CSE 304 – Compiler Design
Fall, 2023
Assignment 03 – AST Construction

Points: 40
Assigned: Wednesday, 10/04/2023
Due: **Friday, 11/03/2023, at 11:59 PM**

Problem: Constructing an AST for `Decaf` Programs

This assignment asks you to build a module to construct ASTs for `Decaf` programs. Recall the description of `Decaf`'s syntax is given in its manual. Your AST construction will begin with the PLY-based lexical analyzer and parser you developed for HW2.

Overall, your AST builder 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.

As an AST is being constructed, you will perform additional syntax checks (defined later in this document).

Upon successful construction of the AST, it will then print the AST to standard output in a reasonable form (outlined later in this document). The AST itself should follow the structure described below.

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 HW2)
2. **Parser:** `decaf_parser.py` – PLY/yacc parser specification file.
   (modified from HW2 with actions to build AST)
3. **AST:** `decaf_ast.py` – table and class definitions for `Decaf`'s AST.
4. **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.
5. **Documentation:** `README.txt` which documents the contents of all the other files.

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

Input:

Your program should read the name of a file from the command-line. This file will contain the `Decaf` program that your syntax checker must read and parse. During the parse, an Abstract Syntax Tree representing the source program will be constructed and the program will be checked for several kinds of errors.

Output:

The result of the execution of the parser and AST builder against the `Decaf` program in the input file should be written to standard output.

- A pretty print (described below) of the AST – if the `Decaf` program is syntactically correct
- An error message describing the first syntax error in the `Decaf` program and where it occurred (line and column number) – if the `Decaf` program is not syntactically valid

Submission Instructions:

1. Submit all of your files in a single compressed archive named: `cse304_a03.zip`
2. Make sure you include your name, netid, and Student ID number as comments at the top of your source code files.
3. Make sure you use any specified, file, function, and class names (and any other names specified in the instructions). Grading may be partially automated and the use of incorrect names will interfere.

This is an independent programming assignment. Any collaboration on coding, with anyone other than your recognized partner, will be considered a violation of academic honesty.

Amendments to (simplifications of) the `Decaf` Specification:

- No Arrays: there are no arrays in our subset of `Decaf`.
- No Implicit field accesses: all field accesses in our subset of `Decaf` will specify the object explicitly.
- No Implicit method calls: all method calls in our subset of `Decaf` will specify the object explicitly.

AST Structure:

The core of the AST is a set of symbol tables, and a set of tree-structured data that links these tables. For `Decaf`, the top-level table is a Class Table that holds information about each class in the input program.

Class Table: Contents

The class table is a collection of records. You may implement this in any appropriate way: dictionaries, tuples, etc. Each record in the table holds information about a class. The necessary information for a class are:

Class Record:

- *Class name*: Name of the class
- *Super class name*: Name of the super class (if any) of the current class.
- *Class constructors*: The set of constructors defined in the class.
- *Methods*: The set of methods defined in the class.
- *Fields*: The set of fields defined in the class.

Note that `Decaf` does not have nested class definitions, so the above information is complete for each class.

The information about individual constructors, methods, and fields themselves are stored as record structures with the following information:

Constructor Record:

- *Constructor id*: Note that the name of a constructor in `Decaf` is identical to its containing class. So, we will identify each constructor by a unique integer id. This id should be unique across the entire program: no two constructors, irrespective of their containing class, should have the same id.
- *Constructor visibility*: which encodes the visibility (`public`/`private`) of the constructor.
- *Constructor parameters*: sequence of formal parameters of the constructor. Each parameter is a variable (a reference to a variable in the variable table, described later).
- *Variable table*: A table of variables containing information on all the formal parameters and local variables of the constructor (described later).
- *Constructor body*: is a statement (an instance of a *statement* record, described later).

