Compilers

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today’s Lecture

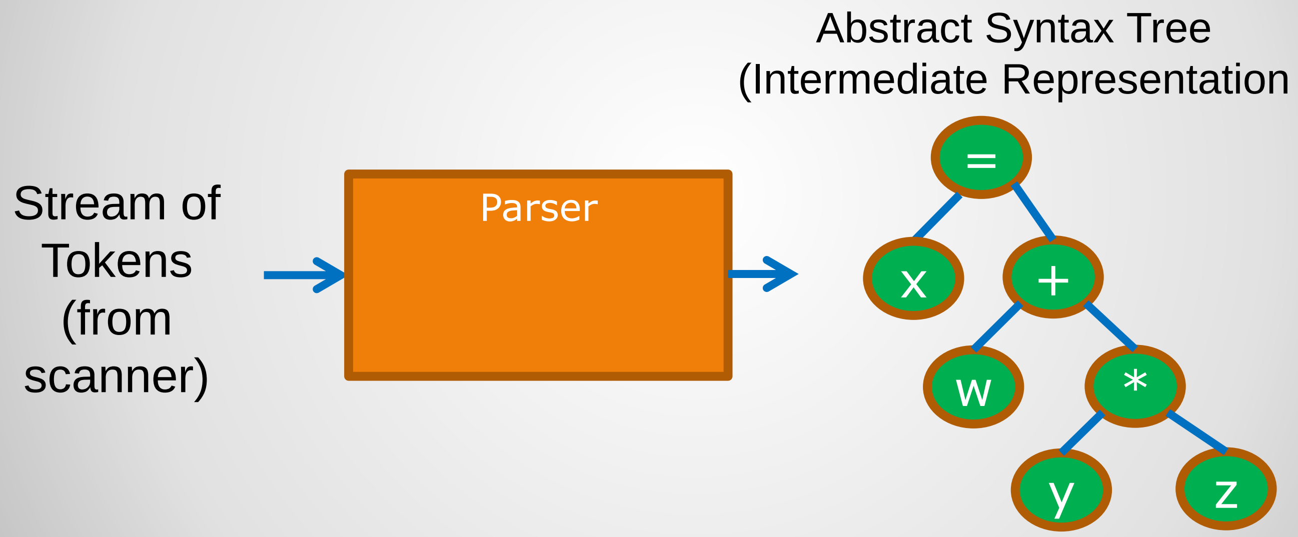
- Abstract Syntax Tree

Abstract Syntax Tree

- An abstract syntax tree (AST) is an intermediate representation of the program.
- An AST leaves out details that would appear in a normal parse tree created by a grammar (this is where the term abstract comes from).
- A parse tree created by a grammar would have nodes corresponding to all nonterminals used in substitutions that were applied during parsing (an AST would not).

Generating the AST

- The parser is responsible for generating the abstract syntax tree.
- As the program is being parsed it builds the AST.
- If parsing is successful, the parser produces an AST as its output.



Stream of Tokens (from scanner)

Parser

Abstract Syntax Tree
(Intermediate Representation

=

x

+

w

*

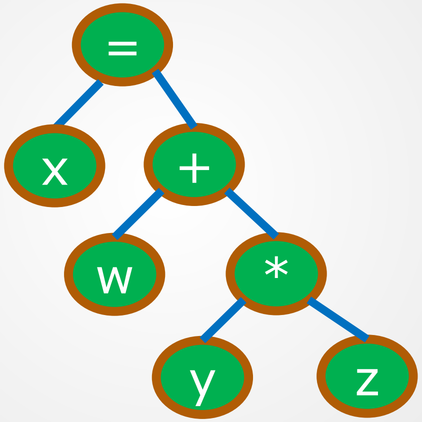
y

z

Generating the AST

- AST for x=w+y*z:

Abstract Syntax Tree
(Intermediate Representation



Leaf nodes are for variables (and integer literals) in the AST

w,x,y,z are the variables in this example

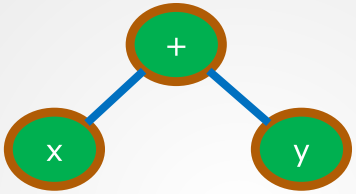
Interior nodes are for operators in the AST

=, +, * are the operators in this example

AST Operator Node

- The children of an operator node are its operands.
- For example, the + operator has two operands, so it has two children.

