Exercise 3 – Semantic Analysis

Compilation 0368-3133

Due 18/12/2025, before 23:59

1 Assignment Overview

The third exercise implements a semantic analyzer that recursively scans the AST produced by CUP, and checks if it contains any semantic errors. The semantic analyzer takes as input a single text file containing a **L** program, and outputs a single text file indicating whether the program is semantically valid. When the input program is semantically valid, the semantic analyzer adds metadata to the abstract syntax tree, which is needed for later phases. The added metadata content will not be checked in this exercise. However, if you have time, it is recommended to implement this addition in Exercise 3. In Tutorial 6, we will discuss which semantic information should be attached to AST nodes.

2 The L Semantics

This section describes the semantics of **L**, and provides a multitude of legal and illegal example programs. To fully understand the exercise requirements, please refer to Tutorial 5 on semantic analysis. Read the assignment carefully and make sure your implementation enforces all the semantic rules described in this document.

2.1 Types

Primitive types: The **L** programming language defines two primitive types: *integers* and *strings*.

Composite types: Additional types may be defined as follows:

- *Classes:* declared with data members and methods.
- *Arrays:* defined over an existing (non-void) type `T`.

Class and array definitions may appear only in the global scope. Any use of a class type or array type (including variable declarations, field declarations, parameter types, return types, and array element types) is legal only if that type has been previously defined in the global scope.

Void: The type *void* can only be used as a function return type. Variables cannot have type void (local/global variables, parameters, class fields).

2.2 Classes

Ordering and inheritance rules in classes:

- A class can extend only previously defined classes. In particular, the class hierarchy has a tree structure, and no class directly or indirectly extends itself.
- Within a class declaration, a field or method can only be referenced after it has been defined.

Table 1 provides examples illustrating these rules.

Allocating a class: A class is allocated using `new T`, with the following rules:

- `T` is a previously defined class.
- The resulting type of the allocation is the class `T`.

Accessing a class: Members of a class can be accessed using `v.f`, with the following rules:

- The variable `v` must have a class type.
- The name `f` must be a previously defined member of that class or one of its superclasses.
- The resulting type is the field's type (for fields) or the method's return type (for methods).

Method overloading, overriding and shadowing:

- Method *overloading* (defining multiple methods with the same name but different signatures in the same class) is illegal.
- Method *overriding* (redefining a method inherited from a superclass in a derived class) is allowed. When overriding a method, the return type and the types and order of arguments must match *exactly* the original method in the superclass. Subclass substitutions in the signature are **not** allowed.
- *Shadowing* is not allowed:
  - It is illegal for a class to define a field with the same name as an existing field or method in the same class or any of its superclasses.
  - It is illegal for a class to define a method with the same name as an existing field in the same class or any of its superclasses.

Table 2 summarizes these facts.

Inheritance If class `Son` is derived from class `Father`, then any place in the program that semantically allows an expression of type `Father`, should semantically allow an expression of type `Son`. Note that transitivity applies: if class `Father` is derived from class `Grandfather`, then class `Son` is also (indirectly) derived from class `Grandfather`. For example, see Table 3.

Nil: Any place in the program that semantically allows an expression of type class, should semantically allow `nil` instead. For example, see Table 4.

2.3 Arrays

An array type must be defined over a previously declared (non-void) type. For example, a one-dimensional integer array:

```
array IntArray = int[];
```

And a two-dimensional integer array (array of arrays):

```
array IntMat = IntArray[];
```

Allocating an array: An array is allocated using `new T[e]`, with the following rules:

- `T` is a previously declared type.
- The size expression `e` must be of type int.
- If `e` is a **constant** expression, it must be greater than 0.

Accessing an array: Elements of an array can be accessed using `v[e]`, with the following rules:

- The type of the variable `v` must be of an array.
- The subscript expression `e` must be of type int.
- If `e` is a **constant** expression, it must be greater than or equal to zero
- The resulting type is the element type `T` of the array that `v` is defined over.

Non-interchangeable types: If two array types are defined over the same element type `T`, they are considered *distinct* and are not interchangeable. This means that even if two arrays store the same type of elements, each array type is treated separately by the type system. For example, see Table 5.

Nil: Any place in the program that semantically allows an expression of type array, should semantically allow `nil` instead. For example, see Table 6.

2.4 Assignments

An assignment `x := e` is legal when the type of expression `e` is *compatible* with the type of variable `x`.

