CM4106 - LANGUAGES AND COMPILERS

THE PARSER - PART2

THIS WEEK

- Recursive Descent Parsing
- LL1 - Languages
- Language Syntax vs Semantics
- Syntax to Parser
- Worked example of (reduced) Triangle

RECURSIVE DESCENT PARSING

- We saw last week how a language definition (Grammar) can be mapped on to methods to create a parser
- There were a few things happening that allow this process to work
- Part of it is to do with the Grammar

TRIANGLE REDUCED

NON-TERMINALS

Program	::=	Command
Command	::=	Single-Command (;single-Command)*
Single-Command	::=	V-name (:=Expression|(Expression))
	|	if Expression then Single-Command
		else Single-Command
	|	while Expression do Single-Command
	|	let Declaration in Single-Command
	|	begin Command end
Expression	::=	Secondary-Expression
	|	let Declaration in Expression
	|	if Expression then Expression else Expression
Secondary-Expression	::=	Primary-Expression
	|	Secondary-Expression Operator Primary-Expression
Primary-Expression	::=	Integer-Literal
	|	Character-Literal
	|	V-name
	|	Identifier (Actual-Parameter-Sequence)
	|	Operator Primary-Expression
	|	(Expression)
V-name	::=	Identifier
Declaration	::=	Single-Declaration (; Single-Declaration)*
Single-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

- What actually is a language definition or Grammar?
- A limited set of non-terminals
- Each non-terminal has a set of productions
- How do we know which production to take?

EXAMPLE `begin putint(1) end`

PROGRAM	COMMAND	SINGLE COMMAND	
COMMAND	SINGLE COMMAND	VARIABLE NAME	
IDENTIFIER	ACTUAL PARAMETER SEQUENCE	ACTUAL PARAMETER	
EXPRESSION	SECONDARY EXPRESSION	PRIMARY EXPRESSION	INTEGER

KIND=BEGIN, SPELLING="BEGIN"
KIND=IDENTIFIER, SPELLING="PUTINT"
KIND=LEFTPAREN, SPELLING="("
KIND=INTLITERAL, SPELLING="1"
KIND=RIGHTPAREN, SPELLING=")"
KIND=END, SPELLING="END"
KIND=ENDOFTEXT, SPELLING=""

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

LL(1) GRAMMARS

- L - Left to right
  - The statement is always read left to right with no backtracking.

```
begin putint(1) end
```

LL(1) GRAMMARS

- L - Left production priority
  - We try to use the left most production rule

Single-Declaration ::= **const** Identifier **~** Expression | **var** Identifier **:** Type-Denoter

Try this   Before this

Actual-Parameter ::= Expression | var V-name

Try this   Before this

LL(1) GRAMMARS

- (1)
  - One token look ahead
  - the next token, uniquely identifies the production to take

if next token in const, take this production

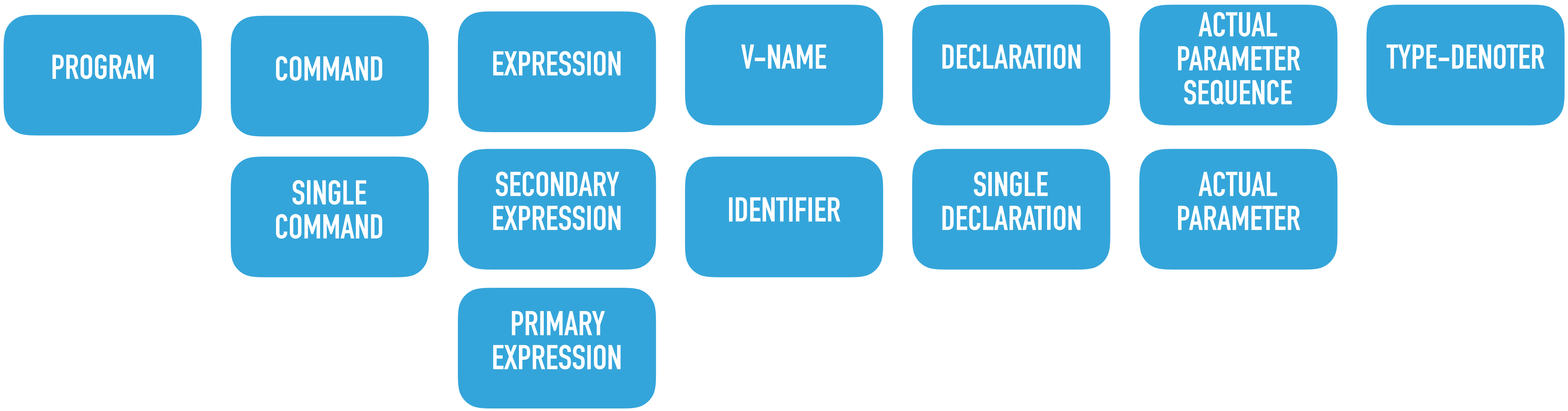
Single-Declaration ::= **const** Identifier **~** Expression
| **var** Identifier **:** Type-Denoter

