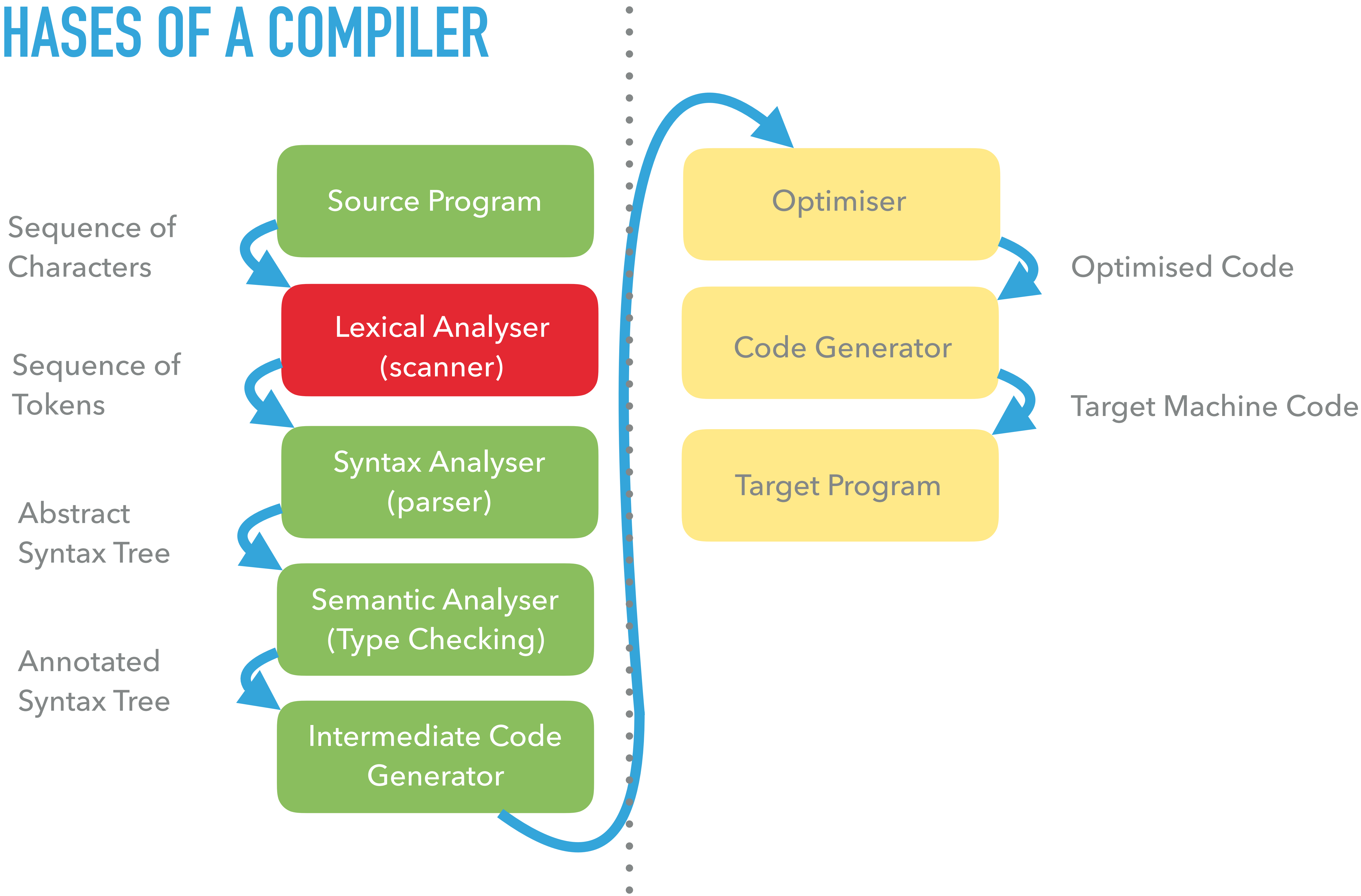
CM4106 LANGUAGES AND COMPILERS

THE SCANNER

THIS WEEK

- We will look at the first stage in the compilation process
- How languages are defined
- How we build a scanner to provide the TOKENS that form a language

PHASES OF A COMPILER



THE SCANNER

- INPUT: takes characters from source code
- OUTPUT : a sequence of tokens
- What it does:
  - groups characters into recognised tokens
  - Identify and ignore whitespace
- Performs Error Checking :
  - eg bad characters
  - Unterminated strings “BigBaddaBoom

LANGUAGE DEFINITIONS

- To be able to analyse a source file we need to understand the language it it written in.
- Languages have rules, Grammars, that define how they are written.
- We can use these rules to know how to process the language
- The Grammar specifies the **SYNTAX** of the language

