COMP 430 Lecture 1

Kyle Dewey

About Me

- My research:
  - Automated test case generation, particularly on testing compilers
- Fifth time teaching this course

About this Class

- See something wrong? Want something improved? Email me about it! (kyle.dewey@csun.edu)
- I generally operate based on feedback

Bad Feedback

- This guy sucks.
- This class is boring.
- This material is useless.

-I can’t do anything in response to this

Good Feedback

- This guy sucks, *I can't read his writing.*
- This class is boring, *it's way too slow.*
- This material is useless, *I don't see how it relates to anything in reality.*

- I can't fix anything if I don't know what's wrong

-I can actually do something about this!

Motivation

When will I implement a compiler?

When will I implement a compiler?

Probably never.

- *When will I need to reuse my own code?*
- *When will I need to understand how a language works?*
- *When will I need to work on a team?*
- *When will I need to understand why a language was designed a certain way?*

- *When will I need to reuse my own code?*
- *When will I need to understand how a language works?*
- *When will I need to work on a team?*
- *When will I need to understand why a language was designed a certain way?*

Basically always.

-Knowledge of why a language was designed a particular way gives you an appreciation for the features a language has, and can help you spot BS when someone advocates for a given language.

Understanding Language Behavior

-Towards motivating why compiler knowledge can help when it comes to understanding language behavior

Understanding Language Behavior

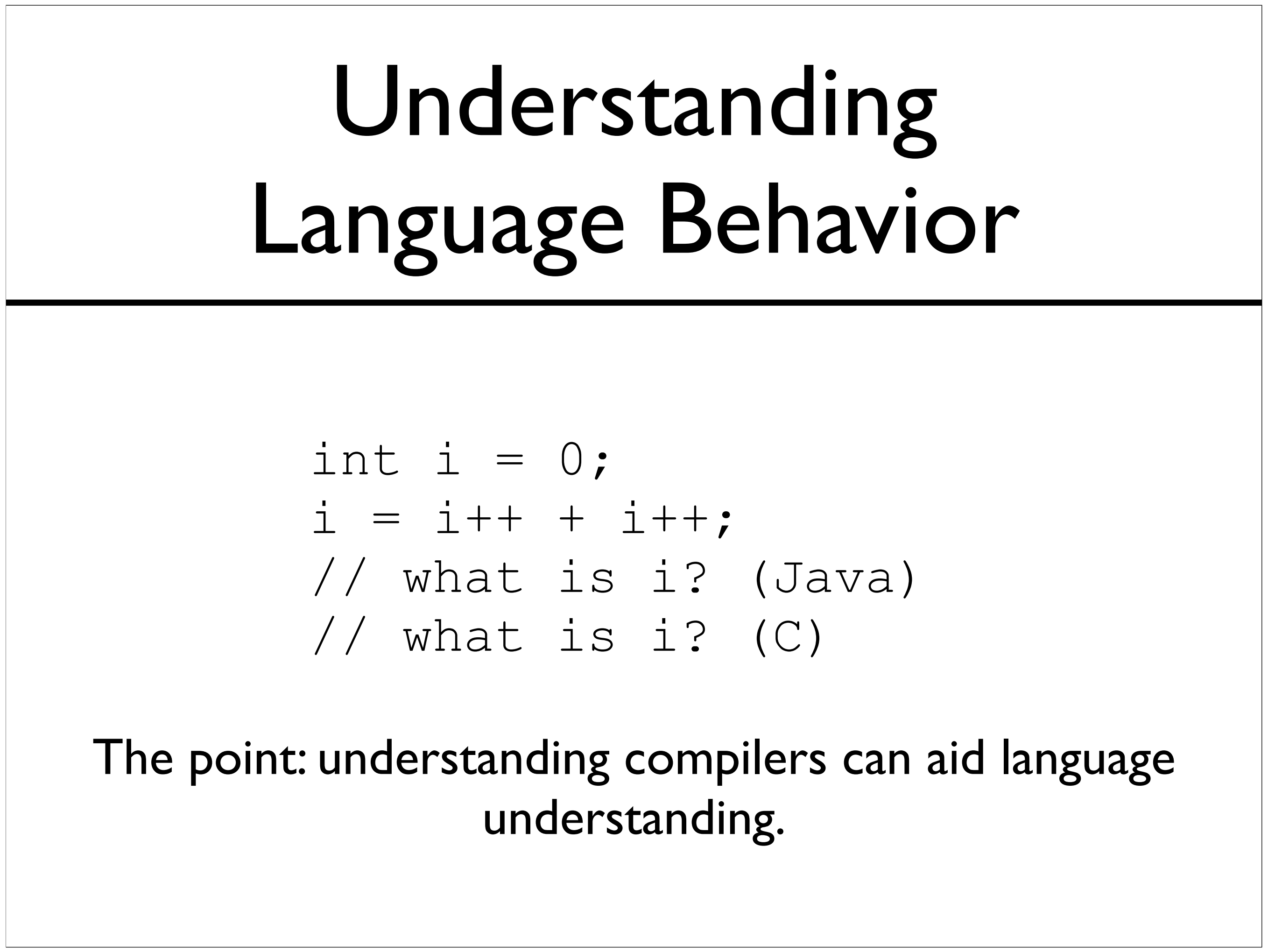
```
int i = 0;
i = i++ + i++;
// what is i? (Java)
```

