COMP 520 - Compilers

Lecture 06 – PA2 Intro

PA1 Due 1/31 11:59pm

- Make sure to use office hours resources

Quick Recap on LL(1)

- Consider the following: for every CFG rule, you do not know the sequence corresponding to that rule, but you DO know that the starters are disjoint.

- Is it LL(1)?

Quick Recap on LL(1)

- Consider the following: for every CFG rule, you do not know the sequence corresponding to that rule, but you DO know that the starters are disjoint.
- E.g., A ::= ?, B ::= ?, C ::= ?, and

$$\forall_{X,Y\in\{A,B,C\}}(\text{Starters}(X) \cap \text{Starters}(Y) = \emptyset) \vee (X = Y)$$

- Is it LL(1)?

Decisions.. decisions..

- The CFG rules are a decision, which is why it is important to know what terminals start a rule
- But inside the sequence, you also have decisions

- A ::= ba*Bb
- B ::= ac | ε
- Is this LL(1)?

Decisions.. decisions.. (2)

- A ::= ba*Bb
- B ::= ac | ε
- Is this LL(1)?
- Formally, let's check:

Predict(a) = Starters(a) = {a}

Predict(Bb) = Starters(Bb)⨁Followers(A)

= Starters(B)⨁Starters(b)

Cool property, where did Followers(A) disappear off to? Hint: Starters(Bb) not nullable.

Decisions.. decisions.. (3)

- A ::= ba*Bb
- B ::= ac | ε
- Is this LL(1)?
- Formally, let's check:

Predict(a) = Starters(a) = {a}

Predict(Bb) = Starters(Bb)⊕Followers(A)

= Starters(B)⊕Starters(b)

= {a,ε} ⊕ {b} = {a,b}

Not LL(1)

Sequences have decisions in them

- If we do not know where we are in the *sequence* when only looking at the current Token, then how can we claim we are LL(1)?

- A ::= ba*Bb
- B ::= ac | ε

Followers Example

- Consider:

S ::= A$

A ::= BDA | a

B ::= D | b

D ::= d | ε

First step: FL_0(A) = (ST($) ∪ ST($\varepsilon$)) \ {$\varepsilon$} = {$}

From the rule: $FL_0(A) = \left(\bigcup_{C \Rightarrow \alpha A \beta} \text{Starters}(\beta)\right) \setminus \{\varepsilon\}$

We see A is in the first and second rule.

Followers Example

S ::= A$
A ::= BDA | a
B ::= D | b
D ::= d | ε

First step: FL_0(A) = (ST($) ∪ ST($\varepsilon$)) \ {$\varepsilon$} = {$}

From the rule: $FL_0(A) = \left(\bigcup_{C\Rightarrow\alpha A\beta} \text{Starters}(\beta)\right) \setminus \{\varepsilon\}$

We see A is in the first and second rule.

Second iteration: FL_1(A) = FL_0(A) ∪ FL_0(A) = {$} ∪ {$} = {$}

From the rule: $FL_{i+1}(A) = FL_i(A) \cup \cdots \cup FL_i(C)$

- Where $C \Rightarrow \alpha A\beta$ and Nullable(β)

We find every rule C where A is to the left of Nullable sequences.

This would be Rule 2, because A is to the left of nothing.

Followers Example

S ::= A$
A ::= BDA | a
B ::= D | b
D ::= d | ε

First step: FL_0(A) = (ST($) ∪ ST($\varepsilon$)) \ {$\varepsilon$} = {$}

From the rule: $FL_0(A) = \left(\bigcup_{C\Rightarrow\alpha A\beta} \text{Starters}(\beta)\right) \setminus \{\varepsilon\}$

We see A is in the first and second rule.

Second iteration: FL_1(A) = FL_0(A) ∪ FL_0(A) = {$} ∪ {$} = {$}

From the rule: $FL_{i+1}(A) = FL_i(A) \cup \cdots \cup FL_i(C)$

- Where $C \Rightarrow \alpha A\beta$ and Nullable(β)

We find every rule C where A is to the left of Nullable sequences.

This would be Rule 2, because A is to the left of nothing.

Final Followers Rule

- Followers(A) = $\{t \mid S \Rightarrow^* \alpha A t \beta\} \cup \begin{cases} \{\varepsilon\} & \text{if } S \Rightarrow^* \alpha A \\ \{\} & \text{otherwise} \end{cases}$

- Don’t forget to union with $\{\varepsilon\}$ when $S \Rightarrow^* \alpha A$

Fixed Point

- Although the previous example was simple, some problems will require you to iterate until you hit a fixed point.

- Let’s look at the Nullable inductive definitions

Nullable Example

S ::= A$
A ::= BDA | a
B ::= D | b
D ::= d | ε

$N_0(S) = N(A\$) = N(A) \wedge N(\$) = N(A) \wedge \text{false} = \text{false}$

$N_0(A) = (N(B) \wedge N(D) \wedge N(A)) \vee N(a) = N(B) \wedge N(D) \wedge N(A)$

$N_0(B) = N(D) \vee N(b) = N(D) \vee \text{false} = N(D)$

$N_0(D) = N(d) \vee N(\varepsilon) = \text{false} \vee \text{true} = \text{true}$

	N_0	N_1	N_2	N_3
S	**F**			
A	**?**			
B	**?**			
D	**T**			

Nullable Example (2)

S ::= A$
A ::= BDA | a
B ::= D | b
D ::= d | ε

N_0(S) = N(A$) = N(A) ∧ N($) = N(A) ∧ false = false

N_0(A) = (N(B) ∧ N(D) ∧ N(A)) ∨ N(a) = N(B) ∧ N(D) ∧ N(A)

N_0(B) = N(D) ∨ N(b) = N(D) ∨ false = N(D)

N_0(D) = N(d) ∨ N(ε) = false ∨ true = true

N_1(B) = N_0(D) = true

N_1(A) = true ∧ true ∧ N(A) = N(A)

	N_0	N_1	N_2	N_3
S	F	**F**		
A	?	**?**		
B	?	**T**		
D	T	**T**		

If you see recursion…

- $N_i(A) = N_{i-1}(A)$

- A ::= BDA | a
- Try to rewrite this without recursion if possible, otherwise continue iterating

	N_0	N_1	N_2	N_3
S	F	F		
A	?	?		
B	?	T		
D	T	T		

If you see recursion… (2)

