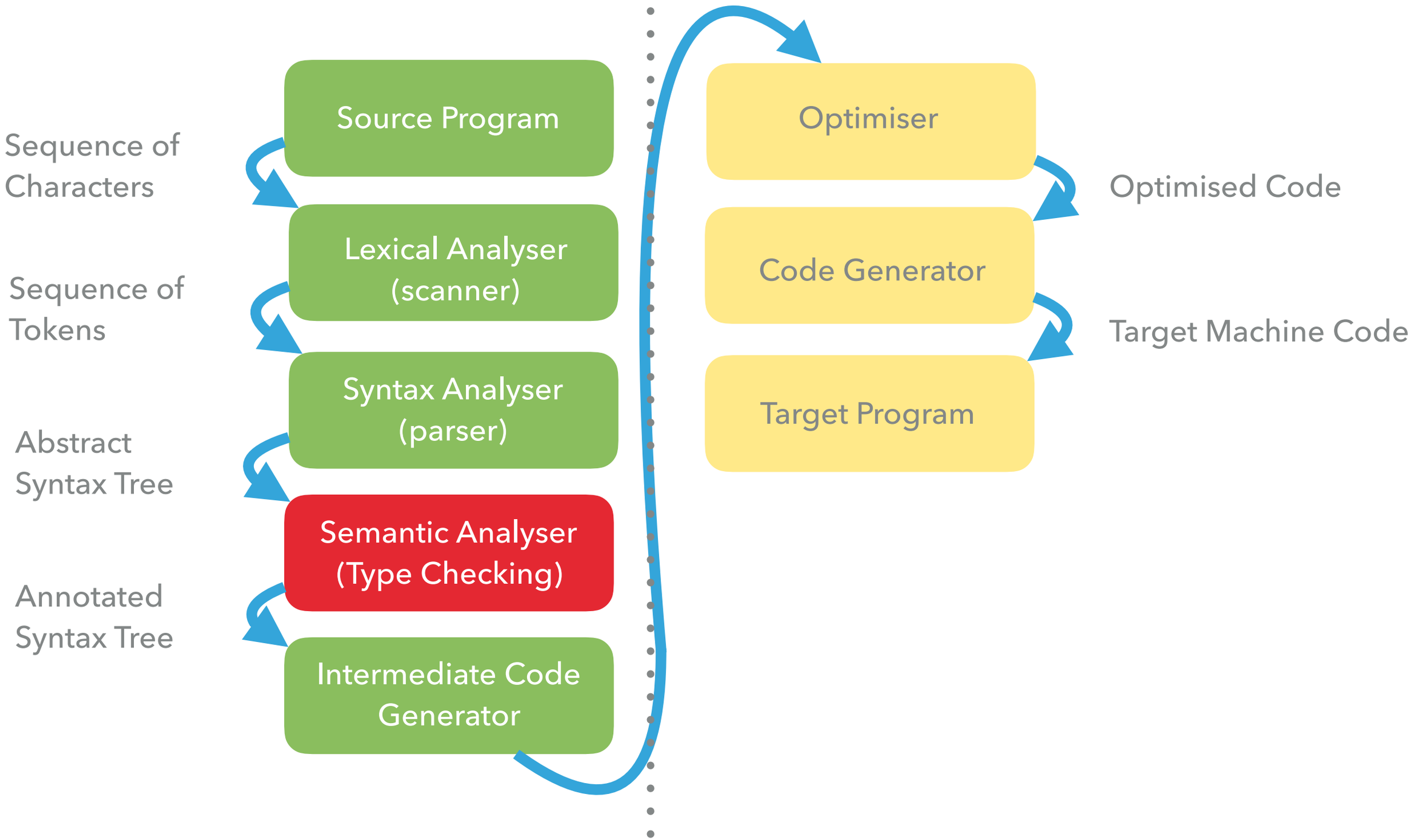
SEMANTIC ANALYSIS WITH THE VISITOR PATTERN

