



# COMP 520 - Compilers

## Programming Assignment 1 – Syntactic Analysis

# Programming Assignment 1

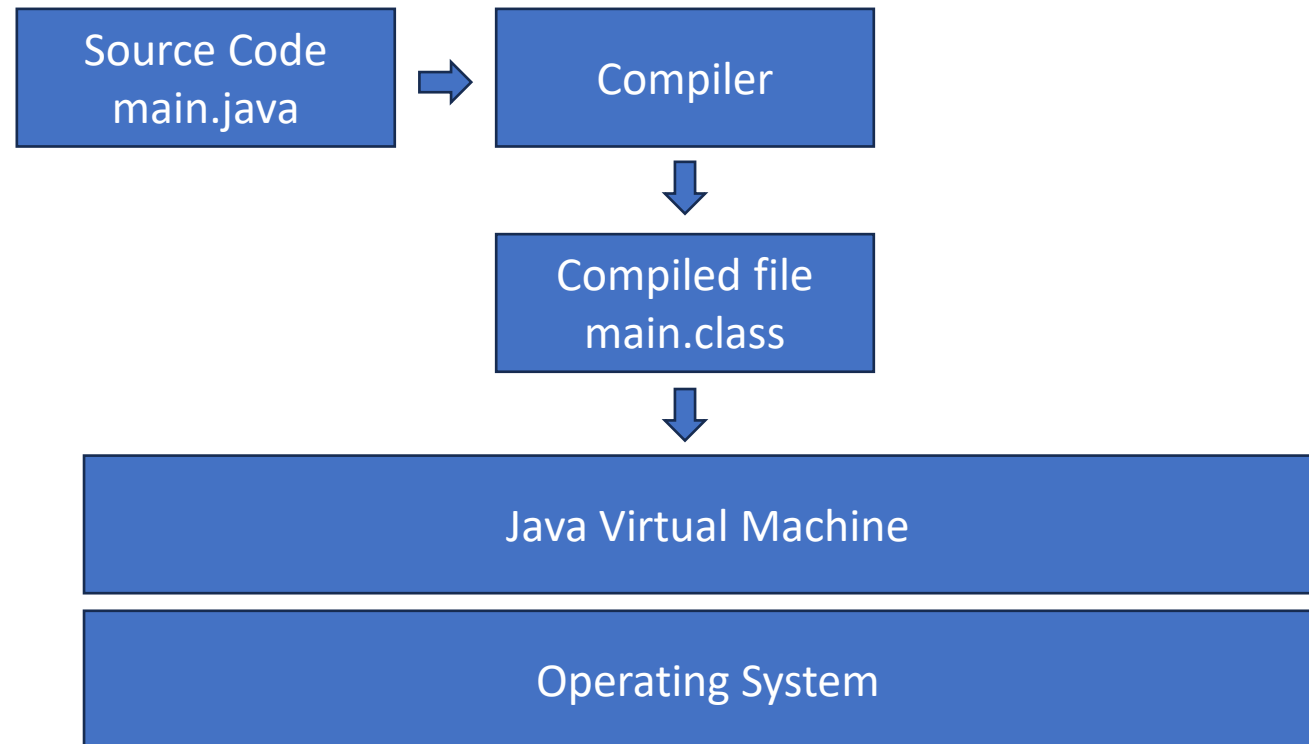
- PA1 has two parts, Lexical Analysis and Syntax Checking.
- Start early, and visit office hours if you have any questions.
- Autograder will be up later today (1/16)
- Due: 1/31/24 at 11:59pm

# Goals

- miniJava is a subset of the Java programming language.
- A miniJava program is a valid Java program.
- Semester Goal: Create a compiler that can take in a miniJava source code file and output a binary file that can be executed.

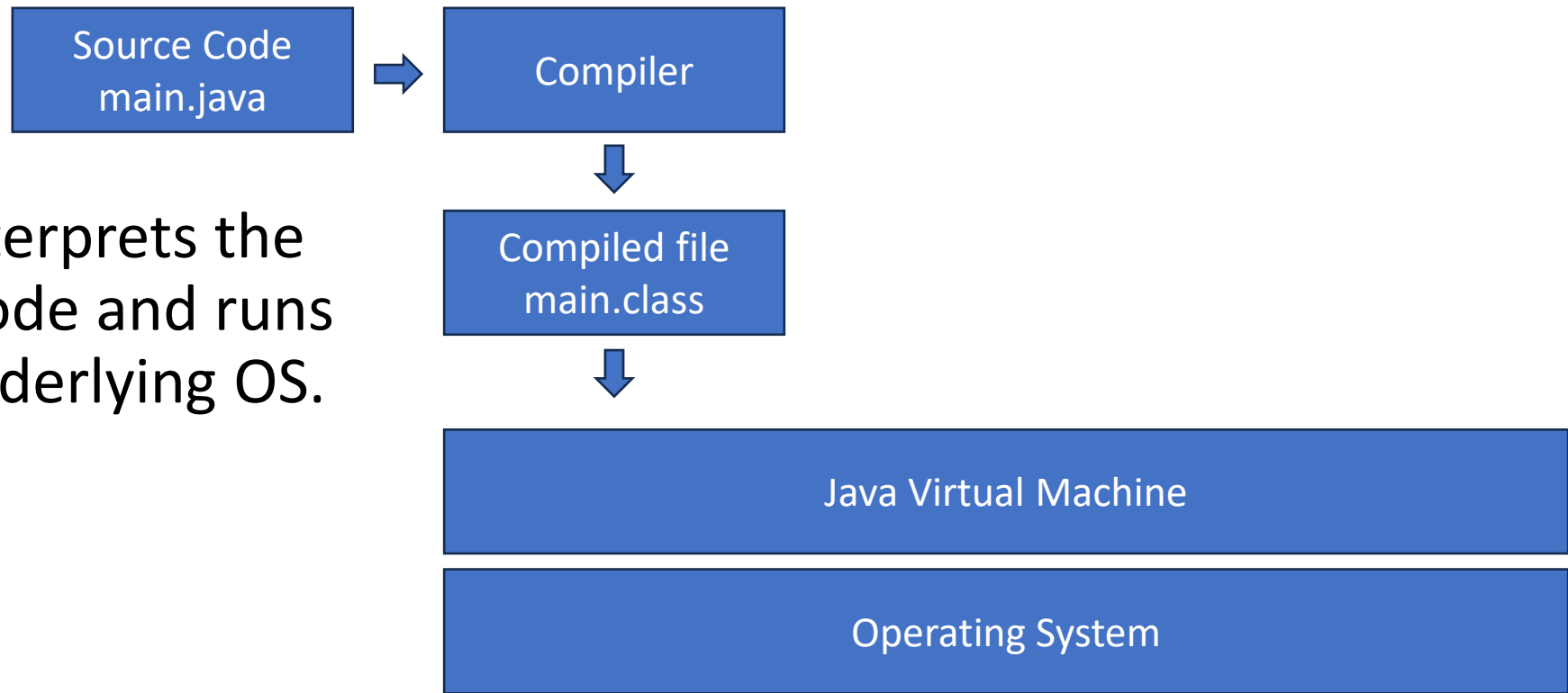
# Java Runtime

- Executable binaries for Java run on the JVM



# Java Runtime

- This allows Java programs to run “cross-platform”



- The JVM interprets the compiled code and runs it on the underlying OS.

