CIS 606: Programming Project 1

Brian Howard

June 11, 1996

Contents

1 Introduction

The first programming assignment is to construct a lexical analyzer and symbol table for PSub, a subset of Pascal based on the language described in the Appendix of the Dragon book. This "literate program" was prepared using `noweb`.

2 Main Program

The main program for this phase repeatedly calls the lexer to get the next token from the input (`stdin` by default, or another file may be specified as a command line argument), then prints it out. At the end, it prints out the symbol table which has been constructed.

2 ⟨*pp1.c* 2⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include "global.h"
#include "symtab.h"
#include "token.h"
#include "lexer.h"

char *progname, /* name under which lexer is being run */
     *filename; /* name of PSub input file */

main(int argc, char *argv[]) {
        Token currTok;
        FILE *fileptr;

        ⟨process arguments 3a⟩

        do {
                currTok = getNextToken();
                printf("%-20s %-20s %-20s\n",
                       showTokenKind(getTokenKind(currTok)),
                       getTokenLexeme(currTok),
                       showTokenValue(getTokenValue(currTok)));
        } while (getTokenKind(currTok) != EofTOK);
        printf("\n");
        printSymbolTable();
        exit(EXIT_SUCCESS);
}
```

Defines:
`argc`, used in chunk 3a.
`argv`, used in chunk 3a.
`filename`, used in chunks 3, 10b, and 11c.
`fileptr`, used in chunks 3a, 7b, and 11d.
`progname`, used in chunks 3, 5, and 7a.

Uses `getNextToken` 8c, `getTokenKind` 14, `getTokenLexeme` 14, `getTokenValue` 14, `printSymbolTable` 6, `showTokenKind` 15, `showTokenValue` 16, and `Token` 12a.

If there are any arguments, the first is taken as the name of a PSub program to be lexed. Otherwise, the standard input is lexed.

3a ⟨*process arguments* 3a⟩≡ (2)
```
progname = argv[0];
if (argc > 1) {
        filename = argv[1];
        if (fileptr = fopen(filename, "r")) {
                initLexer(fileptr);
        } else {
                perror(progname);
                exit(EXIT_FAILURE);
        }
} else {
        filename = "stdin";
        initLexer(stdin);
}
```
Uses `argc` 2, `argv` 2, `filename` 2, `fileptr` 2, `initLexer` 11d, and `progname` 2.

2.1 Global Variables

We maintain the program name, input file name, and current line number for reporting errors.

3b ⟨*global.h* 3b⟩≡
```
#ifndef GLOBAL_H
#define GLOBAL_H

extern char *progname;
extern char *filename;
extern int lineno;

#endif GLOBAL_H
```
Uses `filename` 2, `lineno` 8b, and `progname` 2.

3 Symbol Table

3.1 Interface

The interface to the symbol table consists of a definition for the abstract type `SymTabEntry` and a function `lookupSymbol()` which takes a string argument and returns the corresponding table entry for that symbol, creating a new entry if necessary. There is also a function `printSymbolTable()` which prints the contents of the table for debugging purposes.

3c ⟨*symtab.h* 3c⟩≡
```
#ifndef SYMTAB_H
#define SYMTAB_H

⟨SymTabEntry ADT 4⟩

SymTabEntry lookupSymbol(const char *symbol);

void printSymbolTable(void);

#endif SYMTAB_H
```
Uses `lookupSymbol` 5, `printSymbolTable` 6, and `SymTabEntry` 4.

For this project, each symbol table entry only needs to contain the name of the identifier; future projects will require additional fields such as type and scope information. Identifier names are assumed to be less than `MAXSYMLEN` characters, although attempts are made to check that this is not exceeded.

4 ⟨*SymTabEntry ADT* 4⟩≡ (3c)

```
#define MAXSYMLEN 100
typedef struct {
        char name[MAXSYMLEN];
} *SymTabEntry;

/* Constructor */
SymTabEntry createEntry(const char *name);

/* Destructor */
void destroyEntry(SymTabEntry entry);

