

COMP 3290 Introduction to Compiler Construction

Assignment 3 Fall 2008

In this assignment you will program the symbol-table management routines for the C3290 interpreter, and use these routines in the parser. Once again, you will be given much of the implementation in the free code.

The symbol table ADT must support scoping. It will also be used to hold automatic variables created in activation records (stack frames). (The other information in a stack frame, particularly return address and return value, will be implemented using a different data structure.) Our symbol table will be implemented as a linked list. A linked list is simple to implement, has fast insertion but slow search times and good support for scope rules. My implementation does not use a dummy node, so special cases must be considered.

Symbol Table Entries

The symbol table must be able to hold entries of various types: integers, doubles, strings, built-in functions, user-defined functions and arrays. These entries have links to other entries so that they may be added to the linked list. In order to accommodate these various types we will use a class hierarchy, which has already been defined for you in Symbol.java.

The top of the symbol table entry hierarchy is the abstract ST_ENTRY class. It defines the information needed by all entries, as described below.

| | |
|-------|--|
| Name | The symbol name stored as a String. |
| Scope | The scope level of the symbol, an integer ≥ 0 . |
| Link | A link to the next entry in the list. |

The abstract class ST_Literal is the parent class of the literal types (STT_Int, STT_Double and STT_String). It defines the Boolean value isConst, which is set to true for constants, and false for variables.

The actual types for each symbol are represented by concrete subclasses of ST_ENTRY and ST_Literal, shown in Table 1 below. Each of these classes defines an appropriate field to contain the value for that type of symbol table entry.

| <i>Type</i> | <i>Value</i> |
|---------------------|--|
| STT_Int | An integer. |
| STT_Double | A double. |
| STT_String | A String. |
| STT_Array | A reference to an array of ST_ENTRY. |
| STT_BuiltinFunction | A function code (see Table 2). |
| STT_UserFunction | A reference to the starting line of the function and a list of the parameters. |

Table 1: Symbol Table Entry types

Function Codes

Table 2 below lists the built-in functions, number of arguments, and the function codes

More details on the behaviour of these functions will be given in a later assignment. The function *pi* could be handled as a predefined constant but making it a function prevents accidental assignments to that symbol.

COMP 3290 Introduction to Compiler Construction
Assignment 3 Fall 2008

| <i>Name</i> | <i>Args</i> | <i>Code</i> | <i>Meaning</i> |
|-------------|-------------|-------------|---|
| sign | 1 | FUNC_SIGN | Sign of argument |
| abs | 1 | FUNC_ABS | Absolute value |
| sin | 1 | FUNC_SIN | Trigonometric sin function |
| cos | 1 | FUNC_COS | Trigonometric cos function |
| tan | 1 | FUNC_TAN | Trigonometric tan function |
| asin | 1 | FUNC_ASIN | Arcsin |
| acos | 1 | FUNC_ACOS | Arccos |
| atan | 1 | FUNC_ATAN | Arctan |
| ln | 1 | FUNC_LN | Natural logarithm |
| exp | 1 | FUNC_EXP | Exponential |
| sqrt | 1 | FUNC_SQRT | Square root |
| int | 1 | FUNC_INT | Truncate a real to an integer |
| pi | 0 | FUNC_PI | The constant π |
| rand | 0 | FUNC_RAND | Random number in the range $0 \leq x < 1$ |

Table 2: C329 Built-in Functions

Symbol-Table Management Routines

The following symbol-table management routines will be coded:

Scope Support Functions

void **st_push_scope** () – Increment the current scope value stored in the variable *Global.current_scope*.

void **st_pop_scope** () – Delete all the symbols belonging to the current scope. Decrement the current scope.

Table Access Functions

ST_ENTRY **st_access** (String *name*) - Look for the symbol indicated by the parameter *name*. If it exists in the symbol table, return a pointer to its entry. If it does not exist, return *null*. This function is used to check if a symbol has been defined or not and to return it if it has been defined. **This function has been provided for you.**

void **st_insert** (ST_ENTRY *item*) – Insert the symbol table entry *item* into the symbol table. The item is inserted before any items of equal (or lower) scope, but after items of greater scope, so the list must be traversed to find the correct insertion position.

void **st_delete** (String *name*) – Delete the first occurrence of a symbol table entry with the given *name* (i.e. the entry with the highest scope). If there is no such symbol, do nothing.

