CS 444
Assignment 4

Mark Rada (`marada`)
Riel Fuller (`r3fuller`)
Michael Forsyth (`mfforsyt`)

This document outlines the design of name resolution, type checking, and other static analysis phases of the Joos 1W language as implemented in our compiler. It also discusses some challenges faced while implementing these phases.

Design

All components of the compiler implemented for name resolution, type checking, and static analysis are centered around the AST. Either by transforming the AST, replacing portions of the AST with more convenient data structures, or by appending functionality to classes which are already part of the AST. After parsing was completed, the compiler had an AST for each compilation unit (file) which was named on the command line. We then begin building up entity structures for each compilation unit in order to simplify environment building and name resolution.

sentence fragment

Entities

The first step in building the type environment was to create an `Entity` class to represent each type of declared Java entity. We define an entity in the same way that the JLS does:

> A declared entity is a **package**, **class** type, **interface** type, member (class,interface, **field**, or **method**) of a reference type, **parameter** (to a method, constructor, or exception handler), or **local variable**.

For the entities which Joos supports; `Package`, `Class`, `Interface`, `Field`, `Method`, `Parameter`, and `LocalVariable`; we have created classes that make accessing the metadata of the entity easier and more efficient than repeatedly navigating an AST, and share a common superclass of `Entity`.

The `Compiler` object keeps a reference to each `Class` or `Interface` compilation unit that is being declared, as well as the root pseudo `Package`.

Package Hierarchy

The root package is only a pseudo package; it has no name, and is not equivalent to the unnamed default package specified by the JLS. The root pseudo package exists only to provide

a convenient container for packages which do not have a parent package. For instance, the `java.lang.Object` class declares that it belongs to package `java.lang`; which would create a `Package` named `java` that is a member of the root pseudo package. `java.lang` would also create a `Package` named `lang` which is a member of the `java` package. Finally, the `Class` named `Object` whose compilation unit declared the package, will be added as a member of the `lang` package.

Each `Package`, `Class`, and `Interface` will become a member of some `Package`, and all be part of the same namespace. In this manner, we can iterate over each compilation unit and, if the compilation unit declares a package, we can create the required hierarchy of packages and then add the compilation unit to the package.

If a compilation unit does not declare a package, it will be added to the special default package, as specified in the JLS. Though the JLS allows for multiple default packages, we found it necessary to have only a single default package in order to pass Marmoset tests.

If a package declaration names a package which clashes with an already declared compilation unit, then an error is printed and compilation is aborted. A similar result occurs if a compilation unit's fully qualified name would be the same as the fully qualified name of a package. For instance, if a compilation unit declared `package java.lang.Object;` as its package, there would be an error because a compilation unit, `Class java.lang.Object`, already exists.

Type Environment

To complete the type environment, for each compilation unit, we parsed the `import` statements from each compilation unit and searched through the package hierarchy starting from the root pseudo package. Since `import` is required to use fully qualified names, we know that if a single-type import names a `Package` then that is an error, and if a package import names a `Class` or `Interface`, then it is also an error. If nothing is found in the package hierarchy, then that is also an error.

For single-type imports, we keep an ordered list of compilation unit entities that were named by single-type import statements. Duplicate single type imports are considered errors and will cause compilation to abort with an error message. Similarly, single type imports where the simple name of the compilation clashes with a previous single type import, or the simple name of the compilation unit, will also be treated as an error in the source code.

For package imports, we keep a list of `Package` entities that were named by package import statements. The implicit `java.lang.*` package import is done first, and then any other package imports follow.

Each compilation unit then gets its own list of single type imports and package imports, and shares a reference to the root pseudo package. This tuple of information forms the type environment for a compilation unit, and is used to resolve names which must name a type. It is also used to help disambiguate ambiguous names during name resolution and type checking.

Building The Entity Hierarchy Skeleton

Once the package hierarchy and type environment are built, we proceed with building the entity hierarchy for each compilation unit, which will replace parts of the AST.

For each `Class`, we parse the `extends` and `implements` clauses of the declaration header, and link the names using the type environment. We then proceed to build an entity for each `Constructor`, `Method`, and `Field` declared in the body of the `Class`, maintaining an ordered list for each type of entity.

