Exercise 3

Compilation

Due 15/12/2019, before 14:00

1 Introduction

We continue our journey of building a compiler for the invented object oriented language Poseidon. 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 Poseidon 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 exercise 3, but the best time to design and implement this addition is exercise 3.

3 The Poseidon Semantics

This section describes the semantics of Poseidon. It avoids a too formal exposition, and instead provides a multitude of sematically valid and invalid example programs.

3.1 Scoping Rules

A major task of any semantic analyzer is the resolution of names (identifiers). The following paragraphs describe the scoping mechanism of Poseidon.

Poseidon defines four kinds of scopes: block scopes of if and while statements, function (or method) 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.

The outermost (global) scope is the *only* place where classes, arrays and (global) functions can be defined. Note (for the record) that most programming lanuages *do* allow nested declarations of classes (here is an example of a nested Java class from google/guava, and here is an example of a nested CPP class from LLVM). In that aspect, it is worth mentioning that languages like Java and CPP started supporting nested definitions of functions as lambda expressions (see refernces here and here). The outermost scope also holds the library function names (PrintInt, PrintString and PrintTrace) and the primitive types (int and string). Note that the restrictions regarding the global scope are actually *enforced at the sytactic level* (how?), so in itself, this paragraph will not yield too much work on your side ☺.

Identifiers with the same name can *not* exist in the same scope, except for the obvious case of overriding a method in a derived class. This means that, for instance, 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, PrintString and PrintTrace). In addition, it is explicitly stated here that methods overloading is *illegal* in Poseidon.

Methods overloading is *illegal* in Poseidon, in spite of the fact that it is a popular feature in most modern programming languages. Poseidon avoids it altogether, and when a class has two methods with the same name that differ only in their signatures, an error is issued. For similar reasons, a method in a class can *not* have the same name of a previously defined data member in that class. Following the same logic, a data member in a class can *not* have the same name of a previously defined method or data member.

Resolving a variable identifier follows the same principal, with the slight difference that variables can be declared in all four kinds of scopes. Table 11 summarizes these facts.

3.2 Types

The Poseidon 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. Recall, that defining classes and arrays is only possible in the uppermost (global) scope. The exact details follow.

3.2.1 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. Following the same concept, a method `M1` can *not* refer to a method `M2`, if `M2` is defined after

`M1` in the class. In contrast to all that, a method `M` *can* refer to a data member `d`, even 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;`
	`}`
		OK

Table 1: Referring to classes, methods and data members

Methods overloading is *illegal* in Poseidon. Similarly, it is illegal to define a variable with the same name of a previously defined variable, 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 Poseidon.

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

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

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

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

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.

3.3 Assignments

Assigning an expression to a variable is clearly legal whenever the two have the same type. In addition, following the concept in 3.2.1, 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 3.2.1 and 3.2.2, 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`
	`{`
	`    void Init() { tail := NEW IntList; }`
	`    int head;`
	`    IntList tail;`
	`}`
		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.

3.4 If and While Statements

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

3.5 Return Statements

According to the syntax of Poseidon, 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`.

3.6 Equality Testing

Testing equality between two expressions is legal whenever the two have the same type. In addition, following the same reason in 3.3, 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.

3.7 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; }`
	`}`
		OK

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.

3.8 Execution Flow

To avoid a too-early exposition of execution flow ideas, the semantic analyzer will *allow* control to reach end of non-void functions (or methods). Similarly, whenever the semantic analyzer will detect unreachable code, it will *not* consider it an error. Table 10 summarizes these facts.

1	`CLASS Father`
	`{`
	`    int foo() { }`
	`}`
		OK

1	`CLASS Father`
	`{`
	`    int foo(){if(5){return 8;}}`
	`}`
		OK

3	`void foo(Father f)`
	`{`
	`    return 80;`
	`    return 22;`
	`}`
		OK

Table 10: Execution Flow.

3.9 Library Functions

Poseidon 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 Poseidon 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 code for this exercise resides (as usual) in subdirectory EX3 of the course repository. COMPILATION/EX3 should contain a makefile building your source files to a runnable jar file called COMPILER (note the lack of the .jar suffix). Feel free to use the makefile supplied in the course repository, or write a new one if you want to. Before you submit, make sure that your exercise compiles and runs on a fresh installation of Ubuntu 18.04 LTS. This is the formal running environment of the course.

Execution parameters compiler receives 2 input file names:

InputPoseidonProgram.txt
OutputStatus.txt

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 := “80”;`
	`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 11: Scope Rules.

Program	::=	dec$^+$
dec	::=	varDec | funcDec | classDec | arrayDec
varDec	::=	ID ID [ASSIGN exp] ’;’
	::=	ID ID ASSIGN newExp ’;’
funcDec	::=	ID ID ′(′ [ID ID [’,’ ID ID]*] ′)′ ′{′ stmt [stmt]* ′}′
classDec	::=	CLASS ID [EXTENDS ID] ′{′ cField [cField]* ′}′
arrayDec	::=	ARRAY ID = ID ′[′ ′]′
exp	::=	var
	::=	′(′ exp ′)′
	::=	exp BINOP exp
	::=	[var ’.’] ID ′(′ [exp [’,’ exp]*] ′)′
	::=	[′−′] INT | NIL | STRING
var	::=	ID
	::=	var ’.’ ID
	::=	var ′[′ exp ′]′
stmt	::=	varDec
	::=	var ASSIGN exp ’;’
	::=	var ASSIGN newExp ’;’
	::=	RETURN [exp] ’;’
	::=	IF ′(′ exp ′)′ ′{′ stmt [stmt]* ′}′
	::=	WHILE ′(′ exp ′)′ ′{′ stmt [stmt]* ′}′
	::=	[var ’.’] ID ′(′ [exp [’,’ exp]*] ′)′ ’;’
newExp	::=	NEW ID | NEW ID ′[′ exp ′]′
cField	::=	varDec | funcDec
BINOP	::=	+ | − | ∗ | / | < | > | =
INT	::=	[1 − 9][0 − 9]* | 0

Table 12: Context free grammar for the Poseidon programming language.

Precedence	Operator	Description	Associativity
1	$:=$	assign	
2	$=$	equals	left
3	$<, >$		left
4	$+, -$		left
5	$*, /$		left
6	[	array indexing	
7	(	function call	
8	.	field access	left

Table 13: Binary operators of Poseidon along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.