CRICOS PROVIDER 00123M

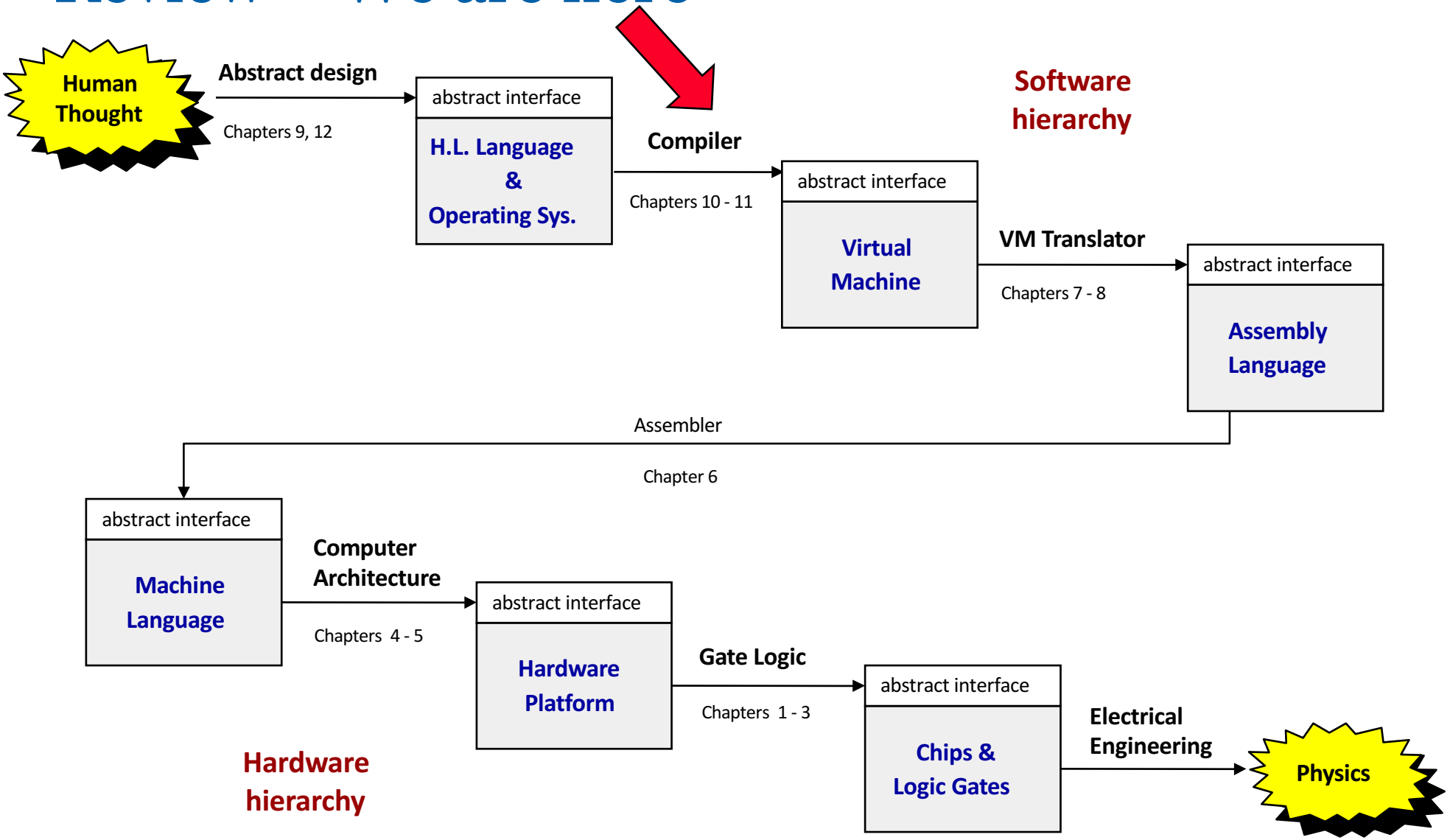
School of Computer Science

COMP SCI 2000 Computer Systems
Lecture 17

adelaide.edu.au

seek LIGHT

Review – We are here



Review – Last Lecture

- The Jack Language

Preview – Next Lectures

- Syntax Analysis
  - Lexical Analysis (tokenising)
  - Recursive Descent Parsing
- What parsing is ?
- How will we construct a parser ?