/* Selector */
char *getSymbolName(const SymTabEntry entry);
```

Defines:
  `MAXSYMLEN`, used in chunks 5 and 7a.
  `SymTabEntry`, used in chunks 3c, 5–7, 13a, and 16.
Uses `createEntry` 7a, `destroyEntry` 7a, and `getSymbolName` 7a.

3.2 Implementation

The symbol table implementation uses the C library function `tsearch()`. From the man page:

```
#include <search.h>

void *tsearch(const void *key, void **rootp,
     int (*compar)(const void *, const void *));
```

> `tsearch()` is a routine for manipulating binary search trees. It is generalized from *Knuth (6.2.2) Algorithm T.* All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
>
> `tsearch()` is used to build and access the tree. `key` is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to `*key` (the value pointed to by `key`), a pointer to this found datum is returned. Otherwise, `*key` is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. `rootp` points to a variable that points to the root of a tree. A `NULL` value for the variable pointed to by `rootp` denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

The comparison function we provide uses `strncasecmp()` to perform a case-insensitive test, since case is not significant in PSub.

5 ⟨*symtab.c* 5⟩≡ 6▷

```
#include <search.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "global.h"
#include "symtab.h"

⟨SymTabEntry ADT implementation 7a⟩

static void *root = NULL; /* the root of the symbol table */

static int compareEntries(const void *entry1, const void *entry2) {
        return strncasecmp(((const SymTabEntry)entry1)->name,
                           ((const SymTabEntry)entry2)->name,
                           MAXSYMLEN);
}

SymTabEntry lookupSymbol(const char *symbol) {
        SymTabEntry newEntry, *foundEntryPtr;

        newEntry = createEntry(symbol);
        foundEntryPtr = (SymTabEntry *)tsearch((void *)newEntry, &root,
                                               compareEntries);
        if (foundEntryPtr == NULL) {
                perror(progname);
        }
        if (*foundEntryPtr != newEntry) {
                destroyEntry(newEntry);
        }
        return *foundEntryPtr;
}
```

Defines:

`lookupSymbol`, used in chunks 3c and 11b.
`root`, used in chunk 6.
Uses `createEntry` 7a, `destroyEntry` 7a, `MAXSYMLEN` 4, `progname` 2, and `SymTabEntry` 4.

To print the symbol table, we use the C library function `twalk()`. From the man page:

> `#include <search.h>`
>
> `void twalk(void *root, void (*action)(void *, VISIT, int));`
>
> `twalk()` traverses a binary search tree. `root` is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) `action` is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type `typedef enum { preorder, postorder, endorder, leaf } VISIT;` (defined in the `<search.h>` header), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

Our auxiliary function `printEntry()` prints the symbol table entry when the `VISIT` parameter is either `postorder` or `leaf`, which means that each interior node is printed between its left child and its right child; this gives what is commonly called an *inorder* traversal (despite the misleading use of `postorder`). The result is that the entries are printed in sorted order.

6 ⟨*symtab.c* 5⟩+≡ ◁5

```
static void printEntry(const void *entry, VISIT order, int level) {
        if (order == postorder || order == leaf) {
                printf("%s\n", getSymbolName(*(SymTabEntry *)entry));
        }
}

void printSymbolTable(void) {
        printf("Symbol Table\n");
        printf("------------\n");
        twalk(root, printEntry);
        printf("------------\n");
}
```

Defines:
`printSymbolTable`, used in chunks 2 and 3c.
Uses `getSymbolName` 7a, `root` 5, and `SymTabEntry` 4.

7a ⟨*SymTabEntry ADT implementation* 7a⟩≡ (5)
```
SymTabEntry createEntry(const char *name) {
        SymTabEntry entry;

        if (entry = (SymTabEntry)malloc(sizeof entry)) {
                entry->name[0] = 0; /* set to null string */
                strncat(entry->name, name, MAXSYMLEN);
                return entry;
        } else {
                perror(progname);
        }
}

void destroyEntry(SymTabEntry entry) {
        free((void *)entry);
}

char *getSymbolName(const SymTabEntry entry) {
        return entry->name;
}
```
Defines:
`createEntry`, used in chunks 4 and 5.
`destroyEntry`, used in chunks 4 and 5.
`getSymbolName`, used in chunks 4, 6, and 16.
Uses `MAXSYMLEN` 4, `progname` 2, and `SymTabEntry` 4.

4 Lexical Analyzer

4.1 Lexer Interface

The interface to the lexer consists of two functions: `getNextToken()`, which returns the next token found on the input, and `initLexer()`, which takes as an argument the file pointer which the lexer should use for input. It depends on the abstract type `Token`, defined in `token.h`, as well as `SymTabEntry`, defined in `symtab.h`.

7b ⟨*lexer.h* 7b⟩≡
```
#ifndef LEXER_H
#define LEXER_H

