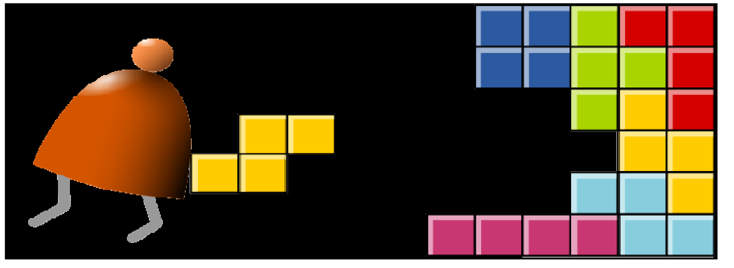
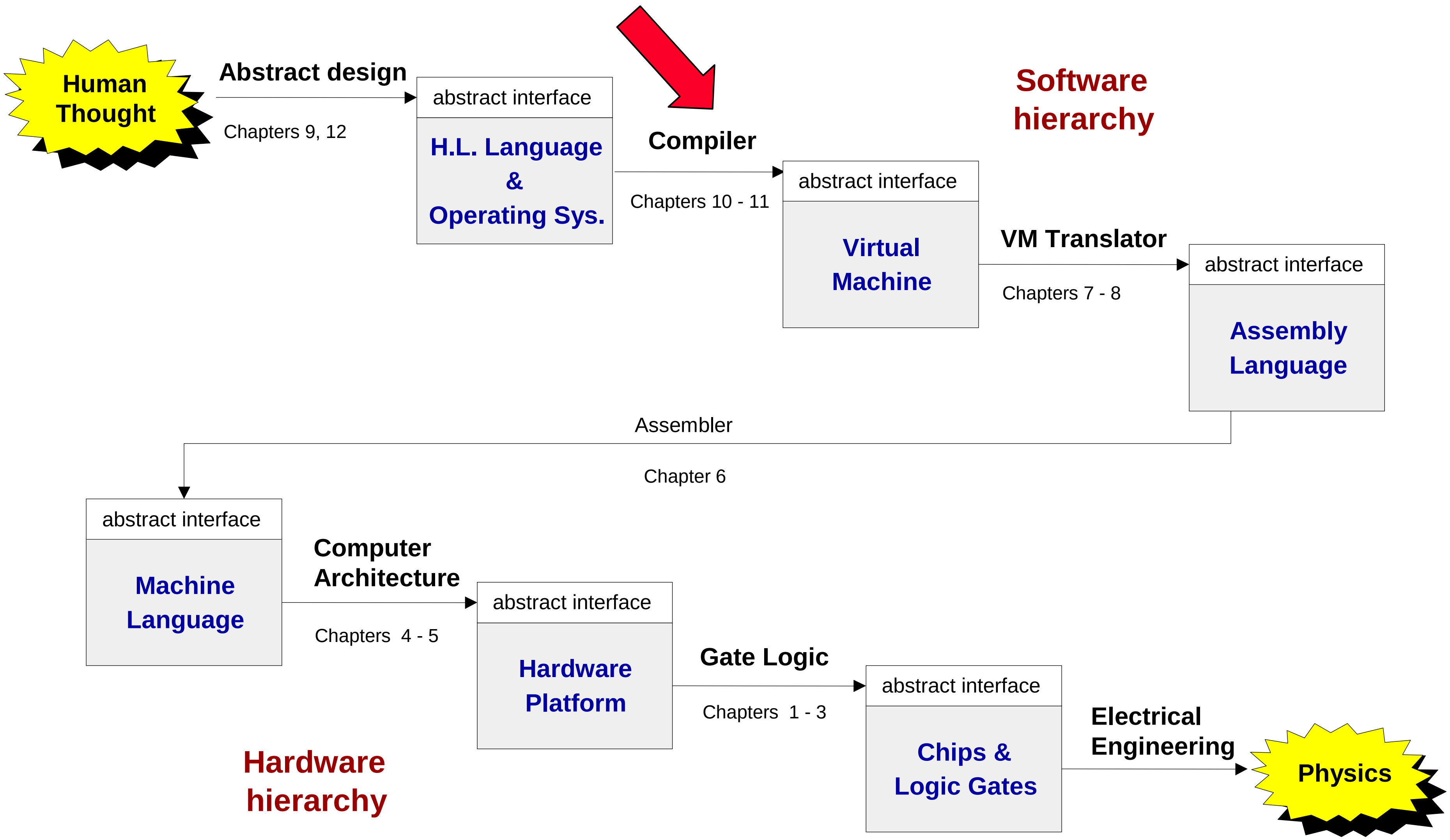
Compiler I: Syntax Analysis



Building a Modern Computer From First Principles

www.nand2tetris.org

Course map


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

Motivation: Why study about compilers?