EXAMPLE

- A Phone Number = (01224) 262789
- An area code - 5 numbers surrounded by brackets
- then 6 numbers
- Ok so
- a Phone number = an area code + 6 numbers
- an area code = “(“+5 numbers+”)”
- a number = 0,1,2,3,4,5,6,7,8,9

TERMINAL & NON-TERMINAL SYMBOLS

a Phone number = an area code + 6 numbers

an area code = (+5 numbers+)

a number = 0,1,2,3,4,5,6,7,8,9

Non-Terminal Symbols
(something that needs further description)

Terminal Symbols
(The basic building blocks of the language)

BNF (BACKUS-NAUR FORM)

- With the proliferation of languages during the 50’s and 60’s there was a need to develop some way of describing what was going on.
- A notation that didn’t use weird characters and was it’s self capable of being read by both humans and computers
- Backus - Naur Form was created and can be used to express the Grammar of a language

BNF SPECIFICATION

In BNF, nonterminals are enclosed in angle brackets: **<digit>**.

Terminals are not.

If it is not clear which symbols are terminals, then terminals may be enclosed in quotes.

Our "=" is replaced by "**::=**"

vertical bars (**|**) are used to show options.

BNF PHONE NUMBER

```
<phone-number> ::= <area-code> <6-dig>

<area-code> ::= "(" <digit> <digit> <digit> <digit> <digit> “)"

<6-dig> ::= <digit> <digit> <digit> <digit> <digit> <digit>

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

EBNF - EXTENDED BNF

```
phone-number = [area-code] 6-dig

area-code = "(" digit digit digit digit digit ")"

6-dig = digit digit digit digit digit digit>

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Braces `{ ... }` surround sections that are optional and repeatable (like "`*`" in regular-expression syntax).

Brackets `[ ... ]` surround sections that are optional and not repeatable (like the "`?`" shortcut).

We can use parentheses `( ... )` for grouping.

MINI TRIANGLE LANGUAGE

Triangle is a small, but realistic, Pascal-like language with let-in constructs for local declarations.

```
!this is a daft program
let
 const MAX ~ 10;
 var n: Integer
in begin
 getint(var n);
 if (n>0) /\ (n<=MAX) then
  while n > 0 do begin
   putint(n); puteol();
   n := n-1
  end
 else
end
```

Comment

Let-in construct for local variables

Sequence of commands can be grouped with begin-end

Else is required, but may be empty

TRIANGLE DEFINITION

```
Program          ::= single-Command
Command          ::= single-Command
                   | Command ; single-Command
single-Command   ::= skip
                   | V-name := Expression
                   | Identifier ( Expression )
                   | if Expression then single-Command else single Command
                   | while Expression do single-Command
                   | let Declaration in single-Command
                   | begin Command end
```

Expression	::=	primary-Expression
	|	Expression Operator primary-Expression
primary-Expression	::=	Integer-Literal
	|	V-name
	|	Operator primary-Expression
	|	(Expression)

```
V-name              ::= Identifier
Declaration         ::= single-Declaration
                      | Declaration ; single-Declaration

single-Declaration  ::= const Identifier ~ Expression
                      | var Identifier : Type-denoter

Type-denoter        ::= Identifier
Operator            ::= + | - | * | / | < | > | = | \
Identifier          ::= Letter | (Letter | Digit)*
Integer-Literal     ::= Digit | Digit *
Comment             ::= ! Graphic* eol
```

HOW DOES THIS RELATE TO THE SCANNER?

- What does the scanner actually need to do
- The scanner needs to identify the basic components of the language.
- These are the language TOKENS

```
Identifier ::= Letter (Letter|Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
Comment ::= ! Graphic* eol
```

TOKENS

- The Interface between the scanner and the parser.
- A TOKEN is a basic symbol of the source program
- It May consist of several characters - but each character may not have any special significance
- E.e L,E, T has no significance but “LET” does.
- Tokens can be classified by type e.g “Y” and “Integer” are Identifiers “5” is a Digit

