What is a compiler?

Compilation in the context of language processing

Overview of a compiler

Front-end issues (analysis):

- **lexical analysis** (scanning): characters → tokens
- **syntax analysis** (parsing): tokens → abstract syntax trees
- **semantic analysis** (type checking): annotate ASTs

Intermediate representation generation:

abstract syntax trees (ASTs) → intermediate representation (IR)

Back-end issues (synthesis):

- **run-time storage representations**
- **optimizations**
- **target code generation:** IR→ assembly code

Example

Lexical analysis

"Scanning"

Read in characters, clump into **tokens**

- recognize **reserved words**
  - keywords, operators, punctuation
- form identifiers
- strip out whitespace in the process

Regular expressions used to specify the tokens

Enter identifiers into the **symbol table**

- data structure with entry for each identifier
- fields for identifier attributes

Specifying tokens: regular expressions

Example:

```
Ident    ::= Letter AlphaNum*
Integer  ::= Digit+
AlphaNum ::= Letter | Digit
Letter   ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Digit    ::= '0' | ... | '9'
```

Syntax analysis

"Parsing"

Read in tokens, turn into an **abstract syntax tree** based on syntactic structure

- operators are interior nodes
- operands are leaves

Checks that statements are correct

Filter out "noise" tokens

Syntactic structure specified by a **grammar** of the language

- resolve precedence

Add attributes to symbol table

Specifying syntax: context-free grammars

BNF is a popular notation for CFG's

Example:

```
Stmt      ::= AsgnStmt | IfStmt | ...
AsgnStmt  ::= LValue := Expr ;
LValue    ::= Id
IfStmt    ::= if Test then Stmt [else Stmt] ;
Test      ::= Expr = Expr | Expr < Expr | ...
Expr      ::= Term + Term | Term - Term | Term
Term      ::= Factor * Factor | ... | Factor
Factor    ::= - Factor | Id | Int | ( Expr )
```

Semantic analysis

"Typechecking"

Given AST:

- perform static consistency checks: **type checking**
- figure out what declaration each name refers to
- annotate the AST

Intermediate Representation

Given annotated AST & symbol table:

- produce **3 address code**

Allows:

- multiple languages for the same target machine
- multiple target machines for the same language

Storage layout

Given symbol tables,
determine how & where variables will be stored at run-time

What representation for each kind of data?

How much space does each variable require?

In what kind of memory should it be placed?

- static, global memory
- stack
- heap

Where in that kind of memory should it be placed?

- e.g. what stack offset

Optimizations

Machine independent optimizations, e.g.,

- constant folding
- constant propagation
- common subexpression elimination

Target code generation

Given an IR,
produce relocatable target code

Translate IR into **machine-specific** target instructions

- **instruction selection**
- **register allocation**
- **code scheduling**

Phases & Passes

Phase: a function of the compiler

Pass: read in & process program representation & write out

Multiple passes:

+ more flexibility in phase order
+ can do repeated optimizations
+ requires less memory
- slower