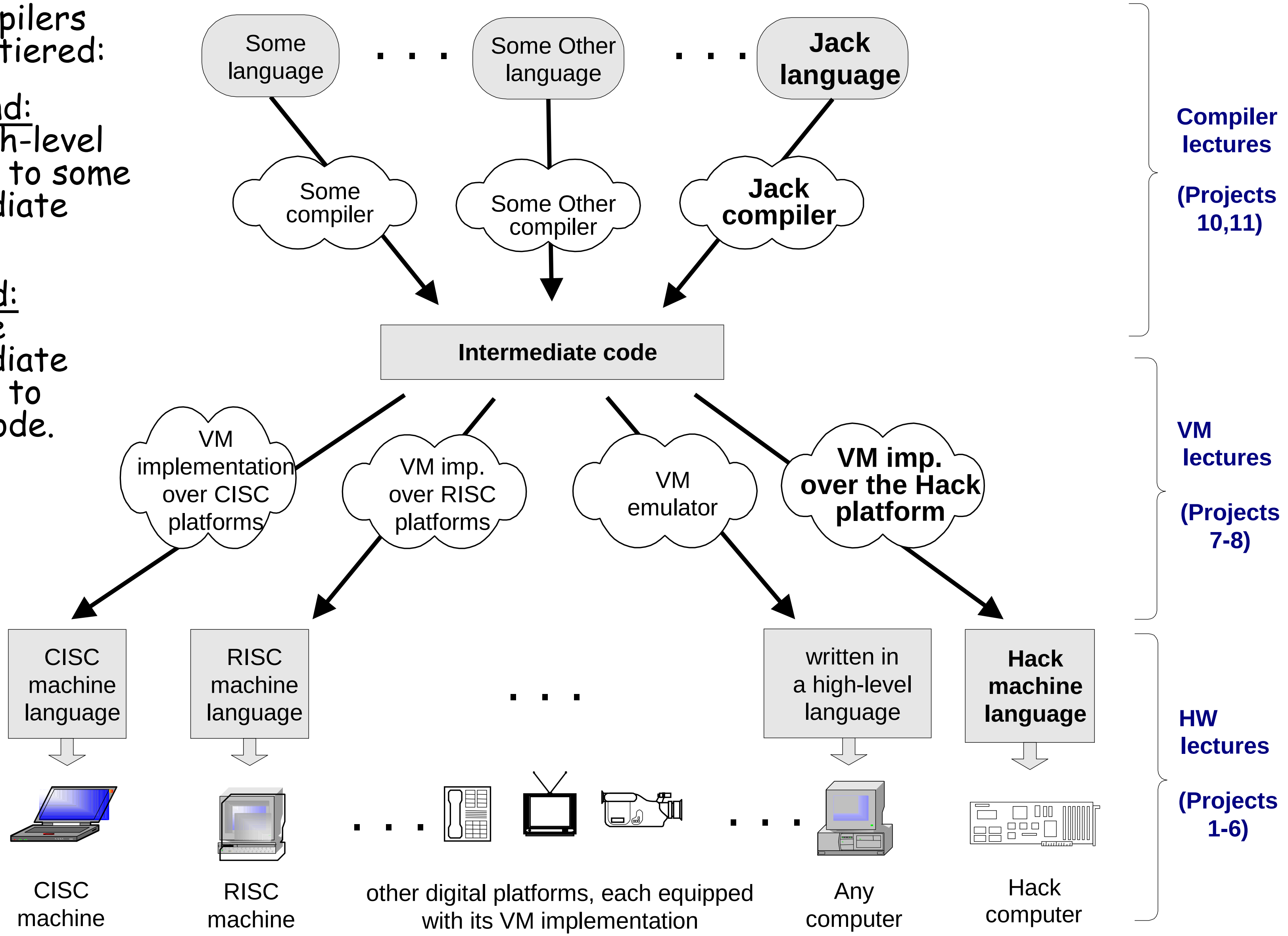
Because Compilers …

- 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".