THIS WEEK

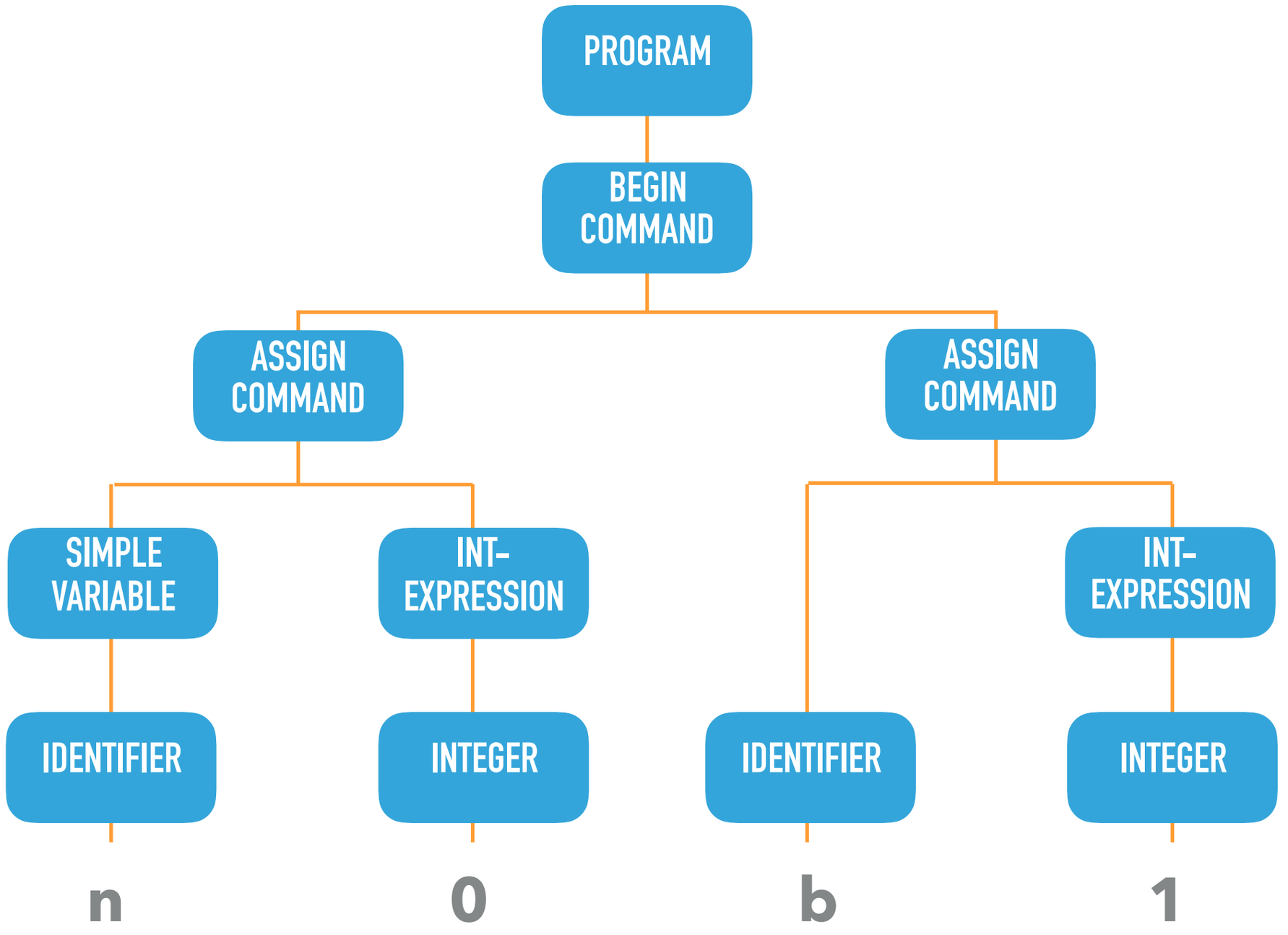
- Performing Semantic Analysis
- Checking against the semantic rules
- Updating the Symbol Table
- Checking Types

PHASES OF A COMPILER



FROM LAST WEEK

- We now have the AST from the Parser
- So how do we see if it's semantically correct?