- For primitive types, this requires that `x` and `e` have exactly the same type.
- For arrays, if `e = new T`, then `x` must be of type array defined over type `T`. Note that subclass substitutions are **not** allowed. Otherwise, `x` and `e` have exactly the same type.
- For class types, either `x` and `e` have the same type, or the type of `e` is a subclass of the type of `x` (e.g., if `Son` derives from `Father`, a variable of type `Father` can be assigned a value of type `Son`).
- Recall that assigning `nil` to array and class variables is legal. However, assigning `nil` to int and string variables is *illegal*.

Table 7 provides examples illustrating these rules.

2.5 Functions and Control Structures

If and While: The type of the condition inside if and while statements is the primitive type int.

Return Statements: According to the syntax of **L**, return statements can only be found inside functions. Since functions can *not* be nested, it follows that a return statement belongs to *exactly one* function.

- If a function has return type void, its return statements must be empty (`return;`).
- If a function has a non-void return type `T`, then every return statement must return an expression whose type is compatible with type `T`.

A function/method may have control flow paths without a return statement, even if the return type of the function is not `void`. For example, the following function is semantically valid:

```
int f(int x) {
  if (x = 7) {
    return 1;
  }
}
```

Function and method Calls: Only previously defined functions or methods can be called. When calling a function or method, each argument must have a type compatible with the corresponding parameter in its signature. The type of a function or method call expression is the return type of the called function or method.

2.6 Operators

Binary Operations: The binary operations `-,*,/,<,>` are performed only between integers. The only exception is the `+` binary operation, which can be performed between two integers or between two *strings*. The resulting type of a semantically valid binary operation is the primitive type int, with the single exception of adding two strings, where the resulting type is a string. When performing division (using the `/` operator), if the divisor is a **constant**, it must not be 0. Table 9 summarizes these facts.

Equality Testing: Equality testing (`=`) is legal only when the two expressions are *comparable*.

- For primitive types and arrays, this requires that both expressions have exactly the same type.
- For class types, this requires that either both expressions have exactly the same class type, or that the type of one expression is a subclass of the type of the other expression. For example, if class `Son` is derived from class `Father`, then an expression of type `Father` can be compared for equality with an expression of type `Son`, and vice versa.
- The resulting type of a semantically valid comparison is the primitive type int.
- Recall that an expression of an array or class type may be tested for equality with `nil`. However, comparing `nil` to variables of type int or string is *illegal*.

Table 8 summarizes these facts.

2.7 Scope Rules

L defines four kinds of scopes:

1. if, else and while block scopes
2. Function scopes
3. Class scopes
4. The global scope

Table 10 summarizes key points from this section.

Identifier declarations: An identifier may appear only once in a given scope. Different scopes may contain declarations with the same name. An identifier cannot use a reserved keyword, including primitive type names. In **L**, all identifiers share a single namespace, including type names, variable names, and function names. This means a single name cannot be used for more than one of these at the same time.

Name resolution: When an identifier is used, its declaration is resolved by searching from the innermost scope outward, ending with the global scope. The only exception to this is inside a class scope (see below). Once a class is declared, it can be referenced within its own declaration body. Similarly, a function can call itself recursively. If no declaration is found for an identifier, it is considered a semantic error.

Global scope: Only the global scope may contain:

- Class and array type declarations
- Global variables
- Function definitions

Each name must be distinct from all other global names, reserved keywords, and library function names (PrintInt, PrintString).

Name resolution in class scope: When referring to a name within a class declaration:

1. The search starts in the *class scope* itself.
2. Then continues along the chain of members in its superclass(es).
3. Finally, it moves to the global scope if the name is not found in the class hierarchy.

2.8 Library Functions

L defines the following library functions: PrintInt and PrintString.
The signatures of these functions are as follows:

```
void PrintInt(int i)      { ... }
void PrintString(string s){ ... }
```

Library functions are treated as if they are defined in the global scope.

3 Input and Output

Input: The input for this exercise is a single text file, the input **L** program.

Output: The output is a *single* text file that contains a *single* word:

- When the input program is semantically correct: `OK`
- When there is a lexical error: `ERROR`
- When there is a syntax or semantic error: `ERROR(`*location*`)`
  where *location* is the line number of the *first* error that was encountered. In case of a semantic error, you may assume that if an error occurs, the entire command or construct containing the error is in the same line.

