COMP 520 - Compilers

Lecture 11 – Recap of Contextual Analysis in PA3

Midterm 1 Results

- Maximum: 100 (18)
- Median: 95.5
- Mean: 94.8

Midterm 1 Results (2)

- Question 2: comparable to previous midterm
- Question 3: was exactly from a previous midterm
- Question 4: was harder than a previous midterm

Midterm 1 Results (3)

- Question 2: comparable to previous midterm
- Question 3: was exactly from a previous midterm
- Question 4: was harder than a previous midterm

- In comparison, the average for this class was more than 10 points higher than an earlier class

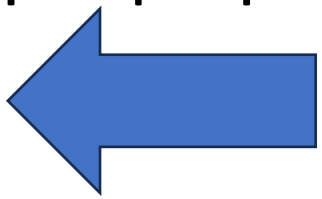
Upcoming Code Generation

- We will have a few in-class demos where we SHOW how a processor works.

- Will show visualizations of what will happen in a program in some live demos.

LL(1), 4 Statement Cases

```
this(.id)*(= Expr ;| [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id ( .id(.id)*(=Expr;|[Expr]=Expr;|(ArgList?);)
      | =Expr; | (ArgList?);
      | [(] id = Expr;|Expr] = Expr;)
      | id = Expr;
)
```



Was missing from earlier slide

Please note: miniJava

- miniJava does not have typecasting nor automatic conversions.
- miniJava does not allow boolean × int × Op

Programming Assignment 3

Identification and Type-Checking

You own the ASTs for PA3 and forward

- Can add, edit, remove any ASTs you want.

Strategy

- Two separate Visitor implementations
- First identification, then type-checking

Strategy (2)

- Two separate Visitor implementations
- First identification, then type-checking

- It is possible to do this in one AST traversal. Optional, is a PA5 extra credit item.

Identification Goal

- Every Identifier gets a "decl" field added, of type Declaration
- We want to locate where every identifier is declared.

- Could be a VarDecl, ParameterDecl, MemberDecl, ClassDecl

Why?

- Why is it that only syntax checking, and context checking is enough for ensuring an input program is correct?

Today

- First, we will cover type-checking
- Then, we will page identification back in

- Goal: Learn how to properly ensure type checking is implemented, and go backwards to determine how to enable type checking with identification.

PA3 – Type Checking

Type-Checking Table

- In miniJava, type order does not matter, so A×B×op is the same as B×A×op

- This means we can simplify our TypeChecking table.

miniJava – Types must match

- For miniJava, the types must match. There is no automatic type conversions nor manual typecasting.

miniJava – Types must match

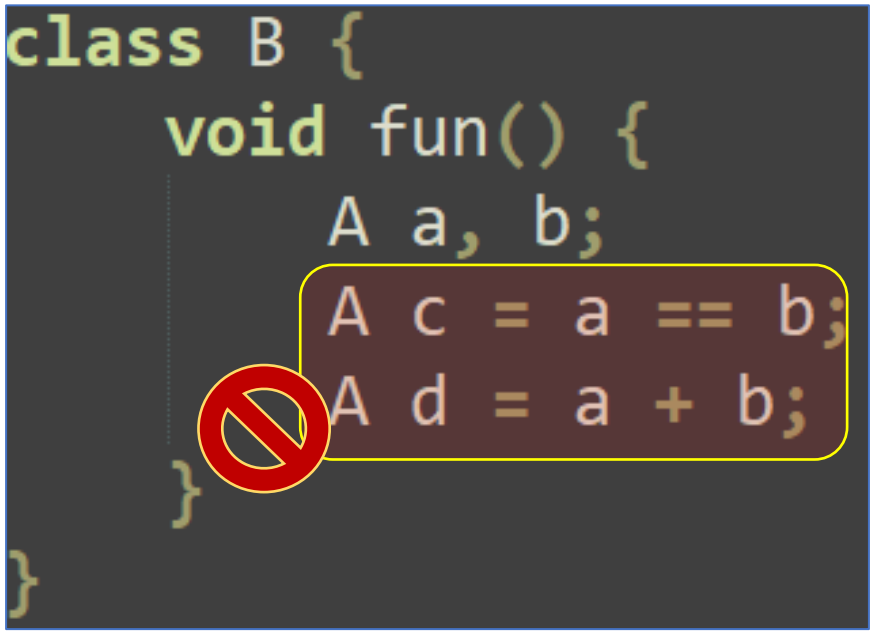
- For miniJava, the types must match. There is no automatic type conversions nor manual typecasting.

