CST 8152 Compilers - Assignment #3

Due Date: prior to or on 24 November 2016
Earnings: 6% of your final grade plus up to 2 % bonus (see st_sort() function)
Purpose: Implementing and Incorporating a Symbol Table
You are to implement and incorporate a symbol table component in your PLATYPUS compiler. The symbol table component consists of two parts: a Symbol Table Manager (hereafter abbreviated STM) and a Symbol Table Database (hereafter abbreviated STDB). The STM provides utilities (service functions) for manipulation of the STBD. The STDB is a repository for VID attributes. Each variable identifier is associated with one record in the database. Five VID attributes will be defined in the symbol table: variable name, type, initial value, line number, and one reserved attribute. The scanner can identify and store in the symbol table only four of those attributes: the lexeme for the VID, the line number of the line where the corresponding VID appears for the first time in the source program, the default type, and initial value of the variable. The last two might be changed later by the parser and the semantic analyzer. The fifth attribute (*reserved*) is reserved for future use and **must not be used** in this implementation.

Complete the following tasks:

Task 1: Declaring the Data Structures for the Symbol Table Database

You are to use linear structures to implement the STDB. The STDB consists of three interconnected elements:

- ***Database Descriptor***. It links the database elements and stores data about different parameters of the database. It is to be implemented as a non-dynamic structure described by the **S**ymbol **T**able **D**escriptor (hereafter abbreviated **STD**) structure declaration below;
- ***Database Record Table***. It stores the VID records. Each VID has one record (entry). The record structure (contents) is described by the **S**ymbol **T**able **V**ariable **R**ecord (hereafter abbreviated **STVR)** structure declaration below. The Database Record Table is to be implemented as a dynamically allocated array of VID records (STVR structures).
- ***Lexeme Storage***. It stores the variable names. It is to be implemented as a dynamic self-incrementing character array (hereafter abbreviated **CA**). The previously developed buffer utility is to be used to implement the lexeme storage.

You are to use the following declarations for the database implementation:

```
typedef union InitialValue {
  int int_val;   /* integer variable initial value */
  float fpl_val; /* floating-point variable initial value */
  int str_offset; /* string variable initial value (offset) */
} InitialValue;

typedef struct SymbolTableVidRecord {
  unsigned short status_field;  /* variable record status field*/
  char * plex;          /*  pointer to lexeme (VID name) in CA */
  int o_line;           /* line of first occurrence */
  InitialValue i_value; /* variable initial value */
  size_t reserved;      /*reserved for future use*/
}STVR;

typedef struct SymbolTableDescriptor {
  STVR *pstvr;         /* pointer to array of STVR */
  int st_size;        /* size in number of STVR elements */
  int st_offset;      /*offset in number of STVR elements */
  Buffer *plsBD; /* pointer to the lexeme storage buffer descriptor */
} STD;
```

Where:

plex is a pointer to the VID lexeme inserted by the STM into the CA.
o_line is the line number of the line where the corresponding VID occurs for the first time in the source program.
status_field is a field containing different flags and indicators. In cases when storage space must be as small as possible, the common approach is to pack several data items into single variable; one common use is a set of single-bit or multiple-bit flags or indicators in applications like compiler symbol tables. The flags are usually manipulated through different bit-wise operations using a set of "masks." Alternative technique is to use *bit-fields.* Using bit-fields allows individual fields to be manipulated in the same way as structure members are manipulated. Since almost everything about bit-fields is implementation dependant, this approach should be avoided if the portability is a concern. In this implementation, you are to use bit-wise operations and masks (see *bitmask.c* example).

Each flag or indicator uses one or more bits of the status field. The flags and the indicators describe the status of the variable record. For example, the status field can contain an indication that the variable is out of scope, or that the record (variable entry) is deleted. In this PLATYPUS compiler implementation the status field has the following structure:

Bit	**MSB** 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0 **LSB**
Contents	1	1	1	1	1	1	1	1	1	1	1	1	1	x	x	x
Description	reserved for future use must be set to 1 and stay 1 all the time in this implementation													data type indicator		update flag

The LSB bit (bit 0) is a single-bit *update flag*. It is used to indicate that the default data type of a variable has been changed. This will allow the parser and the semantic analyzer to detect errors. The default value of the *update flag* is **zero**.

