // pop array
  NT_arrlen = 'Q', // get array length
  NT_const  = 'K',
  NT_typeof = '?',
  NT_typexp = 266,
  NT_asgsum = '(', // NAME+=exp
  NT_asgdif = ')', // NAME-=exp
  NT_pow    = 257,
  NT_asgprd = 258, // NAME*=exp
  NT_asgdiv = 259, // NAME/=exp
  NT_asgmod = 260, // NAME%=exp
  NT_dowhle = 261,
  NT_lshift = 262,
  NT_rshift = 263,
  NT_input  = 265
};
```

When the parser recognizes the start symbol of the grammar, it will run an action which will call the `evalStruct()` function. This function knows what to do exactly for each kind of node, and can traverse the AST which has been built during parsing. In particular, it is performed a depth-first traversal of the tree, recursively visiting the subtrees of each node and then the node itself. The `evalStruct()` function returns the value of the tree or subtree from each call.

In the case of an `IF/THEN` or `IF/THEN/ELSE` construct, the nodetype will be '`I`', and the `evalStruct()` function will handle this kind operation in this way:

```
struct evalValueOfSymbol * evalStruct(struct ast *a)
{
...
/* control flow */
  case 'I':
    evosL = evalStruct( ((struct flow *)a)->cond );
    if ( evosL->val.value != 0) {
      if( ((struct flow *)a)->tl) {
          evosR = evalStruct( ((struct flow *)a)->tl);
      } else {
          evosR->val.value = 0.0; // a default value
          evosR->type = STT_NaN;
          evosR->sizeOfArray = 0;
      }

    } else {
        if( ((struct flow *)a)->el) {
          evosR = evalStruct(((struct flow *)a)->el);
          debug_print("if NO if YES \n");
        } else {
          evosR->val.value = 0.0;         // a default value
          evosR->type = STT_NaN;
          evosR->sizeOfArray = 0;
        }
      }
    type = evosR->type;
    sizeOfArray = evosR->sizeOfArray;
    vos = evosR->val;
    break;
...
}
```

The `WHILE/DO` and `DO/WHILE` constructs work in similar way. The node for the `FOR LOOP` needs 4 children instead. In the bison grammar file:

```
loopstmt:     WHILE '(' exp ')'  bodylist                          { $$ =
newflow('W', $3, $5, NULL); }
            | DO bodylist WHILE '(' exp ')'                        { $$ =
newflow(NT_dowhle, $5, $2, NULL); }
            | FOR '(' assignexp ';' exp ';' exp ')' bodylist       { $$ =
newflowFor($5, $9, $3, $7); }
            ;
```

Where the function `newflowFor()` simply adds the children to the parent node, and label the parent in a proper way (so that `evalStruct()` can understand what to when called).

```
struct ast * newflowFor(struct ast *cond, struct ast *tl, struct ast
*initialAsgn, struct ast *finalIncrement)
{
  struct flowFor *a = malloc(sizeof(struct flowFor));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'X';
  a->cond = cond;
  a->tl = tl;
  a->initialAsgn = initialAsgn;
  a->finalIncrement = finalIncrement;
  return (struct ast *)a;
}
```

In fact, when `evalStruct()` evaluates a node labeled with nodetype ‘`X`’, performs the following actions:

```
struct evalValueOfSymbol * evalStruct(struct ast *a)
{
...
  case 'X':
    v = 0.0;              /* a default value */
    evosR->val.value = v;

    if( ((struct flow *)a)->tl) {
      evalStruct(((struct flowFor *)a)->initialAsgn); // i=0
      while( evalStruct(((struct flow *)a)->cond)->val.value != 0) { // i<n
        evosR = evalStruct(((struct flow *)a)->tl);
        evalStruct(((struct flowFor *)a)->finalIncrement); // i++
      }
    }

    type = evosR->type;
    sizeOfArray = evosR->sizeOfArray;
    vos = evosR->val;
    break;              /* last value is value */

...