- Does this mean we can use a REALLY simple type-checking table where both types must match, **and the result type is that type**?

miniJava – Types must match

- Does this mean we can use a REALLY simple type-checking table where both types must match, **and the result type is that type**?

- Still need to formally clarify Type rules.



Type-Checking Table (2)

Type Checking Rules		
Operand Types	**Operand**	**Result**
boolean × boolean	&&, ||	boolean
int × int	>, >=, <, <=	boolean
int × int	+, -, *, /	int
α × α	==, !=	boolean
int	(Unary) -	int
boolean	(Unary) !	boolean

ClassType

- If two objects are both ClassType, are they comparable?

ClassType (No polymorphism in miniJava)

- If two objects are both ClassType, are they comparable?
- No, the underlying Identifier text must match.
- Why is this enough?

```
1  class A {
2  }
3
4  class B {
5      void fun() {
6          A a = this;
7      }
8  }
```

What type is ArrayType?

- **Recall:** `new int[4]`
- **This expression is of** `ArrayType(IntType)`
- **Thus, it can only be assigned to variables of type** `ArrayType(IntType)`

- `IntType` **is shorthand for:** `BaseType( TypeKind.INT )`

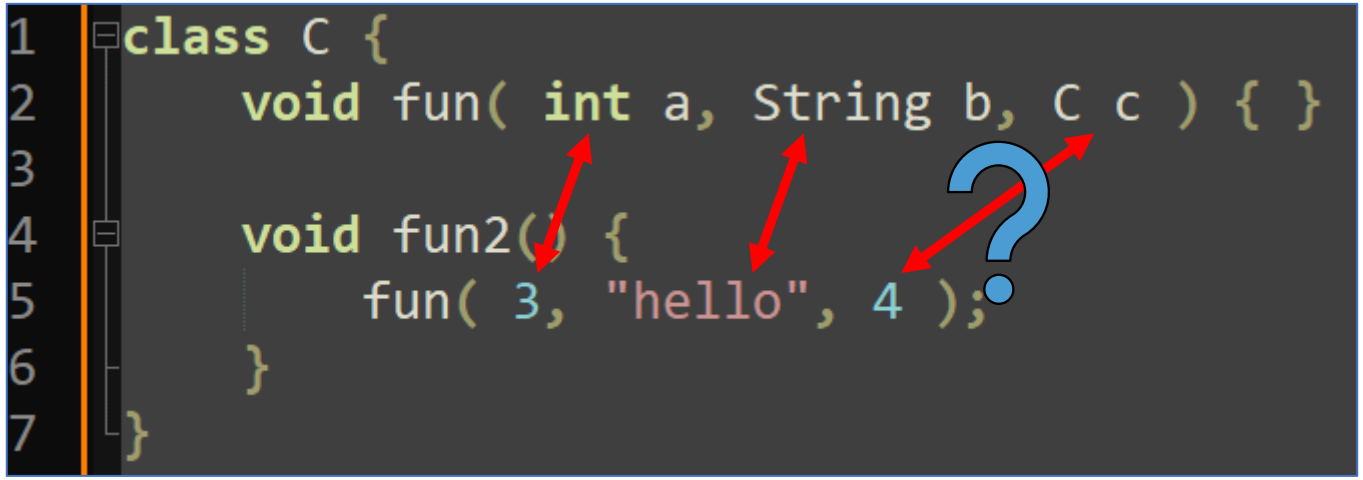
What type is ArrayType? (2)

- For array types:
  - First: Are both types ArrayType?
  - Second: Do the element types match? (Recursion)

- Recursion needed to match ArrayType of ArrayType of ArrayType of ClassType.

