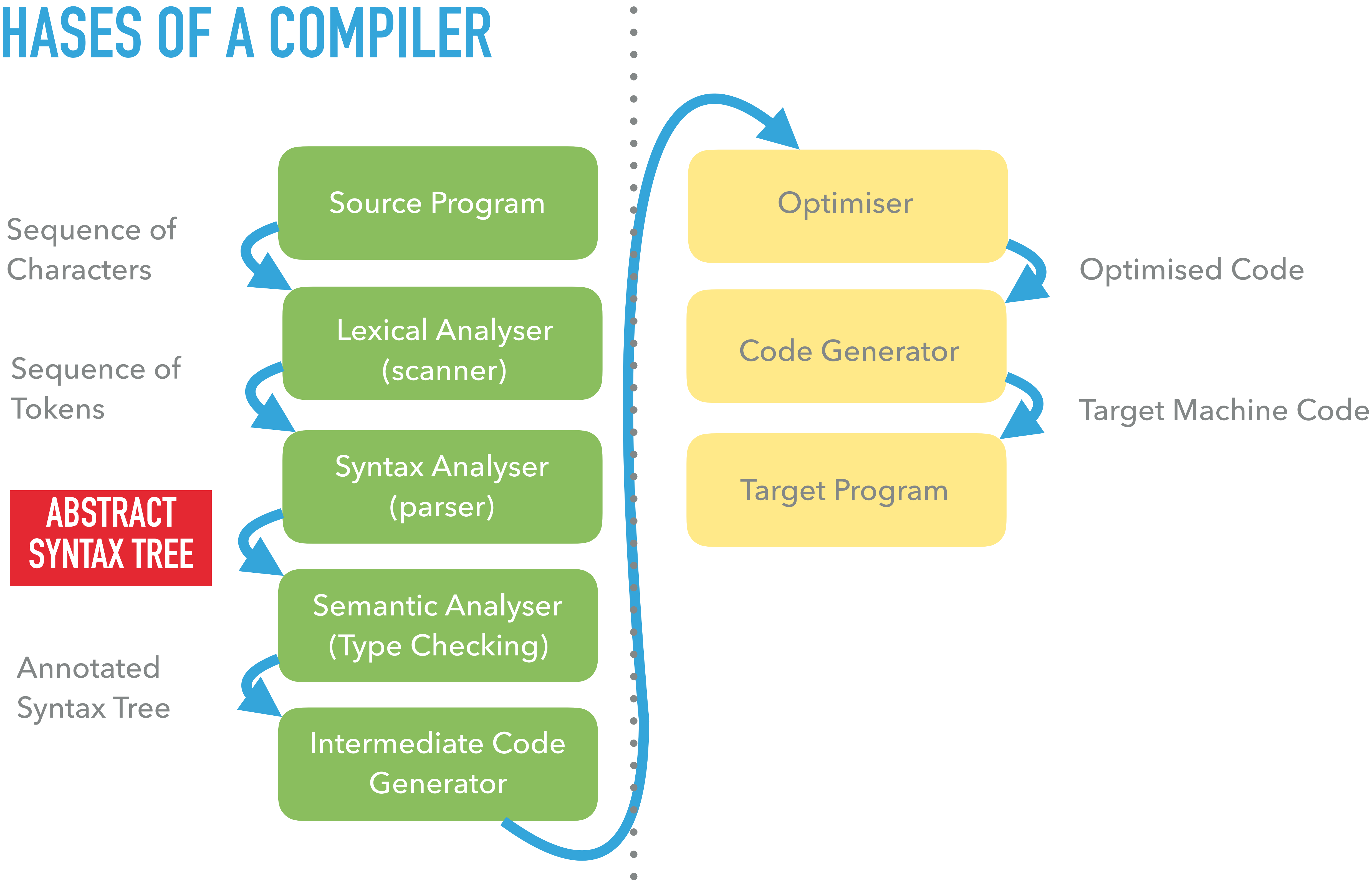
CM4106 - LANGUAGES AND COMPILERS

GENERATING SYNTAX TREES

THIS WEEK

- Syntax Trees - recap
- Creating trees in the parser
- Tree objects
- Syntax Tree as a Class Diagram

