CS164 Programming Languages and Compilers **Spring 2019**

Programming Project 2

Assigned: October 3, 2020 **Due:** October 30, 2020 at 11:59pm

1 Overview

The three programming assignments in this course will direct you to develop a compiler for ChocoPy, a statically typed dialect of Python. The assignments will cover (1) lexing and parsing of ChocoPy into an abstract syntax tree (AST), (2) semantic analysis of the AST, and (3) code generation.

For this assignment, you are to implement semantic analysis and type checking for ChocoPy. This phase of the compiler takes as input the AST of a parsed, syntactically valid ChocoPy program, and outputs the same AST with additional type information added to expression nodes, and (possibly) error messages corresponding to semantic errors in the input program.

This assignment will likely require much more effort than PA1, so **start early**. This assignment also allows for a large amount of flexibility in design choices. Make sure to read through this document and the ChocoPy reference manual thoroughly before deciding on an implementation strategy.

2 Getting started

The setup this is essentially the same as last time. You can use the same team repository and working directory as you did for Project 1.

- First make sure that your current working directory is clean (that is, `git status` shows no untracked files or uncommitted changes.)
- You will find a skeleton for the project in our shared repository. Let's assume you have cloned your team repository into a directory `team-repo`. Once cloned, any one of you can set things up in the master branch with

```
$ cd team-repo
$ git fetch shared
$ git merge shared/proj2 -m "Start project 2 from skeleton"
$ git push
```

  (Again, only one member should do this, or all kinds of conflicts will result). Other team members will now be able to pull this to their own machines with

```
$ cd team-repo
$ git pull
```

  After these steps, the code for the project will be in the `proj2` subdirectory of your local working directory. Run any of the commands referred to below in that subdirectory.

- Although these directions assume you put your project in the `master` branch, that isn't really necessary. If your team desires, for example, to keep each project in a separate branch, that works, too (but obviously, you must all agree on the structure!). If you do this, you are responsible for learning the proper Git procedures to accomplish it. Whatever you do, your project ***must*** be in the subdirectory `proj2` of the branch you work on, just as it is laid out in the `shared/proj2` directory you merge in.

- The additional software required is the same as for Project 1: Git, Apache Maven and JDK 8+.

- Run `mvn clean package`. This will compile the starter code, which implements a tiny subset of the semantic analysis for ChocoPy. Your goal is to implement the full semantic analysis for ChocoPy, including type checking, as per the language reference manual. This document specifies the expected output format of typed ASTs.

