CSC4180 Assignment 4: Compiler Frontend for Oat v.1

Release Date: April 8, 2025
Due Date: April 28, 2025
TA in Charge: Mr.LiuYuxuan

1 Summary of Tasks to Complete

- **Implement A Simple Semantic Analyzer for Oat v.1 (60%)**
  - [**Basic Requirement (30%)**]: Implement semantic analysis for 2-level scope, and do IR generation for highly limited instructions. Should pass testcases 0, 1, and 2.
  - [**Avanced Requirement (30%)**]: Implement semantic analysis for multi-level scope (if-else, for loop, while loop), and do IR generation for the three instructions. Should pass testcases 3, 4, and 5.
  - You will receive some bonus credits if you could come up with more test cases targeting other tricky scenarios to test the robustness of the semantic analyzer in A4's scope. To share your test cases with others, you can create a pull request and submit it to our GitHub repository CSC4180-Compiler.
- **Get Familiar with LLVM's Optimization Passes Through llvm-tutor (30%)**
  - [**Installation**]: Download llvm-tutor of the specific commit under your personal directory in the cluster machine. The link refers to an old commit for llvm-10 installed in our cluster so that you have no problem reproducing the results. Feel free to switch to the latest version of llvm-tutor and install it in your PC for recent updates and more optimization passes.
  - [**Have a try with llvm-tutor's written passes (15%)**]: Follow the step-by-step instructions to reproduce the results of each optimization passes provided by llvm-tutor. Pick one pass that you are interested in, explain its functionality and attach the screenshot(s) of executing this pass in your report.
  - [**Have a try with LLVM's built-in passes (15%)**]: Check the list of LLVM's built-in passes, find one pass that you are interested in, explain its functionality and attach the screenshot(s) of executing this pass in your report.
- **Report (10%)**: Submit a technical report about this assignment.

2 Introduction

In Assignment 2 and 3, you have implemented scanner and LL(1) parser by hand, and finally outputs the abstract syntax tree (AST) for Oat v.1 Language. However, the AST cannot be used to do IR generation yet since there are still some information missing, such as the type for each literal or variable, and the scope for each variable. Recall the overall structure of compiler, there are totally 4 stages in front-end phase, and there are still 2 stages to go, which are semantic analysis and IR (Intermediate Representation) generation. Semantic Analysis aims to fill the missing type and scope information as we said, and produces an agumented AST, which is used to do IR generation.

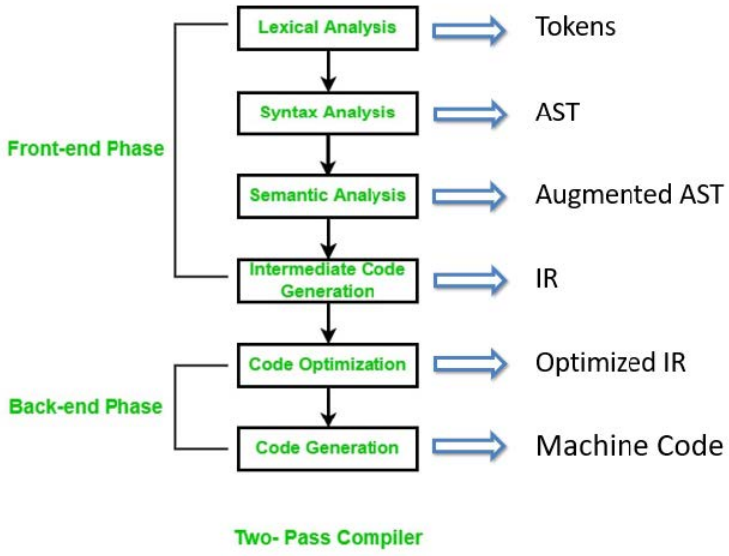

Figure 1: Overview of Compiler

3 Template Program [Not Necessary to Follow]

The template of this assignment has implemented various functionalities, and hopefully it can save you some time of dirty and time-consuming works to focus on sementic analysis and IR generation. You don't need to modify the following things unless you are pretty clear about what you are doing.

