VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF PROGRAMMING LANGUAGES - CO3005

ASSIGNMENT 4

Static Checker

HO CHI MINH CITY, 07/2021

ASSIGNMENT 4

Version 1.0

After completing this assignment, you will be able to

- explain the principles how a compiler can check some semantic constraints such as type compatibility, scope constraints,... and
- write a medium (300 - 500 lines of code) Python program to implement that.

1 Specification

In this assignment, you are required to write a static checker for a program written in D95. To complete this assignment, you need to:

- Read carefully the specification of D95 language
- Download and unzip file assignment4.zip
- If you are confident on your Assignment 3, copy your D95.g4 into src/main/d95/parser and your ASTGeneration.py into src/main/d95/astgen and you can test your Assignment 3 using D95 input like the first three tests (400-402).
- Otherwise (if you did not complete Assignment 3 or you are not confident on your Assignment 3), don't worry, just input AST as your input of your test (like test 403-405).
- Modify StaticCheck.py in src/main/d95/checker to implement the static checker and modify CheckSuite.py in src/test to implement 100 testcases for testing your code.

2 Static checker

A static checker plays an important role in modern compilers. It checks in the compiling time if a program conforms to the semantic constraints according to the language specification. In this assignment, you are required to implement a static checker for D95 language.

The input of the checker is in the AST of a D95 program, i.e. the output of the assignment 2. The output of the checker is nothing if the checked input is correct, otherwise, an error message is released and the static checker will stop immediately.

For each semantics error, students should throw corresponding exception given in StaticError.py inside folder src/main/d95/checker/ to make sure that it will be printed out the same as expected. Every test-case has at most one kind of error. The semantics constraints required to check in this assignment are as follows.

2.1 Redeclared Constant/Function/Parameter

In D95, a variable is declared with the first usage (by assignment statment) but the function and constant must be declared before used. The function and constant name must be unique in global scope and the parameter must be unique in its function scope, otherwise, the exception Redeclared(<kind>,<identifier>) is released, where <kind> is the kind of the <identifier> (Constant/Function/Parameter) in the second declaration.

The scope of an identifier (variable, parameter) is informally described as in D95 specification. All function and constant declarations are in global scope and their scope is the entire program. The function must be declared before all its invocations (just like in C/C++). All function names are unique that means a function name cannot be similar to any other built-in function names or other function names.

2.2 Undeclared Identifier

The exception UndeclaredIdentifier(<identifier>) is released when there is an <identifier> is used but its declaration cannot be found. The identifier can be a variable, constant, parameter or function. This exception is also thrown when the identifier is used as a function name of a function call but there is no such a function declared before in the program.

2.3 Type Cannot Be Inferred

If an identifier is used but its type has not been inferred yet, the exception TypeCannotBeInferred(<statement>) will be released. To infer the type of a variable, a constant or a function, D95 reads the program from the beginning to the end and applies the following rules:

- When a variable/ constant is initialized in its declaration, the type of the variable is also the type of the initialized expression.
- The type of an identifier (variable, constant, parameter or function) must be inferred in the first appearance of the identifier's usage of and cannot be changed. If its type cannot be inferred in the first use, the innermost statement containing the first use of the identifier is sent with the exception.
- If an expression can be inferred to some type but some of its components cannot be inferred to any type, the innermost statement containing the type-unresolved component will be associated with the exception. For example, the expression in the right hand side of the statement y = a + _foo(x) can be inferred to type float as the result of + is in type int and y, a and the return type of foo can also be inferred to type float (rules in section 2.5), but we cannot infer the type of x, then the exception is raised with the assignment statement.

