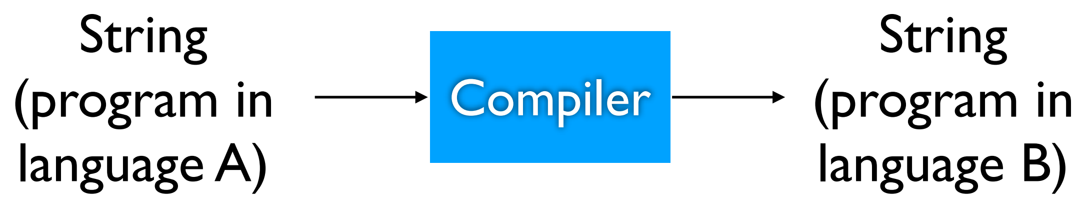
COMP 430: Tokenization

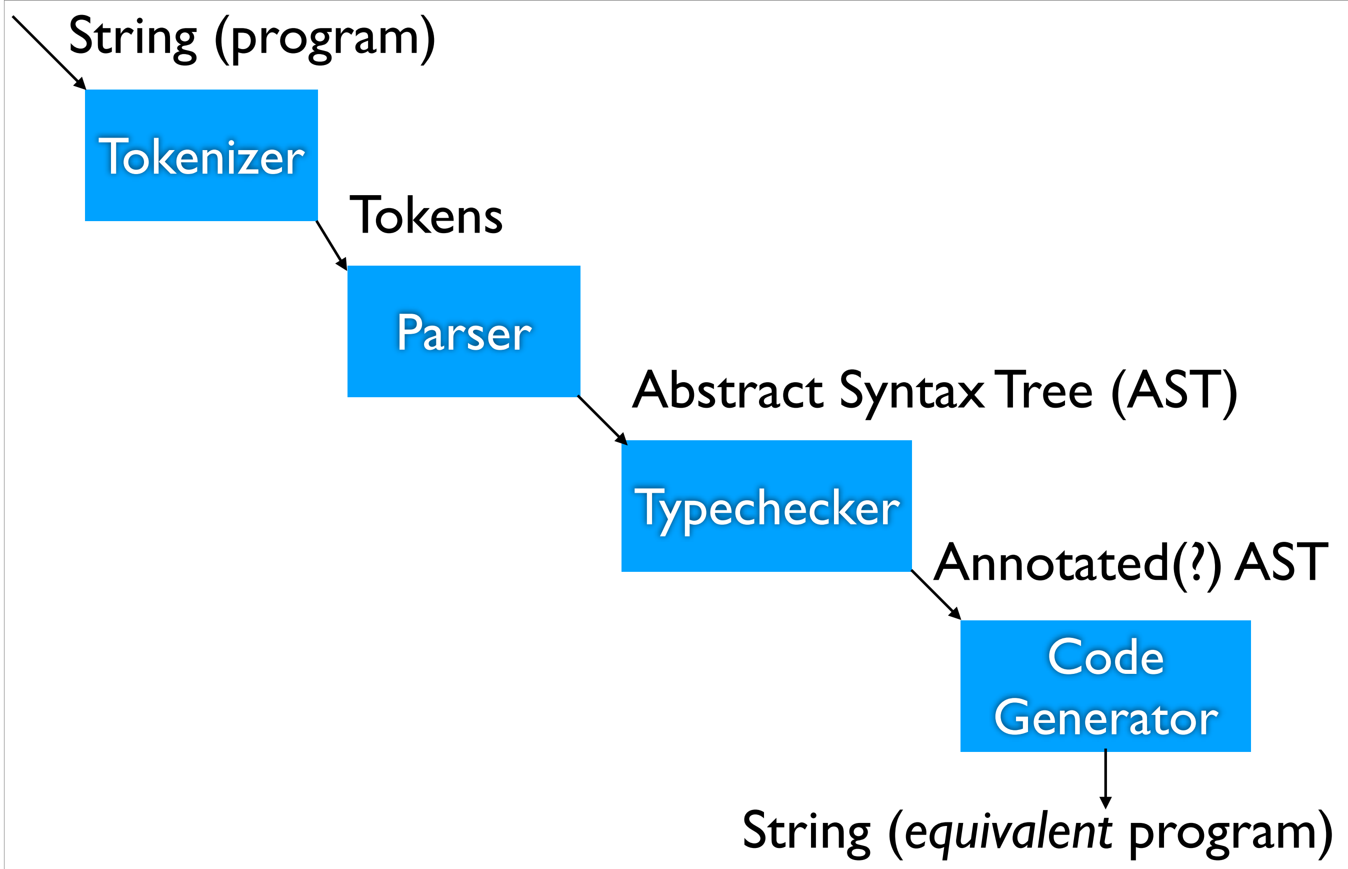
Kyle Dewey

Birds-eye View

