**Syllabus**

<https://github.com/Software-Analysis-and-Transformation/Syllabus#csc-81020-software-analysis-and-transformation>

Piazza

**Teacher**

Software Engineer at Apple, Inc. (DRM) itunes, iBook, and the App store. Automated Software evolution, source code recommendation systems for correctly and efficiently evolving large and complex software.

**Research Areas**

Code Structure, Code Transformation, Project/Software Management, Performance Analysis, Performance Improvements, Error Analysis, Error Correction, Security Analysis, Security Correction, Automation Features (Adding, Removing, Replacing, Creating, Detecting). Testing (Consistency, Behavior, Maintainability).

Control Flow analysis - CFG

Data Flow Analysis –

Abstraction Challenges (correctness, precision, effiencent computation)  
Precondition Checking – Code transformation enable by analysis in refactoring

**Software Transformation**

performance optimization, decompiling, obfuscating, for large software problems and task.

Traditional – Source(.java/.cpp) => Compiler => Output(.class/.out/.exe)

Refactoring – Source1 => Refactorer.exe => Source2.

Refactoring - process of reorganizing software for improved design while perserving orginal functionality.

What are refactoring improvements – speed, maintainablitiy, security, behavior maintance, same input/output

**Software Analysis**  
program correctness, understanding, evolution, testing large software projects and task.

Runtime Analysis - Program.exe => Analysis Program => Analysis Output

Precompilation Anaysis - Program.cpp => Analysis Program => Analysis Output

Inspection Number of Variable? Number of Functions? Resuable Architecture? Test converage, any bugs, security vulnerabilities?

Compile Time Anaysis – the ide contains code smells (defects), code refactoring

Testing tools – regression testing

Performance tools

Debugging

Static Analysis

Dyamic Anaysis

**AST**

represents the abstract syntactic structure of source code of a PL built by parsers

Datastructure

Result of Syntax Analysis

Intermediate representation of Program (IR)

Properites - edited, enhanced, precise, barebones

Stores position, meta data for each element, to print useful information

Documentation / Inherited nature

CFG - reduces ambiguity

Operator overload - final function are determined based on context. For example + operator.

Smaller in height than the parse tree.

Denote Variable types and declaration location

Order of executable statements

Identifiers and assigned values

Source code should be similar to original in appearence and execution, recompilation

**Parse Tree**

Syntax Analysis stage

Datastruture

(Program) Source code Translation

(Program) Source code Analysis

Contextual Analysis

Sound Software Development Principals

Enhance modularity

Ease development

Sound Node class Hiercharchy in modification of the AST

Compiler Determines correctness by traversing the AST (Visitor Pattern)

**Semantic Anaylsis**

AST used extensively

Correct uses of elements and verification of the program

Symbol table generation

AST serves as the base for code generation (IR) or (IL)

AST can be engineered in a language agnostic fashion

**(PDE) Plugin Development Environment**OSGi toolingComponent tooling[UI](http://127.0.0.1:63942/help/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm#ui) - A rich set of models, tools and editors to develop plug-ins and OSGi bundles[API Tools](http://127.0.0.1:63942/help/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm#api) - Tooling to assist API documentation and maintenance[Build](http://127.0.0.1:63942/help/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm#build) - Ant based tools and scripts to automate build processes<https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fui%2Fpart%2FMultiPageEditorPart.html>  
New project > Plugin project > Try Each of these templates > Open plugin.xml > launch plugin.  
UI Component – responsible for all wizard and editors  
Build Component – responsible for building and packaging your component  
org.eclipse.pde.ui.pluginContent – responsible for creating template wizards  
Templates – HelloWorld, HelloWorldCommand,   
Plugin.xml – bring up the configuration parameters of the plugin  
Exporting Plugin for Deployment – File > Export > Deployable plugin / Fragment   
Installing Plugin for Development - Help -> Install New Software -> Add -> Archive  
AbstractHandler -   
IWorkbeanchWindowActionDelete -   
AbstactUIPlugin -   
MultipageEditorPart -   
IResourceChangeListener –

**Refactoring**Automated Refactoring – programming technigque for improving code quality for maintainability and extensibility while preserving existing behavior. Transformation / manipulation of the programExamples – variable renaming, block changesLTK – Language Toolkit can be used to handle automated refactoring.LTK plugins (PDE) - org.eclipse.ltk.core.refactoring, org.eclipse.ltk.ui.refactoring, AST search for language elements. Find/Replace analyse toolsIntuition - <http://www.eclipse.org/articles/Article-LTK/ltk.html>

**Compiler Research and Understanding**

**Compiler Front End / Program represenentation**  
scanner, parser, semantic analyzer, code generator  
scanner – lexical analyzer or string (foo.java) => tokenizer => tokens  
syntax analyzer – creates a tree from the tokens (AST) Transformation process  
semantic analyzer – type checking anc conversion, compiletieme analysis is done