Motivation: Why study about compilers?

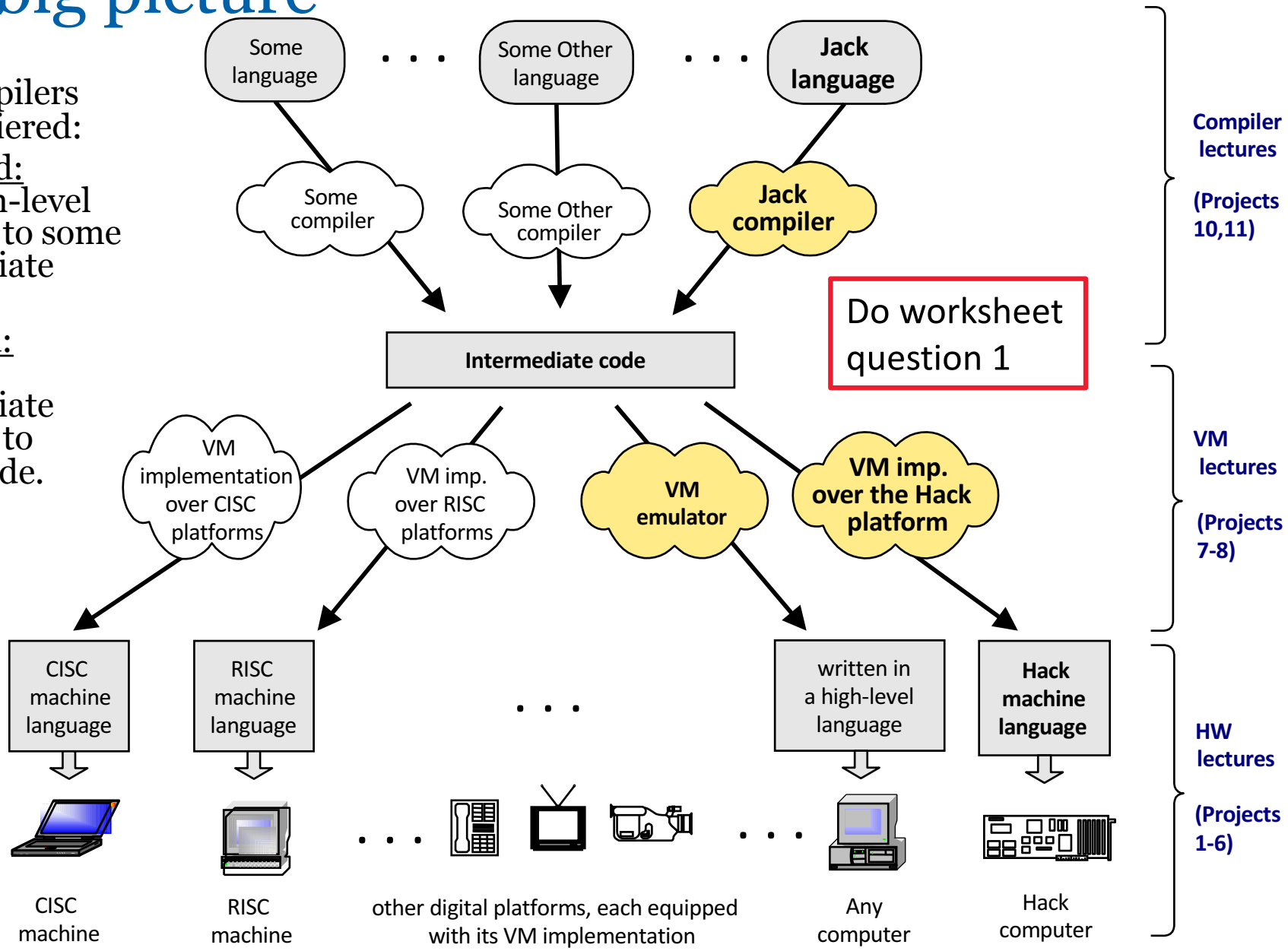
Because Compilers ...