Download template from GitHub Repository

3.1 Summary of Template Program

1. Flex scanner for Oat v.1 Language
2. Bison parser for Oat v.1 Language, which outputs AST as .dot file
3. Transfer from .dot file to Python TreeNode structure, and TreeNode Visualization
4. Definitions of SymbolTable, NodeType, and DataType data structures in Python
5. Runtime.c file for Oat v.1's Built-In Functions like (print_int)
6. Basic Framework and TODO instruction comments for semantic analysis and IR codegen
7. Semantic analysis handler and IR codegen handler function definitions for binary and unary operators

3.2 File Structure of Template Program

```
csc4180-a4-template
|-
|--- llvmlite-examples Sample Programs to Generate LLVM IR with llvmlite
|-
|--- testcases          Directory of testcases-related materials
|---  | ast             Directory to store AST-related materials
|---  | test1-6.oat     Six Testcase Programs in Oat.v1
|-
|--- main.cpp           Program entrance of cpp-based scanner and parser
|--- scanner.l          Flex scanner
|--- parser.y           Bison parser
|--- node.cpp           Definition of ASTNode and util functions
|--- node.hpp           Definition of ASTNode and util functions
|--- Makefile           Build executable for cpp-based scanner and parser
|--- get_input_ast.sh   Script to get input AST .dot and png files
|-
|--- a4.py              [TODO]: all the logics of semantic analysis and IR generation here
|--- runtime.c          Build-In Functions of Oat v.1 Language Implemented by C
|--- runtime.ll         Compiled LLVM IR file for runtime.c by Clang
|--- verify.sh          Script to execute the compiler frontend for testcases in batch
|-
```

How to Compile and Execute Template

Simply execute **bash ./verify.sh** and you can successfully get the input AST for your work under /testcases/ast.

```
bash ./verify.sh
```

4 Details of Basic Requirement

4.1 Semantic Analysis Part

The AST you need to manipulate looks like the following. Every node has a token class, some of them (ID, INTLITERAL, STRINGLITERAL) has lexeme indicating the identifier or value. However, all nodes have type NONE, which means their types are unknown. Another challenging issue for the following Oat program is that there are two variable x, one defined globally and other defined locally. They must be distinguished by your semantic analyzer so that the global x equals to 5, and the local x equals to 10 when printing them out.

4.1.1 Input AST for test0.oat before semantic analysis

```
global x = 5;
int main() {
    print_int(x);
    var x = 10;
    print_int(x);
    return x;
}
```

4.1.2 Output AST for test1.oat after semantic analysis

The output AST should be something like the following, with types for necessary nodes, and rename the node as **lexeme-scope_id** so that it is pretty clear where the variable is declared and defined.