Token getNextToken(void);

void initLexer(FILE *fileptr);

#endif LEXER_H
```
Uses `fileptr` 2, `getNextToken` 8c, `initLexer` 11d, and `Token` 12a.

4.2 Lexer Implementation

Here is the structure of the lex specification:

8a ⟨*lexer.l* 8a⟩≡

```
%{
#include <stdlib.h>
#include "global.h"
#include "symtab.h"
#include "token.h"
#include "lexer.h"
⟨lexical C declarations 8b⟩
%}
⟨lexical definitions 9a⟩
%%
⟨lexical rules 10b⟩
%%
⟨lexical C functions 11d⟩
```

The actions of the lexer include keeping track of the current line number, `lineno`, for reporting errors:

8b ⟨*lexical C declarations* 8b⟩≡ (8a) 8c ▷

```
int lineno = 1;
```

Defines:
`lineno`, used in chunks 3b, 10b, and 11c.

`flex` uses the macro `YY_DECL` to declare the lexing function; we want it to be called `getNextToken()` and have return type `Token`:

8c ⟨*lexical C declarations* 8b⟩+≡ (8a) ◁ 8b 8d ▷

```
#define YY_DECL Token getNextToken(void)
```

Defines:
`getNextToken`, used in chunks 2 and 7b.
Uses `Token` 12a.

The scanned token will be collected in `currToken`. For convenience, we define the macro `setCT()` to initialize the appropriate fields using `setToken()`:

8d ⟨*lexical C declarations* 8b⟩+≡ (8a) ◁ 8c

```
Token currToken;

#define setCT(kind,val) setToken(currToken, kind, yytext, val)
```

Defines:
`setCT`, used in chunks 10 and 11.
Uses `setToken` 14 and `Token` 12a.

The definitions start with a case-insensitive version of the alphabet:

9a ⟨*lexical definitions* 9a⟩≡ (8a) 9b ▷

```
a       [Aa]
b       [Bb]
c       [Cc]
d       [Dd]
e       [Ee]
f       [Ff]
g       [Gg]
h       [Hh]
i       [Ii]
j       [Jj]
k       [Kk]
l       [Ll]
m       [Mm]
n       [Nn]
o       [Oo]
p       [Pp]
q       [Qq]
r       [Rr]
s       [Ss]
t       [Tt]
u       [Uu]
v       [Vv]
w       [Ww]
x       [Xx]
y       [Yy]
z       [Zz]
```

An integer is one or more digits, and a real number is an integer followed by either a fractional part or an exponent, or both:

9b ⟨*lexical definitions* 9a⟩+≡ (8a) ◁ 9a 9c ▷

```
integer [0-9]+
frac    "."{integer}
expt    {e}[-+]?{integer}
real    {integer}({frac}{expt}?|{expt})
```

Defines:
`integer`, used in chunk 11b.
`real`, used in chunk 11b.

An identifier is a letter followed by one or more alphanumerics; the case is preserved but is not significant in searching the symbol table.

9c ⟨*lexical definitions* 9a⟩+≡ (8a) ◁ 9b 9d ▷

```
ident   [A-Za-z][A-Za-z0-9]*
```

Defines:
`ident`, used in chunk 11b.

Whitespace is spaces, tabs, carriage returns, and newlines. Newlines are handled separately to maintain the line number count

9d ⟨*lexical definitions* 9a⟩+≡ (8a) ◁ 9c 10a ▷