- Are an essential part of applied computer science
- Are very relevant to computational linguistics
- Are implemented using classical programming techniques
- Employ important software engineering principles
- Train you in developing software for transforming one structure to another (programs, files, transactions, ...)
- Train you to think in terms of ”description languages”.

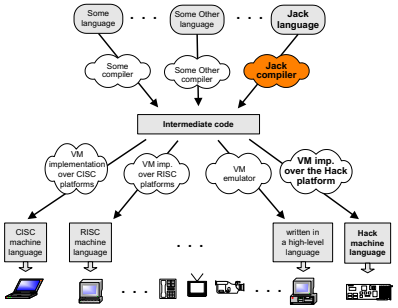
The big picture

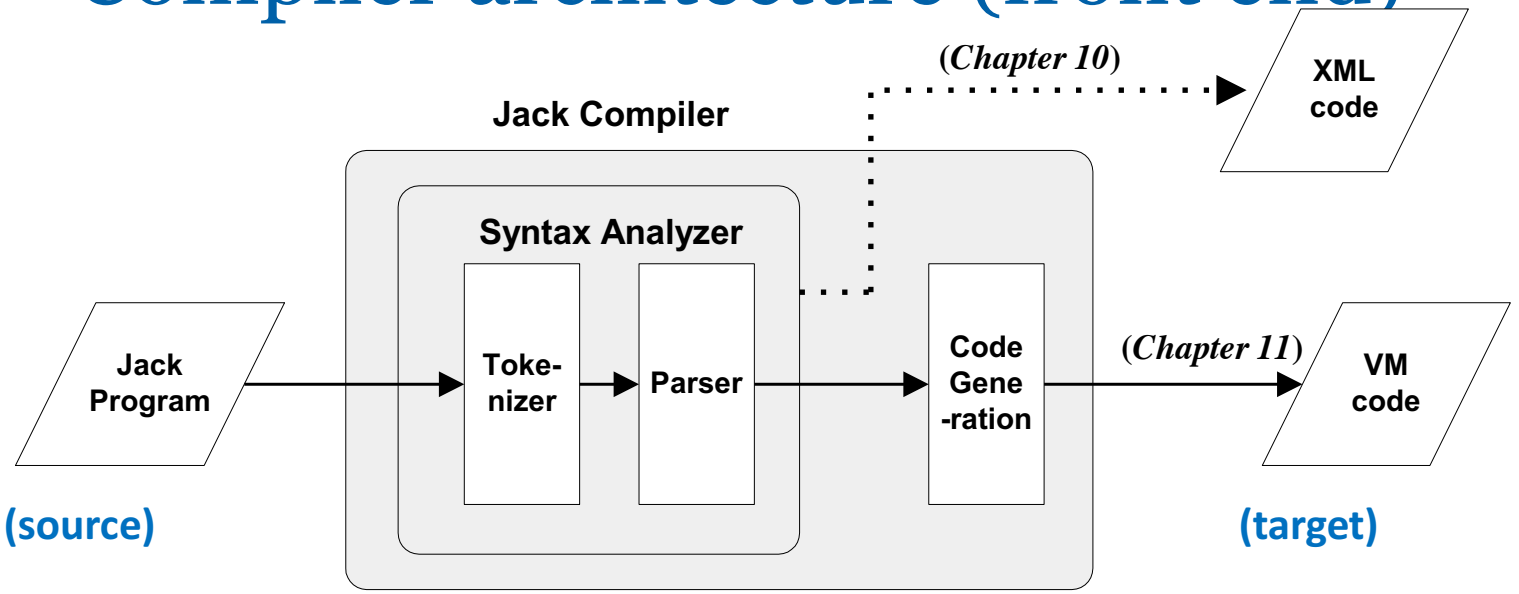
Modern compilers are two-tiered:

- Front-end: from high-level language to some intermediate language
- Back-end: from the intermediate language to binary code.



Compiler architecture (front end)





- Syntax analysis: understanding the semantics implied by the source code
  - Tokenizing: creating a stream of "tokens"
  - Parsing: matching the token stream with the language grammar

  XML output = one way to demonstrate that the syntax analyzer works
- Code generation: reconstructing the semantics using the syntax of the target code.