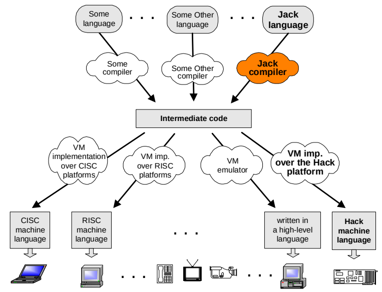
The big picture

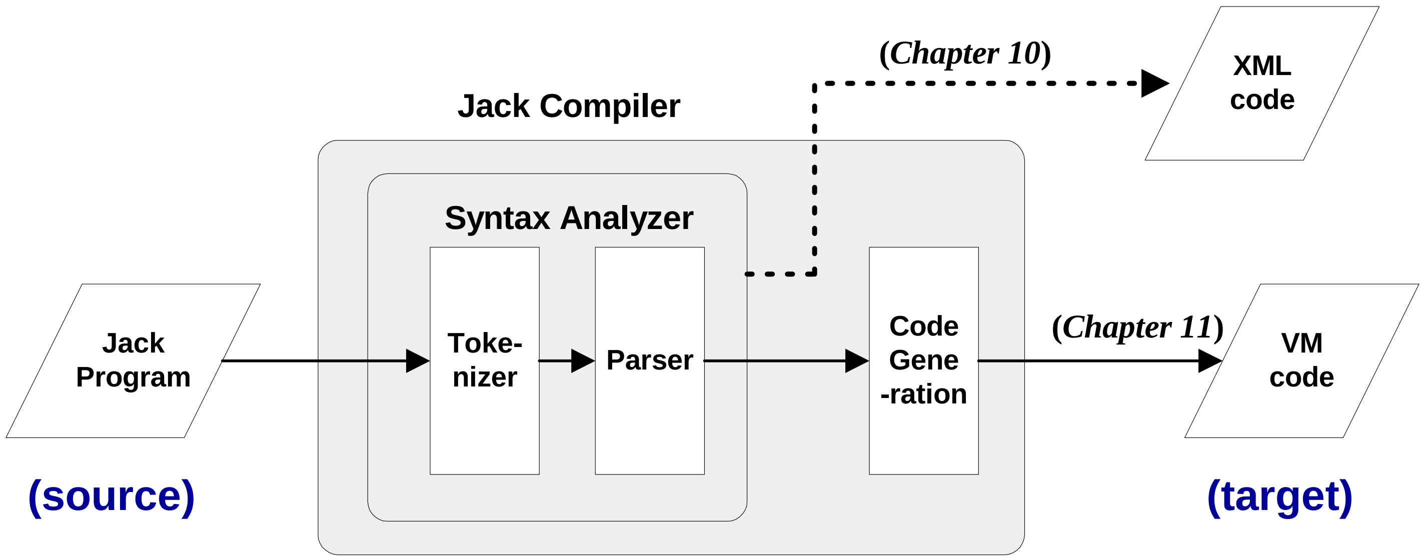
Modern compilers are two-tiered:

- 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 "atoms"
  - Parsing: matching the atom 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.

Tokenizing / Lexical analysis

Code fragment

```
while (count<=100) { /** demonstration */
      count++;
      // body of while continues
      ...
```

tokenizer

Tokens

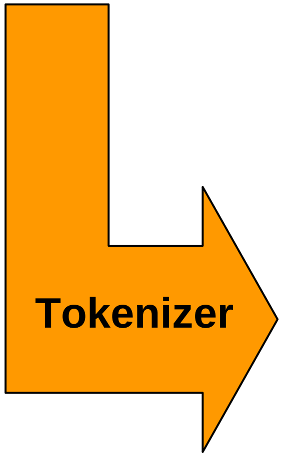
```
while
(
count
<=
100
)
{
count
++
;
...
```

- Remove white space
- Construct a token list (language atoms)
- Things to worry about:
  - Language specific rules:
    e.g. how to treat "++"
  - Language-specific classifications:
    keyword, symbol, identifier, integerCconstant, 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).

Jack Tokenizer

Source code

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



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 tokenizer discussed thus far is part of a larger program called *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 weather 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 in an expression in a :
  - Natural language (English, …)
  - Programming language (Jack, …).