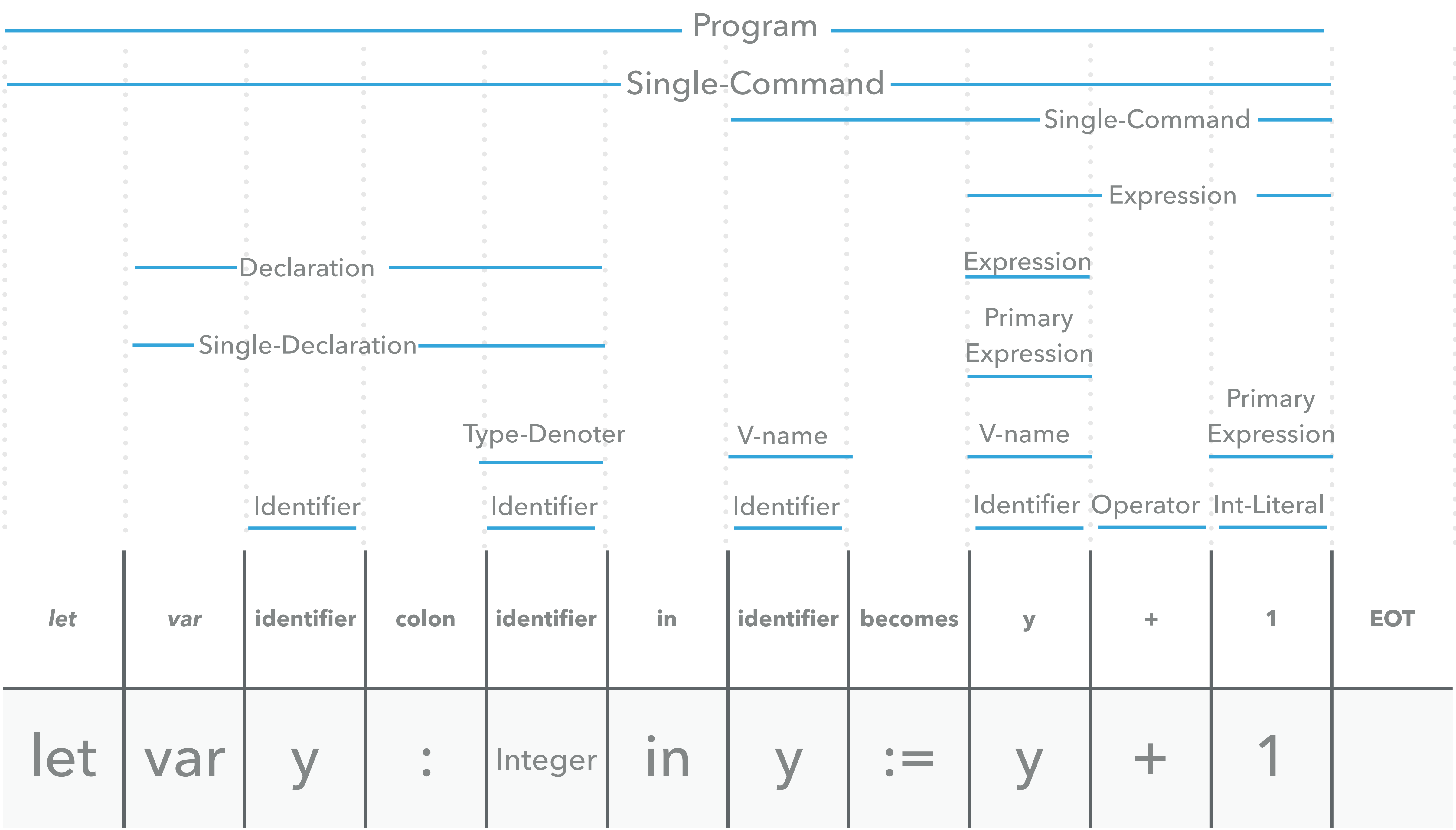
SO WHAT DOES THAT MEAN?

Very simple program in Mini-Triangle

```
let var y: Integer
in !new year
y := y+1
```

let	*var*	identifier	colon	identifier	in	identifier	becomes	identifier	operator	int-lit	EOT
let	var	y	:	Integer	in	y	:=	y	+	1	

RELATES DIRECTLY BACK TO OUR SYNTAX DEFINITION



IMPLEMENTATION

- What do we need?
  - A way of representing the TOKENS
  - A way of representing the type of possible tokens
  - A way of holding the source code
  - A way of outputting / storing the tokens to the parser

TOKEN CLASS

- Needs to represent the TOKENS
  - What the token is (spelling)
  - What type it is (e.g int-lit, identifier)
    - Later we may want to hold the location so we can report any errors

BASIC TOKEN CLASS STRUCTURE

```
public class Token {
  public TokenKind Kind { get; private set; }
  public string Spelling { get; private set; }

  public Token(TokenKind kind, string spelling)
  {
    Kind = kind;
    Spelling = spelling;
  }
  public override string ToString() {
    return string.Format("Kind={0}, spelling=\"{1}\"", Kind, Spelling);
  }
}
```

PROBLEMS?

- One issue sat the minute is that we could create an identifier that is any list of characters
- This isn't correct…. we have certain reserved words in our language
  - let, begin, while etc.
  - how do we deal with these

TOKENKIND CLASS (ENUM)

```
public enum TokenKind
{
    // literals, identifiers, operators...
    IntLiteral, Identifier, Operator,

    // reserved words - must be in alphabetical order..
    Begin, Const, Do, Else, End, If, In, Let, Then,
    Type, Var, While,

    // special tokens...
    EndOfText, Error
}
```

UPDATED TOKEN CLASS

This is a rather long winded way of making a dictionary out of our list of reserved words.

```
static readonly IDictionary<string, TokenKind> ReservedWords =
 Enumerable.Range((int)TokenKind.Array, (int)TokenKind.While).Cast<TokenKind>()
 .ToDictionary(kind => kind.ToString().ToLower(), kind => kind);
```

We can then check our dictionary to see if an identifier is a reserved word.