- Run the following command to test your analysis against sample inputs (JSON outputs from parsing) and expected outputs. Only one test will pass with the starter code:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=.s \
      --dir src/test/data/pa2/sample --test
```

Windows users should replace the colon between the JAR names in the classpath with a semicolon: `java -cp "chocopy-ref.jar;target/assignment.jar" ....` This applies to all `java` commands listed in this document.

You will probably get tired of typing '`java -cp ...chocopy.ChocoPy --pass=.s`' over and over. Those of you using Bash may want to use the **alias** command to create an abbreviation. Those using IntelliJ, of course, can simply set up suitable run and debug configurations.

3 Files and directories

The assignment repository contains a number of files that provide a skeleton for the project. Some of these files should not be modified, as they are essential for the assignment to compile correctly. Other files must be modified in order to complete the assignment. You may also have to create some new files in this directory structure. The list below summarizes each file or directory in the provided skeleton. They are all under the `proj2` subdirectory.

- `pom.xml`: The Apache Maven build configuration. You do not need to modify this as it is set up to compile the entire pipeline. We will overwrite this file with the original `pom.xml` while autograding.

- `src/`: The `src` directory contains manually editable source files, some of which you must modify for this assignment. Classes in the `chocopy.common` package may not be modified, because they are common to your assignment and the reference implementation/test framework. However, you are free to duplicate/extend these classes in the `chocopy.pa2` package or elsewhere.

  - `src/main/java/chocopy/pa2/StudentAnalysis.java`: This class is the entry point to the semantic analysis phase of your compiler. It contains a single method: `public static Program process(Program input, boolean debug)`. The first argument to this method will be the AST produced by the parser in JSON format, and the return value should be the output of semantic analysis in JSON format. The second argument to this method is true if the --debug flag is provided on the command line when invoking the compiler. The starter code contains a bare-bones implementation of this method that reads an AST from JSON into a Program node, and serializes a result AST node into JSON. The semantic analysis in the starter code performs two passes over the AST; you may have to modify this method to add more passes or create more data structures such as the type hierarchy.

- src/main/java/chocopy/common/astnodes/*.java: This package contains one class for every AST-node kind that appears in the expected input/output JSON format (ref. Section 4.1.2 and Figure 1).
- src/main/java/chocopy/common/analysis/NodeAnalyzer.java: An interface containing method overloads for every node class in the AST hierarchy. Section 5.1 describes its use.
- src/main/java/chocopy/common/analysis/AbstractNodeAnalyzer.java: A dummy implementation of the NodeAnlyzer interface.
- src/main/java/chocopy/common/analysis/SymbolTable.java: This class contains a sample implementation of a symbol table, which is a essentially a map from strings to values of a generic type T. You can extend/duplicate this class in the chocopy.pa2 package if you wish to add functionality.
- src/main/java/chocopy/common/analysis/types/*.java: This package contains a hierarchy of classes that are used in the starter code to build a type environment. You may want to add more classes to this hierarchy; refer to Section 5.1 for details.
- src/main/java/chocopy/pa2/DeclarationAnalyzer.java: This class implements a simple pass over the AST that analyzes global variable declarations and builds a symbol table. You can modify this class to add the remaining semantic checks and analyze more declarations. This is simply a suggested starting point and you are free to discard this class if you do not want to use it.
- src/main/java/chocopy/pa2/TypeChecker.java: This class implements a simple pass over the AST that assigns types to expressions when given a typing environment in the form of a symbol table. You can modify this class to add the remaining typing rules. This is simply a suggested starting point and you are free to discard this class if you do not want to use it.
- src/test/data/pa2: This directory contains ChocoPy programs for testing your semantic analysis.
  - /sample/*.py - Sample test programs covering a variety of semantic and typing rules of the ChocoPy language that you need to handle in this assignment.
  - /sample/*.py.ast - ASTs corresponding to the same test programs in JSON format. These will be the inputs to your semantic analysis when testing.
  - /sample/*.py.ast.typed - Typed ASTs corresponding to the test programs. These are the expected outputs of your semantic analysis.
  - /student_contributed/good.py - A test program that is semantically valid and well-typed. You have to modify this file to create a program that covers as many typing rules in your implementation as possible.
  - /student_contributed/bad_types.py - A test program that contains type checking errors. You have to modify this file to cover as many type errors as your implementation

supports. You will also use excerpts from this file to explain type-error recovery in the writeup (ref. Section 4.4). Your type-error recovery mechanism should be consistent with the rules defined in Section 4.3.2.

  * /student_contributed/bad_semantic.py - A test program that contains multiple violations of the checks listed in Section 4.2. You have to modify this file such that your analyzer reports multiple distinct violations. For these rules only, the error recovery mechanism in your implementation need not match that of the reference compiler.

- target/: The target directory will be created and populated after running mvn clean package. It contains automatically generated files that you should not modify by hand. This directory will be deleted before your submission.

- chocopy-ref.jar: A reference implementation of the ChocoPy compiler, provided by the instructors.

- README.md: You will have to modify this file with a writeup.

4 Assignment goals

The objective of this assignment is to build a semantic analysis for ChocoPy that takes as input a ChocoPy abstract syntax tree (AST) in JSON format, and annotates expressions in the AST with inferred types.

To get the AST in the first place, you can run an input ChocoPy program through the staff-provided parser. The output of this parser then forms the input to the semantic analysis phase. This two-step procedure can be performed by performing the following two commands (on a single line each):

1.
```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=r \
      --out <ast_json_file> <chocopy_input_file>
```

2.
```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=.s \
      --out <typed_ast_json_file> <ast_json_file>
```

where <chocopy_input_file> is a ChocoPy program (usually with a .py extension), <ast_json_file> is the parsed AST in JSON format (usually with a .ast extension), and <typed_ast_json_file> is the type-annotated AST in JSON format (usually with a .ast.typed extension).

To simplify development, you can also combine the above two commands into a single command that pipes the output of the first phase into the input of the second phase, without creating an AST JSON file. The combined command (which is equivalent to running the above two commands) is as follows:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=rs \
      --out <typed_ast_json_file> <chocopy_input_file>
```

where <chocopy_input_file> is a ChocoPy program. In all cases, you can omit the --out <file> arguments to have the command print the JSON to standard output.

Reference solution You can run the reference compiler's second stage in the following way:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=.r \
      <ast_json_file>
```

You can also chain the parser and semantic analysis of the reference compiler together:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=rr \
      <chocopy_input_file>
```

4.1 Input/output specification

The input to the semantic analysis phase will be an AST in JSON format. The output of the semantic analysis phase is also expected to be in JSON format. In the absence of semantic errors, the output should be the same AST with all expressions annotated with value-types. In case of a semantic error, the output should contain a list of semantic errors along with source locations corresponding to the AST nodes that contain the semantic errors.