Tokenising / Lexical analysis

Code Fragment

```
while ( count < 100 ) /** demonstration code */
{
     let count = count + 1 ;
}
```

Tokens

```
while
(
count
<
100
)
{
let
count
=
count
+
1
;
}
```

- Remove white space
- Construct a token list (language tokens)
- Things to worry about:
  - Language specific rules:
    e.g. how to treat "++"
  - Language-specific classifications:
    keyword, symbol, identifier, integerConstant, stringConstant,...
- While we are at it, we can have the tokenizer record not only the token, but also its lexical classification (as defined by the source language grammar).

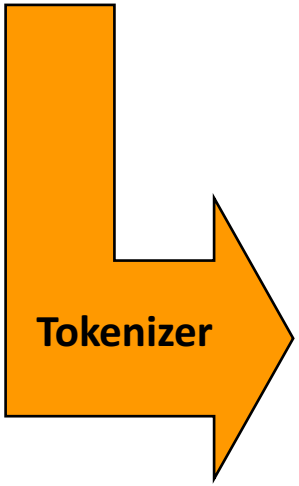
Do worksheet question 2

Jack Tokenizer

Source code

```
if (x < 153) {let city = "Paris";}
```

Char	XML Entity
<	<
>	>
'	'
"	"
&	&



Tokenizer's output

```
<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> &lt; </symbol>
  <integerConstant> 153 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
```

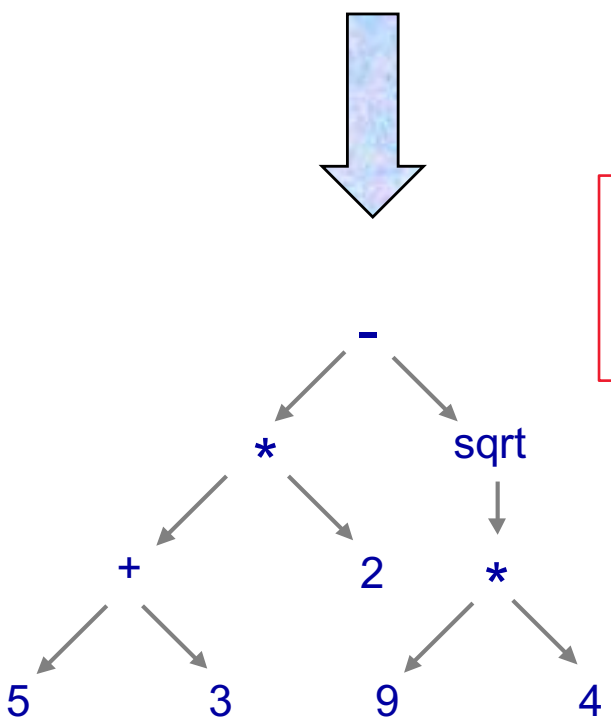
Parsing

- The tokeniser discussed thus far is part of a larger program called a *parser*
- Each language is characterized by a *grammar*.
  The parser is implemented to recognize this grammar in given texts
- The parsing process:
  - A text is given and tokenized
  - The parser determines whether or not the text can be generated from the grammar
  - In the process, the parser performs a complete structural analysis of the text
- The text can be an expression in a :
  - Natural language (English, …)
  - Programming language (Jack, …).