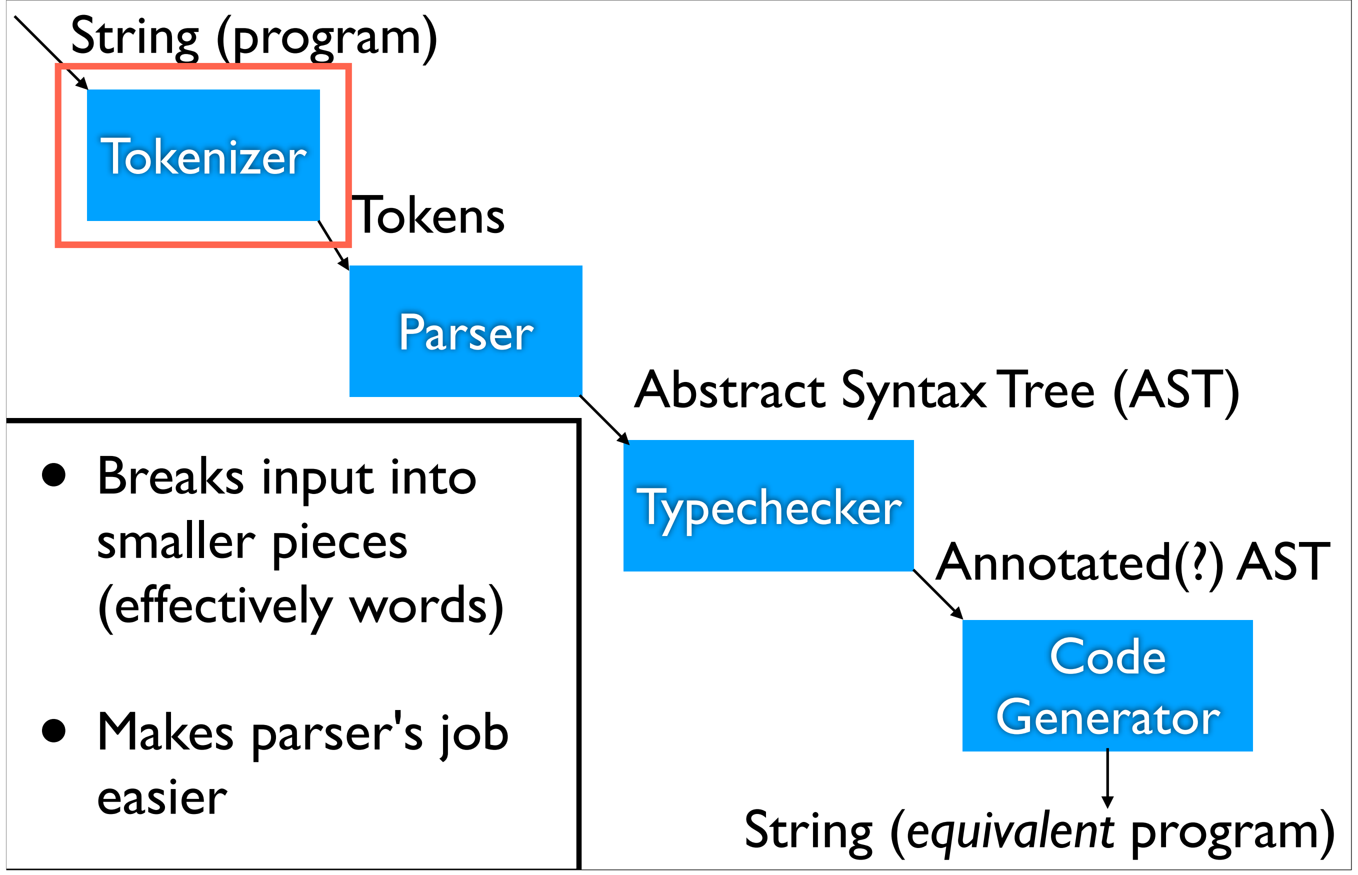
Compiler


String (program in language A)
Compiler
String (program in language B)

