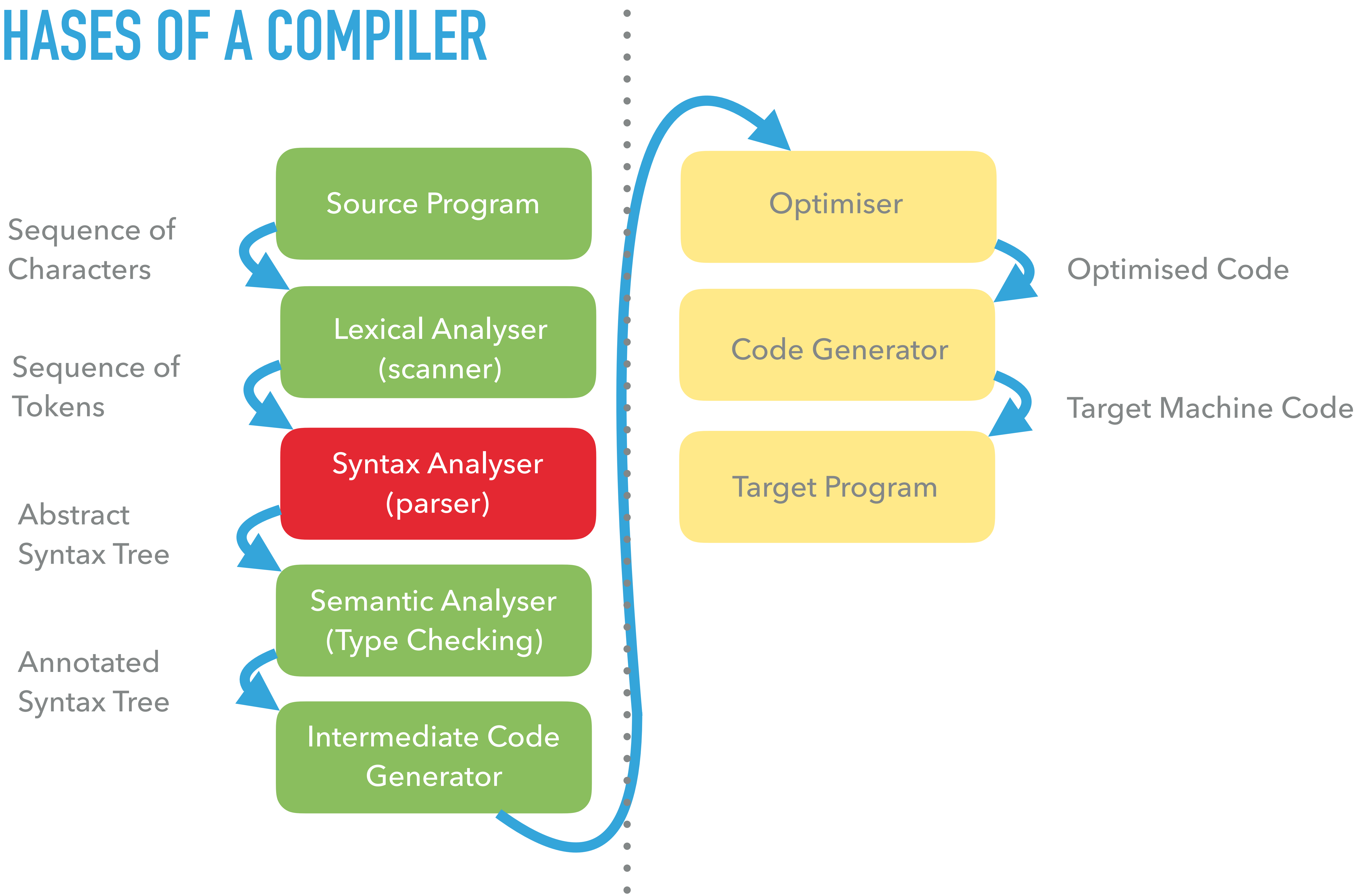
CM4106 - LANGUAGES AND COMPILERS

THE PARSER - PART1

THIS WEEK

- We will look at the second stage in the compilation process
- How language definitions help parsing
- How we build a parser to check the syntax of a language

