Mitchell Coakley

Student #: 1013103

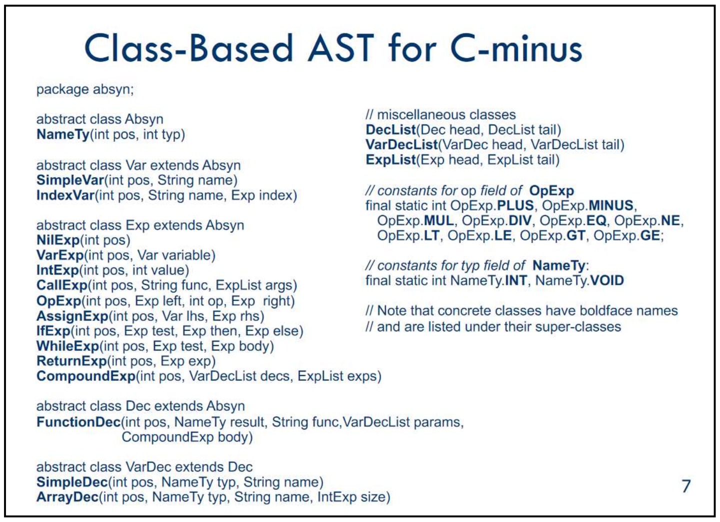
mcoakley@uoguelph.ca

March 5th, 2021

CIS*4650 Checkpoint 3

1. Development:

For my project, checkpoint 3 started with redoing checkpoint 1. Initially, I did not simplify the grammar for checkpoint 1, and as a result checkpoint 2 was difficult to grasp. I created classes based off the ones in lecture 6 for the C Minus language. The CUP grammar was then simplified by starting at a non-terminal (for me this was expression) and attempting to simplify the structure and its immediate relative structures. After this was started, the entire grammar could be simplified by simplifying connected non-terminals until the entire parser was simplified.



Class-Based AST for C-minus

package absyn;

abstract class Absyn
NameTy(int pos, int typ)

abstract class Var extends Absyn
SimpleVar(int pos, String name)
IndexVar(int pos, String name, Exp index)

abstract class Exp extends Absyn
NilExp(int pos)
VarExp(int pos, Var variable)
IntExp(int pos, int value)
CallExp(int pos, String func, ExpList args)
OpExp(int pos, Exp left, int op, Exp right)
AssignExp(int pos, Var lhs, Exp rhs)
IfExp(int pos, Exp test, Exp then, Exp else)
WhileExp(int pos, Exp test, Exp body)
ReturnExp(int pos, Exp exp)
CompoundExp(int pos, VarDecList decs, ExpList exps)

abstract class Dec extends Absyn
FunctionDec(int pos, NameTy result, String func,VarDecList params, CompoundExp body)

abstract class VarDec extends Dec
SimpleDec(int pos, NameTy typ, String name)
ArrayDec(int pos, NameTy typ, String name, IntExp size)

// miscellaneous classes
DecList(Dec head, DecList tail)
VarDecList(VarDec head, VarDecList tail)
ExpList(Exp head, ExpList tail)

// constants for op *field of* **OpExp**
final static int OpExp.**PLUS**, OpExp.**MINUS**, OpExp.**MUL**, OpExp.**DIV**, OpExp.**EQ**, OpExp.**NE**, OpExp.**LT**, OpExp.**LE**, OpExp.**GT**, OpExp.**GE**;

// constants for typ field of **NameTy**:
final static int NameTy.**INT**, NameTy.**VOID**

// Note that concrete classes have boldface names
// and are listed under their super-classes

7

An attempt was made to simplify the constants for OpExp and IfExp into an enum for each respective class. Unfortunately, this did not work due to how java stores enums. Accessing the enums inside of the AST classes was fine, however, attempting to access the enum values outside of the AST classes resulted in failure. This even happened if the enums were "public".

After the parser was remade, error checking was added back to it. Error checking is similar to how it was in my original version of c1. Error checks were put on the non terminals that are general - expression, statement, function_declaration, declaration_list, etc.