4 Submission Guidelines

Each group should have **only one** student submit the solution via the course Moodle. Submit all your code in a single zip file named `<ID>.zip`, where `<ID>` is the ID of the submitting student. The zip file must have the following structure at the top level:

1. A text file named `ids.txt` containing the IDs of all team members (one ID per line).
2. A folder named `ex3/` containing all your source code.

Inside the `ex3` folder, include a makefile at `ex3/Makefile`. This makefile must build your source files into the semantic analyzer, which should be a runnable jar file located at `ex3/SEMANT` (note the lack of .jar suffix). You may reuse the makefile provided in the skeleton, or create a new one if you prefer.

Command-line usage: `SEMANT` receives 2 parameters (file paths):

- `input` (input file path)
- `output` (output file path containing the expected output)

Self-check script: Before submitting, you *must* run the self-check script provided with this exercise. This script verifies that your source code compiles successfully using your makefile and passes several provided tests. You must execute the self-check on your submission zip file within the school server (`nova.cs.tau.ac.il`) and ensure that your submission passes all checks. Follow these steps:

1. Download `self-check-ex3.zip` from the course Moodle and place it in the same directory as your submission zip file. Make sure no other files are present in that directory.
2. Run the following commands:
   ```
   unzip self-check-ex3.zip
   python self-check.py
   ```

Make sure that your `ids.txt` file follows the required format, and that your makefile does *not* contain hard-coded paths. The self-check script does **not** verify these aspects automatically and you are responsible for ensuring that your code compiles correctly in any directory.

Note: The script fails if any provided test fails. If the failure is not due to the output format and you cannot pass all tests by the submission date, it is still acceptable to submit.

Submissions that fail to compile or do not produce the required runnable will receive a grade of 0, and resubmissions will not be allowed.

5 Starter Code

The exercise skeleton can be found in the repository for this exercise. The skeleton provides a basic semantic analyzer, which you may modify as needed. Some files of interest in the provided skeleton:

- `src/ast/*.java` (Classes for the AST nodes with an additional `semantMe` method, implementing the AST visitor pattern described in tutorial 5.)
- `src/types/*.java` (Classes for the different types returned by the `semantMe` method.)
- `src/symboltable/*.java` (Hash table-based symbol table implementation available for you to use. You may modify the code as needed.)
- The `input` and `expected_output` folders contain input tests and their expected results.

Note that you need to use *your own* LEX and CUP configuration files from previous exercises, as well as your own AST nodes classes, as the ones in the skeleton provide only a partial implementation.

Building and running the skeleton: From the `ex3` directory, run:

```
$ make
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into `SEMANT`

For debugging, run:

```
$ make debug
```

This performs the following additional steps:

- Runs `SEMANT` on `input/Input.txt`
- Generates an image of the resulting syntax tree
- Generates images describing the changes in the symbol table

Note: the steps performed in debug mode should not be performed when running `make` using your own makefile.

Table 1: Referring to classes, methods and data members

1	`class Son extends Father`
	`{`
	`  int bar;`
	`}`
	`class Father`
	`{`
	`  void foo() { PrintInt(8); }`
	`}`
		ERROR

2	`class Edge`
	`{`
	`  Vertex u;`
	`  Vertex v;`
	`}`
	`class Vertex`
	`{`
	`  int weight;`
	`}`
		ERROR

3	`class UseBeforeDef`
	`{`
	`  void foo() { bar(8); }`
	`  void bar(int i) { PrintInt(i); }`
	`}`
		ERROR

4	`class UseBeforeDef`
	`{`
	`  void foo() { PrintInt(i); }`
	`  int i;`
	`}`
		ERROR

Table 2: Method overloading and variable shadowing are both illegal in **L**.

1	`class Father`
	`{`
	`  int foo() { return 8; }`
	`}`
	`class Son extends Father`
	`{`
	`  void foo() { PrintInt(8); }`
	`}`
		ERROR

2	`class Father`
	`{`
	`  int foo(int i) { return 8; }`
	`}`
	`class Son extends Father`
	`{`
	`  int foo(int j) { return j; }`
	`}`
		OK

3	`class IllegalSameName`
	`{`
	`  void foo() { PrintInt(8); }`
	`  void foo(int i) { PrintInt(i); }`
	`}`
		ERROR

4	`class Father`
	`{`
	`  int foo(Father f) { return 8; }`
	`}`
	`class Son extends Father`
	`{`
	`  int foo(Son s) { return j; }`
	`}`
		ERROR

5	`class IllegalSameName`
	`{`
	`  int foo;`
	`  void foo(int i) { PrintInt(i); }`
	`}`
		ERROR

6	`class Father`
	`{`
	`  int foo;`
	`}`
	`class Son extends Father`
	`{`
	`  string foo;`
	`}`
		ERROR

7	`class Father`
	`{`
	`  int foo;`
	`}`
	`class Son extends Father`
	`{`
	`  void foo() { }`
	`}`
		ERROR

Table 3: Class Son is a semantically valid input for foo.

`class Father { int i; }`
`class Son extends Father { int j; }`
`void foo(Father f) { PrintInt(f.i); }`
`void main(){ Son s; foo(s); }`
	OK

Table 4: nil sent instead of a (Father) class is semantically allowed.

`class Father { int i; }`
`void foo(Father f){ PrintInt(f.i); }`
`void main(){ foo(nil); }`
	OK

Table 5: Non interchangeable array types.

`array gradesArray = int[];`
`array IDsArray = int[];`
`void F(IDsArray ids){ PrintInt(ids[6]); }`
`void main()`
`{`
`  IDsArray ids := new int[8];`
`  gradesArray grades := new int[8];`
`  F(grades);`
`}`
	ERROR

Table 6: nil sent instead of an integer array is semantically allowed.

`array IntArray = int[];`
`void F(IntArray A){ PrintInt(A[8]); }`
`void main(){ F(nil); }`
	OK

Table 7: Assignments.

1	`class Father { int i; }` `Father f := nil;`	OK

2	`class Father { int i; }`
	`class Son extends Father { int j; }`
	`Father f := new Son;`
		OK

3	`array gradesArray = int[];`
	`array IDsArray = int[];`
	`IDsArray i := new int[8];`
	`gradesArray g := new int[8];`
	`void foo() { i := g; }`
		ERROR

4	`string s := nil;`	ERROR

Table 8: Equality testing.

1	`class Father { int i; int j; }`
	`int Check(Father f)`
	`{`
	`  if (f = nil)`
	`  {`
	`  return 800;`
	`  }`
	`  return 774;`
	`}`
		OK

2	`int Check(string s)`
	`{`
	`  return s = “abc”;`
	`}`
		OK

3	`array gradesArray = int[];`
	`array IDsArray  = int[];`
	`IDsArray  i:= new int[8];`
	`gradesArray g:=new int[8];`
	`int j := i = g;`
		ERROR

4	`string s1;`
	`string s2 := “abc”;`
	`int i := s1 = s2;`
		OK

Table 9: Binary Operations.

1	`class Father`
	`{`
	`  int foo() { return 8/0; }`
	`}`
		ERROR

2	`class Father { string s1; string s2; }`
	`void foo(Father f)`
	`{`
	`  f.s1 := f.s1 + f.s2;`
	`}`
		OK

3	`class Father { string s1; string s2; }`
	`void foo(Father f)`
	`{`
	`  int i := f.s1 < f.s2;`
	`}`
		ERROR

4	`class Father { int j; int k; }`
	`int foo(Father f)`
	`{`
	`  int i := 620;`
	`  return i < f.j;`
	`}`
		OK

Table 10: Scope Rules.

1	`int salary := 7800;`
	`void foo()`
	`{`
	`  string salary := “six”;`
	`}`
		OK

2	`int salary := 7800;`
	`void foo(string salary)`
	`{`
	`  PrintString(salary);`
	`}`
		OK

3	`void foo(string salary)`
	`{`
	`  int salary := 7800;`
	`  PrintString(salary);`
	`}`
		ERROR

4	`string myvar := “ab”;`
	`class Father`
	`{`
	`  Father myvar := nil;`
	`  void foo()`
	`  {`
	`  int myvar := 100;`
	`  PrintInt(myvar);`
	`  }`
	`}`
		OK

5	`int foo(string s) { return 800;}`
	`class Father`
	`{`
	`  string foo(string s)`
	`  {`
	`  return s;`
	`  }`
	`  void Print()`
	`  {`
	`  PrintString(foo(“Jerry”));`
	`  }`
	`}`
		OK