CSC3003S Compilers – Assignment 2:
Semantic Analysis, Error Reporting, Intermediate Representation, Compilation

Introduction

This assignment will follow on from assignment 1, based on the made-up programming language called *ula* (unconventional language). You will be expected to do some basic Semantic Analysis and Error Reporting, and then do Intermediate Representation (IR) code generation and compilation using LLVM.

The abstract syntax tree generated from an *ula* input file should 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.

The abstract syntax tree generated from an *ula* input file tree should then be traversed to generate LLVM Intermediate Representation (IR) code which is output to the screen and also to a corresponding **.ir* file. That code file should also then be run to produce the output to the screen and to a corresponding **.run* file.

Finally, for the ambitious, you can write a program which outputs the assembly language code generated by LLVM for an *ula* code file.

Tools

Python with llvmlite (llvmlite.pydata.org), should be the programming language and compiler tool used for this assignment. LLVMlite is a lightweight Python binding for LLVM.

A. Semantic Analysis and Error Reporting [30 marks]

Input, Output and Testing

The input **.ula* source code file should be specified as a command line parameter when your semantic analysis and error checking program *errors_ula.py* is run, e.g.

errors_ula.py my_program.ula

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, parse(syntactic) or semantic - outputting it to the screen and also in a corresponding file, *my_program.err*.

Download the *ula_error_samples.zip* file containing **.ula* code input files and their corresponding output **.err* files, indicating what the output should look like and can be used to test your program.

Semantic Analysis

Ula 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, 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 parsing(syntactic analysis), occurs then this should be output to the screen and to a corresponding **.err* file. It should indicate the type of error (lexical, parse or semantic) and on which line number it occurred, e.g.

lexical error on line 2

To simplify the problem we’ll assume the program file has no empty lines.

B. Intermediate Representation and Compilation(Running) [50 marks]

LLVMlite

For this part of the assignment you will be using llvmlite(llvmlite.pydata.org) which is a lightweight Python binding for LLVM. It has been installed in the senior labs. When developing your assignment ensure you are using the correct version of Python which has the llvmlite library installed:

Microsoft Windows: C:\Program Files (x86)\Miniconda3\python.exe

Ubuntu Linux: /usr/local/miniconda3/bin/python3

You are able to install this on your personal computers, but this is done at your own risk. No responsibility will be taken for the failure of a personal llvmlite installation. As always your assignments are expected to be developed and run on the senior lab computers. I have though provided notes for personal llvmlite installation at the end of this assignment should you still chose to install llvmlite on a personal computer.

Input, Output and Testing

The input **.ula* source code file should be specified as a command line parameter when your intermediate code generation and compile/run programs are run, e.g.

ir_ula.py my_program.ula

run_ula.py my_program.ula

The intermediate representation *ir_ula.py* program should generate intermediate representation code outputting it to the screen and also in a corresponding file, *my_program.ir*.

The compilation/run *run_ula.py* program should compile and run the intermediate representation code outputting the result of the last assignment expression of the input file to the screen and also in a corresponding file, *my_program.run*.

Download the *ula_irrun_samples.zip* file containing **.ula* code input files and their corresponding output **.ir* and **.run* files, indicating what the output should look like and can be used to test your program.

Intermediate Representation

You will need to traverse the abstract syntax tree and use the llvmlite library to generate the Intermediate Representation code. Consider the following two references to do this:

- ir_code_gen.py example attached
- LLVMlite reference (llvmlite.pydata.org), particularly The IR Layer section

When generating the IR code, ensure the following (as in the ir_code_gen.py example):

- the module name is “ula”,
- you define a necessary function with the name “main”,
- create a block in the function with the name “entry”

Note: It may be sensible to first generate a more compact abstract syntax tree, from what was generated in Assignment 1.

Compilation/Running

To compile and run from IR, consult the following reference:

- LLVMlite reference (llvmlite.pydata.org), particularly The LLVM Binding Layer section, noting “Compiling a trivial example” code example

C. NerdZone: Assembly Language [20 marks]

Consult the LLVM Reference and write a program *asm_ula.py* which produces a corresponding assembly language **.asm* file for a particular **.ula* file passed as a command line parameter. Sample **.asm* files will also be found in the *ula_irrun_samples.zip* file.

Due

09h00, Friday 25 September 2015

Submit

Submit *errors_ula.py*, *ir_ula.py*, *run_ula.py* and *asm_ula.py* to the automarker in a single ZIP file called 'ABCXYZ123.zip' (where ABCXYZ123 is YOUR student number).

Notes

LLVMlite Installation

Overall installation instructions here: http://llvmlite.pydata.org/en/latest/install/index.html

The simplest way to do it is:

1. Download MiniConda with Python 3.4 for the necessary environment here: http://conda.pydata.org/miniconda.html

2. Install MiniConda.

3. Run this command line instruction, which downloads all the necessary dependencies:

    conda install --channel numba llvmlite

[Note: This will install the latest version of llvmlite 0.7.0 . Do not run the "conda install llvmlite" command line instruction as specified in the installation instructions, since it will install the older 0.6.0 version which won't work for our purposes.]

4. When compiling/running your Python programs, ensure that you are referencing the MiniConda (Continuum Analytics) installation of Python, otherwise your LLVMlite programs won't compile/run.