if next token in var, take this production

IMPORTANCE OF LL(1) GRAMMARS

- Recursive Descent Parsing only works with this type of grammar.
- We rely on completely on not needing to backtrack the token stream
- and only needing to see the next token, to choose the next production rule

POSSIBLE STATES



- Our Grammar defines the states our parser can be in.
- This is the set of Non-Terminals

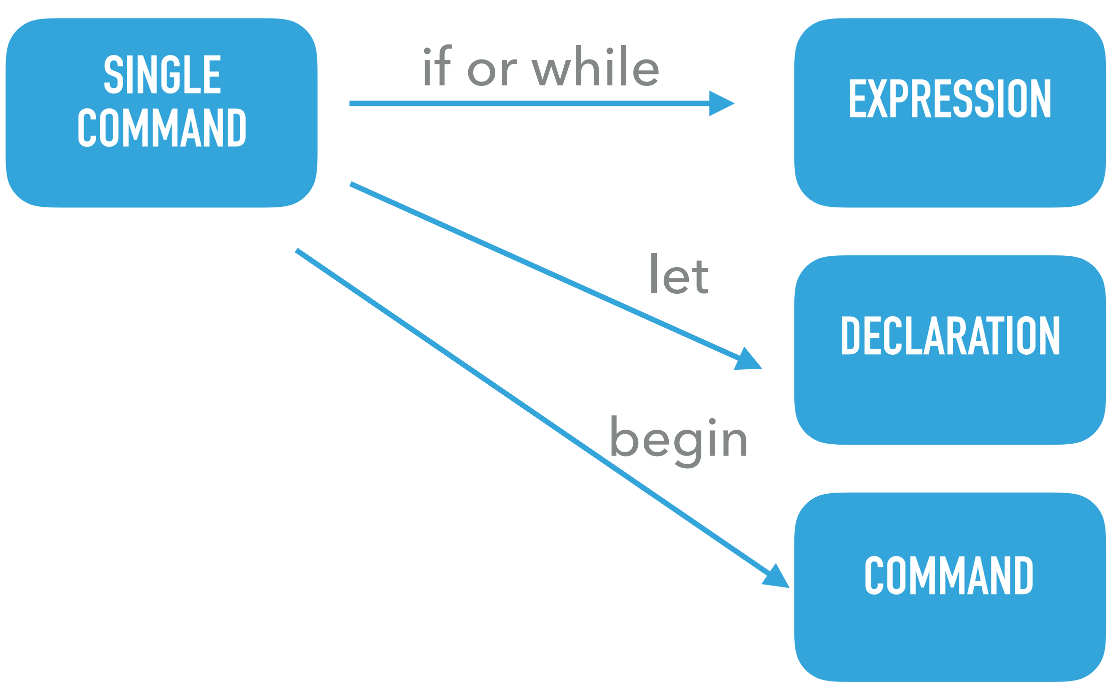
NOTICE ANYTHING?

- We have a limited (FINITE) number of Non-Terminals (STATES)
- When we are in one STATE there is a limited number of states we can then enter
- The transition is defined by a unique token (EVENT)
- What have we got?
- **A FINITE STATE MACHINE**

PARSER AS A FINITE STATE MACHINE

- If we are in the Single-Command state
- and receive an IF token what state do we enter?