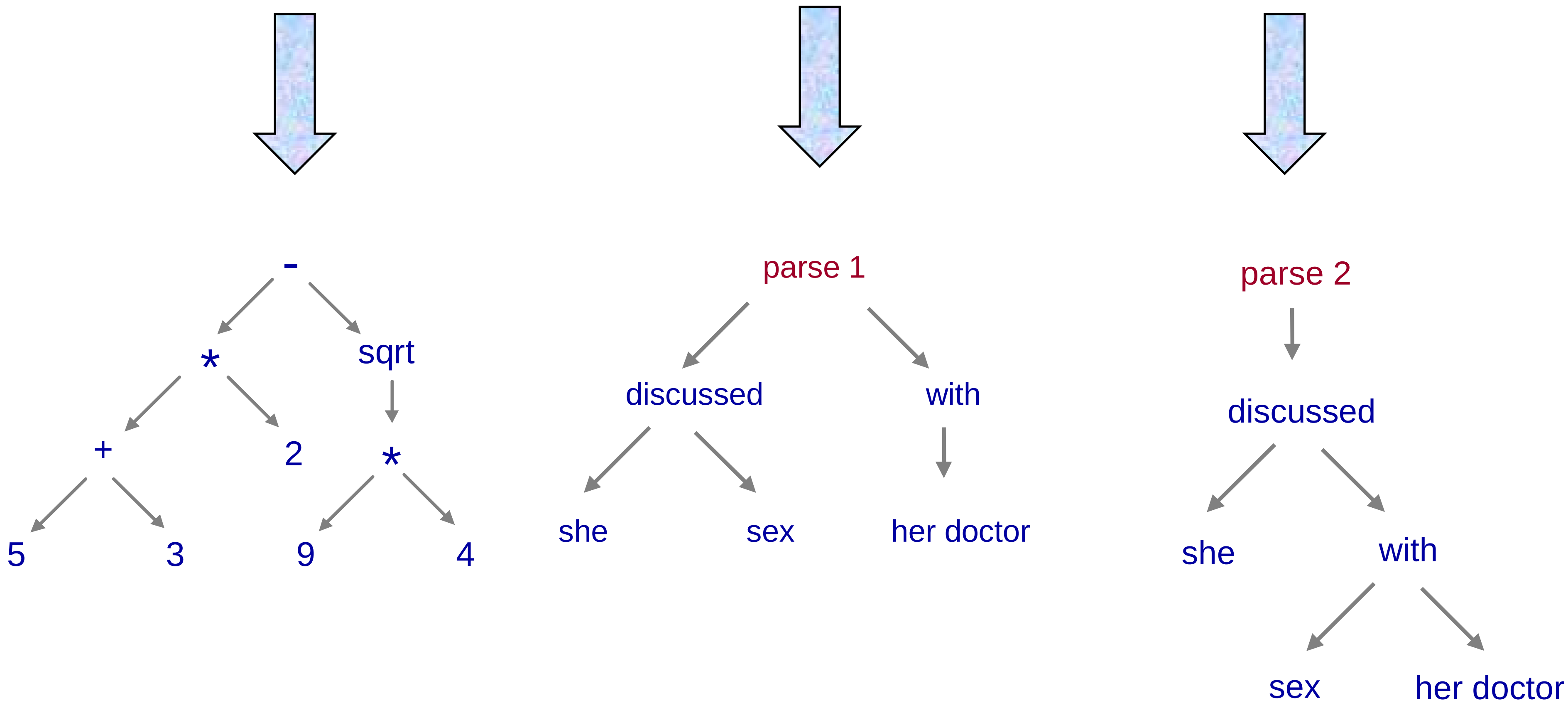
Parsing examples

Jack

```
(5+3)*2 – sqrt(9*4)
```

English

```
she discussed sex with her doctor
```



More examples of challenging parsing

Time flies like an arrow

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

<u>I</u> never said she stole my money

I <u>never</u> said she stole my money

I never <u>said</u> she stole my money

I never said <u>she</u> stole my money

I never said she <u>stole</u> my money

I never said she stole <u>my</u> money

I never said she stole my <u>money</u>

A typical grammar of a typical C-like language

Grammar

```
program:        statement;

statement:        whileStatement
                | ifStatement
                | // other statement possibilities ...
                | '{' statementSequence '}'

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

ifStatement:      simpleIf
                | ifElse

simpleIf:       'if' '(' expression ')' statement

ifElse:         'if' '(' expression ')' statement
                'else' statement

statementSequence:    ''   // null, i.e. the empty sequence
                    | statement ';' statementSequence

expression:       // definition of an expression comes here

// more definitions follow
```