- $N_i(A) = N_{i-1}(A)$

- A ::= BDA | a
- Try to rewrite this without recursion

- A ::= (BD)*a
- N(A) = true ∧ N(a) = false

	N_0	N_1	N_2	N_3
S	F	F	**F**	
A	?	?	**F**	
B	?	T	**T**	
D	T	T	**T**	

Nullable Example (3)

	$\mathbf{N_0}$	$\mathbf{N_1}$	$\mathbf{N_2}$	$\mathbf{N_3}$
S	F	F	F	F
A	?	?	F	F
B	?	T	T	T
D	T	T	T	T

Fixed Point
at $\mathbf{N_2} \rightarrow \mathbf{N_3}$

From your feedback:

- I won't ask for Predict sets in midterms/finals, but Nullable, Starters, and Followers is fair game
- Instead, Predict can be practiced in the first WA (to be released on Thursday)

PA2 – Intro

ASTs and Operator Precedence

Recap of PA1

- Getting started may be difficult
- Syntax Analysis can be somewhat difficult, even for miniJava (imagine C++)

- It still feels like we are far from having a fully functional compiler … right?

PA1

- You will be happy to know, you actually accomplished quite a bit of the compiler in PA1

- PA2 can be described as two things:

1. “Package syntax into data structures”
2. “Operator precedence”

PA2 Expectations

- Learn about Abstract Syntax Trees
- Package the miniJava syntax into AST data structures
- Return one AST that encapsulates the entire program being compiled

- Data must be organized in a consistent manner, e.g., in a "Method List" data structure, methods appear in order as they appear in the source code

PA2 Expectations

- Learn about Abstract Syntax Trees
- Package the miniJava syntax into AST data structures
- Return one AST that encapsulates the entire program being compiled

- Data must be organized in a consistent manner, e.g., in a “Method List” data structure, methods appear in order as they appear in the source code

Abstract Syntax Trees

Abstract Syntax Trees

- Recall the graph exercise for the CFG in Lec02, we will try to build a graph like that except for the input source file

- Only syntactically valid programs have an AST

- Building an AST is *easier* with an EBNF grammar rather than the original recursive CFG,

  Where *easier* in this context just means less to write down, although some problems may also arise when generating the AST grammar.

Abstract Syntax Trees

- Recall the graph exercise for the CFG in Lec02, we will try to build a graph like that except for the input source file

- Only syntactically valid programs have an AST

- Building an AST is *easier* with an EBNF grammar rather than the original recursive CFG,

  Where *easier* in this context just means less to write down, although some problems may also arise when generating the AST grammar.

AST Example

- Consider the CFG:
- S ::= E $
- E ::= E Op T | T
- T ::= (E) | num
- Op ::= + | *

What is the EBNF of this?

CFG -> EBNF

CFG

- S ::= E $
- E ::= E Op T | T
- T ::= (E) | num
- Op ::= + | *

EBNF

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num
- Op ::= + | *

EBNF AST

- Now that we have the grammar in the much easier EBNF, we can construct an AST.

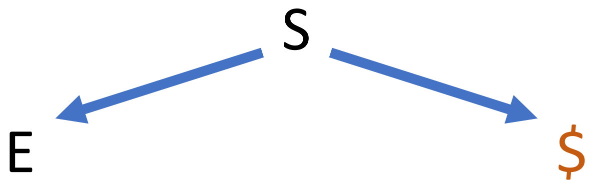
- An AST is a graph that describes the structure of your **input source code**.

S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

AST Construction

Let's construct the syntax tree for 2 + (3 * 4) $

Apply: S ::= E $, Left: E=2 + (3 * 4), Right: $

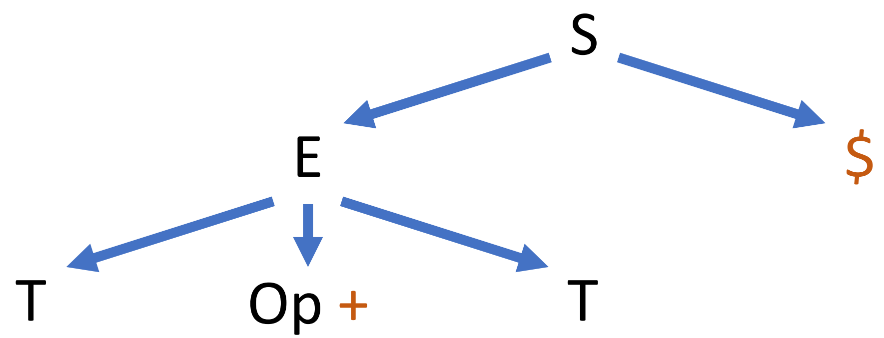


S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

AST Construction

Let's construct the syntax tree for 2 + (3 * 4) $

Apply: E ::= T Op T, Left: E=2, Op=+, Right: T=(3 * 4)

