

## B.43 Specification for program “nametbl” (Document ES9)

### Name

nametbl – Functions for a symbol table

### Usage

**nametbl** input-file

### Description

**nametbl** reads commands from a file and processes them in order to test a few functions. Considered together, the functions implement a symbol table for a certain computer language. The only scope for all symbols is “global.” No check of type semantics is done.

The symbol table stores the following information for each symbol:

Symbol name.

Object type of the symbol: {OT\_NO\_INF, SYSTEM, RESOURCE}.

Resource type of the symbol: {RT\_NO\_INF, RT\_SYSTEM, FUNCTION, DATA}.

### Options

None.

### Input Data

Only the following commands are allowed in the input files. Each command must start on a new line. Lower and upper case is relevant in the symbols and in the object and resource identifiers. The separator between commands and arguments is a space; this means that spaces may not appear within a symbol.

- `ins <Symbol>`

Inserts the symbol in the table. The object type of the symbol receives the value “OT\_NO\_INF” and the resource type receives the value “RT\_NO\_INF.” If the symbol already exists, an error message is printed.

- `tot <Symbol> <ObjTyp>`

Enters the object type <ObjTyp> for the symbol <Symbol>, where <ObjTyp> must be either “SYSTEM” or “RESOURCE”; the value “OT\_NO\_INF” is not allowed. The symbol must be entered in the table, otherwise a corresponding error message is printed. The previous value for the object type is overwritten and lost.

- `trt <Symbol> <ResTyp>`

Enters the resource type <ResTyp> for the symbol <Symbol>, where <ObjTyp> must be either “RT\_SYSTEM”, “FUNCTION”, oder “DATA”; the value “RT\_NO\_INF” is not allowed. (The value “RT\_SYSTEM” is used in the case where the object type of the symbol has the value “SYSTEM” and information concerning the resource type is therefore not required.) The symbol must be entered in the table, otherwise a corresponding error message is printed. The previous value for the resource type is overwritten and lost.

- `sch <Symbol>`  
Searches for the symbol `<Symbol>` and prints out the corresponding information. If the symbol is not found in the table, a corresponding error message is printed.
- `prt`  
Prints out the count of entries in the symbol table and the complete contents.

### Example

```
% cat eingabe
ins qwe
ins rty
tot qwe SYSTEM
prt
% nametbl eingabe
Eingabedatei 'eingabe' wird bearbeitet.

Die Zeile 'ins qwe' wird ausgewertet:

Die Zeile 'ins rty' wird ausgewertet:

Die Zeile 'tot qwe SYSTEM' wird ausgewertet:

Die Zeile 'prt' wird ausgewertet:
Die Tabelle hat die folgenden 2 Eintraege:
Name      : qwe
oType     : SYSTEM
rType     : RT_NO_INF
-----
Name      : rty
oType     : OT_NO_INF
rType     : RT_NO_INF
-----
Ende der Eingabedatei 'eingabe'.
```

### Authors

Goldmann, Klemke, Knecht, Lott.

### Bugs

The functionality for reading files is only for test purposes and is therefore not especially fault-tolerant. For example, one can assume that incorrectly spelled commands, commands with missing arguments, or commands with too many arguments will not be handled especially carefully.

## B.44 Supplement “Code reading and nametbl” (Document E19)

This is the supplemental sheet for the program “nametbl” and technique “code reading.”

