Homework 2

CS 132 - Dis 1A - 10/11/2019 2:00pm-3:50pm

Shuyang Liu

Midterm on October 24

Review Session on October 22

LL1 Academy

http://ll1academy.cs.ucla.edu/

Homework 2 : Type Checking

Why do we need Type Checking?

First of all, **what is type in the context of programming language?**

Why do we need type at all?

What is Type Checking? What's a type safe language?

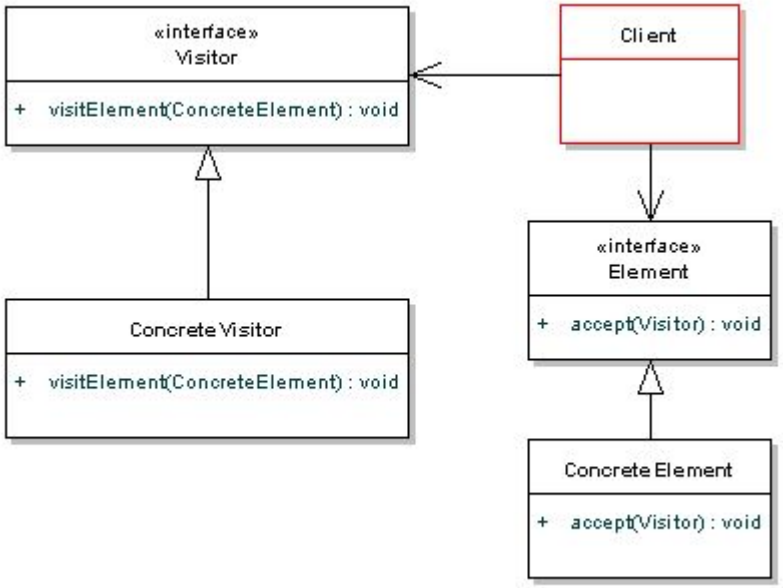
Why we should all use type safe language?

- Security and Type Safety

What are some modern Type Safe Languages?

- Spoiler: not Java... https://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html

Visitor Pattern



Visitor Pattern

One of the commonly used design pattern for object oriented programs

> *"the visitor design pattern is a way of separating an algorithm from an object structure on which it operates" --- Wikipedia*

Overloading in Java

Just override the visit methods and add functionality in each visit method

Read More:

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma (Author), Richard Helm (Author), Ralph Johnson (Author), John Vlissides (Author), Grady Booch (Foreword)

Visitor Pattern in JTB

PlusExpression.java

```
public <R,A> R
accept(visitor.GJVisitor<R,A> v, A argu)
{

      return v.visit(this,argu);

}
```

GJDepthFirst.java

```
/**
    * f0 -> PrimaryExpression()
    * f1 -> "+"
    * f2 -> PrimaryExpression()
 */
public R visit(PlusExpression n, A argu) {
      R _ret=null;
      n.f0.accept(this, argu);
      n.f1.accept(this, argu);
      n.f2.accept(this, argu);
      return _ret;
}
```

Setting up JTB and JavaCC for MiniJava

You have two options:

- Setting up JTB and get the MiniJava Parser manually
  - Not recommended
  - You need to download the minijava.jj grammar file for MiniJava
- **Use `cs132.tar` on CCLE**
  - **Recommended**
  - It already has everything you need
  - (Optional) you can open it with Intellij IDEA and it sets up everything automatically for you
  - **You only need to submit the source files that you wrote, no need to submit any generated class files.**

Setting up for hw2 (the easy way with `cs132.tar`)

1. Download **`cs132.tar`** from CCLE
2. `cd src/parse/java`
3. `java -jar ../../../misc/jtb132.jar ../../../grammars/minijava.jj`
4. `java -cp ../../../misc/javacc.jar javacc jtb.out.jj`
5. Create a Java file `src/main/java/Typecheck.java`
6. Create your main method in `Typecheck.java`
7. Add `import syntaxtree.*;` at the beginning of the file (just for testing whether the libraries are recognizable, you can change it later according to your need)
8. `gradle build`
9. `gradle run`

Writing your code faster: Using an IDE

Intellij IDEA is an IDE with a lot of helpful features. It can make your code development process much faster

- No need to remember all of the library/package names
- No need to worry about missing a semi-colon somewhere
- Auto-complete
- Build-in support for gradle
- https://www.jetbrains.com/idea/

Eclipse is similar

How to get started using the MiniJava Parser?