# miniJava

- The miniJava compiler will NOT output JVM bytecode.
- Instead, the compiler will output bytecode that will run natively on a modern x86\_64 Linux operating system.

# miniJava

- The miniJava compiler will NOT output JVM bytecode.
- Instead, the compiler will output bytecode that will run natively on a modern x86\_64 Linux operating system.
- We lose cross-platform capabilities
- We gain learning how compilers target physical processors/OSes



# What are the goals of PA1?



# What are the goals of PA1?

**Lexical Analysis**

**Syntactic Analysis**

# What are the goals of PA1?

## **Lexical Analysis**

- Create the language's lexicon
- The lexical unit is the Token

## **Syntactic Analysis**

# What are the goals of PA1?

## **Lexical Analysis**

- Create the language's lexicon
- The lexical unit is the Token
- Take in a few letters at a time, and return a Token

## **Syntactic Analysis**

# What are the goals of PA1?

## **Lexical Analysis**

- Create the language's lexicon
- The lexical unit is the Token
- Take in a few letters at a time, and return a Token
- Responsible for removing whitespace and comments
- Output is a stream of tokens

## **Syntactic Analysis**

# What are the goals of PA1?

## **Lexical Analysis**

- Create the language's lexicon
- The lexical unit is the Token
- Take in a few letters at a time, and return a Token
- Responsible for removing whitespace and comments
- Output is a stream of tokens

## **Syntactic Analysis**

- Input is a stream of tokens
- Only care about syntax

# What are the goals of PA1?

## **Lexical Analysis**

- Create the language's lexicon
- The lexical unit is the Token
- Take in a few letters at a time, and return a Token
- Responsible for removing whitespace and comments
- Output is a stream of tokens

## **Syntactic Analysis**

- Input is a stream of tokens
- Only care about syntax
  - This analyzer doesn't see whitespace, nor comments
  - Only concerned about code syntax

# What are the goals of PA1?

## Lexical Analysis

- Create the language's lexicon
- The lexical unit is the Token
- Take in a few letters at a time, and return a Token
- Responsible for removing whitespace and comments
- Output is a stream of tokens

## Syntactic Analysis

- Input is a stream of tokens
- Only care about syntax
  - This analyzer doesn't see whitespace, nor comments
  - Only concerned about code syntax
- Output is an AST (PA2)

# Whitespace!

- Consider the following C code:

X---X	X-----X	X-----X
X-- - X	X-- - -X	X-- - --X

- Predecrement, Postdecrement, Subtraction, Negation
  - X      X--      x-y      -X



# Scanning C++

- Consider the following C++ code:

Foo<Bar>	cin >> var	Foo<Bar<Bazz>>
----------	------------	----------------

- Operator >>
- Question: How would you *correctly* scan >> in the example above?



# Let's start with Lexical Analysis



# PA1- TokenType.java

- This is the equivalent of Java's TokenKind class (more on this soon)

# PA1- TokenType.java

- This is the equivalent of Java's TokenKind class (more on this soon)
- In the TokenType enumeration, we list the possible types of tokens we want to stream to our syntactic analyzer.
- But what are the types of tokens that we have??

# Taking a look at Java's TokenKind

- <https://www.javadoc.io/static/org.kohsuke.sorcerer/sorcerer-javac/0.11/com/sun/tools/javac/parser/Tokens.TokenKind.html>

# Taking a look at Java's TokenKind

- <https://www.javadoc.io/static/org.kohsuke.sorcerer/sorcerer-javac/0.11/com/sun/tools/javac/parser/Tokens.TokenKind.html>
- Pretty much everything is in there!
- If you want, you can organize your TokenType similarly

# Analytically determine TokenType

- In-class exercise to determine where LexicalAnalysis ends and Syntactic Analysis starts
- Graphs!



How would one even define  
“Syntax”?



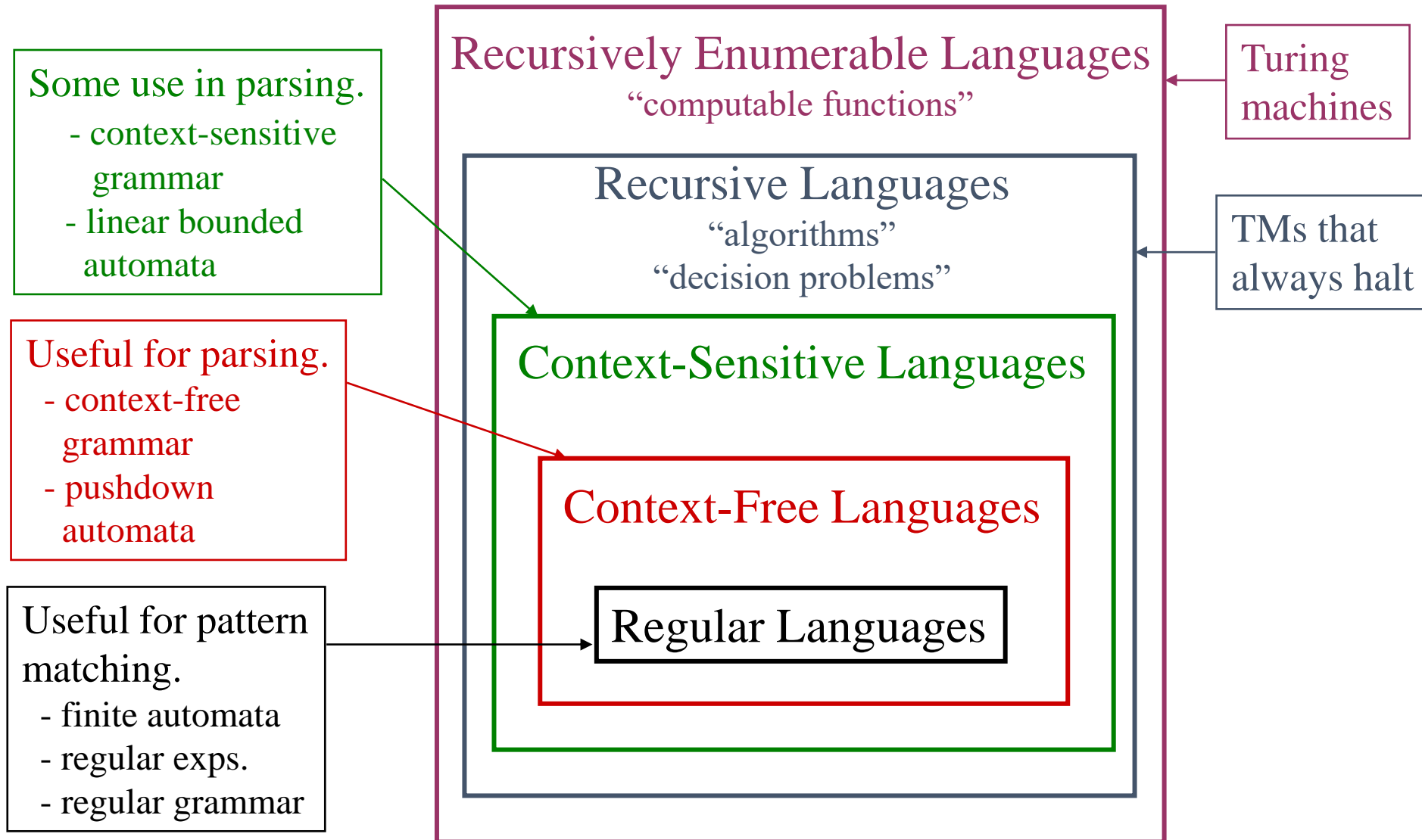
# Syntax

- Arrangement of words and phrases to create a **well-formed** sentence in a language
- For example: Article Noun Verb Article Noun:
  - The cat chases the mice
- What is another way to use the above syntax?