SEMANTIC ANALYSIS RECAP

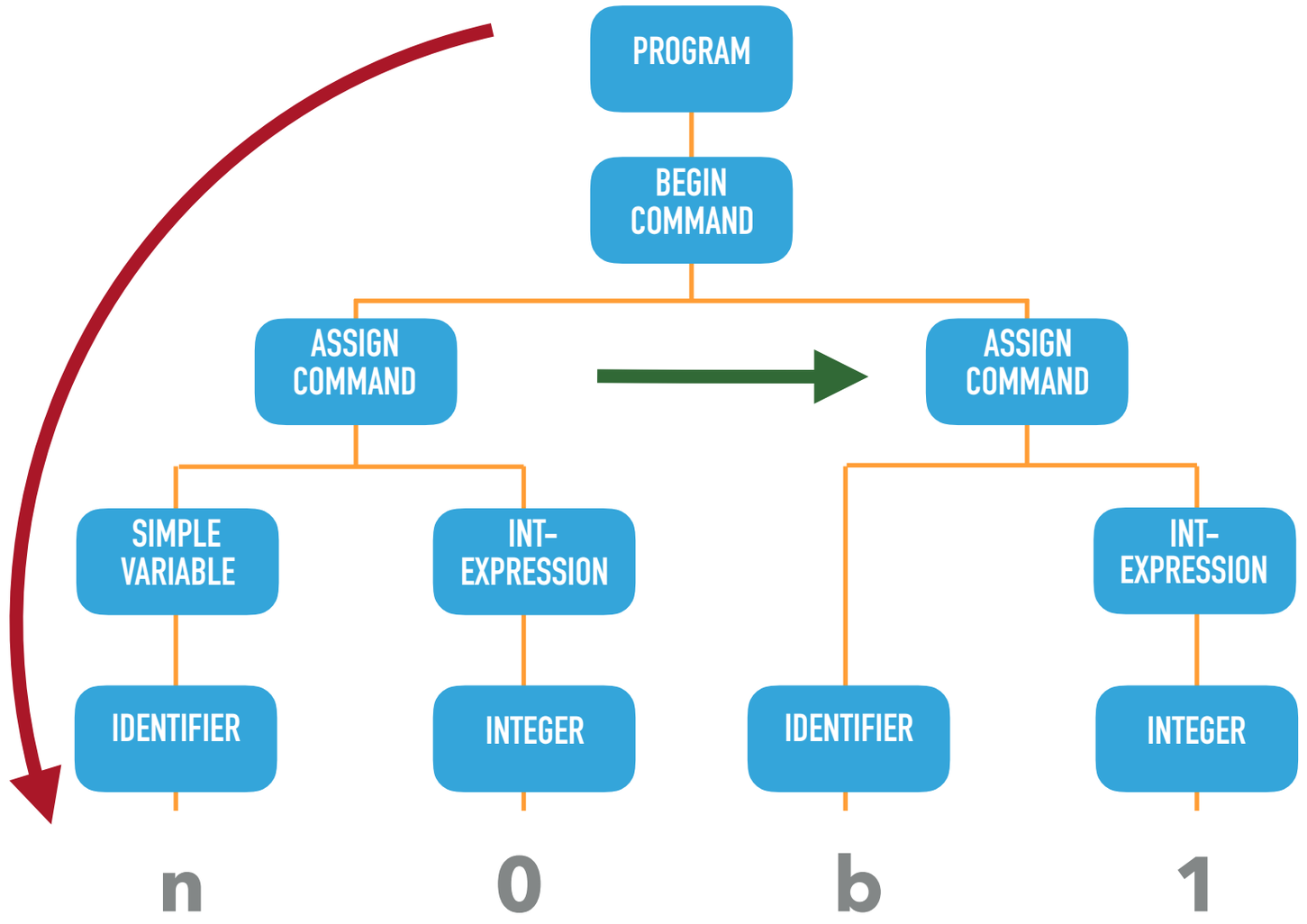
- Every language will have semantic rules, or constraints which govern what the language can do.
- You already know these for the languages you use
  - if (3) then 4 else 8
  - int v = “wobble”
- You know inherently these are wrong, but they are actually defined somewhere