–Java: 1 (0++ returns 0 and increments i, then 1++ returns 1 and increments i, 0 + 1 = 1)

Understanding Language Behavior

```
int i = 0;
i = i++ + i++;
// what is i? (Java)
// what is i? (C)
```

-Undefined behavior (your fault as the programmer)
-Reflects major differences between the design mindset of Java (a safe, predictable language) and C (a fast language which the compiler can optimize the hell out of)



Understanding Language Behavior

```
int i = 0;
i = i++ + i++;
// what is i? (Java)
// what is i? (C)
```

The point: understanding compilers can aid language understanding.

-Undefined behavior (your fault as the programmer)

Course Design

- Emphasis on modern compilers
  - Minimal parsing
  - Minimal ultra low-level stuff

-We don't get into LL, LR, bison, flex, etc. If you know them, you can use them. I don't see them often in practice. These were originally developed because memory was scarce in the early days.
-I don't require you to compile to assembly. You can compile to JavaScript for all I care, and this isn't so strange anymore. In fact, compiling to assembly is now relatively strange (just use LLVM)

Course Design

- Emphasis on modern compilers
  - Minimal parsing
  - Minimal ultra low-level stuff
- It's about writing code

-There is a lot of theory that goes into compilers, but my interest is more on the pragmatic side.
-Everything is about the compiler you're writing.

Course Design

- Emphasis on modern compilers
  - Minimal parsing
  - Minimal ultra low-level stuff
- It's about writing code
- It's about teamwork

-Yes, you have to work in teams. But I'm going to put some mechanisms in place to keep people from leeching. You're writing relatively large software, and this is done in teams.

Choose Your Own Adventure

- Choose from one of nine premade language design proposals; tweak as desired
  - Alternatively, if you have prior experience: design your own language with certain kinds of features
- Incrementally implement those features
- By the end, you'll have a compiler

Fair Warning

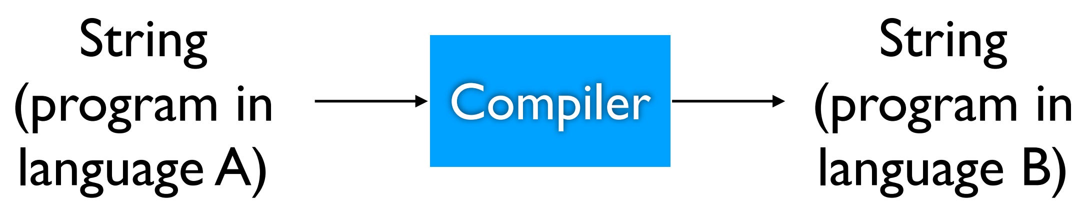
- This is a **lot** of work
- I will try to give you effectively lab time in class, when possible
- As we progress, lectures may get more specialized (depends on you)