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;
}
```

- Simple (terminal) forms / complex (non-terminal) forms
- Grammar = set of rules on how to construct complex forms from simpler forms
- Highly recursive.

Parse tree

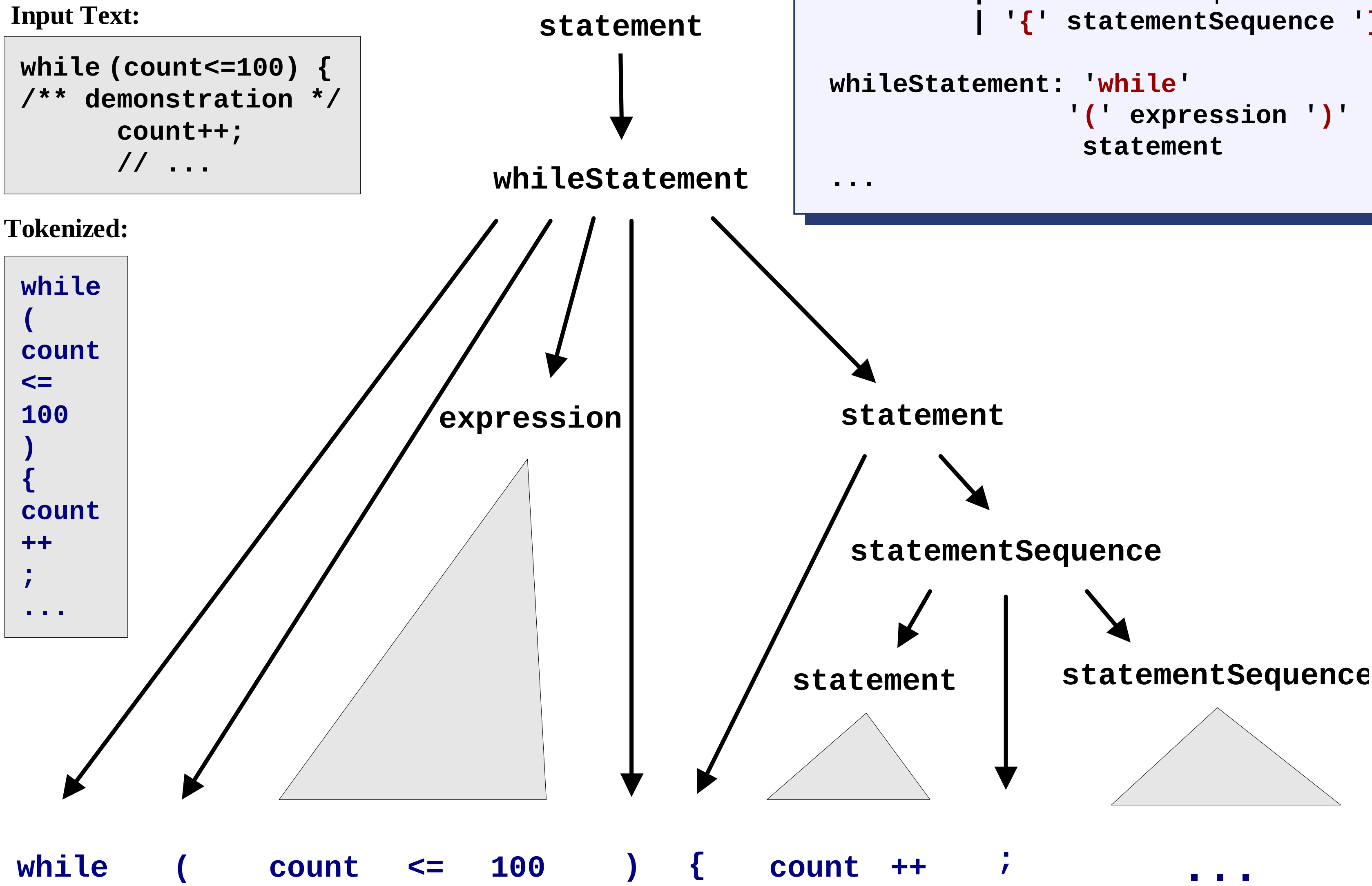
```
program:  statement;

statement: whileStatement
         | ifStatement
         | // other statement possibilities ...
         | '{' statementSequence '}'

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

Input Text:

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

Tokenized:

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



Recursive descent parsing

```
...
statement:   whileStatement
           | ifStatement
           | ...            // other statement possibilities follow
           | '{' statementSequence '}'

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

ifStatement: ...            // if definition comes here

statementSequence:  ''      // null, i.e. the empty sequence
                  | statement ';' statementSequence

expression: ...   // definition of an expression comes here

...               // more definitions follow
```

code sample

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

- Highly recursive
- LL(0) grammars: the first token determines in which rule we are
- In other grammars you have to look ahead 1 or more tokens
- Jack is almost LL(0).

