Phases of a Compiler
scanner

• Compiler starts by seeing only characters

```c
if (a < 4) {
    b := 5
}
```
• Compiler starts by seeing only text
• Not very easy to read!

```javascript
'if ('a' < 4') {
  'b' = '5'
}
```
• Compiler starts by seeing only text
• Not very easy to read!
• Scanner converts this into a series of tokens

```
'i' 'f' '(' 'a' '<' '4' ')' '{' '\n' 't' 'b' ': ' '=' '5' '\n' '}
```
• Compiler starts by seeing only text
  • Not very easy to read!
• Scanner converts this into a series of tokens
  • One item for each “word” in the program
• Compiler starts by seeing only text
  • Not very easy to read!
• Scanner converts this into a series of tokens
  • One item for each “word” in the program
• But we still do not know what the structure of the program is
• Converts string of tokens into a parse tree or an abstract syntax tree.
• Captures syntactic structure of code (i.e., “this is an if statement, with a then-block”)

![Diagram of a parse tree for the code snippet: if (ID(a) OP(<) LIT(4)) { ID(b) = LIT(5) }]

```plaintext
if (ID(a) OP(<) LIT(4)) {
  ID(b) = LIT(5)
}
```
• Converts string of tokens into a **parse tree** or an **abstract syntax tree**.
• Captures syntactic structure of code (i.e., “this is an if statement, with a then-block”)
• Think: diagramming a sentence

```
if-stmt
  <
  cond-expr
  rhs

stmt_list
  then-clause

assign_stmt
  lhs
  rhs

a < b
4 < 5
```
semantic actions

• Interpret the **semantics** of syntactic constructs
  • Note that up until now we have only been concerned with what the **syntax** of the code is
  • What’s the difference?
syntax vs semantics

- **Syntax**: “grammatical” structure of language
  - What symbols, in what order, are a legal part of the language?
  - What is a valid “sentence”?
  - But something that is syntactically correct may mean nothing!
    - “colorless green ideas sleep furiously”
- **Semantics**: meaning of language
  - What does a particular set of symbols, in a particular order, mean?
  - What does it mean to be an if statement?
  - “evaluate the conditional, if the conditional is true, execute the then clause, otherwise execute the else clause”
a note on semantics

• How do you define semantics?
  • Static semantics: properties of programs
    • All variables must have a type
    • Expressions must use consistent types
    • Can define using attribute grammars
  • Dynamic semantics: how does a program execute?
    • Documentation
    • Can define an operational or denotational semantics for a language
      • Well beyond the scope of this class!
  • For many languages, “the compiler is the specification”
semantic actions

• Actions taken by compiler based on the semantics of program statements
  • Building a *symbol table*
  • Generating *intermediate representations*
symbol tables

• A list of every declaration in a program
  • Variables, functions, types, etc.

• Keeps track of key information about a symbol
  • Variables: scope, type, location (for global variables)
  • Structure definitions: names of fields, types of fields, layout of structure
  • Functions: return type, argument types and names
  • …
intermediate representation

• Also called *IR*
• A (relatively) low level representation of the program
• But not machine-specific!
• One example: *three address code*

```
bge a, 4, done
mov 5, b
done: //done!
```

• Each instruction can take at most three operands (variables, literals, or labels)
  • Note: no registers!
• Transforms code to make it more efficient
• Different kinds, operating at different levels
  • High-level optimizations
    • Loop interchange, parallelization
    • Operates at level of AST, or even source code
  • Scalar optimizations
    • Dead code elimination, common sub-expression elimination
    • Operates on IR
  • Peephole optimizations
    • Strength reduction, constant folding
    • Operates on small sequences of instructions
 optimizer

• Transforms code to make it more efficient
• Different kinds, operating at different levels
  • High-level optimizations
    • Loop interchange, parallelization
    • Operates at level of AST, or even source code
  • Scalar optimizations
    • Dead code elimination, common sub-expression elimination
    • Operates on IR
  • Peephole optimizations
    • Strength reduction, constant folding
    • Operates on small sequences of instructions
bool collatz(unsigned int128 x) {
    while (true) {
        if (x <= 1)
            return true;
        if (x % 2)
            x = x >> 1;
        else
            x = 3 * x + 1;
    }
}

https://gcc.godbolt.org/z/Wrfeo18of
code generation

- Generate assembly from intermediate representation
- Select which instructions to use
- Schedule instructions
- Decide which registers to use

```
bge a, 4, done
mov 5, b
done: //done!
```
```
lw r1 a
li r2 4
bge r1 r2 done
li r3 5
sw r3 b
done:
```
code generation

- Generate assembly from intermediate representation
- Select which instructions to use
- Schedule instructions
- Decide which registers to use

bge a, 4, done
mov 5, b
done: //done!

li r1 4
lw r2 a
blt r1 r2 done
li r1 5
sw r1 b
done:
next: putting it all together

Or: How do these phases interact?