```
white   [ \t\r]+
newline \n
```

Defines:
`newline`, used in chunk 10b.
`white`, used in chunk 10b.

The exclusive states COMMENT1 and COMMENT2 handle the body of comments (respectively surrounded by `{ }` and `(* *)`, so that we can continue to process newlines inside them.

10a ⟨*lexical definitions* 9a⟩+≡ (8a) ◁9d

```
%x COMMENT1 COMMENT2
```

Defines:
`COMMENT1`, used in chunk 10b.
`COMMENT2`, used in chunk 10b.

Here are the rules for dealing with whitespace and comments:

10b ⟨*lexical rules* 10b⟩≡ (8a) 10c▷

```
{white} /* eat up whitespace */

{newline} lineno++;

"{" BEGIN(COMMENT1);
"(*" BEGIN(COMMENT2);

<COMMENT1,COMMENT2>{newline} lineno++;

<COMMENT1>[^}\n]+ /* ignore the comment body except for newlines */
<COMMENT2>[^*\n]+ /* ignore the comment body except for newlines */
<COMMENT2>"*" /* ignore lone stars in the comment body */

<COMMENT1>"}" BEGIN(INITIAL);
<COMMENT2>"*)" BEGIN(INITIAL);

<COMMENT1,COMMENT2><<EOF>> {
        fprintf(stderr, "Error: unclosed comment at EOF in %s, line %d\n",
                filename, lineno);
        return setCT(EofTOK, setNoValue());
        }

<<EOF>> return setCT(EofTOK, setNoValue());
```

Uses `COMMENT1` 10a, `COMMENT2` 10a, `filename` 2, `lineno` 8b, `newline` 9d, `setCT` 8d, `setNoValue` 16, and `white` 9d.

Here are the rules for dealing with keywords:

10c ⟨*lexical rules* 10b⟩+≡ (8a) ◁10b 11a▷

```
{p}{r}{o}{g}{r}{a}{m}       return setCT(ProgramTOK, setNoValue());
{p}{r}{o}{c}{e}{d}{u}{r}{e} return setCT(ProcedureTOK, setNoValue());
{f}{u}{n}{c}{t}{i}{o}{n}    return setCT(FunctionTOK, setNoValue());
{b}{e}{g}{i}{n}             return setCT(BeginTOK, setNoValue());
{e}{n}{d}                   return setCT(EndTOK, setNoValue());
{i}{f}                      return setCT(IfTOK, setNoValue());
{t}{h}{e}{n}                return setCT(ThenTOK, setNoValue());
{e}{l}{s}{e}                return setCT(ElseTOK, setNoValue());
{w}{h}{i}{l}{e}             return setCT(WhileTOK, setNoValue());
{d}{o}                      return setCT(DoTOK, setNoValue());
{r}{e}{p}{e}{a}{t}          return setCT(RepeatTOK, setNoValue());
{u}{n}{t}{i}{l}             return setCT(UntilTOK, setNoValue());
{v}{a}{r}                   return setCT(VarTOK, setNoValue());
{i}{n}{t}{e}{g}{e}{r}       return setCT(IntegerTOK, setNoValue());
{r}{e}{a}{l}                return setCT(RealTOK, setNoValue());
{b}{o}{o}{l}{e}{a}{n}       return setCT(BooleanTOK, setNoValue());
{n}{o}{t}                   return setCT(NotTOK, setNoValue());
```

Uses `setCT` 8d and `setNoValue` 16.

Here are the rules for dealing with operators:

11a ⟨*lexical rules* 10b⟩+≡ (8a) ◁10c 11b▷
```
":="    return setCT(AssignTOK, setNoValue());

"="     return setCT(RelOpTOK, setOpValue(EqOP));
"<>"    return setCT(RelOpTOK, setOpValue(NotEqOP));
"<"     return setCT(RelOpTOK, setOpValue(LessOP));
"<="    return setCT(RelOpTOK, setOpValue(LessEqOP));
">"     return setCT(RelOpTOK, setOpValue(GrtrOP));
">="    return setCT(RelOpTOK, setOpValue(GrtrEqOP));