2. Development - Time Rush:

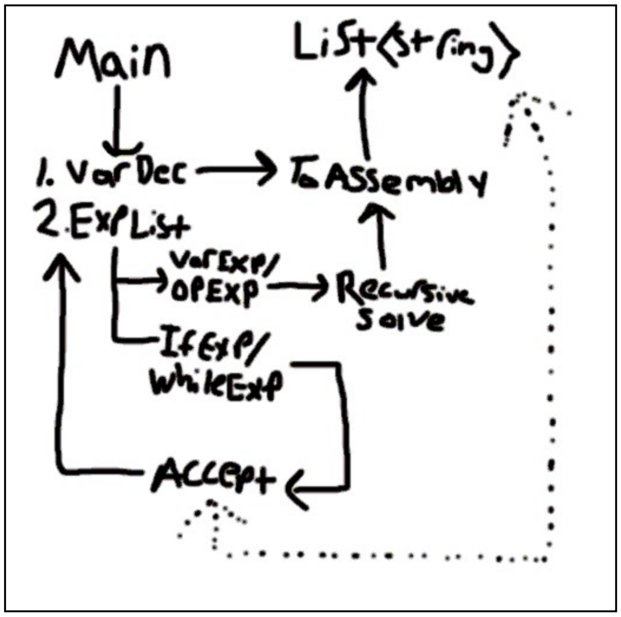
The initial due date was Friday April 9th, and after remaking C1 the due date was less than a week away. As stated in lecture, it is helpful to have an intermediate representation, but not absolutely necessary. Thus the decision was made to entirely skip checkpoint 2. At first this was a temporary decision to see how far c3 could be done without a symbol table or proper type checking. After a brief investigation I determined that c2 could be safely skipped with minimal detriment. Of course, this means that error correction in c2 is nonexistent and is only minimally present in c3 to prevent major crashes.

No symbol table is generated, no types are checked, and no "-s" function is available. The abstract syntax tree goes directly into a class that converts it into a string representative of assembly. This string is then output when the "-c" option is enabled.

2. Development - MitchsMarvelousMachinecodeMaker.java:

This class uses the visitor pattern to turn an abstract syntax tree directly into assembly. The function accepts an Absyn object much like the ShowTreeVisitor class does. Because checkpoint 2 was skipped, this function assumes the input AST is perfectly valid, and has no major flaws. There is minor error checking in the class, but for the most part it will output assembly if it receives a valid AST.

The function follows a similar pattern to ShowTreeVisitor until it detects the main class. Once it detects the main class it does the following:



Immediately the class takes the variable declarations and assigns space in memory to them. If the variables are arrays, it allocates an according space. It then adds assembly code to a string list that represents the current scope. This assembly zeros all variable data that was previously occupying the memory space.

After variable assignment, the class tries to parse the inner compound statement. It assigns variables and then goes through the inner exp list. If the expression is an if/while expression, similar code is activated to parse the inner structure of each. Multiple new scopes are created when making each of these structures. The list that the commands are written to is changed as a result. This is necessary to be able to use the jumps that each of these structures provides. The internal sections of each if/else/while are computed before the jumps are put in place. By finding the length of the internal string arrays, you can deduce the length of the internal structure and put jumps in place accordingly. After computing the jumps, you can merge them, the previous global structure, and the temporary internal sections together.

If variable assignment or an operator expression is reached, the "recursiveSolve" function is called. This function takes in an OpExp and puts the result into a chosen register. It does this by assigning temp variables to each intermediate product and working from the bottom up. Frankly, if I was going to redo this project in any way in the future, I would heavily simplify this function.

After all parts of the syntax tree are accepted the output string list is merged into a single string with line breaks and the output is delivered to a file.

3. Limitations and Assumptions:

Checkpoint 2 was skipped, which means that there is little to no type checking. The "recommended path" for developing the assembly code seemed to be based on a finished a2. As a result, the assembly generated from my class is far different from the code generated from the standard implementation.

Most statements work as intended, although there are a few flaws that came as a result of my hasty implementation. All variables are global after being defined due to no symbol table being present. No functions are defined outside of main. No function calls except those to output(); and input(); are accepted. No array bound checking is present, nor is any other runtime error protection.