THE SEMANTIC ANALYSIS STAGES

- **Identification**
  - we apply the source language's scope rules to relate each identifier to its declaration (if it has one)
- **Type Checking**
  - we apply the source language's type rules to determine if the an expressions type matches the expected type

SEMANTIC ANALYSIS ALGORITHM

- Run over our AST **depth-first left-to-right**
- **Visit** each node of the AST and **decorate** it with info about the type.



IMPLEMENTING IDENTIFICATION & TYPE CHECKING

- **Identification** can be achieved by making an explicit link from the each applied occurrence of an identifier back to it's declaration (using the symbol table)

- **Type-Checking** can be performed by storing each expressions inferred type (eg int, bool etc) at the root of the node where it is used

- There are a number of ways we could do this, but we are going to use the Visitor Pattern

THE VISITOR PATTERN

- This is a design pattern that lets you perform operations on the elements of an object **without changing** the objects themselves.
- Our objects are the **nodes of our AST**
  - We don't want to change the actual node objects, just see what they are
  - and if they match the semantic rules

IMPLEMENTATION OF THE VISITOR PATTERN

- Process
  - define a special Visitor class to visit the nodes of the tree
  - add (only one) visit method to the Abstract Syntax Tree (AST) classes (in your Syntax Tree folder)
  - these methods let the visitor visit the AST nodes

SIMPLIFIED EXAMPLE

```
public class IfCommand : Command
  public Object Visit(Visitor v, Object arg) {
    return v.VisitIfCmd(this, arg);
  }
}
```

- Our IfCommand class (in our syntax tree) has a new method called Visit.
- This method receives an instance of visitor and some arguments
- this visitor instance is then used to run a visitor method specific to the IfCommand to do the identification and type checking

IFCOMMAND AST

- Quick reminder of our AST class for the IFCommand
- Contains all the other AST classes needed to represent an IFCommand

if **Expression** then **Single-Command** else **Single-Command**

```
public class IfCommand : Command {
    readonly Expression _expression;
    readonly Command _trueCommand;
    readonly Command _falseCommand;

    public IfCommand(Expression expression, Command trueCommand,
    Command falseCommand, SourcePosition position) : base(position) {

      _expression = expression;
       _trueCommand = trueCommand;
       _falseCommand = falseCommand;
}
```

WHAT VISITIFCOMMAND LOOK LIKE?

```
public Void VisitIfCommand(IfCommand ast, Void arg)
{
  var expressionType = ast.Expression.Visit(this);
  ast.TrueCommand.Visit(this);
  ast.FalseCommand.Visit(this);
  return null;
}
```

- Our VisitIfCommand method, in the Visitor Class receives the AST representation of the IfCommand (last slide)
- It can then access the separate components and call the visitor methods for those components.

