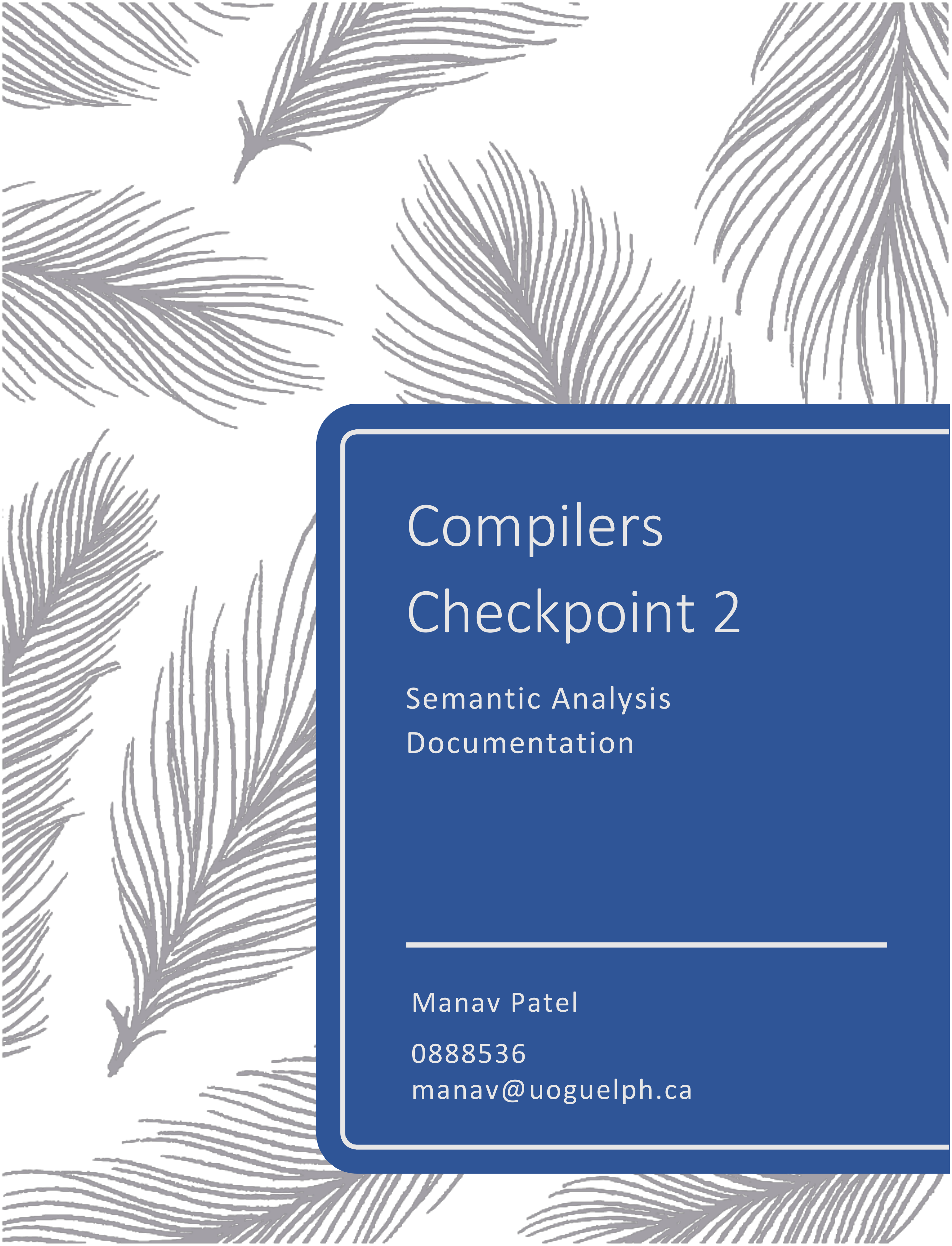

Compilers Checkpoint 2

Semantic Analysis
Documentation

Manav Patel

0888536
manav@uoguelph.ca

Table of Contents

Overview

checkpoint 2 was focused on the implementation of semantic analysis for the cminus compiler. Previously in checkpoint 1 we implemented jflex and cup into one cohesive unit that was responsible for creating an abstract syntax tree. In this checkpoint, we traverse the tree in post order and check for semantic errors such as undefined variables, malformed expressions, missing return statements etc. The implementation of the semantic analyzer has 4 main steps.

1) Traverse the syntax tree that is created from checkpoint 1 by creating a new tree visitor class
2) Create a new data structure that represents a symbol table
3) Add new declarations to the symbol table
4) Check for errors in one pass

Design Process for Semantic Analysis

The first major aspect I needed to take into consideration is the data structure that would hold the symbol table. At first I thought of using a linked list of hashmaps but that didn’t provide me with the flexibility of moving up and down the list with ease, hence I went with the option of using an ArrayList of hashmaps. The hashmap consisted of strings as keys and a DefSym object as the value. The key would be a name of the variable or function. The DefSym object consisted of a Type object which represented either int, void or int array. It also consisted of the name of the variable and an empty arraylist of parameters for function declarations. Once I had the data structure built I began working on the SemanticAnalyzer.java program which would basically implement the tree visitor pattern. Whenever a declaration was found (SimpleDec, ArrayDec, or FuncDec) I would add the declaration to the latest scope of the symbol table. The function declarations had a special case, I added their declaration to the global scope which was the last element in the arraylist. Whenever a new code block was found either a function, if statement or while statement I would add a new entry into the hashmap that represented a new scope or block. Everytime a function was exited or if/while block was exited that scope was removed from the hashmap.

Once the basics of the semantic analyzer was implemented I began to add some error checking. The first error checking was undefined variables. Every time a new SimpleVar was found a lookup for it's type was performed in the symbol table. If the variable did not exist the program would print out the error to stderr.

The second error case I worked on was the capturing of assignment errors. What I did was check the left side of the assignment and store it's type, then I would check each type of the righthand side and if it did not match the type of the left side the program would report an error.

The third major error I worked on was checking the types of a parameter with the arguments. This proved to be the hardest part of this assignment as conceptually it didn't fit with the post order traversal. What I did was create a Boolean flag that represented if the current SimpleVar is an argument or not. If it is I would look up the global table and get the list of parameters for that function definition. This is where storing the paramertes in a DefSym object proved to be useful, I simply compared the list to the current variables type and would throw an error if the types didn’t match.

Learning Outcomes

I learned a great deal about semantic analysis and how the compilers store scopes of variables in a symbol table. One of the mistakes I made was actually adding a new entry to the symbol table whenever a compound expression was found as a compound expression consisted of a new code block, but this proved to be wrong. Adding a new scope to the compound statement would add new entries when it wasn’t needed. Another lesson I learned was really solidifying my knowledge of the visitor pattern