CSE450
Translation of Programming Languages

Introduction

Please take 4 index cards, one of each color
(4 cards per student)
Overview

- Administrative Items
- History of Compilers
- Compiler Construction Example
- Tools Overview
Course Information

- Meeting Time: **Tuesday/Thursday 3-4:20 p.m.**
- Meeting Place: **1279 Anthony Hall**
- Pre-requisites: **CSE331 and CSE320 or ECE331**
  
  I expect a solid understanding of **Python**, along with basic knowledge of **algorithms and data structures** (searching, sorting, linked lists, tree structures). You **need** programming experience.

- Web Page: [http://www.cse.msu.edu/~cse450/](http://www.cse.msu.edu/~cse450/)
Recommended Textbooks

- **Compilers: Principles, Techniques, and Tools**  
  by Aho, Lam, and Ullman  

- **Flex & Bison**  
  by Levine  
  (Optional)
Instructors

- **Instructors:** Dr. Joshua Nahum
- **Office:** 1124 Engineering Building
- **Office Hours:** Tuesdays/Thursdays 2-3pm (hour before lecture)
  - Also by appointment.
- **Teaching Assistant:** Hayam Abdelrahman
  - **Office Hours:** Tuesday / Thursdays 11am-1pm, Simpsons Lab - 3353 Engineering
- **Teaching Assistant:** Grant King
  - **Office Hours:** Monday/Wednesday 10am-noon, Simpsons Lab, 3353
- Office hours subject to change, check website and Piazza for announcements.
Where are the instructor's emails?

- Email isn't the best way to reach myself or your TA. All communication should be directed through the class forum on Piazza (linked to from the website).

- Advantages for Piazza over email:
  - Faster: Both instructors can respond to queries
  - Consistency: Both instructors can see each other's responses
  - Easier: Piazza makes it easier to format questions containing code
  - Useful: We can make good questions visible to the class if others may benefit

- All emails directed to the instructors will be directed to ask via Piazza instead.
Grading

- **Projects**: There will be 8 projects. Each will be worth between 6% and 8% of your final grade.

- **Homeworks**: There will be at least 12 short homeworks; the top 10 homeworks will each count for 2% of your grade.

- **Exams**: There will be a single exam (tentative scheduled for November 15th (about 2/3 of the way through the semester).

- **Extra Credit**: Most projects and the exam will have opportunity for extra credit. I will also take into account class participation when assigning your final grade.

60% - Projects

20% - Homework

20% - Exam

Extra Credit
Flash Card Question: Is there a final exam for this class?

Instructions: Using the index cards provided, hold up the card corresponding to your answer.

Yes, it is worth 20 points

Yes, it is worth 10 points

No

No, but there is an exam in November
Projects

- Basically, one big project in 8 parts. Building a complete compiler for a simple, web-based graphical interface language.

- Projects will be due at 10pm on their due date. Late projects will lose 1 point of the grade per 24 hours. No project can be more than 4 days late.

- Projects will be compiled and tested on CSE linux systems. If your code doesn't run, you will receive no points for the project.

- We'll go over the Project details next lecture.
Projects

- Project 1: Lexical Analysis (a Scanner) for a simple calculator language with basic math and printing.
- Project 2: Add Syntax Analysis as the next phase of our simple compiler (a Parser)
- Project 3: Output our simple language into Intermediate Code.
- Project 4: Add flow control (conditionals and loops) and new data types (char) to our language and add basic Semantic Analysis to our compiler
- Project 5: Add arrays and strings to our language.
- Project 6: Generate Target Code (the virtual assembly language Tubecode)
- Project 7: Add functions
- Project 8: Optimize the output from our compiler!
HackerRank Homeworks

- In addition to the 8 projects, there will be weekly short homework assignments.
- These will be implemented with HackerRank, a website used to conduct programming contests and interviews.
- The homeworks will be distributed (via your MSU email) on Friday night and always due on Thursday at 10pm.
- No late turn in allowed, if you Haven't clicked submit by the deadline, you may receive a 0 for that assignment.
- There will be at least 12 homework assignments, each worth 2 points, but only the top ten assignments will count for your grade.
Academic Honesty

- **Do your own work!** You may use the web as a resource as long as you give credit and don’t try to pawn off your work on someone else.

- **Do not** copy another student's code or exam answers, use code independently implemented by someone else without attributing credit, provide false information to the instructor about matters related to the course, or facilitate another student in any of these activities.

- We will be using plagiarism detection software. Violation of these rules will result in a final grade of **0.0**.
Why should you care?