WHAT DOES THIS DO?

```
var expressionType = ast.Expression.Visit(this);
```

- We are asking the Expression if it has a type.
- Because of polymorphism, the visit method will execute on the correct type of Expression we have.
- in this case a BinaryExpression

VISIT BINARY EXPRESSION (NOT WHOLE METHOD)

Expression Operator Expression

```
public TypeDenoter VisitBinaryExpression(BinaryExpression ast, Void arg)
    {
        var e1Type = ast.LeftExpression.Visit(this);
        var e2Type = ast.RightExpression.Visit(this);

        var binding = ast.Operator.Visit(this);

        var bbinding = binding as BinaryOperatorDeclaration;
```

We can visit the two expressions

And then the operator

This is a bit more complicated, we need to go and check where the Operator has been declared and what possible types the operator can work with

BINARY OPERATOR DECLARATION

```
public BinaryOperatorDeclaration(Operator op, TypeDenoter firstArgument, TypeDenoter
secondArgument, TypeDenoter result) : base(SourcePosition.Empty)
        {
            _operator = op;
            _firstArgument = firstArgument;
            _secondArgument = secondArgument;
            _result = result;
        }
```

- A Binary Operator has a couple of components
- the operator itself eg (+, -, >, <, =) etc
- the first argument type (eg int-lit, char-lit)
- the second argument type (eg int-lit, char-lit)
- and the result.... 1 + 2 would result in an int 1< 2 would result in a bool

STANDARD ENVIRONMENT

- But we haven't declared any of this in our source?
- So where do the values come from?
- StandardEnvironment.cs contains all the standard definitions for the language, built in functions eg getInt(), putInt()
- But also standard operator declarations

```
 public static readonly BinaryOperatorDeclaration LessDecl
= DeclareStdBinaryOp("<", IntegerType, IntegerType, BooleanType);
```

- in this case < must have 2 Integers and results in a Boolean

TYPE CHECKING – REST OF THE VISIT BINARY EXPRESSION

```
if (bbinding != null)
{
    if (bbinding.FirstArgument == StandardEnvironment.AnyType)
    {
        // this operator must be "=" or "\="
        CheckAndReportError(e1Type.Equals(e2Type), "incompatible argument types for \"%\"",
                ast.Operator, ast);
    }
    else
    {
        CheckAndReportError(e1Type.Equals(bbinding.FirstArgument),
                "wrong argument type for \"%\"", ast.Operator, ast.LeftExpression);
        CheckAndReportError(e2Type.Equals(bbinding.SecondArgument),
                "wrong argument type for \"%\"", ast.Operator, ast.RightExpression);
    }
    return ast.Type = bbinding.Result;
}
ReportUndeclaredOrError(binding, ast.Operator, "\"%\" is not a binary operator");
return ast.Type = StandardEnvironment.ErrorType;
```

For '=' we can have any type on each side of the operator so we just check if that is the case for this operator

If so we just check if the type of the right hand side matches the left hand side

Otherwise we check the types against what the operator says we can have

We then return the type the operator says we should have as a result

If it all goes horribly wrong, generate an error

BACK TO OUR IFCOMMAND

```
public Void VisitIfCommand(IfCommand ast, Void arg)
    {
        var expressionType = ast.Expression.Visit(this);
        CheckAndReportError(expressionType == StandardEnvironment.BooleanType,
            "Boolean expression expected here", ast.Expression);
        ast.TrueCommand.Visit(this);
        ast.FalseCommand.Visit(this);
        return null;
}
```

- We have now got the type from the expression so can check this against the required type.
- From our semantic rules an IF must use a boolean condition. So if it does not, generate an error
- otherwise go ahead and visit the two commands

WHERE DOES THE SYMBOL TABLE COME IN

- When we declare a variable or constant we need to add the identifier to the Symbol (or ID) table

```
Identifier _identifier;
Expression _expression;

public ConstDeclaration(Identifier identifier, Expression expression,
SourcePosition position)
: base(position)
{
        _identifier = identifier;
        _expression = expression;
}
```

- Our ConstDeclaration looks like this, it has an identifier and an expression
- representing the AbstractSyntax **const** Identifier **~** Expression
- Something like const MAX ~ 10

VISITING A DECLARATION

```
public Void VisitConstDeclaration(ConstDeclaration ast, Void arg)
{
     ast.Expression.Visit(this);
     _idTable.Enter(ast.Identifier, ast);

     CheckAndReportError(!ast.Duplicated,
        "identifier \"%\" already    declared”, ast.Identifier, ast);

     return null;
}
```

- So we visit the Expression (leaving any errors to that method)
- Then we try to enter the new identifier in to the table
- finally we run a check to see if its a duplicate.

