CSE4305: Compilers for Algorithmic Languages
CSE5317: Design and Construction of Compilers

Leonidas Fegaras
General Course Information

Instructor: Leonidas Fegaras
Office: GACB 115 (General Academic Classroom Bldg)
Phone: (817) 272-3629
Email: fegaras@cse.uta.edu
Office hours: Tuesday and Thursday 2:00-3:00pm

GTA: Talha Nassar

Course web page: http://lambda.uta.edu/cse5317/
Catalogue Description

- Review of programming language structures, translation, and storage allocation.
- Introduction to context-free grammars and their description.
- Design and construction of compilers including lexical analysis, parsing and code generation techniques.
- Error analysis and simple code optimizations will be introduced.
Objectives

• The goal of this course is to give a working knowledge of the basic techniques used in the implementation of modern programming languages.

• The course is centered around a substantial programming project: implementing a complete compiler for a realistic language.

• Students successfully completing this course will be able to apply the theory and methods learned during the course to design and implement optimizing compilers for most programming languages.
Reasons to Take this Course

• understand better
  – programming languages (principles & semantics)
  – computer architecture and machine code structure
  – the relation between source programs and generated machine code

• get a good balance of theory & practice

• complete a substantial programming project (a compiler for a realistic language)
  – get programming experience and become a better programmer
  – learn how to work in groups
Prerequisites

• Prerequisites:
  – CSE3302 (Programming Languages)
  – CSE3315 (Theoretical Concepts)
  – CSE3322 (Computer Architecture I)

• Students must:
  – have knowledge and programming experience with Java;
  – be familiar with the functions of modern computer architectures and be able to program in an assembly language;
  – be familiar with data structure concepts and algorithms (such as lists, trees, sorting, hashing, etc).

• Students without adequate preparation are at substantial risk of failing this course.
Textbook

- Required Textbook and Notes:
  - Lecture Notes, available at http://lambda.uta.edu/cse5317/
- Lecture slides are based on notes
- You may find the following texts useful for additional background and explanation:
Grading

• The final grade will be based on
  – 30% project
  – 20% first midterm exam
  – 20% second midterm exam
  – 30% final exam (comprehensive)
• The course work will be the same for graduates and undergraduates.
• Final grades will be assigned according to the following scale:
  – A: score >= 90
  – B: 80 <= score < 90
  – C: 70 <= score < 80
  – D: 60 <= score < 70
  – F: score < 60
• Sometimes, I use lower cutoff points, depending on the overall performance of the class.
• After the first grades are posted, you can check your grades online at the course web page.
Reading Assignments

- Completing reading assignments before the class period in which the material is discussed is essential to success in this class.
- Not all the assigned material will be covered in class, but you will be responsible for it on exams.
Exams

• All exams are closed-book and closed-notes.
• The second midterm exam will cover the material of the second part of the course only, while the final exam will cover the material from the first lecture up to and including the last lecture.
• Makeup exams will be given only when the instructor (at least 3 days before the exam) has approved the request to change the exam time. Approval will be given for illness, sickness or death in the family only.
Project

- The course project is to construct a compiler for a small programming language and will involve: lexical analysis, parsing, semantic analysis (type-checking), and code generation for a MIPS architecture.
- This project will be done in Java. You may use your own PC but your programs should work correctly on gamma.
- The project is to be completed in seven stages spaced throughout the term and will be done by groups of 3 students.
- Project reports will be marked 20%-off per day. No further extensions will be allowed. No excuses, no exceptions.
Cheating

- You are allowed to collaborate with students of your project group only. **No copying is permitted.** Cheating involves giving assistance to or receiving assistance from members of other groups, copying code from the web, etc.
- The punishment for cheating is a zero in the assignment and will be subject to the university's academic dishonesty policy.
- If you have any questions regarding an assignment, see the instructor or teaching assistant.
Special Accommodations

• If you require an accommodation based on disability, I would like to meet with you in the privacy of my office, during the first week of the semester, to make sure you are appropriately accommodated.
Project

• The project will be done in groups of three students. It is your responsibility to find two other students and organize a project team.

• The course project is to construct a compiler for a small programming language, called PCAT.

• It will involve:
  – lexical analysis
  – parsing
  – semantic analysis (type checking)
  – code generation for a MIPS architecture.

• The project is to be completed in seven stages spaced throughout the term.
Survival Tips

- Select your teammates very carefully. Your project grade will depend on them. Choose teammates whose abilities complement yours. For example, you may be good in Java and this person may be good in computer architecture and assembly programming. That way your group will be strong in all aspects of this project.