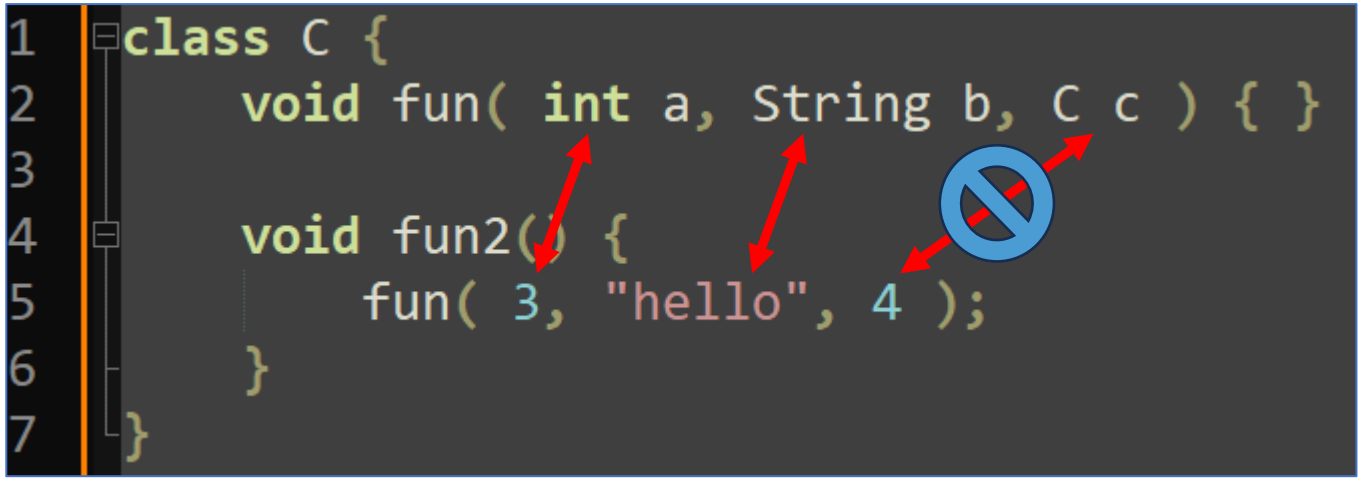
Type-Checking Methods

- Scoped Identification **only uses context and identifiers**. Therefore, overloading methods by parameter types/counts is not allowed in miniJava.



Type-Checking Methods (2)

- As such, make sure there is an expression for every parameter, and that the types match.



Type Errors

- The ErrorType is compatible with ALL other types, and the result type is always another ErrorType and this does not cause an error to be reported.

- If a type is not allowed in an operation with another type, then the result type is an ErrorType.

Unsupported Type

- The UNSUPPORTED type is not compatible with any type (including itself) and causes an error to be reported. The result type will be ErrorType.

- Make sure String's type is UNSUPPORTED, otherwise String can be initialized with new String(), which is not implemented in miniJava.

Unsupported Type (2)

- The String predefined class is an UNSUPPORTED type.
- String is not supported in miniJava, but available to be implemented as a part of PA5.

- We need String to be able to declare the main method.

Unsupported Type (3)

- UNSUPPORTED×ErrorType
- Question: should an error be reported?

Unsupported Type (4)

- UNSUPPORTED×ErrorType does not need to be reported (only way ErrorType exists is if an error was reported earlier anyway).

- But it can be reported if you want to report an extra error where String is utilized.

Type-Checking Strategy (1)

- Implement a TypeChecking Visitor that uses a TypeDenoter return type.

- Visiting a node synthesizes a TypeDenoter for that type.

Traverse an AST bottom-up?

- Not talking about SR parsing here.
- We're done with Parsing and Syntactic Analysis

Traverse an AST bottom-up? (2)

- The leaf nodes of an AST are visited first.
- Why?

Traverse an AST bottom-up? (2)

- The leaf nodes of an AST are visited first.