For each class member, we resolve the return type to either a basic type, an array of a basic type, or we must search the type environment of the compilation unit to resolve the named type or named type array. If the type name cannot resolve then we print an appropriate error message and abort compilation. One exception to this process is that we treat the return type of a constructor to be `void`, as that is equivalent to the semantics required by the JLS and allows more code reuse between entities.

`Interface` declarations follow the same process for building their entity hierarchy, with the obvious exception that there will not be any `Field` or `Constructor` declarations for an `Interface`.

Then, for each `FormalParameter` of a `Method` or `Constructor`, we follow the same process for resolving the type. Resolving the types of local variable declarations and allocations is deferred until after hierarchy checking.

Hierarchy Checking

We perform hierarchy checking as soon as enough of the hierarchy is constructed, which is described in the previous section.

Linking classes and interfaces in the hierarchy occurs in essentially two passes. In the first pass, we link superclasses and superinterfaces and check simple constraints. One of the most important checks is the circularity check, which is done in the obvious way by recursively traversing up the class hierarchy. This check ensures that the second pass of hierarchy checking terminates.

In the second pass, we link members to their class or interface. We also link overridden methods to their parent and compute the contains set (called `all_methods` and `all_fields`) for each class and interface. To do this, we traverse the interface hierarchy, and then the class hierarchy in breadth-first order: we sort by an attribute `depth`, which is 0 for `java.lang.Object` and `superclass.depth + 1` (resp. `max superinterfaces.depth + 1`) for every other class (interface). The contains set of a class or interface (a simple array of `Method` or `Field` objects) is actually computed by taking an outer join of the class' own members with the contains set of its parent, joining by the signature of the member. Visiting each class / interface ordered by depth means that the contains sets of the superclass and superinterfaces are guaranteed to be complete.

Finally, as part of the second pass, non-abstract classes check their contains set to ensure that they have no abstract methods and all their interfaces are implemented. Other checks are done along the way in the obvious fashion.

Completing The Entity Hierarchy

Once hierarchy checking is completed, we build entities for `LocalVariableDeclaration`s, resolving their type using the type environment, and resolve types for allocation expressions. Though there are other parts of an expression or expression statement which must refer to a type, we defer resolving those types until type checking.

The decision to separate some of the type resolution phase was the result of how the Marmoset test suites are split up between assignment 3 and assignment, and our desire to use the Marmoset test suite for additional regression tests.

Type Checking

Type checking is performed once hierarchy checking is completed and it is possible to resolve the names of identifiers used in `Method`/`Constructor` bodies and `Field` initializers to be resolved correctly. We also use this compilation phase to perform constant folding and the conservative flow analysis.

Type checking is implemented recursively for every AST class. The leaves of the AST are `Token` objects, and serve as the base case for the recursive checking. A `Token` in the AST will always pass type checking because it is either impossible to define type checking for a token, as is the case for certain keywords which remain in the AST; or the token is trivially type correct, as will be the case for any literals.

Type checking starts with the compilation units, which will then ask their members to type check, and then each member will ask their body to type check. For convenience of propagating the type checking context, we treat `Field` initializers as if they were wrapped in a block.

The type checking context, which is the return type for the member being checked, the type environment for the compilation unit, and an ordered list of `LocalVariable`s and `FormalParameter`s defined for a given block, is propagated through the AST by appending the information to the `Block` AST node, and giving AST node a method for finding their direct parent `Block`.

To simplify the behaviour of scope information in a `Block`, we perform various transformations on the AST as a pre-type checking procedure which we hope will also simplify code generation.

AST Transformations

Due to the limitations of our LR(1) grammar, an AST may have some peculiarities in its structure. This was necessary for creating an unambiguous and more simple grammar, but makes post-parsing operations on the tree more difficult due the many different structures that are created to model very similar concepts.

To get around this obstacle, we perform simple transformations on each AST while it is being built, and then some more transformations afterward during the weeding of the ASTs.

`for-while` Transformation

The simplest transformations is transforming every instance of a `for` loop to a `while` loop.

The initialization of the `for` loop is transformed into a `Statement` (if one exists) and is placed as the first statement of a new `Block` which will wrap the new `while` loop. The `Block` to wrap the `Statement` and loop is meant to limit the scope of variable declared in the `for` initializer.

