CS-3323 Principles of Programming Languages – Final Exam

May 1st – May 6th, 2020

Name: ____________________

ID# ____________________ **Score:** ____________________

Instructions (READ CAREFULLY):

- This exam is worth a total of 30 points.
- An extra credit question worth 5 points is also included.
- You have until May 6th, 6:30pm to upload your solution.
- You can use books, slides, handouts, and any sort of notes.
- You can work in groups of up to 3 students, that is, yourself and two more.
- **All students must upload the same solution by the deadline, both in Gradescope and in Canvas (Final Exam directory at the root of the file space)**.
- Each student must upload a single PDF file with all the answers. One possible way is printing a copy of the exam, filling it (either by hand or electronically), and then scanning it. If you don't have access to a scanner, you can take a picture of each page and then transfer the photos to a computer. Regardless of the source device (scanner or camera/phone), consolidate all solutions and pages in a single MS WORD file (or similar word editor) and export it to a PDF format.
- The estimated time to solve this exam is 4 hours. However, preparing the PDF file with the solutions will likely take time. Please consider the overhead of scanning/etc in order to submit your exam solutions by the deadline.
- If you have any question or if something is unclear, email me.
- If your question is deemed as a way of luring me to give you a solution to the exam problems, you will be outright ignored.

Before proceeding, disclose the students of your group. List their names and student ID (9-digit number):

Discussions with peer classmates is encouraged, but your solution can only be identical to that of the members in your group. Disclose below the classmates outside your group (name and ID) with whom you have exchanged opinions, arguments and/or rationale of solutions for this exam:

Context:

- In programming assignments 3–4 you developed a simple compiler with support for integer and floating point arithmetic, and basic control-flow constructs. However, the language and compiler did not support any sort of array data type.
- The following exam questions will guide you through the process of adding support for 1-dimensional integer arrays at both the language level and at the compiler level.
- You can refer to the file grammar.y and all other files of the 3rd and 4th programming assignments for more precise implementation details. You can find all the files in Canvas.
- You are also free to modify the files in order to test that your solution is correct or makes sense. However, that is not a requirement for getting full credit in the exam.

Problem 1: Declaring 1-dimensional arrays (10 points) The following is an excerpt from the grammar.y file (both from the 3rd and 4th assignment), which permits to declare scalar variables (non-arrays) with data types:

```
declaration: datatype T_ID
  {
    assert (symtab);
    assert (itab);
    symbol_t * sym = symbol_create (symtab, $2, $1);
    assert (sym);
    symbol_add (symtab, sym);
  }
    ;
```

First, recall that function `symbol_create` adds a new symbol to the symbol table. The datatype used to create the new symbol is obtained from the `d`atatype non-terminal in the right-hand side of the rule, while the name of the symbol is the string recognized together with the `T_ID` token.

Suppose now that the declaration of variables is modified by adding a non-terminal `array_size` (see below). You can observe that it expects a number such as 10 or 5. If, however, that number is omitted, then the semantic action is to return 1 via the parser's stack (assignment $$ = 1). Obviously, the datatype of the non-terminal `array_size` will be `int`, similar to lines 58, 66–67 in grammar.y. (NOTE: T_INTEGER is the token returned by the scanner when an integer number is recognized, whereas T_DT_INT is the token associated to the keyword `int` in our language.)

```
declaration: datatype array_size T_ID
  {

  }
  ;

array_size : T_INTEGER { $$ = $1; }
           |   { $$ = 1; }
           ;
```

Problem 1A (2 points):

Refer to the definition of `struct simple_symbol` in symtab.hh. The current data structure for declaring variables only has 3 fields: name, addr and datatype. This is not sufficient for declaring 1-dimensional arrays. In 5 or fewer lines, describe why is the current data structure not adequate for supporting 1-dimensional arrays.

Problem 1B (2 points):

Propose one or more modifications to the simple_symbol data structure to be able to declare 1-dimensional arrays. You can assume that you only have to support arrays of integers.

Problem 1C (2 points):

Describe now the modifications you would have to perform on `symbol_create` in order to use it in the modified grammar. You don't need to write code, but if it helps to explain yourself, you can re-write part or all of `symbol_create` to describe the changes you need to do.

Problem 1D (2 points):

Now that you have added support for the new symbol data structure and symbol creation, describe in words the steps or tasks to perform in the new semantic action. You can ignore the declaration of scalar (regular) variables.

```
declaration: datatype array_size T_ID
  {

  }
  ;
```

Problem 1E (2 points):

Given the grammar definition, would it be possible to declare an array in the following way?

```
int n myarray;
```

Briefly justify your answer.

