CMPU 331 · Compilers · Fall 2019

Project - Semantic Analyser

Due Tuesday, November 26, 2019 by 11:59pm

1 Overview

The four project milestones will direct you to design and build a compiler for the DL language, which is a simple subset of the C language. Each assignment will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each assignment will ultimately result in a working compiler phase which can interface with other phases. You will be implementing the projects in Python.

For this assignment, you will write a semantic analyzer that performs a set of checks on the AST that you built in the parsing phase.

You may work either individually or in pairs for this project.

2 Getting Started

You will submit your code through GitHub Classroom, so the first step is to clone a copy of the skeleton files from GitHub. In Moodle, under the topic "Semantic Analysis", you'll find a link to "Semantic Analyzer on GitHub". Follow that link, select your team (or create a new team), and then click the "Accept this assignment" button to create your own private copy of the skeleton files. Your copy will be a GitHub repository located at `https://github.com/vassar-cs331-2019b/semantic-yourusername` (substitute your GitHub username). From that page, you can copy the URL for cloning by clicking the green button "Clone or download", and then paste the URL onto the command-line as an argument to the `git clone` command.

```
$ git clone https://github.com/vassar-cs331-2019b/semantic-yourusername.git
```

The clone command will create a directory named `semantic-yourusername` on your machine, and in that directory you'll find the following files:

- `LICENSE` - This file contains the open source license for the code. Below the copyright line with my name on it, add another copyright line with your name. (Or, just copy this file from your lexer project repository.)
- `setup.py` - This file contains metadata about the code. Replace my name and email with yours. (Or, just copy this file from your lexer project repository.)
- `runtests.py` - This script is a simple test runner. You can call it with `python3 runtests.py` to run all the tests.
- `sly/lex.py` and `sly/yacc.py` - These files are libraries used in the lexer and parser. You won't make any changes to these files, they are only included in the skeleton so you don't need to install them.
- `dl/ast.py` - This library contains the class definitions for the AST node types you will use in your semantic analyzer.

- `dl/lexer.py` and `tests/test_lex.py` - These are a complete implementation and tests for the lexer stage of the compiler. You can (optionally) replace them with your own implementation from the lexer project submission, together with any tests you added.
- `dl/parser.py` and `tests/test_parser.py` - These are a complete implementation and tests for the parser stage of the compiler. You can (optionally) replace them with your own implementation from the parser project submission, together with any tests you added.
- `dl/semantic.py` - This file contains a skeleton for a DL semantic analyzer. There are comments indicating where you need to fill in code, but you are welcome to make any modifications to the skeleton. The skeleton is functional, and will pass two tests, but it doesn't do much.
- `tests/test_semantic.py` - This file contains tests for the semantic analyzer, including at least one test for each type AST node, and one test of a longer code example. The provided tests are not exaustive, you may want to add more as you work. You won't be graded on the tests you add, but testing more thoroughly can increase your confidence that your semantic analysis is working as it should.
- `tests/simple.dl` - This file is a code example in the DL language, which is used by one of the tests.

The CS lab workstations have all the software you need installed already.

3 Semantic Analysis Implementation

In this assignment you will write a semantic analyzer for DL. The assignment makes use of two resources: a tree traversal library and a collection of class definitions for abstract syntax tree (AST) nodes. The output of your semantic analyzer will be an AST. This will be the same AST you constructed using semantic actions of the parser generator, but your semantic analyzer will add more information about symbols and types to the AST.

You might find it helpful to refer to these sources as you work:

- Appendix A - "The DL Language" starting on page 247 of the textbook "A Practical Approach to Compiler Construction" by Des Watson
- Table 5.1 - "Parse tree nodes" on page 120 of the textbook "A Practical Approach to Compiler Construction" by Des Watson

3.1 Visitor Methods for AST Nodes

The skeleton includes class definitions for a collection of AST node types that you used in the parser, and will use again in the semantic analyzer. These are inspired by the node types defined in the textbook (on page 120), but not exactly the same. A description of the AST node types was provided in the instructions for the parser project assignment, so you will likely find it useful to refer back to that description.

For each AST node type, the semantic analyzer will need a single method that knows how to perform any semantic analysis checks. For example, for `Variable` AST nodes, you will need to implement a method in the semantic analyzer named `visit_Variable`, which receives a single argument named `node`, and checks whether the variable for that node was declared.

The skeleton semantic analyzer includes a visitor method for every AST node type, but some of them haven't been implemented yet. For the ones that still need to be implemented, you will need to delete

the single line `pass` from the body of the method (which just tells Python "do nothing for this method, but also don't complain that the method isn't implemented yet"), and replace it with your own code. Below is a list of the visitor methods included with the skeleton and a description of what they should do.

3.1.1 visit_Integer

Already implemented. Just sets the inferred type of the node to `int`.

3.1.2 visit_Program

Already implemented. Creates a new scope in the symbol table, visits `node.declarations` if there are any declarations, visits `node.body`, and then exits the scope.

3.1.3 visit_Block

Already implemented. Just iterates through all the statements in the block and visits each one.

3.1.4 visit_Declarations

Already implemented. Just iterates through all the declarations and visits each one.

