CSE M501 Project Report, Spring 2018
AN (group id)
David Porter, Wei Lin
davidpor, veilam

Overview

The goal of the project was to build a compiler from scratch that will help us learn about all of the components and intricacies of modern compilers. The language that was targeted for the project is MiniJava, a small subset of full Java. The fact that MiniJava is a subset of proper Java allowed us to more easily validate and test our implementation as we could always compare it to the full Java.

Our compiler consisted of multiple major components: the scanner, parser, semantic checker, X86 assembly code generator as well as CSEM501 additions of a MiniJava to Python transpiler, to be discussed later.

Scanner

The scanner provides the expected functionality of recognizing tokens in source program. We used the JFlex framework to built out the scanner component, specifying regular expression all of the tokens that are part of the MiniJava language.

Parser & Semantic Checker

The parser uses the MiniJava grammar to build an AST tree out of the tokens emitted by the scanner. The parser was implemented using the CUP framework. MiniJava is specified in a BNF form, however, it is ambiguous if literally translated into CUP. As a result, we had to resolve the ambiguity and modify the grammar as needed.

The semantic checker was a bit tricky to implement as it required a deeper understanding of the MiniJava semantics and the information in the source program. We used the visitor pattern to traverse the AST tree in multiple passes to built out the semantic checker. Each pass collected more and more information about the source program which was ultimately used to check the semantics of it. The design that we arrived to had a GlobalSymbolTable which is a dictionary that maps the name of class to a ClassSymbolTable object. The class symbol table has information about the class, like the name, what class it is inherited from, class variables, and also a reference to a MethodSymbolTable object. The MethodSymbolTable is also a dictionary which maps a method name to a Method object which contains all the information about the method (like it's return type, arguments, and variable declarations). The use of HashMaps and this nested pattern turned out to be a good decision as it allowed for quick lookups, O(1) during semantic checking.

After the two passes (one to get GlobalSymbolTable information and the second to get ClassSymbolTable information and MethodSymbolTable information) we had a third pass that actually did the semantic checking itself. The semantic checking pass went through the AST tree and validated the expected language rules. We built multiple helpers to assist in this process, for example a method that returns the expected type for an Expression (looking up the necessary details in all of the symbol tables as part of the process). Our semantic checker ended up looking similar to a unit test framework in the sense that it asserts that the type is of some expected type and if not, an error is recorded. Here is an example for the while loop check that asserts that the expression in a while loop is a boolean.

```
// Exp e;
// Statement s;
   public void visit(While n) {

       // e should be bool
       errorChecker.addSemanticError(
               SemanticsHelpers.checkExpressionTypeEqualTo(
                       n.e,
                        globalSymbolTable,
                       classSymbolTable,
                       methodSymbolTable,
                       Type.DataType.BOOLEAN_DATYPE
               )
       );
       n.e.accept(this);
       n.s.accept(this);
   }
```

Our semantic helper class will automatically check if the expression will evaluate to a boolean and if not, an error will be recorded with all the details including line number information. If the expression is more complicated than a simple boolean, like a call to a function that returns a boolean, the method will recursively find the final type that the expression will evaluate to and validate it.

Code Gen

The last major component of our compiler is the Assembly Code Generator. We targeted X86 Linux as our platform. The assembly generator is another visitor that does a pass through the AST tree emitting assembly code as it goes. We decided to come up with some conventions, namely using one register to pass data around (%rax). We also made a decision to dedicate the (%rdi) register to pass around “this” argument which is a pointer to the class object. Arguments were passed in register, however started from %rsi (since %rdi was occupied by “this”). By

passing around “this” pointer, it allowed us to have quick access to “this” pointer to access variables and vtable information locally. The trickiest part of the assembly code generator was to ensure that data was not being overwritten when visiting nested style expressions (e.g. a while loop with a call inside it). We resolved this issue by pushing arguments and other registers that could be overwritten on the stack, prior to evaluating an inner expression to avoid having the data overwritten accidently.

In terms of the language features we implemented all of the MiniJava features. Specifically we included, arithmetic expressions, if/while, object creation, dynamic dispatch, arrays, doubles. As extra credit, we also implemented double arrays and the Math.sqrt function for doubles.

In terms of language features that don’t quite work, we weren’t able to get proper shadowing support of class variables working, due to some design decisions we made earlier on the project that made implementation much harder. If we had more time we would go back and redesign a few things to make this work.

Testing

In terms of testing, early on we realized that testing would be one of the most significant time investments and important areas to work on. In terms of tests, our original objective was to get all of the included sample programs working properly. We also created a test suite which is a collection of many small java programs that exercise specific language features. Something that was a great investment of our time was to write some automated scripts that can automatically take a directory of Java programs, run all of them against the normal javac compiler and then run the same programs in our MiniJava compiler and compare and report the results. This automated script was incredibly useful and served as our regression test suite that gave us confidence that we did not break anything with new features that were implemented.

Extra Features / Extra credit

As for extra features, we implemented new language features including double arrays and a Math.sqrt function for doubles.

As for split of work, we mostly pair programmed and often used video calls and screen sharing to make building out the compiler simpler. Having two people working together made it much easier to debug things and overall was much more fun! Both of us also understood all the design decisions and areas since they were made together.