}
```

In the whole program, there are approximately 50 distinct kinds of nodetype, which are handled by `evalStruct()`, so it would be boring list them all here, since many are very similar between themselves. A large number of node-types are for handling expression, since we can have arithmetical expressions, logical expressions, bitwise expressions, assignments and even expressions with strings (in order to perform things as concatenation). The Bison grammar for the expressions is:

```
arithmexp:    exp '+' exp             { $$ = newast('+', $1, $3); }
            | exp '-' exp             { $$ = newast('-', $1, $3); }
            | exp '*' exp             { $$ = newast('*', $1, $3); }
            | exp '/' exp             { $$ = newast('/', $1, $3); }
            | exp '%' exp             { $$ = newast('%', $1, $3); }
            | exp POW exp             { $$ = newast(NT_pow, $1, $3); }
            | '(' exp ')'             { $$ = $2; }
            | '-' exp %prec UMINUS    { $$ = newast('M', $2, NULL); }
            | '+' exp %prec UPLUS     { $$ = $2; }
            ;

logicexp:     exp AND exp             { $$ = newast('&', $1, $3); }
            | exp OR  exp             { $$ = newast('O', $1, $3); }
            | NOT exp                 { $$ = newast('!', $2, NULL); }
            ;

bitwisexp:    exp '|' exp             { $$ = newast('|', $1, $3); }
            | exp '^' exp             { $$ = newast('^', $1, $3); }
            | exp '&' exp             { $$ = newast('B', $1, $3); }
            | '~' exp                 { $$ = newast('~', $2, NULL); }
            | exp LEFTSHIFT exp       { $$ = newast(NT_lshift, $1, $3); }
            | exp RIGHTSHIFT exp      { $$ = newast(NT_rshift, $1, $3); }
            ;

incrmentexp:  NAME INCREMENT          { $$ = newIncrement($1, 0); }
            | NAME DECREMENT          { $$ = newDecrement($1, 0); }
            | INCREMENT NAME          { $$ = newIncrement($2, 1); }
            | DECREMENT NAME          { $$ = newDecrement($2, 1); }
            ;

exp: exp CMP exp              { $$ = newcmp($2, $1, $3); }
   | arithmexp                { $$ = $1; }
   | logicexp                 { $$ = $1; }
   | bitwisexp                { $$ = $1; }
   | NUMBER                   { $$ = newnum($1);  }
   | BIFUNC '(' exp ')'       { $$ = newfunc($1, $3); }
   | TYPEOF '(' exp  ')'      { $$ = newTypeOfExp($3); }
   | TYPEOF '(' arraydecls ')' { $$ = newTypeOfExp($3); }
   | NAME                     {
                                  if($1->type == STT_NOTYPE && $1->scope !=
ST_function) { yyerror("Semantic error. The variable '%s' is not defined
yet.", $1->name); YYERROR; } $$ = newref($1); }
   | ERRNAME                  { YYERROR; }
   | assignexp                { $$ = $1; }
   | NAME '(' explist ')'     { $$ = newcall($1, $3); }
   | NAME '(' ')'             { $$ = newcall($1, NULL); }
   | NAME '[' exp ']'         { $$ = newarrayelast($1, $3); }
   | NAME '[' exp ']' '=' exp { $$ = newasgnArray($1, $3, $6); }
   | arrpushpop               { $$ = $1; }
   | incrmentexp              { $$ = $1; }
   | STREXP                   { $$ = newstring($1); }
   | STRERR                   { YYERROR; }
   | READ '(' ')'             { $$ = newinput(); }
  ;
assignexp: NAME '=' exp      { $$ = newasgn($1, $3); }
         | NAME ASGNSUM exp  { $$ = newasgnsumdif($1, $3, 0); }
         | NAME ASGNDIF exp  { $$ = newasgnsumdif($1, $3, 1); }
         | NAME ASGNPROD exp { $$ = newasgnsumdif($1, $3, 2); }
         | NAME ASGNDIV exp  { $$ = newasgnsumdif($1, $3, 3); }
         | NAME ASGNMOD exp  { $$ = newasgnsumdif($1, $3, 4); }
        ;
```

Please note that the use of a not defined identifier produces a semantic error.
An interesting arithmetic expression is the one associated to the binary plus, since it has a different behavior according to the type of the subtrees of its operands.
First, we need to create a node labeled '+' with 2 children:

```
struct ast * newast(enum ntype nodetype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->l = l;
  a->r = r;
  return a;
}
```

(Note that this function can be used by any other binary operator, since it takes nodetype as parameter).
When `evalStruct()` evaluates a node labeled with nodetype '+' it performs an arithmetical sum if both operands are doubles, and a string concatenation if there is a type mismatch. This is possible due to the fact that each subtree of AST knows its type.