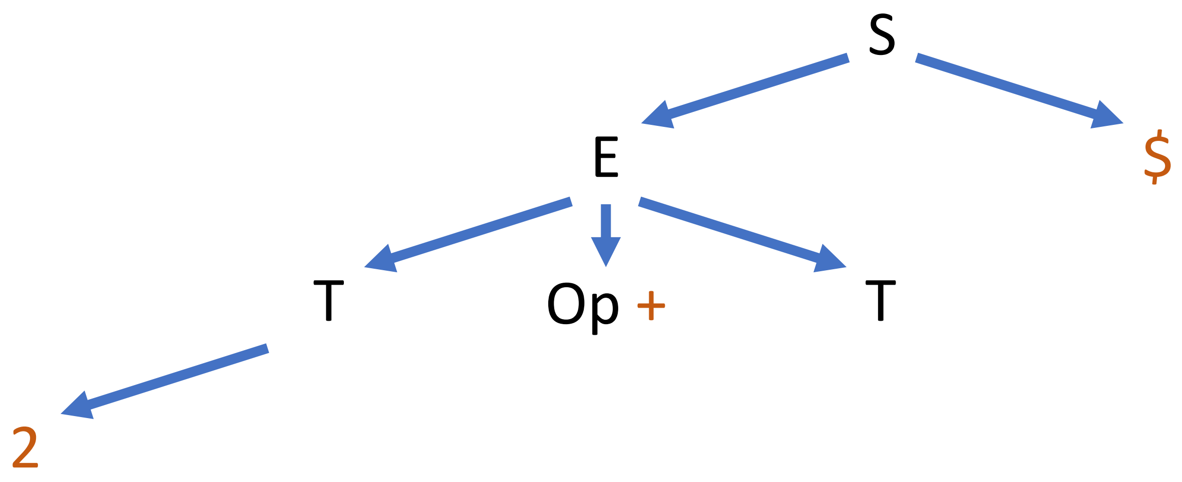


S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

AST Construction

Let's construct the syntax tree for 2 + (3 * 4) $

Apply: T ::= num

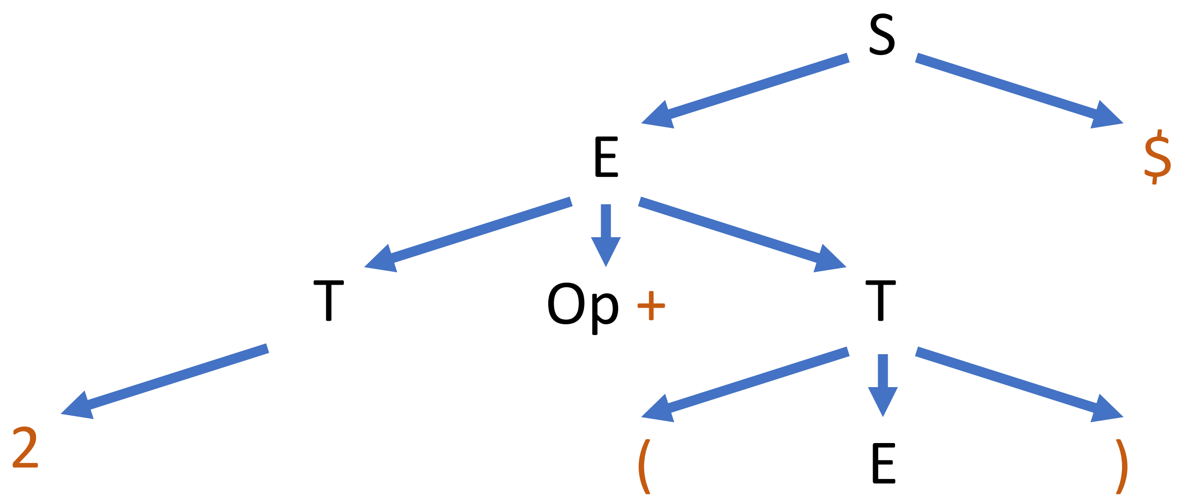


S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

AST Construction

Let's construct the syntax tree for 2 + (3 * 4) $

Apply: T ::= (E), Left: (, Middle: E=3*4, Right:)