Bits 2 and 1 are used as a variable *data type indicator*. The combination **01** indicates a floating-point type, the combination **10** indicates an integer type, and the combination **11** indicates a string type. The default value of the *data type indicator* is **00**.
The rest of the bits are reserved for further use and must be set by default to **1**.

pstvr is a pointer to a dynamically allocated array of STVR.
st_size is the size of the array of STVR (in number of elements).
st_offset is the offset from the beginning of the array of STVR to the first available empty element in the array of STVR.
plsBD is a pointer to a Buffer Descriptor structure. The character array (CA) of this buffer is used to store the variable names (lexemes). You must use your ***buffer*** utility functions (Assignment 1) to create and manipulate this storage area.
For the first record in the symbol table **plex = plsBD->cb_head**. For the second, third and so on, you must calculate the pointer **plex** using **addc_offset**.
Hint: It is similar to what you have done with the string literals in the scanner, but in this assignment the big difference is that, you are to use *pointers* instead of offsets to indicate the location of the variable name in the lexeme buffer character array. Since you are not allowed to manipulate the data members of the buffer structure directly, you must use an appropriate buffer function that can return a pointer to a location indicated by addc_offset.

Before proceeding with Task 2, draw a structure diagram of the STDB depicting clearly the date structures involved and the links among their elements.

Task 2: Implementing the Symbol Table Manager (STM)

Your STM must provide the following services or functions:

`STD st_create(int st_size)`

This function creates a new (empty) symbol table. It declares a **local variable** of type STD (Symbol Table Descriptor). Then it allocates dynamic memory for an array of STVR with ***st_size*** number of elements. Next, it creates a *self- incrementing* buffer using the corresponding ***buffer*** function and initializes the ***plsBD*** pointer. It initializes the ***st_offset*** to zero. The function returns a STD structure. If the operation has been successful, it sets the STD ***st_size*** to ***st_size***; otherwise it sets the STD ***st_size*** to 0. *Note:* STD is not dynamically allocated.

`int st_install(STD sym_table, char *lexeme, char type, int line)`

This function installs a new entry (VID record) in the symbol table.
First, It calls the ***st_lookup()*** function to search for the lexeme (variable name) in the symbol table.

If the lexeme is not there, it installs the new entry at the current ***st_offset***. The function sets ***plex*** and ***o_line*** to their corresponding values, and the ***status_field*** to its default value (see above). Then it sets the *data type indicator* to a value corresponding to the type of the variable specified by the formal parameter ***type***. The value of the ***type*** parameter can only be ***I*** for integer type, ***F*** for floating-point type, and ***S*** for string type. If the variable is of type string the function sets the ***update flag*** to 1. To set the ***data type indicator*** and the ***update flag*** you **must** use ***bitwise*** operations (see *bitmask.c* example). The function sets the ***i_value*** to zero for integer and floating-point variables, and to –1 for string variables. The function returns the current offset of that entry from the beginning of the array of STVR (the array pointed by ***pstvr***). Before returning the function must increment the ***st_offset*** of the "global" ***sym_table*** by 1.

If it finds the lexeme in the symbol table, it returns the corresponding offset. If the symbol table is full, it returns –1. The lexemes are entered in the symbol table as strings (C-type strings). **Big Gargantuan Warning:** This function can create dangling pointers. Take all necessary steps to prevent that from happening.

`int st_lookup(STD sym_table, char *lexeme)`

This function searches for a lexeme (variable name) in the symbol table. The search is performed **backward** from the last entry to the beginning of the array of STVR. If it has been found, it returns the offset of the entry from the beginning of the array of STVR. Otherwise, it returns –1.

`int st_update_type(STD sym_table,int vid_offset,char v_type)`

The function updates the ***data type indicator*** in the variable entry (STVR) specified by

vid_offset. The type of the variable is specified by the argument ***v_type***: **F** for floating-point type and **I** for integer type. String type can not be updated. First, it checks the ***update flag*** (LSB) of the ***status_field*** of the entry. If it is equal to 1, the type has been already updated and the function returns –1. Otherwise, the function updates the *data type indicator* of the ***status_field***, sets the LSB of the ***status_field*** to **1**, and returns ***vid_offset***. You **must** use bit-wise operations and masks to perform the specified set operations on the status field.

```
int st_update_value(STD sym_table, int vid_offset,
                    InitialValue i_value)
```

The function updates the ***i_value*** of the variable specified by ***vid_offset***. On success it returns ***vid_offset***. On failure it returns –1.

```
char st_get_type (STD sym_table, int vid_offset)
```

The function returns the type of the variable specified by ***vid_offset***. It returns **F** for floating-point type, **I** for integer type, or **S** for string type. On failure it returns –1. You **must** use bit-wise operations and masks to determine the type.

```
void st_destroy(STD sym_table)
```