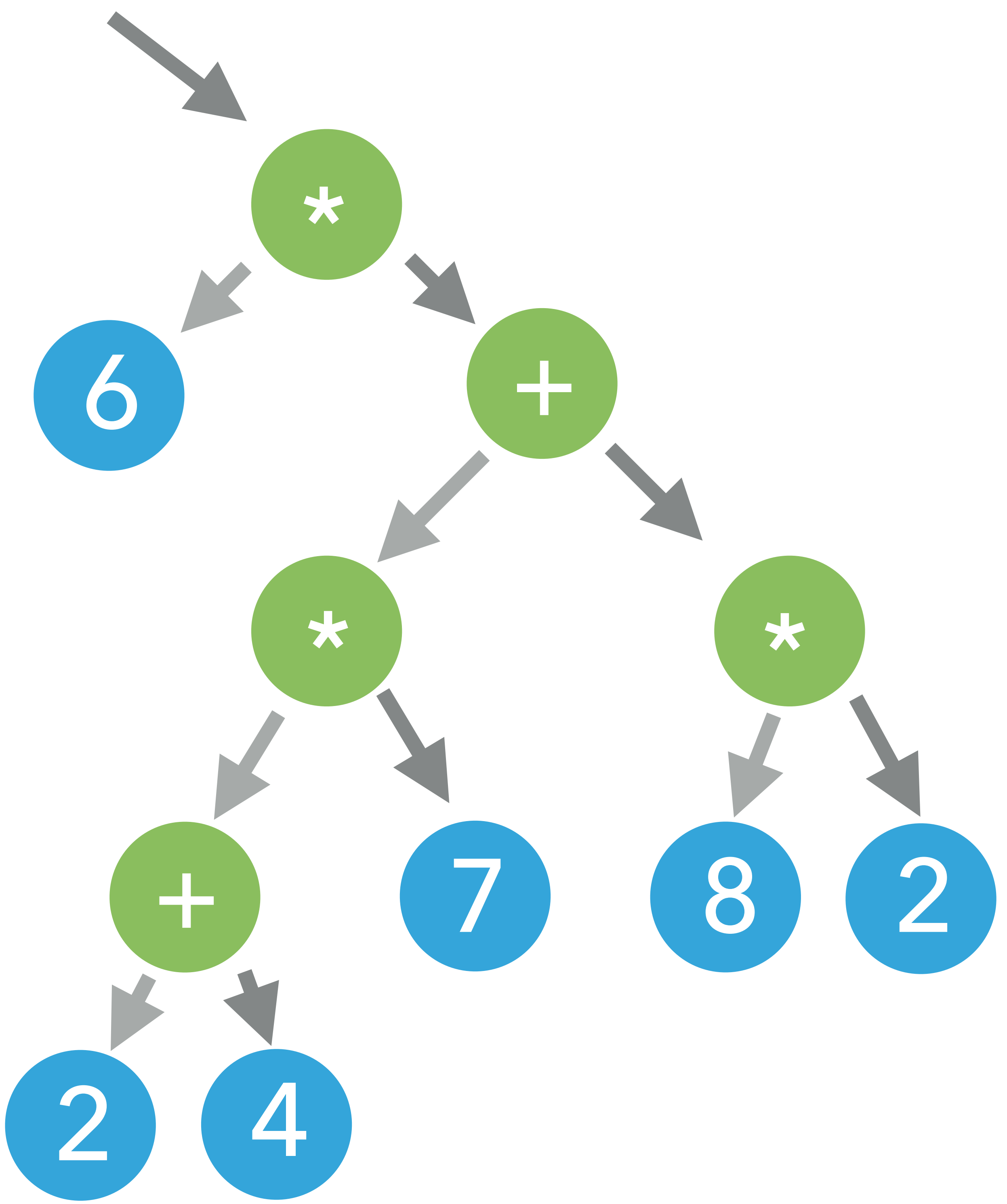
PHASES OF A COMPILER



RECAP

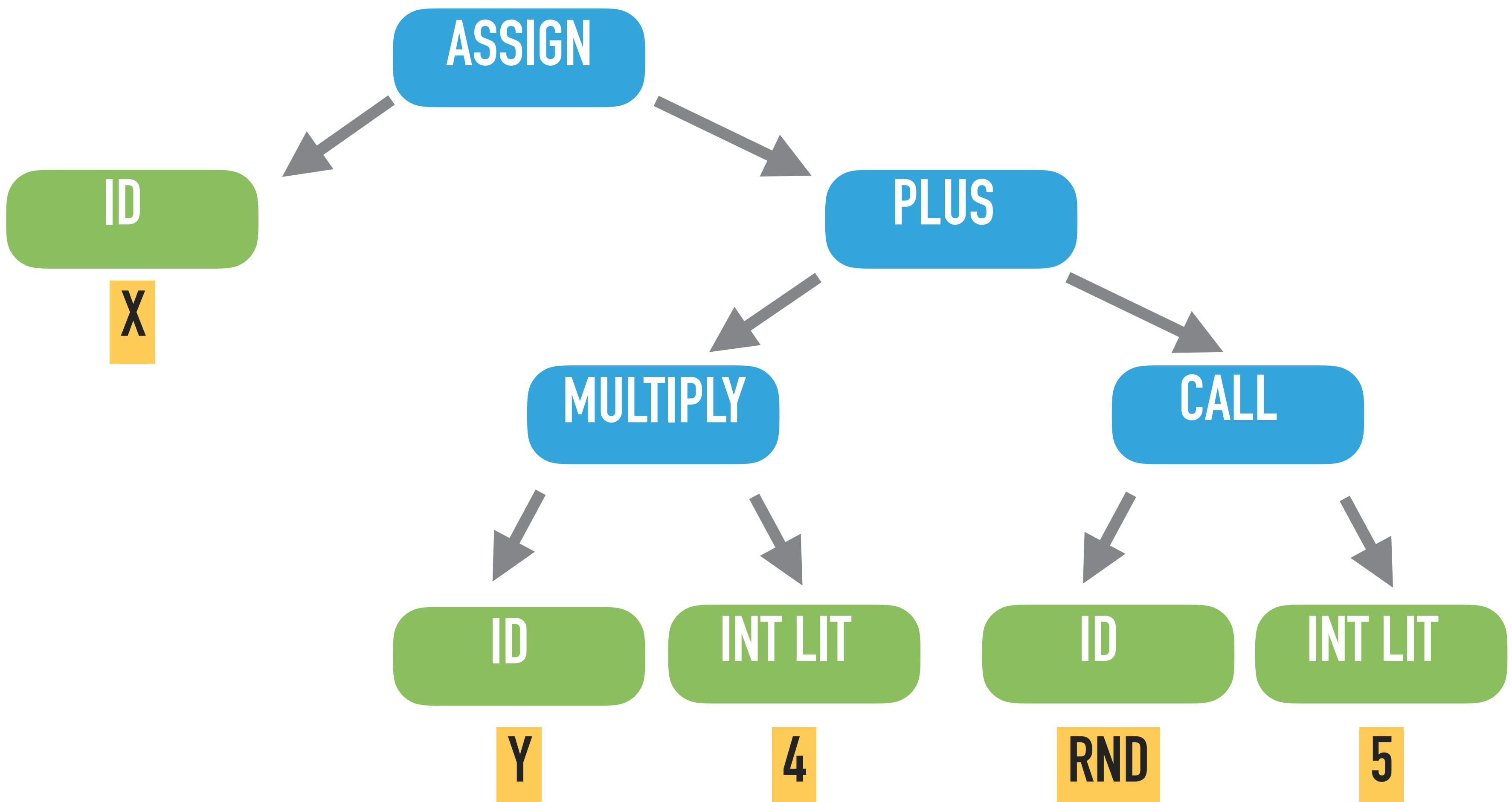
- Way back in week 2 we looked at a simple syntax trees
- We even worked one out by hand for a simple equation
- then we saw how the same process could be used in a language.