Single-Command	::=	V-name (**:=**Expression|**(**Expression**)**)
	|	**if** Expression **then** Single-Command **else** Single-Command
	|	**while** Expression **do** Single-Command
	|	**let** Declaration **in** Single-Command
	|	**begin** Command **end**



PREDICT SETS OF A GRAMMAR

- The predict sets of a grammar are the set of terminals that define which production to choose.
- What is the predict set of Single-Command?
- Identifier, if, while, let, begin

```
Single-Command  ::=  V-name  (:=Expression|(Expression))
                  |  if Expression then Single-Command
                     else Single-Command
                  |  while Expression do Single-Command
                  |  let Declaration in Single-Command
                  |  begin Command end

V-name          ::=  Identifier
```

FOLLOW SETS OF A GRAMMAR

- The follow sets of a grammar are the set of terminals that can appear directly after a Token
- In many cases it is clear, but in some special cases can be more difficult.

FOLLOW SET

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

- the follow set for BEGIN is anything that is in the Predict set for Command
  - identifier, if, else, while, let, begin
  - and the END token

TRAILING ELSE

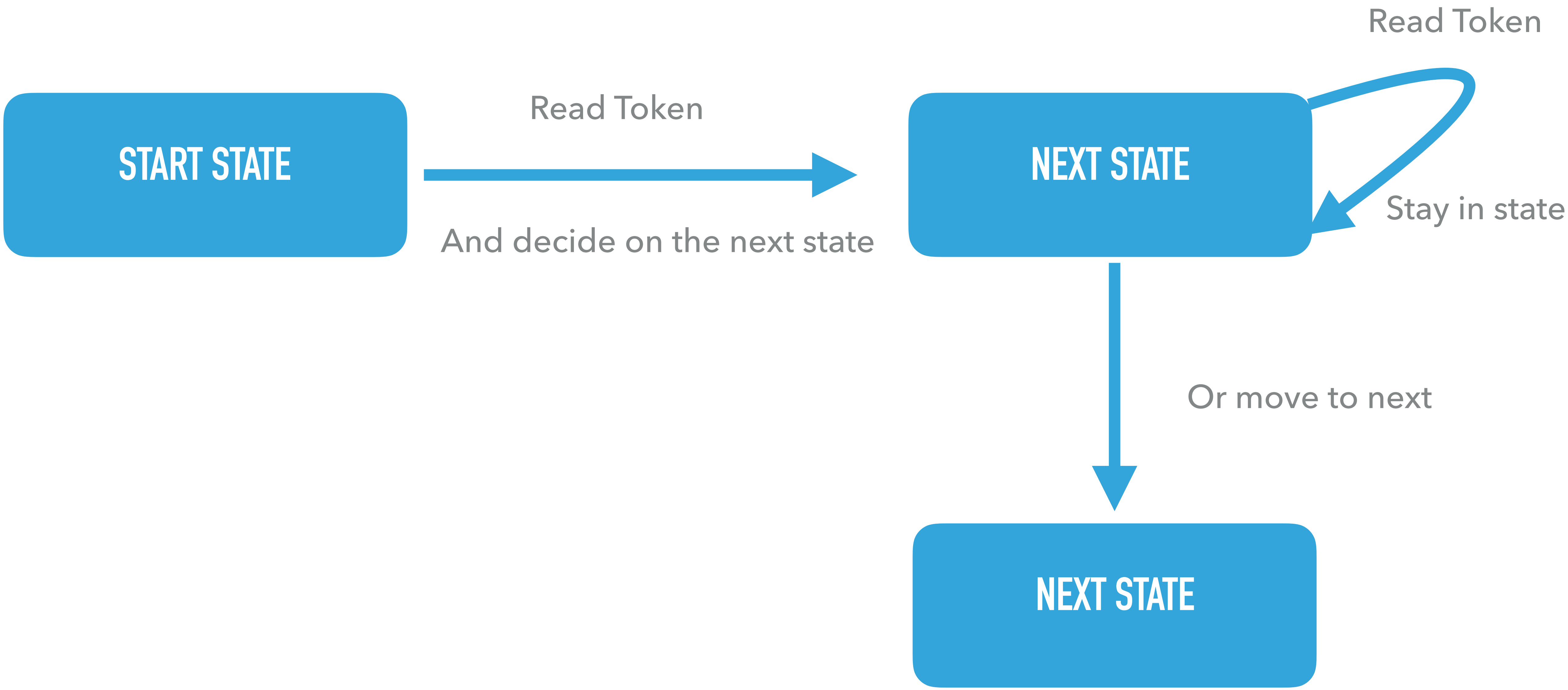
- A number of you noticed that there is no definition for an empty ELSE statement. How would we deal with an empty else?
- What is the Follow set for the ELSE statement if we assume it can be empty?