- It's up to you to decide how to divide the project work among your teammates. It's highly unprofessional to come to me and complain about your teammates. You should meet, solve your differences, and divide the work as a professional team.

- Your project grade will not depend on your abilities alone, but on how well your team achieves all the above tasks.
Survival Tips (cont.)

• Start working on programming assignments as soon as they are handed out. Do not wait till the day before the deadline. You will see that assignments take much more time when you work on them under pressure than when you are more relaxed.

• Design carefully before you code. Writing a well-designed piece of code is always easier than starting with some code that "almost works" and adding patches to make it "really work".
Platform and Tools

- You will do your project on gamma using Java.
  - there are many on-line manuals for Java (see the project web page).
- To make coding easier in Java, you are required to use the Gen package to build abstract syntax trees and intermediate representation trees.
- You will also use a MIPS code simulator, called SPIM, to run the assembly code generated by your compiler.
Platform and Tools (cont.)

• You can do the project on your own PC, but
  – you need to make sure that your programs compile and run correctly on gamma before you submit them electronically
  – if your programs do not compile on gamma, you will get a zero in the assignment.

• To install the project on your own Linux or Windows PC, you
  – install Sun's J2SE SDK (the Java runtime/compiler)
  – install SPIM (the MIPS emulator)
  – download the System.jar archive that contains the CUP, JLex, and Gen classes
  – download the project and compile it.
Program Grading

- Programs will be graded according to their correctness, style, and readability.
- Programs should behave as specified in the assignment handouts.
- Bad data should be handled gracefully; your program should never have run-time errors like dereferencing a null pointer or using an out-of-bounds index.
- Special cases should be handled correctly.
- Unnecessarily inefficient algorithms or constructs should be avoided; however, efficiency should never be pursued at the expense of clarity or simplicity.
- Programs should be well documented, modular, and flexible, i.e. easy to modify. Indentation should reflect program structure. Use meaningful identifiers.
Program Grading (cont.)

• Avoid static variables and side effects as much as possible. You should never use side effects during the semantic actions of a parser.
• The grader should be able to understand the program without undue strain.
• I will provide some test programs, but these programs will not test your compiler exhaustively. It is your responsibility to test every statement in your program by some piece of test data. Thorough testing is essential to establish the reliability of your code.
• Don't even think about adding fancy features until the required work is completely debugged. A correctly working simple program is worth much more (both in this class and in actual practice) than a fancy program with bugs.
Cheating

- You are allowed to collaborate with students of your project group only. **No copying is permitted.**
- Cheating involves giving assistance to or receiving assistance from members of other groups, copying code from the web, etc.
- You are **required** to use the Gen package (using the Meta class interface for tree construction and pattern matching). It will be taken as cheating if you use your own data structures or interface (since this would mean that you have copied the code from elsewhere).
- The punishment for cheating is a zero in the assignment and will be subject to the university's academic dishonesty policy.
- If you have any questions regarding an assignment, see the instructor or teaching assistant.
Deliverables

- Project phases:
  1) Lexical Analysis: Due M 2/7, worth 6% of your project grade.
  2) Parsing: Due W 2/16, worth 14% of your project grade.
  3) Abstract Syntax: Due M 3/7, worth 14% of your project grade.
  4) Type-Checking: Due M 3/28, worth 18% of your project grade.
  5) Simple IRs: Due M 4/11, worth 18% of your project grade.
  6) Rest of IRs: Due W 4/20, worth 16% of your project grade.
  7) Instruction Selection: Due W 5/4, worth 14% of your project grade.

- The due time of each project is the midnight of the indicated due day
- You will hand-in your project source code electronically
- You may hand-in your source files as many times as you want; only the last one will be taken into account
- Projects will be marked 20%-off per day. So, there is no point submitting a project more than 4 days late! No further extensions will be allowed. No excuses, no exceptions.
Solution

• There is a solution jar archive `Solution.jar`
  – It provides all the classes (scrambled, without the Java source code), so you can compare the output of your program with that of the solution.
  – For each project phase, you can compare the output of your program with that of the solution.

• You can run the solution PCAT compiler over a test PCAT program, say `tests/hanoi.pcat`, using the command
  ```
solution 7 hanoi
  ```
inside your project directory.

• If you mess up a project phase you can still do the next project phases by removing the appropriate source files from your directory.
  – That way, the missing classes will be copied from the `Solution.jar` file, rather than compiled from your sources.
By Monday January 31

• Find a team: Stay after class and talk to your classmates
• Each team will send one email to the GTA with information about the team members (firstname and lastname only)
• If you cannot find a team or need to add a third member, I will help you after class (on Monday 1/31)
What is a Compiler?