(6 * (((2+4) * 7) + (8*2)))



LANGUAGE SPECIFIC

x = 4 * y + rnd(a);



GRAMMARS AND CONCRETE SYNTAX

- In our grammar, every production rule defines how that phrase should be constructed in the language

singleCommand ::= begin Command end

- the grammar tells us the order that these phrases must come in the language

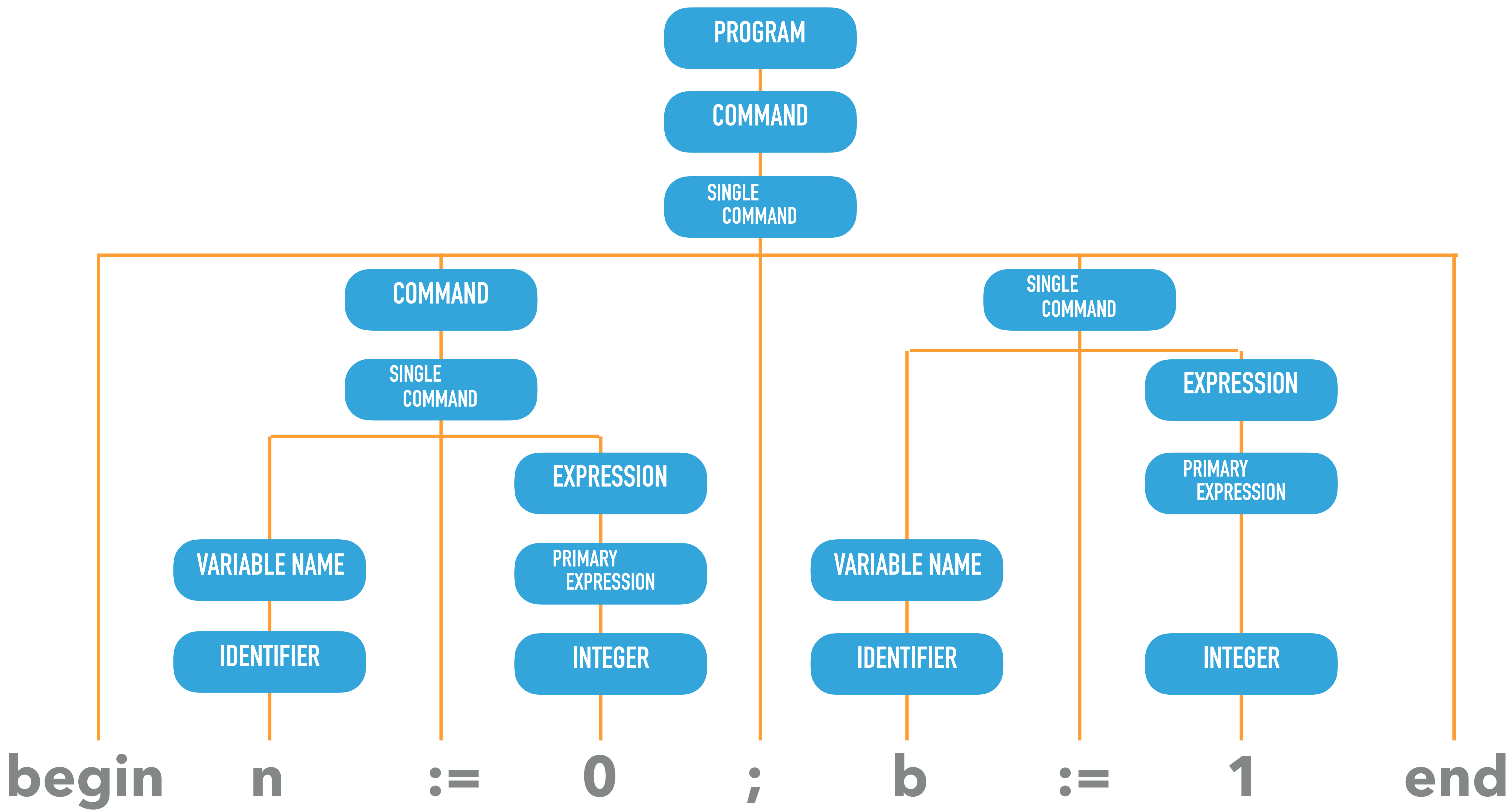
Command ::= singleCommand(;singleCommand)*