This function frees the memory occupied by the symbol table dynamic areas and sets ***st_size*** to 0.

```
int st_print(STD sym_table)
```

This function prints the contents of the symbol table to the standard output (screen) in the following format (see the test files):

```
Symbol Table
_____________

Line Number  Variable Identifier
...          ...
```

The function returns the number of entry printed or –1 on failure.

```
static void st_setsize(void)
```

This “internal” function sets ***st_size*** to 0. You must use this function when you want to set ***st_size*** to 0 in some function which does not have access to the global *sym_table* variable.

```
static void st_incoffset(void)
```

This “internal” function increments ***st_offset*** by 1. You must use this function when you want to increment ***st_offset*** in some function which does not have access to the global *sym_table* variable.

```
int st_store(STD sym_table)
```

This function stores the symbol table into a file named ***$stable.ste***. This file is a text file. If the file already exists in the current directory, the function overwrites it. The function uses ***fprintf()*** to write to the file. First, it writes ***st_size***. Then for each symbol table entry, it writes the ***status_field*** (in hex format), the length of the lexeme, the lexeme, the line number, and the initial value. To output the appropriate initial value you **must** use the ***st_get_type()*** function. The data items in the file are separated with a space (see *fileio.c* example). On success the function prints a message "Symbol Table stored" and returns the number of records stored; it returns –1 on failure.

```
int st_sort(STD sym_table, char s_order)
```

The required implementation of this function is simple. The body of the function contains only one statement: returns 0; the function does not sort the symbol table.

Bonus Tasks – Implementation of the Symbol Table sort function

Bonus 1 (1%)

Implement the function so that it sorts the array of symbol table entries (variable records) by variable name (lexeme) in ascending (s_order = 'A') or descending (s_order = 'D') order. On success it returns 1; otherwise it returns –1.
You **must** use the C standard library ***qsort()*** function the sort the array in order to get the bonus mark. Your test output **must** also match the provided test output **ass3r_stsa.out**

Bonus 2 (1%)
After sorting, reorganize the lexemes in the symbol table lexeme storage buffer so that they are in the same order as the VID records in the sorted table, that is, the first record lexeme pointer (*plex*) must point to the first lexeme, the second to the second and so on. The receive credit for the implementation of Bonus 2 the bonus testing must be reflected in your test plan and a test output must be provided proving that the bonus works.

Note: All symbol table functions **must** check for valid symbol table before performing their tasks**.** If the symbol table is not valid, the function must return an appropriate indication.

Task 3: Incorporating the Symbol Table

All constant definitions, function declarations (prototypes) and declarations (except for the ***sym_table*** global variable declaration and the static function(s) declarations) which are required for the implementation of the symbol table must be located in a file called ***stable.h***. The file must not allow duplicate inclusions by the preprocessor. The ***sym_table*** *global variable declaration*, the static function(s) declarations, and all function definitions must be located in a file named ***stable.c***. You are not allowed to change the names of the variables, the names of the functions, and the names of the

function parameters. All they must be declared/defined as specified in the assignment. Any change will be considered an error of **GOOGOLY** proportion. You can add more functions (they should be static) if necessary. If you decide to add a function (and you definitely need to add some for the bonus implementation), you must explain and justify your decision in the header comments of the function.

To incorporate the symbol table into your compiler project you must modify the following PLATYPUS components:

The symbol table descriptor STD is shared among different compiler components (.c files) as **a global variable** named ***sym_table***. It is defined in the main program and must be declared in the other compilation units if they need it.

token.h
Remove the ***char vid_lex[]*** member from the union declaration.
Add a new member ***int vid_offset*** to the union ***TokenAttribute***.
Modify ***scanner.c*** VID accepting functions accordingly.

scanner.c
Modify your VID accepting functions. They must keep the **same functionality** as specified in Assignment 2, but instead of storing the variable name (lexeme) in the token attribute it installs it in the symbol table along with other variable attributes. When SVID is recognized and formed properly, the function calls the ***st_install ()*** function with a ***type*** parameter ***S*** and then sets the token attribute ***vid_offset*** to the offset returned by the function. When AVID is recognized and formed properly, the function calls the ***st_install ()*** function with a ***type*** parameter ***F*** or ***I*** depending on the default type of the arithmetic variable identifier (see the PLATYPUS informal language specification), and then sets the token attribute ***vid_offset*** to the offset returned by the function.