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

- ; - if another single command is following
- end - if we were in a begin-end
- End of Text - if we are the last statement

CODING THE PARSER

THE PROCESS


Read Token
START STATE
Read Token
NEXT STATE
Stay in state
And decide on the next state
Or move to next
NEXT STATE

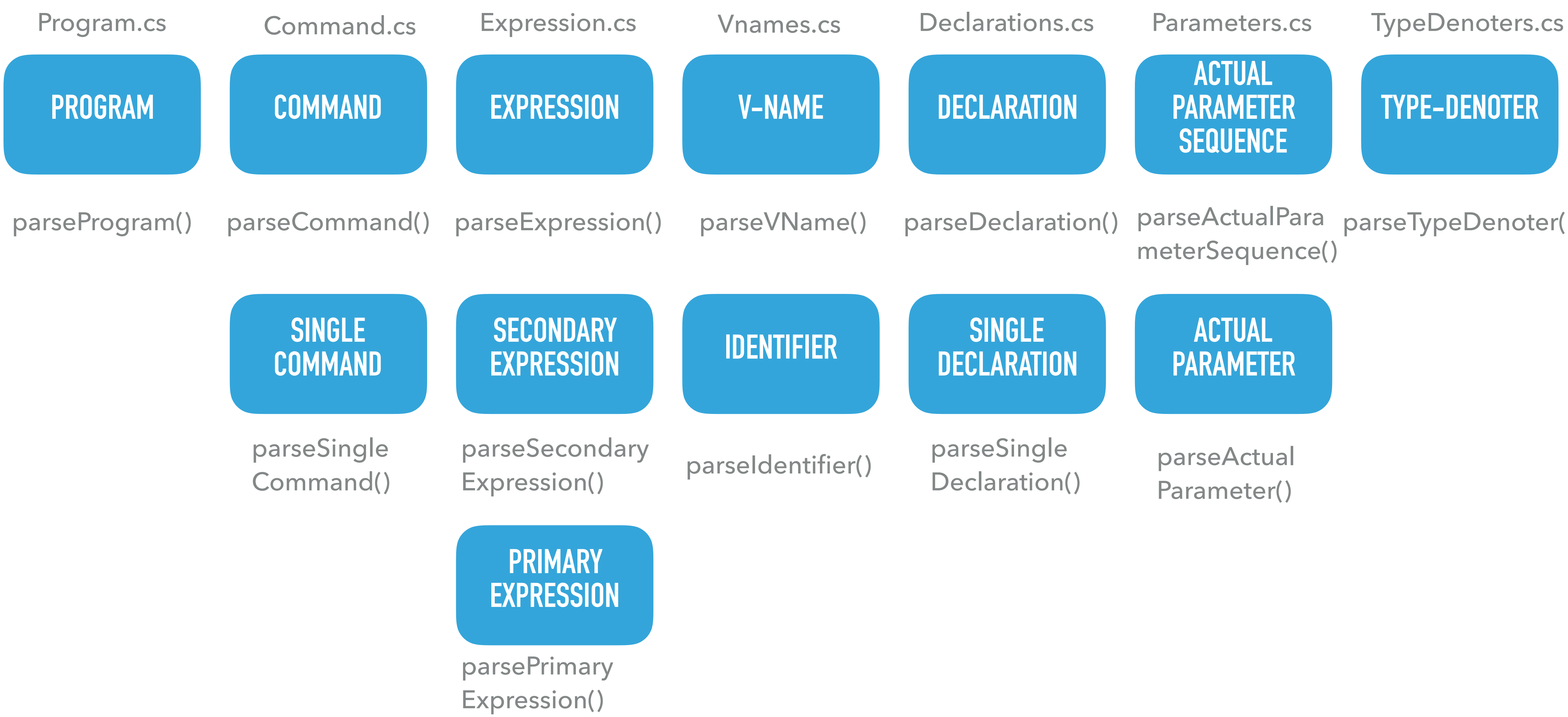
NON-TERMINALS

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

- Here is our language definition again.

HOW DOES THIS EFFECT OUR CODE?



- We have a method that represents every non-terminal in our grammar, these are the states of our state machine

LOOKING MORE CLOSELY - PARSER-COMMANDS.CS

Command ::= Single-Command (**;**single-Command)*

Single-Command ::= V-name (**:=**Expression | **(**Expression**)**)

| **if** Expression **then** Single-Command

else Single-Command

| **while** Expression **do** Single-Command

| **let** Declaration **in** Single-Command

| **begin** Command **end**

V-name ::= **Identifier**

DEALING WITH COMMAND

Command ::= Single-Command (;single-Command)*

The command rule states that a command is composed of a **single-command** followed by zero or more **single-commands** separated by a **;**

Directly translated into code -

```
void ParseCommand()
{
   ParseSingleCommand();
   while (_currentToken.Kind == TokenKind.Semicolon)
   {
      AcceptIt();
      ParseSingleCommand();
   }
}
```

DEALING WITH SINGLE COMMAND

Single-Command ::= V-name (**:=**Expression | **(**Expression**)**)
| **if** Expression **then** Single-Command
else Single-Command
| **while** Expression **do** Single-Command
| **let** Declaration **in** Single-Command
| **begin** Command **end**

The Single-Command rule states that a Single-Command is composed of one of a number of production rules.

To decide what production to take we need to look at the Predict set

STRUCTURE

```
void ParseSingleCommand(){
 switch (_currentToken.Kind){

   case TokenKind.Identifier:
   { break; }