- This is called the **concrete syntax**, the grammar defines exactly how the source language should be written

CONCRETE SYNTAX TREE

- The tree below is the concrete syntax tree for

begin n:=0; b:=1 end



SYNTAX VS SEMANTICS

- When we start to think about semantics, i.e once the language is parsed, the syntax becomes less important.
- for example **X > 10** and **10 < X** the syntax is different

identifier operator int-lit vs **int-lit operator identifier**

- but the semantics are the same, the expression means exactly the same thing. Is the value of x less than 10

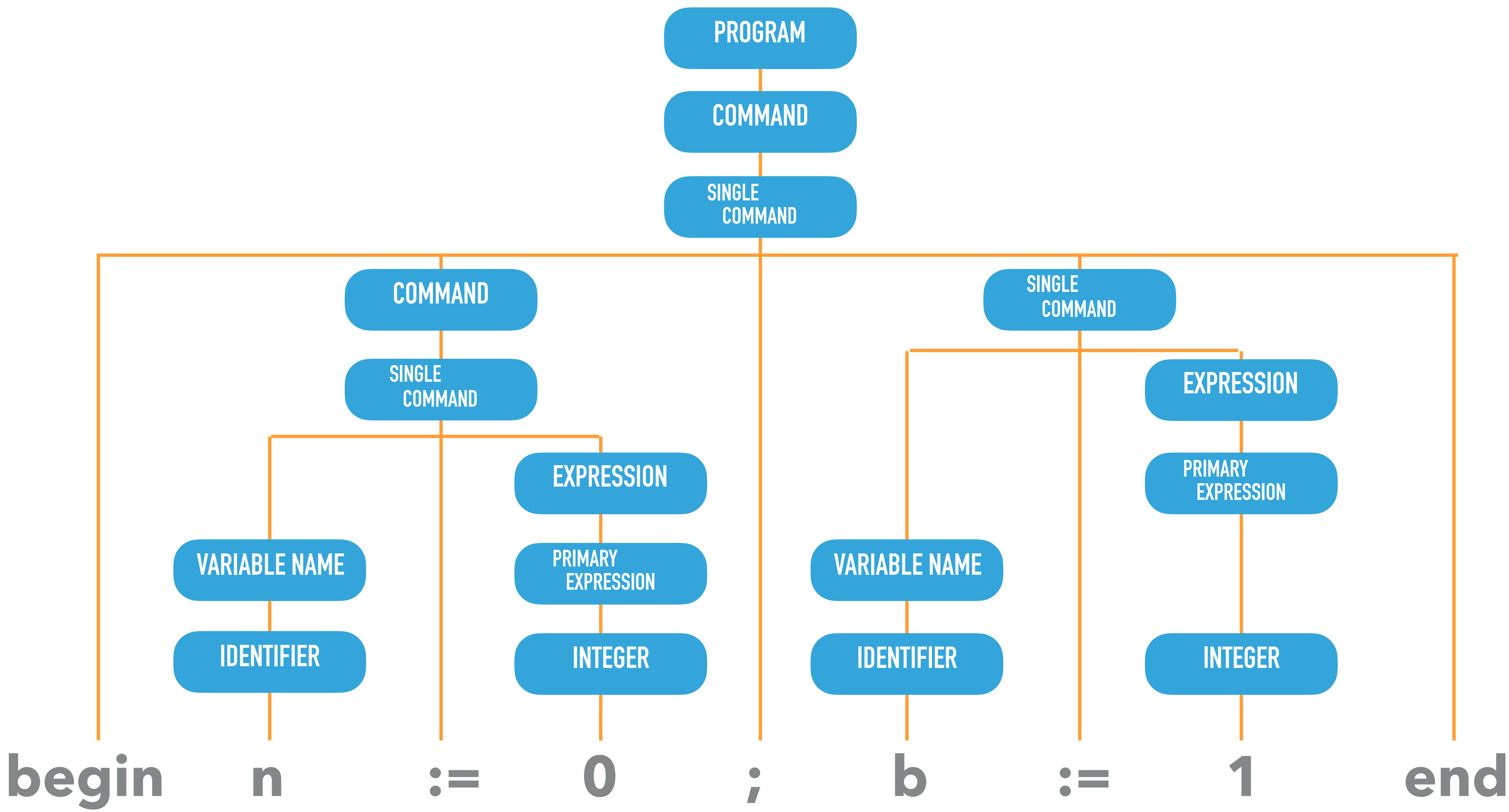
PARSER & SYNTAX

- Our parser checks the concrete syntax for us.
- Our parser has already identified what production rule we are in. eg singleCommand, singleDeclaration etc
- So when we pass on our syntax tree to the semantic analysis phase we don't need to keep track of this anymore, we only need to identify what type of phrase we are dealing with
- This allows us to reduce the tree substantially.

