Chapter 10:

Compiler I: Syntax Analysis

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken

This presentation contains lecture materials that accompany the textbook "The Elements of Computing Systems" by Noam Nisan & Shimon Schocken, MIT Press, 2005.

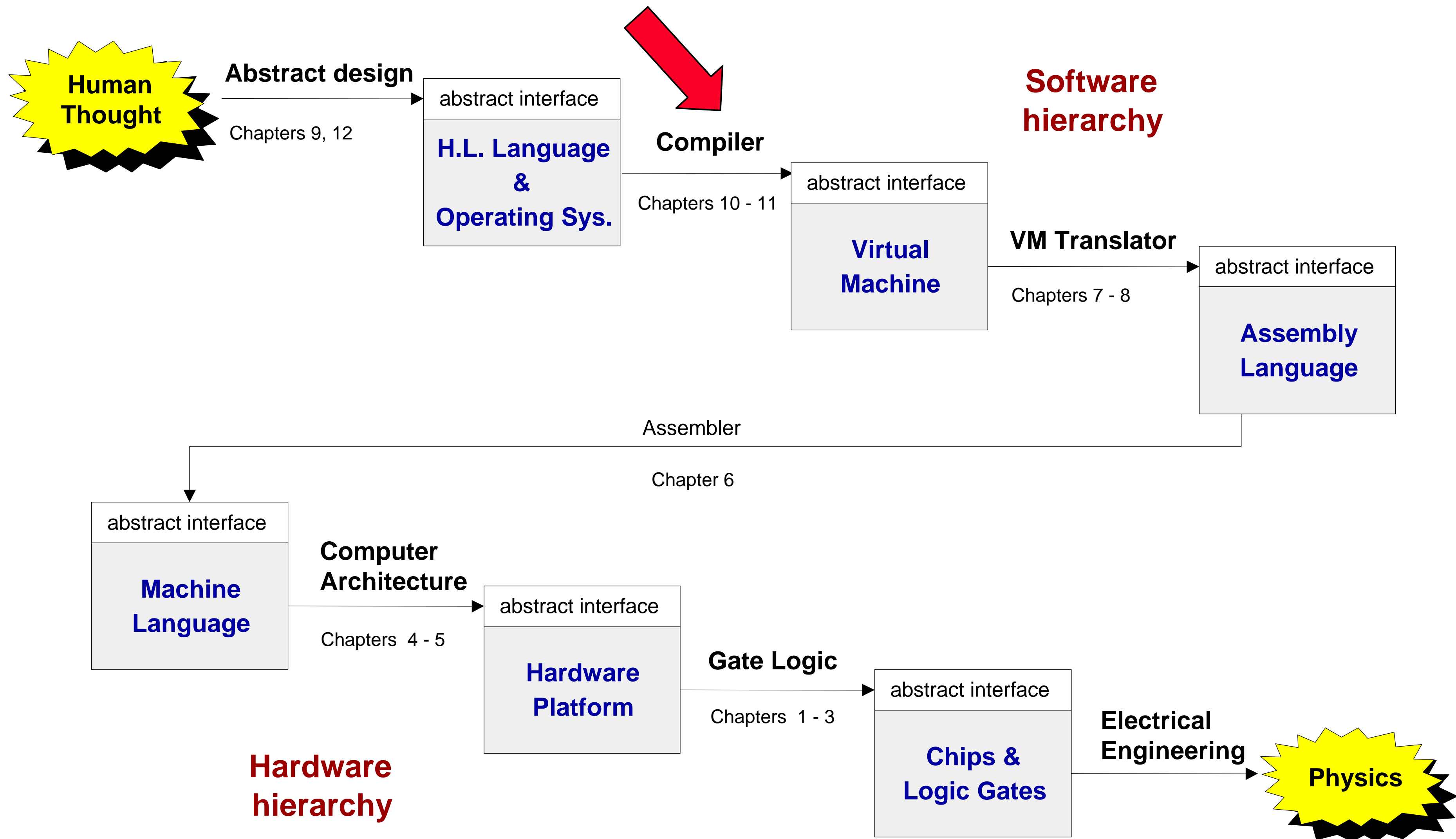
The book web site, www.idc.ac.il/tecs , features 13 such presentations, one for each book chapter. Each presentation is designed to support about 3 hours of classroom or self-study instruction.

You are welcome to use or edit this presentation for instructional and non-commercial purposes.

If you use our materials, we will appreciate it if you will include in them a reference to the book's web site.