The interface to your semantic analysis will be the class `chocopy.pa2.StudentAnalysis`. In particular, the commands listed in this document will invoke the static method `StudentAnalysis.process(Program input, boolean debug)`, which returns its output as a `Program`. The flag `debug` is set to true if the `--debug` option is given on the command-line when invoking the compiler. You may use this flag to conditionally print debugging messages. The flag will be unset during autograding.

The starter code contains a bare-bones implementation of `StudentAnalysis.process()`, which performs very limited semantic and type checking. You are free to change the contents of this method in any way you like. Section 5.1 describes the starter code in more detail.

The Sections 4.1.1 and 4.1.2 describe what a JSON format is and what kind of JSON objects the AST nodes contain. If you are familiar with these concepts from PA1, you can skip to Section 4.1.3.

4.1.1 JSON format

JSON is a notation for representing a tree of objects. A JSON object is a set of key-value pairs called *properties*, represented using curly braces:

```
{ <key1>: <value1>, <key2>: <value2>, ... }.
```

For example,

```
{ "product" : "iPad Pro", "company": "Apple", "year": 2016,
  "released": true }.
```

Keys are always strings delimited by double quotes; the values can be strings, integers, booleans (`true`/`false`), the value `null`, other JSON objects, or JSON arrays. Arrays are represented as a list of values delimited by square brackets: `[<value1>, <value2>, ...]`. A complete specification for JSON may be found at `https://json.org`.

In our AST representation, we denote each AST node using a JSON object. Such a JSON object has a particular *kind*, which specifies what keys the object must contain and what types the corresponding values will take. For example, the `Identifier` kind specifies one property, with a key called `name`, whose value must be a string corresponding to the name of the identifier. Similarly, the `UnaryExpr` kind specifies two properties: a string-valued `operator`, and a property with key operand whose value is of kind Expr. Kinds can extend other kinds, including the properties specified by the extended kind as a subset of their own properties. This mirrors the subtyping relations between the AST node types they represent. Both Identifier and UnaryExpr extend kind Expr, and therefore JSON objects of these kinds may appear as values whenever an object of kind Expr is expected. All kinds in our AST directly or indirectly extend the Node kind, which specifies two properties: (1) a string-valued property called kind that simply contains the name of the node's kind and (2) location, an array of integers. The following is a sample JSON representation of the AST corresponding to the unary expression (-foo):

```
{
  "kind": "UnaryExpr",
  "operator": "-",
  "operand": {
    "kind": "Identifier",
    "name": "foo",
    "location" : [ 1, 3, 1, 5 ]
  },
  "location" : [ 1, 2, 1, 5 ]
}
```

The location array always contains four integers and describes source code location information for the corresponding AST node: (1) the line number of the leftmost character, (2) the column number of the leftmost character, (3) the line number of the rightmost character, and (4) the column number of the rightmost character.

4.1.2 AST-node kinds

For this assignment, we list the set of all kinds required to serialize ASTs in Figure 1. We use the syntax **kind** K {...} to define a kind and **kind** K **extends** S {...} to define a kind K that extends kind S. Properties are defined as <k>:<v> where <k> is the name of the key and <v> is the type of the value. Value types are one of **string**, **int**, **bool**, a JSON object of kind K, or a JSON array of type t represented as [t]. Properties that may contain **null** values are suffixed with a question mark.

When provided with a syntactically valid ChocoPy program, the output of the parser should be a JSON object of kind Program. Most AST-node kinds correspond directly to production rules in the grammar. A notable exception is the IfStmt kind, which only contains one elseBody even though the grammar allows a sequence of elif statements. The if-elif-else form is syntactic sugar; the parser de-sugars elifs as an elseBody with exactly one IfStmt in its body. Refer to the language reference manual for an example of this equivalence.

4.1.3 Differences from Programming Assignment 1

The JSON object kinds listed in Figure 1 mostly resemble those specified in PA1, where you were expected to develop a ChocoPy parser. For those familiar with the JSON nodes in PA1, here are the key changes in PA2:

- New object kinds Type, ValueType, ClassValueType, ListValueType, and FuncType have been added. These will be used to store information about types of program expressions inferred

```
kind Node {
  kind: string,
  location: [int],
  errorMsg: String?
}

kind Program extends Node {
    declarations: [Declaration],
    statements: [Stmt]
}

kind Declaration extends Node { }

kind ClassDef extends Declaration {
    name: Identifier,
    superClass: Identifier,
    declarations: [Declaration]
}

kind FuncDef extends Declaration {
    name: Identifier,
    params: [TypedVar],
    returnType: TypeAnnotation,
    declarations: [Declaration],
    statements: [Stmt]
}

kind VarDef extends Declaration {
    var: TypedVar,
    value: Literal
}

kind GlobalDecl
  extends Declaration {
    variable: Identifier
}

kind NonlocalDecl
  extends Declaration {
    variable: Identifier
}

kind TypedVar extends Node {
    identifier: Identifier,
    type: TypeAnnotation
}

kind TypeAnnotation
  extends Node { }

kind ClassType extends TypeAnnotation {
    className: string
}

kind ListType extends TypeAnnotation {
    elementType: TypeAnnotation
}

kind Type {
  kind: string
}
```

```
kind ValueType extends Type { }

kind ClassValueType extends ValueType {
    className: string
}

kind ListValueType extends ValueType {
    elementType: ValueType
}

kind FuncType extends Type {
    parameters: [ValueType]
    returnType: ValueType
}

kind Stmt extends Node { }

kind ExprStmt extends Stmt {
    expr: Expr
}

kind ReturnStmt extends Stmt {
    value: Expr?
}

kind AssignStmt extends Stmt {
    targets: [Expr],
    value: Expr
}

kind IfStmt extends Stmt {
    condition: Expr,
    thenBody: [Stmt],
    elseBody: [Stmt]
}

kind WhileStmt extends Stmt {
    condition: Expr,
    body: [Stmt]
}

kind ForStmt extends Stmt {
    identifier: Identifier,
    iterable: Expr,
    body: [Stmt]
}

kind Expr extends Node {
    inferredType: Type?
}

kind Identifier extends Expr {
    name: string
}

kind BinaryExpr extends Expr {
    left: Expr,
    operator: string,
    right: Expr
}
```

```
kind CompareExpr extends Expr {
     operands: [Expr],
     operators: [string]
}

kind UnaryExpr extends Expr {
    operator: string,
    operand: Expr
}

kind IfExpr extends Expr {
    condition: Expr,
    thenExpr: Expr,
    elseExpr: Expr
}

kind CallExpr extends Expr {
    function: Identifier,
    args: [Expr]
}

kind MethodCallExpr extends Expr {
    method: MemberExpr,
    args: [Expr]
}

kind IndexExpr extends Expr {
    list: Expr,
    index: Expr
}

kind MemberExpr extends Expr {
    object: Expr,
    member: Identifier
}

kind ListExpr extends Expr {
    elements: [Expr]
}

kind Literal extends Expr { }

kind NoneLiteral extends Literal { }

kind StringLiteral extends Literal {
    value: string
}

kind IntegerLiteral extends Literal {
    value: int
}

kind BooleanLiteral extends Literal {
    value: bool
}

kind Errors extends Node {
    errors: [Error]
}

kind CompilerError extends Node { }
```

Figure 1: Kinds of JSON objects corresponding to AST nodes

after type checking, for use by the code-generation phase. ValueType and its two sub-kinds are analogous to TypeAnnotation and its two subtypes. FuncType carries information about the formal parameter types and return type of a function. In ChocoPy, it does not represent the type of a first-class value, but may be useful during code generation for determining coercions needed to pass arguments. The difference between TypeAnnotation and ValueType is that the latter does not extend Node; therefore, ValueType objects do not have a locations property. This should make sense since the types assigned during semantic analysis are not actually present in the source code.

- The kind Expr has a new property: inferredType, which may be null. In the ASTs produced by the parser, this property is null for every expression. The semantic analysis infers types *for every program expression that can evaluate to a value.* Specifically, the inferredType property will remain null *only* for Identifier objects that appear directly in the properties of FuncDef, ClassDef, TypedVar, GlobalDecl, NonlocalDecl, and MemberExpr.

- The Node kind has a new property: errorMsg. In the ASTs produced by the parser, this property is null for every node. The errorMsg will be non-null for a Node if there was an error in checking that node. For a well-typed ChocoPy program, the errorMsg property will be null for every node in the output of the semantic analysis phase. It is acceptable for null-valued properties to simply be omitted in a JSON representation.

4.2 Semantic checks

This section enumerates a list of semantic rules that your analysis should check. Violations of these rules leads to a semantic error. For each semantic rule, we list the name of one or more test files, provided in the src/test/data/pa2/sample directory, which contain a program that violates only this semantic rule in one or more lines.

1. Identifiers must not be redefined in the same scope. See bad_duplicate_global.py, bad_duplicate_local.py, and bad_duplicate_class.py.

2. Variables and functions may not shadow class names. See bad_shadow_local.py.

3. Nonlocal and global declarations must only refer to local variables declared in outer scope or global variables respectively. See bad_nonlocal_global.py.

4. In class definitions, the declared super-class must either be object or be a user-defined class that has been defined previously in the program. See bad_class_super.py.