With the current implementation I have, it is possible to implement function calls and some aspects of runtime error management. It would not be feasible to implement non global variables in my current project without a major overhaul of my assembly generation class.

4. Challenges:

The biggest hurdle in the entire project was generating valid assembly for long OpExp/VarExp expressions. The recursiveSolve function was created to solve this issue. In general, the function creates two temp variables, and fills them by either directly putting values into them (In the case of IntExp or VarExp) or by calling itself. After filling the temp variables, the function performs an OpExp operation on them. The result is stored in an output register. The output register can be changed as a parameter of the function. My program uses “2” as its output register for no particular reason.

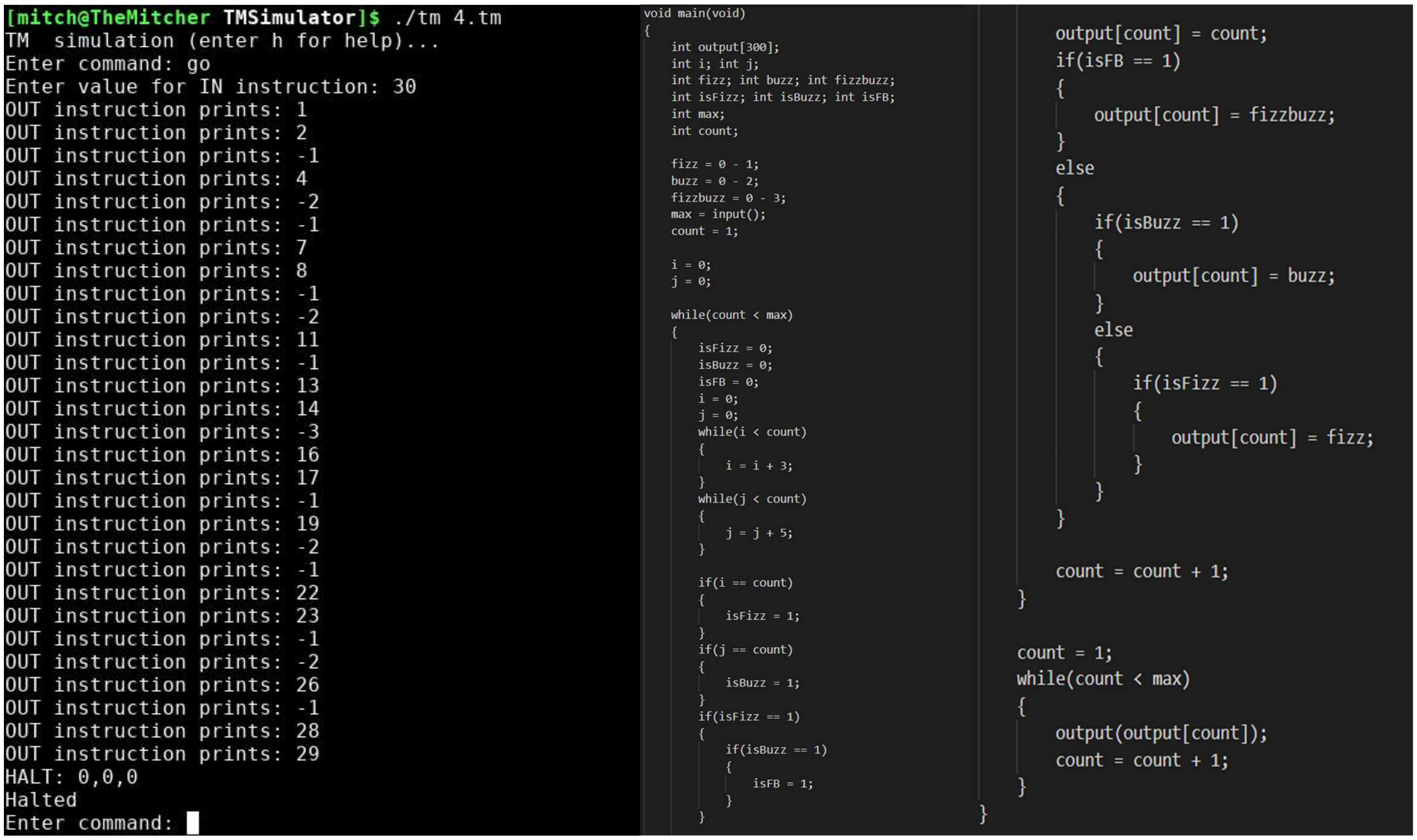
The recursiveSolve method uses only 2 registers and is very inefficient but can solve all OpExp with no function calls besides input() in them.