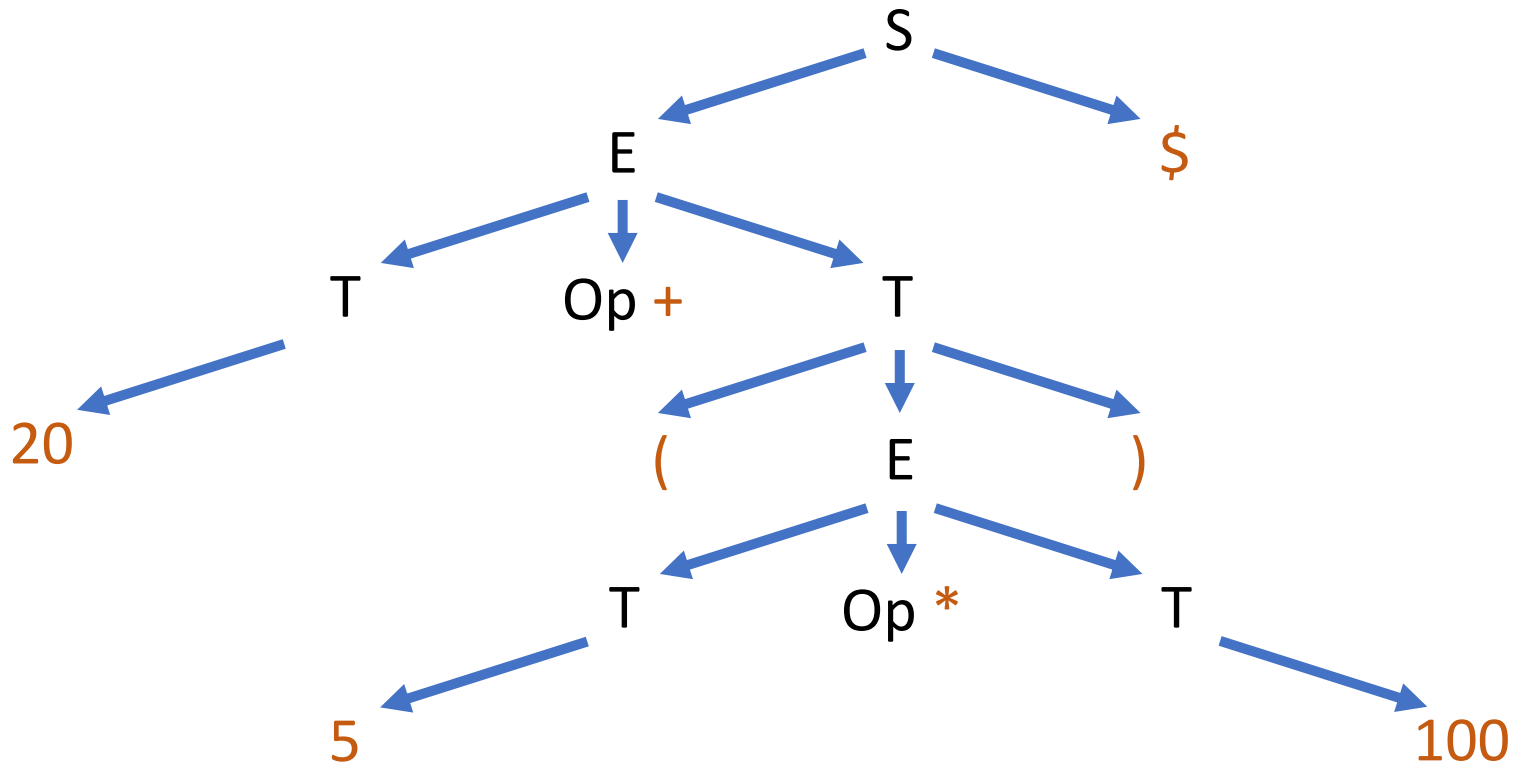
- If I have an expression: int b = 20 + 5 * 100
- I don't know the type of that expression just by looking at the "Expression" AST

Traverse an AST bottom-up? (3)

- If I have an expression: int b = 20 + 5 * 100
- I don't know the type of that expression just by looking at the "Expression" AST