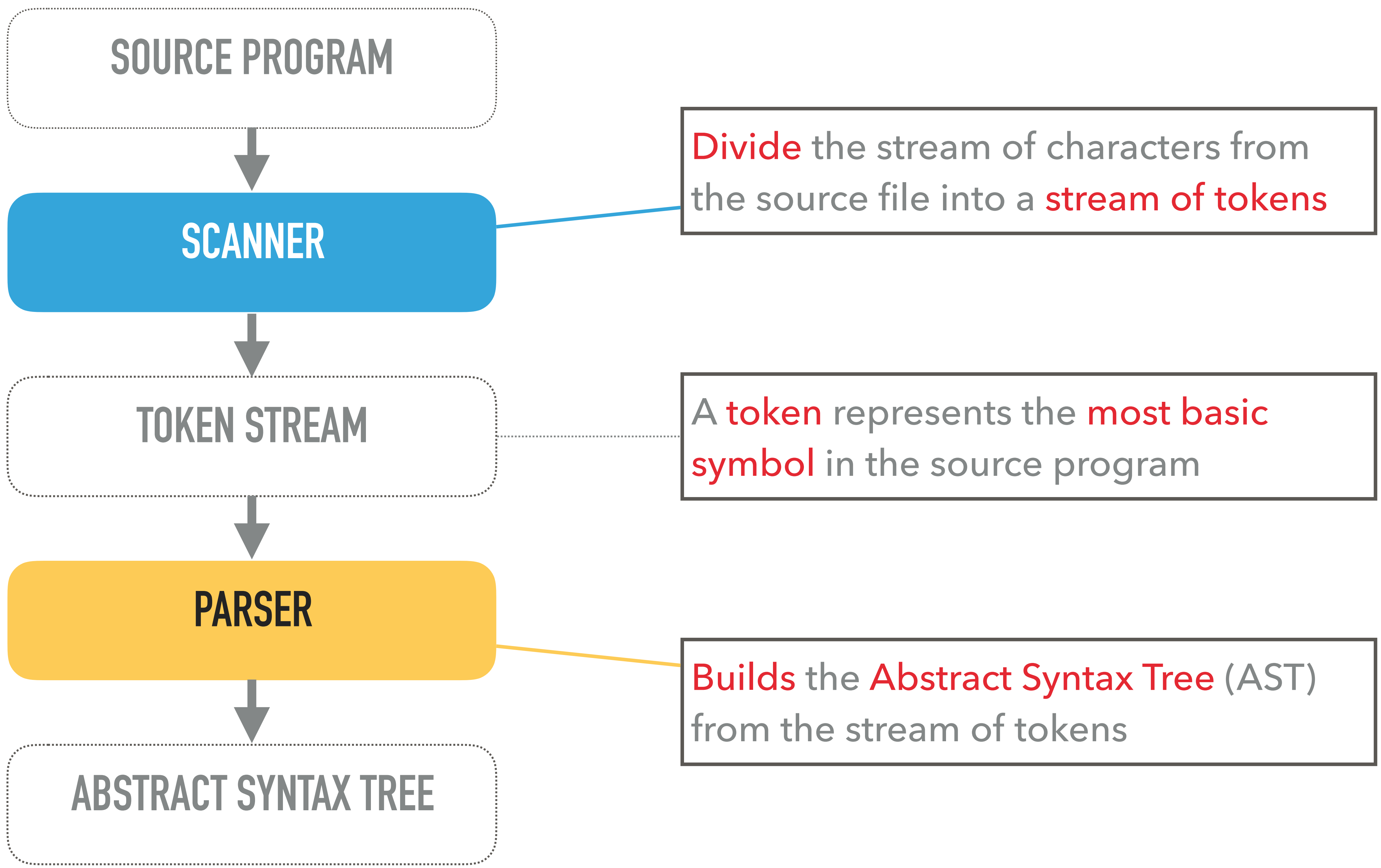
PHASES OF A COMPILER



THE PARSER

- INPUT: sequence of tokens from the scanner
- OUTPUT : AST (Abstract Syntax Tree)
- What it does:
  - groups tokens in to sentences (instructions)
- Performs Error Checking :
  - Syntax Errors, e.g x= y *= 5
  - Can check some semantics, eg x is not declared

MORE DETAIL


SOURCE PROGRAM
SCANNER
TOKEN STREAM
PARSER
ABSTRACT SYNTAX TREE
Divide the stream of characters from the source file into a stream of tokens
A token represents the most basic symbol in the source program
Builds the Abstract Syntax Tree (AST) from the stream of tokens