# Syntax

- Arrangement of words and phrases to create a **well-formed** sentence in a language
- For example: Article Noun Verb Article Noun:
  - The cat chases the mice
- However, this does not always make well-formed sentences!
  - The cat chases **a** mice

# Languages Covered in 455



# What about Regular Languages?

- Can anyone give me the regular expression for a US telephone number?

# What about Regular Languages?

- Can anyone give me the regular expression for a US telephone number?
- Something like: `1?\h?(\d){3}\h*-?\h*(\d){3}\h*-?\h*(\d){4}`
- And that doesn't even include using parenthesis in the area code, like (234) 567 - 8901

# What about Regular Languages?

- Can anyone give me the regular expression for an email address?

# What about Regular Languages?

- Can anyone give me the regular expression for an email address?
- Did you guess something like:
  - `[\w-\._]+@[\w-]+\.[\w-]+`

[illegible]



# Regular Expressions

- Okay, maybe regular languages might be a little too complex when trying to specify the syntax for a programming language
- Are there other problems with regular languages?

# Consider the Grammar for Identifiers

- Identifier ::= Letter | Identifier Letter | Identifier Digit | Identifier \_
- Letter ::= a | b | c ...
- Digit ::= 0 | 1 | 2 ...
- This grammar is not regular (in its current form without modification, it does not translate to a regular language) *Why?*

# Consider the Grammar for Identifiers

- Identifier ::= Letter | Identifier Letter | Identifier Digit | Identifier \_
- Letter ::= a | b | c ...
- Digit ::= 0 | 1 | 2 ...
- This grammar is not regular (in its current form without modification, it does not translate to a regular language)
- Conveniently, we can just move around where the recursion occurs

# Consider the Grammar for Identifiers

- Identifier ::= Letter | Identifier Letter | Identifier Digit | Identifier \_
- Letter ::= a | b | c ...
- Digit ::= 0 | 1 | 2 ...
- Identifier ::= (a | b | c...)( a | b | c ... | 0 | 1 | 2 ... | \_ )\*
- And now the recursion is on the right-hand-side

# Syntax

- Formally describing the syntax of a programming language is best done through a context-free grammar
- Additional lexical rules needed (comments, whitespace, etc.)
- Note identifiers cannot be reserved words, the lexical analyzer will output such words as their reserved TokenType and the syntax will throw an error when it got a reserved word instead of an identifier.
- Contextual constraints can also be made formal, but more on this in PA2

# Context-Free Grammars

- CFGs are an excellent way to describe the syntax of a language.
- They are clear, easier to understand than regular expressions, and give you a bit more flexibility. (No need to worry about left-regular or right-regular CFGs)

# Context-Free Grammers

- CFGs are an excellent way to describe the syntax of a language.
- Consider the following definition of a sentence:

Sentence	::= Subject Predicate Object
Subject	::= Article Noun
Predicate	::= Verb
Object	::= Article Noun
Article	::= a   the
Noun	::= dog   cat   mice
Verb	::= chase   chases

# CFG Components

Sentence	::= Subject Predicate Object
Subject	::= Article Noun
Predicate	::= Verb
Object	::= Article Noun
Article	::= a   the
Noun	::= dog   cat   mice
Verb	::= chase   chases

<b>Terminals</b>	{a, the, dog, cat, mice, chase, chases}
<b>Non-terminals</b>	{Sentence, Subject, Predicate, Object, Article, Noun, Verb}
<b>Start symbol</b>	Sentence



# Language

- The language generated from our CFG is a set of sentences
- Each sentence can be generated by repeated application of the rules
- Note: Each valid sentence generated from the CFG can be viewed entirely as an ordered tuple of terminals.

# Recursion

- What if we want to allow “the dog and the cat” as the subject?
- Currently it is:

Sentence ::= Subject Predicate Object

# Recursion

- What if we want to allow “the dog and the cat” as the subject?

Sentence ::= Subject (Conjunction Subject)\* Predicate Object

# Recursion

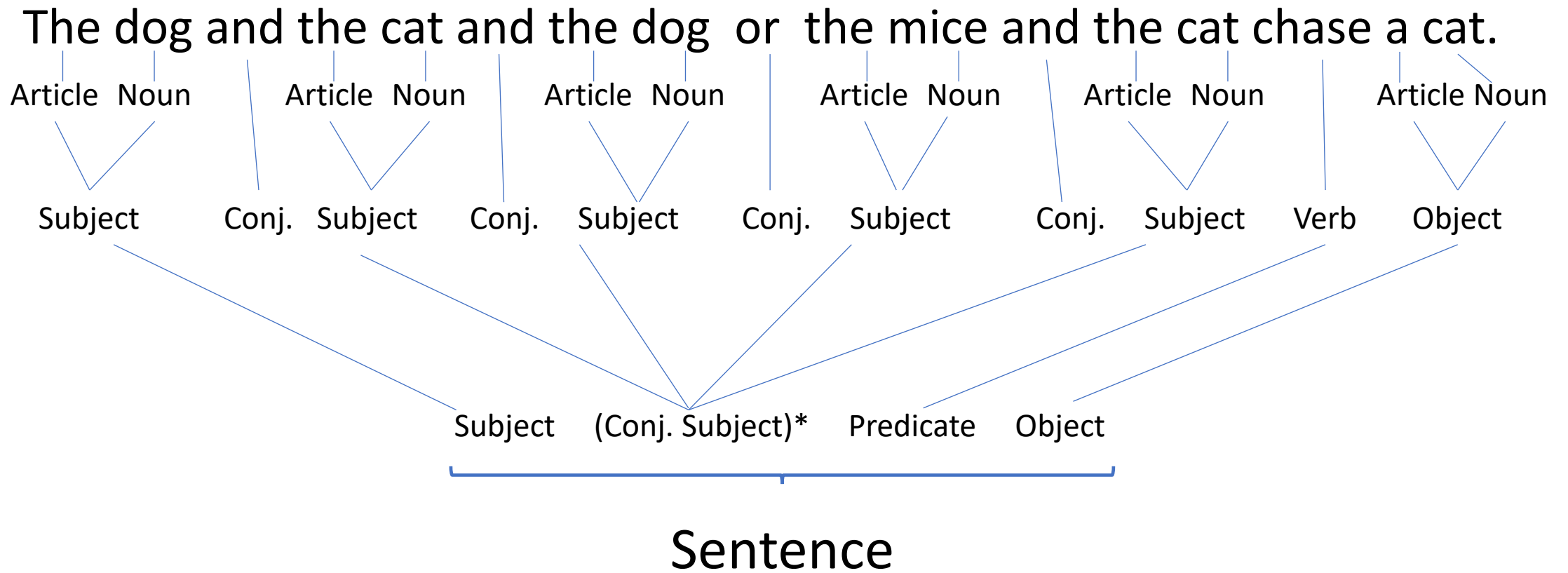
- What if we want to allow “the dog and the cat” as the subject?