Parser implementation: a set of parsing methods, one for each rule:

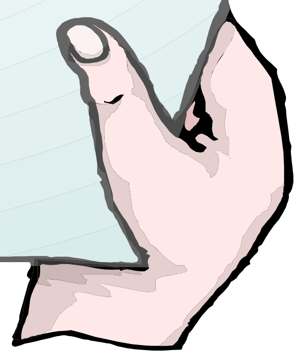
- **`parseStatement()`**
- **`parseWhileStatement()`**
- **`parseIfStatement()`**
- **`parseStatementSequence()`**
- **`parseExpression()`.**

A linguist view on parsing

Parsing:

One of the mental processes involved in sentence comprehension, in which the listener determines the syntactic categories of the words, joins them up in a tree, and identifies the subject, object, and predicate, a prerequisite to determining who did what to whom from the information in the sentence.

(Steven Pinker,
The Language Instinct)



The Jack grammar

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	'**class**' | '**constructor**' | '**function**' | '**method**' | '**field**' | '**static**' | '**var**' | '**int**' | '**char**' | '**boolean**' | '**void**' | '**true**' | '**false**' | '**null**' | '**this**' | '**let**' | '**do**' | '**if**' | '**else**' | '**while**' | '**return**'
symbol:	'**{**' | '**}**' | '**(**' | '**)**' | '**[**' | '**]**' | '**.**' | '**,**' | '**;**' | '**+**' | '**-**' | '*****' | '**/**' | '**&**' | '**|**' | '**<**' | '**>**' | '**=**' | '**~**'
integerConstant:	A decimal number in the range 0 .. 32767.
StringConstant	'"' A sequence of Unicode characters not including double quote or newline '"'
identifier:	A sequence of letters, digits, and underscore ('_') not starting with a digit.
Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:
class:	'**class**' className '**{**' classVarDec* subroutineDec* '**}**'
classVarDec:	('**static**' | '**field**') type varName (',' varName)* '**;**'
type:	'**int**' | '**char**' | '**boolean**' | className
subroutineDec:	('**constructor**' | '**function**' | '**method**') ('**void**' | type) subroutineName '**(**' parameterList '**)**' subroutineBody
parameterList:	((type varName) (',' type varName)*)?
subroutineBody:	'**{**' varDec* statements '**}**'
varDec:	'**var**' type varName (',' varName)* '**;**'
className:	identifier
subroutineName:	identifier
varName:	Identifier

'x': x appears verbatim
x: x is a language construct
x?: x appears 0 or 1 times
x*: x appears 0 or more times
x|y: either x or y appears
(x,y): x appears, then y.

The Jack grammar (cont.)

Statements:	
statements:	statement*
statement:	letStatement | ifStatement | whileStatement | doStatement | returnStatement
letStatement:	'**let**' varName ('**[**' expression '**]**')? '=' expression '**;**'
ifStatement:	'**if**' '**(**' expression '**)**' '**{**' statements '**}**' ('**else**' '**{**' statements '**}**')?
whileStatement:	'**while**' '**(**' expression '**)**' '**{**' statements '**}**'
doStatement:	'**do**' subroutineCall '**;**'
ReturnStatement	'**return**' expression? '**;**'
Expressions:	
expression:	term (op term)*
term:	integerConstant | stringConstant | keywordConstant | varName | varName '**[**' expression '**]**' | subroutineCall | '**(**' expression '**)**' | unaryOp term
subroutineCall:	subroutineName '**(**' expressionList '**)**' | (className | varName) '**.**' subroutineName '**(**' expressionList '**)**'
expressionList:	(expression ('**,**' expression)*)?
op:	'+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
unaryOp:	'-' | '~'
KeywordConstant:	'**true**' | '**false**' | '**null**' | '**this**'

'x':	x appears verbatim
x:	x is a language construct
x?:	x appears 0 or 1 times
x*:	x appears 0 or more times
x|y:	either x or y appears
(x,y):	x appears, then y.