- A call statement to a type-unresolved function is valid when all its parameter types can be inferred by the corresponding argument types and its return type can be inferred to VoidType. If there exists at least one type-unresolved parameter, the exception is raised with the call statement. Note that if the number of the arguments is not the same as the number of the arguments, the exception concerned in Section 2.4 is raised.
- A function call to a type-unresolved type function is valid if all its parameter types and the return type can be resolved. Otherwise, the innermost statement containing the function call is associated to the exception. Note that if the number of the arguments is not the same as the number of the arguments, the exception concerned in Section 2.5 is raised.
- The types of both sides of an assignment must be the same (i.e. same scalar type or same element type and same list of dimensions for array type) so that if one side has resolved its type, the other side can be inferred to the same type. If both sides cannot be resolved their types, the exception is raised with the assignment.
- For each statement, all variables appear in the statement must have type resolved otherwise the innermost statement containing the type-unresolved variable will be associated with the raised exception.

2.4 Type Mismatch In Statement

A statement must conform the corresponding type rules for statements, otherwise the exception TypeMismatchInStatement(<statement>) is released. The type rules for statements are as follows:

- There is type compatibility rules for int, float and boolean types:
  - The identifier with int/ boolean type can be assigned to the float-type one.
  - The identifier with boolean type can be assigned to the int-type one.
- The type of a conditional expression in an if statement must be boolean.
- The type of condition expression in do while and while statements must be boolean.
- For an assignment, the left-hand side (LHS) can be in any type except VoidType. The right- hand side (RHS) is the same type as that of the LHS or followed by the type compatibility rules.
- For a call statement <method name>(<args>), the callee must have VoidType as return type. The number of arguments and the number of parameters must be the same. In addition, the type of each argument must be the same as the corresponding parameter.
- For a return statement, if the return type of the enclosed function is VoidType, the expression in the return statement must be empty. Otherwise, the type of the return expression must be the same as the return type of the function.

2.5 Type Mismatch In Expression

An expression must conform the type rules for expressions, otherwise the exception TypeMismatchInExpression(<expression>) is released. The type rules for expression are as follows:

- For binary expression:
  - With any operator accepted for operands of int/float type, the operands could be different, but in int or float type. It returns float type if at least one of their operands is in float type, otherwise, return int type.

    If one of operands is type-unresolved, it must be inferred as float type.
  - With other operators, the expression accepts only one type described in specification. It returns boolean type for expession with relational operator
- For array accessing expression:
  - For indexed, multi-indexed array: the type of index must be in int type.
  - For associative array: the type of index must be followed by the type of key in the array reference.
  - For an multi-indexed array accessing E[E1]...[En], E must be in array type with n dimensions.
- For unary expression, the type rules are described in the D95 specification.
- For a function call <function name>(<args>), the number of the actual parameters must be the same as (or followed by type compatibility rules in 2.4) that of the formal parameters of the corresponding function. The type of each argument must be same as the type of the corresponding parameter.

2.6 Invalid Array Literal

This exception will be raised if the following rules is volated:

- For indexed array literal: all of the elements are in the same type and the type of first element will be applied for the rest in array. It can be the the element type of array.

  Two indexed array literal are considered as the same type if they have the same type in element type.
- For associative array literal: all of associative expression in this array will be the same for types of key (int/ string) and value (in any type).

  For example: array(1 => "abc", 2 => "bcd") is valid but array(1 => "abc", "2" => "bcd" is invalid.

Two associative array literal are considered as the same type if they have the same type in key and value.

- For multi-indexed array literal: like in indexed array literal.

2.7 No entry point

There must be a function whose name is **_main** in a D95 program. Otherwise, the exception NoEntryPoint() is released.

3 Submissions

This assignment requires you submit 2 files: StaticCheck.py containing class StaticChecker with the entry method check, and CheckSuite.py containing 100 testcases.

File StaticCheck.py and CheckSuite.py must be submitted in "Assignment 4: Submission".

The deadline is announced in course website and that is also the place where you MUST submit your code.

4 Plagiarism

You must complete the assignment by yourself and do not let your work seen by someone else. If you violate any requirement, you will be punished by the university rule for plagiarism.

5 Change log