Sentence ::= Subject (Conjunction Subject)\* Predicate Object

- This will enable:

The dog and the cat and the dog and the mice and the cat chase a cat.

# Recursion



# Limitations of CFGs

- Can generate incorrect sentences.

“The cat chase a mice”

- Thus, *simple* CFGs are suitable for describing syntax, but “chase” contextually needs to be “chases” and “mice” cannot be preceded with “a”.

# Limitations of CFGs

- Can generate incorrect sentences.

“The cat chase a mice”

- Thus, *simple* CFGs are suitable for describing syntax, but “chase” contextually needs to be “chases” and “mice” cannot be preceded with “a”.
- For PA1, we only focus on syntactic analysis and not contextual. Therefore, “The cat chase a mice” is a valid sentence.. for now.

# Mini-Triangle Language

- Expressions for mini-Triangle, for simple arithmetic operations
- $\text{Exp} ::= \text{PrimExp} \mid \text{Exp Oper PrimExp}$
- $\text{PrimExp} ::= \text{intlit} \mid \text{id} \mid \text{Oper PrimExp} \mid ( \text{Exp} )$
- $\text{Oper} ::= + \mid - \mid * \mid / \mid < \mid > \mid =$



# Mini-Triangle Language

- Commands for Mini-Triangle
- Program  $::= \text{Cmd}$
- Cmd  $::= \text{id} := \text{Exp} \mid \text{let Decl in Cmd}$
- Decl  $::= \text{var id} : \text{type}$

# Examples

- Is this a valid command for mini-Triangle?

```
let var x: Integer in x := 5 + (2*10)
```

- For each token, classify it according to the grammar.
  - Use Oper, intlit, id, let, : (colon), type, in, := (assign), var, lparen, rparen

# Examples

- Is this a valid command for mini-Triangle?

```
let var x: Integer in x := 5 + (2*10)
```

- For each token, classify it according to the grammar.
- Now group the tokens according to the grammar symbols.
  - What tokens constitute a Decl, Cmd, Exp, PrimExp?

# Examples

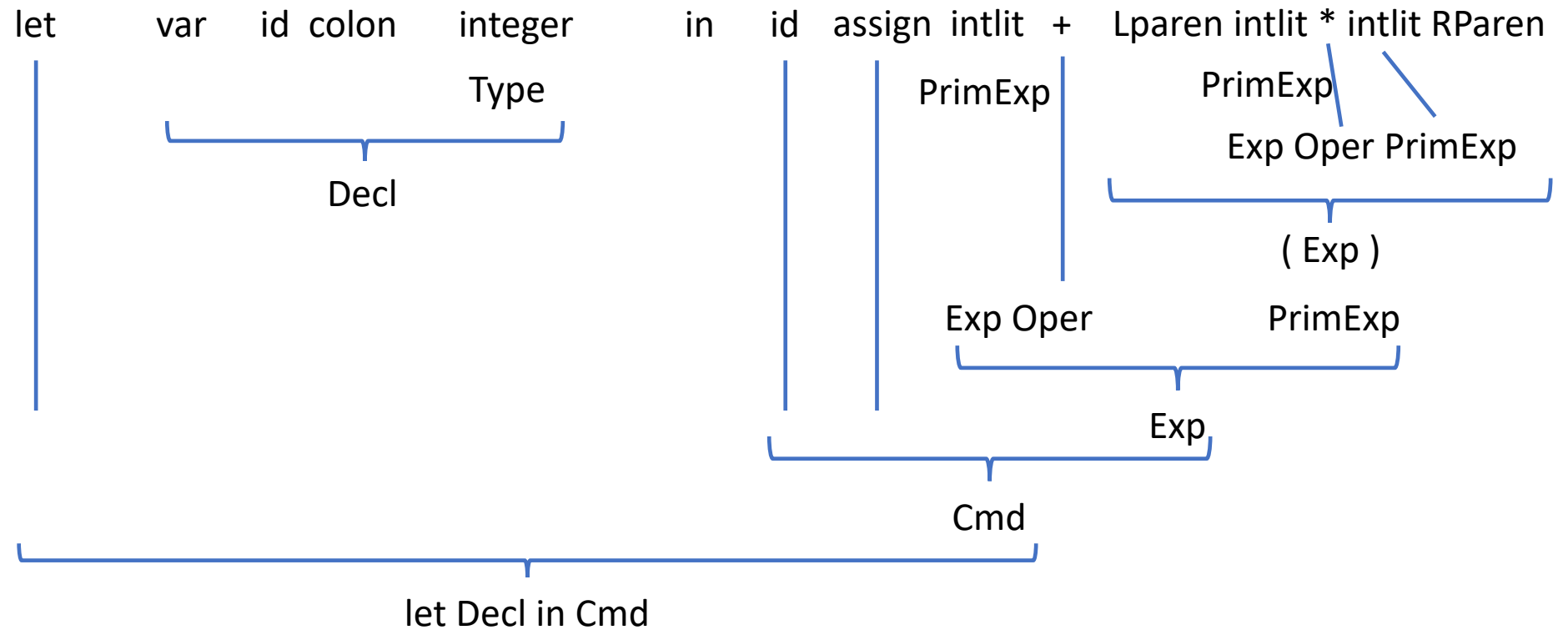
- Is this a valid command for mini-Triangle?

```
let var x: Integer in x := 5 + (2*10)
```

```
let      var    id colon  integer      in   id  assign intlit +  Lparen intlit * intlit RParen
```

# Examples

let var x: Integer in x := 5 + (2\*10)





# Step 1: Where to begin in PA1?

# Simple Arithmetic Scanner/Parser

- Courtesy of Professor Prins, a simple Arithmetic Scanner and Parser is on the course website.
  - This corresponds to arithmetic parsing in miniTriangle
- You can use this as an example on how Scanners and Parsers are structured.

# Starter Code

- PA1 starter code is on the course website
- You do not have to use the starter code!
- The only thing that is important is that the Compiler class contains your main function, and the Compiler class is contained in the miniJava package.





# Starter Code (2)

- Two easy steps!
- Step 1: Download the starter code from github
- Step 2: In the “pa1/src/miniJava” folder, copy these starter files to your project
- How this works depends on your IDE.

# Error Reporter

- Start with something easy, the error reporter is an object that records errors as they arrive.
- Complete the methods:
  - `hasErrors`– check to see if the `errorQueue` is non-empty
  - `outputErrors`– iterate through the `errorQueue` and output all strings

# Error Reporter – Extra Credit

- The error reporter contains one of few PA5 extra credits that can be applied in PA1 and not impact the autograder.
- Consider augmenting the error reporter by requiring any call to `reportError` to also specify a line and column number.
- This will prove extremely helpful when debugging your code!