- Compilers draw together all of the theory and techniques that you’ve learned about in most of your previous computer science courses.
- You will learn how to write a simple compiler to solve the kinds of problems you may face in a career as a programmer.
- You will gain a deeper understanding of how compilers work and be able to write better code.
- You will learn techniques to write other useful tools, such as parsers, interpreters, and debuggers.

**Warning:** It will be LOT of programming!
Overview

- Administrative Items
- Overview and History of Compilers
- Compiler Construction Example
- Tools Review
History of Compiler Development

1953 IBM develops the 701 EDPM (Electronic Data Processing Machine), the first general purpose computer, built as a "defense calculator" in the Korean War.

No high-level languages were available, so all programming was done in assembly language.
History of Compilers

As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly language programming.

“Much of my work has come from being lazy.”

-- John Backus
History of Compilers

As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly language programming.

In 1953, John Backus came up with the idea of "speed coding", and developed the first interpreter. Unfortunately, this was 10-20 times slower than programs written in assembly.

He was sure he could do better.
History of Compilers

In 1954, Backus and his team released a research paper titled "Preliminary Report, Specifications for the IBM Mathematical FORMula TRANslating System, FORTRAN."

The initial release of FORTRAN I was in 1956, totaling 25,000 lines of assembly code. Compiled programs ran almost as fast as handwritten assembly!

Projects that had taken two weeks to write now took only 2 hours. By 1958 more than half of all software was written in FORTRAN.
Modern Compilers

Compilers have not changed a great deal since the days of Backus. They still consist of two main components:

The front-end reads in the program in the source languages, makes sense of it, and stores it in an internal representation…

…and the back-end, which converts the internal representation into the target language, perhaps with optimizations. The target language used is typically an assembly language, but it is often easier to use a more established, higher-level language.
Modern Language Translation

Computers are now powerful enough that interpreters are common and effective. Translation strategies are varied and depend mostly on how much pre-processing is done before and during execution, and how CPU specific the program is. The simplest translation schemes are:

- **Interpreters** run a program "as is" with little or no pre-processing, but no changes need to be made to run on a different platform

- **Compilers** take time to do extensive pre-processing, but will generally run a program 2 to 20 times faster
Many languages use a combination of compiler and interpreter to get some of the many benefits of each.

- Java and Microsoft’s .NET are hybrid languages that compile into virtual/intermediate assembly languages (Java bytecode, CIL), which can then be interpreted on any computer.
- Most modern interpreters perform “just in time” compilation to convert routines into optimized native code when they are first used.
- Often, languages have both compilers and interpreters available, although one style tends to be much more common.
3 Minute Break

- Once per class (time permitting), I'll give a 3 minute break. Use this time to play Pokémon, talk, and/or use the facilities.

Overview

- Administrative Items
- Overview and History of Compilers
- Compiler Construction Example
- Tools Review
Structure of a Compiler

Source Language

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Int. Code Generator

Intermediate Code

Code Optimizer
Target Code Generator

Target Language

Front End

Back End
Example Compilation

Source Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Int. Code Generator
Intermediate Code
Code Optimizer
Target Code Generator
Target Language
Example Compilation

Source Code:
\[ \text{cur\_time} = \text{start\_time} + \text{cycles} \times 60 \]
Source Code:
cur_time = start_time + cycles * 60

Lexical Analysis:
ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
Example Compilation

Source Code:
\[ \text{cur\_time} = \text{start\_time} + \text{cycles} \times 60 \]

Lexical Analysis:
ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)

Syntax Analysis:
```
ASSIGN
  ID(1)
  ADD
    ID(2)
    MULT
      ID(3)
      INT(60)
```
Example Compilation

Source Language → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Int. Code Generator → Intermediate Code → Code Optimizer → Target Code Generator → Target Language

Syntax Analysis:
ASSIGN
  ID(1)
  ADD
    ID(2)
    MULT
      ID(3)
      INT(60)
Example Compilation

Source Language
- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Int. Code Generator

Intermediate Code
- Code Optimizer
- Target Code Generator
- Target Language

Syntax Analysis:
ASSIGN
  ID(1) ADD
    ID(2) MULT
      ID(3) int2float
        INT(60)

Semantic Analysis:
ASSIGN
  ID(1) ADD
    ID(2) MULT
      ID(3) int2float
        INT(60)
Example Compilation

Semantic Analysis:

ASSIGN

ADD

MULT

int2float

ID(1)

ID(2)

ID(3)

INT(60)
Example Compilation

Semantic Analysis:

ASSIGN
  ID(1)
ADD
  ID(2)
  ID(3)
MULT
  int2float
  INT(60)