CONCRETE SYNTAX TREE

- The tree below is the concrete syntax tree for

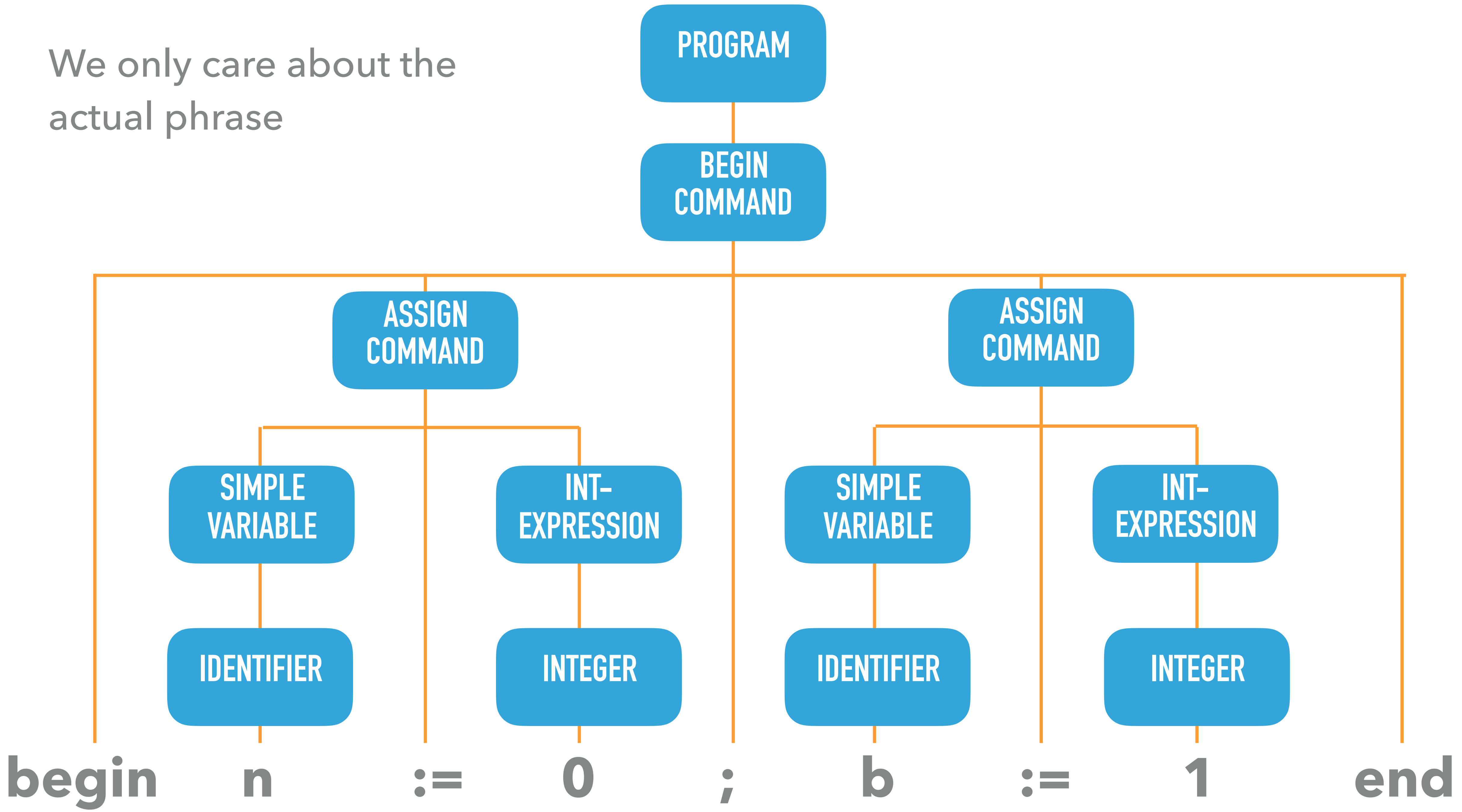
begin n:=0; b:=1 end



REDUCING THE PRODUCTIONS

begin n:=0; b:=1 end

We only care about the actual phrase



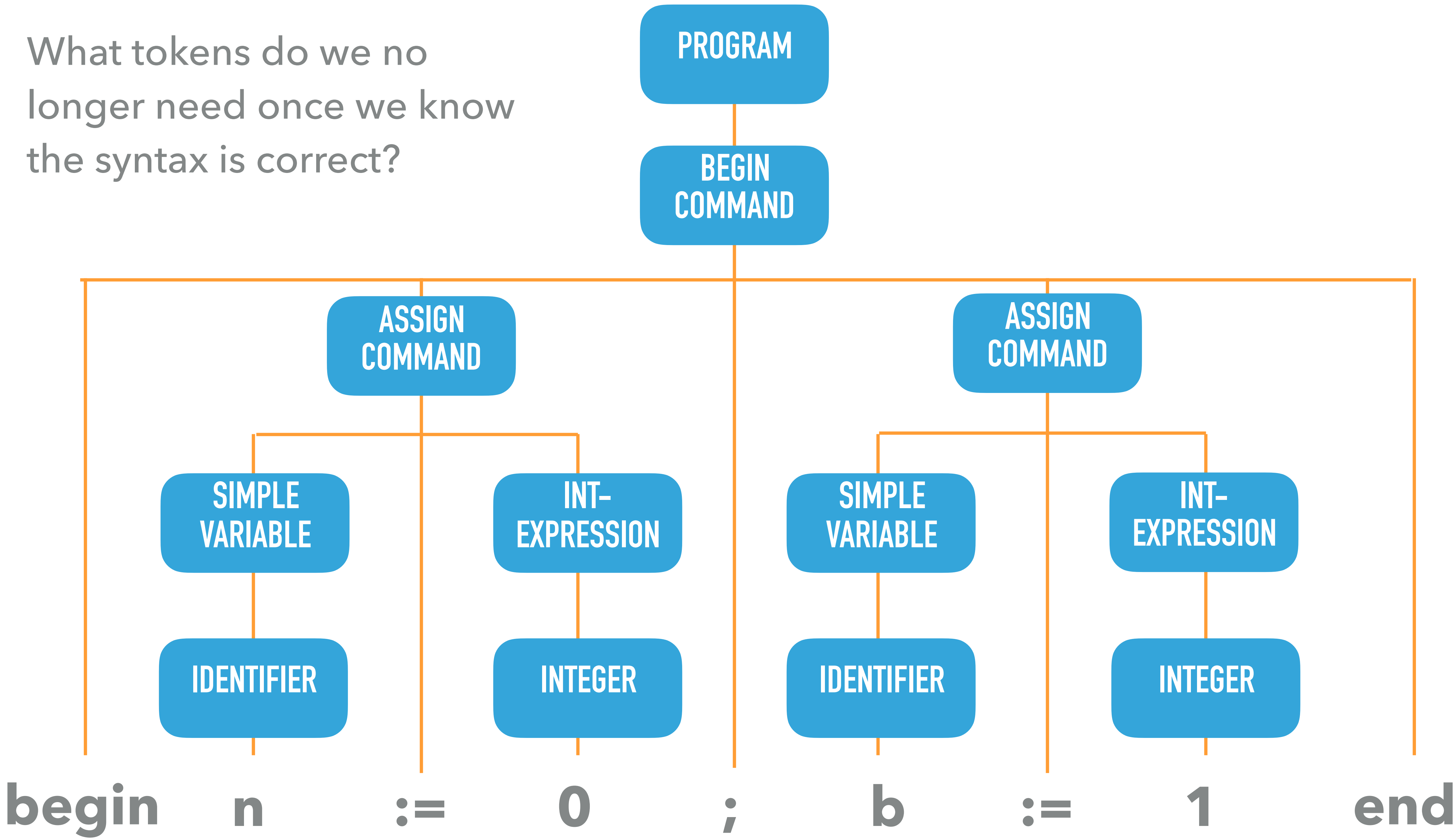
REDUCING TREE PRODUCTIONS

- We group the productions into sensible phrases
- in this case our Program is composed of a BeginCommand
- Our Command is composed of two Assignment Commands
- Our Assignments are composed of simple variable integer expressions.
- We are starting to apply meaning to the source code

REDUNDANT TOKENS

begin n:=0; b:=1 end

What tokens do we no longer need once we know the syntax is correct?



WHAT HAVE WE DONE?

- We know that if we have a successful Begin command from the parser

singleCommand ::= begin Command end

  - so we know that it must have started with a **Begin** token ended with an **End** token
  - All we need to know for semantic analysis is that we have a BeginsCommand

FOR AN ASSIGNMENT STATEMENT

- Again for a successful parse we must have the following format

singleCommand ::= Vname := Expression

Vname ::= Identifier **Primary-Expression ::= Int-Lit**

  - so we know that it must have started with a **Identifier** ,have a **Becomes** (:=) token and end with an **Int-Literal**
  - All we need to know for semantic analysis the values of these parts.
  - an assignment is always something = something we we can even ignore the :=

ABSTRACT VS CONCRETE SYNTAX

- the **concrete syntax** is used to make the **parser**
- We have already done this

NON-TERMINALS

PRODUCTIONS

Program	::=	Command
Command	::=	Single-Command (;single-Command)*
Single-Command	::=	V-name (:=ExpressionI(Expression))
	I	**if** Expression **then** Single-Command **else** Single-Command
	I	**while** Expression **do** Single-Command