   case TokenKind.Begin:
   { break; }

   case TokenKind.Let:
   { break; }

   case TokenKind.If:
   { break; }

   case TokenKind.While:
   { break; }
```

- We create a switch statement that RECOGNISES the different options in our predict set for Single-Commands
- Identifier, Begin, Let, If and While
- We will add some more code to these over the next few slides

SINGLE-COMMAND PRODUCTION RULES

V-name (:=Expression | (Expression))

The first production rule states that we need a variable name followed by either a becomes token (:=) and an expression or an expression in parenthesis ()

We can only be entering VName if we receive an Identifier

```
case TokenKind.Identifier:
{
  ParseVname();
  if (_currentToken.Kind == TokenKind.LeftParen){
    AcceptIt();
    ParseExpression();
    Accept(TokenKind.RightParen);
  }
  else
  {
    Accept(TokenKind.Becomes);
    ParseExpression();
  }
  break;
}
```

First Parse the variable name

Next if we find a parenthesis "(" accept it, call parseExpression() to deal with the expression then look for a ")"

Otherwise look for a Becomes token and parse that expression

SINGLE-COMMAND PRODUCTION RULES

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

The second production deals with the If statement, which only starts if we get an if statement.

```
case TokenKind.If:
{
    AcceptIt();
    ParseExpression();
    Accept(TokenKind.Then);
    ParseSingleCommand();
    Accept(TokenKind.Else);
    ParseSingleCommand();
    break;
}
```

ParseExpression

Look for the THEN token

ParseSingleCommand

Look for the ELSE token

ParseSingleCommand

SINGLE-COMMAND PRODUCTION RULES

while Expression **do** Single-Command

The third production deals with the WHILE statement, which only starts if we get an while token.

```
case TokenKind.While:
{
  AcceptIt();
  ParseExpression();              ParseExpression
  Accept(TokenKind.Do);           Look for the DO token
  ParseSingleCommand();           ParseSingleCommand
  break;
}
```