Intermediate Code:

temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
Example Compilation

Intermediate Code:
```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```
Example Compilation

Intermediate Code:
- temp1 = int2float(60)
- temp2 = id3 * temp1
- temp3 = id2 + temp2
- id1 = temp3

Optimized Code (step 0):
- temp1 = int2float(60)
- temp2 = id3 * temp1
- temp3 = id2 + temp2
- id1 = temp3
Intermediate Code:
\[
\begin{align*}
temp1 &= \text{int2float}(60) \\
temp2 &= \text{id3} \times temp1 \\
temp3 &= \text{id2} + temp2 \\
id1 &= temp3
\end{align*}
\]

Optimized Code (step 1):
\[
\begin{align*}
temp1 &= 60.0 \\
temp2 &= \text{id3} \times temp1 \\
temp3 &= \text{id2} + temp2 \\
id1 &= temp3
\end{align*}
\]
Example Compilation

Intermediate Code:
 temp1 = int2float(60)
 temp2 = id3 * temp1
 temp3 = id2 + temp2
 id1 = temp3

Optimized Code (step 2):
 temp2 = id3 * 60.0
 temp3 = id2 + temp2
 id1 = temp3
Example Compilation

Intermediate Code:
\[
\begin{align*}
\text{temp1} &= \text{int2float}(60) \\
\text{temp2} &= \text{id3} \times \text{temp1} \\
\text{temp3} &= \text{id2} + \text{temp2} \\
\text{id1} &= \text{temp3}
\end{align*}
\]

Optimized Code (step 3):
\[
\begin{align*}
\text{temp2} &= \text{id3} \times 60.0 \\
\text{id1} &= \text{id2} + \text{temp2}
\end{align*}
\]
Intermediate Code:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```
Example Compilation

Intermediate Code:
\[
\begin{align*}
\text{temp1} &= \text{int2float}(60) \\
\text{temp2} &= \text{id3} \times \text{temp1} \\
\text{temp3} &= \text{id2} + \text{temp2} \\
\text{id1} &= \text{temp3}
\end{align*}
\]

Optimized Code:
\[
\begin{align*}
\text{temp1} &= \text{id3} \times 60.0 \\
\text{id1} &= \text{id2} + \text{temp1}
\end{align*}
\]

Target Code (PowerPC-ish):
\[
\begin{align*}
lfd & \quad r1, \text{memory(id3)} \\
lfd & \quad r2, \text{const(60.0)} \\
fmul & \quad r1, r1, r2 \\
lfd & \quad r2, \text{memory(id2)} \\
fadd & \quad r1, r1, r2 \\
stfd & \quad r1, \text{memory(id1)}
\end{align*}
\]
Overview

- Administrative Items
- Overview and History of Compilers
- Compiler Construction Example
- Tools Review
Tools

- **Lex** uses regular expressions to generate a scanner for you that will convert streams of characters to streams of more meaningful tokens.

- **Yacc** uses a set of simple rules to convert from a stream of tokens to a structured abstract syntax tree.

- The **PLY** Python library adds bindings to Lex and Yacc to Python.

- **Git** is a version control system to facilitate source code management and distributed development. We will be using GitHub.com.

- **Piazza.com** is a web-based discussion forum for classes that we will use to facilitate course interaction outside of the classroom.
Why Lex and Yacc?

In structured programming, there are two tasks that occur over and over:

- Divide input into meaningful units
- Discovering the relationship among those units

These two operations turn out to be easy to automate.
Lex and Yacc

**Lex** reads in a collection of regular expression, and uses it to write a C or C++ program that will perform lexical analysis. This program is almost always *faster* than one you can write by hand.

**Yacc** reads in the output from Lex (a stream of tokens) and parses it using its own set of rules (based on grammars). This is almost always *slower* than a hand written parser, but *much faster* to implement. Yacc stands for "Yet Another Compiler Compiler".
Git

Instructions: Using the index cards provided, hold up the card corresponding to your answer.

How much experience do you have with distributed version control (DVC)?

- Never Heard Of DVC / git
- Heard Of DVC / git
- Have Used DVC / git
- Proficient With DVC / git
Final Notes

- Everyone will receive an email from HackerRank inviting them to a demo homework assignment (not worth any points).
- I'll announce that the email has been sent out on Piazza.
- Also, on the website (and on Piazza) there is a link to a mandatory survey. Please fill this out ASAP, as I need to know your GitHub usernames to create your private repos for the projects.
- Bring your index cards to every lecture. I'll have replacements if you forget them.