Syllabus

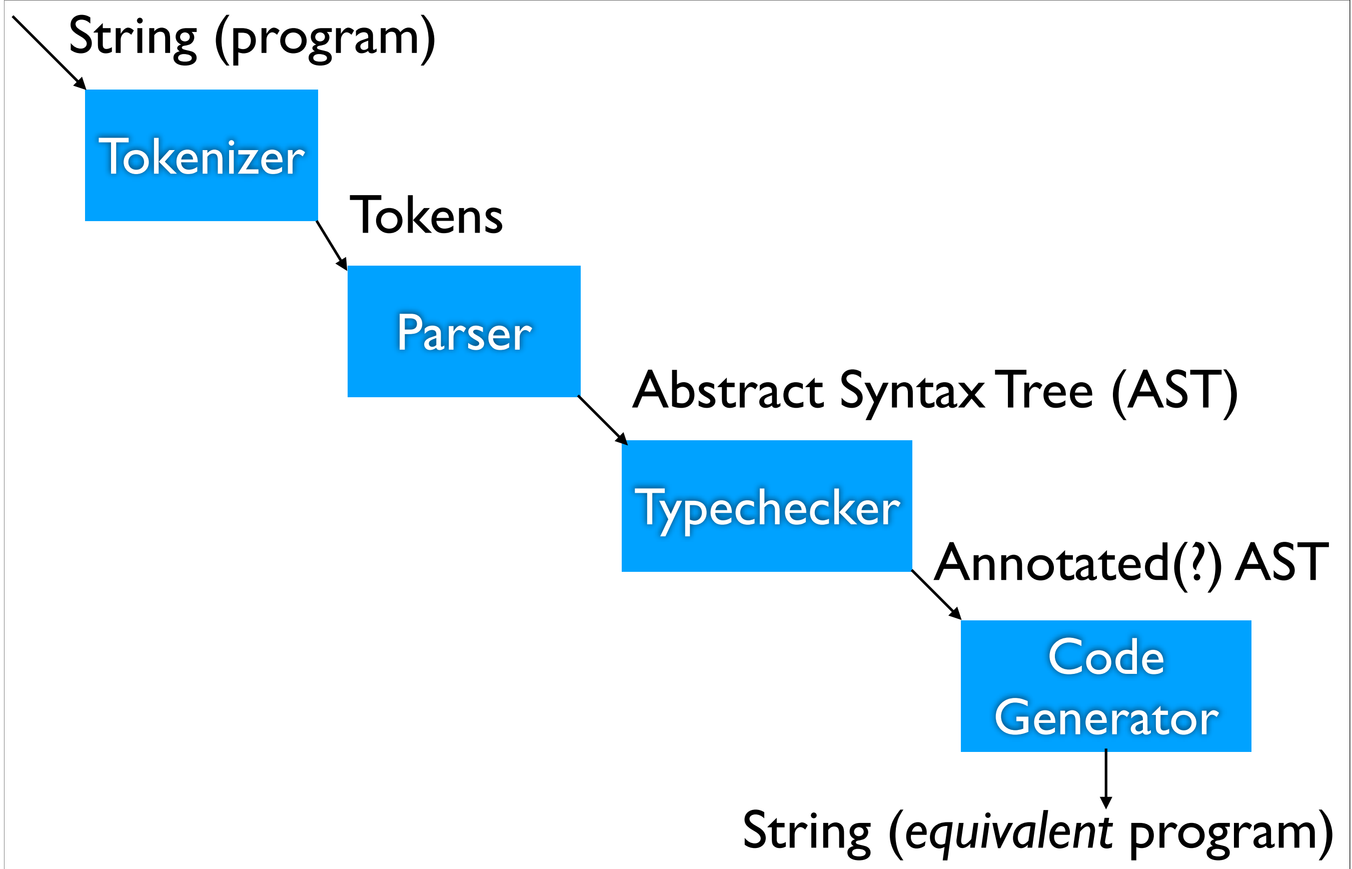
Project Information

Birds-eye View

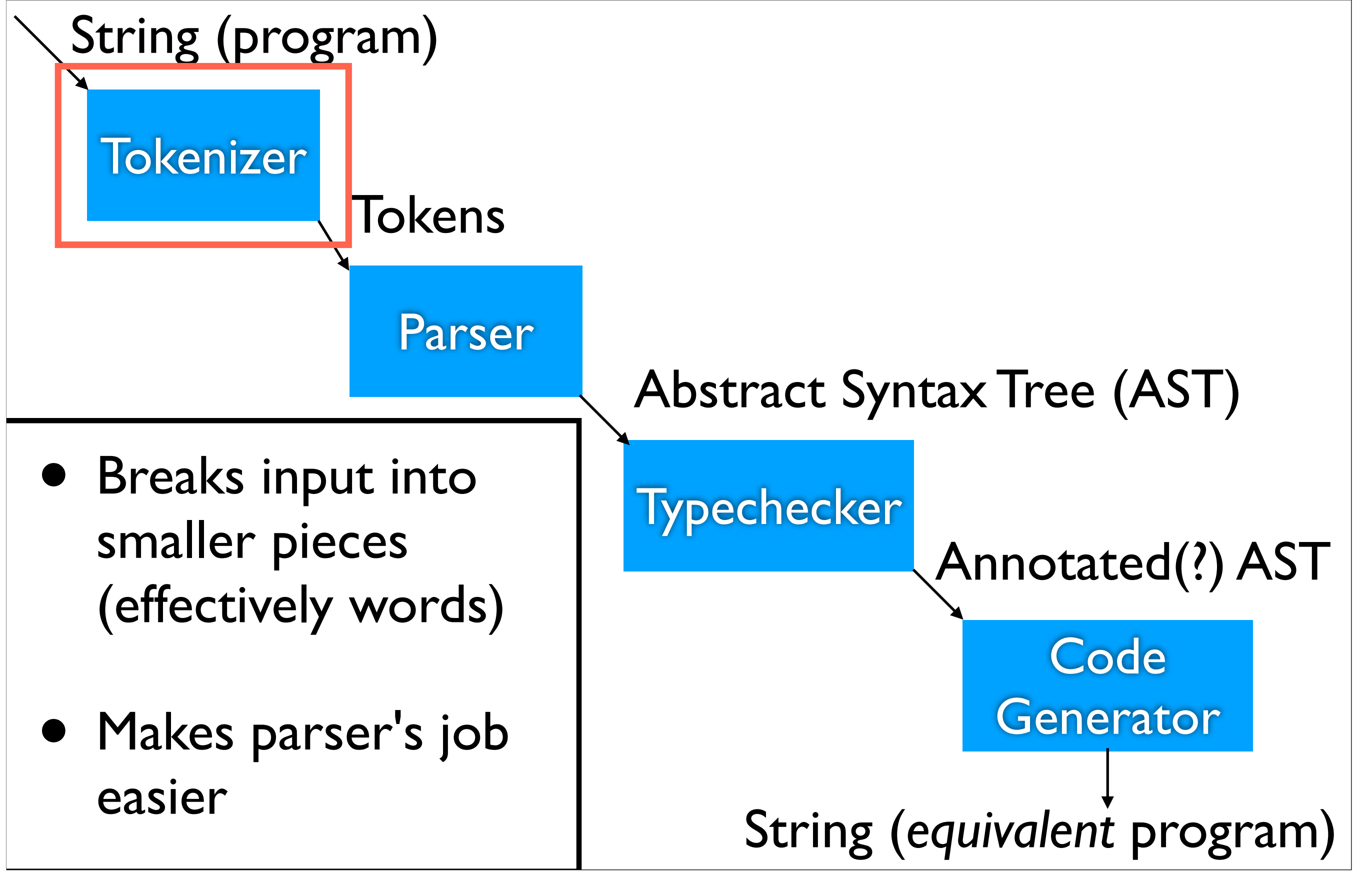
Compiler