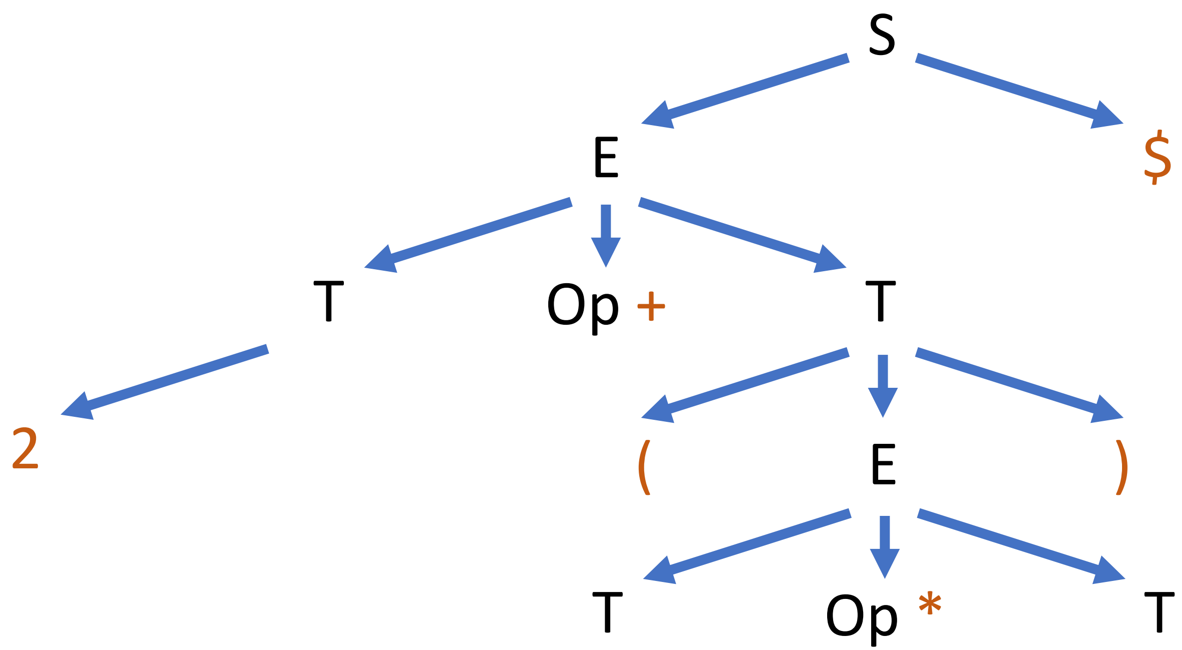


S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

AST Construction

Let's construct the syntax tree for 2 + (3 * 4) $

Apply: E ::= T Op T, Left: T=3, Middle: Op=*, Right: T=4

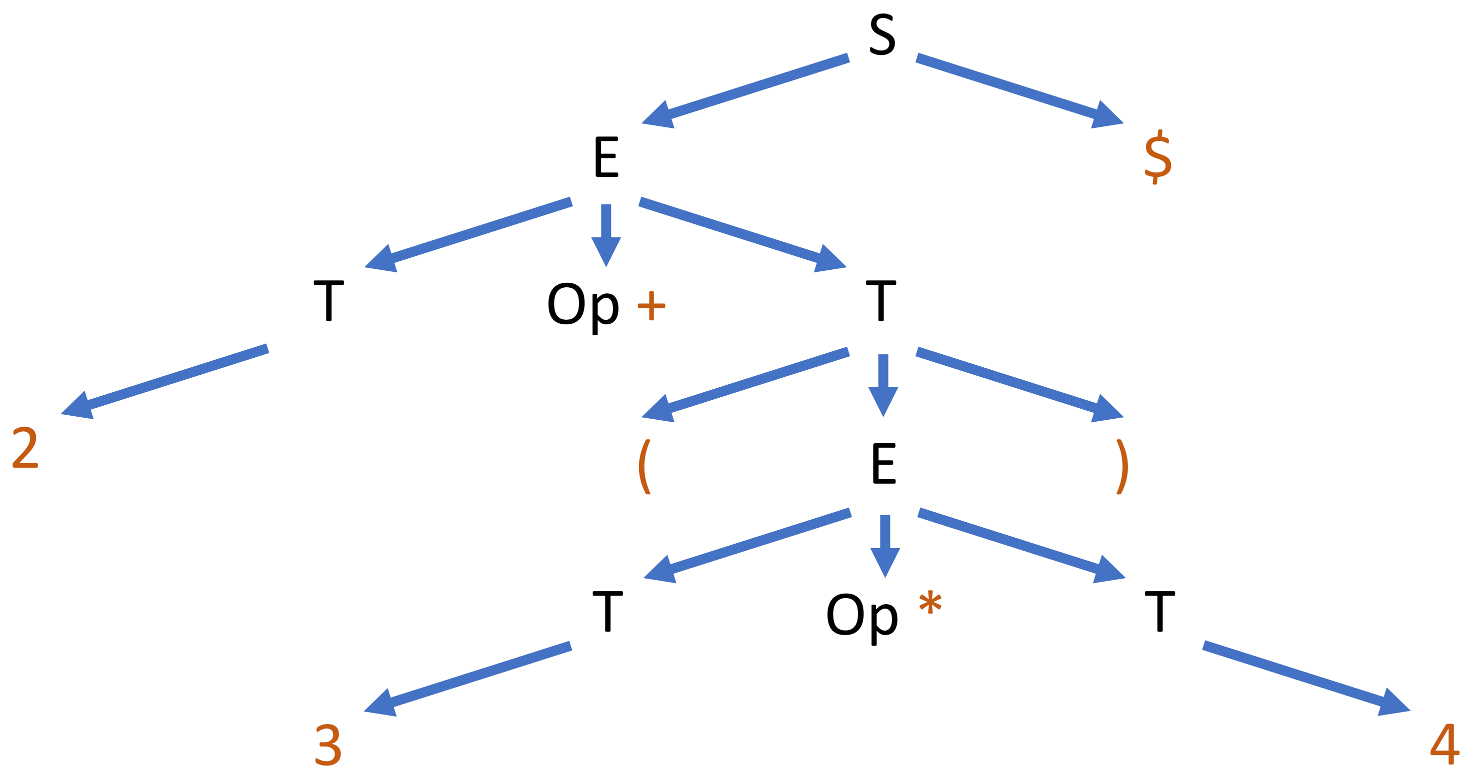


S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

AST Construction

Let's construct the syntax tree for 2 + (3 * 4) $

Apply: T ::= num



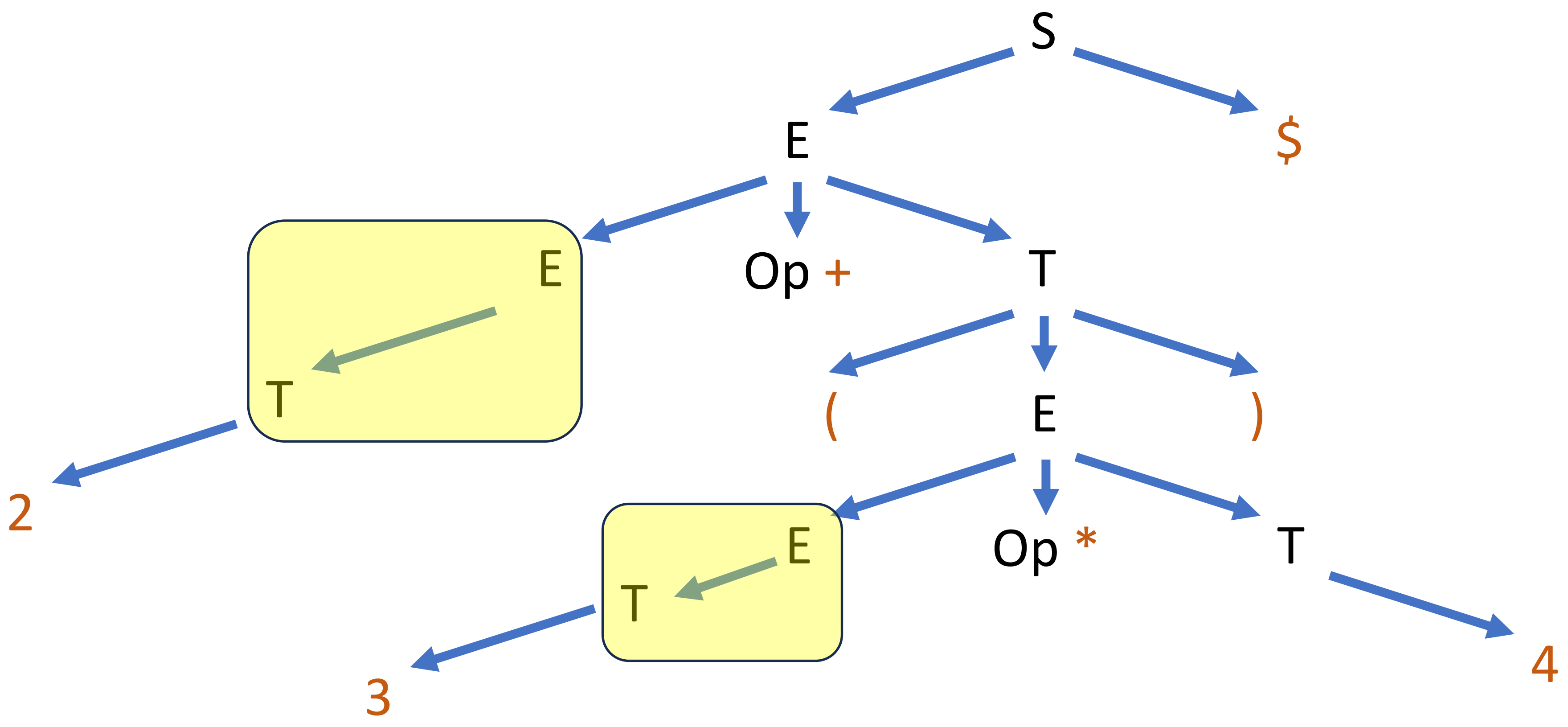
What if we used the CFG?

