Exercise 3

Compilation 0368-3133

Due 9/1/2025, before 23:59

1 Introduction

We continue our journey of building a compiler for the invented object oriented language **L**. In order to make this document self contained, all the information needed to complete the third exercise is brought here.

2 Programming Assignment

The third exercise implements a semantic analyzer that recursively scans the AST produced by CUP, and checks if it contains any semantic errors. The input for the semantic analyzer is a (single) text file containing a **L** program, and the output is a (single) text file indicating whether the input program is semantically valid or not. In addition to that, whenever the input program is valid semantically, the semantic analyzer will add meta data to the abstract syntax tree, which is needed for later phases (code generation and optimization). The added meta data content will not be checked in exercises 3, but the best time to design and implement this addition is exercise 3.

3 The L Semantics

This section describes the semantics of **L**, and provides a multitude of legal and illegal example programs.

3.1 Types

The **L** programming language defines two native types: *integers* and *strings*. In addition, it is possible to define a *class* by specifying its data members and methods. Also, given an existing type `T`, one can define an *array* of `T`'s. Note, that defining classes and arrays is only possible in the uppermost (global) scope. The exact details follow.

3.1.1 Type void

The type *void* can only be used as a return type in a declaration of a function (or method). Variables can't be declared as *void* (local/global variables, parameters, class fields).

3.1.2 Classes

Classes contain data members and methods, and can only be defined in the uppermost (global) scope. They can refer to/extend only previously defined classes, to ensure that the class hierarchy has a tree structure. A method `M1` *can't* refer to a method `M2`, if `M2` is defined after `M1` in the class. Following the same concept, a method `M` *can't* refer to a data member `d`, if `d` is defined *after* `M` in the class. Table 1 summarizes these facts.

When accessing a class data member using the dot operator (.), the variable being accessed must be of a class type.

Methods overloading is *illegal* in **L**, with the obvious exception of overriding a method in a derived class. When overriding a method, the return type and argument types must be *exactly the same* and in the same order. Similarly, it is illegal to define a variable with the same name of a previously defined variable (shadowing), or a previously defined method. Table 2 summarizes these facts.

Inheritance If class `Son` is derived[1] 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`. For example, see Table 3.

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

3.1.3 Arrays

Array types can only be defined in the uppermost (global) scope. They are defined with respect to some previously defined type (that can't be the type *void*), as in the following example:

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

Defining an integer matrix, for example, is possible as follows:

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

Allocating and accessing an array An array is allocated using the new operator. When allocating an array of type `T`, the type specified after the new operator must also be `T`. The allocated size must be an *integral size*. If the size expression is **constant**, it must be greater than zero. Similar to allocation, accessing an array entry is semantically valid only when the subscript expression has an integer type, and if the size expression is **constant**, it must be greater than zero. In addition, the type of the variable accessed must be of an array.

Interchangeable types If two arrays of type `T` are defined, they are *not* interchangeable (see Table 5).

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

3.2 Assignments

Assigning an expression to a variable using operator := is legal whenever the two have the same type. In addition, following the concept in 3.1.2, if class `Son` is derived from class `Father`, then a variable of type `Father` can be assigned an expression of type `Son`. Furthermore, following the concept in 3.1.2 and 3.1.3, assigning `nil` to array and class variables is legal. In contrast to that, assigning `nil` to int and string variables is *illegal*.

To avoid overly complex semantics, we will enforce a strict policy when initializing data members inside classes (note that data members are allowed to be uninitialized): a declared data member inside a class can be initialized *only with a constant value* (that matches its type). Specifically, only constant integers, strings and `nil` can be used, and even a simple expression like $5 + 6$ is forbidden. Table 7 summarizes these facts.

3.3 If and While Statements

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

[1]Note that the concept of a derived class is transitive: if class `Father` is derived from class `Grandfather`, then class `Son` is also (indirectly) derived from class `Grandfather`.