RESERVED WORDS

```
if (kind == TokenKind.Identifier)
{
 TokenKind match;
 if (ReservedWords.TryGetValue(spelling, out match))
 {
  Kind = match;
 }
}
```

So when we create an identifier, we search our list of reserved words

If it exists, then the user "must" have meant to use the keyword

So set the KIND of the TOKEN to the special type

SCANNER CLASS

- We now have a representation we can use to define our TOKENS How do we get them in the first place?
- The scanner has to read a source file character by character and build up the tokens.

SCANNER CLASS STRUCTURE

```
public class Scanner : IEnumerable<Token>
{
 SourceFile _source;
 StringBuilder _currentSpelling;



 public Scanner(SourceFile source)
 {
  _source = source;
  _source.Reset();
  _currentSpelling = new StringBuilder();
 }
```

YEILDING TOKENS

```
public IEnumerator<Token> GetEnumerator(){
 while (true)
 {
  while (_source.Current == '!' ||
      _source.Current == ' ' ||
     _source.Current == '\t' ||
     _source.Current == ‘\n')
  {
   ScanSeparator();
  }

 _currentSpelling.Clear();

 var kind = ScanToken();

 var token = new Token(kind, _currentSpelling.ToString());

 yield return token;

 if (token.Kind == TokenKind.EndOfText) { break; }
 }
}
```

While we have white space keep

dealing with white space

Other wise we must be in a new token

Start reading the token

Create the token from the current spelling

If we are at the end of the input, finish

TAKE IT

```
void TakeIt()
{
 _currentSpelling.Append((char)_source.Current);
 _source.MoveNext();
}
```

Simply appends the current character to the current spelling (eg the token characters we have so far)

REMOVE WHITE SPACE

```
void ScanSeparator(){
 switch (_source.Current){
  case '!':
   _source.SkipRestOfLine();
   _source.MoveNext();
   break;
  case ' ':
  case '\n':
  case '\r':
  case '\t':
   _source.MoveNext();
   break;
  }
 }
```

Switch the current character

If it's a ! we have a comment so ignore the rest of the line

If it's any other type of whitespace just move on

SCANNING TOKENS

```
TokenKind ScanToken()
{
 switch (_source.Current){
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
   case '9':
     TakeIt();
     while (IsDigit(_source.Current))
     {
       TakeIt();
     }
     return TokenKind.IntLiteral;
   }
 }

 bool IsDigit(int ch)
 {
     return '0' <= ch && ch <= '9';
 }
```

Switch the current character in the source

If it is a Digit fall through and take it

While it is still a digit keep taking characters

As soon as it is not, return what you have

Helper function elsewhere checks for you

SOURCECODE CLASS

- To keep our compiler tidy we need a helper class to deal with our sourcefile.
- Ill give you this in the labs, but there are a few methods and fields that you will need to understand

SOURCEFILE FIELDS

public string Name { get; private set; }

Returns the name of the source file - useful in error messages

public bool IsValid { get { return _source != null; } }

Returns a bool determining if the source file provided is valid

public Location Location { get { return new Location(_lineNumber, _index); } }

Return the current Location in the file (line number & position) - again thing error message

public int Current { get { return _buffer == null ? -1 : _buffer[_index]; } }

Most importantly returns the current character

SOURCEFILE METHODS

`SkipRestOfLine()`

Ignores the processing of the rest of the line

`MoveNext()`

Moves to the next character

`Reset()`

Resets the file to the beginning

COMPILER CLASS

- The files that starts it all off
- Currently it should have access to an instance of your sourcefile and scanner classes

```
public class Compiler{

  const string ObjectFileName = “obj.tam";
  SourceFile _source;
  Scanner _scanner;

  Compiler(string sourceFileName){
    _source = new SourceFile(sourceFileName);
    _scanner = new Scanner(_source);
  }

  public static void Main(string[] args)
  {
    if (args.Length != 1)
    {
      ErrorReporter.ReportMessage("Usage: Compiler.exe source");
      return;
    }

    var sourceFileName = args[0];

    if (sourceFileName != null)
    {
      var compiler = new Compiler(sourceFileName);
      var _tokens = _scanner.GetEnumerator();


    }
  }
}
```

SUMMARY

- A lot of this code I will give you so you ca start to build up a framework
- But you are going to have to flesh it out to build up your course work.
- The examples here and in the labs will be using mini-triangle, you course work will be full-triangle a slightly fuller language.