### Brief description of the library functions used

- `assert(expression)`  
Macro that expects the expression to be true at the time the program is executed and in that case does nothing. Otherwise an error message with the expression is printed and the program is ended.
- `char* malloc(unsigned size)`  
Returns a pointer to a block of memory that is at least as large as `size` bytes. In case of failure, `NULL` is returned.
- `int free(ptr)`  
Gives a previously allocated block back to the system. `ptr` must have been allocated using `malloc` or a similar function. If successful, 1 is returned; otherwise 0 is returned.
- `int strcmp(s1, s2)`  
Compares two strings. Returns a value greater than, equal to, or less than 0 depending on whether `s1` is lexicographically (i.e., ASCII value) greater than, equal, or less than `s2`.
- `char *strdup(s1)`  
Duplicates the string `s1`. Allocates memory for this purpose using `malloc()` and returns a pointer to the duplicate if successful, otherwise `NULL`.
- `int sscanf(s, format, pointer)`  
Reads characters from string `s`, converts them according to the specification string `format`, and writes the result into the variables pointed to by the pointers. This is the reverse function of `printf`; the format specification strings are identical.
- `char *fgets(s, n, stream)`  
Reads characters from the stream `stream` and writes them in the string `s`, until either `n-1` characters have been read or one of `NEWLINE` or `EOF` are seen. At `EOF` the value `NULL` is returned, otherwise `s`.
- `char *tsearch(char *key, char **rootp, int (*compar)())`
- `char *tfind(char *key, char **rootp, int (*compar)())`
- `void twalk(char *root, void (*action)())`  
These functions implement a binary search tree. All required comparisons are performed with the function `compar`, which the user must provide for use by the tree functions. The function `compar` is made known to the tree functions with the help of a function pointer. `compar` is invoked with two arguments, which point to the elements to be compared. The return value must be less than, equal to, or larger than 0, depending on whether the first element is less than, equal to, or larger than the second (analogous to `strcmp`).

`tsearch` can both build and search a tree. If an element is found which has the same key as the element to which key points, a pointer is returned. This pointer points to a pointer which

points to the element that was found. Otherwise the element pointed to by `key` is inserted into the tree and a pointer-to-a-pointer to the new element is returned. An illustration about the double indirection:

```

returned   +---+           +-----+
----->|   |----->|  node  |
pointer    +---+ pointer  +-----+
                        /   \

```

Only pointers are copied; i.e., the user is responsible for reserving memory for the tree elements (e.g., with `malloc`). `rootp` points to a variable and this variable points to the root of the tree. To produce the first node in the tree (the root), `NULL` must be stored in the variable, thereafter the variable receives a pointer to the root of the tree.

`tfind` is identical to `tsearch`, with the exception that if the element is not found, `NULL` is returned and no new node is added to the tree.

`twalk` traverses a tree or a subtree, because any node can be used for the `root` argument. Take note that only a pointer to `root` can be given, not a pointer-to-a-pointer as used by `tsearch` and `tfind`. `action` is the function which is invoked for each node in the tree. This function must also be provided by the user. It receives three arguments: a pointer to the tree element, information about when the element is visited, and the depth of the tree element (`root = 0`). The visit information is an enumerated type `typedef enum {preorder, postorder, endorder, leaf} VISIT`; where `preorder`, `postorder`, and `endorder` represent the first, second, and third visit during a depth-first, left-to-right traversal of the tree. `leaf` refers to the leaf of a tree.

## Reminder

Don't produce any abstractions for the test-scaffold functions.

## B.45 Supplement “Functional testing and nametbl” (Document E29)

This is the supplemental sheet for the program “nametbl” and technique “functional testing.”

### Necessary inputs

- Which documents belong to this exercise?
  1. Document ES9, the specification of the component
  2. Document EQ9, the source code of the component, which you will receive after you have written test cases and diagnosed failures.

- How do I fetch the files which I need?

Do the following:

1. First create a new directory for this exercise with the “mkdir” command.

```
mkdir ft-nametbl
```

2. Then change to the new directory with the “cd” command.

```
cd ft-nametbl
```

3. Finally, enter the following command:

```
tar xf ~prakt00/Exercise5/ft-nametbl.tar
```

- What should I have?

The following files must be available:

```
Makefile      nametbl      run-suite    test-dir
```

### Creating equivalence classes and test cases

The test scaffolding offers commands and their parameters to simplify addressing the program’s functions. Do not create equivalence classes for syntactically incorrect commands! That would only test the test scaffolding. Examples of unnecessary test cases:

```
delete all
```

Command not defined.

```
ins
```

Argument is missing.

### Reminder

Make sure that your test cases generate output so that you can detect failures!