Operator Node (+)



x an y are children of + because they are its operands

AST and Parse Tree Example

- Sample grammar for addition:

E → F E'

E' → + F E'

E' → ε

F → id

F → intliteral

What is the parse tree for the following: x (x is just one id)

- Sample grammar for addition:

E → F E'

E' → + F E'

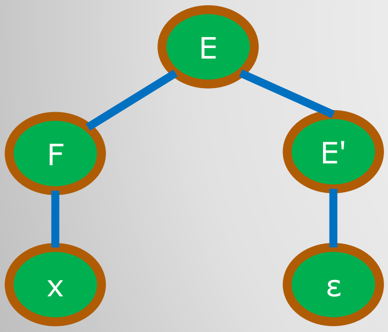
E' → ε

F → id

F → intliteral

What is the abstract syntax tree for the following: x (x is just one id)

Parse Tree



AST and Parse Tree Example

AST and Parse Tree Example

- Sample grammar for addition:

E → F E'

E' → + F E'

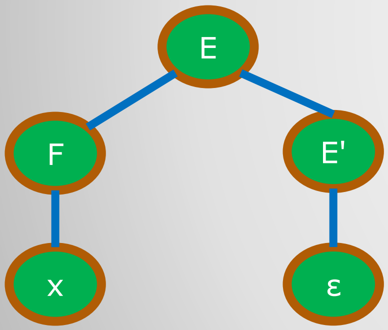
E' → ε

F → id

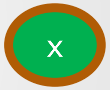
F → intliteral

What is the abstract syntax tree for the following: x (x is just one id)

Parse Tree



Abstract Syntax Tree



In contrast to the parse tree, the AST does not contain nodes for all nonterminals.

Example AST Nodes

- Sample grammar for addition:

E → F E'

E' → + F E'

E' → ε

F → id

F → intliteral

What is the <u>parse tree</u> for the following: x + y

- Sample grammar for addition:

E → F E'

E' → + F E'

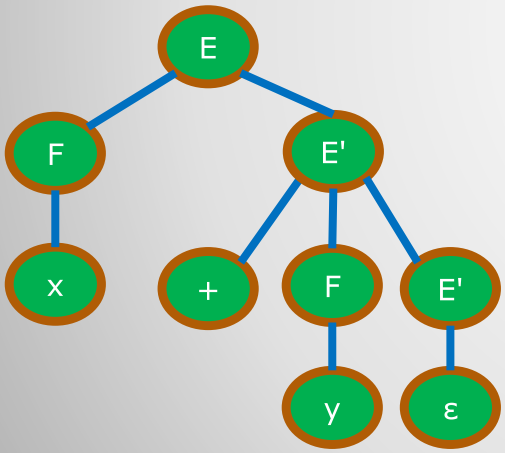
E' → ε

F → id

F → intliteral

What is the abstract syntax tree for the following: x + y

Parse Tree



Example AST Nodes

Example AST Nodes

- Sample grammar for addition:

E → F E'

E' → + F E'