	I	**let** Declaration **in** Single-Command
	I	**begin** Command **end**
Expression	::=	Secondary-Expression
	I	**let** Declaration **in** Expression
	I	**if** Expression **then** Expression **else** Expression
Secondary-Expression	::=	Primary-Expression
	I	Secondary-Expression Operator Primary-Expression
Primary-Expression	::=	Integer-Literal
	I	Character-Literal
	I	V-name
	I	Identifier (Actual-Parameter-Sequence)
	I	Operator Primary-Expression
	I	(Expression)
V-name	::=	Identifier
Declaration	::=	Single-Declaration (; Single-Declaration)*
Single-Declaration	::=	**const** Identifier ~ Expression
	I	**var** Identifier : Type-Denoter
Actual-Parameter-Sequence	:==	(Actual-Parameter (,Actual-Parameter)*)
Actual-Parameter	::=	Expression
	I	var V-name
Type-denoter	::=	Identifier

ABSTRACT VS CONCRETE SYNTAX

- the **abstract syntax** is used to make the Abstract Syntax Tree
- Its a cut down version of the concrete syntax that describes the phrases we have found in the parser

NON-TERMINALS

Program	::=	Command
Command	::=	V-name := Expression
	I	V-name (Expression)
	I	Command ; Command
	I	**if** Expression **then** Single-Command
		else Single-Command
	I	**while** Expression **do** Single-Command
	I	**let** Declaration **in** Single-Command
	I	**begin** Command **end**
Expression	::=	**let** Declaration **in** Expression
	I	**if** Expression **then** Expression **else** Expression
	I	Expression Operator Expression
	I	Integer-Literal
	I	Character-Literal
	I	V-name
	I	Identifier (Actual-Parameter-Sequence)
	I	Operator Primary-Expression
	I	(Expression)
V-name	::=	Identifier
Declaration	::=	Declaration (; Declaration)*
	I	**const** Identifier ~ Expression
	I	**var** Identifier : Type-Denoter
Actual-Parameter-Sequence	:==	(Actual-Parameter (,Actual-Parameter)*)
Actual-Parameter	::=	Expression
	I	var V-name
Type-denoter	::=	Identifier

