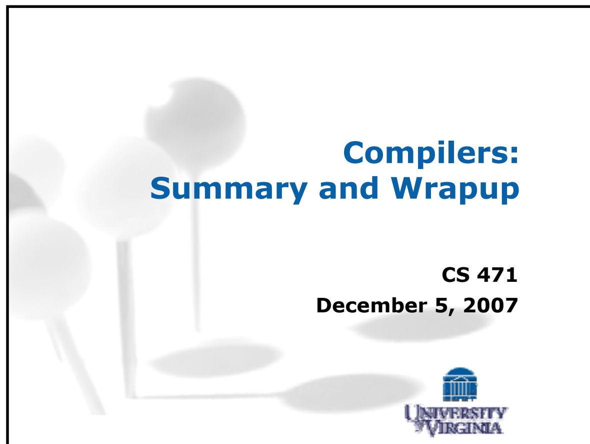
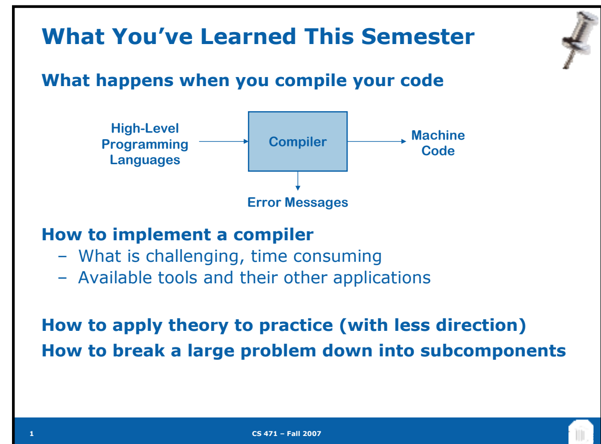




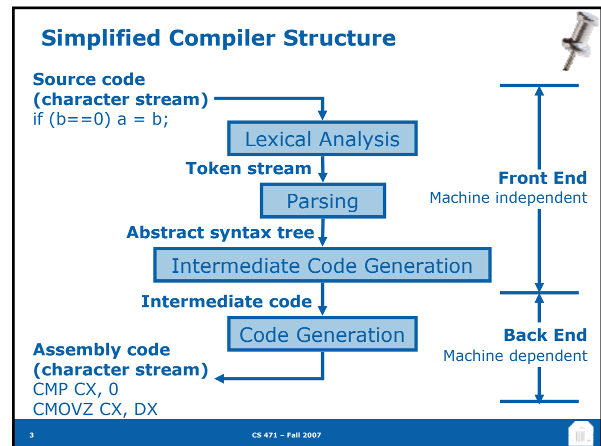
Goals of a Compiler

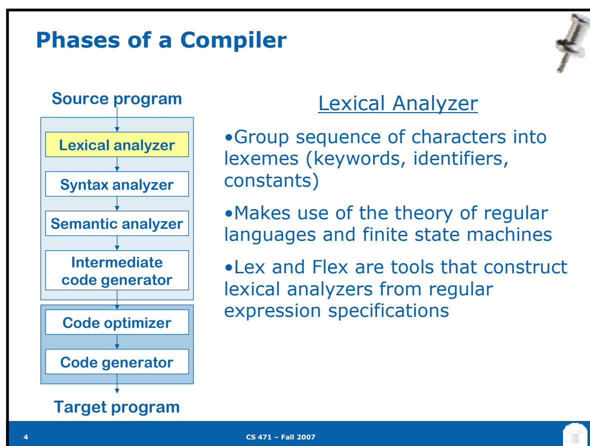
A compiler's job is to

- Lower the abstraction level
- Eliminate overhead from language abstractions
- Map source program onto hardware efficiently
  - Hide hardware weaknesses, utilize hardware strengths
- Equal the efficiency of a good assembly programmer

Optimizing compilers should *improve* the code

- Performance*
- Code size
- Security
- Reliability
- Power consumption





Regular expressions

Language – set of strings
String – finite sequence of symbols
Symbols – taken from a finite alphabet

Specify languages using regular expressions

Symbol	a	one instance of a
Epsilon	ε	empty string
Alternation	R | S	string from either L(R) or L(S)
Concatenation	R · S	string from L(R) followed by L(S)
Repetition	R*	0 or more strings from L(R)

Finite Automata

Automaton (DFA) can be represented as:

- A transition table

[^"]*

	"	non-"
0	1	error
1	2	1
2	error	error

- A graph

Language

Each string is accepted or rejected

1. **Starting in the start state**
2. **Automaton follows one edge for every character (edge must match character)**
3. **After n-transitions for an n-character string, if final state then accept**

Language: set of strings that the FSA accepts

Lex: A Lexical Analyzer Generator

- **Lex produces a C program from a lexical specification**
- **http://www.epaperpress.com/lexandyacc/index.html**

```
%%
DIGITS  [0-9]+
ALPHA  [A-Za-z]
CHARACTER {ALPHA}|_
IDENTIFIER  {ALPHA}({CHARACTER}|{DIGITS})*
%%
if                                  {return IF; }
{IDENTIFIER}                        {return ID; }
{DIGITS}                            {return NUM; }
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+) {return REAL; }
.                                   {error(); }
```


Phases of a Compiler

Parser

- Convert a linear structure – sequence of tokens – to a hierarchical tree-like structure – an AST
- The parser imposes the syntax rules of the language
- Work should be linear in the size of the input (else unusable) → type consistency cannot be checked in this phase
- Deterministic context free languages and pushdown automata for the basis
- Bison and yacc allow a user to construct parsers from CFG specifications

Context-Free Grammar Terminology

$S \rightarrow (S)\ S$

$S \rightarrow \varepsilon$

- **Terminals**
  - Token or ε
- **Non-terminals**
  - Syntactic variables
- **Start symbol**
  - A special nonterminal is designated (*S*)
- **Productions**
  - Specify how non-terminals may be expanded to form strings
  - LHS: single non-terminal, RHS: string of terminals or non-terminals
- Vertical bar is shorthand for multiple productions

Shift-Reduce Parsing

Bottom-up parsing uses two kinds of actions: Shift and Reduce

Shift: **Move I one place to the right**

- Shifts a terminal to the left string

E + (I int) ⇒ E + (int I)

Reduce: **Apply an inverse production at the right end of the left string**

- If E → E + (E) is a production, then

E + (E + (E) I) ⇒ E +(E I)

Shift-Reduce Parsing Table

terminal symbols | non-terminal symbols
state | next actions | next state on reduction

Action table

1. shift and goto state n
2. reduce using $X \rightarrow \gamma$
   - pop symbols γ off stack
   - using state label of top (end) of stack, look up X in **goto table** and goto that state

- DFA + stack = push-down automaton (PDA)

Parsing Table

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

Yacc / Bison

- **Yet Another Compiler Compiler**
- **Automatically constructs an LALR(1) parsing table from a set of grammar rules**
- **Yacc/Bison specification:**

file.y

```
parser declarations
%%
grammar rules
%%
auxiliary code
```

→ bison –vd file.y -or- yacc –vd file.y → y.tab.c, y.tab.h, y.output

Phases of a Compiler

Source program → Lexical analyzer → Syntax analyzer → Semantic analyzer → Intermediate code generator → Code optimizer → Code generator → Target program

Semantic Analysis

- Calculates the program's "meaning"
- Rules of the language are checked (variable declaration, type checking)
- Type checking also needed for code generation (code gen for a + b depends on the type of a and b)

Typical Semantic Errors

Traverse the AST created by the parser to find:

- **Multiple declarations:** a variable should be declared (in the same scope) at most once
- **Undeclared variable:** a variable should not be used before being declared
- **Type mismatch:** type of the left-hand side of an assignment should match the type of the right-hand side
- **Wrong arguments:** methods should be called with the right number and types of arguments

Tiger Symbol Tables

Two namespaces … two symbol tables

- **Namespace for types**
- **Namespace for vars and functions**
- **Called tenv and venv, respectively**

Phases of a Compiler

Source program
- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Intermediate code generator
- Code optimizer
- Code generator

Target program

Intermediate Code Generation

- Makes it easy to port compiler to other architectures (e.g. Pentium to MIPS)
- Can also be the basis for interpreters (such as in Java)
- Enables optimizations that are not machine specific

Intermediate Code

- Abstract machine code – (**I**ntermediate **R**epresentation)
- Allows machine-independent code generation, optimization

AST → IR (optimize) → x86, PowerPC, Alpha

Phases of a Compiler

Source program
- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Intermediate code generator
- Code optimizer
- Code generator

Target program

Intermediate Code Optimization

- Constant propagation, dead code elimination, common sub-expression elimination, strength reduction, etc.
- Based on dataflow analysis – properties that are independent of execution paths