Compiler Architecture



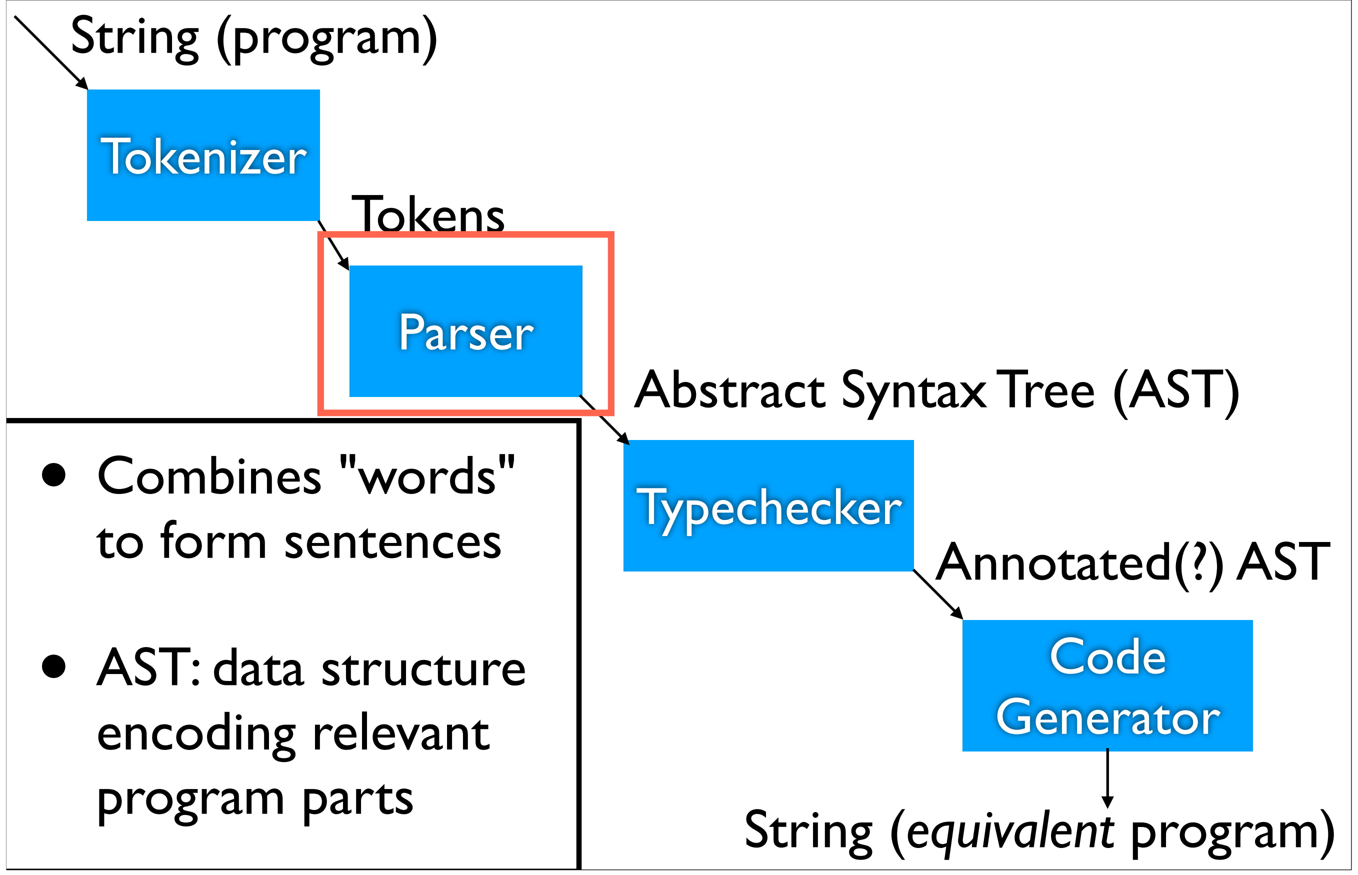
Compiler Architecture



- Breaks input into smaller pieces (effectively words)
- Makes parser's job easier

-For example, "return" has special meaning in most programs. It makes sense to look at "return" as one unit, instead of the separate characters 'r', 'e', 't', 'u', 'r', 'n'
-Errors here are usually treated as syntax errors; errors tend to be basic in nature

