## CS 502: *Compilers: Principles and Practice*

Important facts:

*Name:* Dr. Tony Hosking
*Email:* `hosking@cs.purdue.edu`
*Office:* LWSN 3154L

Basis for grades:

| | |
|---|---|
| 20% | midterm |
| 30% | final |
| 40% | project |
| 10% | homeworks |





## Things to do

- make sure you have a working XINU account
- review Java development tools
- find `http://www.cs.purdue.edu/~hosking/502`



## Compilers

What is a compiler?

- a program that translates an *executable* program in one language into an *executable* program in another language
- we expect the program produced by the compiler to be better, in some way, than the original

What is an interpreter?

- a program that reads an *executable* program and produces the results of running that program
- usually, this involves executing the source program in some fashion

This course deals mainly with *compilers*

Many of the same issues arise in *interpreters*



## Motivation

Why study compiler construction?

Why build compilers?

Why attend class?

## Interest

Compiler construction is a microcosm of computer science

**artificial intelligence**
greedy algorithms, learning algorithms

**algorithms**
graph algorithms, union-find, dynamic programming

**theory**
DFAs for scanning, parser generators, lattice theory

**systems**
allocation and naming, locality, synchronization

**architecture**
pipeline management, hierarchy management, instruction set use

Inside a compiler, all these things come together



## Isn't it a solved problem?

*Machines are constantly changing*

Changes in architecture ⇒ changes in compilers

- new features pose new problems
- changing costs lead to different concerns
- old solutions need re-engineering

*Changes in compilers should prompt changes in architecture*

- New languages and features



## Intrinsic Merit

*Compiler construction is challenging and fun*

- interesting problems
- primary responsibility for performance (*blame*)
- new architectures ⇒ new challenges
- *real* results
- extremely complex interactions

*Compilers have an impact on how computers are used*

Some of the most interesting problems in computing



## Experience

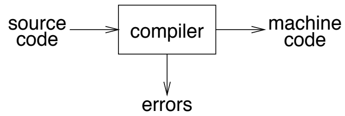
*You have used several compilers*

*What qualities are important in a compiler?*

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

*Each of these shapes your expectations about this course*

## Abstract view



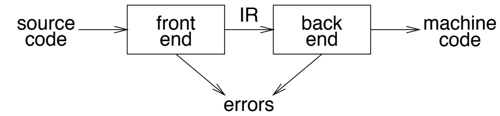

Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

*Big step up from assembler — higher level notations*



## Traditional two pass compiler




Implications:

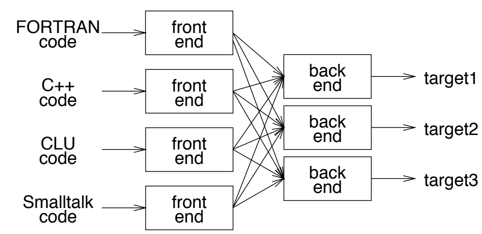
- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes ⇒ better code

*Front end is* $\mathbf{O}(n)$ *or* $\mathbf{O}(n \log n)$



*Back end is NP-complete*

## A fallacy

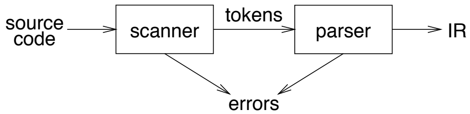



Can we build $n \times m$ compilers with $n + m$ components?

- must encode *all* the knowledge in each front end
- must represent *all* the features in one IR
- must handle *all* the features in each back end

*Limited success with low-level IRs*

## Front end



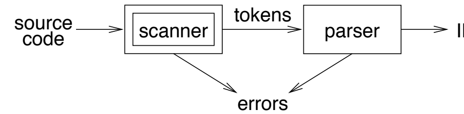

Responsibilities:

- recognize legal procedure
- report errors
- produce IR
- preliminary storage map
- shape the code for the back end

*Much of front end construction can be automated*



## Front end

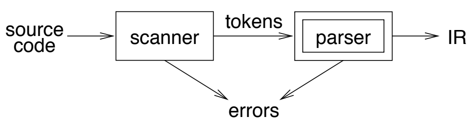



Scanner:

- maps characters into *tokens* – the basic unit of syntax
  `x = x + y;`
  becomes
  <id, `x`> = <id, `x`> + <id, `y`> `;`
- character string value for a *token* is a *lexeme*
- typical tokens: *number*, *id*, `+`, `-`, `*`, `/`, `do`, `end`
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed
  ⇒ use specialized recognizer (as opposed to `lex`)



## Front end




Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

*Parser generators mechanize much of the work*



## Front end

*Context-free syntax* is specified with a *grammar*

<sheep noise> ::= `baa`
| `baa` <sheep noise>

*The noises sheep make under normal circumstances*

This format is called *Backus-Naur form* (BNF)

Formally, a grammar $G = (S, N, T, P)$ where

$S$ is the *start symbol*
$N$ is a set of *non-terminal symbols*
$T$ is a set of *terminal symbols*
$P$ is a set of *productions* or *rewrite rules*
($P : N \to N \cup T$)

## Front end

*Context free syntax* can be put to better use

| | | | |
|---|---|---|---|
| 1 | <goal> | ::= | <expr> |
| 2 | <expr> | ::= | <expr> <op> <term> |
| 3 | | \| | <term> |
| 4 | <term> | ::= | `number` |
| 5 | | \| | `id` |
| 6 | <op> | ::= | `+` |
| 7 | | \| | `-` |

*Simple expressions with addition and subtraction over tokens* `id` *and* `number`

$S$ = <*goal*>
$T$ = `number`, `id`, `+`, `-`
$N$ = <*goal*>, <*expr*>, <*term*>, <*op*>
$P$ = 1, 2, 3, 4, 5, 6, 7



## Front end

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

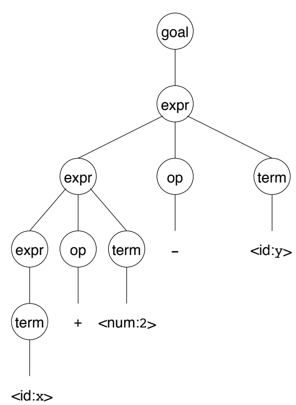
| Prod'n. | Result |
|---|---|
| | <goal> |
| 1 | <expr> |
| 2 | <expr> <op> <term> |
| 5 | <expr> <op> `y` |
| 7 | <expr> `- y` |
| 2 | <expr> <op> <term> `- y` |
| 4 | <expr> <op> `2 - y` |
| 6 | <expr> `+ 2 - y` |
| 3 | <term> `+ 2 - y` |
| 5 | `x + 2 - y` |

To recognize a valid sentence in some CFG, we reverse this process and build up a *parse*



## Front end

A parse can be represented by a *parse*, or *syntax*, tree

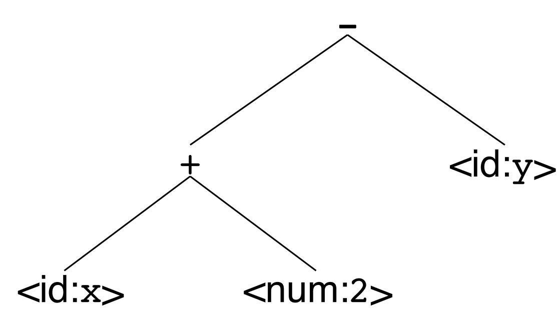



Obviously, this contains a lot of unnecessary information



## Front end

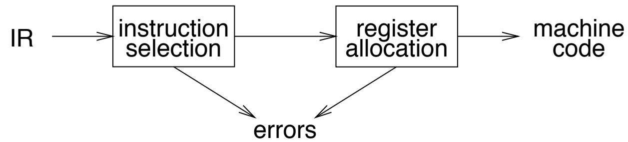
So, compilers often use an *abstract syntax tree*




This is much more concise

Abstract syntax trees (ASTs) are often used as an IR between front end and back end