VAR DECLARATION

- the visit var declaration works in exactly the same way

```
public Void VisitVarDeclaration(VarDeclaration ast, Void arg)
{
    ast.Type = ast.Type.Visit(this);

    _idTable.Enter(ast.Identifier, ast);

    CheckAndReportError(!ast.Duplicated,
            "identifier \"%\" already declared", ast.Identifier, ast);
    return null;
}
```

Visit the type-denoter

then add the ID to the table

Check we haven't created a duplicate

WHAT ABOUT SCOPE

- Remember that the LET command defines the scope in our language
- So when we visit the let command we need to open a new scope for the Symbol Table

```
public LetCommand(Declaration declaration, Command command, SourcePosition
position)
            : base(position)
        {
            _declaration = declaration;
            _command = command;
        }
```

VISITING A LET COMMAND

- Its actually pretty simple

```
public Void VisitLetCommand(LetCommand ast, Void arg)
{
    _idTable.OpenScope();

    ast.Declaration.Visit(this);
    ast.Command.Visit(this);

    _idTable.CloseScope();
    return null;
}
```

Open the scope

visit the decarations and then any other commands within the scope

Close the scope

LIES

```
public class IfCommand : Command
  public Object Visit(Visitor v, Object arg) {
    return v.VisitIfCmd(this, arg);
  }
}
```

- At the start I showed you a visit method that looked like this.
- This isn't strictly true, its a bit more complicated.

INTERFACES AND GENERICS

- To structure our semantic analysis we will use interfaces to define what methods our visitor class must have
- we will separate this out into a number of interfaces to defined the type of phrases (eg declaration, expression, terminal) being visited
- and will use generics to deal with the parameters at each stage.

THE VISITOR INTERFACES

- IActualParameterSequenceVisitor.cs
- IActualParameterVisitor.cs
- ICommandVisitor.cs
- IDeclarationVisitor.cs
- IExpressionVisitor.cs
- IFormalParameterSequenceVisitor.cs
- IFormalParameterVisitor.cs
- IIdentifierVisitor.cs
- ILiteralVisitor.cs
- IOperatorVisitor.cs
- IProgramVisitor.cs
- ITypeDenoterVisitor.cs
- IVnameVisitor.cs

- Each of these interfaces defines what methods a visitor to these types of phrases should have
- they all use generics to cover the returns and arguments

IExpressionVisitor.cs

```
TResult VisitBinaryExpression(BinaryExpression ast, TArg arg);
TResult VisitCallExpression(CallExpression ast, TArg arg);
TResult VisitCharacterExpression(CharacterExpression ast, TArg arg);
TResult VisitEmptyExpression(EmptyExpression ast, TArg arg);
TResult VisitIfExpression(IfExpression ast, TArg arg);
TResult VisitIntegerExpression(IntegerExpression ast, TArg arg);
TResult VisitLetExpression(LetExpression ast, TArg arg);
TResult VisitUnaryExpression(UnaryExpression ast, TArg arg);
TResult VisitVnameExpression(VnameExpression ast, TArg arg);
```

- the return type TResult and the parameters TArg are generic types.
- We need to use this approach as there are a number of things we could be returning , eg different phrases or Void
- Similarly for the parameters, later we will use the visitor pattern for code gen and need this flexibility.
- The actual types are worked out at run time