Analysis and Transformation

Most optimizations require some global understanding of program flow
- Moving, removing, rearranging instructions

Achieve understanding by discovering the control flow of the procedure
- What blocks follow/are reachable from other blocks
- Where loops exist (focus optimization efforts)
- We call this **Control-Flow Analysis**

Also, find and connect definitions and uses of variables
- We call this **Data-Flow Analysis**

All approaches to control-flow analysis require
- Identification of basic blocks
- Construction of a flow graph of the procedure

When to Apply Optimization

HIR	AST	Inlining
		Specialization
	IR	Constant folding
		Constant propagation
		Value numbering
MIR	Canonical IR	Dead code elimination
		Loop-invariant code motion
		Common sub-expression elimination
	Abstract Assembly	Strength reduction
		Branch prediction/optimization
		Register allocation
LIR	Assembly	Loop unrolling
		Cache optimization

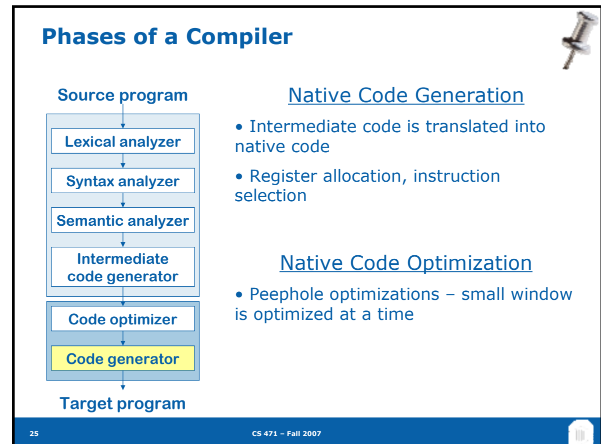
Scope of Optimization

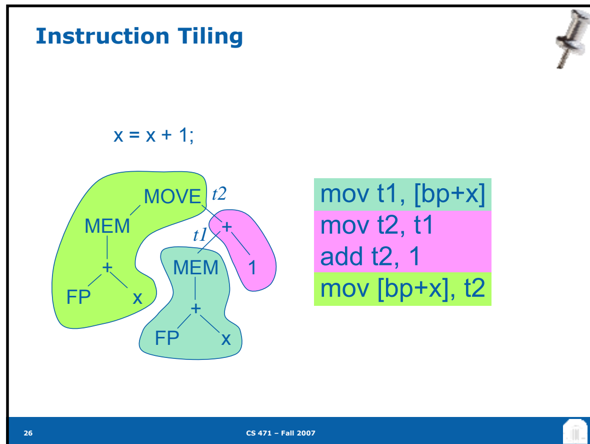
Local **(or single block)**
- Confined to straight-line code
- Simplest to analyze

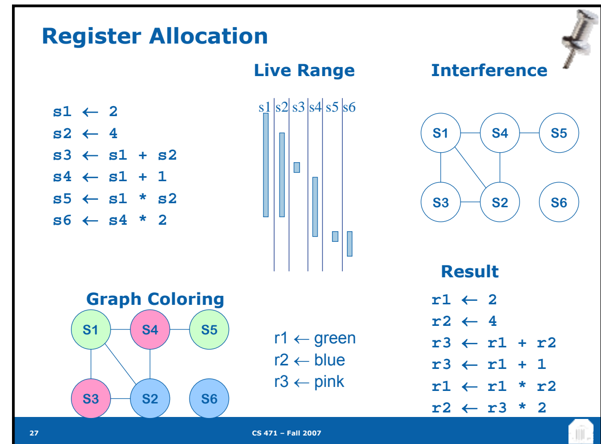
Intraprocedural **(or global)**
- Consider the whole procedure

Interprocedural **(or whole program)**
- Consider the whole program

Flag	Description
-O0	Barely any transformations, just code generation. Target code can be debugged with no information loss.
-O1	Some transformations that preserve execution ordering. Debuggability of the generated code is hardly affected. User variables should not disappear and function inlining is not done.
-O2	More aggressive transformations. May affect execution ordering and usually provide faster code. Debuggability may be somewhat compromised by disappearing user variables and function bodies.
-O3	Very aggressive transformations that may or may not provide better code. Some semantics may be modified. Execution order completely distorted. Debuggability is seriously compromised.
-Os	Optimize for size. Enables transformations that reduce generated code size. May sometimes improve application performance because there is less code to execute. May lead to reduced memory footprints which may produce fewer page faults.
-O4	There is no -O4. Anything above -O3 is treated as -O3.







