Nolife Compiler (Part 1):
A Context-Sensitive Analyzer
CS 5810: Fall 2018

Due Date: Monday, October 29, 2018 at *8am*

Purpose

This project is intended to give you experience building a context-sensitive analyzer. I have provided a `JavaCC` grammar file. If you wish to code in something other than Java, you may. However, you will need to use a different parser generator. I have a grammar for Nolife that is written for `ANTLR`. I believe that `ANTLR` has parser generators for Java, C and a few other languages. Besides the parser specification, I will provide a solution to part 1 of this project in Java and if you program in something else, you will not have that solution for part 2 of this project.

You will get experience building an abstract syntax tree and performing context-sensitive analysis. Read this document and the accompanying Nolife documents completely before embarking on this adventure.

Project Summary

Your task is to construct a context-sensitive analyzer that accepts the Nolife programming language. Your parser will construct an abstract syntax tree (AST) to represent the program structure. After the AST is constructed, another pass must be made over the AST to perform type checking. The output of your program will be a listing of the abstract syntax tree for the input program or a list of error messages (if there is a type error).

The Scanner and Parser

I have provided a JavaCC file that contains the grammar for Nolife (`Nolife.jj`) and a main routine in the file `Nolife.java`.

1. I recommend that you develop your project in Eclipse and use the JavaCC plugin to generate the parser.

2. Add actions to be performed on the various after each rule in the grammar has been recognized. The code in these actions will construct the abstract syntax tree. The form of the abstract syntax tree is specified in a document entitled "An Abstract Syntax Tree for Nolife". That document also specifies the format for listing the tree.

Context-Sensitive Analysis

Compiling a Nolife program requires a large amount of knowledge that cannot be detected in a context-free parse. This suggests a compiler design that includes a separate context-sensitive analyzer. The analyzer should perform the following tasks:

1. assign a type to each expression and subexpression,

2. find any context-sensitive errors in the program,

These tasks can be performed in two separate passes over the tree, or they can be compressed into a single tree-walking pass. Of course, since each pass requires use of an accurate block-structured symbol table, a single-pass implementation will be more efficient.

The next two subsections describe these tasks in more detail.

Table for `+`, `-`, `*`		
	int.	float
int.	int	float
float	float	float

Table for `OR`, `AND`			
	char	int	float
char	char	*error*	*error*
int	*error*	int	int
float	*error*	int	float

Table for $<, \leq, \geq, >, =, <>$			
	char	int	float
char	int	*error*	*error*
int	*error*	int	int
float	*error*	int	int

Figure 1: conversion tables for mixed mode expressions

Mixed Type Expressions

Nolife supports three basic data types: *integer*, *floating point*, and *character*. Each expression and subexpression has a type that can be determined at compile time. Your compiler should determine (1) the type of each subexpression, (2) where coercions must be inserted, and (3) where invalid type combinations exist.

Figure 1 gives type conversion tables for several of the Nolife operators.

1. The type of a subscripted name is wholly determined by the array's type declaration — it is independent of the type of the subscript expression.
2. Similarly, the type of a function call is determined by the function's definition rather than by the types of any actual parameters at the call site.
3. Assignment uses an idiosyncratic and asymmetric rule. For a left hand side of type integer or float, the right hand side is converted to the type of the left hand side. If the left hand side is of type character, the right hand side must have type character.
4. The result of a `NOT` always has type integer. The operand of `NOT` must be of type integer.
5. Relational operators always produce an integer. Comparisons between characters and numbers make no sense; they are illegal. Comparisons between integers and floats produce integer results. To perform the comparison, the integer is converted to a float.
6. MOD is only defined on integers and always produces an integer.

Your compiler will need to insert the appropriate conversions and report any expressions that would require an illegal coercion.

Context-Sensitive Errors

Your compiler should detect the following context-sensitive errors and report them to the user on `stderr`, `cerr` or `System.err` as appropriate for your implementation language.

1. a variable is referenced but not declared
2. a variable is declared multiple times in a single scope
3. a variable is declared, but never referenced
4. any type mismatch (*illegal mixed type expression*)
5. any mismatch between actual parameters at a call site and the definition of the called procedure
6. a constant-valued subscript that is outside the declared bounds of an array
7. incorrect number of dimensions in a variable reference
8. a procedure call that invokes a function (*incorrectly discarding the return value*)

Operator	Precedence
$*$, `MOD`	5
$+$, $-$	4
<,<=,=,>=,>,<>	3
NOT	2
AND, OR	1

Figure 2: Operator precedences in Nolife

9. a function call that invokes a procedure (*incorrectly using a non-existent return value*)
10. a return statement whose type does not match the function definition
11. a return statement in the main program
12. a function with no return statement

This list should not be considered exhaustive. Extra credit will be given for any extra errors detected. Submit with your compiler a file containing the error and a README file explaining the type of additional errors you catch. Points will be given on a subjective evaluation of significance.

Nolife Specification

The syntax of Nolife is specified in a document entitled "Nolife: A Language For Practice Implementation".

1. Operator precedences in Nolife are specified in Figure 1. Multiplication has the highest priority, `AND` and `OR` have the lowest. This already handled in the grammar provided.
2. The scope of a name is the region of the program in which the name can be used. Variables declared in a subprogram are only visible within that subprogram. They obscure an identically named variable in the surrounding scope. The scopes of distinct subprograms are disjoint – a name declared in one subprogram is not visible inside another subprogram.

   The scope defined by the main program is called the global scope. Names declared in the global scope are accessible from the point of declaration to the end of the program, including the body of the main program and any subprograms that do not redeclare the same name. All variables declared in the main program are in the global scope.
3. Function names are in the global scope. This has two important implications. First, a function can be called from any other function, provided that the calling function has not redefined that name. (This is true, even if the function being called appears later in the source text.) Second, any function can be called recursively.
4. The lifetime of a variable is limited to the execution of the procedure (main program or function) that defines the scope in which it is declared. Thus, the value of a variable x declared in function f ceases to exist after the function returns.

Requirements

Your code will be tested on 32-bit Ubuntu 16.04 and MUST work there. You will receive no special consideration for programs which "work" elsewhere, but not under 32-bit Ubuntu 16.04.

Your code should be well-documented. If you use Eclipse, submit a zip file containing your Eclipse project. You must include a jar file named nlc.jar that includes where your main routine is. I expect to type `java -jar nlc.jar foo.nl` and have the AST or type errors dumped to the console.

If you choose not to use Eclipse, you must submit all of your files, INCLUDING a `Makefile` to build your compiler. I will just type "make" and expect everything to "come out right." (Come out right in this context means I can type "`nlc foo.nl`" and I will get the AST of foo.nl or the type errors dumped to the console.

There is no hidden data for this project. All input is provided in a file name `ParserTestfiles.zip`. I have also provided the grading sheet in the file `NolifePart1Gradesheet.docx`.

Your program will be graded on whether it produces a correct AST for all correct programs and whether it catches the required context-sensitive errors.