IN THE AST CLASSES

- The AST Class LetExpression then would look like this

```
public class LetExpression : Expression
    {
        Declaration _declaration;

        Expression _expression;

        public LetExpression(Declaration declaration, Expression expression, SourcePosition position)
        : base(position)
        {
            _declaration = declaration;
            _expression = expression;
        }

        public Declaration Declaration { get { return _declaration; } }

        public Expression Expression { get { return _expression; } }

        public override TResult Visit<TArg, TResult>(IExpressionVisitor<TArg, TResult> visitor, TArg arg)
        {
            return visitor.VisitLetExpression(this, arg);
        }
}
```

VISIT METHOD

- Our visit method uses the generic types as well
- each visit method receives a instance of visitor that implements the visitor interface for that type of phrase
- in this case IExpressionVisitor

```
public override TResult Visit<TArg, TResult>
                (IExpressionVisitor<TArg, TResult> visitor, TArg arg)
{
        return visitor.VisitLetExpression(this, arg);
}
```

APPLYING THE PATTERN

- For every production in our abstract syntax we will have a visitor method e.g:
  - public TypeDenoter VisitCallExpression(CallExpression ast, Void arg)
  - public Void VisitIfCommand(IfCommand ast, Void arg)
  - public TypeDenoter VisitIntegerLiteral(IntegerLiteral literal, Void arg)
- In every class within our AST we will have a Visit method that allows the visitor to visit that point in the AST e.g
  - public override TResult Visit<TArg, TResult>(IExpressionVisitor<TArg, TResult> visitor, TArg arg)
  - public override TResult Visit<TArg, TResult>(ICommandVisitor<TArg, TResult> visitor, TArg arg)

THE CHECKER CLASS

- to get all of this to work we create a class that implements our visitor interfaces
- We can call this the Checker
- It will actually implement our semantic rules (as we saw a couple of slides ago, with the canst and var declarations)
- Similar to our parser we will use partial classes to structure the Checker