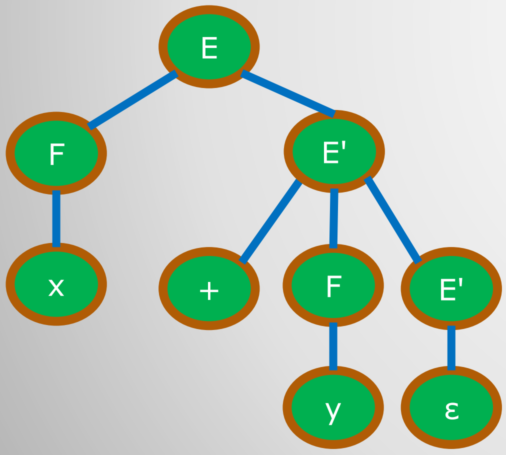
E' → ε

F → id

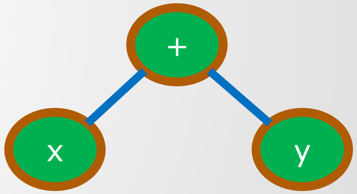
F → intliteral

Parse tree and abstract syntax tree for: x + y

Parse Tree



Abstract Syntax Tree



Interior nodes are for operators in the AST

Most of the nodes from the parse tree do not appear in the AST

Example AST Nodes

- Sample grammar for addition:

E → F E'

E' → + F E'

E' → ε

F → id

F → intliteral

What is the parse tree for the following: x + y + z

Example AST Nodes

- Sample grammar for addition:

E → F E'

E' → + F E'

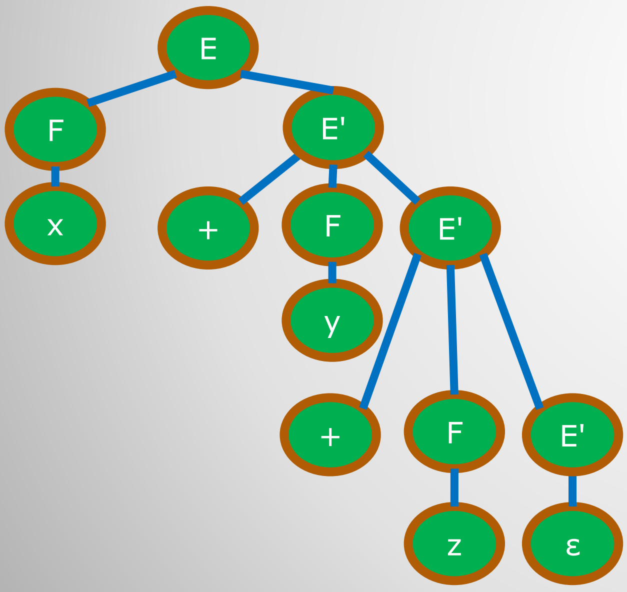
E' → ε

F → id

F → intliteral

What is the <u>abstract syntax tree</u> for the following: x + y + z

Parse Tree



Example AST Nodes

- Sample grammar for addition:

E → F E'

E' → + F E'