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

Compiler Architecture



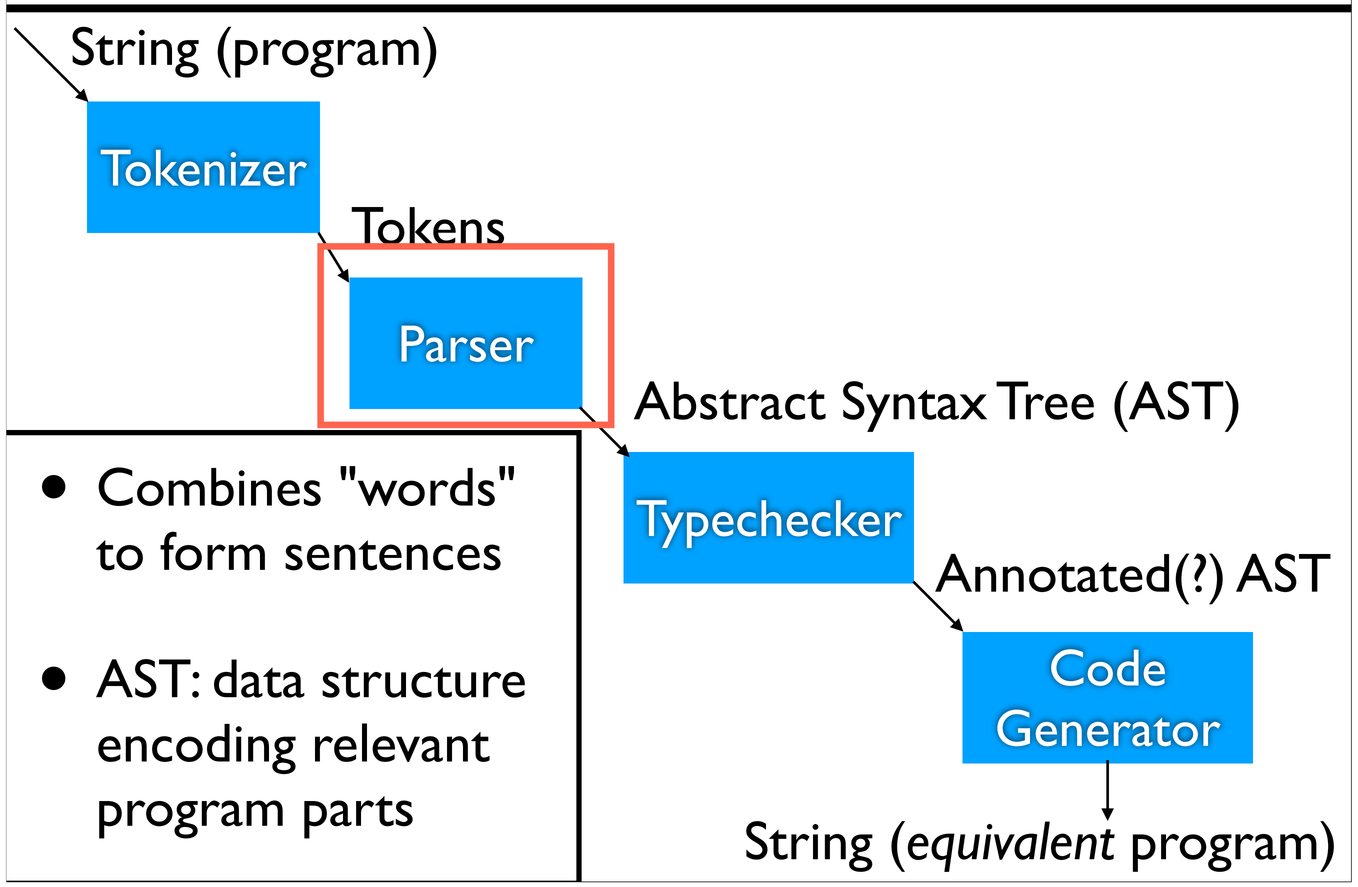
Compiler Architecture