PRODUCTIONS

DESCRIBING THE PRODUCTIONS

Program ::= Command

- how would we describe this?
  - Only one production rule, so lets just call it Program

DESCRIBING THE PRODUCTIONS

Command	::=	V-name :=Expression	AssignCommand
	|	V-name (Expression)	CallCommand
	|	Command ; Command	SequentialCommand
	|	if Expression then Single-Command else Single-Command	IfCommand
	|	while Expression do Single-Command	WhileCommand
	|	let Declaration in Single-Command	LetCommand
	|	begin Command end	BeginCommand

- We can define all the types of productions

EXPRESSIONS

Expression	::=	let Declaration in Expression	→ LetExpression
	|	if Expression then Expression else Expression	→ IfExpression
	|	Expression Operator Expression	→ BinaryExpression
	|	Integer-Literal	→ IntegerExpression
	|	Character-Literal	→ CharacterExpression
	|	V-name	→ VnameExpression
	|	Identifier (Actual-Parameter-Sequence)	→ CallExpresson
	|	Operator Expression	→ UnaryExpression
	|	(Expression)	→ Expression

- Again this is clear, apart from the bracketed expression, in semantic terms this makes no difference
- e.g x = 2 is the same as (x = 2)

DECLARATIONS

Declaration	::=	Declaration (; Declaration)* →	SequentialDeclaration
	|	**const** Identifier **~** Expression →	ConstDeclaration
	|	**var** Identifier **:** Type-Denoter →	VarDeclaration

- Similar Process again, and for the rest of our productions.

WHAT DOES THIS MEAN

- Every source program will have its own AbstractSyntaxTree that defines the order of the phrases in that program
- When the Parser successfully parses a phrase in the program we know we can build up the tree using the identified phrases
- eg Program-> BeginCommand->AssignCommand… etc..etc

IMPLEMENTING THIS IN CODE

- Up until now all you have done, and all you need to do for the first coursework, is print out the method calls from your parser.
- But how would we generate the Abstract Syntax Tree from our code?

OBJECTS AS THE SYNTAX TREE

- We can use objects to represent the syntax tree
- We create a class for each production in our Abstract Grammar
- Each instance of that class will itself have instances of classes representing the constituent productions
- eg class Program will have an instance of class Command

ABSTRACT VS CONCRETE SYNTAX

- the **abstract syntax** is used to make the Abstract Syntax Tree
- For each of the productions here we have a class that represents that part of the tree

NON-TERMINALS

Program	::=	Command
Command	::=	V-name := Expression
	|	V-name (Expression)
	|	Command ; Command
	|	if Expression then Single-Command
		else Single-Command
	|	while Expression do Single-Command
	|	let Declaration in Single-Command
	|	begin Command end
Expression	::=	let Declaration in Expression
	|	if Expression then Expression else Expression
	|	Expression Operator Expression
	|	Integer-Literal
	|	Character-Literal
	|	V-name
	|	Identifier (Actual-Parameter-Sequence)
	|	Operator Primary-Expression
	|	(Expression)
V-name	::=	Identifier
Declaration	::=	Declaration (; Declaration)*
	|	const Identifier ~ Expression
	|	var Identifier : Type-Denoter
Actual-Parameter-Sequence	:==	(Actual-Parameter (,Actual-Parameter)*)
Actual-Parameter	::=	Expression
	|	var V-name
Type-denoter	::=	Identifier

PRODUCTIONS

PROGRAM

```
public void ParseProgram()
{
   _currentToken = _tokens.Current;
   ParseCommand();
}
```

- Lets look at our ParseProgram in our Parser
- Currently it will just call the ParseCommand method to deal with the Command the Program must be composed of. **Program ::= Command**

PROGRAM SYNTAX TREE

```
public class Program {
    Command _command;

    public Program(Command command)
    {
      command = command;
    }

    public Command Command { get { return _command; } }
}
```

- Our program class represents our Program
- in our Abstract Grammar a program only contains a command so thats all we need in here.

NEW PROGRAM PARSER

Our method now returns a program

```
public Program ParseProgram()
{
   _currentToken = _tokens.Current;

   var command = ParseCommand();
   var program = new Program(command);

   return program;
}
```

We get a command object from the ParseCommand method

And use this to create a program

- Now if we update our parser to use our program object
- Whenever a Program is successfully parsed, our parser will create an instance of the Program class, containing an instance of the Command class.

LET EXPRESSION EXAMPLE

```
case TokenKind.Let:
{
   AcceptIt();
   ParseDeclaration();
   Accept(TokenKind.In);
   ParseExpression();
}
```

Use AcceptIt to consume “Let” Token

ParseDeclaration

Accept IN Token

ParseExpression

- At the minute your parser code for the LET Expression looks a bit like this.

LET EXPRESSION EXAMPLE

```
case TokenKind.Let:
{
   AcceptIt();
   var declaration = ParseDeclaration();
   Accept(TokenKind.In);
   var expression = ParseExpression();
   return new LetExpression(declaration, expression);
}
```

Use AcceptIt to consume “Let” Token

Get a declaration object from ParseDeclaration

Accept IN Token

Get a expression object from ParseExpression

Finally create a LetExpression using the declaration and expression objects

- We Know a LetExpression needs a declaration and an expression
- So we create these from the parseDeclaration and parseExpression methods.