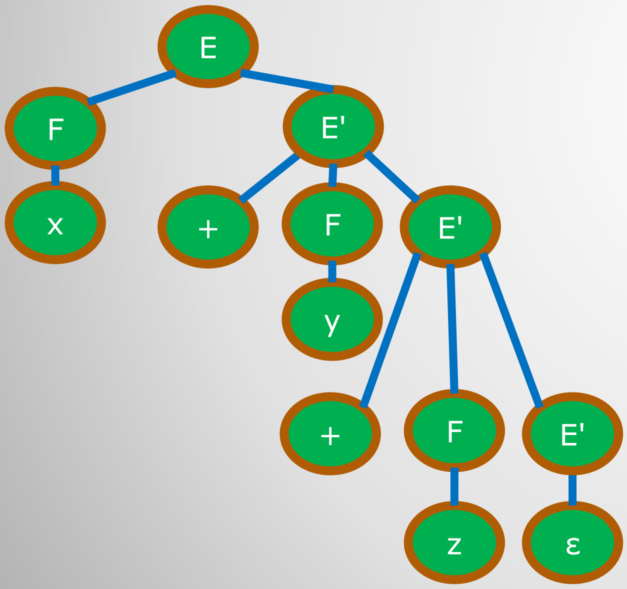
E' → ε

F → id

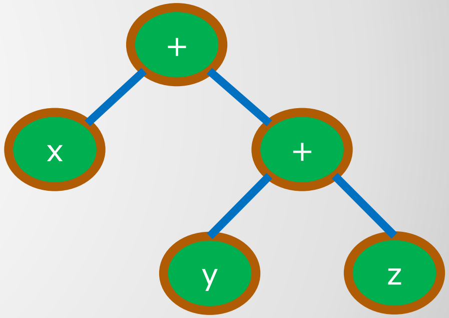
F → intliteral

Parse tree and abstract syntax tree for: x + y

Parse Tree



Abstract Syntax Tree



AST Node Classes

- Now on to AST node classes…

AST Node Base Classes

- All nodes can be derived directly or indirectly from the following class:

```
abstract class ASTBase {
}
```

- The expression related classes can be from the following class:

```
abstract class Expr extends ASTBase{
}
```

- The statement classes can be derived from the following class:

```
abstract class Stmt extends ASTBase {
}
```

AST Node Id

- The id class should store the variable's name.
- An id is an expression so it should inherit from Expr.

```
class Id extends Expr {
    Declare String name

    Id Constructor (String name) {
        Set this.name to name
    }
}
```

- An id can be the only item on the right side of an assignment and needs to be able to function like an expression. For example:

```
Declare int x
Declare int y
Set x to y
```

Y is on the right side, so it needs to be evaluated as an expression

AST Node Sum

- What would the Sum (+) class and its AST diagram look like?
- How many children? What are the types of the children?

AST Node Sum

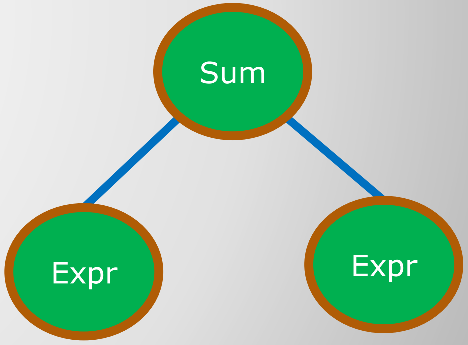
- What would the Sum (+) class and its AST diagram look like?
- How many children? What are the types of the children?
- The + operator has two operands so it should have two children.
- The operands are expressions.

```
class Sum extends Expr {
    Declare Expr lhs
    Declare Expr rhs

    Sum Constructor(Expr lhs, Expr rhs) {
        Set this.lhs to lhs
        Set this.rhs to rhs
    }
}
```

Sum inherits from Expr. Sum can be used as an operand for another Expr.

The Sum class is used for operator (+) nodes in the AST



lhs and rhs are children of Sum

AST Node Assignment

- An assignment stores an expression value into a variable.
- What would the Assign class and its AST diagram look like?

- An assignment stores an expression value into a variable.
- What would the Assign class and its AST diagram look like?

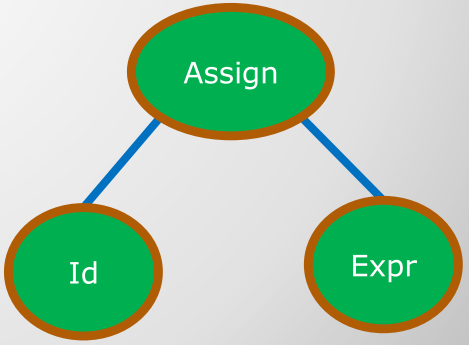
```
class Assign extends Stmt {
    Declare Id id
    Declare Expr rhs

    Assign Constructor(Id id, Expr rhs) {
        Set this.id to id
        Set this.rhs to rhs
    }
}
```

Id being assigned to

Value to put in the id (rhs means right hand side)

The Assign class has an id and expression as children



Assign inherits from Stmt (an assignment is a specific type of statement)

AST Node Assignment

AST Node Collection of Statements

- A StmtCollection stores multiple statements.
- What would the StmtCollection class and its AST diagram look like?

