CSE M 501 Project Report, Autumn 2020
Group-id: AM
Names: Chanwut Kittivorawong, Frank Qin
UW-net-ids: chanwutk, qzh

Codegen

Working language features

Arithmetic expressions	If/While	Object creation
Dynamic dispatch	Arrays	NullPointerException
IndexOutOfBoundException with an error message stating bound and index accessed	Function calling with more than 5 parameters	Short-circuiting when executing a boolean binary expression

Extension

We added `try/catch` block, `throw`, and `RuntimeException` to MiniJava.
`RuntimeException` is the base class for all the exception classes in MiniJava. In other words, any class that extends `RuntimeException` can be thrown and caught. We force users to use `RuntimeException` so that we do not have to add support for annotating exceptions in the signature of each function.
Similar to Java, one try block can be followed by multiple catch blocks. The first (and only) catch block that has the type of its parameter exception compatible with the thrown exception is executed.
For throwing, whenever we have a throw statement, it will jump to the most inner catch block. If there is no catch block wrapping around the throw statement, it will return out of the function with a flag that an exception is thrown. After every call, we check for an exception and throw with the same exception if the function call throws an exception.

Example code 1 (throw):

Java	Assembly of the highlighted part

`class Test {`
`  public static void main(String[] args)`
`{`
`        System.out.println(1);`
`  }`
`}`
 	
`class C0 extends RuntimeException {}`
 	
`class C1 {`
`  public int run() {`
`    throw new C0();`
`    return 0;`
`  }`
`}`
	`C1$run:`
	`        push %rbp`
	`        movq %rsp, %rbp`
	`        sub $8, %rsp`
	`        movq %rdi, -8(%rbp)`
	`        sub $8, %rsp`
	`        movq $8, %rdi`
	`        call mjcalloc`
	`        add $8, %rsp`
	`        lea C0$(%rip), %rdx`
	`        movq %rdx, (%rax) ; initialize the exception object`
	`        test %rax, %rax`
	`        je .$NullPointer$handler`
	`        movq $1, %rdx ; return with exeption`
	`        leave`
	`        ret`
	`        movq $0, %rax ; normal return`
	`        movq $0, %rdx`
	`        leave`
	`        ret`

Explanation

Whenever we return a value from a function, we set %rax to the return value, and %rdx as a flag for testing if the function returns or throws. If the function does not throw, it sets %rdx to 0 and %rax contains the return value. Otherwise, it sets %rdx to 1 and sets %rax pointing to the exception object.

Example code 2 (try/catch):

Java

```
class Test {
  public static void main(String[] args)
{
    System.out.println(1);
  }
}

class C0 extends RuntimeException {}

class C1 {
  public int run0() {
    return 1;
  }
  public int run() {
    int a;
    try {
      a = this.run0();
    } catch (C0 e) {
      a = 2;
    }
    return a;
  }
}
```

Assembly of the highlighted part

```
C1$run:
        push %rbp
        movq %rsp, %rbp
        sub $16, %rsp
        movq %rdi, -8(%rbp)
        movq $0, (%rsp)
        movq -8(%rbp), %rax
        test %rax, %rax
        je .$NullPointer$handler
        movq %rax, %rdi
        movq (%rdi), %rax
        call *8(%rax)
        test %rdx, %rdx ; check if call has an exception
        je .L6
        jmp .L4
.L6: ; no exception, continue execution
        movq %rax, -16(%rbp)
        jmp .L5
.L4: ; catch
        lea -16(%rbp), %rsp
        push %rax
        movq -24(%rbp), %rdi
        lea C0$(%rip), %rsi
        call .$instanceof ; check if type of exception is compatible
        test %rax, %rax
        je .L7
        movq $2, %rax
        movq %rax, -16(%rbp)
        add $8, %rsp
        jmp .L5
.L7: ; exception not match, re-throw exception
        pop %rax
        movq $1, %rdx
        leave
        ret
.L5: ; after try-catch block
        movq -16(%rbp), %rax
        movq $0, %rdx
        leave
        ret
```

Explanation

In the assembly after the call instruction, it checks if %rdx is 1 or not. If it is not, it jumps to .L6, which is continuing the execution. Otherwise, it jumps to .L4, which is the first catch block. It then checks for an instance of thrown error (pointed by old %rax). Check if it is equal to C0. If so, it executes the catch block and jumps to .L5 (after catch block) afterward. Otherwise, it jumps to .L7, which means that there is no catch with a compatible type with the thrown object. Then, it throws the same exception out of run().