S ::= E $
E ::= E Op T | T
T ::= (E) | num
Op ::= + | *

Without EBNF

Let's construct the syntax tree for 2 + (3 * 4) $

Tree is lengthier, and can get quite messy



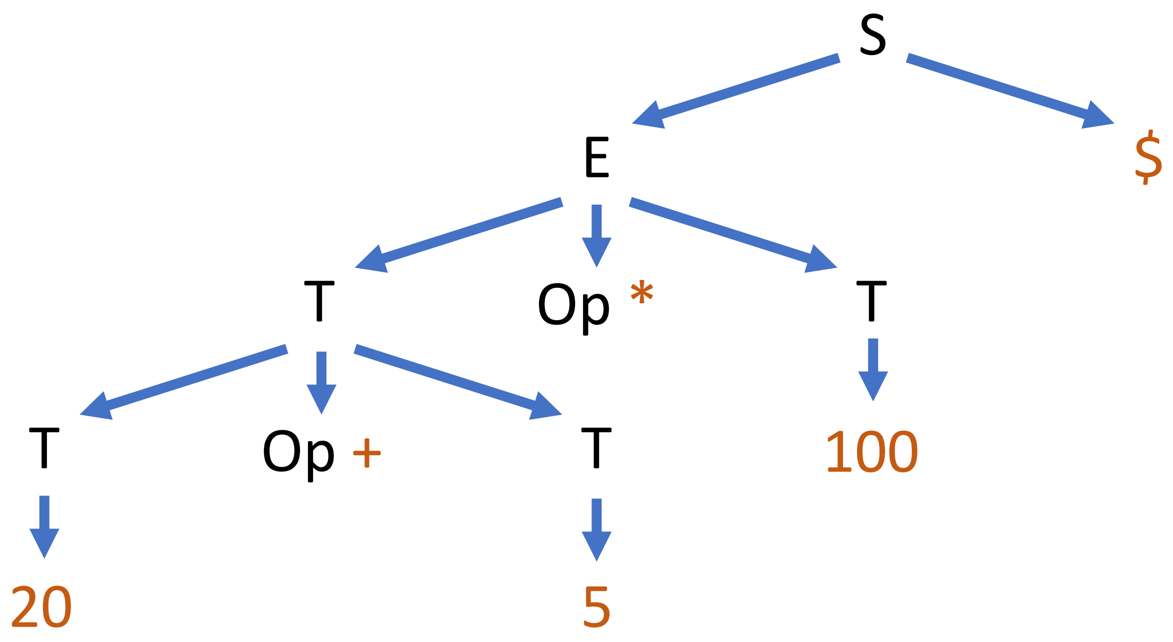
A quick look at math in expressions

More on this in Thursday’s lecture (Lec07)

S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

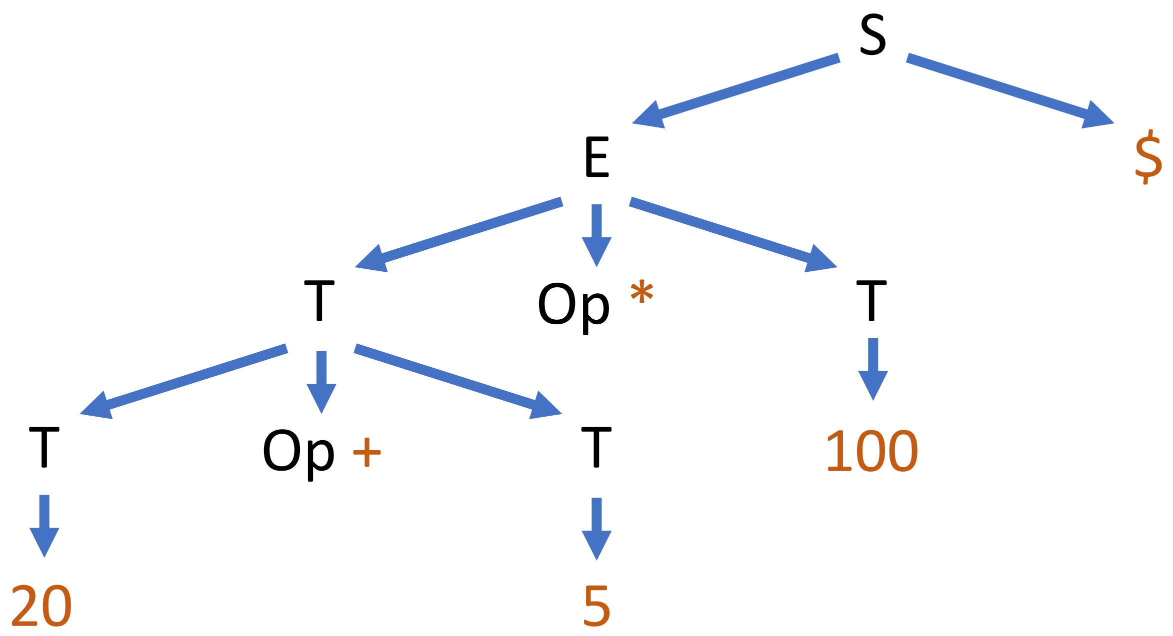
Any problems with this AST?