## Document E29, page 2: To be given out with the code!

### Brief description of the library functions used

- `assert(expression)`

Macro that expects the expression to be true at the time the program is executed and in that case does nothing. Otherwise an error message with the expression is printed and the program is ended.
- `char* malloc(unsigned size)`

Returns a pointer to a block of memory that is at least as large as `size` bytes. In case of failure, `NULL` is returned.
- `int free(ptr)`

Gives a previously allocated block back to the system. `ptr` must have been allocated using `malloc` or a similar function. If successful, 1 is returned; otherwise 0 is returned.
- `int strcmp(s1, s2)`

Compares two strings. Returns a value greater than, equal to, or less than 0 depending on whether `s1` is lexicographically (i.e., ASCII value) greater than, equal, or less than `s2`.
- `char *strdup(s1)`

Duplicates the string `s1`. Allocates memory for this purpose using `malloc()` and returns a pointer to the duplicate if successful, otherwise `NULL`.
- `int sscanf(s, format, pointer)`

Reads characters from string `s`, converts them according to the specification string `format`, and writes the result into the variables pointed to by the pointers. This is the reverse function of `printf`; the format specification strings are identical.
- `char *fgets(s, n, stream)`

Reads characters from the stream `stream` and writes them in the string `s`, until either `n-1` characters have been read or one of `NEWLINE` or `EOF` are seen. At `EOF` the value `NULL` is returned, otherwise `s`.
- `char *tsearch(char *key, char **rootp, int (*compar)())`
- `char *tfind(char *key, char **rootp, int (*compar)())`
- `void twalk(char *root, void (*action)())`

These functions implement a binary search tree. All required comparisons are performed with the function `compar`, which the user must provide for use by the tree functions. The function `compar` is made known to the tree functions with the help of a function pointer. `compar` is invoked with two arguments, which point to the elements to be compared. The return value must be less than, equal to, or larger than 0, depending on whether the first element is less than, equal to, or larger than the second (analogous to `strcmp`).

`tsearch` can both build and search a tree. If an element is found which has the same key as the element to which `key` points, a pointer is returned. This pointer points to a pointer which points to the element that was found. Otherwise the element pointed to by `key` is inserted into

the tree and a pointer-to-a-pointer to the new element is returned. An illustration about the double indirection:

```

returned  +---+           +-----+
----->| |----->| node |
pointer  +---+ pointer +-----+
                        /  \

```

Only pointers are copied; i.e., the user is responsible for reserving memory for the tree elements (e.g., with `malloc`). `rootp` points to a variable and this variable points to the root of the tree. To produce the first node in the tree (the root), `NULL` must be stored in the variable, thereafter the variable receives a pointer to the root of the tree.

`tfind` is identical to `tsearch`, with the exception that if the element is not found, `NULL` is returned and no new node is added to the tree.

`twalk` traverses a tree or a subtree, because any node can be used for the `root` argument. Take note that only a pointer to `root` can be given, not a pointer-to-a-pointer as used by `tsearch` and `tfind`. `action` is the function which is invoked for each node in the tree. This function must also be provided by the user. It receives three arguments: a pointer to the tree element, information about when the element is visited, and the depth of the tree element (`root = 0`). The visit information is an enumerated type `typedef enum {preorder, postorder, endorder, leaf } VISIT;` where `preorder`, `postorder`, and `endorder` represent the first, second, and third visit during a depth-first, left-to-right traversal of the tree. `leaf` refers to the leaf of a tree.

## B.46 Supplement “Structural testing and nametbl” (Document E39)

This is the supplemental sheet for the program “nametbl” and technique “structural testing.”

This exercise primarily tests several C functions. To save you time and effort in this exercise, a test scaffold was created. The driver functions required for this scaffold are in the precompiled file “driver.o” and are bound in at link time. The drivers take care of function such as testing for and opening input files, allocating necessary memory, etc. See also the notes about one visible driver function that appear below.

### Necessary inputs

- Which documents belong to this exercise?
  1. Document ES9, the specification of the component, which you will receive after you have created test cases and attempted to reach 100% coverage.
  2. Document EQ9, the source code of the component

- How do I fetch the files which I need?

Do the following:

1. First create a new directory for this exercise with the “mkdir” command.

```
mkdir st-nametbl
```

2. Then change to the new directory with the “cd” command.

```
cd st-nametbl
```

3. Finally, enter the following command:

```
tar xf ~prakt00/Exercise5/st-nametbl.tar
```

---

*path*

---

- What should I have?

