ASSIGNMENT 3

CS 473: Compiler Design / Fall 2022

Description

In this assignment, you will build the Abstract Syntax Tree (AST) and perform semantic analysis for the language Albatross. The language is described in a separate document. Your analysis should reject invalid programs with type errors, that use variables/functions not defined, or that call functions with the incorrect number of arguments. You should use the lexical parser that you implemented in Assignment 1 and the syntactic parser that you Implemented for Assignment 2.

The Albatross Language

The Albatross language is defined in a separate document.

Building and Using the AST

You should edit the semantic rules for the parser so they build the AST. The provided files `ast.h` and `ast.c` define the structure of the AST, with all the required nodes and data-structures to represent the program.

The rest of the assignment performs several traversals of the AST that you build during parsing, as described below:

Symbol Resolution

You must implement the first traversal in files `semantic_analysis_symbols.h` and `semantic_analysis_symbols.c`. Your submission should populate a global symbol table with global variables, a function table with all the functions, and a local symbol table for each function with local variables and arguments. Then, your submission should check that all variables and functions used were defined.

Type Checking

You must implement the second traversal in files `semantic_analysis_types.h` and `semantic_analysis_types.c`. Make sure to collect typing information during symbol resolution so you can use that information now. This traversal should ensure that all types match: unary and binary operations are performed on integers, variable assignment matches the variable type, and function calls have matching formal and actual argument types.

Computing Function Frames

You must implement the third traversal in files `frames.c` and `frames.h`. This traversal should compute the function frames. For each function, the frame contains a table for the types of the arguments and local variables, another table that maps each argument/local variable to an integer (described below), and the return type of the function.

Your submission should assign numbers to arguments as follows. The first argument is number 0. The second argument is number 1, and so on.

Your submission should assign numbers to local variables as follows. The first local variable has the number of the last argument plus 1. The second local variable has the number of the first local variable plus one, and so on.

For instance, the following function: `f(int i, int j) { int local1; int local2; … }` has the frame:

```
0. i
1. j
2. local1
3. local2
```

Note that the data-structure that keeps the frames also has tables for the type of each argument and local variable. If you fill these tables early (i.e., during symbol resolution), you can then use it during type-checking.

Printing Output

File `printAst.c` provides the full implementation that prints the results described below. It is a good idea to inspect this file closely as it shows a full AST traversal that you can reuse to implement the functionality required as described above.

Name and Type Errors

Your submission should terminate with error for programs with any of the following contents:

- Global variables with the same name, regardless of type
- Function arguments with the same name as existing global variables, regardless of type
- Function arguments with the same name on the same function, regardless of type
- Function arguments with the same name on different functions **are not an error**
- Local variables with the same name as existing global variables, regardless of type
- Local variables with the same name as function arguments for the same function, regardless of type
- Local variables with the same name on the same function, regardless of type
- Local variables with the same name on different functions **are not an error**
- Function with the same name, regardless of signature
- Functions and variables with the same name **are not an error**
- Inconsistent use of types in expressions and statements (e.g., assigning a string to an int variable, or using a string as the condition for an if statement)

For programs that contain name and type errors, your submission should exit with the **return code 3**. You can print an error message describing what is wrong with the input program, but that is optional.

Note that crashing on a program that should be rejected does not count as rejecting that program. Your submission should not crash or exhibit any memory errors for any possible input programs.

Format of the output

The printer in file `printAst.c` generates the following output:

- Variable declaration: `Variable declared "<name>" type <type>`
- Function declaration: `Function declared "<name>" returns <type>`
    - Arguments in declaration: `Argument "<name>" type <type> position <N>`
    - Local variables in declaration: `Local variable "<name>" type <type> position <N>`
- Variable read: `Variable read "<name>" type <type>`
- Argument/Local variable read: `Argument/local read "<name>" type <type> frame position <N>`
- Variable written: `Variable written "<name>" type <type>`
- Argument/Local variable written: `Argument/local written "<name>" type <type> frame position <N>`
- Function called: `Function called "<name>" returns <type>`

For instance, for the following Albatross program:

```
var s string := "";

fun f2 int (a1 int, a2 string) {
  a1 := a1;
  a2 := s;
  return a1;
}

return f2(0,s);
```

The expected for this is:

```
Variable declared "s" type string
Function declared "f2" returns int
    Argument "a1" type int position 0
    Argument "a2" type string position 1
Argument/local read "a1" type int frame position 0
Argument/local written "a1" type int frame position 0
Variable read "s" type string
Argument/local written "a2" type string frame position 1
Argument/local read "a1" type int frame position 0
Variable read "s" type string
Function called "f" returns int
```

Entry Point

You will modify file `lexer.lex` with all the rules for the lexer from Assignment 1.

You will modify file `parser.y` with all the rules from Assignment 2, and modify the semantic actions to create the AST as described in this document.

You will modify files `semantic_analysis_(symbols|types).[ch]` and `frame.[ch]`, which already contain an empty AST traversal, with code to perform the right actions when traversing the AST.

You will not modify any other file in the provided code.

Due Date and Resubmission Policy

This assignment is due on **October 8 2022** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** for undergraduate students. You can resubmit your assignment until **October 15 2022** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment3` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Graduate students

On top of the behavior described above, graduate students have to implement support for reading/writing arrays as described in the Albatross language specification. You can add rules to your grammar to accept variable declarations as arrays without initialization, and functions that accept arrays as arguments.

Each regular test is only worth 8% for graduate students. The remaining 20% will be graded as described below.

This language extension will be graded via one video screen-cast (**through Gradescope**) that answers the questions below by explaining how your code. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

1. Does your submission support symbol resolution on expressions that read from arrays? (e.g, arr[n], arr[function(n,m))]
2. Does your submission support type checking on expressions that read from arrays? (e. arr[n+10])
3. Does your submission support symbol resolution on expressions that write to arrays? (e.g, arr[n] := 10; , arr[function(n,m))] := arr[m];)
4. Does your submission support type checking on expressions that write to arrays? (e.g, arr[n] := 10; , arr[function(n,m))] := arr[m];)

The maximum length for the video is 5 minutes, instructors will stop watching at the 5 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf(“expected result”).`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.