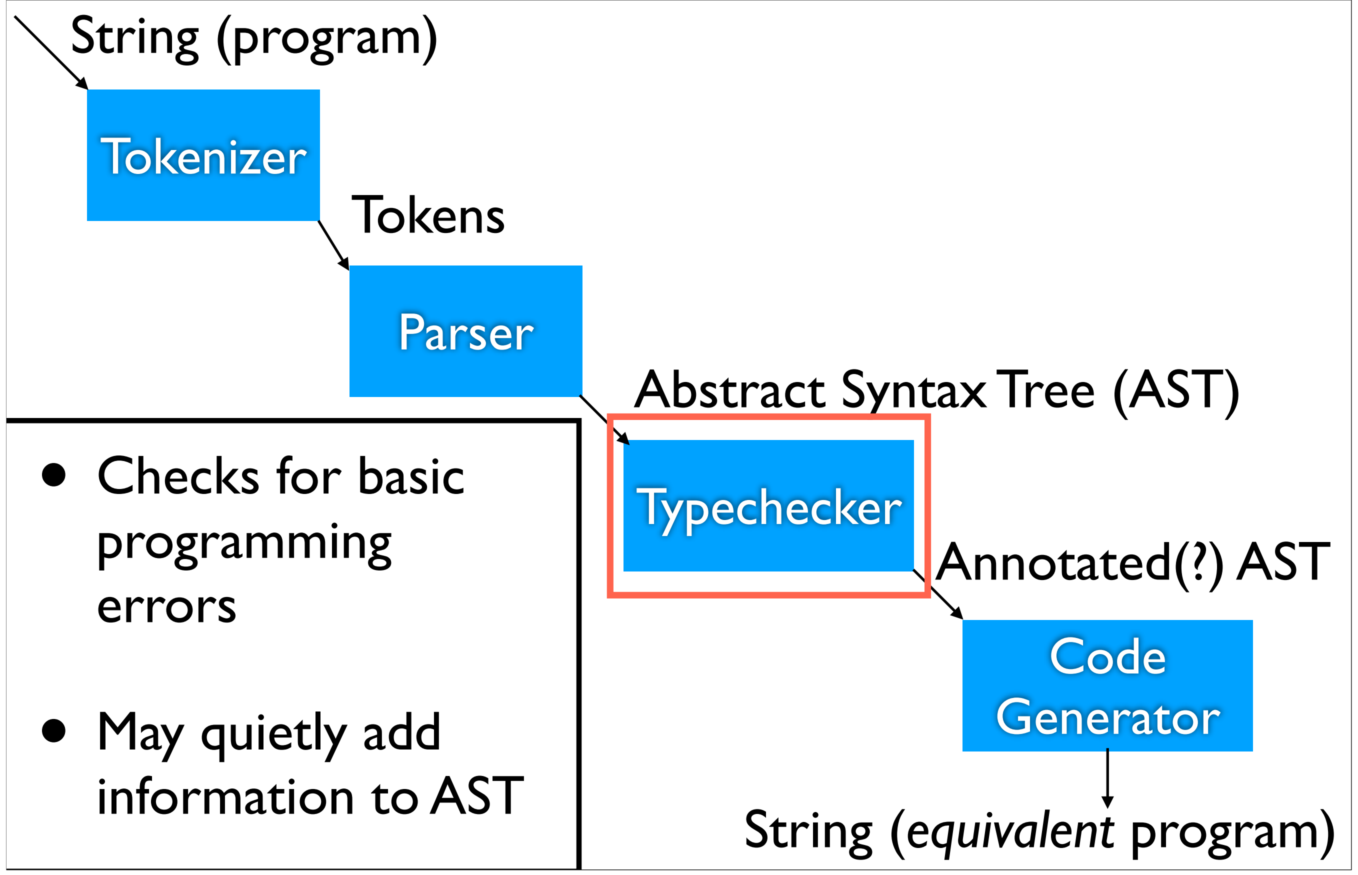
Compiler Architecture



- Combines "words" to form sentences
- AST: data structure encoding relevant program parts

-Many program parts are irrelevant (e.g., comments and whitespace)
-Also handles operator precedence and parentheses
-All syntax errors are from the tokenizer or parser; most are from the parser

