CSE 304 – Compiler Design
Spring, 2023
Assignment 04 – Semantic Analysis

Points: 40
Assigned: Wednesday, 03/29/2023
Due: **Wednesday, 04/12/2022, at 11:59 PM**

Description:

In this assignment, you will build a type checker for `Decaf`. Recall the description of `Decaf's` syntax is given in its manual, and the AST you built in Assignment 03. The type checker you will build will:

- Add type information to the AST
- Resolve names used in field access and method invocation expressions

As in the past, you will produce a compiler front end that will take a single command-line argument, which is a file name; if that file contains a syntactically valid `Decaf` program, then it will construct an AST for that program. Upon successful construction of the AST, it checks the type constraints as outlined in this handout (see the section on type checking) and resolves any names that could not be resolved when the AST was constructed (see section on name resolution). Any errors found during type checking and name resolution should be reported with sufficient detail (including the nature of the error and the line numbers in the source program corresponding to the error). If type checking and name resolution complete without any error, then print the AST decorated with types to standard output (see section on printing).

These instructions will fall into the following sections:
1. Changes to the AST Structure
2. Type Checking Constraints
3. Name Resolution
4. Printing the AST
5. Initialization
6. Simplification Amendments
7. Assignment Path
8. Deliverables

1. Changes to AST Structure:

The AST structure remains mostly unchanged from Assignment 03. The changes are as follows. (AST structures not mentioned below are unchanged).

Type Record:

From Assignment 03:

> Each type is either an *array* type or an *elementary* type. Elementary types, in turn, may be one of the *built-in* types: `int`, `float`, `boolean`, or `string`; or a *user-defined* type, which is a class name.

- To the above, we add `void` as a built-in type (this should have been part of the type specification in assignment 03)
- We add a *class-literal* type, which has a class name as a parameter. While *user(A)* denotes an object belonging to class "*A*", *class-literal(A)* denotes the class "*A*" itself.

The use of *class-literal* types will be described when type constraints for expressions are explained (see the section on type checking below).

- We also add `error` and `null` as built-in elementary types to indicate type errors and null object references, respectively.

Subtype Relation:

The type checker will rely on a subtype relation defined over types. This relation follows the discussion in class, with one addition: for the purposes of type checking, we will consider `int` to be a subtype of `float`. In summary:

- `Type` *`T`* is a subtype of itself (i.e., the subtype relation is reflexive).
- `int` is a subtype of `float`
- `user(`*`A`*`)` is a subtype of `user(`*`B`*`)` if *A* is a subclass of *B*.
- `null` is a subtype of `user(`*`A`*`)` for any class *A*
- `class-literal(`*`A`*`)` is a subtype of `class-literal(`*`B`*`)` if *A* is a subclass of *B*.

Changes to Expressions:

Three expressions—field access, method invocation, and new object creation—will all have a new piece of information in the AST denoting the actual field/method/constructor accessed in this expression. This information is determined by name resolution. For instance, consider a field access expression `e.x`. With name resolution, (in the absence of any errors), we will be able to determine that the expression accesses field with id *j* (note: field id's are unique). Then the field access expression `e.x` will contain information that specifies that the field expression refers to field *j*. Similarly, with name resolution, we will determine the unique method or constructor used in a method invocation or new object creation expressions, respectively.

2. Type Checking Constraints:

Statement Records:

Each statement record will now have an additional field indicating whether or not the statement is type correct (i.e., has no type errors).

1. *If-stmt*: This statement is type correct if the *condition* part is of *boolean* type, and the "Then" and "Else" parts are type correct.
2. *While-stmt*: This statement is type correct if the *condition* is of *boolean* type and the loop body is type correct.
3. *For-stmt*: This statement is type correct if the *condition* part is of *boolean* type and the *initializer expression*, *update expression*, and the loop body are all type correct.
4. *Return-stmt*: This statement is type correct if the type of the expression matches the declared return type of the current method.
    - If the expression is empty, then the declared return type of the method must be `void` (and vice versa).
    - If the expression is not empty, then it must be type correct, and its type must be a sub-type of the declared return type of the method.
5. *Expr-stmt*: This statement is type correct if the expression is type correct.
6. *Block-stmt*: This statement is type correct if all the statements in the sequence are type correct.