TOKEN STREAM

Simple program uses a process (proc) to increment a variable

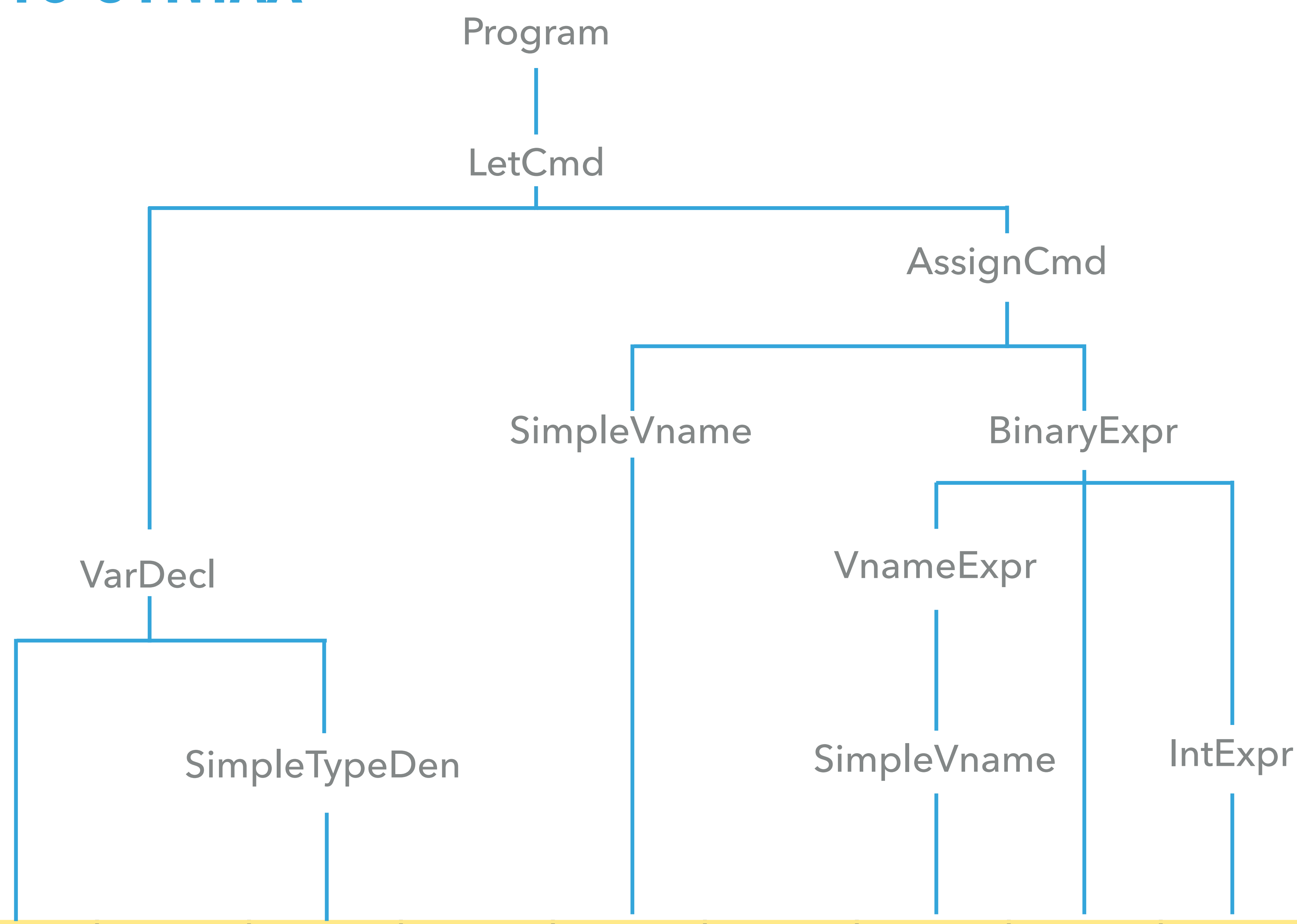
```
let
  var n: Integer;
  proc inc() ~ n := n + 1
in
  begin n:= 0;
    while n < 5 do inc();
  putint(n)
  end
```

let
var
n
:
Integer
;
proc
inc
(
)
~
n
:=

DIFFERENCE FROM SCANNER

- Scanner's goal is to turn a stream of characters in to a stream of tokens
- Each token has a type (kind) and a value (spelling)
- The Parser is only interested in the kind (e.g Identifier or IntLiteral) and not in the spelling (my_very_very_long_identifier_name)