Input: 20 + 5 * 100 $



S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

Any problems with this AST?
Input: 20 + 5 * 100 $

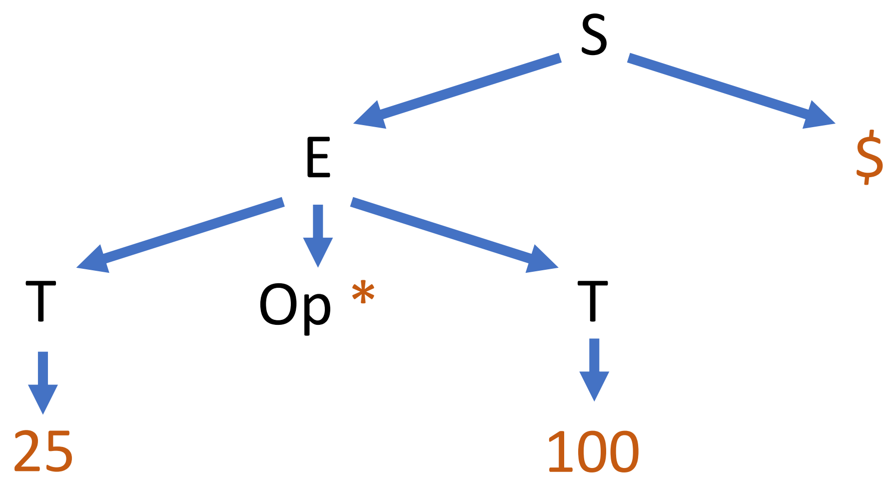


Emulate Execution
20 + 5 = 25

S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

Any problems with this AST?

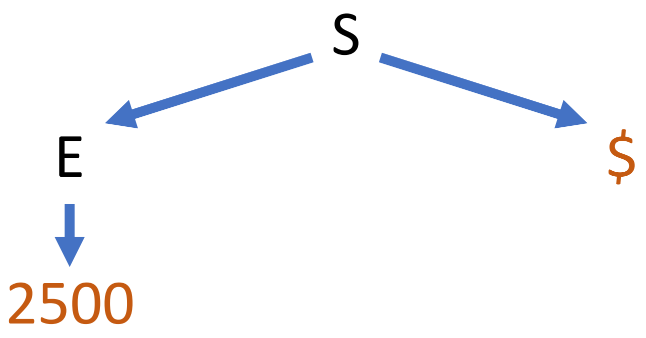
Input: 20 + 5 * 100 $



Emulate Execution
20 + 5 = 25
25 * 100 = 2500

S ::= E $
E ::= T (Op T)*
T ::= (E) | num
Op ::= + | *

Any problems with this AST?
Input: 20 + 5 * 100 $



Emulate Execution
20 + 5 = 25
25 * 100 = 2500

But that isn't correct!

- 20 + 5 * 100 is not 2500
- Precedence rules must be enforced for the correct AST to be generated.

- This can be tricky, but we can modify our grammar to make this quite easy (next lecture)

Abstractness of ASTs

What types of Expressions do we have?

- Consider: UnaryExpr, BinaryExpr, CallExpr, IxExpr, RefExpr, LiteralExpr, NewArrayExpr, NewObjectExpr

- All of these are an “Expression”
- So this rule: Expression Op Expression ≡ BinaryExpr
- But each of those Expressions can be any other type of Expression.

What types of Expressions do we have?

- Consider: UnaryExpr, BinaryExpr, CallExpr, IxExpr, RefExpr, LiteralExpr, NewArrayExpr, NewObjectExpr
- So this rule: Expression Op Expression ≡ BinaryExpr

- Is this a syntactically valid expression?

-3 + new A()

What types of Expressions do we have?

- Consider: UnaryExpr, BinaryExpr, CallExpr, IxExpr, RefExpr, LiteralExpr, NewArrayExpr, NewObjectExpr
- So this rule: Expression Op Expression ≡ BinaryExpr

- Is this a syntactically valid expression?

-3 + new A()

- Yes, but the types do not match, however for PA2, perfectly fine

What types of Expressions do we have?

- Is this a syntactically valid statement?

```
boolean A = -3 + new A();
```

What types of Expressions do we have?

- Is this a syntactically valid statement?

```
boolean A = -3 + new A();
```

- Yes, but the types do not match, and A makes no sense in its context.
- Still perfectly fine for PA2

Definitions for ASTs

- Consider WhileStmt ::= while (Expression) Statement

- We want to capture this in a data structure, so we create the class **WhileStmt** which extends **Statement**

Definitions for ASTs

- Consider WhileStmt ::= while (Expression) Statement

- We want to capture this in a data structure, so we create the class **WhileStmt** which extends **Statement**

if(currentToken.getType() == TokenType.While) {

- accept(while); accept(‘(‘);
- Expression e = parseExpression();
- accept(‘)’);
- Statement s = parseStatement();

Definitions for ASTs

- Consider WhileStmt ::= while (Expression) Statement
- We want to capture this in a data structure, so we create the class **WhileStmt** which extends **Statement**

if(currentToken.getType() == TokenType.While) {

- accept(while); accept(‘(‘);
- Expression e = parseExpression();
- accept(‘)’);
- Statement s = parseStatement();
- return new WhileStmt(e, s);

AST Implementations

- The class definitions for ASTs are quite mundane and likely what you expect them to be.
- E.g., **TypeDenoter** is the abstract type for "Type" and $\mathrm{parseType}$ can return **ArrayType**, **BaseType**, **ClassType**, each of which extend **TypeDenoter**

AST Implementations

- The class definitions for ASTs are quite mundane and likely what you expect them to be.
- E.g., **TypeDenoter** is the abstract type for "Type" and parseType can return **ArrayType**, **BaseType**, **ClassType**, each of which extend **TypeDenoter**

- As such, all ASTs are already implemented and available on the course website.

PA2 Restrictions

- You must use the AST implementations available on the course website.
- The autograder checks to make sure your AST is constructed correctly and in the proper order.

Quick note on AST Grammars

AST Grammars

Consider the grammar:

- S ::= E
- E ::= T (Op T)*
- T ::= (E) | num

- We want to parse Expressions, so create a rule:
- S ::= E
- For simplicity, add the $ terminal
- S ::= E $
  - (See augmented grammars, worth a google or check the textbook)

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

- First let's denote "E" as an "Expression" as that is the symbol in our start state
- What are the types of expressions we can encounter?

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

- First let's denote "E" as an "Expression" as that is the symbol in our start state

- What are the types of expressions we can encounter?
- Locate all instances of E and any non-terminal it encompasses

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Locate all instances of E and any non-terminal it encompasses

- T, and T Op T

Non-terminal T, so we also have

- (E) and num

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Looks like we have three types of expressions:

Just "T", so:

$$T \begin{cases} (\,E\,) \\ \text{num} \end{cases}$$

T Op T

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

$$T \begin{cases} (\ E\ ) \\ \text{num} \end{cases}$$

T Op T

(E) is just an Expression that is later resolved, so this isn't unique.