AST Node Collection of Statements

- A StmtCollection stores multiple statements.
- What would the StmtCollection class and its AST diagram look like?

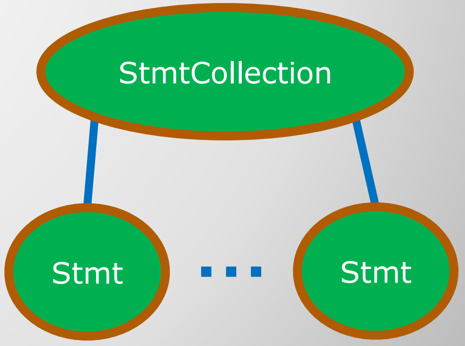
```
class StmtCollection extends ASTBase {
    Declare List of Stmt

    Define method to add a Stmt to the list
}
```

List of Stmt. Any class that derives from Stmt can be added to the list

The StmtCollection class has multiple Stmt instances as children (there is no limit to the number of children)

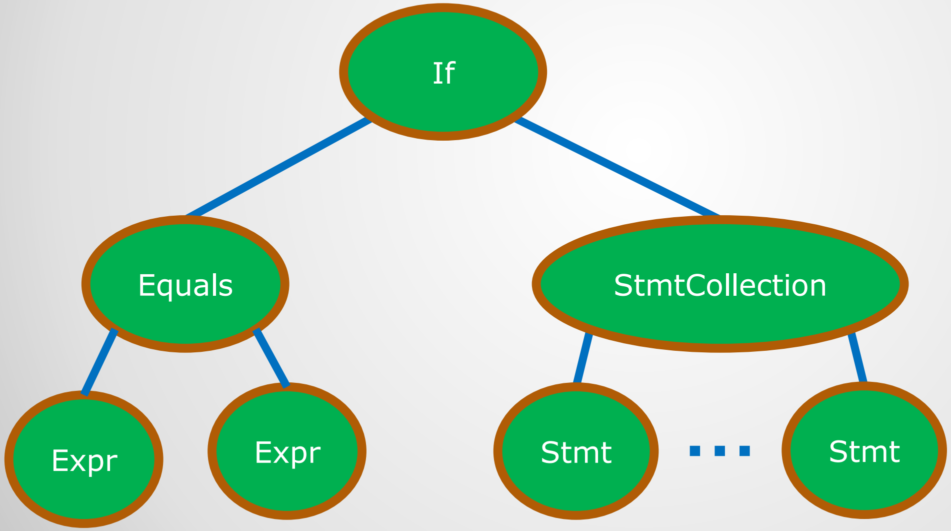
StmtCollection inherits from ASTBase. It does not inherit from Stmt or Expr. It is not a specific type Stmt. It cannot be an operand to an Expr.



AST Node Assignment

- Assume that an if statement only uses equals when testing.
- Assumes that an Equals class has been defined that tests if two expressions are equal.
- What would the AST diagram be for this if statement?

- Assume that an if statement only uses equals when testing.
- Assumes that an Equals class has been defined that tests if two expressions are equal.
- What would the AST diagram be for this if statement?