Parsing examples

Jack

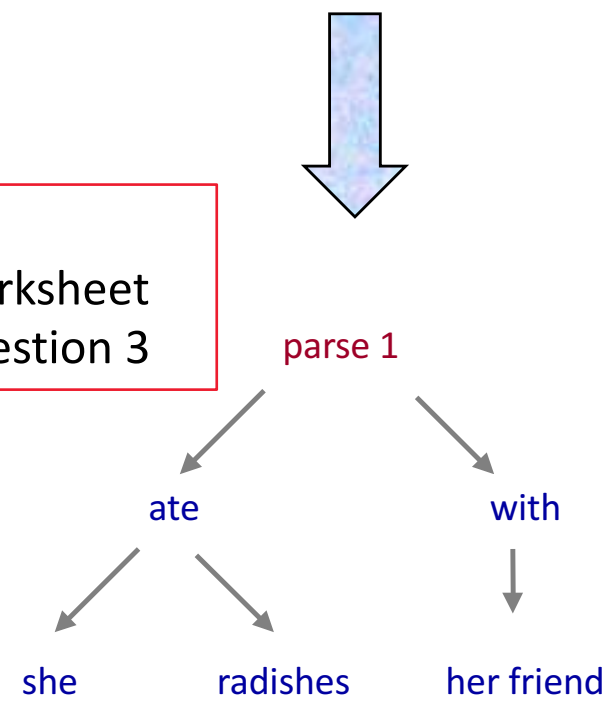
```
((5+3)*2) – sqrt(9*4)
```

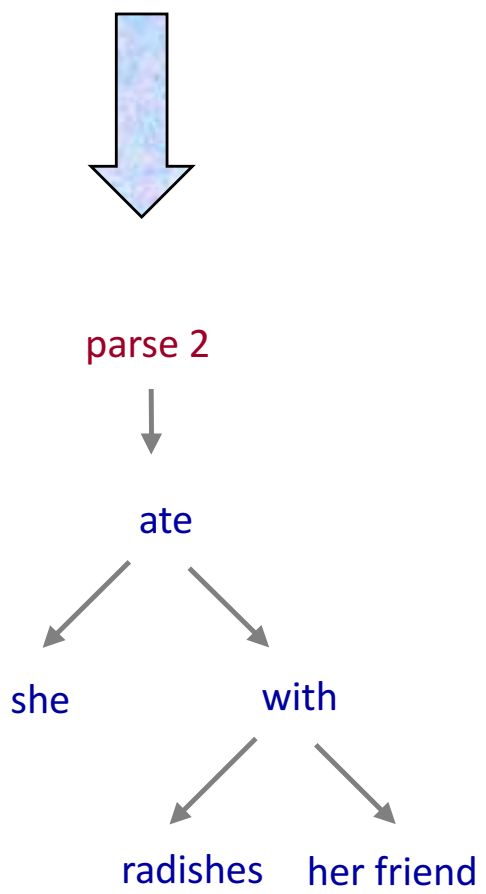


English

```
She ate radishes with her friend
```

Do worksheet question 3





More examples of challenging parsing

Time flies like an arrow
Fruit flies like a banana

We gave the monkeys the bananas because they were hungry
We gave the monkeys the bananas because they were over-ripe

I never said she stole my money	
I never said she stole my money	Someone else said it
I never said she stole my money	I did not say it
I never said she stole my money	I implied it
I never said she stole my money	Someone did, not necessarily her
I never said she stole my money	I considered it borrowed
I never said she stole my money	She stole something else of mine
I never said she stole my money	She stole something but not money

A typical grammar of a typical C-like language

Grammar

```
program:         statement

statement:         whileStatement
                 | ifStatement
                 | 'statement' ';'
                 | '{' sequence '}'

whileStatement: 'while' '(' 'expression' ')' statement

ifStatement:     'if' '(' 'expression' ')' statement
                 ( 'else' statement )?
sequence:        '' | statement sequence
```

Code sample

```
while (expression)
{
  if (expression)
     statement;
     while (expression)
     {
        statement;
        if (expression)
           statement;
     }
  while (expression)
  {
     statement;
     statement;
  }

  if (expression)
  {
    statement;
    while (expression)
    {
       statement;
       statement;
    }
    if (expression)
       if (expression)
          statement;
  }
}
```

- A grammar is a set of rules that describe all legal examples of a language.
- It has simple (terminal) forms
- It has complex (non-terminal) forms
- It is highly recursive.