Method Record:

- *Method name*: name of this method.
- *Method id*: A unique integer id for this method. This id should be unique across the entire program: no two methods, irrespective of their containing class, should have the same id.
- *Containing class*: the name of the class where this method is defined.
- *Method visibility*: which encodes the visibility (`public`/`private`) of the method.
- *Method applicability*: which describes whether this is a class method (i.e. `static`) or an instance method (non-`static`).
- *Method parameters*: sequence of formal parameters of the method. Each parameter is a variable (a reference to a variable in the variable table, described later).
- *Return type*: an instance of a *type* record (described later) that describes the declared return type of this method. This information is omitted for void methods (i.e. those that return no value).
- *Variable table*: A table of variables containing information on all the formal parameters and local variables of the method (described later).
- *Method body*: is a statement (an instance of a *statement* record, described later).

Field Record:

- *Field name*: name of this field.
- *Field id*: A unique integer id for this field. This id should be unique across the entire program: no two fields, irrespective of their containing class, should have the same id.
- *Containing class*: the name of the class where this field is defined.
- *Field visibility*: which encodes the visibility (`public`/`private`) of the field.
- *Field applicability*: which describes whether this is a class field (i.e. `static`) or an instance field (non-`static`).
- *Type*: an instance of a *type* record (described later) that describes the declared type of this field.

You may find it convenient to store the set of all constructor records in a separate constructors table; method records in a separate methods table; and field records in a separate fields table.

Variable Table: Contents

Each constructor/method is associated with a table with information on all variables (formal parameters or local variables) declared in that constructor/method. Each variable holds the following information:

Variable **Record**:

- *Variable name*: Name of the variable
- *Variable id*: A integer that forms an unique id for this variable. No two distinct variables in a constructor or method should have the same id. Note that scopes determine when two variables are distinct; this issue is described in detail later in this document. Note that the uniqueness is only within the constructor/method; two distinct variables in two different methods may have the same id.
- *Variable kind*: (formal/local) indicating whether this variable is a formal parameter or a local variable.
- *Type*: an instance of a *type* record (described later) that describes the declared type of this variable.

Type Record: Contents

- *Type*: the name of the type

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

For instance:

- Consider a variable `x1` declared *`int x1`*. Then `x1`'s type is the elementary built-in type `int`.
- Consider a variable `x2` declared *`List x2`*. Then `x2`'s type is the elementary user-defined type referring to class `List`.

Note: we may modify the contents of type records in later assignments. The content specified above is sufficient for this homework.

Statement Record: Contents

There are several kinds of statements, each with its own contents:

1. *If-stmt*: has three pieces of information:
    - the condition of the "if", which is an expression (described later),
    - the "then" part, which is another statement, and
    - the "else" part, which is another statement.
2. *While-stmt*: has two pieces of information:
    - the loop-condition of the "while", which is an expression (described later),
    - the loop body, which is another statement.
3. *For-stmt*: has four pieces of information:
    - the initializer expression, which is an expression (described later),
    - the loop condition, which is another expression (described later),
    - the update expression, which is another expression (described later),
    - the loop body, which is another statement.
4. *Return-stmt*: has one optional piece of information:
    - the return value, specified by an expression (described later).
5. *Expr-stmt*: has one piece of information:
    - the expression that comprises of this statement.
6. *Block-stmt*: has one piece of information:

    - a sequence of statements that comprise of this block.
7. *Break-stmt*: representing `break`.
8. *Continue-stmt*: representing `continue`.
9. *Skip-stmt*: representing an empty statement (as in an empty else part of an "if" statement).

Each statement record additionally contains the line number range in the input corresponding to it. (This information may be used in later assignments for signaling additional errors/warnings).

Expression Record: Contents

There are many kinds of expressions, each with its own contents:

1. *Constant-expression*: which, in turn, is one of the following six forms:
    - *Integer-constant*: with the integer value as its information.
    - *Float-constant*: with the floating-point value as its information.
    - *String-constant*: with the string value as its information.
    - *Null*, *True*, and *False*: with no additional information.
2. *Var-expression*: has one piece of information: the id of the referenced variable. Note that if there are multiple variables of the same name defined in nested blocks, the scope rules (defined later) specify which if those variables will be the referenced one.
3. *Unary-expression*: has two pieces of information: the operand (an expression) and the unary operator. Unary operators are either `uminus` or `neg`. Note that the unary operator "`+`" in the concrete syntax has no effect. So an expression of the form "`+25`" will be represented as though it was simply "`25`".
4. *Binary-expression*: has three pieces of information: the two operands (both expressions themselves) and the binary operator. Binary operators are one of `add`, `sub`, `mul`, `div`, `and`, `or`, `eq`, `neq`, `lt`, `leq`, `gt`, and `geq`.
5. *Assign-expression*: has two pieces of information: the left- and right- hand sides of the assignment (both expressions).
6. *Auto-expression*: has three pieces of information to represent auto-increment and auto-decrement expressions (e.g. "`x++`"). The three pieces of information are the operand (e.g. "`x`") which is an expression, whether the operation is an auto-increment (e.g. `x++`) or auto-decrement (e.g `x--`), and whether the operation is post (e.g. `x++`) or pre (e.g. `++x`).
7. *Field-access-expression*: has two pieces of information to represent `p.x`: the base (e.g. `p`), which is an expression, and the field name (e.g. `x`), which is a string.
8. *Method-call-expression*: has three pieces of information to represent `p.f(x, y)`: the base (e.g. `p`), which is an expression, the method name (e.g. `x`), which is a string, and a sequence of expressions representing the arguments to the method call (e.g. `x`, `y`). Note that the argument sequence may be empty.
9. *New-object-expression*: has two pieces of information for creating a new object as in `new a(i, x)`: the base class name (e.g. `a`), and the sequence of expressions representing the arguments to the constructor (e.g. `i`, `x`). Note that the argument sequence may be empty.
10. *This-expression*: to denote `this`.
11. *Super-expression*: to denote `super`.
12. *Class-reference-expression*: with the referred class name as its information. This expression is used to denote the value of literal class names.

Each expression record additionally contains the line number range in the input corresponding to it. (This information may be used in later assignments for signaling additional errors/warnings).

Scopes of names:

`Decaf` is statically, lexically scoped. Programs have different kinds of named entities: classes, methods, fields and variables. The access rules for these names differ slightly, as explained below.

- **Classes**: The language has a flat class structure: you cannot nest classes. A class name can only be either *within* that class or in some *subsequent* class definition. For example, if `C` is a class, then we can use `C` to refer to this class inside `C`'s definition itself (i.e., in its fields and methods and their definitions); or we can use `C` to refer to this class later in the program (in a class whose definition occurs after class `C` in the program).

- **Methods**: A method's name will be used *only in the context of an explicit object or class reference*. That is, method calls will always be of the form:
  1. `p.f(x)` for some object `p` (note: `p` may be `this` or `super`) [for instance methods]; or
  2. `C.f(x)` for some class `C` (for class methods).

  A method will not be invoked on the current object implicitly; for such uses, the current object will be specified explicitly using `this`. For instance, our subset of `Decaf` will not have assignments of the form `x = f(y);` instead, it will be written as `x = this.f(y).`

  Your AST builder or parser need not check whether the input program meets this restriction. You can simply assume that any program used with your AST builder will satisfy this restriction.

- **Fields**: A field's name will be used *only in the context of an explicit object or class reference*. That is, field accesses will always be of the form:
  1. `p.x` for some object `p` (note: `p` may be `this` or `super`) [for instance fields]; or
  2. `C.f(x)` for some class `C` [for class fields].

  This means that the current object will never be accessed implicitly; the current object will be specified explicitly using this. For instance, our subset of `Decaf` will not have assignments of the form `x = 2` for some field `x`; instead, it will be written as `this.x = 2`.

  Your AST builder or parser need not check whether the input program meets this restriction. You can simply assume that any program used with your AST builder will satisfy this restriction.

- **Variables:** Statements in a method have a block structured defined by blocks delimited by "`{`" and "`}`". The blocks form a tree structure. Note that variables can only be declared at the beginning of blocks. A variable's scope extends throughout the block it is declared in, as well as in any enclosed block, except if explicitly overridden by another declaration of a variable of the same name. For the purposes of determining their scope, formal parameters of a method will be treated the same way as local variables declared in the outer-most block of method body.

