Exercise 5

Due 9/6/2017

1 Introduction

This exercise implements a semantic analyzer for the StarKist programming language. The syntax of StarKist is given in StarKist.lex and StarKist.y (Flex and Bison) files. The semantic analyzer traverses the AST built by Bison, and checks for semantic errors. For example, it makes sure the type of the condition inside an `if` statement is `int`.

2 The StarKist Programming Language

The StarKist programming language is a simple invented object oriented language, that supports arrays, objects and inheritance. It is strongly typed, and uses a runtime mechanism to ensure the safety of array and object accesses.

2.1 Objects and Arrays Allocation

Objects and arrays are allocated on the heap. There is no need to free unused memory, as StarKist acts as if it were a part of a running environment that contains a garbage collection.

Arrays StarKist supports arrays of arbitrary types. That is, if `T` has already been defined, then

```
var array := T[80]
```

allocates an array of 80 consecutive `T`'s on the heap. Two dimensional arrays are also possible, though their definition is somewhat less straight forward:

```
type TARRAY = array of T
```

followed by

```
var matrix = TARRAY[3]
```

Objects Objects are allocated without the ability to call a constructor. This makes the interface for objects creation rather simple:

```
var dan := new Citizen
```

2.2 Records and Classes

Records are the equivalent of structures in C. They may contain an arbitrary number of (comma separated) fields, and may be recursive:

```
type IntList = {head:int, tail:IntList}
```

Classes are collections of fields and methods, and can *not* be recursive. As StarKist does not support forward declarations, a method m may refer to a data field d only if d is defined before m. Similarly, method m_2 may refer to method m_1 only if m_1 is defined before m_2. StarKist does *not* support method overloading, and so a class can not have two functions with the same name, even if their signature is different. The following example is illegal as it contains method overloading:

```
class G = {phoneNumber:int; function salary():int = 8;}
class F extends G = {age:int; function salary(p:int):int = 8+age;}
```

However, overriding a method in a derived class is clearly legal:

```
class G = {phoneNumber:int; function salary(p:int):int = 8;}
class F extends G = {age:int; function salary(p:int):int = 8+age;}
```

2.3 Subtyping

Inheritance induces a subtyping relation. If `F` extends `G`, then we say that `F` $\leq$ `G`. Clearly, the relation $\leq$ is transitive. If `F` $\leq$ `G`, then an expression of type `F` can be used whenever the program expects an expression of type `G`. Note that for every class, record or array type `F`, we have `nil` $\leq$ `F`.

2.4 Scope Rules

When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier in order. For example, it is possible that variable `x` is contained here in two different scopes:

```
class F = {x:int; function f(p:int)=let var x:=80 in x+p end;}
```

The `x` in `x+p` is the local variable. However, the next example involving a derived class and its super is illegal:

```
class G = {x:int; function salary():int = 8;}
class F extends G = {x:int; function swim(y:int):int = 600;}
```

Note that if `F` contained a function called `x` that would be illegal too. To

create a graph visualization of the AST, please install graphviz and run

```
$ dot -Tjpeg -o ./AST_Graph.jpeg ./AST_Graph.txt
```

from `EX5/LINUX_GCC_MAKE`

The grammar is given in Table 1. In addition, *all* row vectors have to be of

S	::=	solutionSet
solutionSet	::=	{rowVec} + SP({rowVecList})
	::=	SP({rowVecList})
	::=	{rowVec}
rowVecList	::=	rowVec, rowVecList
	::=	rowVec
rowVec	::=	(num, num)
	::=	(num, num, num)
	::=	(num, num, num, num)
num	::=	[op] int
	::=	[op] int div int
int	::=	0
	::=	$[1-9][0-9]^*$
div	::=	/
	::=	\
op	::=	+
	::=	−

Table 1: Context free grammar for the language of the solution set.

the same size (2, 3 or 4), denominators can not be zero, and denominators can not be negative.

3 Bison

Bison is an LALR(1) parser generator, which receives as input a context free grammar, and implements a parser for that grammar in a single C file. An overall example for using Bison is inside the row operations parser.

4 Input

The input for this exercise is a single text file that contains the solution set.

5 Output

The output is a single text file that should contain a single word: either OK when the solution set has correct syntax, or FAIL otherwise.

6 Submission Guidelines

The code for this exercise resides as usual in subdirectory EX4. The file SolutionSet.y, and possibly SolutionSet.lex should be the only file(s) you change. Please submit your exercise in your GitHub repository under COMPILATION/EX4, and have a makefile there to build a runnable program called Lini. To avoid the pollution of EX4, please remove all *.o files once the target is built. The next paragraph describes the execution of Lini.

Execution parameters Lini recevies 2 input file names:

Input.txt
Output.txt