5. In class definitions, attributes must not override definitions of attributes and methods inherited from base classes. Further, attributes may not be overridden by methods of the same name in sub-classes. See bad_class_attr.py.

6. In class definitions, methods must specify at least one formal parameter, and the first parameter must be of the same type as the enclosing class. See bad_class_method.py.

7. In class definitions, methods can only override methods of the same name, inherited from base classes, as long as the signatures match (with the exception of the first param). See bad_class_method_override.py and bad_class_method_override_attr.py.

8. In function and method bodies, there must be no assignment to variables (nonlocal or global) whose binding is inherited implicitly (i.e., without an explicit nonlocal or global declaration). See `bad_local_assign.py`.

9. Functions or methods that return special types must have an explicit return statement along all paths. See `bad_return_missing.py`.

10. Return statements must not appear at the top level outside function or method bodies. See `bad_return_top.py`.

11. Type annotations should not refer to class names that are not defined. See `bad_type_annotation.py`.

4.3 Error handling

The semantic analysis phase detects two types of semantic errors: violations of the semantic rules listed in Section 4.2, and type-checking errors, which are violations of the typing rules listed in the ChocoPy language reference manual. If the input program contains semantic errors other than typing errors, it need not (but may) report type-checking errors.

4.3.1 Reporting semantic errors

Your implementation should be able to recover from a semantic error and continue analyzing the rest of the program in order to report as many semantic errors as possible. Unlike recovering from parse errors, the error recovery in the semantic analysis is much simpler to perform, since you can simply report an error and continue analyzing the rest of the AST.

For each semantic error, you must report the location of the error in source code using the source location information of an AST node. For rules 1–9 listed in Section 4.2, the semantic error should be reported at the site of the `Identifier` node corresponding to the variable, attribute, class, function, or method whose assignment, declaration, or definition (where applicable) violates a semantic rule. For rule 10, the error should be reported at the `ReturnStmt` node corresponding to the top-level return statement. For rule 11, the error should be reported at the `ClassType` node corresponding to the invalid type annotation. Consult the test outputs or the output of the reference implementation for examples.

The autograder will use the following rule to evaluate your implementation on inputs that contain a semantic error: a test passes only if all the semantic errors reported by the reference implementation are also reported by your submitted implementation. In other words, the semantic errors reported by the reference implementation should be a *subset* of the errors reported by your implementation. Semantic errors will be compared for equality of the error message and left source location for every error node. In case of multiple semantic errors, the order of reported errors does not matter.

When more than one of the rules listed in Section 4.2 is violated for the same program class, method, function, or variable, then the error recovery may depend on the order in which a compiler performs various checks. For example, if there are duplicate classes with the same name `A`, and then a class `B` that is declared as a subclass of `A` defines a method, it is undefined which methods `A` must inherit for the purpose of semantic error reporting. For this reason, you will not be graded on inputs that contain *conflicting* violation of the rules defined in Section 4.2; that is, when the same program entity is involved in multiple distinct violations. The rule of thumb when deciding what constitutes a conflicting pair of errors is as follows: if the reference compiler reports multiple semantic errors, and if fixing one of these errors also implicitly fixes another error or causes more errors to be introduced, then the original errors were *conflicting*. Note that this exception does not apply to type-checking errors, whose error recovery mechanism is fully specified in the next section.

4.3.2 Reporting type checking errors

Nodes in which a semantic error is discovered will be marked (redundantly) with the error message in the `errorMsg` field. This serves the additional purpose of suppressing further messages on a node if there is already one reported.

For ill-typed *expressions*, we would like to infer the *most specific type* that is appropriate for the expression. For example, if there is a problem when type checking a `BinaryExpr` containing the `==` operator, say because the types of its operands do not match, then we insert an error message and infer the class type `bool` for this expression, since we know that comparison operators always result in a boolean. Similarly, if a `CallExpr` fails to type check because one of its arguments does not conform to the declared type of the corresponding formal parameter, then we insert an error message and infer the return type of the function for the entire call expression. However, if we fail to type check a `CallExpr` because the identifier does not actually refer to a function in the current scope, then we have no way to infer any specific type for this expression; therefore, we insert an error message and simply infer the type `object` for the `CallExpr` node.

On test inputs that do not contain semantic errors, the autograder will evaluate your implementation by comparing the type-annotated ASTs output by your implementation with the type-annotated ASTs output by the reference implementation for equivalence: the ASTs represented by the JSON must be exactly the same. Only error messages matter when the program has semantic errors. Therefore, it is essential that your implementation infers the same types and inserts at least the same error messages as the reference implementation. The tests provided in the `sample` directory cover all the type checking errors handled by the reference implementation. You can refer to the test outputs as a guide for determining the appropriate error messages and inferred types.