Alternatives to the Traditional Model

Static Compilation

All work is done "ahead-of-time"

Just-in-Time Compilation

Postpone some compilation tasks

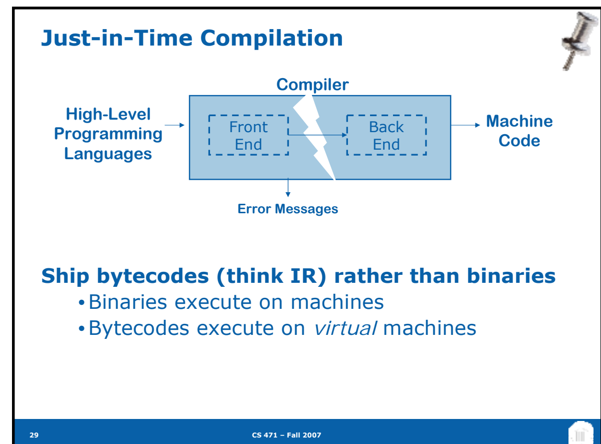
Multiversioning and Dynamic Feedback

Include multiple options in binary

Dynamic Binary Optimization

Traditional compilation model

Executables can adapt



The Big Picture

We now know:
- All of the components of a compiler
- What needs to be done statically vs. dynamically
- The potential impact of language or architecture changes
- Why Java moved the "back-end" to run time

Compiler Wars: Deliverables and Rules

CS 471
December 10, 2007

1. Robustifying your Compiler

(For real this time)

Sample outputs are online

www.cs.virginia.edu/kim/courses/cs471/project/

Contains:
- test[0-9]+.tig.out
- merge.tig.out
- queens.tig.out

2. Generate Test Cases

a) A test case that may break others' lexical analyzers
    kh7fe.lex.tig

b) A test case that may break others' parsers
    kh7fe.parse.tig

c) A test case that may break others' typecheckers
    kh7fe.type.tig

d) A test case that may break others' irGen
    kh7fe.irgen.tig

e) A README file – describe expected output for each test file; what you expect to break in each case; why it's unique
    kh7fe.README.tests

3. Submit your Compilers

a) Your tiger.lex file (We will build your lexical analyzer)
    kh7fe.lex

b) Your grammar (with semantic actions)
    kh7fe.grm

c) Your typechecker (Input: file; Output: type errors)
    kh7fe.typecheck

d) Your IR generator (Input: file; Output: type errors and (if applicable) IR tree)
    kh7fe.irgen

e) README file describing all of the issues you corrected between PA7 and compiler wars
    kh7fe.README.compiler

f) A tar.gz file containing everything in your PA7 directory (assuming it has been updated since PA7)
    kh7fe.finalCompiler.tar.gz

Then What?

Submit everything
    BEFORE SUNDAY DEC 9 @ 9PM FIRM
    (enforced by toolkit)

Show up here on Monday Dec 10 @ 2PM for …

COMPILER WARS!

How Will It Work?

Bracketed progression (like the ACC tournament)
- Round 1: Lexical Analysis
- Round 2: Parsing & AST Generation
- Round 3: Type Checking
- Round 4: IR Generation

Each round: Full bracket progression to determine a winner (single elimination)

Each game: Compare test cases passed with your opponent
- **51 given test cases**
- **2 custom cases**

Tie breaker: Cleanest code (as determined by judges)

Prize structure

Prizes for:
- **Best lexical analyzer**
- **Best parser**
- **Best typechecker**
- **Best ir generator**
- **Best overall**
- **Most improved compiler**
- **Best test case**

Prizes:
- **Corresponding PA late day reprieve (1 day)**
- **Grab bag**

PA8 Grade

50%
submitting all PA8 materials *on time* and showing up for compiler wars

50%
completeness of the final product (PA2-PA7)

Extra Office Hours
Sunday 6-7 PM Olsson 209

Rest of Class Time Today…

1) Please fill out the official evaluations

2) Please send me anonymous feedback if you have detailed suggestions for doing things differently the next time I teach this course
   a) Topic emphasis, e.g. less time on X more time on Y
   b) Homework vs. projects vs. tests
   c) Lab timeslots?
   d) Group vs. individual projects?

Thanks for a Great Semester!