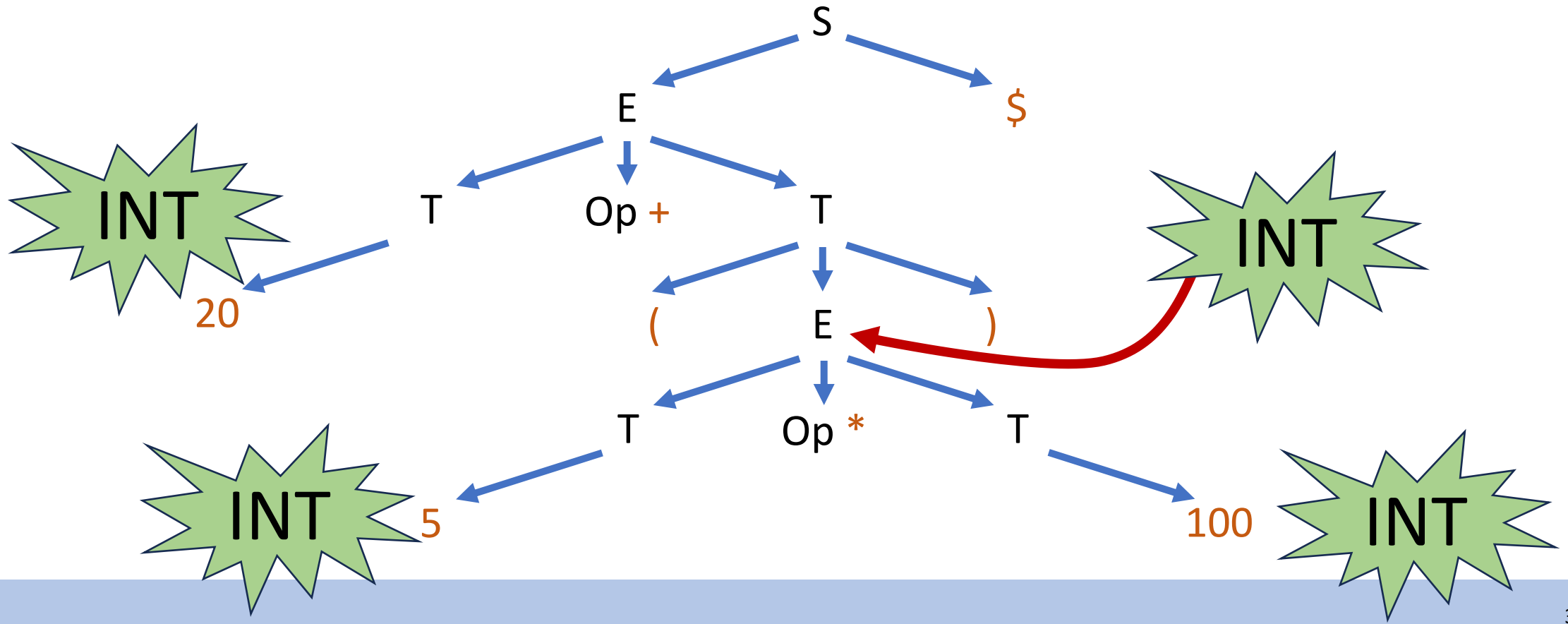
Traverse an AST bottom-up? (3)

- If I have an expression: int b = 20 + 5 * 100



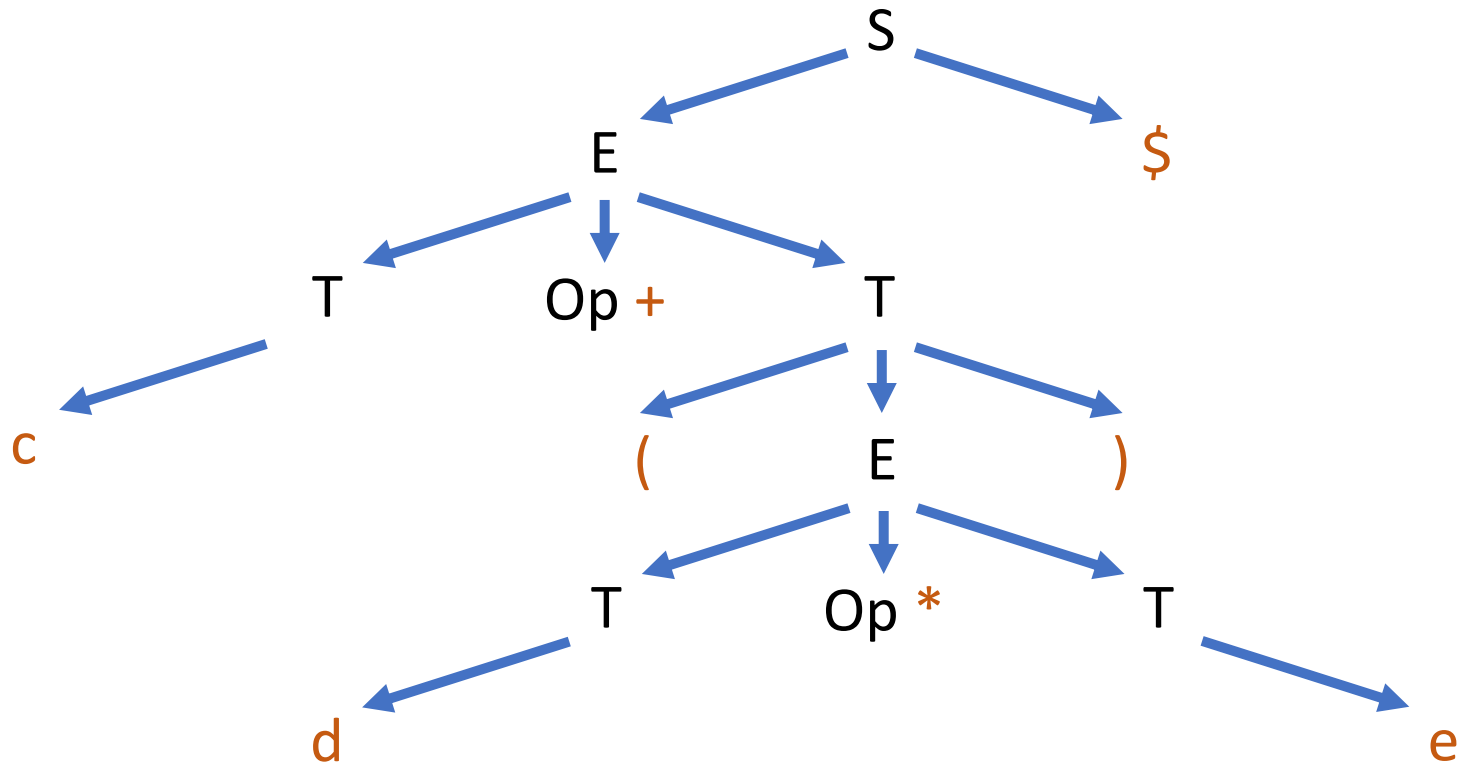
Traverse an AST bottom-up? (4)