As a general rule of thumb: if, when implementing some typing rule, your analysis is unable to prove some premise, then you should attempt to infer the type of the ill-typed node by omitting the inconsistent premise. For example, the typing rule for selecting an element of a list is as follows:

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : \mathit{int} \end{array}}{O, M, C, R \vdash e_1[e_2] : T} \quad [\textsc{list-select}]$$

If say you find that the expression e_1 is of some type $[T]$, but e_2 does not have type `int`. In such case, you can still infer a type T for the list-select expression based on the conclusion *after inserting an appropriate error message*. However, if you find that the expression e_1 is not of a list type, then you do not have any T to assign to the list-select expression; therefore, you infer the type `object`. This rule of thumb can be applied deterministically for almost every typing rule. We next describe some notable subtleties.

An ambiguity arises when type checking the binary operator `+`, since the inferred type for a well-typed `+` expression is different depending on whether *both* of its operands are of type `int`, `str`, or list type. The rule of thumb does not provide a unique solution for when say one operand is an int and the other operand is a str. The analysis should handle ill-typed + expressions in the following way: if at least one operand has type int, then infer int; otherwise, infer object. In either case, an appropriate error message must be inserted at the ill-typed expression.

In general, if there is more than one premise of a typing rule that fails to be true, then the reported error message should correspond to the *topmost* premise that is false, according to the order of premises listed in the typing rules given in the language reference manual.

A special case must be made for type-checking (multi-) assignment statements. First, if any left-hand side is not a valid assignable location (e.g. it is not an assignable variable, object attribute, or list element), then the appropriate error message must be attached to the corresponding target expression itself (which can be an Identifier, MemberExpr, or IndexExpr respectively). Second, if there is a type-checking error due to the conformance of the inferred type of the right-hand-side with one or more targets, then the error message corresponding to the leftmost errorneous target should be attached to the entire AssignStmt node. For example, in the following program:

```
x:int = 1
y:bool = False
x = y = z = "Error"
```

Two error messages will be produced: (1) ”Not a variable: z” attached to the third target, which is the Identifier corresponding to z, and (2) an error that a string cannot be assigned to an integer, attached to the entire AssignStmt on line 3. Note that no error is printed corresponding to the incorrect assignment of a string to the bool variable y in this case.

To verify whether your error handling conforms to the errors reported by the reference implementation, simply test your implementation on the provided sample inputs:

```
    java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=.s \
        --dir src/test/data/pa2/sample --test
```

and look for the test inputs whose names start with the prefix "bad_".

4.4 Writeup

Before submitting your completed assignment, you must edit the README.md and provide the following information: (1) names of the team members who completed the assignment, (2) acknowledgements for any collaboration or outside help received, and (3) how many late hours have been consumed (refer to the course website for grading policy).

Further, you must answer the following questions in your write-up by editing the README.md file (one or two paragraphs per question is fine):

1. How many passes does your semantic analysis perform over the AST? List the names of these passes with their class names and briefly explain the purpose of each pass.

2. What was the hardest component to implement in this assignment? Why was it challenging?

3. When type checking ill-typed expressions, why is it important to recover by inferring the most specific type? What is the problem if we simply infer the type object for every ill-typed expression? Explain your answer with the help of examples in the student_contributed/bad_types.py test.

5 Implementation Notes

5.1 Classes in the starter code

The starter code provided to you provides a basic semantic analysis that can partially handle global variable declarations, integer literals, and some binary operators. **You are not required to use any of these classes for your assignment**. The assignment specification is simply that you implement the `StudentAnalysis.process()` method to produce the expected JSON output. This section describes the classes used in the starter code in case you choose to use them in your assignment.

5.1.1 `StudentAnalysis`

The starter code performs two passes over the input AST. The first pass, named `DeclarationAnalyzer`, collects global variable declarations into a symbol table. The second pass, named `TypeChecker`, uses the symbol table to type check expressions. The symbol table is implemented in class `SymbolTable`.

5.1.2 `SymbolTable`

The `SymbolTable` maps string-valued names to objects of some generic type `T`. The mapped type is deliberately kept generic, since this implementation will also be useful in the subsequent code-generation assignment.

The `SymbolTable` is a useful data structure to manage nested scopes: symbol tables can be constructed with an optional reference to a *parent* symbol table, which is used to delegate a lookup in case of missing entries. The symbol table corresponding to the outermost scope has no parent.

5.1.3 `Type` and `ValueType`

The package `chocopy.common.analysis.types` contains a hierarchy of classes that may be useful for semantic analysis and type checking. The root of this class hierarchy is the abstract class `Type`, which is the type of objects stored by the symbol table for semantic analysis and type checking. You can think of this as the type of objects that can appear in the typing environments. The class `Type` contains value types for predefined classes: `object`, `str`, `int`, and `bool`.