CONNECTION TO SYNTAX



Let	Var	Ident	Colon	Ident	in	ident	becomes	ident	operator	intlit	EOT
let	var	x	:	Integer	in	x	:=	x	+	1	

SYNTAX AND GRAMMARS

- Syntax is concerned with the form of the programs
- We defined these using CONTEXT-FREE GRAMMARS
- A grammar defines a set of STRINGS which we call the language
- Grammars are composed of a set of components that help us define what the language can do

TERMINALS

- A finite set of terminal symbols
  - The most basic (atomic) symbols of a language.
  - These are what we type at a keyboard when we are writing in a program in any language
  - things like while, if and ;

NON-TERMINAL

- A finite set of non-terminal symbols
  - A non-terminal symbol (or just nonterminal) represents a class of phrases in a language
  - Things like Declaration, Command and Program

START SYMBOL

- A start symbol is a special nonterminal.
- It represents the principal (or highest) class of phrases in a language
- Usually the start symbol is Program
  - (alternatives are script or file)

PRODUCTION RULES

- A finite set of production rules
  - Production rules define how the phrases are written
  - eg. int x = 5 | int x = y | int x = y + 6

Declaration := typeDenoter Identifier = IntLiteral | Identifier | Statement

EBNF

- A grammar generates a set of sentences
- Each sentence is a string of terminal symbols
- A sentence has a unique structure which forms its syntax tree

EBNF EXAMPLE

- Mr | Ms generates {Mr , Ms}
- M (r | s) generates {Mr, Ms}
- ps*t generates {pt, pst, psst, pssst, …}
- b a (n a)* generates {ba, bana, banana, bananana, … }
- M (r|s)* generates {M, Mr, Ms, Mrr, Mrs, Msr, Mss, Mrrr … }

PARSING TERMINOLOGY

- Recognition
  - Deciding whether the input string is a sentence of the grammar
- Parsing
  - recognition and and determination of it's phrase structure
- Unambiguous Grammars
  - For today we will assume that there is only one way to parse an input - more on this next week

PARSING STRATEGIES

- Humans parse languages well. As we read or listen we are constantly parsing the words (tokens) into sentences (phrases). We know, instinctively, when something is wrong.
- There are many parsing algorithms used to replicate this process in machines, but they all fall into two basic approaches
- **Bottom-up** vs **Top-Down**

MICRO ENGLISH

- Imagine a very simple language

  Sentence ::= Subject Verb Object .
  Subject ::= I | a Noun | the Noun
  Object ::= me | a Noun | the Noun
  Noun ::= cat | mat | rat
  Verb ::= like | is | see | sees

- Possible sentences

  the cat sees a rat .
  I like the cat .
  the cat see me .
  I like me .
  a rat like me .

BOTTOM-UP PARSING

Sentence ::= Subject Verb Object .
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees

- Just as the name suggests this type of parsing runs from the terminals up to the root node
- the parser will examine the terminals in the input string and try to work out what the rules used to get here were.
- Imagine the input "**the cat sees a rat**"

THE CAT SEES A RAT – BOTTOM UP

Sentence ::= Subject Verb Object .
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees

The first input symbol is THE the parser cannot work out what production rule caused this yet

The Second terminal is CAT the parser recognises this as a NOUN

The parser can then combine the previous input symbol and identify the production as SUBJECT