Another hurdle faced was at the very last minute before submission when I realised that my code for handling getting the value at an array index was broken. I had only recently implemented output(); and so I inadvertently assumed that if the values in memory were correct, arrays were working 100% fine. Arrays were not working 100% fine, and submission had to be delayed to correct the problem.

The root of the problem was two switched registers. Instead of accessing a value at the address of the variable + an offset for the index, my code was trying to incorrectly access the address of the variable + the register number 2. This was fixed quickly.

4. Results and Conclusion:

My compiler can generate code for fac.tm, as well as a variety of custom C Minus files made for it. The other example files, sort.cm and gcd.cm, are incompatible due to their use of local scope and function calls. Example files 1-4.cm were made to be successfully compiled by my compiler. The most complex, 4.cm, is a recreation of the simple game FizzBuzz. It compiles properly to 319 lines of assembly. Results are posted below:



```
[mitch@TheMitcher TMSimulator]$ ./tm 4.tm
TM  simulation (enter h for help)...
Enter command: go
Enter value for IN instruction: 30
OUT instruction prints: 1
OUT instruction prints: 2
OUT instruction prints: -1
OUT instruction prints: 4
OUT instruction prints: -2
OUT instruction prints: -1
OUT instruction prints: 7
OUT instruction prints: 8
OUT instruction prints: -1
OUT instruction prints: -2
OUT instruction prints: 11
OUT instruction prints: -1
OUT instruction prints: 13
OUT instruction prints: 14
OUT instruction prints: -3
OUT instruction prints: 16
OUT instruction prints: 17
OUT instruction prints: -1
OUT instruction prints: 19
OUT instruction prints: -2
OUT instruction prints: -1
OUT instruction prints: 22
OUT instruction prints: 23
OUT instruction prints: -1
OUT instruction prints: -2
OUT instruction prints: 26
OUT instruction prints: -1
OUT instruction prints: 28
OUT instruction prints: 29
HALT: 0,0,0
Halted
Enter command: 
```

```
void main(void)
{
    int output[300];
    int i; int j;
    int fizz; int buzz; int fizzbuzz;
    int isFizz; int isBuzz; int isFB;
    int max;
    int count;

    fizz = 0 - 1;
    buzz = 0 - 2;
    fizzbuzz = 0 - 3;
    max = input();
    count = 1;

    i = 0;
    j = 0;

    while(count < max)
    {
        isFizz = 0;
        isBuzz = 0;
        isFB = 0;
        i = 0;
        j = 0;
        while(i < count)
        {
            i = i + 3;
        }
        while(j < count)
        {
            j = j + 5;
        }

        if(i == count)
        {
            isFizz = 1;
        }
        if(j == count)
        {
            isBuzz = 1;
        }
        if(isFizz == 1)
        {
            if(isBuzz == 1)
            {
                isFB = 1;
            }
        }
        output[count] = count;
        if(isFB == 1)
        {
            output[count] = fizzbuzz;
        }
        else
        {
            if(isBuzz == 1)
            {
                output[count] = buzz;
            }
            else
            {
                if(isFizz == 1)
                {
                    output[count] = fizz;
                }
            }
        }

        count = count + 1;
    }

    count = 1;
    while(count < max)
    {
        output(output[count]);
        count = count + 1;
    }
}
```

The correct results are produced for this program. Overall I am surprised I have gotten my compiler to such a point, despite its shortcomings with error handling and function handling.