# Token class

- Another easy class to complete is the Token class.
- The token is classified by String and a TokenType.
- A token has this underlying string to represent the original text that resulted in a specific TokenType.
- Implement the constructor, `getTokenType` and `getTokenText` methods.
- And thus ends the Java warmup, now we get into the fun stuff!

# TokenType

- Consider the in-class exercise, what types of tokens do you want?
- List these token types in the TokenType enumeration.
- Note: You can optionally follow Java's TokenKind implementation where nothing is consolidated, but it will make your Parser slightly lengthier.

# Compiler.java- contains main method

- How the miniArith example works is that a `FileInputStream` is created, and the PA1 starter files are structured similarly.
- Our file in question is in `args[0]`
  - The autograder will always specify a file path, but it is good practice to error check to make sure whether the argument has been specified or not.
- The Scanner object takes such an input stream to create tokens.
- The Parser object takes a Scanner object and the `ErrorReporter`. It will report syntax errors.

# Scanner Design

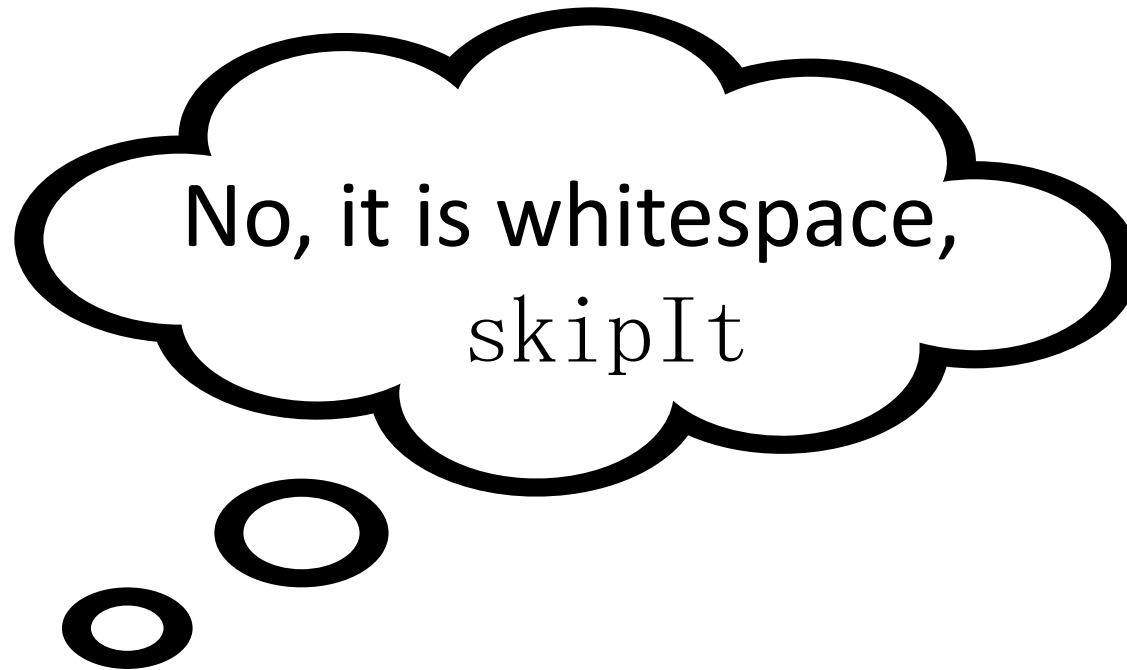


Do I want the  
current letter?

Scanner

Current Letter: " "  
( a space )  
Current Text: ""

# Scanner Design



Scanner

Current Letter: " "  
( a space )  
Current Text: ""



# Scanner Design

I like this letter, but  
unsure of what the TokenType  
is (probably identifier)

takeIt

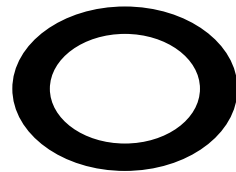
Current Letter: "c"

Current Text: ""

after takeIt:

Current Text: "c"

Scanner ○



# Scanner Design

If it is an identifier  
then keep accepting until  
whitespace or something  
not allowed in identifiers

takeIt

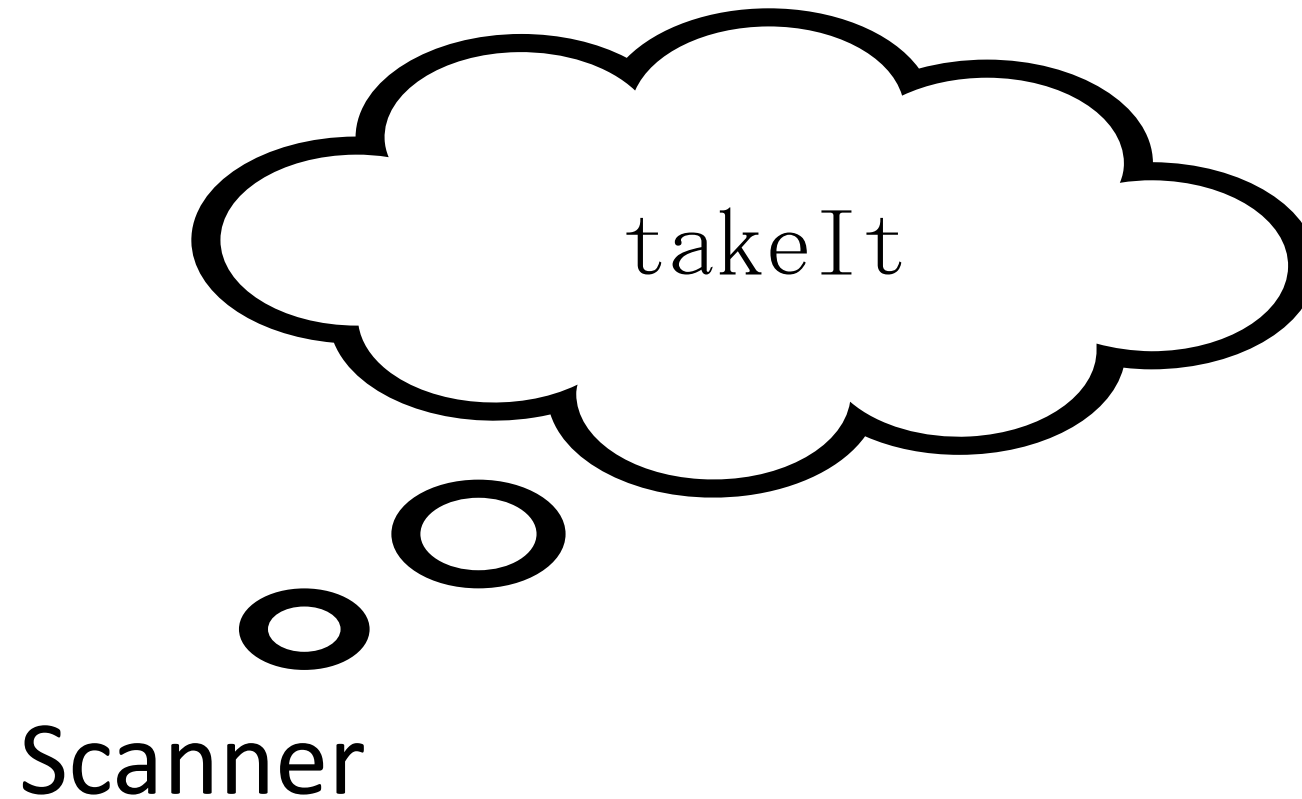
Current Letter: "l"  
Current Text: "c"

after takeIt:

Current Text: "cl"

Scanner

# Scanner Design



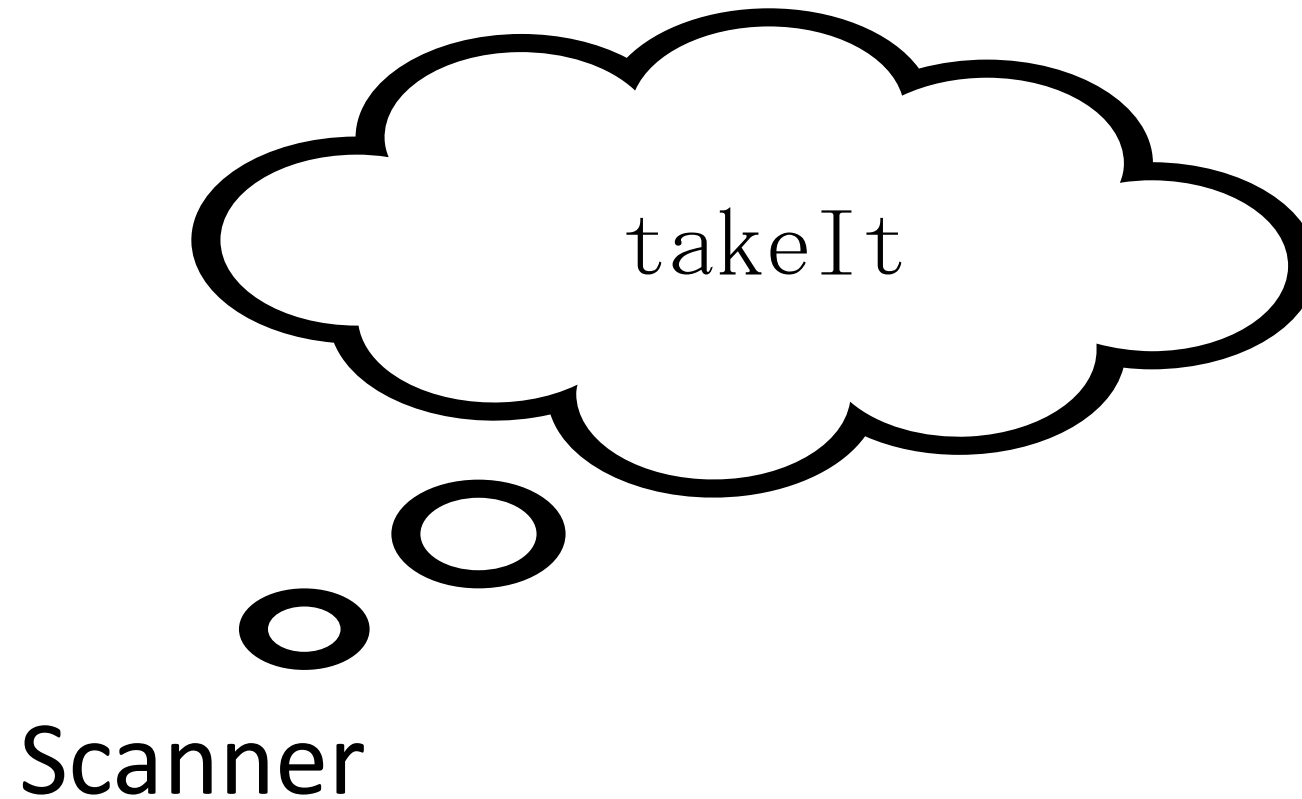
Current Letter: "a"

Current Text: "cl"

After:

Current text: "cla"

# Scanner Design



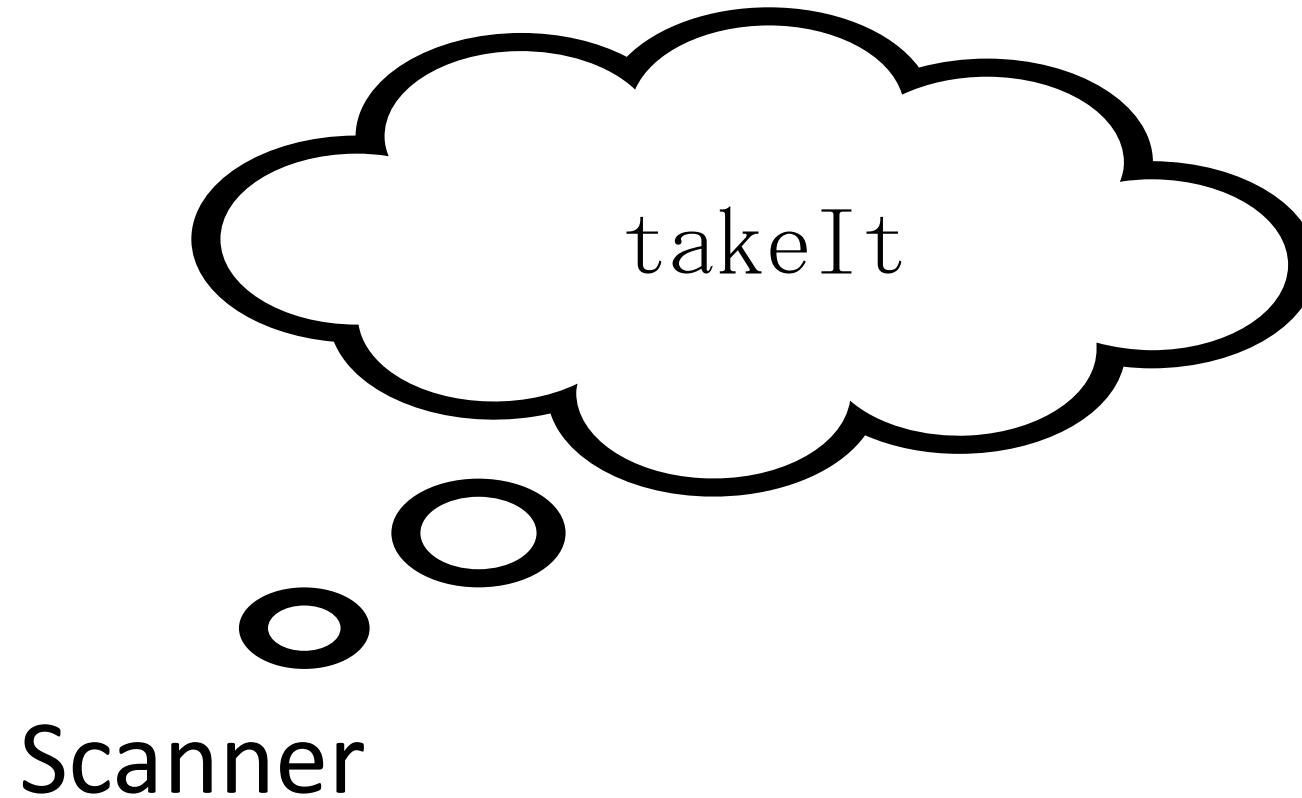
Current Letter: "s"

Current Text: "cla"

After:

Current text: "clas"

# Scanner Design



Current Letter: "s"  
Current Text: "clas"

After:

Current text: "class"



Turns out, it was not an identifier, but it was a reserved word. Create a Token with the TokenType.ClassToken, and return it.