**code generator (IM) – 3 address code / Program representation**address code that the AST maps and optimize to.   
close to target platform language  
type checking and inferencing  
code generation and optimization – generates compile time temporary variable  
Example a = b + c + d <= 3 address code generation => t = b + c; a = t + d;  
Eclipse plugins don’t have access to lower-level IR for analysis but AST can provide code smells to programmer but analysis and transformation can be done easier at lower levels

**3 address Instruction code – assignment and arithmetic**instruction contains at most 3 addresses  
x = y op z – binary op ⬄ t1 = y op z, x = t1  
x = op y – unary op <=> t1 = op y, x = t1  
x = y - assignment  
arrays and flow control instrunctions  
**x, y, z are variable, temporizes or constants  
symbol table  
Left pointer – pointer to symbol table left entry  
Right pointer – pointer to symbol table for right entrys**Example X = y + z + w ⬄t1 = y + z; t2 = t1 + w; x = t2Example X = y + -z <=> t1 = -z; t2 = y + t1; x = t2Example a = b + c + d <= 3 address code generation => t = b + c; a = t + d;

**3 address Instruction code - Flow Control Instrunction  
goto L for flow control: unconditional jump to 3 address instrunction with Label L  
labels are symbolic names  
if(x relop y) goto L: x and y are variables tem[. Constants  
relop = <,<=,==, etc**

**Example if(x < 100 || x > 200 && x != y) x = 0;  
Cleaner translation  
if(x<100) goto L2;  
if(x<=200) goto L1;  
if(x == y) goto L1  
L2: x = 0  
L1: …;**

**Less Clean  
if(x<100) goto L2;  
goto L3;  
L3: if(x<200) goto L4;  
goto L3;  
L4: if(x != y) goto L2;  
goto L1;  
L2: x = 0;  
L1: ..;**

**Compiler Middle Part (Compiler Research Focus)**Compile-time analysis of IM code  
Control Flow graphs   
Single assignment form  
Def-use graph  
Control flow analysis – what instruction executed at runtime (CFG)  
Data flow analysis – what are the effects of these instrunction sequences on the state of the program  
Side-effect analysis  
polyhedral analysis

**Machine-independent optimization id IM (Compiler Research Focus)  
Copy propagation  
dead code elimination  
code motion  
constant propagation  
redundancy elimination  
parallelization  
data locality optimization**

**Other Stuff**

**Rationale**

mobile computing, cybersecurity, and big data, the need for software engineering tools to assist developers in cultivating and managing the complexity of large software systems has never been more evident, giving way to such tools as   
[GitHub](http://github.com/),   
[Pivotal](http://pivotal.io/), and   
[SonarQube](http://www.sonarqube.org/).   
  
Software analysis and transformation has a variety of uses, including

performance optimization (in compilers),   
decompiling,   
obfuscating,   
verifying program correctness,   
domain-specific programming languages,   
and software understanding,   
evolution,   
debugging, and   
testing in software engineering tools.

The material will be useful for students working in   
compilers,   
developer tools,   
systems,   
software engineering, and   
programming languages.

Ofuscation - changing the code (variable names)

* intermediate program representations, including Abstract Syntax Trees (ASTs)
* tree traversal and manipulation algorithms
* control- and data-flow analyses
* type inference
* object-orientation
* pointer/alias analysis
* side-effect analysis
* refactoring
* object-oriented software design patterns

*Compilers: Principles, Techniques and Tools* by Aho,

Aliasing (pointers) polymorphism

Program analysis

Program.src -> program -> Output(variable?, meta-data?, architure is resuable, test coverage, bugs)

Program.exe -> Program.exe (output questions)

Specification, security vulernerabilitys (could be a platform issue)

Static anaylsis => reduces to the Halting problem / rices theorom

Dynamic Analysis

JET just in time compiler some interpretors have this

Refactoring is a semantics preserving of the program (produces the same output)

Not invertable tranformation (lose information)

Research Topics

How you code

Large and complex projects

Lightweight anaylsis the JIT

AST - some compile time analysis

Middle Layer

IR - three-address code, control flow analysis, CFG(control flow gragh), data-flow anaylsis,

SSA - static single assignment form

Gcc - code optimization

Parallizing compilers, data locality

Three-Address Code

Writting Plugins Intileji, Ecylipes etc.

Topics -

AST

IR layer

tree traversal

manipulating algorithms

control flow

data flow

OOP

Type infererence

Pointer / Alias analysis

Refactoring

OO Design Patterns

Knowledge of enterprise-level software design patterns and their relation to refactoring

Problems and solutions

Meta level / Abstact Software Representation

OO type system and inference

Homeworks 25%

Projects 55%

Special Topics 10%

Attendence/Participation 10%