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Thus, we have two types of expressions:

T Op T

num

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Thus, we have two types of expressions:

Define them!

T Op T	BinExpr
num	LiteralExpr

AST Grammars

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Generate AST Grammars:

- Expr ::= Expr Op Expr (BinExpr)
  | num (NumExpr)

Each option has its own AST definition, where options have an "is a" relationship with the parent type.

"NumExpr" is a "Expr"

AST creation is necessary but…

- Generating the theory for what should be in the AST grammars? **Exciting**, even if it is just "find the options."

- Writing the code for every single AST object with the proper "is a" relationship? Well…

AST creation is necessary but…

- Generating the theory for what should be in the AST grammars? Exciting, even if it is just "find the options."

- Writing the code for every single AST object with the proper "is a" relationship? Well…

- We're just going to give you the code for AST objects

AST Layout from PA2 Instructions

Note: What is provided on the right is subject to clarification updates.

Always check Piazza for updates, and grab the latest PA2 instructions from the course website.

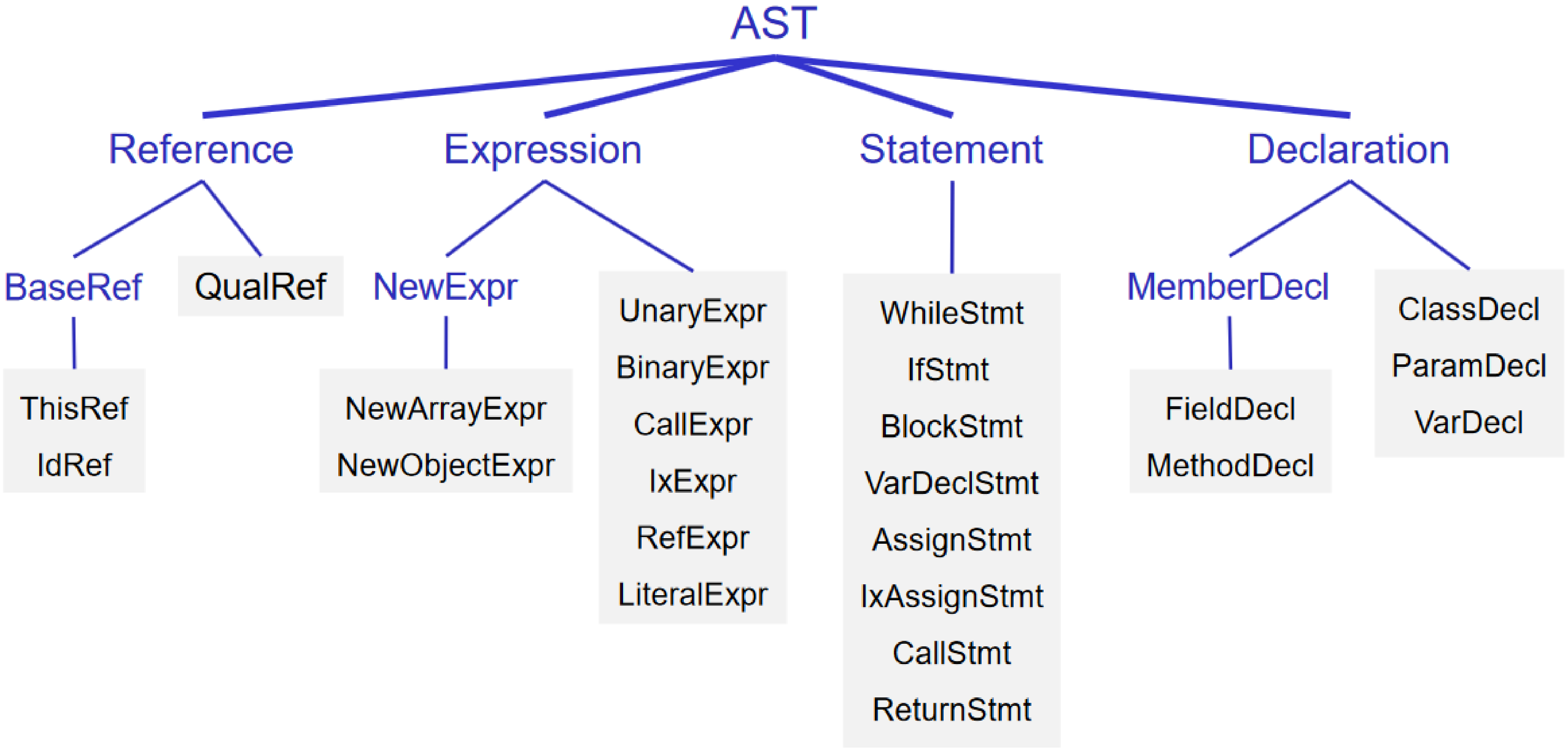
Program	::=	ClassDeclaration* eot	Package
ClassDeclaration	::=	class id { (FieldDeclaration|MethodDeclaration)* }	ClassDecl
FieldDeclaration	::=	Visibility Access Type id ;	FieldDecl
MethodDeclaration	::=	Visibility Access (Type|void) id (ParameterList?) { Statement* }	MethodDecl
Visibility	::=	(public|private)?	*n/a*
Access	::=	(static)?	*n/a*
Type	::=	int | boolean | id | (int|id)[]	TypeDenoter
ParameterList	::=	Type id (,Type id)*	ParameterDeclList
ArgumentList	::=	Expression (,Expression)*	ExprList
Reference	::=	id | this | Reference . id	IdRef | ThisRef | QualRef
Statement	::=	{ Statement* }	BlockStmt
	|	Type id = Expression ;	VarDeclStmt
	|	Reference = Expression ;	AssignStmt
	|	Reference[Expression] = Expression ;	IxAssignStmt
	|	Reference (ArgumentList?) ;	CallStmt
	|	return (Expression)? ;	ReturnStmt
	|	if (Expression) Statement (else Statement)?	IfStmt
	|	while (Expression) Statement	WhileStmt
Expression	::=	Reference	RefExpr
	|	Reference [Expression]	IxExpr
	|	Reference (ArgumentList?)	CallExpr
	|	unop Expression	UnaryExpr
	|	Expression binop Expression	BinaryExpr
	|	(Expression)	Expression
	|	num	LiteralExpr (*IntLiteral*)
	|	true | false	LiteralExpr (*BooleanLiteral*)
	|	new id()	NewObjectExpr
	|	new (int|id) [Expression]	NewArrayExpr

PA2 Overview

Step 1: Import

- Create a package called miniJava.AbstractSyntaxTrees
- Download the zip file on the course website, and import all source files into the package

Step 2: Study AST Implementations



Step 2: AST Implementations

- Package AST has an add method, and accepts ClassDecl to build a list of classes.