Problem 2: Reading from an array cell (10 points) Refer to the grammar.y file of assignment #3, lines 199–206 and lines 239-244, which define how a value stored at a variable is retrieved:

```
a_fact : varref
      {
        symbol_t * res;
        assert ($1 && "Did not find variable");
        res = make_temp (symtab, $1->datatype);
        itab_instruction_add (itab, OP_LOAD, res->addr, $1->datatype, $1->addr);
        $$ = res;
      }
```

```
varref : T_ID
    {
      symbol_t * sym = symbol_find (symtab, $1);
      assert (sym && "Ooops: Did not find variable!");
      $$ = sym;
    }
```

To be able to read from 1-dimensional array cells, we modify the non-terminal `varref` in the following way:

```
varref : T_ID index
    {
      symbol_t * sym = symbol_find (symtab, $1);
      assert (sym && "Ooops: Did not find variable!");
      /* You will add more code here */
      $$ = sym;
    }
index : '[' a_expr ']' { $$ = $2; }
    |    { $$ = NULL; }
    ;
```

The above rules define a non-terminal `index`, which has two rules associated to it. The first rule stores the symbol of the intermediate variable that stores the index being accessed. The second rule is recognized by the parser when no index expression is found. In that case, the null pointer is stored in the parser's stack.

You should notice that in the original grammar, the loading of a variable's value was performed on one of the rules of the non-terminal `a_fact` via the `OP_LOAD` intermediate operation. The objective now is to move the work being performed in `a_fact`'s semantic action to the non-terminal `varref`.

Refer to files icode.hh and icode.cc of programming assignment #3, in particular, the `run ()` function and the `OP_LOAD` intermediate operation. You are asked to extend the intermediate code generation process to support loading the value of a single array cell into a temporary variable. To do so, we will add a new intermediate operation named `OP_LOAD_ARRAY_CELL`. The following parts of this problem will guide you through the steps of reading values of specific array cells into a temporary variable.

Problem 2A (3 points):
Decide whether you need or not to change the definition of the `simple_icode` data structure (See icode.hh). If you decide to modify it, describe how. List the new fields you are adding, together with their datatype and describe how you intend to use it/them. If you decide to not modify the data structure say so, but your answer will have to be consistent with the subsequent parts of this problem.

Problem 2B (4 points):
Briefly describe how will you implement the `OP_LOAD_ARRAY_CELL` intermediate code operation in the `run()` function of icode.cc. Describe the semantics of each of the fields of the `simple_icode` field for the new operation `OP_LOAD_ARRAY_CELL`. Mention what a field represents, e.g. a variable, an address in memory, the source of the load, or the target of the load.

Problem 2C (3 points):

Now that you have implemented the new intermediate operation, complete the semantic action of the new `varref` rule ("You will add more code here") by calling the new operation `OP_LOAD_ARRAY_CELL` :

```
varref : T_ID index
    {
      symbol_t * sym = symbol_find (symtab, $1);
      assert (sym && "Ooops: Did not find variable!");
      /* You will add more code here */
      $$ = sym;
    }
```

Recall that the new rule should behave as follows:

- If the index actually appears, then some specific entry of the array must be read and stored in a temporary variable.
- If no index is found, then the rule above should behave as a regular `OP_LOAD` operation (e.g. `_T10 = a`).

You can write your answer in words or in pseudocode, whatever is easier.

Problem 3: Writing to an array cell (10 points) As next step, we modify the assignment rule of grammar.y from the below form:

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
```

to its new form:

```
assignment : varref '[' a_expr ']' T_ASSIGN a_expr
  {
    /* You will add more code here */
  }
```

Problem 3A (3 points)

Briefly argue why we had to change the assignment rule in our grammar. (HINT: the reason has to do with certain modifications to the non-terminal `varref`).

Problem 3B (4 points)

Now, briefly describe how would you implement a new `OP_STORE_ARRAY_CELL` intermediate code operation in the `run ()` function of icode.cc. This operation is somewhat similar to `OP_STORE` (See lines 144-150 in icode.cc), but it must store a single value in a particular array cell. You can inspire yourself from your own answer of Problem 2B. Your answer to this question must also be consistent with your answer in Problem 2A.

Problem 3C (3 points)

Finally, complete the semantic actions of the new assignment rule below:

```
assignment : varref '[' a_expr ']' T_ASSIGN a_expr
  {
    /* You will add more code here */
  }
```

You must use the new intermediate code operation `OP_STORE_ARRAY_CELL` in the above semantic action. You don't need to worry about how are now regular assignments performed (e.g. `a := 10`). You can write your answer in words or in pseudocode, whatever is easier.

Extra Credit Problem: Array Bounds Check (5 points)

You are asked to add a safety feature in your compiler. The new feature consists on checking that no out-of-bounds access is attempted on an array, neither when reading nor writing. Notice that this `check` is performed at run-time, i.e. when the application is running.

You have total freedom in deciding how to implement this feature. Just bear in mind that it must be a mix of collecting information at compile-time, and using that information to perform the check at run-time.

Please try to be as concise as possible. Rambling will not get you too many extra points.