To resolve the identity of variables given these scope rules, you can use the following procedure. Note that a variable is first encountered as a part of field_access (the option without a preceding "."). If this name is declared as a variable in the current block or any enclosing block (up to the formal parameters of the current method), identify it with its most recent declaration. Otherwise, this could be a reference to a (previously defined) class. If there is a class with the same name, then the expression is a class reference expression. If not—that is, if there is no class with a matching name, then this is an error.

Additional Error Checks:

While constructing an AST, you may be able to detect the following errors related to multiple or confounding declarations:

1. *Classes*: Each class in a `Decaf` program must have a distinct name. Hence, it is an error to have two classes with the same name.
2. *Methods*: Since `Decaf` is an object-oriented language that permits method overloading, it is legal to define multiple methods with the same name in a given class. (In the type checking assignment that will follow, we will refine this to permit multiple methods only of their type signatures are different, but we will be liberal now).
3. *Fields*: Each field declared within a class must have a distinct name. Note that, given the class structure, it is possible for a class as well as its super-class to have fields with same names declared. Hence it is only an error if two fields declared in the same class have same names.
4. *Variables*: Each variable declared within a block must have a distinct name. Following the definition of scopes, the names of formal parameters of a method must be distinct from the names of variables declared in the top-most block of the method's body.

Your AST builder is expected to detect and report such errors.

Printing the AST:

Class table:

Each entry of the class table must be printed out, with the following information in the specified order:

1. `Class name`: < string with name of class >
2. `Super class name`: < Name of the super class (empty if none >
3. `Fields`: < The set of fields defined in the class, in format defined below >
4. `Constructors`: < The set of constructors defined in the class, in format defined below >
5. `Methods`: < The set of methods defined in the class, in format defined below >

Entries of the class table will be separated by a line of "`---`"s.

Field record:

Each field record must be printed in a single line, beginning with "`FIELD: `", with a comma separated list of the following information, in the specified order:

1. Field Id
2. Field Name
3. Containing Class
4. Field visibility
5. Field applicability
6. Field Type (in format defined below)

Constructor Record:

Each constructor record must be printed in four parts:

1. A single line beginning with "`CONSTRUCTOR: `" and with a comma-separated list of the following information in the specified order:
    a. Constructor Id
    b. Constructor visibility
2. A single line, starting with "`Constructor parameters:`" with a comma-separated list of ids (corresponding to the constructor's formal parameters) in the variable table (below).
3. A set of lines starting with "`Variable Table:`", with each of the following lines containing information about a variable in the constructor (format defined later).
4. A set of lines, starting with "`Constructor Body:`" with each of the following lines containing information about the statements in the constructor's body (format defined later).

Method Record:

Each method record must be printed in four parts:

1. A single line beginning with "`METHOD: `" and with a comma-separated list of the following information in the specified order:
    a. Method Id
    b. Method Name
    c. Containing class
    d. Method visibility
    e. Method applicability
    f. Return type in format defined below)
2. A single line, starting with "`Method parameters:`" with a comma-separated list of ids (corresponding to the methods's formal parameters) in the variable table (below).
3. A set of lines starting with "`Variable Table:`", with each of the following lines containing information about a variable in the method (format defined later).
4. A set of lines, starting with "`Method Body:`" with each of the following lines containing information about the statements in the method's body.

Type Record:

Each type must be printed in one of the following ways:

- *Built-In Types*: "`int`", "`float`", "`string`", or "`boolean`".
- *User-Defined Types*: "`user(name)`"

Statements:

Each statement must be:

- Printed starting on a new line
- Containing the statement's kind (e.g. "`For`")
- Followed by an open parenthesis ("`(`")
- Then by a comma-separated print of its component information
- And finally a close parenthesis ("`)`").

A sequence of statements (as in a block statement) must be printed out as a comma-separated list of statements.

Expressions:

Each expression must be:

- Printed starting with the expessions's kind (e.g. "`Binary`")
- Followed by an open parenthesis ("`(`")
- Then by a comma-separated print of its component information
- And finally a close parenthesis ("`)`").

A sequence of expressions (as in the list of arguments to a method call) must be printed out as a comma-separated list of expressions.