- If I have an expression: int b = 20 + 5 * 100



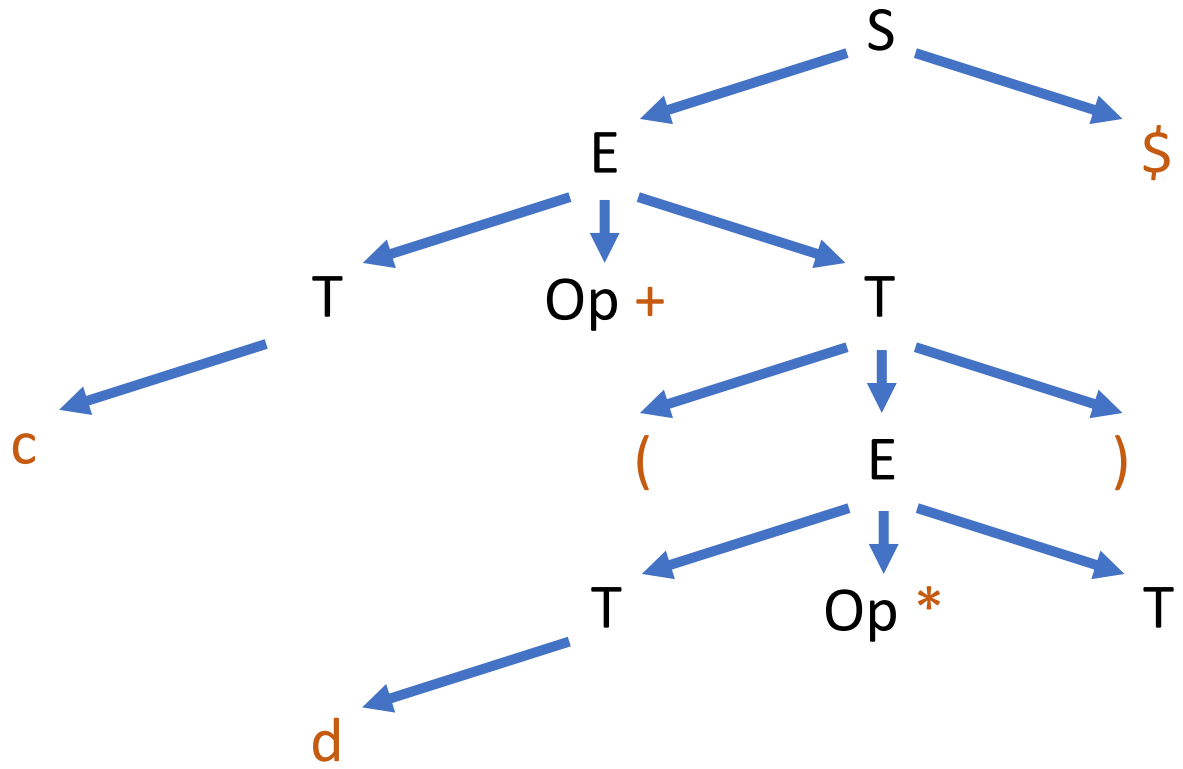
Traverse an AST bottom-up? (5)

- If I have an expression: int b = c + d * e



Traverse an AST bottom-up? (5)

- If I have an expression: int b = c + d * e



How can we determine the type of c, d, e (and b)?