Jack syntax analyzer in action

```
Class Bar {
   method Fraction foo(int y) {
     var int temp; // a variable
     let temp = (xxx+12)*-63;
      ...
   ...
```

Syntax analyzer

```
<varDec>
  <keyword> var </keyword>
  <keyword> int </keyword>
  <identifier> temp </identifier>
  <symbol> ; </symbol>
</varDec>
<statements>
  <letStatement>
    <keyword> let </keyword>
    <identifier> temp </identifier>
    <symbol> = </symbol>
    <expression>
       <term>
         <symbol> ( </symbol>
         <expression>
           <term>
             <identifier> xxx </identifier>
           </term>
           <symbol> + </symbol>
           <term>
             <int.Const.> 12 </int.Const.>
           </term>
    </expression>
    ...
```

Syntax analyzer

- Using the language grammar, a programmer can write a syntax analyzer program (parser)
- The syntax analyzer takes a source text file and attempts to match it on the language grammar
- If successful, it can generate a parse tree in some structured format, e.g. XML.

The syntax analyzer's algorithm shown in this slide:

- If xxx is non-terminal, output:

```
<xxx>
     Recursive code for the body of xxx
</xxx>
```

- If xxx is terminal (keyword, symbol, constant, or identifier) , output:

```
<xxx>
     xxx value
</xxx>
```

JackTokenizer: a tokenizer for the Jack language (proposed implementation)

JackTokenizer: Removes all comments and white space from the input stream and breaks it into Jack-language tokens, as specified by the Jack grammar.			
Routine	**Arguments**	**Returns**	**Function**
Constructor	input file / stream	--	Opens the input file/stream and gets ready to tokenize it.
`hasMoreTokens`	--	Boolean	Do we have more tokens in the input?
`advance`	--	--	Gets the next token from the input and makes it the current token. This method should only be called if *hasMoreTokens()* is true. Initially there is no current token.
`tokenType`	--	`KEYWORD`, `SYMBOL`, `IDENTIFIER`, `INT_CONST`, `STRING_CONST`	Returns the type of the current token.
`keyWord`	--	`CLASS`, `METHOD`, `FUNCTION`, `CONSTRUCTOR`, `INT`, `BOOLEAN`, `CHAR`, `VOID`, `VAR`, `STATIC`, `FIELD`, `LET`, `DO`, `IF`, `ELSE`, `WHILE`, `RETURN`, `TRUE`, `FALSE`, `NULL`, `THIS`	Returns the keyword which is the current token. Should be called only when `tokenType()` is `KEYWORD`.

JackTokenizer (cont.)

`symbol`	--	Char	Returns the character which is the current token. Should be called only when `tokenType()` is `SYMBOL`.
`identifier`	--	String	Returns the identifier which is the current token. Should be called only when `tokenType()` is `IDENTIFIER`
`intVal`		Int	Returns the integer value of the current token. Should be called only when `tokenType()` is `INT_CONST`
`stringVal`		String	Returns the string value of the current token, without the double quotes. Should be called only when `tokenType()` is `STRING_CONST`.

CompilationEngine: a recursive top-down parser for Jack

The CompilationEngine effects the actual compilation output.

It gets its input from a `JackTokenizer` and emits its parsed structure into an output file/stream.

The output is generated by a series of `compilexxx()` routines, one for every syntactic element `xxx` of the Jack grammar.

The contract between these routines is that each `compilexxx()` routine should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and output the parsing of `xxx`. Thus, `compilexxx()` may only be called if indeed `xxx` is the next syntactic element of the input.

In the first version of the compiler, which we now build, this module emits a structured printout of the code, wrapped in XML tags (defined in the specs of project 10). In the final version of the compiler, this module generates executable VM code (defined in the specs of project 11).

In both cases, the parsing logic and module API are exactly the same.

CompilationEngine (cont.)

Routine	Arguments	Returns	Function
Constructor	Input stream/file Output stream/file	--	Creates a new compilation engine with the given input and output. The next routine called must be `compileClass()`.
`CompileClass`	--	--	Compiles a complete class.
`CompileClassVarDec`	--	--	Compiles a static declaration or a field declaration.
`CompileSubroutine`	--	--	Compiles a complete method, function, or constructor.
`compileParameterList`	--	--	Compiles a (possibly empty) parameter list, not including the enclosing "`()`".
`compileVarDec`	--	--	Compiles a `var` declaration.

CompilationEngine (cont.)