The starter code provides a special abstract sub-class of `Type` called `ValueType`. Value types represent types that can be assigned to variables and any program expression that evaluates to a value. Therefore, `ValueType` has two concrete sub-classes corresponding to the two types of values in ChocoPy: `ClassValueType` and `ListValueType`. These two classes have fields that are very similar to the AST-node classes corresponding to variable/attribute type annotations: `ClassType` and `ListType`. The class `ValueType` provides a static method to convert these AST type annotations into value-types for type checking: `public static ValueType annotationToValueType(TypeAnnotation annotation)`.

The `Type` class is also used for the field `inferredType` in the AST-node class `Expr`. If you choose to use the classes provided by the starter code, then you will want to ensure that in the absence of semantic errors, this field is non-`null` for all expressions that may evaluate to values (ref. Section 4.1.3 for a list of nodes where this field may remain `null`). The symbol table may contain other classes of objects that do not necessarily correspond to types of program variables: especially functions and classes. The typing environment can contain type information about variables as well as functions. You may also want to use the symbol-table data structure to store information about class and method definitions. If you wish to use the symbol table provided in the starter code, you may need to create more subclasses of `Type` to accommodate these classes of object.

5.1.4 `NodeAnalyzer` and AST traversal

The `NodeAnalyzer` interface provides a convenient mechanism to separate logic for handling different AST nodes of distinct concrete classes. The `Node` class (which is the root of the AST hierarchy) defines a method `dispatch(NodeAnalyzer<T>)`. When you invoke `node.dispatch(a)` on an AST node whose concrete class is `N`, it will in turn invoke the overloaded `a.analyze(N)` method and return its value. This is done by overriding `Node#dispatch()` in every single concrete AST node class. The class `AbstractNodeAnalyzer` implements this interface with a dummy method for every AST node type that simply returns `null` values. A real analysis will typically extend `AbstractNodeAnalyzer` and override methods corresponding to the nodes that are relevant to that analysis.

This pattern is very useful when processing AST node variables of an abstract class. Take a look at the code of `TypeChecker` in the starter code for a sample use of this pattern. In the method `analyze(BinaryExpr)`, the expression is type checked by first dispatching the type checker on each of its operands, which are of abstract class `Expr`; the logic for each concrete expression class is separated into methods `analyze(IntegerLiteral)`, `analyze(Identifier)`, `analyze(BinaryExpr)`, and so on. Invoking the `dispatch` method on an `Expr` instance leads to the invocation of an appropriate overloaded method in `TypeChecker`.

The return value of the `analyze` methods varies depending on the nature of the analysis. In `TypeChecker`, the analysis returns `ValueType` objects when analyzing expressions and `null` when analyzing nodes that do not evaluate to a value, such as statements and `Program`. In `DeclarationAnalyzer`, the analysis returns `Type` objects when analyzing declarations, in order to store the results of the analysis into a symbol table, and `null` otherwise.

5.1.5 `Errors` and `CompilerError`

The starter code performs error reporting and recovery using two classes. An instance of the `Errors` class is provided to every pass over the AST that checks for various semantic rules; errors are reported by simply adding instances of `CompilerError` to the `Errors` object.

The constructor for class `CompilerError` takes an AST node as its first argument and a message as its second argument. The `locations` property for the `CompilerError` JSON object is populated by copying the source locations for the AST node provided as the first argument.

5.2 Recommendations

This assignment is likely much larger than the previous assignment. However, this assignment also provides much more room for custom design decisions, enabling a flexible implementation strategy. We have provided some directions in the form of a skeleton implementation in the starter code. However, what you end up doing is largely up to you.

Tree traversal This algorithmic style described in Section 5.1.4—a recursive traversal of a complex tree structure—is very important, because it is a very natural way to structure many computations on ASTs. A semantic analysis usually requires multiple passes over the AST to perform various tasks.

Type hierarchy You will probably need to build a data structure that stores the inheritance relationships between classes, both predefined and user-defined. This will be essential in answering queries of type conformance (i.e., subtyping) as well as in computing joins (i.e., least upper bounds). You may want to consider how the various semantic rules of ChocoPy constrain the inheritance graph, in order to implement these operations efficiently.

Type checking You can save a lot of development effort by precisely identifying patterns where similar typing (sub-)rules repeatedly apply and implementing utility methods that can be re-used across different typing rules.