Current Letter: " "  
(whitespace)

Current Text: "class"

After:

Current text: ""

Scanner



# Now let's look at the Parser

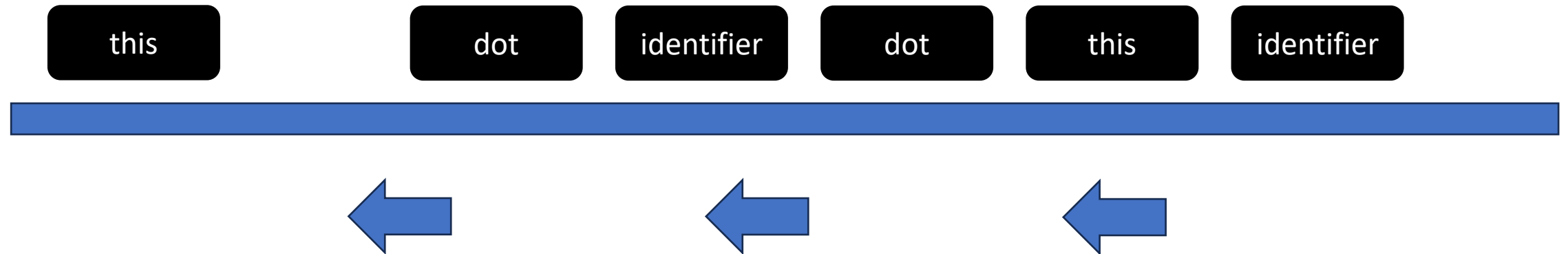
# Parser

- Consider a conveyer belt of Tokens provided by Scanner:

Current Token



Upcoming Tokens...





# Parser

- Assume we are in the Grammar:  $\text{Reference} ::= \text{id} \mid \text{this} \mid \text{Reference} . \text{id}$

Current Token



Upcoming Tokens...

this

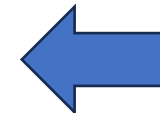
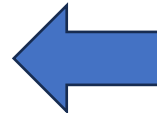
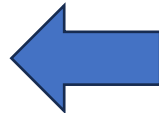
dot

identifier

dot

this

identifier



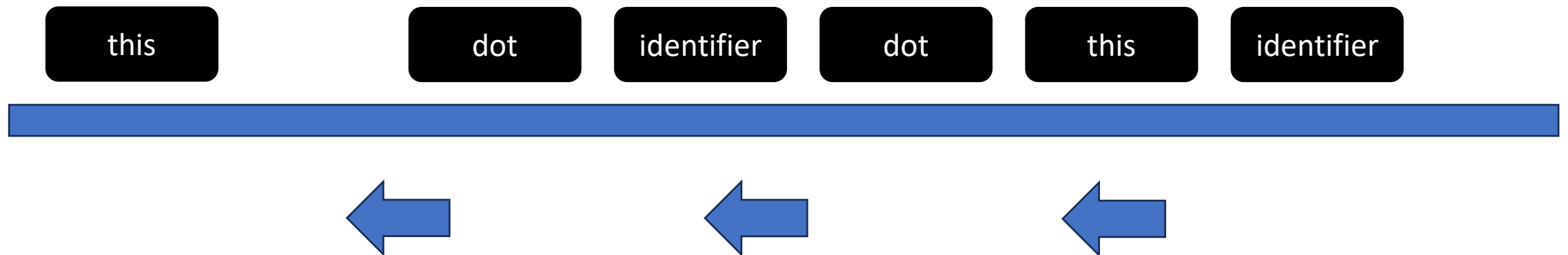
# Parser

- Assume we are in the Grammar:  $\text{Reference} ::= \text{id} \mid \text{this} \mid \text{Reference} . \text{id}$

Current Token




**Is this a valid sequence for Reference?**

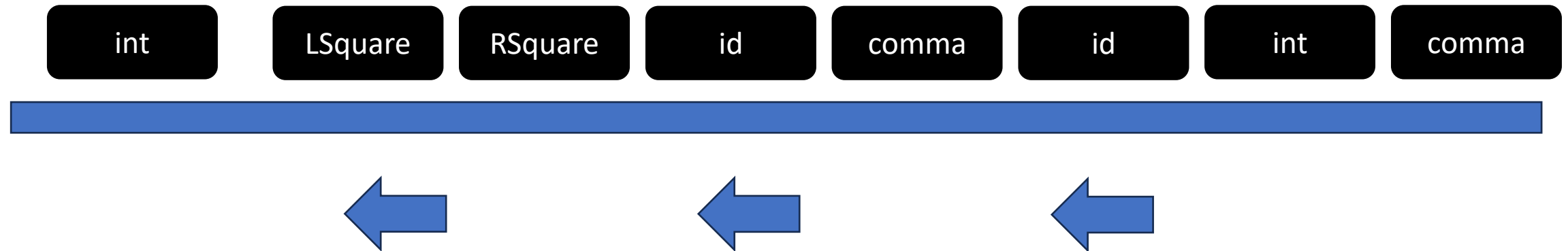


# What does an error look like?

- The parser may “want” a certain TokenType, but the scanner provided a different one!
- What does this look like, and how should the Compiler proceed?

# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type id} (, \text{Type id})^*$
-  Let's process this ParameterList, **Accept or Reject** the current token?

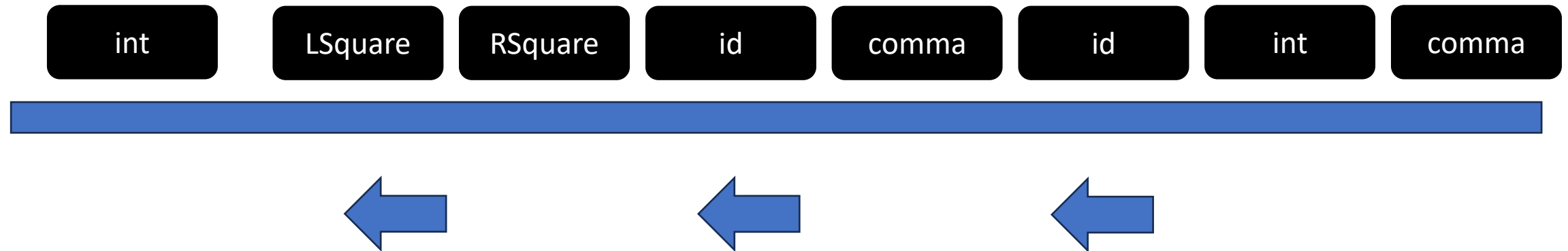


# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type id} (, \text{Type id})^*$



Accept or Reject?