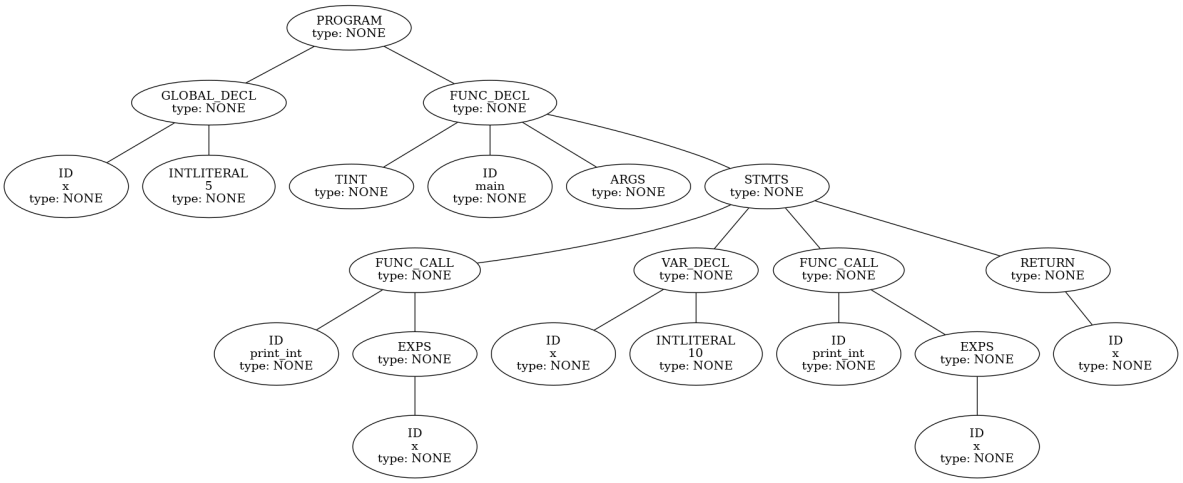

Figure 2: Input AST without Semantic Analysis for the above Oat program

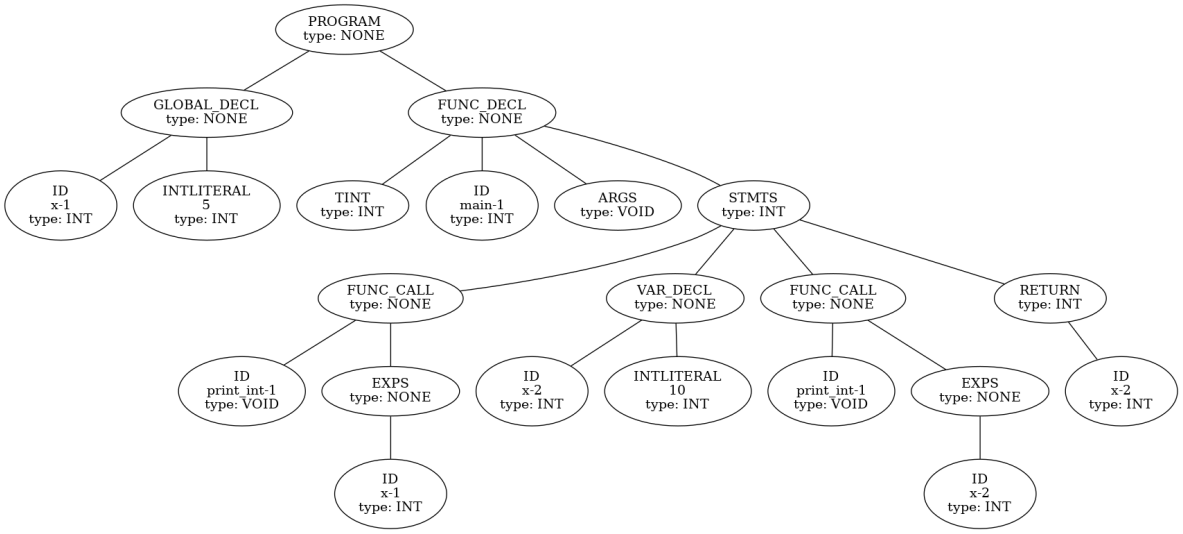

Figure 3: Output AST with Semantic Analysis for the above Oat Program

4.1.3 What You Need to Do

1. Understand the SymbolTable data structure defined in the template, or create one by yourself
2. Analyze each type of node (FUNC_DECL, VAR_DECL, ...) about its children
3. Write handler functions for each type of node, and set data type to it

4.1.4 Notices (README!)

- **In this part, you only need to manipulate at most 2 scopes at a time, one global scope and the other local scope for each function declaration.**

4.2 IR Generation Part

You used to implement simple LLVM IR generation for Micro Language, but at that time, you simply generate them as strings. This method works only for very simple language, and it becomes nearly impossible when it comes to Oat v.1. Therefore, in this assignment, you need to learn a powerful tool called llvmlite, which is a lightweight LLVM-Python binding for writing JIT (Just-in-time) compilers. Using it can make our life easier compared to working with LLVM C API directly.

4.2.1 What You Need to Do

1. Have a look at the sample programs of LLVMlite to briefly understand how program should be implemented to generate the LLVM IR you want.

   Some important concepts include: module, IRBuilder, basic block

2. Analyze each type of node (FUNC_DECL, VAR_DECL, ...) about its children

3. Write handler functions for each type of node to generate corresponding IR

4. There are

4.2.2 Notices (README!)

- **In this part, you only need to support literal data types in LLVM IR, that is, int, bool, and string. Reference or even nested reference types are for Bonus part.**
- **In this part, you only need to deal with basic alloca-load-store within one single basic block. No need to consider conditional branch and multiple basic blocks inside one function.**

5 Details of Advanced Requirement

Conditional branches and loops are commonly used in most programs. To make your compiler more powerful, in this part, you are required to extend the capability of your compiler to these features. Works need to be done in both semantic analysis and IR generation.