```
  case '+': evosL = evalStruct(a->l);
            evosR = evalStruct(a->r);
            if ( (evosL->type == STT_double) && ( evosR->type ==
STT_double) ) {
                v = evosL->val.value + evosR->val.value;
                vos.value = v;
                type = STT_double;
            } else {
                char *al;
                char *ar;
                structToString(&al, evosL);
                structToString(&ar, evosR);
                asprintf(&c, "%s%s", al, ar);

                vos.stringValue = c;
                type = STT_string;
            }
            break
```

In the case of other operators, the type mismatch between operands will produce as result a `NaN`. For example, in the subtraction:

```
case '-': evosL = evalStruct(a->l);
          evosR = evalStruct(a->r);
          if ( (evosL->type == STT_double) && (evosR->type == STT_double) ) {
              v = evosL->val.value - evosR->val.value;
              vos.value = v;
              type = STT_double;
          } else {
              type = STT_NaN;
          }
```

The behavior is similar for others binary (or unary) operators.

3.1 User defined functions

The most difficult thing to implement perhaps was the possibility for the user to define a function. These functions have parameters with unspecified type (in JavaScript fashion) and can return a value (this is of unspecified type too) or not (in this case the user has to state explicitly `return;`). In order to ease the grammar and so the parsing of these functions, a restriction has been done: there must be only one return statement for each function, and it must be at the end of the function definition (otherwise the parser will report a syntactic error). Even if this is an important restriction, the programmer could almost always try to use only a return at the end of a function, perhaps using more temporary variables.

The Bison grammar concerning function definitions is:

```
functionbody: list returnstmt { if ($1 == NULL) $$ = $2;
                                else $$ = newast('L', $1, $2); }
            ;

returnstmt:  RETURN ';'     { $$ = newnum(0); }
           | RETURN exp ';' { $$ = $2; }
           ;

funcdecl:  FUNCTION NAME '(' symlist ')' '{' functionbody '}'   {
                                           dodef($2, $4, $7);   }
            ;

funcnoparamdecl:  FUNCTION NAME '(' ')' '{' functionbody '}'   {
                                        dodef($2, NULL, $6); }
            ;

funcdecls:  funcdecl
          | funcnoparamdecl
          ;
```

The action associated with nonterminal `symlist` builds a list with all formal parameters used in function definition. The scope of these parameters is set to `ST_function`. After the function definition ends, these are put again to `ST_global`. The function which builds the list is:

```c
struct symlist * newsymlist(struct symbol *sym, struct symlist *next)
{
  struct symlist *sl = malloc(sizeof(struct symlist));

  if(!sl) {
    yyerror("out of space");
    exit(0);
  }
  sl->sym = sym;
  sl->next = next;
  return sl;
}
```

And the struct which contains the list of symbols just need to contain a pointer to the symbol and a pointer to the next element in list. In the last element of list, the next pointer links to `NULL` allowing to understand where list ends.

```
/* list of symbols, for an argument list */
struct symlist {
  struct symbol *sym;   /* pointer to symbol */
  struct symlist *next; /* pointer to next element of the list */
};
```

The nonterminal `functionbody` is composed by a list of instruction followed by a return statement. It is necessary to build AST for containing this list of instructions, which will be evaluated in the future when the function will get called with actual parameters.
In order to define a function and store its information in the symbol table, the `dodef()` function is called in the action associated with recognition of full function definition:

```
/* define a function */
void dodef(struct symbol *name, struct symlist *syms, struct ast *func)
{
  struct symlist *sl;
  int nargs;

  if(name->syms) symlistfree(name->syms);
  if(name->func) treefree(name->func);
  name->syms = syms;
  name->func = func;
  name->type = STT_function;

  sl = syms;

  for(nargs = 0; sl; sl = sl->next) {
    nargs++;
    sl->sym->scope = ST_global;
  }

  name->numberOfArguments = nargs;
}
```

With this function we free any eventual precedent definition of the function (`symlistfree()` frees the list of formal parameters already stored in symbol table, while `treefree()` frees the list of statements already stored in symbol table). Then, we assign the new list of formal parameters and the new list of statements in symbol table. We set the type of symbol to `STT_function`. Then we calculate the correct number of arguments (for further semantic check when the function is called) by traversing the list until a `NULL` pointer is found and assign this value to the appropriate field in symbol table.
At this point we have all the information that we will need for further function calls stored in symbol table.
According to the grammar, a function call is:

