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 Table 2, and is almost entirely implemented 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`. Or, it verifies that a variable is never used before it is declared. Note, that the check list for the semantic analyzer is not explicit. It follows from the various properties that make the StarKist language, and it is up to you to assemble and implement this check list.

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 Lexical Considerations

Identifiers may contain letters and digits, and must start with a letter.

The keywords in Table 1 can not be used as identifiers.

int	float	void	string	let
class	nil	new	of	for
if	then	do	in	end
extends				

Table 1: Reserved keyword of StarKist.

White spaces consist of spaces, tabs and newlines characters. They may appear between any tokens. Keywords and identifiers must be separated by a white space, or a token that is neither a keyword nor an identifier.

Comments in StarKist are similar to those used in the C programming language: a comment beginning with the characters // indicates that the remainder of the line is a comment. In addition, a comment can be a sequence of characters that begins with /*, followed by any characters, including newlines, up to the first occurrence of the end sequence */. Unclosed comments are lexical errors.

Integer literals may start with an optional negation sign $-$, followed by a sequence of digits. Non-zero numbers should *not* have leading zeroes. Though integers are stored as 32 bits in memory, they are artificially limited in StarKist to have 16-bits signed values between -2^{15} and $2^{15} - 1$. Integers out of this range are lexical errors.

2.2 StarKist syntax

The grammar of the StarKist programming language is shown in Table 2

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

The next example is *illegal* too, as a variable can not have the same name of function in the same scope:

```
class G = {var phoneNumber:int; function salary():int = 8}
class F extends G = {var salary:int; function Pay():int = salary}
```

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

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

2.4 Arrays, Objects and Records Allocation

Arrays, objects and records 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 must be initialized upon allocation using the following syntax:

```
var salaries:int := int[12] of 7800
```

StarKist supports arrays of arbitrary types, and if `T` is a non primitive type, then

```
var array := T[165] of nil
```

allocates an array of 165 consecutive `T`'s on the heap. Two dimensional arrays (or higher) are also possible, though their definition is somewhat less straight forward:

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

followed by

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

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

```
var dan := new citizen
```

Records must be initialized upon allocation using the following syntax:

```
var oren:citizen := student{100,100,nil,1976}
```

2.5 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`.

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

2.7 Binary Operators

Table 3 contains the list of supported binary operators, along with their associativity and precedence. Note, that binary operators are valid only between integers, and the resulting type of the operation is an integer too. The next (legal) example emphasizes the fact that (relational) operators like $<$ are treated in exactly the same way as "standard" operators like $+$:

```
var oren:int := 18<30
```

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`

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, the input StarKist program.

5 Output

The output is a single text file that should contain a single word: either OK when the StarKist program is correct, or FAIL(*location*) otherwise. *location* is the line number of the *first* error that was found.

6 Submission Guidelines

The code for this exercise resides as usual in subdirectory EX5 of the course GitHub. Currently, the grammar in Starkist.y contains a shift/reduce conflict,

so you should start by fixing this. Next, you need to add the relevant derivation rules and AST constructors for classes. Last, you should implement the missing parts of the Starkist semantic analyzer. The semantic analyzer resides in the file semant.c, and this is where most of your changes will occur. Please submit your exercise in your GitHub repository under COMPILATION/EX5, and have a makefile there to build a runnable program called compiler. Make sure that compiler is created in the same level as the makefile: inside EX5. To avoid the pollution of EX5, please remove all *.o files once the target is built. The next paragraph describes the execution of compiler.

Execution parameters compiler receives 2 input file names:

InputStarkistProgram.txt
OutputStatus.txt

S	::=	exp
exp	::=	letExp
	::=	forExp
	::=	callExp
	::=	assignExp
	::=	exp BINOP exp
	::=	LPAREN exp [';' exp]* RPAREN
	::=	INT | NIL | STRING | NEW ID | var
letExp	::=	LET dec [dec]* IN exp END
forExp	::=	FOR ID ASSIGN exp TO exp DO exp
callExp	::=	[var `'->'`] ID LP [exp [',' exp]*] RP
assignExp	::=	var ASSIGN exp
	::=	var ASSIGN ID LBRACE exp [',' exp]* RBRACE
	::=	var ASSIGN ID LBRACK INT RBRACK OF INT
	::=	var ASSIGN ID LBRACK INT RBRACK OF NIL
var	::=	ID
	::=	var `'->'` ID
	::=	var LBRACK exp RBRACK
tField	::=	ID ':' ID
cField	::=	VAR tField | funcDec
varDec	::=	VAR ID [':' ID] [ASSIGN exp]
typeDec	::=	TYPE ID EQ LB tField [',' tField]* RB
funcDec	::=	FUNC ID LP [params] RP [':' ID] EQ exp
classDec	::=	CLASS ID EQ LB cField [';' cField]* RB
dec	::=	funcDec | varDec | typeDec | classDec

Table 2: Context free grammar for the StarKist programming language.

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

Table 3: Binary operators of StarKist along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.