The update expression of the `for` loop is also transformed into a `Statement`, and is placed as the last statement of the `while` loop's body `Block`, allowing that update to occur at the end of each iteration of the loop, as expected.

The condition of the `for` loop is placed directly into the condition of the `while` loop to preserve behaviour. Since `while` loops require a condition, and `for` loops do not, we place the `true` condition into the `while` loop automatically if the `for` loop does not have an update clause.

The `Block` or `Statement` following the `for` loop is converted into a `Block` (if applicable), and has the initialization and update added to it (again, if applicable). This new block is used as the `while` loop's body.

Inner Block Scoping

For each `Block`, we maintain a list of all identifiers declared in that `Block`. However, this leads to an ordering problem, as we are making the assumption that each declaration in the `Block` occurs before the corresponding identifier is referenced within the `Block`.

To deal with this, we add more inner `Block`s to the `Block` in question. We iterate over each statement in the `Block`, and if we see a declaration occur after an expression statement, or a second declaration in the same `Block`, we wrap the declaration and all following statements in the block in a new sub-`Block`. This way, there is only ever one declaration per block (all others are inside inner blocks), and any identifier that is declared in one block will still be in scope only for the `Block` in which it is declared in the original source code.

Order of Operations

Since our grammar does not preserve all Java operator precedence rules, we need to restructure `SubExpression`s to follow order of operations. Which precedence rules are not reflected in your grammar?

Each valid Joos operation is given a priority, with multiplication, division, and modulus receiving highest priority (parentheses are taken care of by the grammar), and lazy-or having the lowest priority. We pass over each `SubExpression` from right to left, performing a simple rotation if the operator precedence requires it, and recursively checking precedence on the new subtree AST created.

After this pass, the `SubExpression` tree is in the correct shape to follow Java operator precedence.

Constant Folding

We fold constant expressions into a single constant as the final step in type checking an AST node (described below).

Each `SubExpression` in the AST, is either a `Term` containing some value, or two more `SubExpressions` with an `Infixop` operator in between.

Each `SubExpression` has a `literal_value` denoting the its literal value, if it has one according to the expression evaluation rules of Java. If the `SubExpression` on both sides of the operator would produce a constant expression then we can fold the `SubExpressions` into a `Term` representing a new constant expression. If either `SubExpression` is not a constant expression, then `literal_value` is denoted to us by the Ruby `nil` and constant folding is not performed for the AST node.

With constant folding casting expressions, we had to be careful to not throw away type information when folding numeric types, as Java behaviour expects overflow to be handled correctly for the result of an operation.

Forward Reference Checks

As part of type checking, we check forward references of field declarations and local variables. This is also part of the pre-processing phase of type checking, and part of type checking due to assignment specifications, deadlines, and our desire to use the Marmoset tests as additional regression tests.OK.

Due to our rescoping of `Block`s, forward references of local variables are handled by the normal name resolution code that is used for ambiguous names (described below). Checking that a local variable does not reference itself in its initializer is done now as well, even though it is part of assignment 4, as it did not fit in anywhere else.

Forward reference checks for `Field` initializers are done by structural recursion on the `Field` initializer's AST. For every field reference that belongs to the same class and has the same staticness as the field, we check whether or not it is the same field or if the named field appears later in the class declaration.

One tricky case is when the initialized field is the target of an assignment. We handle this by checking a flag that is set on the left branch of an assignment, and cleared for any sub-expression that is not a simple identifier.

The Type Checking Algorithm

The `type_check` method is implemented for each type of AST node which is part of a `Method` block or `Field` initializer. Each type of AST node implements the rules which are relevant to the node. Each AST class splits up `type_check` into four steps:

1. Name resolution
2. Type resolution
3. Type checking
4. Constant folding

The name resolution step only applies to nodes which might have a name to resolve. For example, an `Identifier` must resolve itself, so must a `Selector` naming a `Method`, `Field`, or performing an array access. In each case, name resolution must first determine the context which it can use to search for the name. This is done by navigating the AST to find the previous name in the expression. If there is no previous name, then we use the rules for resolving ambiguous names. In this case, we ask the owning `Block` for context information as described above.