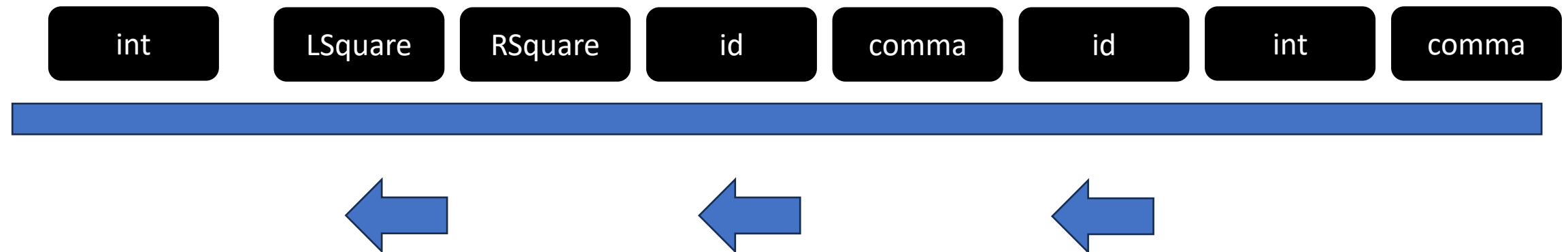


# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type id} (, \text{Type id})^*$



Accept or Reject?

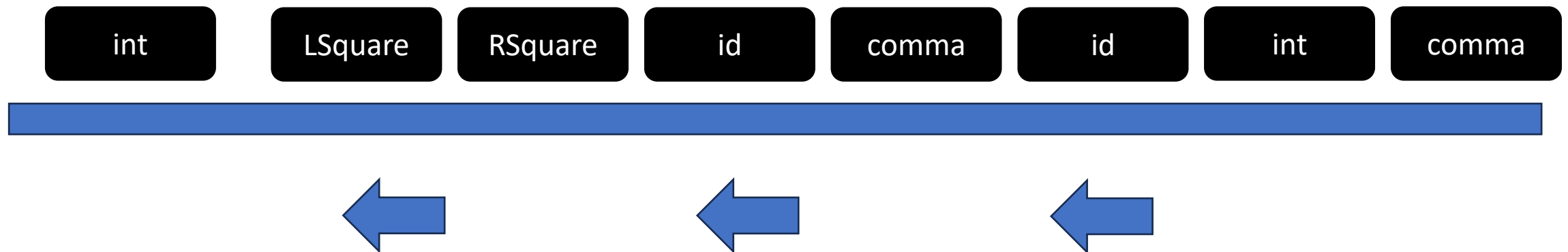


# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type } \textcolor{red}{\text{id}} (, \text{Type } \text{id})^*$



**Accept or Reject?**



# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type id} (, \text{Type id})^*$



**Accept or Reject?**

comma

id

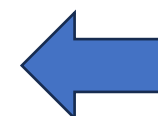
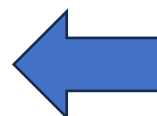
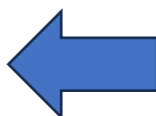
int

comma

boolean

LSquare

RSquare



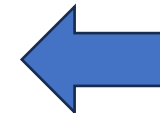
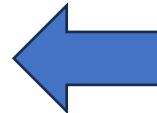
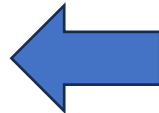


# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type id} (, \text{Type id})^*$



**Accept or Reject?**

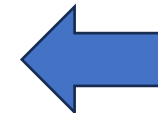
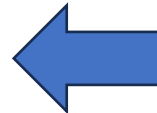
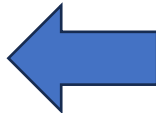


# Parser Errors

- Consider:  $\text{Type} ::= \text{int} \mid \text{boolean} \mid \text{id} \mid (\text{int} \mid \text{id})[]$
- $\text{ParameterList} ::= \text{Type id} (, \text{Type id})^*$



**Accept or Reject?**



# Error! Expected Identifier, but got IntToken

- Should we go to the end of the ParameterList?

# Error! Expected Identifier, but got IntToken

- Should we go to the end of the ParameterList?
- How would we know we ended the ParameterList?

# Error! Expected Identifier, but got IntToken

- Should we go to the end of the ParameterList?
- How would we know we ended the ParameterList?
- It would involve scanning until you find an RParen “)” because ParameterList always has Parens around it.
- But that sounds like it can get complicated quickly.

# Error! Expected Identifier, but got IntToken

- Should we go to the end of the ParameterList?
- How would we know we ended the ParameterList?
- It would involve scanning until you find an RParen “)” because ParameterList always has Parens around it.
- But that sounds like it can get complicated quickly.
- What if we only report the first error?
- Afterall, a syntax error means the program is non-functional, so why check the remaining program at all?

# Both answers are correct

- We will only grade based upon detecting the first error.
- The first solution sounds elegant right? But not really! Let's take a look at that Token stream again:

# Both answers are correct

- We will only grade based upon detecting the first error.
- The first solution sounds elegant right? But not really! Let's take a look at that Token stream again:

int

comma

boolean

LSquare

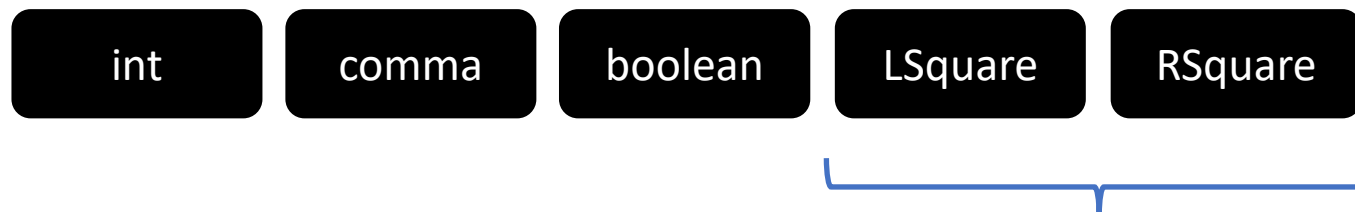
RSquare

- We know ParameterList is broken, but are there more errors we could have reported?



# Both answers are correct

- We will only grade based upon detecting the first error.
- The first solution sounds elegant right? But not really! Let's take a look at that Token stream again:



- Yes, Boolean arrays not allowed in miniJava

# Error Reporting - Wrapup

int

comma

boolean

LSquare

RSquare

- Yes, Boolean arrays not allowed in miniJava
- In a quest to report as many errors and be as descriptive as possible, still encountering problems reporting “every” error.
- Continuing to process ParameterList and NOT jumping to the end will likely result in the parser’s state machine not aligning well.

# When a Syntax error happens...

- Use exception handlers to “unwind” and get out of however deep you are in your parse methods.
- Finally, in PA1, report “Error” on ONE line (println) if any errors exist, THEN proceed to output errors that are relevant towards helping debug the input code.
- If there are no errors, output “Success” on ONE line (println)

End