- 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

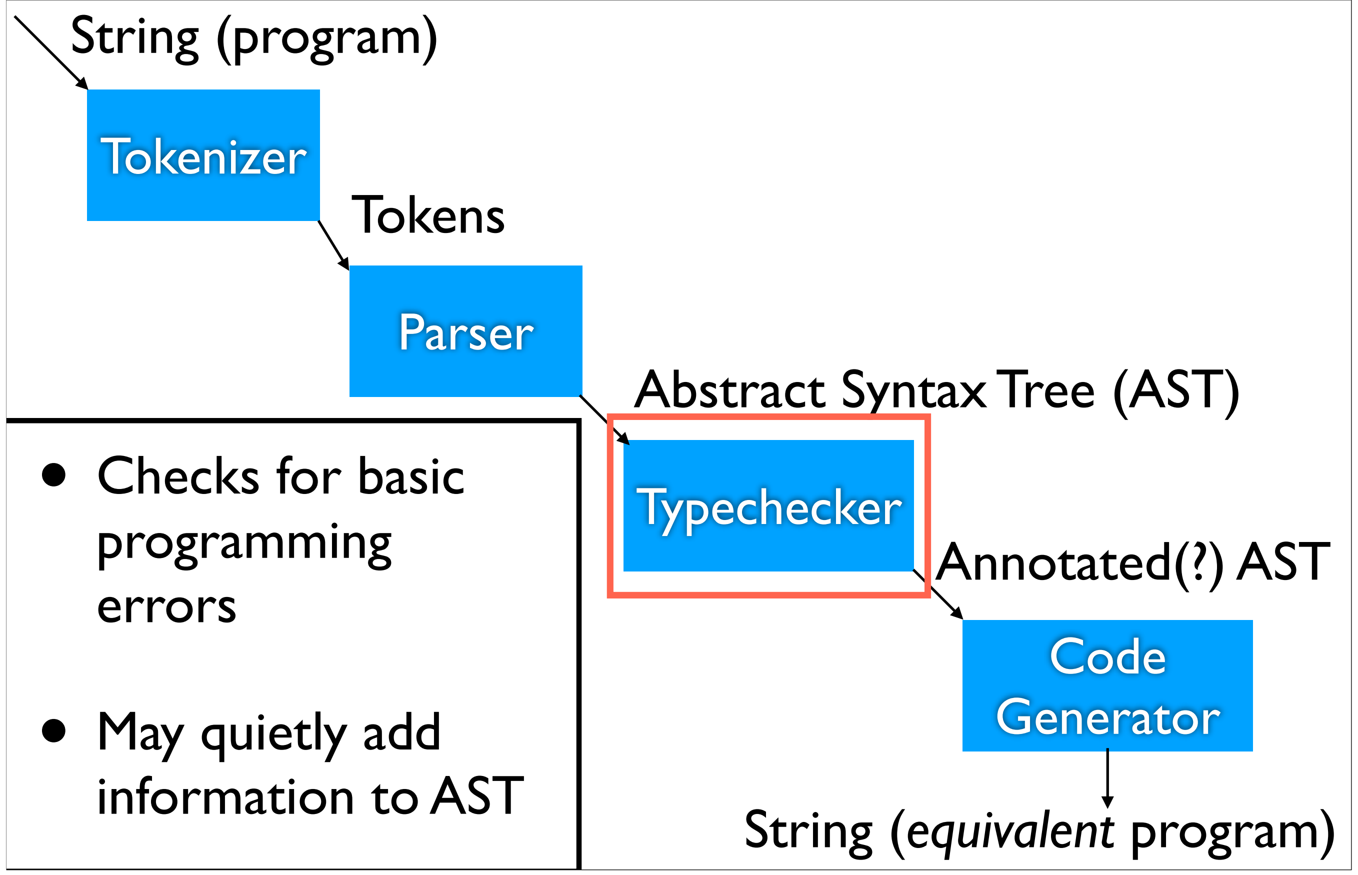
Compiler Architecture