Execution Support Functions

void **st_init** () - This routine will be called before every run of the loaded C3290 program. Initialize the variable *Global.current_scope* to 0. Initialize the pointer *Global.main_line* to *null*. (This is used both to check that the program has a main function and to begin executing there.) Clear the symbol table and initialize it with the predefined symbols, i.e. the names of the built-in functions. The routine *st_store_builtin_function* can be called to insert each built-in function name.

void **st_store_builtin_function** (String *Name*, int *Code*) - The parameter *Name* is a string giving the name of a built-in function. The parameter *Code* is the function code. The number of arguments need not be stored in the symbol table since that will be known from the context in which the function is used. **This function has been provided for you.**

COMP 3290 Introduction to Compiler Construction

Assignment 3 Fall 2008

void **st_store_user_function** (String *symbol*, line_rec *Line*) - The parameter *symbol* is a string giving the name of an identifier. The parameter *Line* is a pointer to the source line where the symbol is defined. Some type checking needs to be done here. Check if there is an entry for *symbol* in the symbol table. If there is, throw a *TypeError* stating that the symbol cannot be redefined (or multiply defined) as a user function. If this symbol has the special name *main*, store *Line* into *Global.main_line*.

void **st_store_const** (ST_Literal *Value*) - Store the constant literal contained in *Value*. First, check if an entry for the symbol contained in *Value* already exists; if so, delete the old entry. Set the new value to a constant and insert it into the table. This new value will have the same scope level as the value it replaces. **This function has been provided for you.**

void **st_store_int** (String *Name*, int *Value*) - Store the integer *Value* for the symbol specified by *Name*. If the entry for the symbol *Name* does not already exist in the symbol table, then one is created, and initialized to the specified value. If an entry for the symbol already exists, then the storage for the old value is released, and the new value is stored. This new value will have the same scope level as the value it replaces. Note however, that the old value may have been something other than an integer value. **This function has been provided for you.**

void **st_store_double** (String *Name*, double *Value*) - Create and set the value of a double symbol. As in *st_store_int*, you must check if an entry specified by *Name* exists, and replace it if so.

void **st_store_string** (String *Name*, String *Value*) - Create and set the value of a String symbol. As in *st_store_int*, you must check if an entry specified by *Name* exists, and replace it if so.

void **st_create_array** (String *Name*, int *Size*) - Create an array entry. Note that the constructor for *STT_Array* objects constructs the array and initializes the values in the array to *null*. This function is called when the array is declared; the values in the array will be set by assignment statements later. As in *st_store_int*, you must check if an entry specified by *Name* exists, and replace it if so.

ST_ENTRY **st_find** (String *Name*) - Find the symbol indicated by *Name*, and return a pointer to its entry in the symbol table. If the symbol does not exist, issue an error message and abort the parse. This function is used when fetching the value of a symbol.

Symbol Table Performance Evaluation

void **st_print_symbol_table** () - This routine prints the contents of the symbol table and is called at various stages of a parse. **This function has been provided for you.** It reports the following:

- the name of the symbol
- the scope of the symbol
- the type of the symbol
- the current value of the symbol

Implementing Dynamic Typing

C3290 is dynamically typed. This means that the type of a variable can change during program execution, not just its value. Every time a symbol is given a value, its type may need to be changed too. The easiest way to do this is to always delete the old entry and insert a new entry, even if the type hasn't changed.

COMP 3290 Introduction to Compiler Construction

Assignment 3 Fall 2008

Token and Rule Reporting Optional

Token and rule reporting has been turned off in this assignment to make the output less bulky. This reporting is controlled by Boolean flags in *Global.java* should you want to enable it again.

Two pass parsing

The C3290 interpreter will use two passes. The first one will be used to parse global variable, constant and function declarations and store their symbol table entries. The second pass will execute the code. The controller code (in *C3290.java*) controls the two passes.

First pass

The first pass is implemented in the Parser function *first_pass*, which is provided for you. The first pass does not do a complete parse of the program; it scans lines, looking for ones that begin with the keywords ***const***, ***function*** or ***var***. When such a line is found, *gen_declaration* is called to create the appropriate symbol table entry and add it to the table. You will modify the code in *gen_declaration* as follows:

function declarations: insert a *STT_UserFunction* entry just before matching *T_Ident* (so that the lexeme is still in *Global.tok_lexeme*). The pointer to the current line is in *Global.cur_line*. We are not interested in parsing the whole function definition during the first pass, so simply return after inserting the entry.

var declarations: Insert an entry for this variable. The modified function *gen_opt_init_or_arraydec* returns an *ST_ENTRY* object that indicates what kind of entry to insert. If null is returned, an uninitialized variable is being declared, so just store an *STT_Int* entry with a value of 0. If *STT_Int* is returned, the variable is initialized to an integer value, so store an *STT_Int* with the appropriate value. Likewise, if *STT_Double*, *STT_String* or *STT_Array* are returned, store the appropriate entry. Use the *store* functions (*st_store_int*, etc.) rather than using *st_insert* just in case the programmer is redefining a variable. You also need to make a copy of *Global.tok_lexeme* just before matching *T_Ident* so you can store the symbol with the entry.

const declarations: simply store the object returned by the modified function. Once again, make a copy of *Global.tok_lexeme* just before matching *T_Ident*.