SINGLE-COMMAND PRODUCTION RULES

let Declaration **in** Single-Command

The fourth production deals with the LET statement, which only starts if we get an LET token.

```
case TokenKind.Let:
{
  AcceptIt();
  ParseDeclaration();          <- ParseDeclaration
  Accept(TokenKind.In);        <- Look for the IN token
  ParseSingleCommand();        <- ParseSingleCommand
  break;
}
```

SINGLE-COMMAND PRODUCTION RULES

begin Command **end**

The final production deals with the BEGIN statement, which only starts if we get an BEGIN token.

```
case TokenKind.Begin:
{
  AcceptIt();
  ParseCommand();          ParseCommand
  Accept(TokenKind.End);   Look for the END token
  break;
}
```

TRAILING ELSE?

- What about our trailing else problem.
- We can listen for the Follow sets here to get round this

```
case TokenKind.Semicolon:
case TokenKind.End:
case TokenKind.Else:
case TokenKind.EndOfText:
{
    break;
}
```

ERRORS

- If none of the tokens that are part of the Predict or Follow sets appear we should identify an error.

```
default:
System.Console.WriteLine(“error");
break;
```

LL(1) CONFLICTS

- Conflicts occur when the parser does not know what production to take.
- When the predict sets are not unique

LETS LOOK AT AN EXAMPLE

Expression ::= **let** Declaration **in** Expression
| **if** Expression **then** Expression **else** Expression
| Expression Operator Expression
| Integer-Literal
| Character-Literal

- What is the predict set for Expression?
  - Integer-Lit, Character-Lit
    - ok thats easy Token.IntLit & Token.CharLit
  - and Token.IF and Token.LET, right?

CONFLICT

Expression	::=	**let** Declaration **in** Expression	Production 1
	|	**if** Expression **then** Expression **else** Expression	Production 2
	|	Expression Operator Expression	Production 3
	|	Integer-Literal	Production 4
	|	Character-Literal	Production 5

- If we receive at Token.IF kind, how do we know if we are dealing with production 2 or production 3?

BREAK IT DOWN INTO FURTHER RULES

"5"

"LET"

Expression	::=	Secondary-Expression	Exp - Production 1
	|	**let** Declaration **in** Expression	Exp - Production 2
	|	**if** Expression **then** Expression **else** Expression	Exp - Production 3
Secondary-Expression	::=	Primary-Expression	SecExp Production 1
	|	Secondary-Expression Operator Primary-Expression	SceExp Production 2
Primary-Expression	::=	Integer-Literal	PriExp - Production 1
	|	Character-Literal	PriExp - Production 2

- Because of left priority have removed the conflict and we never get stuck!
- "Let" = Exp - Production 2
- Integer-Literal - PriExp - Production 1

WHAT ABOUT THIS?

Primary-Expression	::=	Integer-Literal	Production 1
	|	Character-Literal	Production 2
	|	V-name	Production 3
	|	Identifier (Actual-Parameter-Sequence)	Production 4

- Here V-Name and Identifier have the same predict set.
- But Production 3 should always be reached first.

CODE RESOLUTION

Primary-Expression ::= Integer-Literal
| Character-Literal
| V-name
| Identifier (Actual-Parameter-Sequence)

Both these productions produce an Identifier

```
case TokenKind.Identifier:
{
  ParseVname();
  if (_currentToken.Kind == TokenKind.LeftParen)
  {
    AcceptIt();
    ParseActualParameterSequence();
    Accept(TokenKind.RightParen);
  }
  break;
}
```

If we get a parenthesis, we can deal with that

At the minute we are only dealing with SYNTAX (structure) not SEMANTICS (meaning) so this is acceptable. We will develop this further later in the module.

SUMMARY

- Definitions matches the recursive descent parser.
- We will have parse methods for every non-terminal
- Triangle language def for the coursework is up with some sample programs and outputs.
- I'll give you some more code in the labs, if you are stuck ask and I'll go through it in the lab.