Initialization:

The `Decaf` manual says that there are two standard objects in the language: `Out` and `In`. We can treat these two as predefined classes. For AST construction, you should start out with a table that is already populated with the information about these two classes:

- `In`: Consider this as a class with no fields, no constructors, but with the following two methods:
  1. `scan_int`: a public static method that takes no parameters, with a return type of `int`.
  2. `scan_float`: a public static method that takes no parameters, with a return type of `float`.

- `Out`: Consider this as a class with no fields, no constructors, but with the following four methods:
  1. `print`: a public static method that takes one formal parameter (for convenience, call it `i`) of type `int`, and returns nothing.
  2. `print`: a public static method that takes one formal parameter (for convenience, call it `f`) of type `float`, and returns nothing.
  3. `print`: a public static method that takes one formal parameter (for convenience, call it `b`) of type `boolean`, and returns nothing.
  4. `print`: a public static method that takes one formal parameter (for convenience, call it `s`) of type `string`, and returns nothing.

  Note that all methods in class Out have the same name! In a later assignment, we will determine which `print` method to use by resolving this overloading.

Assignment Path:

1. Start with a working PLY parser for `Decaf`.
2. Create a Python class to hold the information for each `Decaf` class. Write a stub of a print method, and refine this later. Ensure you have fields in this Python class to denote class and super class names, collection of fields, constructors and methods. In your parser, add actions to fill out the class and super class names (leave the rest empty initially). Create a table to store these class records, and initialize appropriately. In your parser, add actions to fill this table. In the main driver (that calls the parser), iterate over this table and print its contents.
3. Add actions to synthesize information on fields, and to place this information in the respective class records.
4. Add actions for simple expressions (constants, field access with "this", assignments, binary operations), and simple statements (expression statements, return statements, blocks).

5. Proceed by either expanding on the expressions (to more complex ones such as method calls, new object creation etc.) and statements (to more complex ones such as while loops).

EXAMPLE:

Input `Decaf` Program:

```
class A {
   int x;
   A () {
     this.x = 0;
   }
   int f() {
      return this.x + 1;
   }
   public int g() {
      int i;
      i = this.f();
      i++;
      return i;
   }
}
class B extends A {
   int y;
   public A s;
   B () {
     this.y = 2;
     this.s = new A();
   }
   public int f(int k) {
      return super.f() + k;
   }
}
```

AST (printed out):

```
--------------------------------------------------------------------------
Class Name: A
Superclass Name:
Fields:
FIELD 1, x, A, private, instance, int
Constructors:
CONSTRUCTOR: 1, private
Constructor Parameters:
Variable Table:
Constructor Body:
Block([
Expr( Assign(Field-access(This, x), Constant(Integer-constant(0))) )
])
Methods:
METHOD: 1, f, A, private, instance, int
Method Parameters:
Variable Table:
Method Body:
Block([
Return( Binary(add, Field-access(This, x), Constant(Integer-
constant(1))) )
])
METHOD: 2, g, A, public, instance, int
Method Parameters:
Variable Table:
VARIABLE 1, i, local, int
Method Body:
Block([
Expr( Assign(Variable(1), Method-call(This, f, [])) )
,  Expr( Auto(Variable(1), inc, post) )
,  Return( Variable(1) )
])
--------------------------------------------------------------------------
Class Name: B
Superclass Name: A
Fields:
FIELD 2, y, B, private, instance, int
FIELD 3, s, B, public, instance, user(A)
Constructors:
CONSTRUCTOR: 2, private
Constructor Parameters:
Variable Table:
Constructor Body:
Block([
Expr( Assign(Field-access(This, y), Constant(Integer-constant(2))) )
,  Expr( Assign(Field-access(This, s), New-object(A, [])) )
])
Methods:
METHOD: 3, f, B, public, instance, int
Method Parameters: 1
```

```
Variable Table:
VARIABLE 1, k, formal, int
Method Body:
Block([
Return( Binary(add, Method-call(Super, f, []), Variable(1)) )
])
--------------------------------------------------------------------------------
```