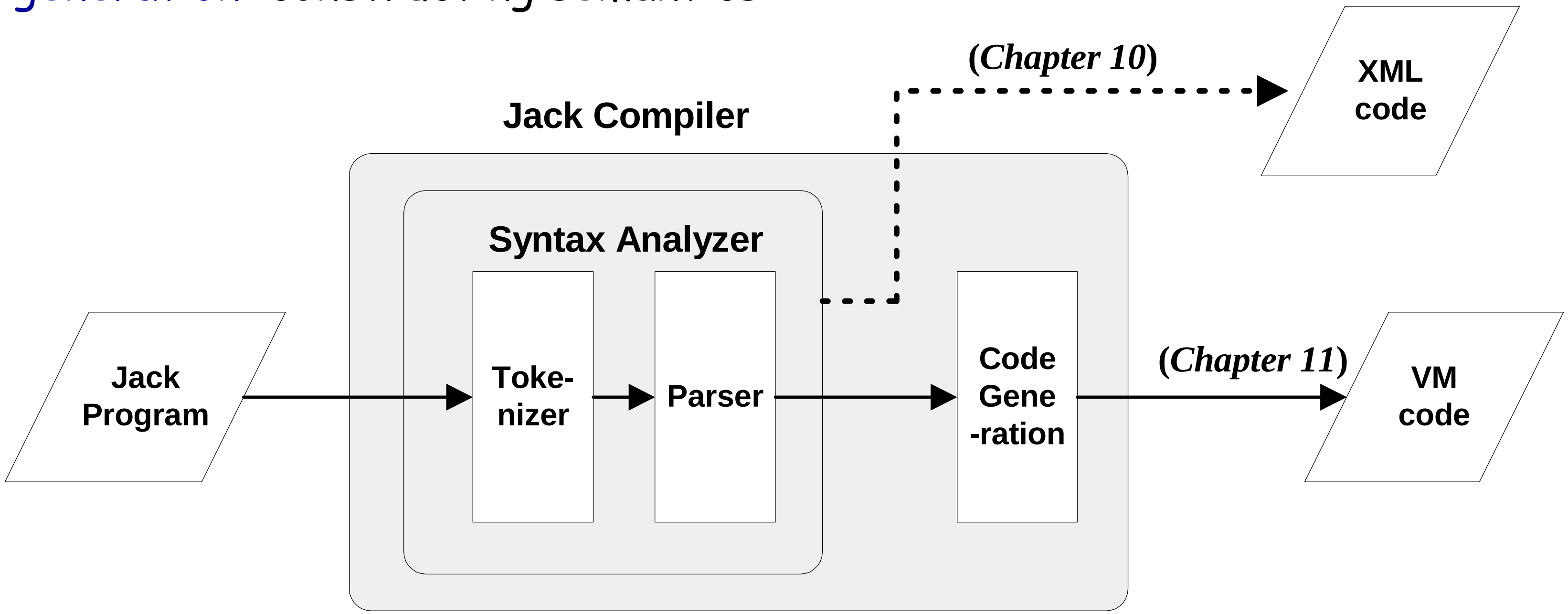
`compileStatements`	--	--	Compiles a sequence of statements, not including the enclosing "{}".
`compileDo`	--	--	Compiles a `do` statement.
`compileLet`	--	--	Compiles a `let` statement.
`compileWhile`	--	--	Compiles a `while` statement.
`compileReturn`	--	--	Compiles a `return` statement.
`compileIf`	--	--	Compiles an `if` statement, possibly with a trailing `else` clause.

CompilationEngine (cont.)

`CompileExpression`	--	--	Compiles an expression.
`CompileTerm`	--	--	Compiles a *term*. This routine is faced with a slight difficulty when trying to decide between some of the alternative parsing rules. Specifically, if the current token is an identifier, the routine must distinguish between a variable, an array entry, and a subroutine call. A single look-ahead token, which may be one of "[", "(", or "." suffices to distinguish between the three possibilities. Any other token is not part of this term and should not be advanced over.
`CompileExpressionList`	--	--	Compiles a (possibly empty) comma-separated list of expressions.

Summary and next step

- Syntax analysis: understanding syntax
- Code generation: constructing semantics



<u>The code generation challenge:</u>

- Extend the syntax analyzer into a full-blown compiler that, instead of generating passive XML code, generates executable VM code
- Two challenges: (a) handling data, and (b) handling commands.

Perspective

- The parse tree can be constructed on the fly
- Syntax analyzers can be built using:
  - **Lex** tool for tokenizing
  - **Yacc** tool for parsing
  - Do everything from scratch (our approach ...)
- The Jack language is intentionally simple:
  - Statement prefixes: **let**, **do**, ...
  - No operator priority
  - No error checking
  - Basic data types, etc.
- Richer languages require more powerful compilers
- The Jack compiler: designed to illustrate the key ideas that underlie modern compilers, leaving advanced features to more advanced courses
- Industrial-strength compilers:
  - Have good error diagnostics
  - Generate tight and efficient code
  - Support parallel (multi-core) processors.