5.3 Web IDE (Experimental)

An experimental web-based ChocoPy IDE is provided in the `web` directory. You need Python 3+ to run this IDE. For PA2, the purpose of the IDE is to able to visualize syntactic and semantic errors inline with the code itself. It is intended to serve as a debugging aid while you develop your compiler, as an alternative to inspecting your JSON output manually.

To run the web-based IDE: open a terminal, `cd` to the `web` directory of the assignment package, and run the following command:

```
python -m WebCompiler 8000
```

Make sure that `python` corresponds to Python 3+. On some installations, you may have to specify the command `python3` instead. This command starts a local web server at port 8000. Leave this process running while you use your Web IDE.

Now, go to a Web browser and navigate to `http://localhost:8000` (replace `8000` if you used a different port). You should see a web page with a code editor, and options to select your own compiler stages (i.e., the student's version) or the reference compiler's stages. Enter a ChocoPy program and click the button to run static checks. The web editor will parse the JSON produced by the compiler and indicate if the ChocoPy program is valid. In case of static errors, hover your mouse over the red cross in the left margin to read the corresponding error message. The source locations associated with compiler errors in the JSON are used to highlight the errornous fragments of code in the editor.

For the Web IDE to work, you must have the web server up and running, and you must have built the JARs using `mvn clean package`. The PA2 starter code does not bundle the student's parser. If you wish to use your own parser instead of the reference parser, make sure to copy the files `ChocoPy.jflex`, `ChocoPy.cup`, and `StudentParser.java` from your PA1 repository into the appropriate directories corresponding to package `pa1`.

Support The Web IDE is an experimental feature provided on a best-effort basis. While the instructors appreciate bug reports for the IDE on Piazza, it is not an integral part of your assignment and as such we cannot guarantee that issues with the IDE (if any) will be addressed by the assignment submission deadline. It is provided only with the hope that it may be useful in avoiding some of the tedious aspects of inspecting the JSON manually. Additionally, you are welcome to send us feature requests or to share improvements to the UI on Piazza (you can edit `web/index.html`).

6 Submission

Submitting your completed assignment requires the following steps:

- Run `mvn clean` to rid your directory of any unnecessary files.
- Add and commit all your progress and push changes to the repository. Run `git commit` followed by `git push origin` to achieve this.
- Tag the desired commit with a tag having the form `proj2-`*n*, where *n* is an integer. If the desired commit is the latest one, the command is simply `git tag proj2-`*n*. Otherwise, it is `git tag proj2-`*n* `<commit-id>` where `<commit-id>` is the commit you want to tag as your submission.
- Push the tag using `git push --tags` or `git push origin proj2-`*n*.
- The last such tag you push (not necessarily the highest numbered) is taken to be your submission

Warning: Do not use the autograder to test and debug your code. We will only run the autograder a limited number of times, and the strategy of making lots of submits trying to get one that passes everything will not be very productive.

7 Grading (100 points)

Grading for the project comes in two pieces: project assessment (25 points, with each team member getting the same score) and team assessment (5 points, individual scores). In a reasonably functioning team, everyone gets full marks. The assessment part is scaled down from 100 points, awarded as follows:

- 82 points for autograder tests (1 point per correct test / 82 tests). These include the tests provided to you in the `samples` directory as well as unseen tests.
- 6 points for the README
  - 2 points for each of the four questions listed in Section 4.4.
  - Only 1 point will be awarded for questions with incomplete or vague responses.
- 6 points for tests written in `src/test/pa2/data/student_contributed`.
  - 2 points for each of `good.py`, `bad_type.py`, and `bad_semantic.py`, for covering a range of typing rules and semantic checks. Only exercise the rules that your implementation handles!
  - Only 1 point will be awarded for a test file with narrow coverage.
  - 0 points will be awarded per test file if there was little to no effort in writing custom tests.
- 6 points code cleanliness and structure.
  - 6 points for: clear naming for variables and other symbols, consistent spacing and punctuation conventions, reasonable modularization of functions and other components, code comments explaining non-obvious logic
  - 3 points for: effort made but imprecise or lacking in quality.
  - 0 points for: little to no effort to organize and document code.

7.1 Extra credit: Bug reports

The reference implementation possibly contains some bugs. If you find a bug, report it by making a post on Piazza with a sample input program and describe how the expected output should differ. The first student/team to report a bug gets extra credit (2 points per unique bug with a maximum of 10 extra credits per team).

Bugs in the reference implementation are defined as (1) unexpected exceptions being reported or (2) violations of the specifications of the assignment or the specifications of the ChocoPy manual, which would lead to incorrect results. Minor mistakes in the ChocoPy manual or this document itself are not considered bugs in the reference implementation, though we would appreciate any such feedback.

The decision on whether to accept a bug report as valid and distinct from previous bug reports is at the discretion of the instructors.