- 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

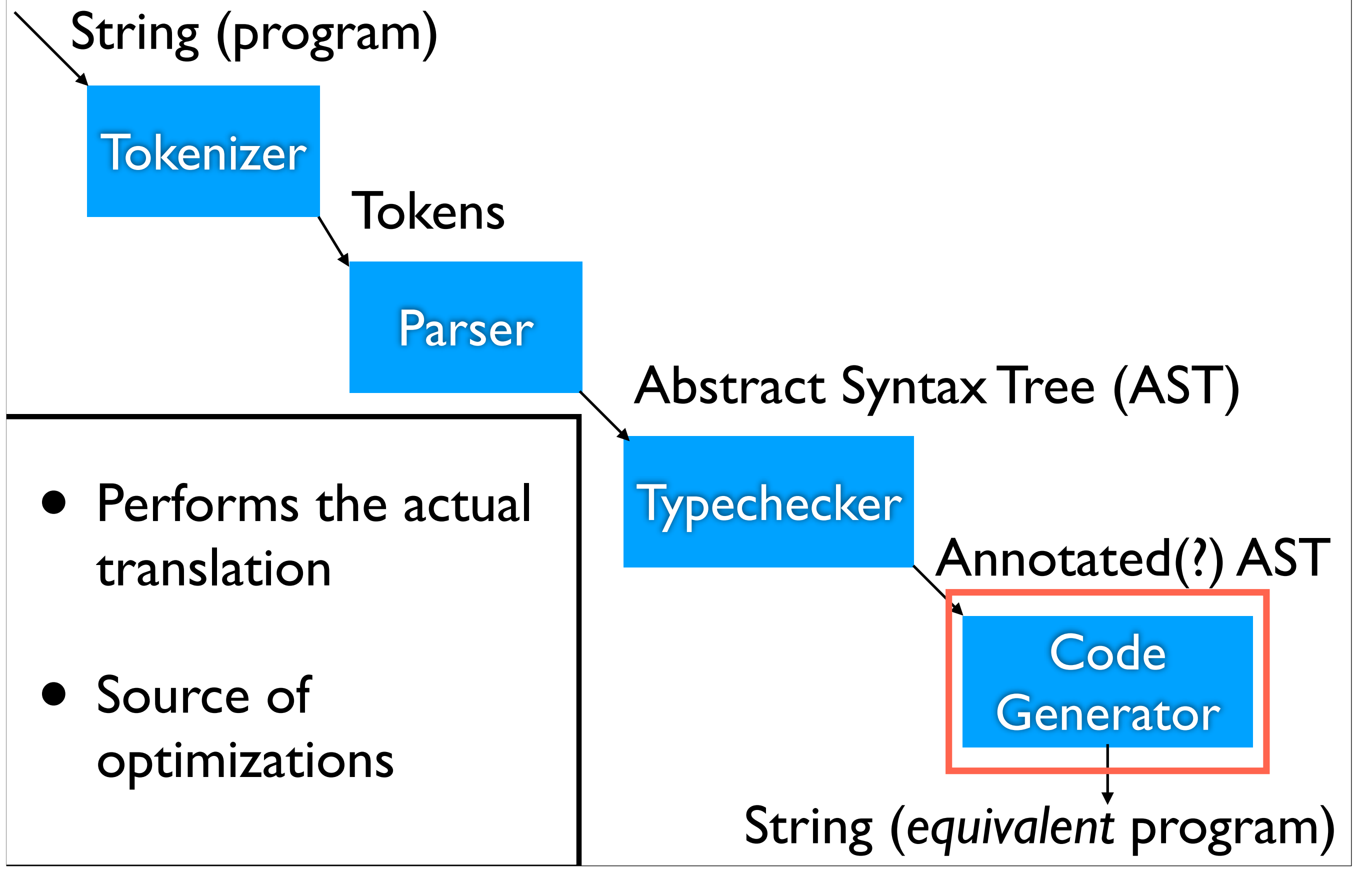
Compiler Architecture



- 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)