Expression Records:

Each expression record will now have an additional field indicating it's type.

1. *Constant-expressions:*
    - *Integer-constant*: has type `int`.
    - *Float-constant*: has type `float`.
    - *String-constant*: has type `string`.
    - *True* and *False* have type `boolean`.
    - *Null* has type `null`.
2. *Var-expression*: has, as its type, the declared type of the corresponding variable.
3. *Unary-expressions*:
    - `uminus e`: has the same type as `e` if `e's` type is `int` or `float`; has type `error` otherwise.
    - `neg e`: has type `boolean` if `e's` type is `boolean`; has type `error` otherwise.
4. *Binary-expressions*:
    - *Arithmetic operations*: `add`, `sub`, `mul`, `div`: have type `int` if both operands have type `int`; otherwise, have type `float` if both operands have type `int` or `float`; otherwise have type `error`.
    - *Boolean operations* and, or: have type `boolean` if both operands have type `boolean`; otherwise have type `error`.
    - *Arithmetic comparisons*: `lt`, `leq`, `gt`, `geq`: have type `boolean` if both operands have type `int` or `float`; otherwise have type `error`.
    - *Equality comparisons* `eq`, `neq`: have type `boolean` if the type of one operand is a subtype of another; otherwise have type `error`.
5. *Assign-expression*: Consider `e1 = e2`. The type of this expression is `e2's` type, provided:
    - `e1` and `e2` are type correct
    - `e2's` type is a subtype of *e1*'s type.
6. *Auto-expression*: has the same type as its argument expression `e`, if `e's` type is `int` or `float`; has type `error` otherwise.
7. *Field-access-expression*: Let `p.x` be the field access expression, and let `z` be the field obtained by name resolution. Then this expression's type is the same as `z's` declared type, provided:
    - `p's` type is `user(A)`, and `z` is a non-static field.
    - `p's` type is `class-literal(A)` and `z` is a static field.

    The type of this expression is `error` if name resolution has errors or the above conditions do not hold.
8. *Method-call-expression*: Let `p.f(e_1, …, e_n)` be the method call expression, and let `h` be the method obtained by name resolution. Then this expression's type is the same as `h's` declared return type, provided:
    - `p's` type is `user(A)`, and `h` is a non-static method.
    - `p's` type is `class-literal(A)`, and `h` is a static method.

    The type of this expression is `error` if name resolution had errors or the above conditions do not hold.
9. *New-object-expression*: Let `A(e_1, …, e_n)` be the constructor and arguments of this expression. The type of this expression is `user(A)` if name resolution succeeds (error, otherwise).
10. *This-expression*: has type `user(A)`, where *A* is the current class.
11. *Super-expression*: has type `user(B)` if *B* is the immediate superclass of the current class; it's `error` if the current class has no superclass.
12. *Class-reference-expression*: has type `class-literal(A)` where *A* is the literal class name in this expression (has type `error` if name resolution fails for this class name).

3. Name Resolution:

`Decaf` resolves overloaded names statically, at compile-time, using type information.

Note: all local variables are resolved while constructing the AST. The only names that are not resolved at that time are the fields, methods, and constructors.

Fields:

Consider a field access expression of the form *e.x* for some base expression *e*.

- If `e's` type is `user(A)`: Then the expression resolves to field with id `j` if *B* is the most specific superclass of *A* such that:
  - *B* has the declared *non-static* field named `x` with id `j`,
  - and the field is accessible (based on the public/private declarations) from the current method.
- If *e*'s type is `class-literal(A)`: same as above except *B* has a static field named `x`.

Methods:

Consider a method call expression of the form `e.f(e_1, …, e_n)` for some base expression *e*.

Let the types of `e_1, …, e_n` be `T_1, …, T_n`, and all the argument expressions are type correct.