{o}{r}  return setCT(AddOpTOK, setOpValue(OrOP));
"+"     return setCT(AddOpTOK, setOpValue(PlusOP));
"-"     return setCT(AddOpTOK, setOpValue(MinusOP));

{a}{n}{d}       return setCT(MulOpTOK, setOpValue(AndOP));
{m}{o}{d}       return setCT(MulOpTOK, setOpValue(ModOP));
{d}{i}{v}       return setCT(MulOpTOK, setOpValue(DivOP));
"*"             return setCT(MulOpTOK, setOpValue(TimesOP));
"/"             return setCT(MulOpTOK, setOpValue(DivideOP));
```
Uses `setCT` 8d, `setNoValue` 16, and `setOpValue` 16.

Here are the rules for dealing with identifiers and numbers (note that the identifier rule has to come after all of the keyword tokens):

11b ⟨*lexical rules* 10b⟩+≡ (8a) ◁11a 11c▷
```
{ident} {
        return setCT(IdentifierTOK, setSymbolValue(lookupSymbol(yytext)));
        }

{integer} return setCT(NumberTOK, setIntegerValue(atoi(yytext)));
{real}    return setCT(NumberTOK, setRealValue(atof(yytext)));
```
Uses `ident` 9c, `integer` 9b, `lookupSymbol` 5, `real` 9b, `setCT` 8d, `setIntegerValue` 16, `setRealValue` 16, and `setSymbolValue` 16.

And here are the rules for dealing with punctuation and everything else:

11c ⟨*lexical rules* 10b⟩+≡ (8a) ◁11b
```
"("     return setCT(LParenTOK, setNoValue());
")"     return setCT(RParenTOK, setNoValue());
":"     return setCT(ColonTOK, setNoValue());
";"     return setCT(SemicolonTOK, setNoValue());
","     return setCT(CommaTOK, setNoValue());
"."     return setCT(PeriodTOK, setNoValue());
.       {
        fprintf(stderr, "Error: illegal character %s in %s, line %d\n",
                yytext, filename, lineno);
        }
```
Uses `filename` 2, `lineno` 8b, `setCT` 8d, and `setNoValue` 16.

All we need to do to initialize the lexer is set `yyin` to the desired input file pointer.

11d ⟨*lexical C functions* 11d⟩≡ (8a)
```
void initLexer(FILE *fileptr) {
        yyin = fileptr;
}
```
Defines:
`initLexer`, used in chunks 3a and 7b.

Uses `fileptr` 2.

4.3 Token Interface

A `Token` contains the kind of the token, the actual lexeme found, and the corresponding interpretation of that lexeme as a value (in the case of a token kind where the lexeme provides additional information). For example, the token `3.14` has kind `NumberTOK`, lexeme `"3.14"`, and value `3.14` (as a `double`).

12a ⟨*token.h* 12a⟩≡

```
#ifndef TOKEN_H
#define TOKEN_H

⟨TokenKind ADT 12b⟩
⟨TokenValue ADT 13a⟩

typedef struct {
        TokenKind kind;
        char *lexeme;
        TokenValue val;
} Token;

/* Initializer */
Token setToken(Token token, const TokenKind kind, char *lexeme,
               const TokenValue val);

/* Selectors */
TokenKind getTokenKind(const Token token);
char *getTokenLexeme(const Token token);
TokenValue getTokenValue(const Token token);

#endif TOKEN_H
```

Defines:
`Token`, used in chunks 2, 7, 8, and 14.
Uses `getTokenKind` 14, `getTokenLexeme` 14, `getTokenValue` 14, `setToken` 14, `TokenKind` 12b, and `TokenValue` 13a.

Here are all of the token kinds of PSub, including the special kind `EofTOK` which is only returned on end-of-file.

12b ⟨*TokenKind ADT* 12b⟩≡ (12a)

```
typedef enum {
        IdentifierTOK, NumberTOK,
        ProgramTOK, ProcedureTOK, FunctionTOK,
        BeginTOK, EndTOK, IfTOK, ThenTOK, ElseTOK,
        WhileTOK, DoTOK, RepeatTOK, UntilTOK,
        VarTOK, IntegerTOK, RealTOK, BooleanTOK,
        NotTOK, AssignTOK, RelOpTOK, AddOpTOK, MulOpTOK,
        LParenTOK, RParenTOK, ColonTOK, SemicolonTOK,
        CommaTOK, PeriodTOK, EofTOK
} TokenKind;