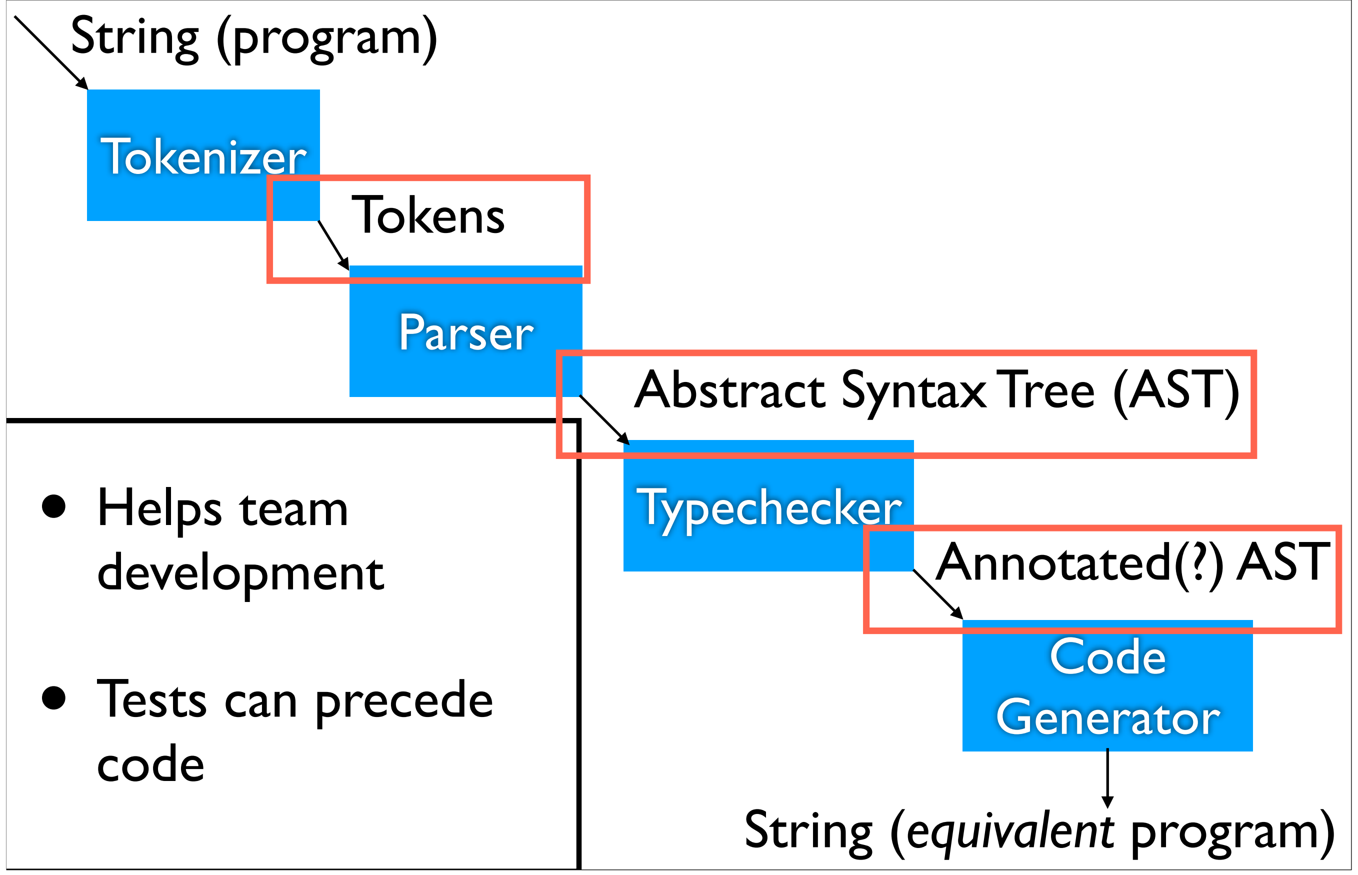
Compiler Architecture



- 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)