- Your parse method should return the Package AST.

Step 3: SourcePosition object

- See PA2 instructions for more details
- When debugging your code, you can enable source positions, but if you are not tracking source positions for your Tokens, then pass null whenever an AST requires a SourcePosition object.

- If you already are tracking positions, then package that data into the SourcePosition object as outlined in the instructions.

SourcePosition for ASTs

- Syntax doesn't occur at a single location, so what is a good way to implement SourcePosition?

- Up to you, but we recommend SourcePosition being overloaded with two constructors, one with just a line/col number, and another with a StartToken and EndToken, and the toString output would show the range over which lines the current syntax spans.
- Recall: This is a PA5 extra credit item

PA2 Overview

- Go through each of your parse methods and return the AST associated with that syntax.

- For example, parseExpression returns the generic Expression AST, but if the current token is "true|false", then it returns:

```
new LiteralExpr(new BooleanLiteral( theToken ), theToken.position )
```

- Where LiteralExpr "is an" Expression

Compiler.java Changes

- As before, output "Error" on its own line (println) if there is a syntax error, then any meaningful error messages you like
- If there are no errors, then…

```java
// Call the parser's parse function
AST programAST = parser.parse();
```

```java
// If there are no errors, output our AST
ASTDisplay display = new ASTDisplay();
display.showTree(programAST);
```

When no errors…

- If there are no errors, ensure there is no other output other than the one generated from display.showTree

Debugging

- If your compiler is not passing a test, download "Gradescope Tests" on the course website for PA2, then find the associated test.
- Note: "pass119.java" is the input source file, and "pass119.java.out" is the AST display that should be generated.
- If there is an error, find the difference in your display versus the .out file

Debugging (2)

- If you need to know where in the source code something went wrong and you have implemented SourcePosition, then go to ASTDisplay.java and set the "showPosition" variable to true.

- NOTE: Only submit your assignment with showPosition set to false, otherwise the autograder will be unable to check your Compiler's output for valid input files.

Example Output

class id {}

```
======= AST Display =========================
Package
  ClassDeclList [1]
  . ClassDecl
  .   "id" classname
  .   FieldDeclList [0]
  .   MethodDeclList [0]
=============================================
```

Example Output

```
class PA2sample {
    public boolean c;
    public static void main(String[] args) {
        if( true )
            this.b[3] = 1 + 2 * x;
    }
}
```

```
======= AST Display =========================
Package
  ClassDeclList [1]
  . ClassDecl
  .   "PA2sample" classname
  .   FieldDeclList [1]
  .   . (public) FieldDecl
  .   .   BOOLEAN BaseType
  .   .   "c" fieldname
  .   MethodDeclList [1]
  .   . (public static) MethodDecl
  .   .   VOID BaseType
  .   .   "main" methodname
  .   .   ParameterDeclList [1]
  .   .   . ParameterDecl
  .   .   .   ArrayType
  .   .   .     ClassType
  .   .   .       "String" Identifier
  .   .   .   "args"parametername
  .   .   StmtList [1]
  .   .   . IfStmt
  .   .   .   LiteralExpr
  .   .   .     "true" BooleanLiteral
  .   .   .   IxAssignStmt
  .   .   .     QualRef
  .   .   .       "b" Identifier
  .   .   .       ThisRef
  .   .   .     LiteralExpr
  .   .   .       "3" IntLiteral
  .   .   .     BinaryExpr
  .   .   .       "+" Operator
  .   .   .         LiteralExpr
  .   .   .           "1" IntLiteral
  .   .   .         BinaryExpr
  .   .   .           "*" Operator
  .   .   .             LiteralExpr
  .   .   .               "2" IntLiteral
  .   .   .             RefExpr
  .   .   .               IdRef
  .   .   .                 "x" Identifier
=============================================
```

Parse Example (assume no precedence)

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Can anyone give me the parse method for parseS() if it was PA1?

Then, we will add ASTs!

Parse Example

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

```
void parseS() {
    parseE();
    accept(EOT);
}
```

Generate AST Grammars:

- Expr ::= Expr Op Expr (BinExpr)

  | num (NumExpr)

Parse Example

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Generate AST Grammars:

- Expr ::= Expr Op Expr (BinExpr)
  | num (NumExpr)

```
Expr parseS() {
    Expr e = parseE();
    accept(EOT);
    return e;
}
```

Parse Example

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Generate AST Grammars:

- Expr ::= Expr Op Expr (BinExpr)
  | num (NumExpr)

```
Expr parseS() {
    Expr e = parseE();
    accept(EOT);
    return e;
}

Expr parseE() {
    Expr e = parseT();
    while(curToken==Operator) {
        OpToken op = new OpToken(curToken);
        accept(Operator);
        Expr rhs = parseT();
        e = new BinExpr(e,op,rhs);
    }
    return e;
}
```

Parse Example

Consider the grammar:

- S ::= E $
- E ::= T (Op T)*
- T ::= (E) | num

Generate AST Grammars:

- Expr ::= Expr Op Expr (BinExpr)
  | num (NumExpr)

```
Expr parseS() {
    Expr e = parseE();
    accept(EOT);
    return e;
}

Expr parseE() {
    Expr e = parseT();
    while(curToken==Operator) {
        OpToken op = new OpToken(curToken);
        accept(Operator);
        Expr rhs = parseT();
        e = new BinExpr(e,op,rhs);
    }
    return e;
}

Expr parseT() {
    if(curToken==LPAREN) {
        accept(LPAREN);
        Expr eInner = parseE();
        accept(RPAREN);
        return eInner;
    } else if(...==NUM) {
        NumExpr e = new NumExpr(curToken);
        accept(NUM);
        return e;
    }
    ... error
}
```

Recommendations

- Work on operator precedence last, because everything else in the Parser is only slightly modified
- (Your implementation may require larger modifications, but hopefully nothing crazy)

- We will make operator precedence very easy in Thursday's lecture

End