If there was a previous name resolved for the expression, then the entire `type_check` phase is done for that identifier and we can ask for the `type` of that identifier. Using the `type` we can search the appropriate namespace to resolve the name.

If the previous identifier named a `Package` then we will search the type hierarchy rooted at the named `Package`. If the previous identifier named a `Class`, then we search static `Field`s and `Method`s, depending on if the identifier belongs to a field access or method call. If the

previous identifier names some other entity, then we search the non-static methods and fields as appropriate.

If the name cannot be resolved, then an appropriate error is printed and compilation is aborted. However, if the name is resolved, then we cache the `Entity` to help with code generation.

Once name resolution is complete, the expected type of the AST node can be resolved. This does not actually check that the children nodes of the AST are type correct, only what the type of the AST node should be. This was done to simplify some of the type checking rules, such as those for infix operators. In the case of a `SubExpression`, we can look at the operator to determine the expected type for the overall `SubExpression`. The only exception in this case is the `+` operator, which has to handle string concatenation. This step of type checking simply follows the type rules as outlined in lecture and the JLS.

After the expected type for an AST node is resolved, the actual check to see if the types of the children AST nodes are in concert with the expected type as defined by the JLS and assignment specification.

A notable challenge for checking types was determining if types could be converted for assignment and comparison. In order to check assignability for references, we implemented the notion of `ancestors` for a type. The `ancestors` of a type is recursive set of superclasses and superinterfaces for compilation unit. If the right side of an assignability check has an ancestor which is the left side of the assignability check, then the assignment or comparison was allowed. Additionally, for casting and `instanceof`, we tested the reverse case of assignability if the first case failed. If the reverse case of assignability succeeded then the cast or `instanceof` check was allowed.

We also take this opportunity to mark a casting statement as either an up cast or a down cast, so that we know if a runtime casting check will be needed. If the reverse assignability check was needed, and succeeded, then we know that the runtime down casting check is required.

Checking assignability of basic types was strange due to how Joos and Java treat the `char` type as numeric with a size of 2 bytes. We thought that it would be assignment compatible with `short` types, but the JLS specifies otherwise and we handle it as a special case.

Once the body of a `Method` is type checked, we check that all `return` statements in the body are assignable to the type of the `Method`. Similarly, we check if the result of a `Field` initializer expression is assignable to the type of the `Field`.

The final step of type checking, was to try and fold any constant expressions, if that was relevant to the AST node. This could only be done safely after the rest of type checking, and is done as part of type checking because it was required for reachability analysis, and reachability analysis augments type checking for `Methods`.

Assignment Checking

As Joos does not allow final fields, parameters, or local variables to be declared, we mostly avoided checking if the left side of an assignment was allowed, with the exception of array `length`.

In order to test the left side of an assignment, we needed to recursively propagate the `lvalue/variable` of the `SubExpression` on the left side of the assignment. Much like with the rest of type checking, we define an `entity` method for each relevant AST class which can determine, based on

JLS rules, if the AST would produce an `lvalue`. If no `lvalue` is produced, then `nil` is the result.

When type checking an assignment AST node, we ask for the `lvalue` of the left side. If we get `nil` then the code is not valid and the compiler exits with an appropriate message. We found that this case is not tested by any particular Marmoset test, or at least not a test that would not fail for another reason.

If we do get an `lvalue`, then we check that it is not the `length` field of an array.

Reachability Analysis

Reachability analysis, as defined in the JLS, seemed to make the most sense as an addition to `Method`, `Block`, and `Statement` type checking.

Following the rules described for the analysis from the JLS, the analysis rules were added to the type checking step of type checking `Method`s, and after type checking step for type checking `Block`s and `Statement`s. This is a result of the requirement that non-terminating `Method`s do not need to perform normal type checking.

Otherwise, reachability analysis was implemented recursively, in a similar fashion to type checking and constant folding, and is a straightforward adaptation of the judgement rules defined in the JLS.

The document is well organized, clearly written, and detailed. You have made good design decisions in your compiler. Your compiler appears to be in good shape for A5.
The document should discuss how you tested your compiler.

Presentation: 20/20
Content: 8/10
Total: 28/30