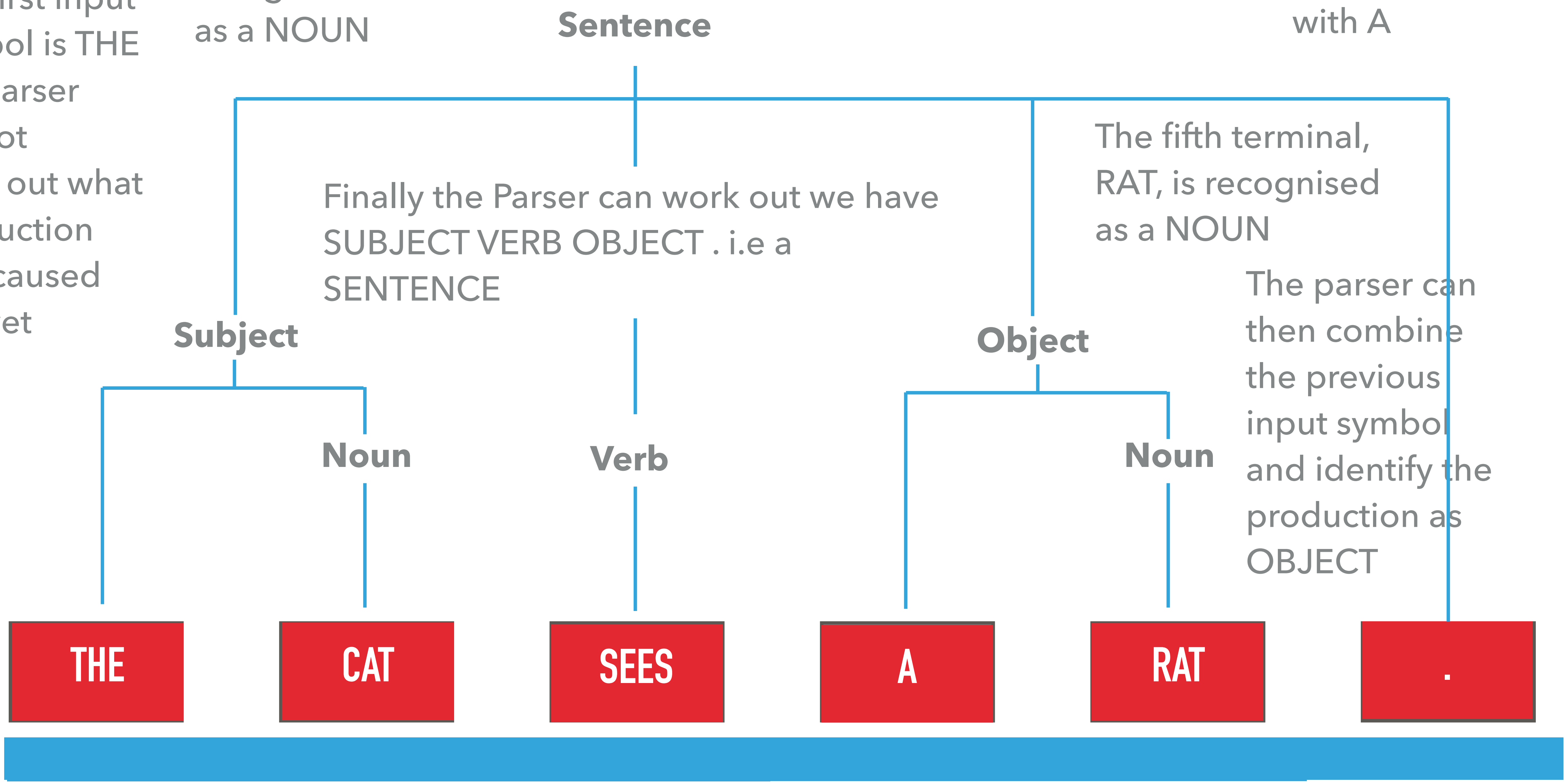
The Third terminal SEES is recognised as a VERB

There is nothing the parser can do with A

The fifth terminal, RAT, is recognised as a NOUN

The parser can then combine the previous input symbol and identify the production as OBJECT