Second pass

The second pass parses the entire program. We will also add more entries to the symbol table, modify others and do some semantic checking. Since the first pass only reaches a limited amount of the parser, we don't need to check if we are doing the second pass.

function declarations: The modified functions *gen_parm_list* and *gen_rest_of_parms* (these modified functions are provided) build the parameter list for the function. They require the symbol table entry for the function that was inserted during the first pass. Before you match the identifier, use *Global.tok_lexeme* to look up the table entry. Before calling *gen_parm_list* you must push a new scope, since the parameters are part of the function scope. These functions add the parameters to a linked list in reverse order (i.e. the last parameter is first in the list). The parameters are all integer entries, which will be replaced by argument values during execution (in a later assignment). Once the parameter list is built, you will add these entries to the symbol table. Use *st_insert* here (instead of *st_store_int*) so that the parameters are always added as new entries (instead of replacing existing entries). Traversing the parameter list is tricky, since the parameter list uses the same link that is used by the symbol table itself.

COMP 3290 Introduction to Compiler Construction

Assignment 3 Fall 2008

Just before leaving the function declaration, print the contents of the symbol table, with a message indicating the point in the parse, and then pop the scope. This code has been provided for you.

Scope

Whenever the program encounters a block we want to enter a new scope level, and when we leave a block we want to pop the scope. Blocks are encountered in three places in our grammar: a function definition, an *if* statement and a *while* statement. Function definitions are recognized in *gen_declaration*. You should push a new scope before recognizing the parameter list and pop it after recognizing the end of the function (as described above). *While* and *if* statements are recognized in *gen_statement*. You should push a new scope after recognizing the keyword and pop the scope after recognizing the end of the statement. When you pop the scope you also want to print the symbol table contents. The scope operations for a function definition statement are provided for you.

Assignment and scanf statements

Assignment statements are recognized in *gen_statement*. The modified function *gen_dest_var* will store the lexeme of the identifier in the global (to Parser.java) variable *dest_var_name*. When an assignment to a non-subscripted variable is made, store a dummy value as follows: if the name ends with "D" (uppercase D), we will store a double. If the name ends with "S" (uppercase S), we will store a String. With anything else, we will store an integer. (In Java, you can use the *endsWith* String function to test the last character of a String.) Assign all numeric values the value of the current line number, and assign string variables the value of their own lexeme.

Store dummy values as above when recognizing the *scanf* statement in *gen_statement*. We'll assume that the user is reading values into non-subscripted variables only to avoid type checking here. Note we only have to store values here during the second pass. The modified *gen_dest_var* has been provided for you.

Type checking in the parser

We will implement as much type checking in the parser as possible without being able to evaluate expressions (which is the next assignment). All type checking is performed on the second pass (except for the first check below).

Main function definition

Once you have finished the first pass, you should check that the *main* function was defined. If it wasn't, throw a *TypeError*. This code is already provided in *first_pass*.

Assignment statements

Type checking for assignment is performed in *gen_opt_subscript*. If a subscripted assignment is recognized, check that the symbol was defined and it is a type that may be subscripted (String or array). This code is provided as a model.

Function calls

A statement consisting of simply a function call may be done using the *call* keyword. This is recognized in *gen_statement*. If a function call is recognized, check that the symbol is defined and that it is either a built-in or user-defined function.

COMP 3290 Introduction to Compiler Construction Assignment 3 Fall 2008

Factors

In *gen_factor* we are expecting symbols to be defined. Retrieve the value for the symbol using *st_find*, which will throw a *TypeError* if the symbol is not in the table. If the entry is found in the table, we need to pass it to *gen_opt_subscript_or_fncall* for further checking. If a subscripted factor is recognized, check that it is a type that may be subscripted. If a function call is recognized, check that the symbol is a function. For the default case, check that the type is a scalar type (Int, Double or String).

Testing

Test files are provided, named *errorxx.c* (containing errors) and *testxx.c* (containing valid code).

Handing in assignments

You will use the electronic handin (the handin directory is *assign3*). Hand in your code for *Parser.java* and *Symbol.java*.