5th year masters project

As part of being in the first offering of the 5th year masters version of this course, we completed an additional project to extend our compiler. The project we choose to learn more is source to source translation or so called “transpiling”. Transpiling has been much more popular recently, especially in the Javascript community which has new language features in versions of the language, but has to support old browsers which don’t have the latest features implemented. The solution is to transpile programs written in the new version of the language to older version of the language, to support all types of browsers.

We decided to explore this area and build a transpiler for MiniJava to the python programming language (both version 2 and 3 are supported). Because of the compiler’s design decisions, the transpiling basically involved creating a new visitor which does an additional pass of the AST tree which generates python code. The challenging part of the project was mapping java language to python language in cases where there are differences. For example, python requires classes to have a explicit constructor unlike MiniJava, so as part of pass for each class we generated a “__init__” method which initializes all of the class variables to None (as per python language requirements). Additionly, in python, if a class extends another class, the initialization of the child class does not automatically initialize the parent class. To overcome this, in the child class constructor, we added a explicit method call to the superclass to initialize the parent’s instance variables. Another interesting case was that python does explicitly have an array type of a fixed size like in MiniJava, rather only a container type more analogous to “ArrayList or vector” data structure. To overcome this, when creating an array in python, we initialize an array to contain zeros for the right number of elements based on the size of the array.

To test our implementation we created an automated test suite that compared the SampleMiniJavaPrograms java output to that of transpiled version of the program, after being run under the python interpreter. All of the Sample Programs work correctly and all of the MiniJava language features are supported for our python transpiler. To use the python transpiler, we added a new command line flag “-PY” which takes in a MiniJavaProgram and will emit a python version of the program. Here is an example of the Linear Search program.

Java Linear Search	Python Linear Search

`class LinearSearch{`
`  public static void main(String[] a){`
`  System.out.println(new LS().Start(10));`
`  }`
`}`
`// This class contains an array of integers and`
`// methods to initialize, print and search the array`
	`class LinearSearch:`
	`  def __init__(self):`
	`  pass`
	
	`  def main(self):`
	`  print(LS().Start(10))`

```java
// using Linear Search
class LS {
    int[] number ;
    int size ;

    // Invoke methods to initialize, print and search
    // for elements on the array
    public int Start(int sz){
       int aux01 ;
       int aux02 ;

       aux01 = this.Init(sz);
       aux02 = this.Print();
       System.out.println(9999);
       System.out.println(this.Search(8));
       System.out.println(this.Search(12)) ;
       System.out.println(this.Search(17)) ;
       System.out.println(this.Search(50)) ;
       return 55 ;
    }
    // Print array of integers
    public int Print(){
       int j ;

       j = 1 ;
       while (j < (size)) {
           System.out.println(number[j]);
           j = j + 1 ;
       }
       return 0 ;
    }
    // Search for a specific value (num) using
    // linear search
    public int Search(int num){
       int j ;
       boolean ls01 ;
       int ifound ;
       int aux01 ;
       int aux02 ;
       int nt ;

       j = 1 ;
       ls01 = false ;
       ifound = 0 ;

       //System.out.println(num);
       while (j < (size)) {
           aux01 = number[j] ;
           aux02 = num + 1 ;
```

```python
class LS(object):
    def __init__(self):
        self.number = None
        self.size = None
        pass

    def Start(self, sz):
        aux01 = None
        aux02 = None
        aux01 = self.Init(sz)
        aux02 = self.Print()
        print(9999)
        print(self.Search(8))
        print(self.Search(12))
        print(self.Search(17))
        print(self.Search(50))
        return 55

    def Print(self):
        j = None
        j = 1
        while j < self.size:
            print(self.number[j])
            j = j + 1
        return 0

    def Search(self, num):
        j = None
        ls01 = None
        ifound = None
        aux01 = None
        aux02 = None
        nt = None
        j = 1
        ls01 = False
        ifound = 0
        while j < self.size:
            aux01 = self.number[j]
            aux02 = num + 1
            if aux01 < num:
                nt = 0
            else:
                if not aux01 < aux02:
                    nt = 0
                else:
                    ls01 = True
                    ifound = 1
                    j = self.size
```

```
            if (aux01 < num) nt = 0 ;
            else if (!(aux01 < aux02)) nt = 0 ;
            else {
               ls01 = true ;
               ifound = 1 ;
               j = size ;
            }
            j = j + 1 ;
         }

         return ifound ;
      }
      // initialize array of integers with some
      // some sequence
      public int Init(int sz){
         int j ;
         int k ;
         int aux01 ;
         int aux02 ;

         size = sz ;
         number = new int[sz] ;

         j = 1 ;
         k = size + 1 ;
         while (j < (size)) {
            aux01 = 2 * j ;
            aux02 = k - 3 ;
            number[j] = aux01 + aux02 ;
            j = j + 1 ;
            k = k - 1 ;
         }
         return 0 ;
      }

}
```

```
            j = j + 1
        return ifound

    def Init(self, sz):
        j = None
        k = None
        aux01 = None
        aux02 = None
        self.size = sz
        self.number = [0] * sz
        j = 1
        k = self.size + 1
        while j < self.size:
            aux01 = 2 * j
            aux02 = k - 3
            self.number[j] = aux01 + aux02
            j = j + 1
            k = k - 1
        return 0



if __name__ == "__main__":
    LinearSearch().main()
```

Summary

In conclusion, having this being partner project was really useful. It was very gratifying to build out a functioning compiler that can compile real MiniJava programs and run. Compilers is definitely less of a mystery now and an interesting area to explore further.