Do worksheet Question 4

Parse tree

Input Text:

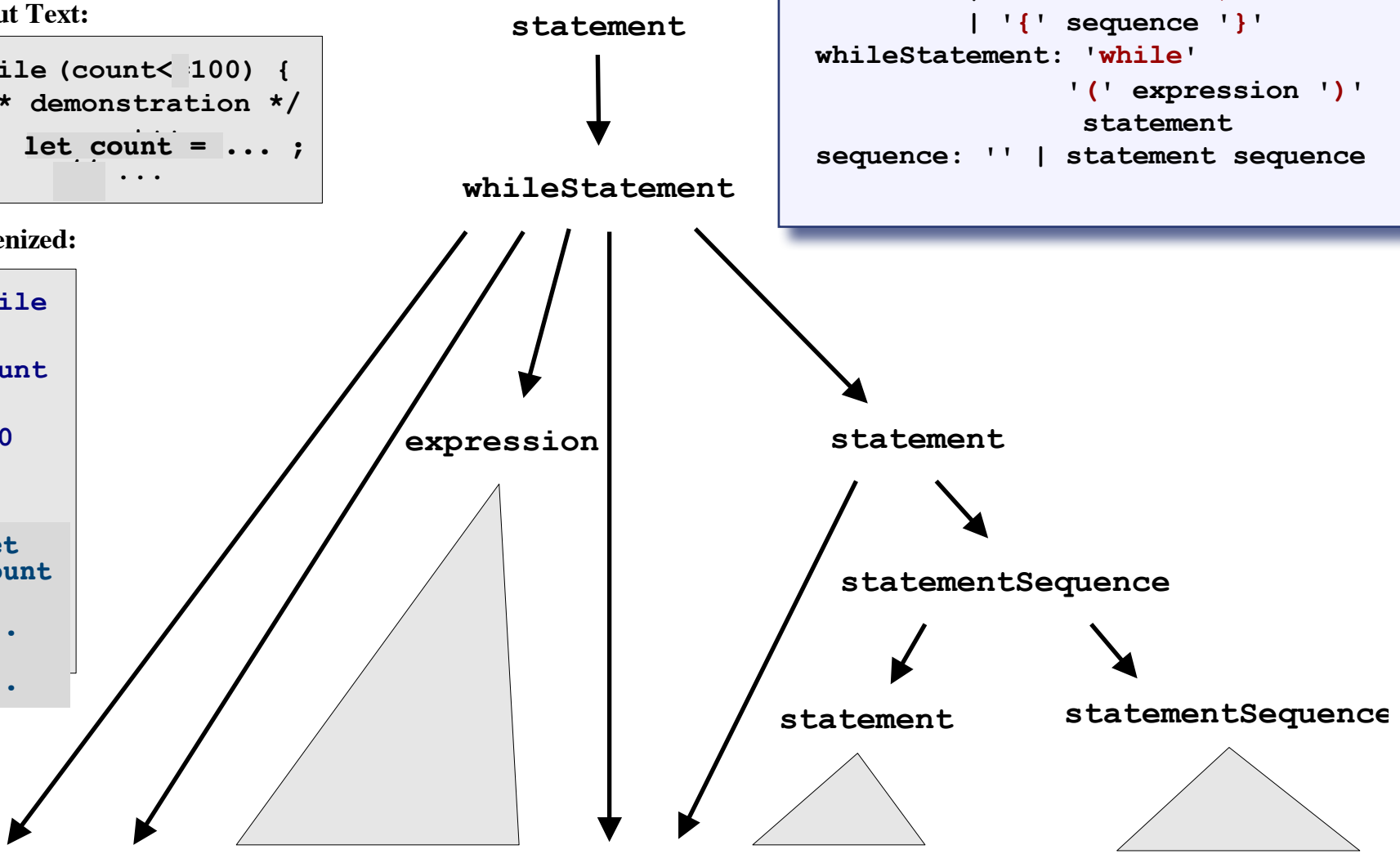
```
while (count< 100) {
/** demonstration */
    let count = ... ;
        ...
```

Tokenized:

```
while
(
count
<
100
)
{
let
count
=
...
;
...
```

```
program:  statement;

statement: whileStatement
         | ifStatement
         | 'statement' ';'
         | '{' sequence '}'
whileStatement: 'while'
                '(' expression ')'
                 statement
sequence: '' | statement sequence
```



Review/Preview

- In this lecture we showed
  - What syntax analysis is
  - What tokenising is
  - What parsing is
  - Potential problems of ambiguous statements
  - Introduction to Grammars
- In the next lecture we show
  - More detail on grammars
  - The Jack grammar
  - The Jack Tokeniser
  - The Jack Parser