/* Converter */
char *showTokenKind(const TokenKind kind);
```

Defines:
`TokenKind`, used in chunks 12a, 14, and 15.
Uses `showTokenKind` 15.

The value component of a **Token** may be one of several types: `int`, `double`, `SymTabEntry`, or `OpKind`. The `type` field identifies what sort of datum is in the `value` field; if it is `NoVAL`, then there is no associated value for this token (for example, it might just be a keyword or punctuation).

13a ⟨*TokenValue ADT* 13a⟩≡ (12a)

```
typedef enum {
        IntegerVAL, RealVAL, SymbolVAL, OpVAL, NoVAL
} TokenValueType;

⟨OpKind ADT 13b⟩

typedef struct {
        TokenValueType  type;
        union {
                int             intval;
                double          realval;
                SymTabEntry     symbolval;
                OpKind          opval;
        } value;
} TokenValue;

/* Initializers */
TokenValue setIntegerValue(const int intval);
TokenValue setRealValue(const double realval);
TokenValue setSymbolValue(const SymTabEntry symbolval);
TokenValue setOpValue(const int opval);
TokenValue setNoValue(void);

/* Converter */
char *showTokenValue(const TokenValue value);

/* Selectors will be needed in future projects, but not yet */
```

Defines:
`TokenValue`, used in chunks 12a, 14, and 16.
Uses `OpKind` 13b, `setIntegerValue` 16, `setNoValue` 16, `setOpValue` 16, `setRealValue` 16, `setSymbolValue` 16, `showTokenValue` 16, and `SymTabEntry` 4.

The type `OpKind` distinguishes which particular operator was seen, since the token kind only discriminates between relational, additive, and multiplicative operators.

13b ⟨*OpKind ADT* 13b⟩≡ (13a)

```
typedef enum {
        EqOP, NotEqOP, LessOP, LessEqOP, GrtrOP, GrtrEqOP, /* RelOps */
        OrOP, PlusOP, MinusOP,                             /* AddOps */
        AndOP, DivOP, ModOP, TimesOP, DivideOP             /* MulOps */
} OpKind;

/* Converter */
char *showOpKind(const OpKind kind);
```

Defines:
`OpKind`, used in chunks 13a and 17.
Uses `showOpKind`.

4.4 Token Implementation

14 ⟨*token.c* 14⟩≡

```
#include "global.h"
#include "symtab.h"
#include "token.h"

⟨TokenKind ADT implementation 15⟩
⟨TokenValue ADT implementation 16⟩

Token setToken(Token token, const TokenKind kind, char *lexeme,
               const TokenValue val) {
        token.kind = kind;
        token.lexeme = lexeme;
        token.val = val;
        return token;
}

TokenKind getTokenKind(const Token token) {
        return token.kind;
}

char *getTokenLexeme(const Token token) {
        return token.lexeme;
}

TokenValue getTokenValue(const Token token) {
        return token.val;
}
```

Defines:
- `getTokenKind`, used in chunks 2 and 12a.
- `getTokenLexeme`, used in chunks 2 and 12a.
- `getTokenValue`, used in chunks 2 and 12a.
- `setToken`, used in chunks 8d and 12a.

Uses `Token` 12a, `TokenKind` 12b, and `TokenValue` 13a.

15 ⟨*TokenKind ADT implementation* 15⟩≡ (14)

