Assignments

Compilers Assignment 2 - Returned

Title	Compilers Assignment 2
Student	Martin Atwebembire
Submitted Date	25-Sep-2014 00:45
Grade	**78.0 (max 100.0)**
History	Thu Sep 25 00:45:55 SAST 2014 Martin Atwebembire (atwmar001) submitted

Instructions

CSC3003S Compilers – Assignment 2: Abstract Syntax Tree,
Semantic Analysis and Error Reporting

Introduction

This assignment will follow on from assignment 1. You will be expected to create an Abstract Syntax Tree (AST), do some basic Semantic Analysis and Error Reporting for a made-up programming language called *mini nulang*, which is based on a reduced grammar for *nulang* (excluding functions).

The Syntactic Analyser (parser) program from Assignment 1 should be modified to take *mini nulang* files (**.mnl*) and should generate an appropriate Abstract Syntax Tree which is output to the screen and to a corresponding **.ast* file.

The Abstract Syntax Tree should then be analysed to do simple semantic analysis using a basic Symbol Table stored in a simple data structure, and then report on any errors which should be output to the screen and also to a corresponding **.err* file. Errors from previous stages, lexical analysis and syntactic analysis, should also be output to the screen and written to the **.err* file.

Tools

The programming language and compiler tools for this assignment can be either

- Java with SableCC, JavaCC, or JFlex and Cup
- Python with PLY
- C++ with Flex and Bison

Although it is strongly recommended that you use Java with SableCC, or Python with PLY, since that is what the reference solution(s) will be implemented in and they have established methods for generating an Abstract Syntax Tree.

Input, Output and Testing

The input **.mnl* source code file should be specified as a command line parameter when your programs are run, e.g.

parse my_program.mnl

errors my_program.mnl

The parser should as before check the tokens conform to the grammar defined below and construct and output an abstract syntax tree as flat text using depth-first traversal, visiting the root, then children from left to right onto the screen and also in a corresponding file, *my_program.ast*.

The semantic analysis program should follow on from parsing and check the program for basic semantic errors (specified below) and then report the first error identified - whether lexical, syntactic or semantic - outputting it to the screen and also in a corresponding file, *my_program.err*.

Download the *parse_mininulang_samples.zip* file containing **.mnl* code input files and their corresponding output **.ast* files (also some **.str* files for comparison purposes), and the *error_mininulang_samples.zip* file containing **.mnl* code input files and their corresponding output **.err* files, indicating what the output should look like and can be used to test your program.

Mini Nulang Grammar

The grammar uses the notation N∗, for a non-terminal N, meaning 0, 1, or more repetitions of N. Bold symbols are keywords and should form their own tokens, and other tokens are in italics.

Which nulang grammar rules have been removed can be viewed here: nulang_reduced.

Program	→ Statement*	// this asterisk indicates closure
Statement	→ *identifier* = Expression	
Expression	→ Expression + Term	
	→ Expression - Term	
	→ Term	
Term	→ Term * Factor	// this asterisk indicates multiplication
	→ Term / Factor	
	→ Factor	
Factor	→ *(* Expression *)*	
	→ *float*	
	→ *identifier*	

Abstract Syntax Tree

The Abstract Syntax Tree is a reduced version of the Parse (Syntax) Tree with unnecessary nodes like e.g. operators and punctuation removed. Single child nodes can be kept, although strictly they should be removed for a more optimal AST.

Semantic Analysis

Mini Nulang (and Nulang) has the following two properties, it is:

- "dynamically typed", in the sense that variables do not have to be explicitly created with a particular type, e.g. *float val*. Once a variable is assigned a value it is considered to be created.

- "functional", in the sense that variables should not be mutable. So a variable can only legally be assigned a value/expression once. If a variable is assigned another value/expression later on, it should be considered a semantic error, having been already defined.

So your semantic analysis should perform two checks:

1. If a variable(identifier) is created/defined on the left hand side of an assignment, it should check if it has already been defined, in which case it should generate an appropriate semantic error.
2. If a variable(identifier) is used in the right hand side of an assignment it should check if it has been defined already, and if not it should generate an appropriate semantic error.

So semantic analysis should be traverse the AST and maintain a simple Symbol Table data structure, which can be a simple data structure, but often is a stack to handle more sophisticated scoping checks for more complex languages.

Error Reporting

If one of the semantic errors specified above, or another error from a previous stage, lexical analysis or syntactic analysis, occurs then this should be output to the screen and to a corresponding *.err file indicating the type of error with some appropriate detail about the error, including the line number, description and the problematic token/identifier/variable. (To simplify the problem we'll assume the program file has no empty lines.)

Due

09h00, Thursday 25 September 2014

Submit

Submit all the source code needed to run your program(s) including a README file containing instructions on how to compile and run your program(s).

Notes

- As with the previous assignment, an elegant solution could be about 250 lines of code and other specification rules you write yourself in total. (The compiler tool(s) will generate quite a bit of "backend" code for you.) But it is tricky and requires mastering the basics of the compiler tool(s), so keep that in mind and start early.
- A sample solution to the previous assignment will be made available here (CompilersAssignment1SampleSolution) once extension and late submission deadlines have passed.
- For the semantic error checking, the SableCC tree traversal Visitor pattern can be used.

Submitted Attachments

- ATWMAR001.zip (329 KB; 25-Sep-2014 00:45)

Additional instructor's comments about your submission

Please comment your code. *_* Semantic errors not recognised.

Back to list

- Centre for Innovation in Learning and Teaching

- University of Cape Town

- Powered by Sakai
- powered by Sakai