## Back end



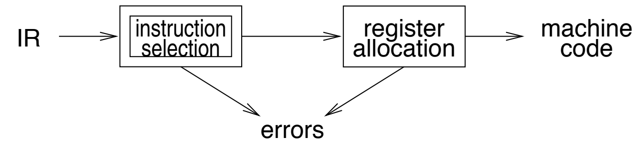

Responsibilities

- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

*Automation has been less successful here*



## Back end

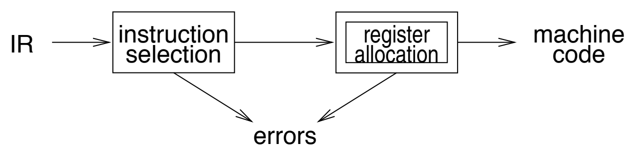



Instruction selection:

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - *ad hoc* techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming



## Back end



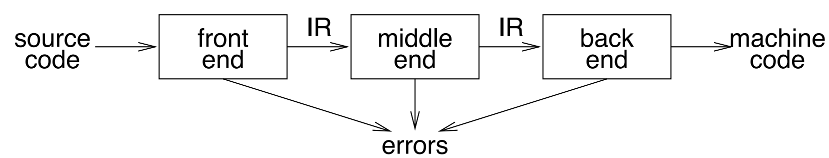

Register Allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for 1 or $k$ registers

*Modern allocators often use an analogy to graph coloring*



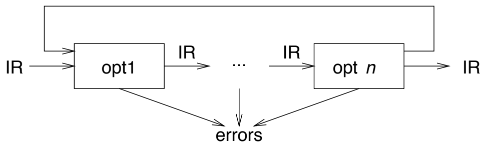
## Traditional three pass compiler




Code Improvement

- analyzes and changes IR
- goal is to reduce runtime
- must preserve values

## Optimizer (middle end)



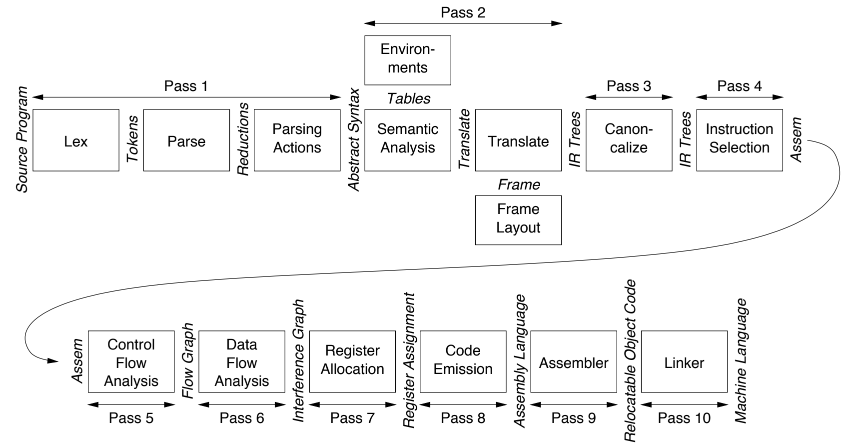

*Modern optimizers are usually built as a set of passes*

Typical passes

- constant propagation and folding
- code motion
- reduction of operator strength
- common subexpression elimination
- redundant store elimination
- dead code elimination



## The MiniJava compiler






## The MiniJava compiler phases

| | |
|---|---|
| Lex | Break source file into individual words, or *tokens* |
| Parse | Analyse the phrase structure of program |
| Parsing Actions | Build a piece of *abstract syntax tree* for each phrase |
| Semantic Analysis | Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase |
| Frame Layout | Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way |
| Translate | Produce *intermediate representation trees* (IR trees), a notation that is not tied to any particular source language or target machine |
| Canonicalize | Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases |
| Instruction Selection | Group IR-tree nodes into clumps that correspond to actions of target-machine instructions |
| Control Flow Analysis | Analyse sequence of instructions into *control flow graph* showing all possible flows of control program might follow when it runs |
| Data Flow Analysis | Gather information about flow of data through variables of program; e.g., *liveness analysis* calculates places where each variable holds a still-needed (*live*) value |
| Register Allocation | Choose registers for variables and temporary values; variables not simultaneously live can share same register |
| Code Emission | Replace temporary names in each machine instruction with registers |