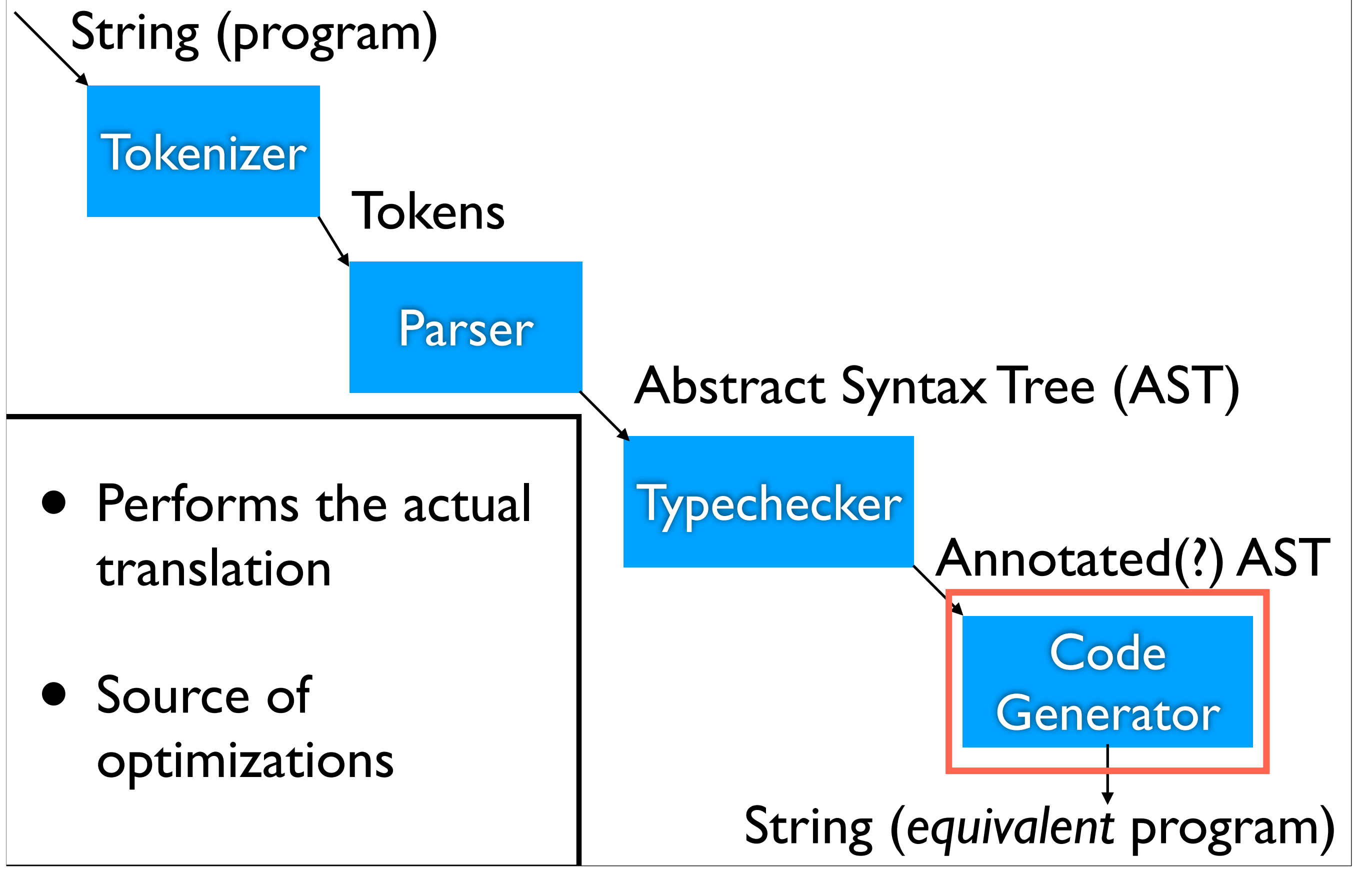
Compiler Architecture



- Checks for basic programming errors
- May quietly add information to AST

-Performs an analysis of the code (and is sometimes called semantic analysis)
-The origin of type errors
-Depending on the language, it may add information to the AST about the types of the values in play (e.g., x + y could refer to integer division or double division in Java; the typechecker will disambiguate between them)
-Can range from relatively simple to being the most complex component, depending on the language (and especially the kinds of errors we want to prevent)