Finally the Parser can work out we have SUBJECT VERB OBJECT . i.e a SENTENCE

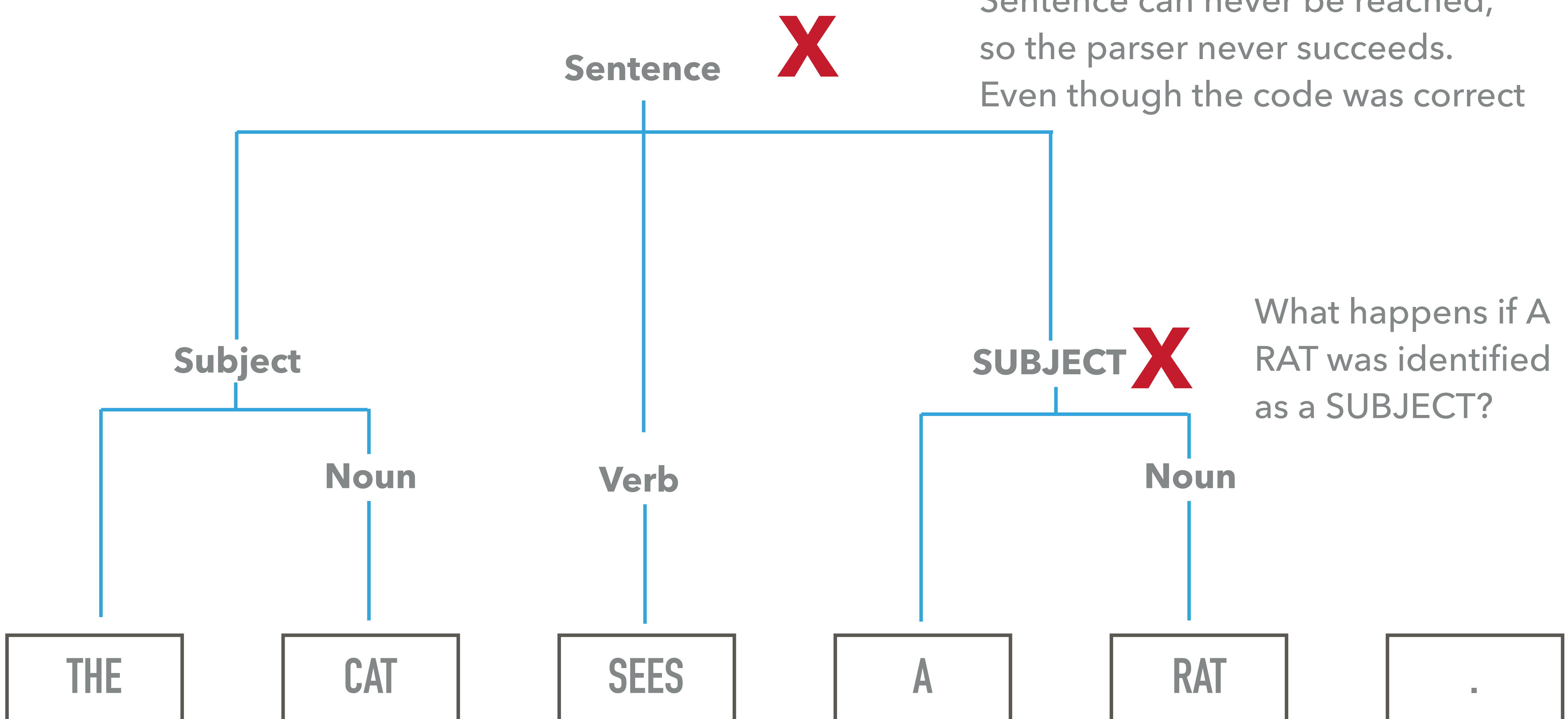


PROBLEMS?

Sentence ::= Subject Verb Object .
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees

- The parser succeeds if the input is reduced to the start symbol
- Bottom up can lead to blind alleys, where no rule can be found

Sentence can never be reached,
so the parser never succeeds.
Even though the code was correct



What happens if A
RAT was identified
as a SUBJECT?

TOP-DOWN PARSING

Sentence ::= Subject Verb Object .
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees

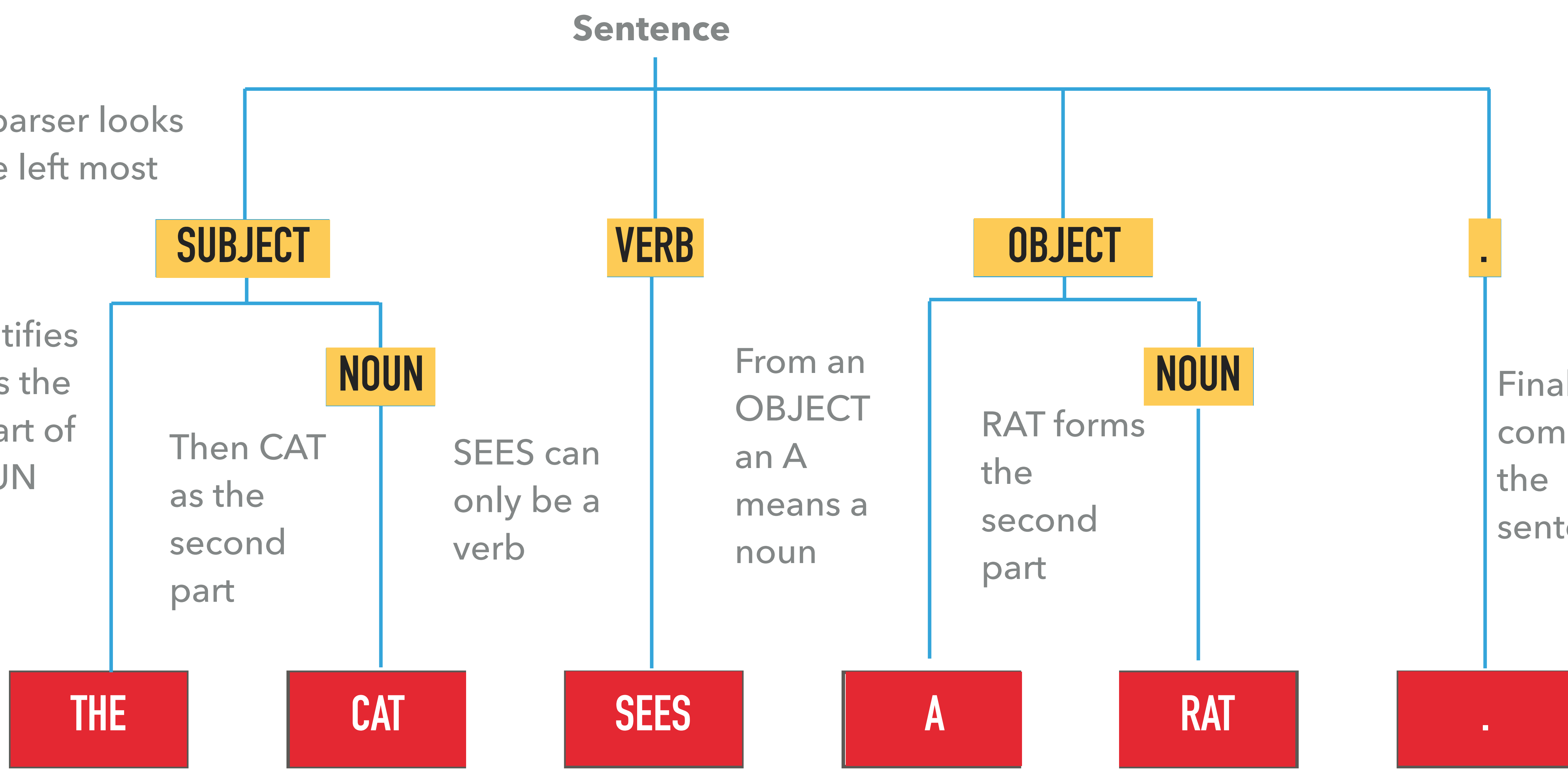
- The parser examines the terminal symbols and reconstructs the syntax tree from the root node down
- again lets look at “the cat sees the rat”

THE CAT SEES A RAT – TOP DOWN

Sentence ::= Subject Verb Object .
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees

The parser begins by making the root node. i.e the start symbol for our language

The parser needs to decide what production rule to apply. For sentence there is only one ::= SUBJECT VERB OBJECT .



