Exercise 3

Due 15/12/2021, 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 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.

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 1: Referring to classes, methods and data members

Methods overloading is *illegal* in **L** , with the obvious exception of overriding a method in a derived class. 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.

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;`
	`}`
	`CLASS Son EXTENDS Father`
	`{`
	`  string foo;`
	`}`
		ERROR

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

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`. For example,

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

nil expressions any place in the program that semantically allows an expression of type class, should semantically allow `nil` instead. For instance,

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

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

3.1.2 Arrays

Arrays can only be defined in the uppermost (global) scope. They are defined with respect to some previously defined type, as in the following example:

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

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

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

In addition, any place in the program that semantically allows an expression of type array, should semantically allow `nil` instead. For instance,

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

Table 5: nil sent instead of an integer array is semantically allowed.

Note that allocating arrays with the new operator must be done with an *integral size*. Similarly, accessing an array entry is semantically valid only when the *subscript expression has an integer type*. Note further that if two arrays of type `T` are defined, they are *not* interchangeable:

`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: Non interchangeable array types.

3.2 Assignments

Assigning an expression to a variable is clearly legal whenever the two have the same type. In addition, following the concept in 3.1.1, 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 3.1.1 and 3.1.2, assigning `nil` to array and class variables is legal. In contrast to that, assigning `nil` to int and string variables is *illegal*. To avoid an overly complex semantics, we will enforce a strict policy of initializing data members inside classes: 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.

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 7: Assignments.

3.3 If and While Statements

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

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`.

3.5 Equality Testing

Testing equality between two expressions is legal whenever the two have the same type. In addition, following the same reason 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.

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 8: Equality testing.

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. Table 9 summarizes these facts.

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 9: Binary Operations.

3.7 Scope Rules

L defines four kinds of scopes: block scopes of if and while statements, function scopes, class scopes and the outermost global scope. 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.

Note that array type declarations and class type declarations can only be defined in the outermost (global) scope. Class type names and array type names must be different than any previously defined variable names, function names, class type names, array type names, primitive type names (int and string) and library function names (PrintInt and PrintString).

(Global) Functions can only be defined in the global scope. When a function is defined in the global scope, its name must be different than any class type name, array type name, previously defined (global) functions, previously defined (global) variables, primitive type names (int and string) and library function names (PrintInt, PrintString and PrintTrace). Following the same reason in 1, functions may only refer to previously defined types, variables and functions.

(Class) Methods can only be defined in the class scope. When a function/method is being called inside a class scope, the declaration of a method with that name is first searched in the class. Then, if no such method is found, the search moves to the super class, and so on. If still not found, the search goes to the global scope for a function with that name. If the declaration is missing there too, a semantic error is issued.

Resolving a variable identifier follows the same principal, with the slight difference that variables can be declared in all four kinds of scopes. Table 10 summarizes these facts.

3.8 Library Functions

L defines three library functions: PrintInt, PrintString and PrintTrace.
The signatures of these functions are as follows:

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

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. Either OK when the input program is correct semantically, or otherwise ERROR(*location*), where *location* is the line number of the *first* error that was encountered.

6 Submission Guidelines

The skeleton for this exercise can be found here. 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. 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*
- Generates an image of the resulting syntax tree
- Generates images which describe the changes in the symbol table

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

Table 10: Scope Rules.