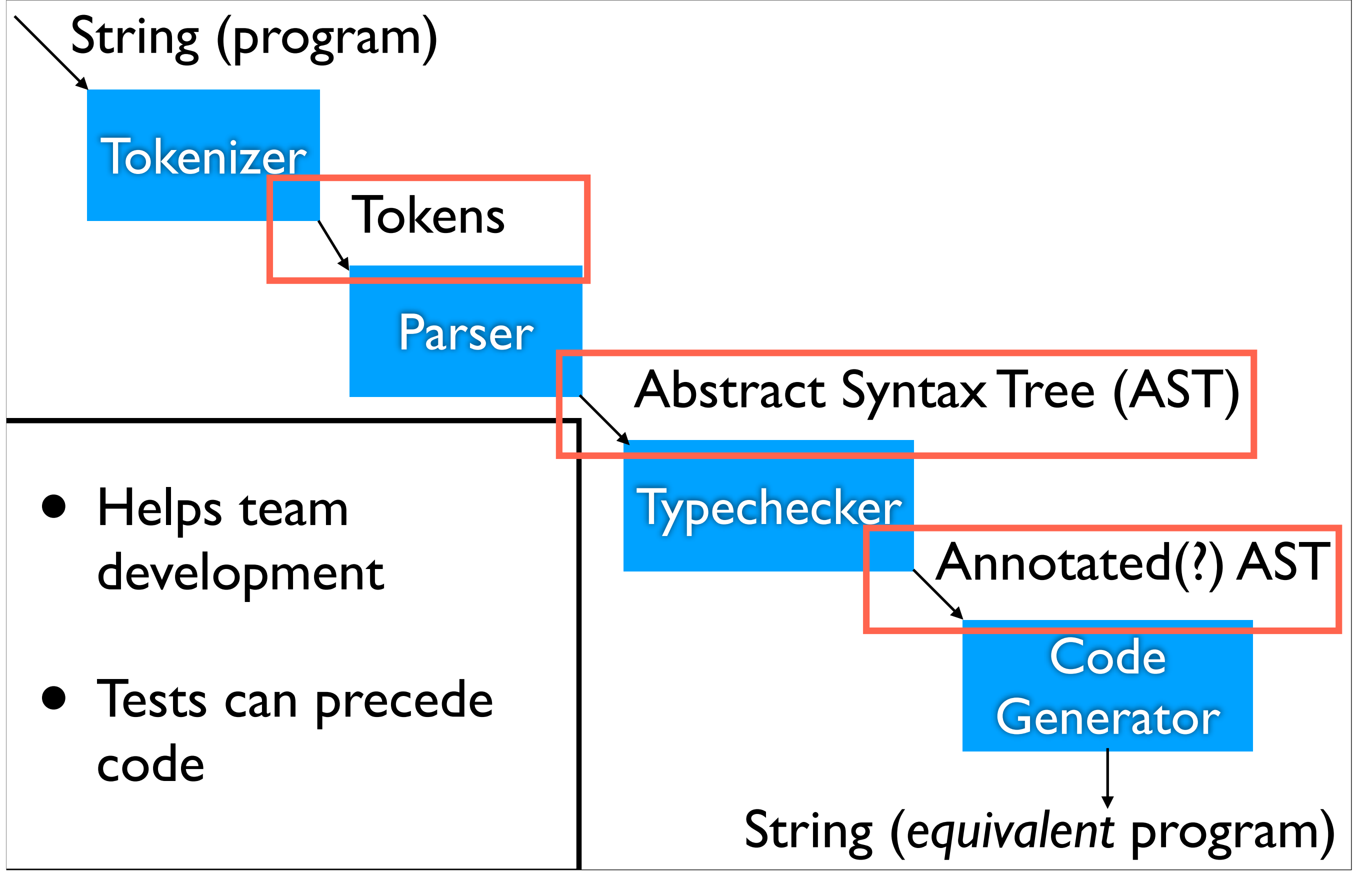
Compiler Architecture



- Performs the actual translation
- Source of optimizations

-In practice, usually the most complex component (especially for low-level targets)
-Often divided into a middle-end and back-end; the middle-end performs target-independent optimizations, whereas the back-end performs target-specific optimizations (the tokenizer, parser, and typechecker form the front-end)