The parser looks at the left most stub

It identifies THE as the first part of a NOUN

Then CAT as the second part

SEES can only be a verb

From an OBJECT an A means a noun

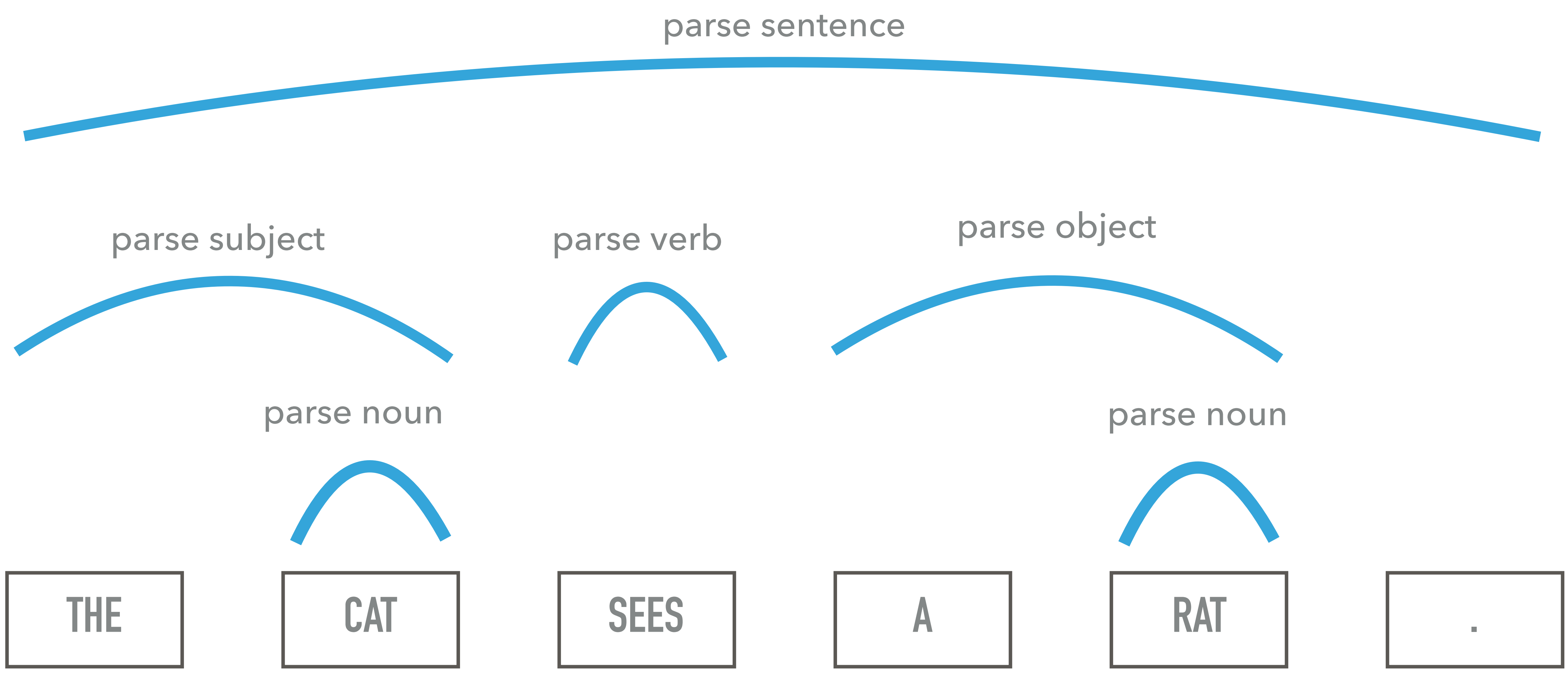
RAT forms the second part

Finally . completes the sentence

RECURSIVE DESCENT PARSING

- This is a TOP-DOWN algorithm
- Based on a system where each NONTERMINAL (N) will have a method called parseN()
- So we end up with a function call stack, that matches the tree structure of the source
- With each parse function recursively calling the next

CALL STACK


parse sentence
parse subject
parse verb
parse object
parse noun
parse noun
THE
CAT
SEES
A
RAT
.

PARSER FOR MICRO-ENGLISH

Sentence ::= Subject Verb Object .

```
void parseSentence() {
  parseSubject();
  parseVerb();
  parseObject();
  accept(“.”);
}
```

Accept is a bit like our Takeit() method in the scanner. It identifies the expected TERMINAL and adds it

PARSE NOUN

Subject ::= I | a Noun | the Noun

```
void parseSubject() {
  if(currentToken == “I”){
        accept(“I”);
  }
  else if (currentToken == “a”){
        parseNoun();
  }
  else if (currentToken == “the”){
        accept(“the”);
        parseNoun();
  }
  else{
  //there has been a horrible syntax error
  }
}
```

I

a Noun

the Noun

Given the current token the parser should be able to identify what production to take

PARSE NOUN

Noun::= cat | mat | rat

```
void parseNoun() {
  if(currentToken == “cat”){
      accept(“cat”);                    cat
  }
  else if (currentToken == “mat”){
       accept(“mat”);                   mat
  }
  else if (currentToken == “the”){
       accept(“rat”)                    rat
  }
  else{
  //there has been a horrible syntax error
  }
}
```

Each terminal is recognised and accepted

ACCEPT

```
void accept (Token expectedToken){
 if(currentToken == expectedToken{
  //do something with our correct token
  currentToken = nextToken();
 }
 else{
  //report a syntax error
 }
}
```

accept lets us check that the terminal we expect is the one we have

THE PARSER

```
public class MicroEnglishParser {

  protected Token currentToken;

  public void parse() {

    currentToken = first token;

    parseSentence();

    //check that no token follows the sentence

}

protected void accept(Token expected) { ... }
protected void parseSentence() { ... }
protected void parseSubject() { ... }
protected void parseObject() { ... }
protected void parseNoun() { ... }
protected void parseVerb() { ... } ... }
```

Store current token (or terminal)

Parse method starts it all off

Get the first token

Try to parse a sentence (the start symbol)

Parsemethods for the other non-terminals

FOR TRIANGLE

- Your parser is going to be more complicated than this.
- Triangle is a more complex language with more non-terminal symbols and productions.
- But the process is exactly the same.
- Its not going to fit in a single class.
- We will start to break this down in the lab