5.1 Semantic Analysis Part

You need to consider cases where there are more than 2 scopes in the stack at a time, since if-else statement and for / while loops create new scopes on top of the function declaration. A even more challenging issue is that nested scope may exist, and you need to smartly use recursion to solve that.

5.2 IR Codegen Part

Implementing IR code generation for if-else, for and while loops need some extra efforts because there can be more than 1 basic blocks inside a function, and the program may jump to other basic blocks by some conditions. Refer to the example llvmlite programs to see how to generate such IR codes.

6 Useful Materials

6.1 Learning Materials

Semantic Analysis Handout Notes from Stanford University, CS143, Summer 2012

[Recommended] Semantic Analysis: Implementation, slides from University of Waterloo

LLVMlite Official Documentation

[Highly Recommended] LLVMlite Sample Programs of IR Generation

What is Static Single Assignment (SSA), Wikipedia

LLVM IR Tutorial by Vince Bridgers et.al from LLVM Developers Conference, Brussels 2019

Other learning materials for LLVM's optimization passes provided by llvm-tutor

6.2 VSCode Extensions for Development

- **LLVM**: Syntax highlighter for LLVM IR (Extension ID: RReverser.llvm)
- **Oat Intellisens**: Syntax highligher for Oat Language (Extension ID: tlcyr4.oat)
- **Yash**: Syntax highlighter for flex/bison (Extension ID: daohong-emilio.yash)

7 Bonus: (Extra Credits 10%)

If you want to challenge yourself and do something harder, here are some alternatives for you:

7.1 Integrate Your Scanner and Parser Implemented in A2 and A3

The template uses flex/bison to do scanning and parsing, but you can also replace them with your own implementation in A2 and A3.

7.2 Write Your Own LLVM Optimization Pass

After going through the instructions from llvm-tutor, you may want to practice writing a LLVM optimization pass by your own. The pass can be either for analysis or for transformation. To do this, you may consider mimicing how llvm-tutor's built-in passes are implemented, and implement your pass in llvm-tutor project. After implementing your opt pass, you may need to test it with some test cases and see the outputs.
Note: the LLVM optimization pass does not need to target Oat v.1 language. It can just target C/C++ for easier generation of test cases.

7.3 Make Your Compiler More Powerful

- Support reference(array) or even nested reference data types in IR code generation.
- Support NEW instructions with size initialization and value initialization
- Support error handling and report for semantic analyzer
- Support type check for function arguments types for semantic analyzer
- Any other features or instructions…

Extra Credit Policy

The extra credits are not going to be added to your final grade since this is an elective course and many of you are interested in compiler and want to do some advanced work. In other words, the extra credits you earn cannot be added to the grade of any programming assignment to make it higher. However, these credits can serve as a proof of your interest and hard work for this course, and I will try my best to offer as high A- rate as I can, probably higher than 40%, by consulting with Prof.CHUNG.
Note: the bonus credits cannot make up for points lost due to late submission.

Submission & Evaluation

7.4 Grading Scheme

- **Basic Requirement of Semantic Analyzer**: 30%

  As long as you can pass testcase 0, 1, and 2.oat
- **Advanced Requirement of Semantic Analyzer**: 30%

  As long as you can pass testcase 3, 4, and 5.oat

- **Journey to LLVM's Optimization Passes**: 30%

  The report should include clear explanations to the two optimization passes that you pick with convincing materials (e.g., screenshots or command line interface records) of executing them.

- **Technical Report**: 10%

  The report doesn't need to be very long, and you don't need to answer any questions. Just write down your thoughts and feelings when implementing this assignment. You can also include what you have tried and learned during this process.

- **Bonus:** 10%

  Refer to section bonus for more details. The grading of this part will be very flexible and highly depend on the TA's judgement. Please specify clearly what you have done for the bonus part so that he do not miss anything.

Policy of Late Submission

- Late submission within 10 minutes after then DDL is tolerated for possible network issues during submission.
- 10 Points deducted for each day after the DDL (11 minutes late will be considered as one day, so be careful)
- Zero point if you submitted your project late for more than two days