Now in the presence of the symbol table, the VID token attribute will be the location of the variable record in the symbol table. If the symbol table is full, the accepting function must print the error message

```
Error: The Symbol Table is full - install failed.
```

and must call the ***st_store()*** function and terminate the program with a call to ***exit()***.
The scanner must not manipulate directly any of the symbol table internal parameters (like ***st_size*** and ***st_offset***) stored in the STD structure (*sym_table*). A symbol table parameter can only be manipulated via the utility functions provided by the Symbol Table Manager (STM).
The symbol table component must be implemented as specified. There are some “issues” in the design of the symbol table. The design flaws are intentional. One of the purposes of this assignment is to illustrate some important concepts in the design of the C language: variable scope and variable "hiding"; function scope (function hiding); dangling pointers.
Try to identify the issues, comment them in your source code, and find workarounds without violating the specifications. Watch for ***memory leaks*** and ***dangling pointers***. Both are possible in this assignment. Do not change the specifications by any means. Changes to the specification will be considered to be a major mistake (**GOOGOLBGE**).

Task 4: Testing the Symbol Table

To test your program use the program ***platy_tt.c***, the input files ***ass3r.pls*** and ***ass3m.pls***. Do not alter the main program source code or the input files.

The main program ***platy_tt.c*** takes four arguments from the command line: the name of a source file (for example, ***ass3r.pls***), an optional symbol table size switch (***-stz***) followed by the an integer number (for example, **-stz 10**), and/or an optional sort switch (**-sts:A** or **–sts:D**). The switch option **-stz** specifies the maximum number of variables that can be stored in the symbol table (the size of STVR). If the switch is missing, the default size of the symbol table is 100 variables. The symbol table is defined as a global variable ***sym_table*** in the main program. The main program reads the arguments specified at the command line, allocates an input buffer, and creates a symbol table with the specified or default size. Then the program opens up the source file, loads the buffer with data from the file, calls the scanner in a loop, and finally prints the symbol table. If the sort switch **–sts** has been specified, it sorts the table in the specified order and prints it again.

IMPORTANT NOTE:
You are still allowed to work on this assignment in the same team you have worked on the scanner. You are not allowed to ***change*** the team but you can break the team and work alone. If you have worked alone, you are not allowed to work on this assignment in a team. Each team must submit one assignment only. The envelope label and the cover page must contain information about both members as required by the Assignment Submission Standard. Additionally, on a separate page (undercover page) containing the name and the student ID# of the team member, each of the team members must give a brief description of the work done by her/him on this assignment (including the names of the functions written by the member). The page must be signed by the members. Each and every member must be involved in some coding and testing. Each member must code and test at least **four (4)** functions and the name of the member must be indicated in the function header. The rest of the functions may have two authors. Be aware that all of the conditions above **must be met** in order to have your assignment accepted and marked.

What to Submit:

Hand in on paper:

1. Fully documented source listings of your ***stable.h*** and ***stable.c*** files.
2. Fully documented source listings of the modified ***token.h*** file.
3. Fully documented listing of your modified **VID accepting functions** in **scanner.c**
4. The test results from testing your symbol table component with the provided main program (***platy_tt.c***) and test files (***ass3r.pls*** and ***ass3m.pls***).
5. A structure diagram of the STDB depicting clearly the date structures involved and the links among their elements.
6. The marking sheet for the assignment with your name filled in.

Digital Drop Box Submission

Compress into a **zip** file the following files: all **.h** files, all **.c** files, all **.pls** files, and your output test files. Include your additional input/output test files if you have any (see bonus task). Upload the **zip** file on Blackboard. The file must be submitted prior or on the due date as indicated in the assignment. The name of the file must be Your Last Name followed by the last three digits of your student number followed by cA3. For example: Brown345_cA3.zip. If your last name is long, you can truncate it to the first 5 letters. **Teams** must submit one .zip file only. The name of the file must contain the names of both members e.g. Brown 45_Fox123_cA3.zip.

Make sure all printed materials (and eventually the disk) are placed into an unsealed envelope and are deposited into the assignment box prior to the end of the due date. If you are late, you must submit the assignment envelope directly to the professor. The submission must follow the course submission standards (**CST8152_ASSAMG.pdf)**. Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

Enjoy the assignment. And do not forget that:

"Education is when you read the fine print. Experience is what you get if you don't." Pete Seeger

And remember to remember:

"Everything is and is not, for everything is fluid, is constantly changing, constantly coming into being and passing away." Heraclitus

CST8152, 27 October 2016, S^R