Course Overview

You should still take notes, read textbook.

Lecture notes

1-week late policy, no incompletes.

Policies

advanced compiler construction, optimizations

Scan/parse using TeX and CUP

Scanning construction (RTE to minimal DFAs)

Programming Projects

20% Programming Projects
30% Final Exam
20% Mid-Term Exam

Basics for Grades:

Prerequisites: CISC 330

Another Apple Textbook is Modern Compiler Implementation by Andrew Appel


Class URL: csd.cisc430.washington.edu

Hours:

TA:

Office: A.V. Williams 4137
email: nan Wang

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Important facts:

Course Overview

Basis for Grades:

Grading:

20% Programming Projects
30% Final Exam
20% Mid-Term Exam

Prerequisites: CISC 330

Another Apple Textbook is Modern Compiler Implementation by Andrew Appel


Class URL: csd.cisc430.washington.edu

Hours:

TA:

Office: A.V. Williams 4137
email: nan Wang

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Important facts:

Course Overview

Basis for Grades:

Grading:

20% Programming Projects
30% Final Exam
20% Mid-Term Exam

Prerequisites: CISC 330

Another Apple Textbook is Modern Compiler Implementation by Andrew Appel


Class URL: csd.cisc430.washington.edu

Hours:

TA:

Office: A.V. Williams 4137
email: nan Wang

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Important facts:

Course Overview

Basis for Grades:

Grading:

20% Programming Projects
30% Final Exam
20% Mid-Term Exam

Prerequisites: CISC 330

Another Apple Textbook is Modern Compiler Implementation by Andrew Appel


Class URL: csd.cisc430.washington.edu

Hours:

TA:

Office: A.V. Williams 4137
email: nan Wang

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Office: A.V. Williams 4137
email: chau-wen tseng@cs.washington.edu

Important facts:
What is a compiler?

A compiler translates an executable program in one language into an executable program in another language.

Implications:
- Need format for object (or assembly) code
- Manage storage of all variables and code
- Generate correct code
- Recognize legal (and illegal) programs

Errors

Compiler

Source code

Machine code

Abstract View

Compiler Overview

What is a compiler?

and software engineering skills. Hopefully you will also improve your programming.

- Software Libraries (Java class libraries)
- Program-generation tools (flex, cpp)
- Programming tools (compilers, debuggers)

Working on compilers, you will learn to use

Compilers are large, complex pieces of software. By

working on compilers, you will learn to use

compilers, you will learn to use

Compiler Overview

Abstract View

What is a compiler?
Many of front end construction can be automated:

- Front end is \( O(n) \) or \( O(n \log n) \)
- Back end is \( \text{NP-complete} \)

**Responsibilities:**
- Recognize legal procedure
- Report errors
- Produce \( \text{preliminary storage map} \)
- Produce \( \text{shape the code for the back end} \)
- \( \text{errors} \rightarrow \text{tokens} \rightarrow \text{scanner} \rightarrow \text{source code} \)

**Implications:**
- Multiple passes \( \leftrightarrow \) better code
- Allows multiple front ends
- Simpler front end
- Back end maps \( \text{into target machine} \)
- Front end maps \( \text{legal code into } \)
- Intermediate language \( (ll) \)
patterns.

We need a powerful notation to specify these patterns. There are three main types of patterns:

- **Keywords and Operators**: Specialized as literal patterns - do, end, operators, and operations.
- **Identifiers**: Alphabetical followed by alphanumeric.
- **Numbers**: Digits from 0-9, 0-9 digits, 0-9 digits, and 0-9 digits.
- **Comments**: White space, tabs, and blank.

Some parts are easy:

- **Syntax**

Some parts are much harder:

- **Tokens**

A scanner must recognize various parts of the language's syntax.

### Specifying Patterns

**Scanner**

**Parser**

**Errors**

**Source**

**Tokens**

**Code**
Scanners can be specified using regular expressions. Patterns can be specified using regular expressions.

Parser generators mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA's, then implementing the DFAs.

Scanners are constructed by translating RE's to DFA's. Then implementing the DFAs.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)

Parse generator mechanize much of the work.

Parser: Scanner tokens source code

tokens

errors

Scanner

Parser

DFA’s, then implementing the DFAs.

Scanners are constructed by translating RE’s to DFA’s.

Regular expressions can then be used to construct

DFA.

Deterministic finite automaton (DFA)
Grammar

Contex t-free synt a x is specified wi th a grammar /

\[ S = \{ d \} \]

\[ \text{S is the start symbol} \]

\[ \text{N is a set of non-terminal symbols} \]

\[ \text{T is a set of terminal symbols} \]

\[ \text{P is a set of productions or rewrite rules} \]

Formally, a grammar \( G \) is defined as:

\[ G = (S, \text{T}, \text{N}, \text{P}) \]

The notation is called Backus-Naur form (BNF).

This grammar defines the set of noises that a sheep

\[ \text{makes under normal circumstances.} \]

\[ <\text{sheep noise}> \]

\[ \text{baa} \]

\[ \text{Sheep noise} :: = \text{baa} \]

Context-free syntax can be put to better use.

Substitution

Grammar
Derivations

Given a grammar, valid sentences can be derived by repeated substitution.

For many grammars, we can generate Table-driven parsers which use a DFA and stack.

To recognize a valid sentence in some C-family language, we reverse this process and build up a parse result.

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Resul</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x - y</td>
</tr>
<tr>
<td>3</td>
<td>2 + x</td>
</tr>
<tr>
<td>4</td>
<td>2 - y</td>
</tr>
<tr>
<td>5</td>
<td>x - y</td>
</tr>
<tr>
<td>6</td>
<td>2 + x</td>
</tr>
<tr>
<td>7</td>
<td>2 - y</td>
</tr>
<tr>
<td>8</td>
<td>2 + x</td>
</tr>
<tr>
<td>9</td>
<td>x - y</td>
</tr>
</tbody>
</table>

Parser Construction

- Generate intermediate code
- Collect information for type checking
- Reverse to reduce
- Parsers can also perform actions during each reduction
Optimizing compilers

Modern optimizers are usually built as a set of passes. 

- Must preserve values
- Goal is to reduce runtime
- Analyzes and changes

Code Improvement

- Translate ir into target machine code
- Choose instructions for each ir operation
- Ensure compatibility with system interfaces
- Decide what to keep in registers at each point
- Optimize ir to reduce runtime
- Analyzes and changes

Responsibilities

- Translate ir into target machine code
- Choose instructions for each ir operation
- Ensure compatibility with system interfaces
- Decide what to keep in registers at each point
- Optimize ir to reduce runtime
- Analyzes and changes

Code Improvement

- Must preserve values
- Goal is to reduce runtime
- Analyzes and changes

Errors

Optimizing compilers

Back end