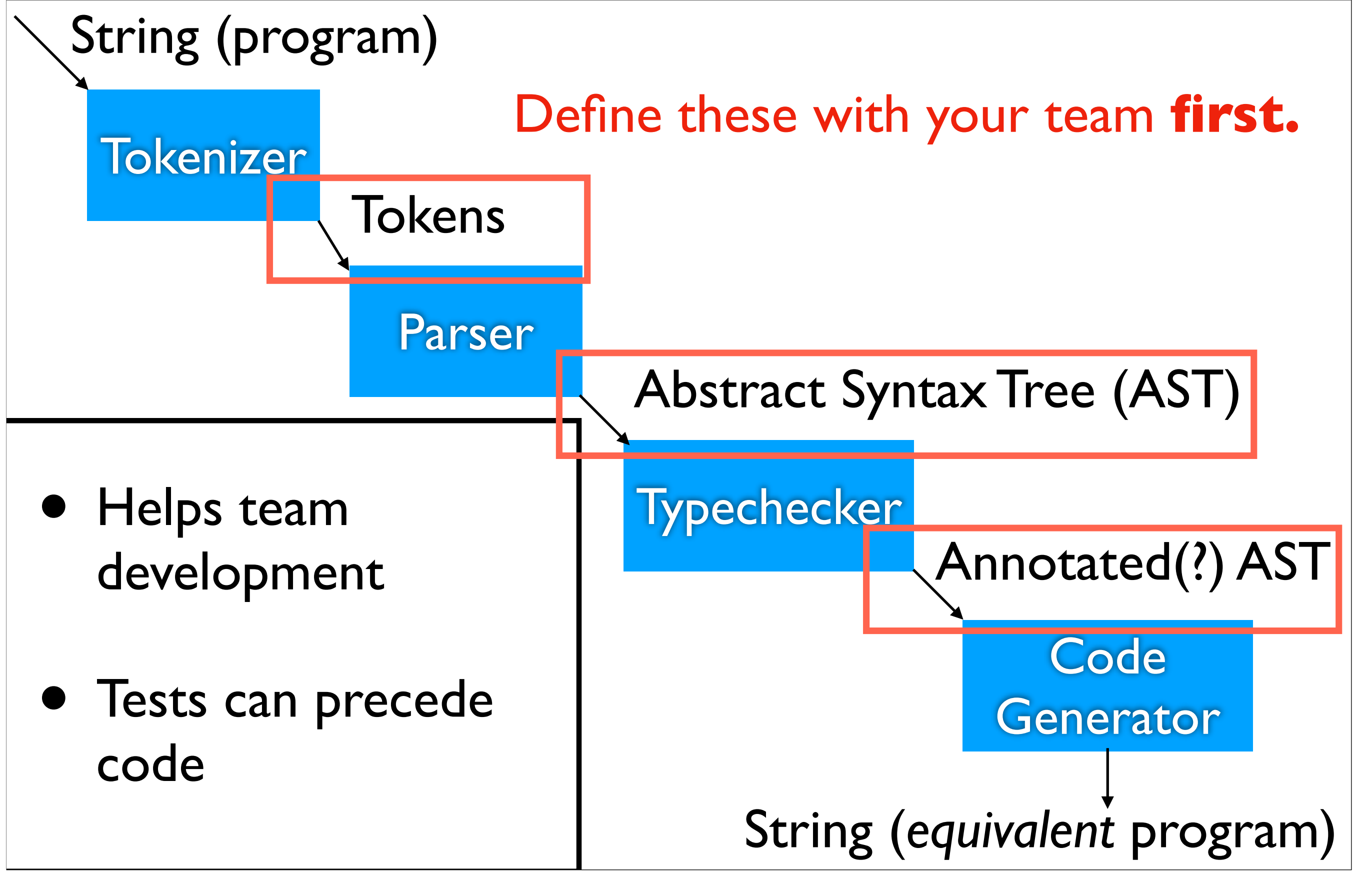
Well-Defined Interfaces



- 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)

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