- If `e's` type is `user(A)`: Then the expression resolved to method with id `j` if *B* is the most specific superclass of *A* such that:
  - *B* has a declared *non-static* method named `f` with id `j` such that:
    - the method `j` is applicable: i.e., the types of the formal parameters of that method are `T1', …, T_n'` and each `T_i` is a subtype of `T_i'`
    - there is no other non-static method named `f` in *B* that is also applicable and has formal parameters types `T_1'', …, T_n''` that are not all subtypes of `T_1', …, T_n'`
    - and the method is accessible (based on the public/private declarations) from the current method
- If `e's` type is `class-literal(A)`: same as above except *B* has a *static* method named `f`.

IMPORTANT NOTE: We will not be dealing with programs with overloaded methods. Each class will contain at most one method with the name `f`. So, we are only required to determine whether the one method with that name in *B* is both applicable (argument types match parameter types) and accessible.

Constructors:

Consider a new object creation expression of the form `new A(e_1, …, e_n)` for some class *A*.

Let the types of `e_1, …, e_n` be `T_1, …, T_n`, and all the argument expressions are type correct.

The constructor resolved for this expression is the constructor with id `j` declared in class *A* such that:

- the constructor `j` is applicable: i.e., the types of the formal parameters of that constructor are `T_1', …, T_n'` and each `T_i` is a subtype of `T_i'`
- there is no other constructor in *A* that is also applicable and has formal parameters types `T_1'', …, T_n''` that are not all subtypes of `T_1', …, T_n'`

- and the constructor is accessible (based on the public/private declarations) from the current method.

***IMPORTANT NOTE*:** We will not be dealing with programs with overloaded constructors. Each class will contain at most one constructor. So, we are only required to determine whether the one constructor defined in *A* is both applicable (argument types match parameter types) and accessible.

4. Printing the AST:

AST printing will follow the same form as in Assignment 03 with the following changes:

- Each field access, method invocation, and new object creation expression will contain additional information: the id of the resolved field/method/constructor respectively. For instance,

  ```
  Field-access(This, x)
  ```

  you should print

  ```
  Field-access(This, x, 1)
  ```

  where "1" is the id of the resolved field name "*x*".

- Each assignment expression will additionally contain the types of the LHS and RHS expressions. Note this is only for assignment expressions, and not for others. For instance, instead of

  ```
  Expr( Assign(Variable(1), Method-call(This, f, [])) )
  ```

  you should print

  ```
  Expr( Assign(Variable(1), Method-call(This, f, []), int, int) )
  ```

5. Initialization

Same as in Assignment 03

6. Simplification Amendments:

We are creating a type checker for a subset of the `Decaf` language. It does not include the following:

- Arrays: there are no arrays in our subset of `Decaf`
- Implicit Field Accesses: all field accesses in our subset of `Decaf` will specify the object explicitly
- Implicit Method Calls: all method calls in our subset of `Decaf` will specify the object explicitly
- No Overloaded Methods/Constructors: Assume that in each class, there is at most one constructor and at most one method with a given name. Also assume that the names of methods defined in a class are distinct from method names in its super classes. *You do not need to check for these constraints. Your type checker can safely assume that they are true.*
- The above assumption leads to a significant simplification in name resolution. This also eliminates several additional error checks.

7. Assignment Path:

1. Start with a working AST builder.
2. Assume no method invocations, constructors, or field accesses; write a type checker that works on this restricted subset of programs. You may want to work with even smaller subsets than these.
3. Proceed by adding field accesses first, and finally method invocations and constructors.

8. Deliverables:

Your AST checker for `Decaf` should be organized into the following files:

1. **Lexer:** `Decaf_lexer.py` – PLY/lex scanner specification file. (*unchanged from A02*)
2. **Parser:** `Decaf_parser.py` – PLY/yacc parser specification file. (*unchanged from A03*)
3. **AST:** `Decaf_ast.py` – table and class definitions for `Decaf`'s AST. (*modified from A03 to add type-related fields and definitions*)
4. **Type Checker:** `Decaf_typecheck.py` – definitions for evaluating the type constraints and for name resolution.
5. **Main:** `Decaf_checker.py` – containing the main python function to put together the parser and lexer, take input from given file, etc., and perform syntax checking.
6. **Documentation:** `README.txt` which documents the contents of all the other files.

We will be looking for a submission with the six files specified above, with the specified names. Deviating from these specifications may entail loss of points.