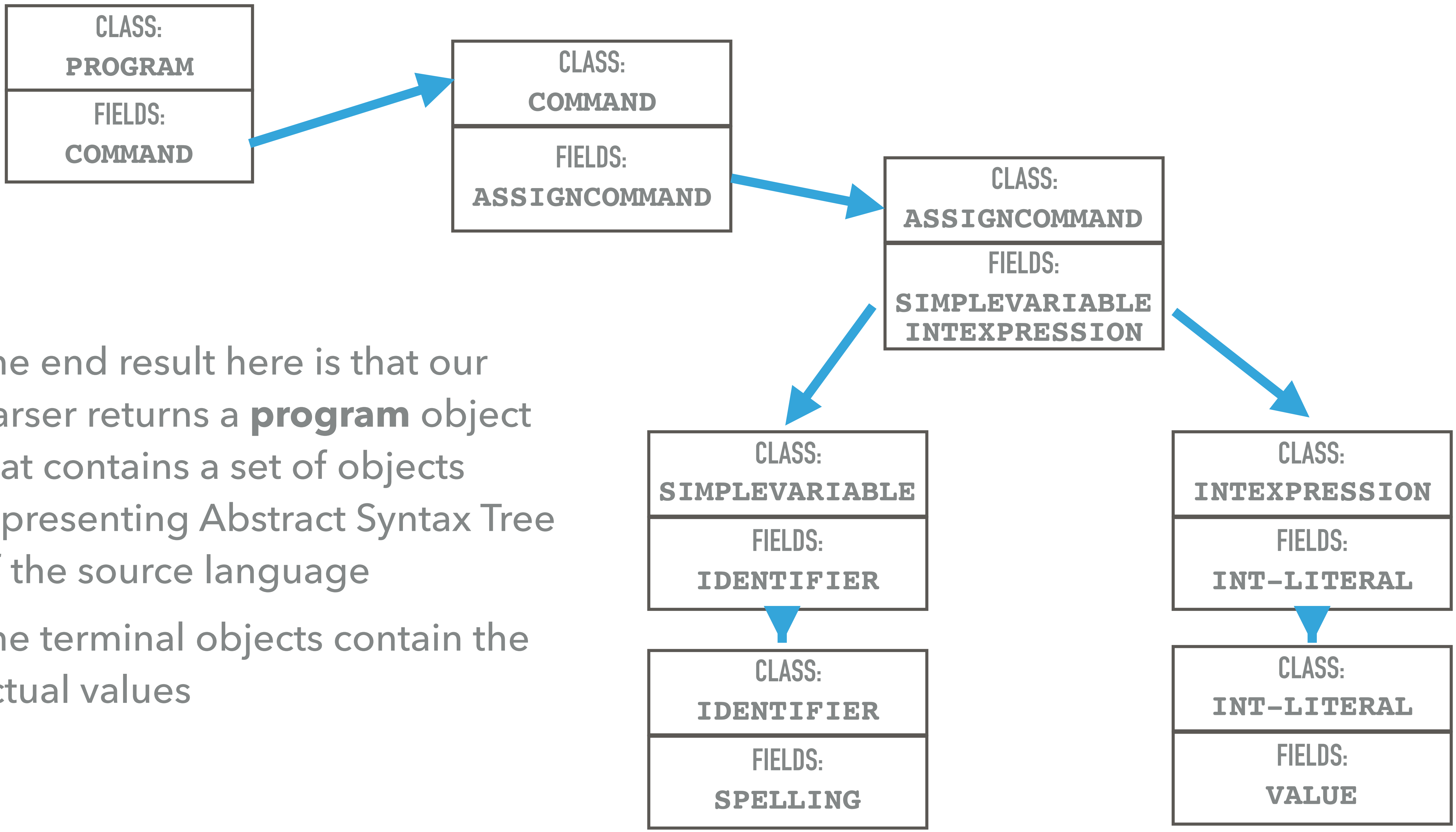
WHAT DOES A LET EXPRESSION LOOK LIKE?

```
public class LetExpression
    {
        Declaration _declaration;

        Expression _expression;

        public LetExpression(Declaration declaration, Expression expression)
        {
            _declaration = declaration;
            _expression = expression;
        }

        public Declaration Declaration { get { return _declaration; } }

        public Expression Expression { get { return _expression; } }

}
```

- Simple class that fits our Abstract Grammar for the LET command **Expression ::= let Declaration in Expression**

ABSTRACT SYNTAX TREE FOR OUR PROGRAM

begin n:=1 end



The end result here is that our parser returns a **program** object that contains a set of objects representing Abstract Syntax Tree of the source language

The terminal objects contain the actual values

SUMMARY

- Concrete Syntax is used to create the parser
  - it defines the production rules and for recursive descent must be LL(1)
- Abstract Syntax is a condensed Grammar used to create the Abstract Syntax Tree (AST)
- Each production rule in the AST needs a class to represent it

LAB THIS WEEK

- The lab this week is tow work on finalising your coursework due on Monday
- I will work my way round and give you an idea of how you are getting on and answer any questions.
- YOU DO NOT NEED TO IMPLEMENT THE CONTENT OF THIS LECTURE IN YOUR CODE…..yet