3.4 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. When a function `foo` is declared to have a `void` return type, then all of its return statements must be *empty* (`return;`). In contrast, when a function `bar` has a non void return type `T`, then a return statement inside `bar` must be *non empty*, and the type of the returned expression must match `T`. A function/method is allowed to 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;
  }
}
```

3.5 Equality Testing

Testing equality between two expressions using the binary operator = is legal whenever the two have the same type. In addition, following the same reasoning as in 3.2, if class `Son` is derived from class `Father`, then an expression of type `Father` can be tested for equality with an expression of type `Son`. Furthermore, any class variable or array variable can be tested for equality with `nil`. But, in contrast, it is *illegal* to compare a string variable to `nil`. The resulting type of a semantically valid comparison is the primitive type int. Table 8 summarizes these facts.

3.6 Binary Operations

Most binary operations $(-, *, /, <, >)$ are performed only between integers. The single exception to that is the + binary operation, that 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.

3.7 Scope Rules

L defines four kinds of scopes: 1. block scopes of if and while statements, 2. function scopes, 3. class scopes, and 4. the outermost global scope. An identifier can be declared *only once within a scope*, but it may appear in multiple distinct scopes simultaneously. When defining an identifier, its name must not match any reserved keyword, including primitive type names (int, string, void) . When an identifier is being used at some point in the program, its declaration is searched for in all of its enclosing scopes. The search starts from the innermost scope, and ends at the outermost (global) scope. The only exception to this is when dealing with classes (see below). Once a class is declared, it can be referenced within its own declaration body. Similarly, a function can call itself.

Global scope Array and class type declarations, as well as functions, can only be defined in the outermost (global) scope. Class and array type declarations, global variables, and functions follow the same scope rules defined above: their names must be different than any previously defined global variable names, function names, class type names, array type names, and reserved keywords. Additionally, their names must differ from library function names (PrintInt and PrintString, see 3.8). Note that as in 3.1.2, functions may only refer to previously defined types, variables and functions.

Class declaration and data members access Class methods can only be defined in the class scope. When a function/method is being called, or when accessing a variable inside a class scope, the search for a declaration with that name first starts in the class itself. If not found, it moves to the superclass, and so on. If still not found, the search goes up to the global scope for an identifier with that name, as described above for general identifier resolution. If the declaration is missing there too, a semantic error is issued. Similarly, when accessing a data member of a class using the dot operator (.), the search follows the same order: class, superclass, and so on. Note that the search is limited to the class's and superclass(es') data members and does not extend to other scopes

Table 10 summarizes key points from this section; for additional examples, see Tutorial 6.

3.8 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 (main) scope.

4 Input

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

5 Output

The output is a *single* text file that contains a *single* word:
When the input program is correct semantically:
OK

When there is a syntax or semantic error:
ERROR(*location*)
where *location* is the line number of the *first* error that was encountered. You can assume that newlines in the input can only appear after ';' or immediately before or after '{' or '}'. See CUP documentation for information on how to access line numbers during rule reductions.

When there is a lexical error:
ERROR

6 Submission Guidelines

The skeleton for this exercise can be found here: https://github.com/noa-schiller/compilation-tau/tree/main/skeletons/ex3.
The makefile must be located in the following path:

- ex3/Makefile

This makefile will build the semantic analyzer (a runnable jar file) in the following path:

- ex3/COMPILER

Feel free to reuse the makefile supplied in the skeleton, or write a new one if you want to.

6.1 Command-line usage

COMPILER receives 2 parameters (file paths):

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

6.2 Skeleton

You are encouraged to use the makefile provided by the skeleton. Some files of intereset in the provided skeleton:

- *jflex/LEX_FILE.lex* (LEX configuration file)
- *cup/CUP_FILE.cup* (CUP configuration file)
- *src/Main.java*
- *src/AST/*.java* (classes for the AST nodes with an additional `SemantMe` method, implementing the AST visitor pattern described in tutorial 6)
- *src/TYPES/*.java* (classes for the different types returned by the `SemantMe` method)
- *src/SYMBOL_TABLE/*.java* (hash-table based symbol table implementation available for you to use)
- *input* and *expected_output* contain input tests and their corresponding expected results

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

To run the skeleton, you might need to install the following packages:

```
$ sudo apt-get install graphviz eog
```

6.2.1 Compiling

To build the skeleton, run the following command (in the *src/ex3* directory):

```
$ make compile
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into *COMPILER*

6.2.2 Debugging (Optional)

For debugging, run the following command (in the *src/ex3* directory):

```
$ make debug
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into *COMPILER*
- Runs *COMPILER* on input/Input.txt
- Generates an image of the resulting syntax tree
- Generates images which describe the changes in the symbol table

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

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

6	`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	`class Father { int i; }` `class Son extends Father { int j := 8; }`	OK
4	`class Father { int i := 9; }` `class Son extends Father { int j := i; }`	ERROR
5	`class Father { int foo() { return 90; } }` `class Son extends Father { int j := foo(); }`	ERROR

6	`class IntList`
	`{`
	`  int head := -1;`
	`  IntList tail := new IntList;`
	`}`
		ERROR

7	`class IntList`
	`{`
	`  IntList tail;`
	`  void Init() { tail := new IntList; }`
	`  int head;`
	`}`
		OK

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

9	`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 = “LosPollosHermanos”;`
	`}`
		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 := “HankSchrader”;`
	`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