Unimplemented language features

Non-extension

We have implemented all features in the standard MiniJava.

Extension

Even though all throwable classes are declared directly or indirectly subclasses of `RuntimeException`, `RuntimeException` is just a language keyword instead of a real class. It is similar to `String` in the standard MiniJava.

Our code-gen cannot trace execution paths. For example, a function with unreachable code after a throw statement does not compile in Java. But, our Minijava does not check for the case. That is, our code-gen will generate code after a throw statement even if it will never run.

Summary of the tests

Scanner

We test a set of sample codes that contains all the token types that we have in Minijava. The tests include ignoring comments and seeing whitespaces as separators for tokens.

Parser

We started by testing a basic empty Minijava program that contains only the main class (test/resources/Minimal.java). Then, we test arithmetics, specifically precedence (test/resources/Precedence.java). The arithmetic that we test in this test is in a simple main class since we only want to test the arithmetic. Finally, we test more complicated structures, such as multiple classes, a class with extends, methods declarations/calls, variables declarations, assignments, and exceptions handling (test/resources/Classes.java).

Type-check

We divided cases into the cases that pass the static type-checker and the cases that do not pass the static type-checker. For the classes that do not pass the type-checker we tested:

- **Cycles in classes hierarchy**: by extending classes in a cycle and compare error result
- **Not-declared identifiers**: by creating a class instance, calling a method, using a variable that does not exist. Then, we compare the error results
- **Overriding error**: by override a method with the same name but the method of the subclass does not have a stronger method signature than the method of the super-class. Overriding errors include:
    - having not stronger return type (not same or sub-type) or not stronger parameter types (not same or super-type)
    - having an incorrect number of parameters
    - these overriding errors of the method from the super-class of its super-class.

Code-gen

We compile a Java program with our Minijava, compile the resulting assembly with boot.c with GCC. Then, we run the compiled program and test the output against the result of the same program, but compiled from javac.
The test cases that we use include:

- all the sample programs.
- test method calls with more many parameters (using the stack to store the values of the parameters)
- test try-catch block
- test try blocks with multiple catch blocks and hierarchy of exception type caught by the catch blocks
- test if exceptions are caught properly from the outer method call.

The exceptions that are thrown without any catch are printed to the standard error. And, we compare the error with our expected error.

Contributions

We divided works for each assignment to do separately.

Scanner

Mick: Added keywords to cup/jflex specs.
Frank: Implemented comment and tests

Parser

Mick: Implemented ASTPrintVisitor for printing AST of a MiniJava program.
Frank: Implemented the MiniJava parser

Type-check

Mick: Implemented visitors for class declarations and type-check.
Frank: Implemented symbol table and types

Code-gen

Mick: Implemented the tests for code-gen.
Frank: Implemented code-gen

Extension

Mick: Implemented the scanner, parser, and type-check and their tests for try-catch blocks and throws. Add supports for scoping in catch blocks in the symbol table.
Frank: Implemented semantics checking and code-gen for exceptions.

Conclusions

Good

We are able to do static type-checking and code-gen working for all the requirements. We are also proud of our exception handling. Exceptions are important for reporting errors. Being able to properly throw and catch exceptions helps users to handle errors better with more options of control-flow features.

Could be better

Right now, we do not have a base exception class (RuntimeException). We would like to create a functional RuntimeException class so that it works similarly to what we have in Java.

Our IndexOutOfBoundException and NullPointerException also work differently than our regular exceptions. Due to our time constraints, we do not implement throwing and catching IndexOutOfBoundException and NullPointerException. Instead, if one of these exceptions occurs, the program aborts with an appropriate message.

Would have done differently

We do not have strings in our Minijava. Error reporting would be benefitted from having strings in Minijava because we can add descriptive descriptions to our errors when throwing errors at the top-level. So, we would like to add support for String type in our Minijava before having exception handling.