The If class needs an equality test and a collection of statements to execute

AST Node Assignment

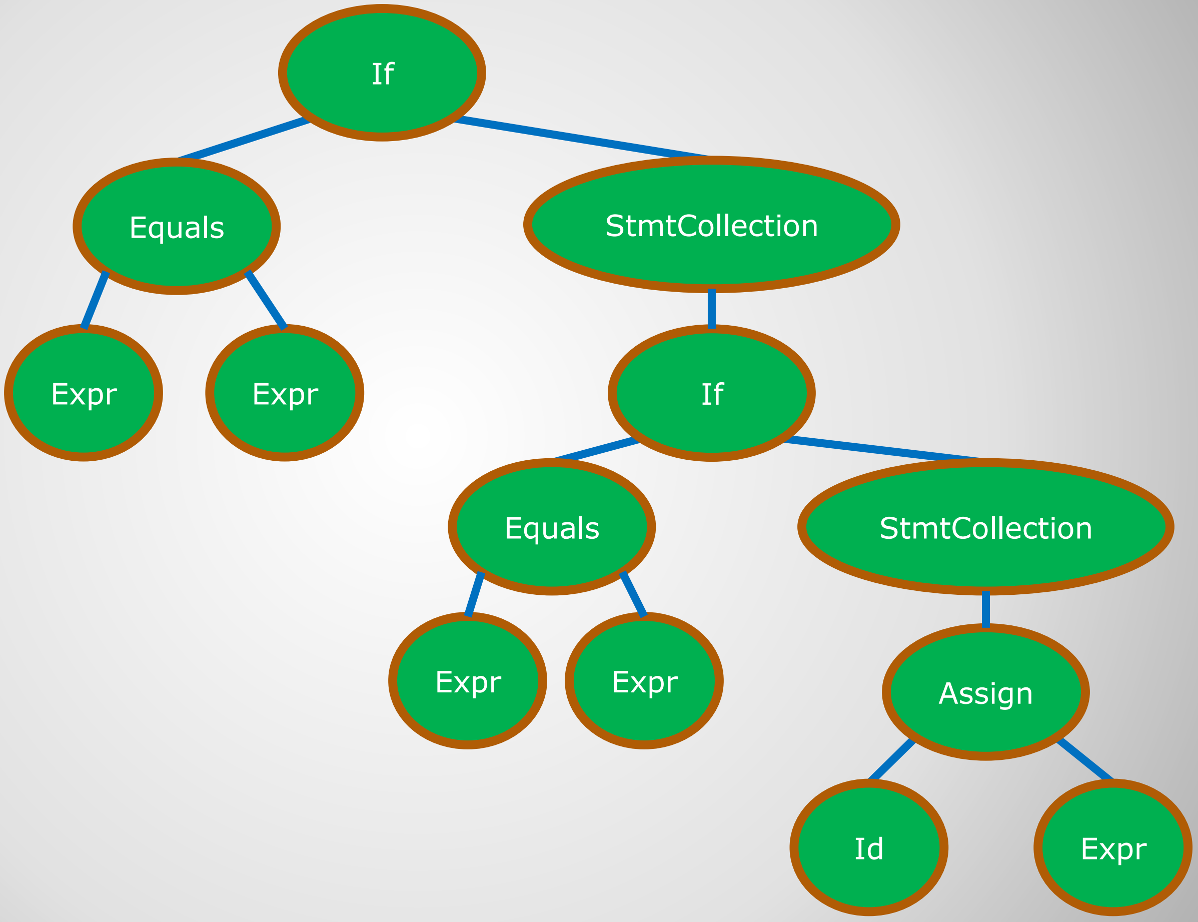
AST Node Assignment

- What would the AST diagram be for this nested if statement?

```
If w = x
   If y = z
      a = b
   endif
endif
```

- What would the AST diagram be for this nested if statement?

```
If w = x
   If y = z
      a = b
   endif
endif
```



AST Node Assignment

- Generating the AST during parsing…

Generating the AST During Parsing

Create AST for an Expression

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

Expr → Factor ExprEnd

ExprEnd → + Factor ExprEnd

ExprEnd → ε

Factor → id

Factor → intliteral

- Assume that the following have been defined:
  - Expr – AST node base class.
  - Sum – AST node class that inherits from Expr. It has two children that correspond to the operands.

- The code on the upcoming slides uses the Expr and Sum AST node classes.