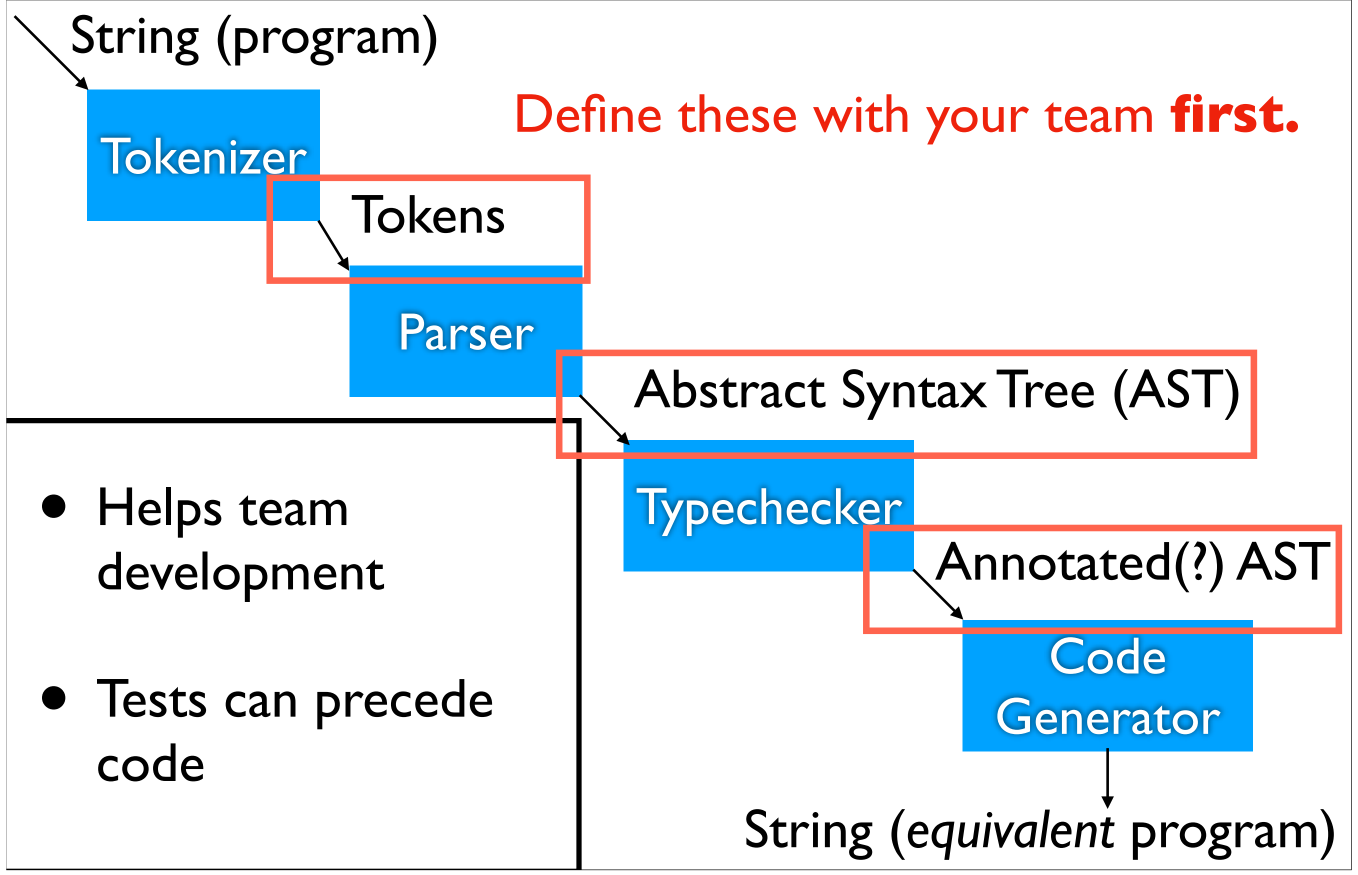
Well-Defined Interfaces



- Helps team development
- Tests can precede code

-Or at least, well-definable
-Assuming the spec is stable(ish), these components could be made independently (we won't be going to this extreme)
-For each component, it should be possible to at least formulate tests without having that component available

Well-Defined Interfaces


String (program)
Tokenizer
Tokens
Parser
Abstract Syntax Tree (AST)
Typechecker
Annotated(?) AST
Code Generator
String (*equivalent* program)
Define these with your team **first.**

- Helps team development
- Tests can precede code

-Or at least, well-definable
-Assuming the spec is stable(ish), these components could be made independently (we won't be going to this extreme)
-For each component, it should be possible to at least formulate tests without having that component available
-If everyone agrees on the interface, it's possible to divide work without stepping on each other's toes. Otherwise, it's a nightmare (based on observations from last time)

Project Information

Into the Lexer / Tokenizer

-These terms mean the same thing

Basic Idea

- Break input into words, called "tokens"
- Every language has its own specific set of tokens

Example

```
if (x < 7) {
  y = true;
} else {
  y = false;
}
```

Example

```
if (x < 7) {
  y = true;
} else {
  y = false;
}
```

`if`	`(`	`var("x")`	`<`
`int(7)`	`)`	`{`	`var("y")`
`=`	`true`	`;`	`}`
`else`	`{`	`var("y")`	`=`
`false`	`;`	`}`	

Tokenization Handout

Livecoded Tokenizer