```
   | NAME '(' explist ')'      { $$ = newcall($1, $3); }
   | NAME '(' ')'              { $$ = newcall($1, NULL); }
```

Where the function `newcall()` is:

```
struct ast * newcall(struct symbol *s, struct ast *l)
{
  struct ufncall *a = malloc(sizeof(struct ufncall));
  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'C';
  a->l = l;
  a->s = s;
  return (struct ast *)a;
}
```

This function just labels this node with nodetype ‘`C`’ and gives to it 2 children: the pointer to symbol (the name of the function) and the pointer to the AST which contains the list of actual parameters.

When `structEval()` encounters this kind of node:

```
  /* call to user defined function */
  case 'C': if( ((struct ufncall *)a)->s->type == STT_function ) {
              evos = calluserStruct((struct ufncall *)a);
              return evos;
            } else {
              yyerror("Semantic error. The identifier '%s' is not a
function", ((struct ufncall *)a)->s->name);
              evos->sizeOfArray = 0;
              evos->type = STT_NOTYPE;
              return evos;
            }
```

First, it checks if this is actually a function, otherwise returns a semantic error. If this is a function, then the `calluserStruct()` function is called.

```
static struct evalValueOfSymbol * calluserStruct(struct ufncall *f)
{

...

}
```

This function performs the following tasks:

(1) Get the number of actual arguments by determining tree depth.

(2) Check if the number of arguments is correct.

(3) Allocate enough space in memory to save values of variables which have the same name of the ones of formal parameters. For example, if we have a variable named ‘`a`’ and a function `f(a)`, we do not want to confuse global variable ‘`a`’ with local variable ‘`a`’. We need to save the values of variables homonymous to formal parameters. In particular, we have to save these fields:

- `val`
- `sizeOfArray`

type

(4) Evaluate the actual arguments, using `evalStruct()`. This must be done since each actual argument can be an arbitrarily complex AST.

(5) Evaluate the body of the function, which will now use the actual argument values when referring to the formal arguments, since the former values in ST have been changed.

(6) Put back the old values of the formal parameters (the values we save in step `(3)`), thus repristinating the original status of ST (the one before that the function call occurs).

(7) Return the value of the body expression.

3.2 Arrays

Another important task is declaring array and handling correctly all associated information in the symbol table. In the grammar:

```
arraydeclright:   '[' arrayelemsast ']'      { $$ = $2; }
                | '[' ']'                    { $$ = NULL; }
                ;
arraydecl: arraydecls ';'

arraydecls:  NAME '=' arraydeclright       { $$ = dodefArrayAST($1, $3); }
           | arraydeclright              {
                symbTEMP = lookup("_"); $$ = dodefArrayAST(symbTEMP, $1); }
        ;

arrayelemsast:  exp                       { $$ = newElemListAST ($1, NULL); }
              | exp ',' arrayelemsast   { $$ = newElemListAST ($1, $3); }
              ;

arrpushpop:   NAME '.' LENGTH             { $$ = getSizeOfArrayNameAST($1); }
            | NAME '.' PUSH '(' exp ')' { $$ = makePushOnArrayAST($1, $5);}
            | NAME '.' POP  '(' ')'       { $$ = makePopFromArrayAST($1); }
;
```

When an array is declared, the function `dodefArrayAST()` is invoked, creating proper node and children in the AST:

```
struct ast * dodefArrayAST(struct symbol *name, struct arrayElemListAST *ar-
rElems)
{
    struct arrayDef *a = malloc(sizeof(struct arrayDef));

    if(!a) {
      yyerror("out of space");
      exit(0);
    }

    a->nodetype = 'U';
    a->s = name;
    a->arrElems = arrElems;
    a->s->type = STT_array_double;
    return (struct ast *)a;

}
```

When the interpreter parses an array declaration, it builds a list of parameters. We save the AST for each of the parameters of this list, allowing correct evaluation later.