Create AST for an Expression

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

Expr → Factor ExprEnd
ExprEnd → + Factor ExprEnd
ExprEnd → ε
Factor → id
Factor → intliteral

Nonterminal methods below return AST nodes (descriptions of each method on upcoming slides)

```
parse()
Declare Expr expr
Set expr to Expr()
```

```
expr() returns Expr
Declare Expr factor
Declare Expr rhsSum
Set factor to factor()
Set rhsSum to exprEnd()
If (rhsSum equals null)
   return factor

Return new Sum(factor, rhsSum)
```

```
exprEnd() returns Expr
Declare Expr factor
Declare Expr rhsSum

If nextToken equals PLUS
   match(PLUS)
   Set factor to factor()
   Set rhsSum to exprEnd()
   If (rhsSum == null)
      Return factor

   Return new Sum(factor, rhsSum)

Return null
```

```
factor() returns Expr
Declare Expr expr

If nextToken equals ID
   Declare idName String
   Set idName to tokenBuffer
   match(ID)
   Set expr to new Id(name)

If nextToken equals INTLITERAL
   Declare intValue Int
   Set intValue to tokenBuffer
   match(INTLITERAL)
   Set expr to new IntLiteral(name)

Return expr
```

Create AST for an Expression

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

Expr → Factor ExprEnd
ExprEnd → + Factor ExprEnd
ExprEnd → ε
Factor → id
Factor → intliteral

parse()

```
Declare Expr expr
Set expr to expr()
```

Expr is the starting nonterminal in this grammar.

The call to expr() in parse returns the root of the AST.

Create AST for an Expression

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

Expr → Factor ExprEnd
ExprEnd → + Factor ExprEnd
ExprEnd → ε
Factor → id
Factor → intliteral

```
expr() returns Expr
Declare Expr factor
Declare Expr rhsSum

Set factor to factor()
Set rhsSum to exprEnd()

If (rhsSum equals null)
   return factor

Return new Sum(factor, rhsSum)
```

The factor variable holds the left side of an expression

The rhsSum variable holds the right side of an expression. We get the right side AST node by calling exprEnd(). exprEnd may return null. If null is returned then it used the ExprEnd → ε production

If rhsSum is null, then there is nothing to the right of factor in this expression. If this is the case, then just return the factor.

If we get here, then there is both a left and right side for the expression so create a new Sum node.

Create AST for an Expression

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

Expr → Factor ExprEnd
ExprEnd → = + Factor ExprEnd
ExprEnd → ε
Factor → id
Factor → intliteral

<u>exprEnd() returns Expr</u>

```
Declare Expr factor
Declare Expr rhsSum

If nextToken equals PLUS
   match(PLUS)
   Set factor to factor()
   Set rhsSum to exprEnd()
   If (rhsSum == null)
      Return factor

   Return new Sum(factor, rhsSum)

Return null
```

Consume the + symbol

Recognize the factor just to the right of +

rhsSum holds the expression to the right of the factor we just recognized. Recursively call exprEnd().

If rhsSum is null, then there is nothing to the right of factor in this expression (just return the factor).

If we get here then there are two operands, return a new Sum node.

If we get here (outside if), then there was no + so the expression is finished. Returning null means it is using the ExprEnd → ε production.

Create AST for an Expression

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

Expr → Factor ExprEnd
ExprEnd → = + Factor ExprEnd
ExprEnd → ε
Factor → id
Factor → intliteral

factor() returns Expr

```
Declare Expr expr

If nextToken equals ID
   Declare idName String
   Set idName to tokenBuffer
   match(ID)
   Set expr to new Id(name)

If nextToken equals INTLITERAL
   Declare intValue Int
   Set intValue to tokenBuffer
   match(INTLITERAL)
   Set expr to new IntLiteral(name)

Return expr
```

Get the id name for the token buffer

Consume the ID symbol

Create a new Id node with the name in the string buffer

Get the int literal from the token buffer

Consume the INTLITERAL symbol

Create a new IntLiteral node with the value from the string buffer

- **End of Slides**

End of Slides