```
char *showTokenKind(const TokenKind kind) {
        switch (kind) {
                case IdentifierTOK:     return "IdentifierTOK";
                case NumberTOK:         return "NumberTOK";
                case ProgramTOK:        return "ProgramTOK";
                case ProcedureTOK:      return "ProcedureTOK";
                case FunctionTOK:       return "FunctionTOK";
                case BeginTOK:          return "BeginTOK";
                case EndTOK:            return "EndTOK";
                case IfTOK:             return "IfTOK";
                case ThenTOK:           return "ThenTOK";
                case ElseTOK:           return "ElseTOK";
                case WhileTOK:          return "WhileTOK";
                case DoTOK:             return "DoTOK";
                case RepeatTOK:         return "RepeatTOK";
                case UntilTOK:          return "UntilTOK";
                case VarTOK:            return "VarTOK";
                case IntegerTOK:        return "IntegerTOK";
                case RealTOK:           return "RealTOK";
                case BooleanTOK:        return "BooleanTOK";
                case NotTOK:            return "NotTOK";
                case AssignTOK:         return "AssignTOK";
                case RelOpTOK:          return "RelOpTOK";
                case AddOpTOK:          return "AddOpTOK";
                case MulOpTOK:          return "MulOpTOK";
                case LParenTOK:         return "LParenTOK";
                case RParenTOK:         return "RParenTOK";
                case ColonTOK:          return "ColonTOK";
                case SemicolonTOK:      return "SemicolonTOK";
                case CommaTOK:          return "CommaTOK";
                case PeriodTOK:         return "PeriodTOK";
                case EofTOK:            return "EofTOK";
        }
}
```

Defines:
`showTokenKind`, used in chunks 2 and 12b.
Uses `TokenKind` 12b.

16 ⟨*TokenValue ADT implementation* 16⟩≡ (14)
⟨*OpKind ADT implementation* 17⟩

```
TokenValue currTokenValue;

TokenValue setIntegerValue(const int intval) {
        currTokenValue.type = IntegerVAL;
        currTokenValue.value.intval = intval;
        return currTokenValue;
}

TokenValue setRealValue(const double realval) {
        currTokenValue.type = RealVAL;
        currTokenValue.value.realval = realval;
        return currTokenValue;
}

TokenValue setSymbolValue(const SymTabEntry symbolval) {
        currTokenValue.type = SymbolVAL;
        currTokenValue.value.symbolval = symbolval;
        return currTokenValue;
}

TokenValue setOpValue(const int opval) {
        currTokenValue.type = OpVAL;
        currTokenValue.value.opval = opval;
        return currTokenValue;
}

TokenValue setNoValue(void) {
        currTokenValue.type = NoVAL;
        return currTokenValue;
}

/* a buffer in which to print out numbers */
static char valueString[100];

char *showTokenValue(const TokenValue val) {
        switch (val.type) {
                case IntegerVAL:
                        sprintf(valueString, "%d", val.value.intval);
                        return valueString;
                case RealVAL:
                        sprintf(valueString, "%G", val.value.realval);
                        return valueString;
                case SymbolVAL:
                        return getSymbolName(val.value.symbolval);
                case OpVAL:
                        return showOpKind(val.value.opval);
                case NoVAL:
                        return "";
        }
}
```

Defines:
- `setIntegerValue`, used in chunks 11b and 13a.
- `setNoValue`, used in chunks 10, 11, and 13a.
- `setOpValue`, used in chunks 11a and 13a.
- `setRealValue`, used in chunks 11b and 13a.
- `setSymbolValue`, used in chunks 11b and 13a.
- `showTokenValue`, used in chunks 2 and 13a.

Uses `getSymbolName` 7a, `showOpKind`, `SymTabEntry` 4, and `TokenValue` 13a.

17 ⟨*OpKind ADT implementation* 17⟩≡ (16)

```
char *showOpKind(const OpKind kind) {
        switch (kind) {
                case EqOP:      return "EqOP";
                case NotEqOP:   return "NotEqOP";
                case LessOP:    return "LessOP";
                case LessEqOP:  return "LessEqOP";
                case GrtrOP:    return "GrtrOP";
                case GrtrEqOP:  return "GrtrEqOP";
                case OrOP:      return "OrOP";
                case PlusOP:    return "PlusOP";
                case MinusOP:   return "MinusOP";
                case AndOP:     return "AndOP";
                case DivOP:     return "DivOP";
                case ModOP:     return "ModOP";
                case TimesOP:   return "TimesOP";
                case DivideOP:  return "DivideOP";
        }
}
```

Uses `OpKind` 13b.

5 Indices

5.1 Code Chunks

5.2 Identifiers