```
struct arrayElemListAST * newElemListAST(struct ast *a, struct arrayElemLis-
tAST *next)
{
  struct arrayElemListAST *sl = malloc(sizeof(struct arrayElemListAST));

  if(!sl) {
    yyerror("out of space");
    exit(0);
  }

  sl->next = next;
  sl->valueAST = a;
  return sl;
}
```

When `evalStruct()` encounters a node labeled with nodetype ‘`U`’:

```
case 'U': dodefArray( ((struct arrayDef *)a)->s, ((struct arrayDef *)a)-
>arrElems );
          vos.array = ((struct arrayDef *)a)->s->val.array;
          type = STT_array_double;
          sizeOfArray = ((struct arrayDef *)a)->s->sizeOfArray;
          break;
```

Only at this moment each AST corresponding to array elements is evaluated, and the list is converted into a standard C double array.

To handle operations as `length`, `push()` and `pop()` on arrays, we have to create appropriate AST and then evaluate correctly with `evalStruct()`.

```
struct ast * makePushOnArrayAST(struct symbol *name, struct ast *
valueToPush)
{
  struct symasgn *a = malloc(sizeof(struct symasgn));
  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'H';
  a->s = name;
  a->v = valueToPush;
  return (struct ast *)a;
}

struct ast * makePopFromArrayAST(struct symbol *name)
{
    struct symref *a = malloc(sizeof(struct symref));
    if(!a) {
      yyerror("out of space");
      exit(0);
    }
    a->nodetype = 'Z';
    a->s = name;
    return (struct ast *)a;
}
```

3.3 Error Handling

The Bison parser detects a syntax error (or parse error) whenever it reads a token which cannot satisfy any syntax rule. An action in the grammar can also explicitly proclaim an error, using the macro YYERROR.

In order to allow error recovery in any situation, the Bison error keyword has been inserted in a start symbol production rule:

```
interprlist:  %empty
            | interprlist stmt {
                                  if(debug) dumpast($2, 0);
                                  handleOutputSwitchStruct($2);
                                  if(interactiveOrFile) printf(KNRM "> ");
                                  treefree($2);
                                }
            | interprlist funcdecls  {
                                 if(interactiveOrFile) printf(KNRM "> ");
                               }

            | interprlist error {  if(interactiveOrFile) printf("> ");}
            ;
```

The yyerror() function provided with this interpreter is enhanced in order to support generic number of arguments and indicate line and column numbers of errors detected:

```
void yyerror(char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  if (numberOfErrors < TOOMANYERRORS) {
    fprintf(stderr, KRED "Line %d:[%d-%d]: error: ", yylineno,
yylloc.first_column, yylloc.last_column);
    vfprintf(stderr, s, ap);
    fprintf(stderr, KNRM "\n");
  } else {
    fprintf(stderr, KRED "Too many errors in your program (more than %d).
Exit.\n" KNRM, TOOMANYERRORS);
    exit(0);
  }

  numberOfErrors++;
}
```

This function comes with a counter of number of errors. This feature is usually indispensable in compilers, since above a certain number of errors they stop reporting errors for avoiding reporting an incomprehensible plethora of mistakes. But, in an interpreter, the utility of such a feature is not so clear, so it is implemented just for testing purposes. The maximum number of error detectable before exiting is stored in the macro TOOMANYERRORS.

In order to correctly count the column numbers of mistakes, we have to redefine the Flex macro YY_USER_ACTION:

```
#define YY_USER_ACTION  yylloc.first_line = yylloc.last_line; \
                        yylloc.first_column = yylloc.last_column; \
                        for (int i = 0; yytext[i] != '\0'; i++) { \
                            if(yytext[i] == '\n') { \
                                yylloc.last_line++; \
                                yylloc.last_column = 0; \
                            } \
                            else { \
                                yylloc.last_column++; \
                            } \
                        }
```

The function that print the number of errors before exiting the interpreter is:

```
void printErrorNumbers() {
    if (numberOfErrors == 0) {
        exit(0);
        return;
    } else {
        printf(KRED "Found: %d errors\n", numberOfErrors);
    }
    exit(0);
}
```

This function is called if the user inserts `exit` or `quit` commands (this is useful for interactive use), or when the `EOF` is encountered.

Bibliography

[1] Floriano Scioscia. Lecture materials of Formal Languages and Compilers course.

[2] Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques and Tools. Pearson New International Edition.

[3] C. Donnelly and R. Stallman. Bison: The Yacc-compatible Parser Generator. 23 October 2013, Bison Version 3.0.2.

[4] J. Levine. Flex & Bison – Unix Text Processing Tools. O'Reilly.