The following files must be available:

```
Makefile      gct-map      nametbl      run-suite    test-dir
```

### Writing test cases

All of the program’s functions are fundamentally tested via the invocation

```
nametbl <input file>
```

A parameter file holds only the name of the input file. This is somewhat awkward, but necessary to allow applying the test driver consistently. The format of the input file is as follows:

```
command1 parameter1 parameter2
command2 parameterX
...
```

Which commands are allowed in the input and which functions are invoked by those commands can be seen in the function `fuehre_kommandos_aus` in the file `nametbl.c`. Because this function belongs to the test scaffold and will not be part of the eventual application, it is not especially robust. You can assume that the function, although sensitive, is free of faults. One activity of the driver from `driver.o` is to open the first file on the command line and to invoke the function `fuehre_kommandos_aus` with a pointer to the opened file. Don't write any test cases to cover the test scaffold!

## Reminder

Make sure that your test cases generate output so that you can detect failures!

## Brief description of the library functions used

- `assert(expression)`

Macro that expects the expression to be true at the time the program is executed and in that case does nothing. Otherwise an error message with the expression is printed and the program is ended.
- `char* malloc(unsigned size)`

Returns a pointer to a block of memory that is at least as large as `size` bytes. In case of failure, `NULL` is returned.
- `int free(ptr)`

Gives a previously allocated block back to the system. `ptr` must have been allocated using `malloc` or a similar function. If successful, `1` is returned; otherwise `0` is returned.
- `int strcmp(s1, s2)`

Compares two strings. Returns a value greater than, equal to, or less than `0` depending on whether `s1` is lexicographically (i.e., ASCII value) greater than, equal, or less than `s2`.
- `char *strdup(s1)`

Duplicates the string `s1`. Allocates memory for this purpose using `malloc()` and returns a pointer to the duplicate if successful, otherwise `NULL`.
- `int sscanf(s, format, pointer)`

Reads characters from string `s`, converts them according to the specification string `format`, and writes the result into the variables pointed to by the pointers. This is the reverse function of `printf`; the format specification strings are identical.
- `char *fgets(s, n, stream)`

Reads characters from the stream `stream` and writes them in the string `s`, until either `n-1` characters have been read or one of `NEWLINE` or `EOF` are seen. At `EOF` the value `NULL` is returned, otherwise `s`.
- `char *tsearch(char *key, char **rootp, int (*compar)())`
- `char *tfind(char *key, char **rootp, int (*compar)())`

- `void twalk(char *root, void (*action)())`

These functions implement a binary search tree. All required comparisons are performed with the function `compar`, which the user must provide for use by the tree functions. The function `compar` is made known to the tree functions with the help of a function pointer. `compar` is invoked with two arguments, which point to the elements to be compared. The return value must be less than, equal to, or larger than 0, depending on whether the first element is less than, equal to, or larger than the second (analogous to `strcmp`).

`tsearch` can both build and search a tree. If an element is found which has the same key as the element to which `key` points, a pointer is returned. This pointer points to a pointer which points to the element that was found. Otherwise the element pointed to by `key` is inserted into the tree and a pointer-to-a-pointer to the new element is returned. An illustration about the double indirection:

```

returned   +--+           +-----+
----->|   |----->|  node  |
pointer    +--+ pointer +-----+
                        /  \

```

Only pointers are copied; i.e., the user is responsible for reserving memory for the tree elements (e.g., with `malloc`). `rootp` points to a variable and this variable points to the root of the tree. To produce the first node in the tree (the root), `NULL` must be stored in the variable, thereafter the variable receives a pointer to the root of the tree.

`tfind` is identical to `tsearch`, with the exception that if the element is not found, `NULL` is returned and no new node is added to the tree.

`twalk` traverses a tree or a subtree, because any node can be used for the `root` argument. Take note that only a pointer to `root` can be given, not a pointer-to-a-pointer as used by `tsearch` and `tfind`. `action` is the function which is invoked for each node in the tree. This function must also be provided by the user. It receives three arguments: a pointer to the tree element, information about when the element is visited, and the depth of the tree element (`root = 0`). The visit information is an enumerated type `typedef enum {preorder, postorder, endorder, leaf} VISIT`; where `preorder`, `postorder`, and `endorder` represent the first, second, and third visit during a depth-first, left-to-right traversal of the tree. `leaf` refers to the leaf of a tree.