Type-Checking Strategy (2)

- Implement a TypeChecking Visitor that uses a TypeDenoter return type.

- **Visiting a node synthesizes** a TypeDenoter for that type.
- Create a method, input is the two TypeDenoters, (or one for Unary), and output is the resultant TypeDenoter.

Type-Checking Strategy (3)

- Implement a TypeChecking Visitor that uses a TypeDenoter return type.

- Visiting a node synthesizes a TypeDenoter for that type.
- Create a method, input is the two TypeDenoters, (or one for Unary), and output is the resultant TypeDenoter.
- Or create a table, but that would have a lot of null entries.

Type-Checking Strategy (2)

- Ensure index expressions are integers

```
A[ IndexExpr ]
```

- Ensure condition expressions in if/while are Boolean

```
if( CondExpr ) / while( CondExpr )
```

- Ensure operands are compatible, and return the appropriate type when visiting that BinExpr/UnaryExpr

```
A a;
a = 3 + a;
```

Back to Identification

Scoped Identification Stack

- Some languages do not let you access all members in a stack.
- This is not the case for Java.

Scoped Identification Stack (2)

- Some languages do not let you access all members in a stack.
- This is not the case for Java.

- Actually, this is mostly not a case for a lot of languages
- For example, `std::priority_queue` says you can't iterate through it, but you can just get the underlying container.

Upcoming Side Note*

- As we will see, many language constraints like public/private are only enforced at the compiler,…
- Except interpreted languages, then it is enforced by the interpreter.
- Key point: hardware doesn't care, memory is memory.

Identification Cache

- Identifier "x" only makes sense in context.
- Even if two identifiers' underlying text is the same, the declaration can be different when appearing in different parts of the code.

```
class A {
    int x;
}

class B {
    int x;
}
```

Both use "x" as the identifier,
but can only tell them apart in context.

Left-most Reference

- Only the left-most reference should be resolved normally (start at the top of the SI stack, then work down).
- Once you know the Declaration of the left-most reference, you have a **context**.

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Left-most Reference

- QualRef(LHS,RHS): LHS is a Reference, and RHS is an Identifier.
- With the type of the LHS (the context), resolve the RHS.
- a.b means ".b" is resolved in the context of the type of "a", which is class "A".

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Left-most Reference

- With the type of the LHS (the context), resolve the RHS.
- **Note:** this means that you can bypass local variables.
- "a.b.c.x" but "b", "c", "x" were all locally defined.

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

QualRef Strategy

- Try to get the "context" by visiting the LHS reference.
- With that context, resolve the RHS.

- E.g. "a.b" will return the context of class "B", thus allowing resolution of "a.b.c" where "c" is in the context of "B"

No one strategy dominates all others

- How you choose to identify "context" is up to you. It can be a String, ClassDecl, TypeDenoter, etc.

- Even more important to plan PA3 than other assignments before starting to code.
- If you change your Visitor's parameter or return type, you may have to redo the entire class declaration!

Other Contextual Constraints

Contextual Analysis

- There are contextual parts of Java (and miniJava) that do not quite fit Identification or Type Checking.
- We can easily implement these as a part of either.

Contextual Analysis (2)

- If an identifier is being declared, then it cannot be used in the expression.
- Even if the expression can be evaluated first!

```
1  class C {
2      int x = 1;
3      void fun() {
4          int x = x + 1;
5      }
6  }
```

Not allowed!

Contextual Analysis (3)

- You cannot have a variable declaration only in a scope to itself.
- A BlockStmt (new scope) is necessary for VarDeclStmt.

```
1  class C {
2      int x = 1;
3      void fun() {
4          if( true )
5              int x = 3;
6      }
7  }
```

Not allowed!

End