- We will mostly study:

  high-level source code  \(\rightarrow\)  compiler  \(\rightarrow\)  low-level machine code

eg, Java program
- easy to understand
- user-friendly syntax
- many high-level programming constructs
- machine-independent
- variables, procedures, classes, ...

eg, MIPS code
- hard to understand
- specific to hardware
- registers & unnamed locations
Architecture

Compiler:

source program → compiler → assembler → linker → loader → result

assembly code → machine code → machine code

libraries

data

Interpreter:

source program → interpreter → result

data

Java uses both a compiler (javac) and an interpreter (java)
## Many Other Translators

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Translator</th>
<th>Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>LaTeX</td>
<td>Text Formater</td>
<td>PostScript</td>
</tr>
<tr>
<td>SQL</td>
<td>database query optimizer</td>
<td>Query Evaluation Plan</td>
</tr>
<tr>
<td>Java</td>
<td>javac compiler</td>
<td>Java byte code</td>
</tr>
<tr>
<td>Java</td>
<td>cross-compiler</td>
<td>C++ code</td>
</tr>
<tr>
<td>English text</td>
<td>Natural Lang Understanding</td>
<td>semantics (meaning)</td>
</tr>
<tr>
<td>Regular Expressions</td>
<td>JLex scanner generator</td>
<td>a scanner in Java</td>
</tr>
<tr>
<td>BNF of a language</td>
<td>CUP parser generator</td>
<td>a parser in Java</td>
</tr>
</tbody>
</table>
Challenges

• Many variations:
  – many programming languages (eg, FORTRAN, C++, Java)
  – many programming paradigms (eg, object-oriented, functional, logic)
  – many computer architectures (eg, MIPS, SPARC, Intel, alpha)
  – many operating systems (eg, Linux, Solaris, Windows)
Qualities of a Compiler

• the compiler itself must be bug-free
• it must generate correct machine code
• the generated machine code must run fast
• the compiler itself must run fast (compilation time must be proportional to program size)
• the compiler must be portable (i.e., modular, supporting separate compilation)
• it must print good diagnostics and error messages
• the generated code must work well with existing debuggers
Challenges

• Building a compiler requires knowledge of
  – programming languages (parameter passing, variable scoping, memory allocation, etc)
  – theory (automata, context-free languages, etc)
  – algorithms and data structures (hash tables, graph algorithms, dynamic programming, etc)
  – computer architecture (assembly programming)
  – software engineering
Addressing Portability

- Suppose you want to write compilers from \( m \) source languages to \( n \) computer platforms. A naïve solution requires \( n \times m \) programs:

\[
\begin{align*}
\text{C++} & \rightarrow \text{MIPS} \\
\text{Java} & \rightarrow \text{SPARC} \\
\text{FORTRAN} & \rightarrow \text{Pentium} \\
\text{FORTRAN} & \rightarrow \text{PowerPC}
\end{align*}
\]

- but we can do it with \( n + m \) programs:

\[
\begin{align*}
\text{C++} & \rightarrow \text{FE} \rightarrow \text{IR} \rightarrow \text{BE} \rightarrow \text{MIPS} \\
\text{Java} & \rightarrow \text{FE} \rightarrow \text{IR} \rightarrow \text{BE} \rightarrow \text{SPARC} \\
\text{FORTRAN} & \rightarrow \text{FE} \rightarrow \text{IR} \rightarrow \text{BE} \rightarrow \text{Pentium} \\
\text{FORTRAN} & \rightarrow \text{FE} \rightarrow \text{IR} \rightarrow \text{BE} \rightarrow \text{PowerPC}
\end{align*}
\]

- IR: Intermediate Representation
- FE: Front-End
- BE: Back-End
Phases

• A typical real-world compiler usually has multiple phases
• The front-end consists of the following phases:
  – **scanning**: a scanner groups input characters into tokens
  – **parsing**: a parser recognizes sequences of tokens according to some grammar and generates *Abstract Syntax Trees* (ASTs)
  – **semantic analysis**: performs type checking and translates ASTs into IRs
  – **optimization**: optimizes IRs
• The back-end consists of the following phases:
  – **instruction selection**: maps IRs into assembly code
  – **code optimization**: optimizes the assembly code using control-flow and data-flow analyses, register allocation, etc
  – **code emission**: generates machine code from assembly code