- If you followed all of the steps setting up the parser files, you can find the parser files in `src/parse/java`
- Only three steps to finish your homework:

  ➳ Read the MiniJava program from stdin

  ➳ Build an AST in memory using the MiniJava parser

  ➳ Traverse the AST using visitor pattern and do the Type Checking

Yep! It's very simple! (⇠ except, not really...)

But How???

➽ **Read the MiniJava program from stdin** (Having an InputStream ready)

```
InputStream in = System.in;
```

(or read from a testing file when you are testing your program)

```
String filename = "testcases/hw2/Basic.java";
```

```
InputStream in = new FileInputStream(filename);
```

But How???

- **Build an AST in memory using the MiniJava parser (one line of code)**

```
Node root = new MiniJavaParser(in).Goal();
```

In order to build this line of code, you need to import:

```
import syntaxtree.Node;
```

But How???

- Traverse the AST using visitor pattern and do the Type Checking

1. Create your own visitor by extending `GJDepthFirst` or `GJNoArguDepthFirst` or `GJVoidDepthFirst`

```
import syntaxtree.*;
import visitor.GJDepthFirst;

public class MyVisiter extends GJDepthFirst {
    ...
```

But How???

➽ **Traverse the AST using visitor pattern and do the Type Checking**

2. **Override the `visit` methods**
    a. In Intellij, you can simply press **Alt+Insert** then select **Override Methods**

```
import syntaxtree.*;
import visitor.GJDepthFirst;

public class MyVisiter extends GJDepthFirst {

    @Override
    public Object visit(NodeList nodeList, Object o) {
        return super.visit(nodeList, o);
    }
    ...
```

But How???

➽ **Traverse the AST using visitor pattern and do the Type Checking**

3. **Write your type checking code inside each visit method**
    a. Each visit method visit a node on the AST
    b. Build a **symbol table (more details later)**
    c. You can do the type checking according to the information you got from the AST and the symbol table
    d. Read the type rules carefully! Do not make assumptions
    e. Unlike last time in homework 1, the specifications of MiniJava should be clear enough

But How???

➽ **Traverse the AST using visitor pattern and do the Type Checking**

4. **Where can I find the documentations?**
    a. The source files are in **`src/parse/java`**
    b. JTB Documentation: http://compilers.cs.ucla.edu/jtb/jtb-2003/docs.html
    c. JavaCC Documentation (you probably don't need this): https://javacc.org/doc

Symbol Table

A **symbol table** is basically a table consisting the information of each variable name. The compiler must either record this information in the IR or re-derive it on demand. (Sometimes you can find a place in the IR with variable information).

In the context of type checking, we need to record down the **types of each variable** and the **inheritance relationships** among each class so that we can "check" whether they follows the **type rules** on the specification.

What data structure though? Anything with the following two functions as interface

1. **`lookUp(name)`** returns the type stored in the table at `h(name)` if one exists. Otherwise, it returns a value indicating that name was not found.

2. **`insert(name,type)`** stores the information in record in the table at `h(name)`. It may expand the table to accommodate the record for name.

A typical choice would be `Map` in Java

You can implement it either by constructing a global table or passing it as an argument to **`visit()`** functions while you are traversing the AST.