3.1.5 visit_VariableDeclaration

Partially implemented. Currently just makes a lower-case type name from `node.var_type` (the type of the node). You will need to:

- Iterate through `node.variables`.
  - Call the built-in Python function `isinstance` to check if the type of each declaration is `Variable` or `ArrayIndex`.
  - If the declaration is a `Variable`, add a symbol to the symbol table by calling `self.st.add_var_symbol` (see `dl/symbols.py` for the definition of this method).
  - If the declaration is an `ArrayIndex`, add a symbol to the symbol table by calling `self.st.add_array_symbol` (see `dl/symbols.py` for the definition of this method).

3.1.6 visit_Integer

Already implemented. Just sets the inferred type of the node to `int`.

3.1.7 visit_Variable

You need to implement this:

- Look up the `node.name` in the symbol table by calling `self.st.find_symbol`.
- If the symbol is found, call the built-in Python function `isinstance` to check if the returned symbol is a `VariableSymbol` or an `ArgumentSymbol`.
- If the symbol is isn't found, or the symbol isn't a `VariableSymbol` or an `ArgumentSymbol`, throw an `UndeclaredVariableError`.
- Set the inferred type of the `node` to the `symbol.type`. Take a look at `visit_Integer` for an example of setting the inferred type.

3.1.8 visit_ArrayIndex

You need to implement this:

- Look up the `node.var.name` in the symbol table by calling `self.st.find_symbol`.
- If the symbol is found, call the built-in Python function `isinstance` to check if the returned symbol is an `ArraySymbol`.
- If the symbol is isn't found, or the symbol isn't an `ArraySymbol`, throw an `UndeclaredVariableError`.
- Set the inferred type of the `node` to the `symbol.type`.

3.1.9 visit_BinOp

You need to implement this. Take a look at how `visit_Assign` and `visit_RelOp` are implemented.

3.1.10 visit_RelOp

Already implemented. Visits `node.left` and `node.right`, then checks that the inferred types for both arguments are either `int` or `bool`. If the type check is successful, sets the inferred type for the `node` to `bool`.

3.1.11 visit_Assign

Already implemented. Visits `node.left` and `node.right`, then checks that the inferred types for both arguments are `int`. If the type check is successful, sets the inferred type for the `node` to `int`.

3.1.12 visit_Print

Already implemented. Just visits the node for the single argument to the `print` statement.

3.1.13 visit_Read

Already implemented. Just visits the node for the single argument to the `read` statement.

3.1.14 visit_Return

Already implemented. Just visits the node for the single argument to the `return` statement.

3.1.15 visit_If

Already implemented. Just visits the nodes for the condition, true block, and else block of the `if` statement.

3.1.16 visit_While

Already implemented. Just visits the nodes for the condition and body of the `while` statement.

3.1.17 visit_FunctionDeclaration

Partially implemented. Currently just calculates how many arguments the function declaration has. You will need to:

- Add a symbol to the symbol table for the function by calling `self.st.add_func_symbol` (see `dl/symbols.py` for the definition of this method).
- Create a new scope in the symbol table for the function.
- If there are any `node.args`, iterate through them, and add each one to the symbol table by calling `self.st.add_arg_symbol`.
- Visit `node.vars` if there are any variable declarations for the function.
- Visit `node.body`.
- Exit the scope for the function.

3.1.18 visit_FunctionCall

You need to implement this:

- Look up the `node.name` in the symbol table by calling `self.st.find_symbol`.
- If the symbol is found, call the built-in Python function `isinstance` to check if the returned symbol is a `FunctionSymbol`.
- If the symbol is isn't found, or the symbol isn't a `FunctionSymbol`, throw an `UndeclaredFunctionError`.
- Set the inferred type of the `node` to `int`. (DL doesn't declare return types, so we will assume that all functions return a single `int` value.)
- Visit `node.args` if the function call has arguments.

- Check that the number of arguments in the function call (zero or more) is the same as the number of arguments from the function declaration (zero or more), and if they don't match, throw an `UndeclaredFunctionError`.

3.1.19 visit_Arguments

Already implemented. Just iterates through all the arguments and visits each one.

3.2 Skipped Tests

In the skeleton, many of the tests are marked as `skip`, because we don't expect these tests to pass yet. Marking the tests as skipped makes them display a clean message when you run `runtests.py`, like this:

```
... skipped 'Arrays not checked yet'
```

When you start working on the productions related to a particular test, first delete the line that marks that test to be skipped, from the test file `tests/test_semantic.py`. The line will look like this:

```
@unittest.skip("Arrays not checked yet")
```

After you delete the skip marker, the test will fail, and display an ugly error message. The error message will give you some clues about what's missing from your semantic analysis. Once you've implemented all the productions needed to pass a particular test, the status report at the end of the test run will change from something like:

```
FAILED (errors=1, skipped=4)
```

To something like:

```
OK (skipped=4)
```

Passing tests is how you know you've implemented your semantic analysis correctly. When all the tests are passing with no skips, you're done.

In general, you'll find it easier to start working on the tests towards the top of the `tests/test_semantic.py` file, since they test features that are easier to implement. The tests grow more complex towards the end of the file, and the very last test parses the DL code example in `tests/simple.dl`.

4 Submitting

To submit your work, `git add` any files you changed, then `git commit` them with a sensible commit message about what you changed, and finally `git push` the changes back to GitHub.

```
$ git add dl/semantic.py
$ git commit -m "Implemented semantic analysis for arrays"
$ git push origin master
```

It's a good idea to commit your work frequently as you go, so you have a backup if anything goes wrong. Only the last version you submit before the deadline will be graded.