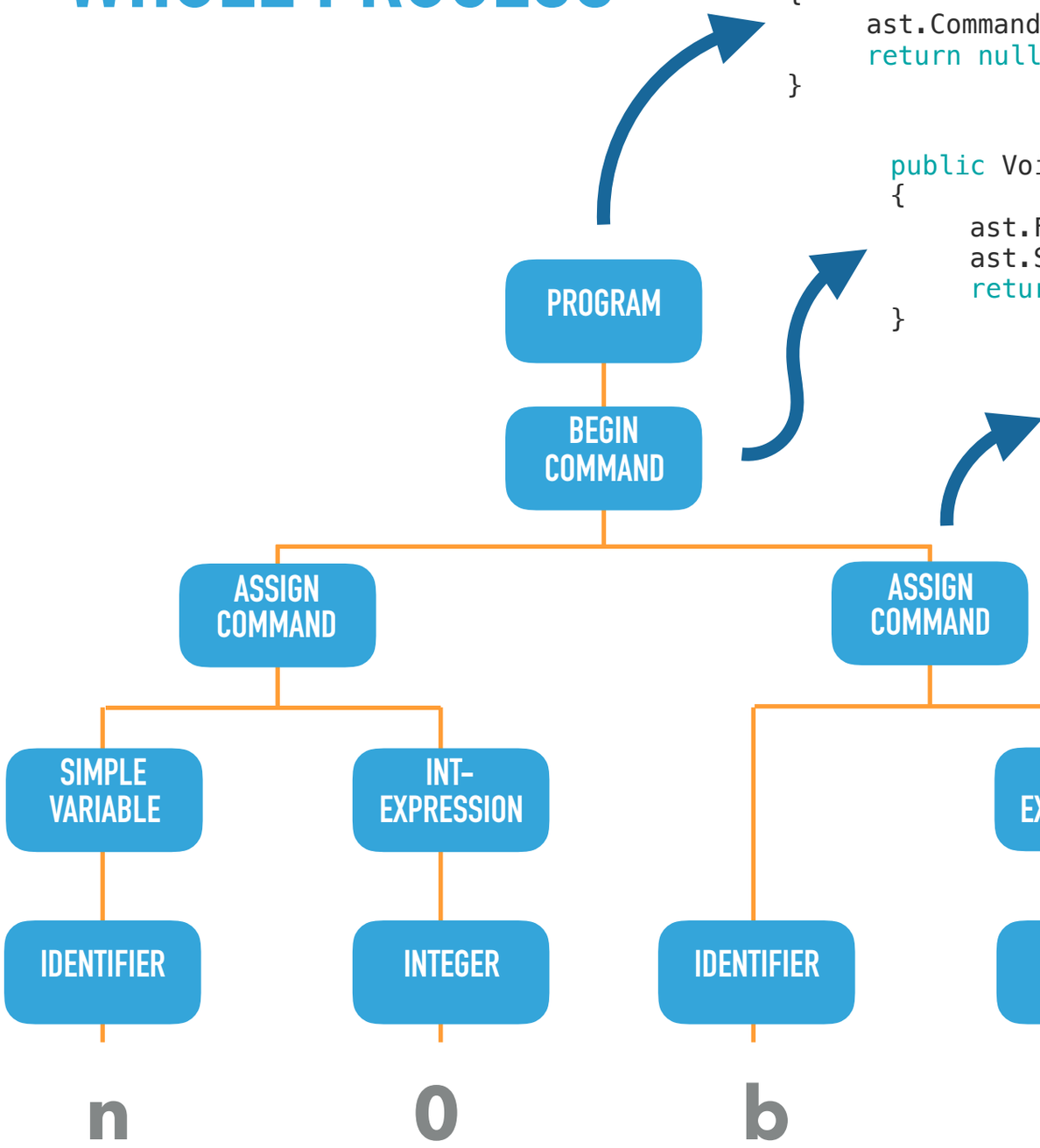
CHECKER STRUCTURE

- Each partial class will implement the visitor interface foe that part of the language e.g

```
public partial class Checker : IProgramVisitor
public partial class Checker : ICommandVisitor
public partial class Checker : IDeclarationVisitor
```

- Using this structure ensures that the checker has the required methods for our language and that we are dealing with the semantic rules
- More on this in the this weeks lab

WHOLE PROCESS




```
public Void VisitProgram(Program ast, Void arg)
{
     ast.Command.Visit(this, null);
     return null;
}
```

```
public Void VisitSequentialCommand(SequentialCommand ast, Void arg)
{
     ast.FirstCommand.Visit(this);
     ast.SecondCommand.Visit(this);
     return null;
}
```

```
public Void VisitAssignCommand(AssignCommand ast, Void arg) {
var vnameType = ast.Vname.Visit(this);
var expressionType = ast.Expression.Visit(this);
CheckAndReportError(ast.Vname.IsVariable, "LHS of assignment is not a
variable",ast.Vname);
CheckAndReportError(expressionType.Equals(vnameType), "assignment
incompatibilty", ast);
return null; }
```

```
public TypeDenoter VisitIntegerExpression(IntegerExpression ast, Void arg)
{
   return ast.Type = StandardEnvironment.IntegerType;
}
```

Finally returns the type!

VNAME – IDENTIFIER COLLISION TO ENSURE AST

- Vname ParseVname()
  {
  var identifier = ParseIdentifier();
  return ParseRestOfVname(identifier);
  }

Called from the parser when we know we have a Name

-

Vname ParseRestOfVname(Identifier firstIdentifier)
{
var startLocation = firstIdentifier.Start;
Vname vname = new SimpleVname(firstIdentifier, firstIdentifier.Position);
return vname;
}

Called if we calledParseIdentifier before being sure we had a name

SUMMARY

- The **Visitor Pattern** lets us visit each of the nodes of our **Abstract Syntax Tree**
- As we visit each node we look at identifying the identifiers and filling the symbol table
- We also check the Types of our expressions
- The visitor patterns leat us do this without polluting the actual objects.