Read More: **Engineering a Compiler** (second edition) by Keith D. Cooper and Linda Torczon - **Chapter 5 Section 5.5**

Handling Scopes

```
. . .
int value=10;

void pro_one()
{
   int one_1;
   int one_2;

   {                    \
      int one_3;         |_  inner scope 1
      int one_4;         |
   }                    /

   int one_5;

   {                    \
      int one_6;         |_  inner scope 2
      int one_7;         |
   }                    /
}
```

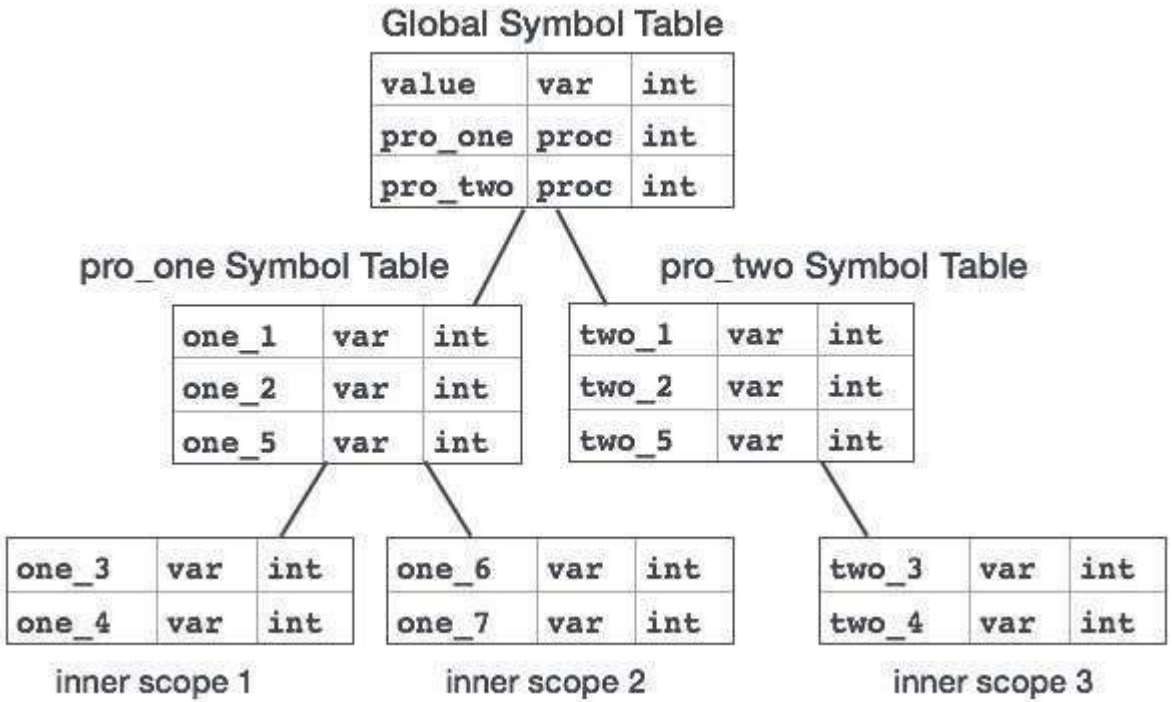
```
void pro_two()
{
   int two_1;
   int two_2;

   {                    \
      int two_3;         |_  inner scope 3
      int two_4;         |
   }                    /

   int two_5;
}
. . .
```

Reference: https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm

Handling Scopes



Type Checking

I was not gonna suggest any “cool stuff to do” this time because this project is probably gonna take you a long time if you are aiming for 100% on the grading server (a **LOT** of hidden edge cases...)

But in case you are bored...

Here are some cool stuffs you can do (and as always, please **do NOT** include them in the homework submission)

- As always, more **automation**
  - Can we generate a type checker by feeding the machine a set of type rules? (just like how JavaCC generated a parser for MiniJava)
- **Optimization** of the Symbol Table
  - How do we improve the implementation of symbol table?
- **Imperative style** vs. **Functional style**
  - If you implemented the type checker the imperative way, maybe you would like to try the functional version of it as well

Recommended Readings

Modern Compiler Implementation in Java (the textbook)

- **Chapter 4** on visitor pattern
- **Chapter 5** on Semantic Analysis (Type Checking and Symbol Table)

Design Patterns: Elements of Reusable Object-Oriented Software

- Visitor pattern

Next Time, more type checking...