And, if you have any comments, you can reach us at tecs.ta@gmail.com

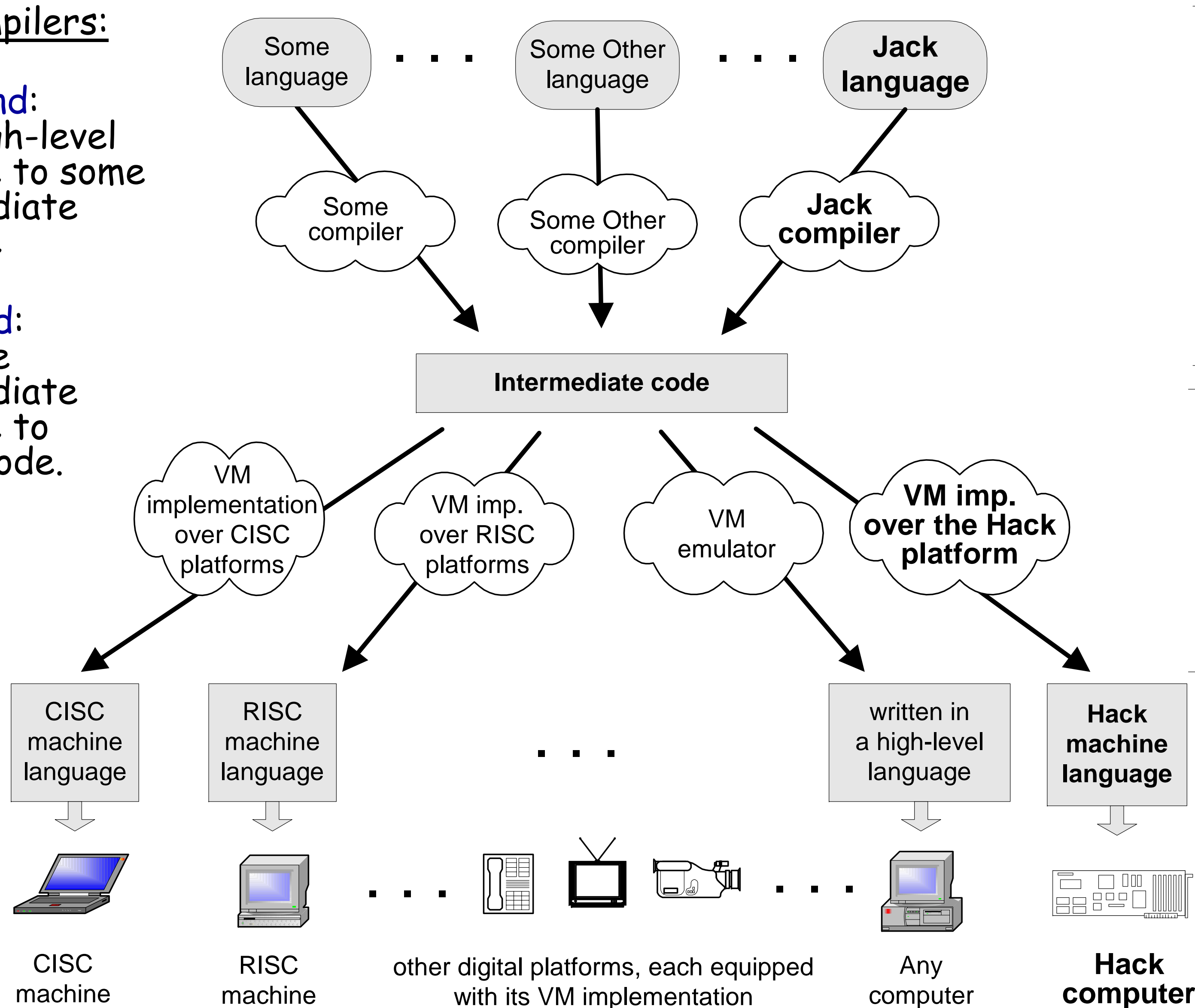
Course map



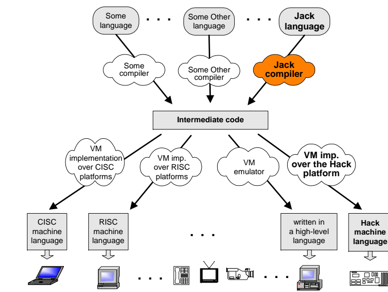
The big picture

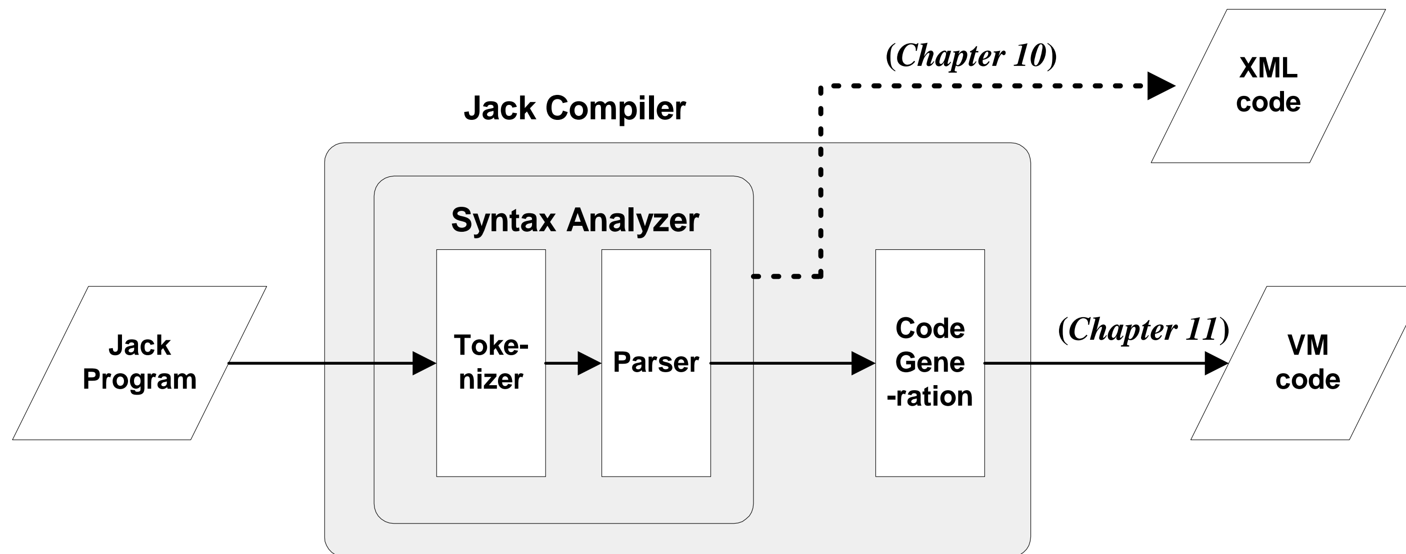
Modern compilers:

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



Compiler architecture (front end)





Front-end:

- Syntax analysis: understanding the semantics implied by the source code
  - ❑ Tokenizing: creating a list of "atoms"
- Code generation: reconstructing the semantics using a target syntax
  - ❑ Parsing: matching the atom list with the language grammar

  XML output = proof that the syntax analyzer is parsing correctly
- Code generation: reconstructing the semantics using the target syntax.

Tokenizing / Lexical analysis

Code fragment

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

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 token types:
    keyword, identifier, operator, constant, ...

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>
```

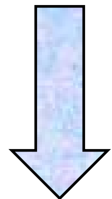
Parsing

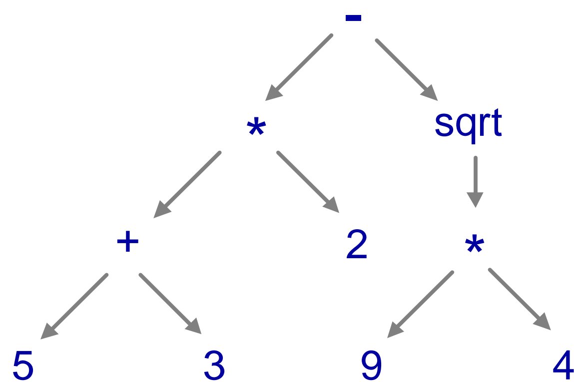
- Each language is characterized by a *grammar*
- A text is given:
  - The parser, using the grammar, can either accept or reject the text
  - In the process, the parser performs a complete analysis of the text
- The language can be:
  - Context-dependent (English, …)
  - Context-free (Jack, …).

Examples

context free

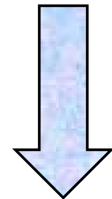
```
(5+3)*2 – sqrt(9*4)
```

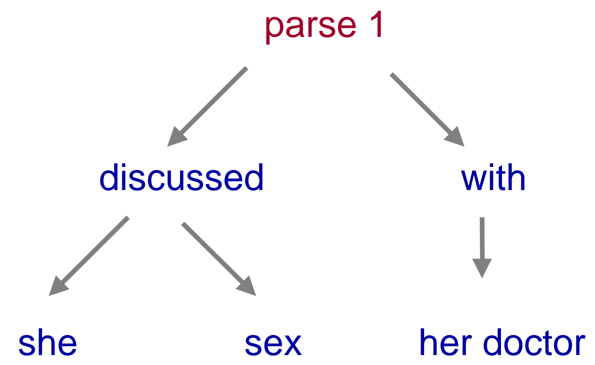


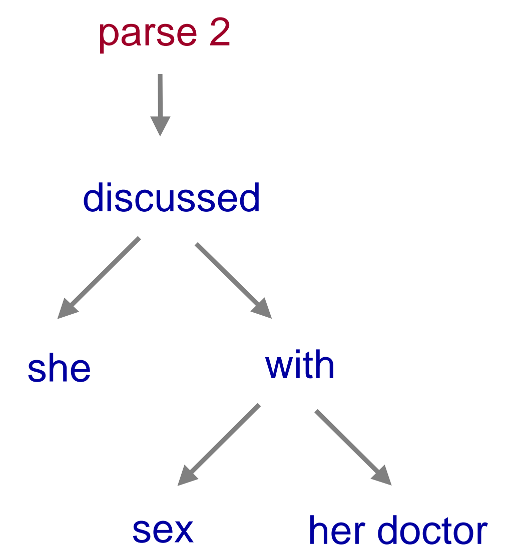


context dependent

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







A typical grammar (C/Java-like)

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

code sample

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

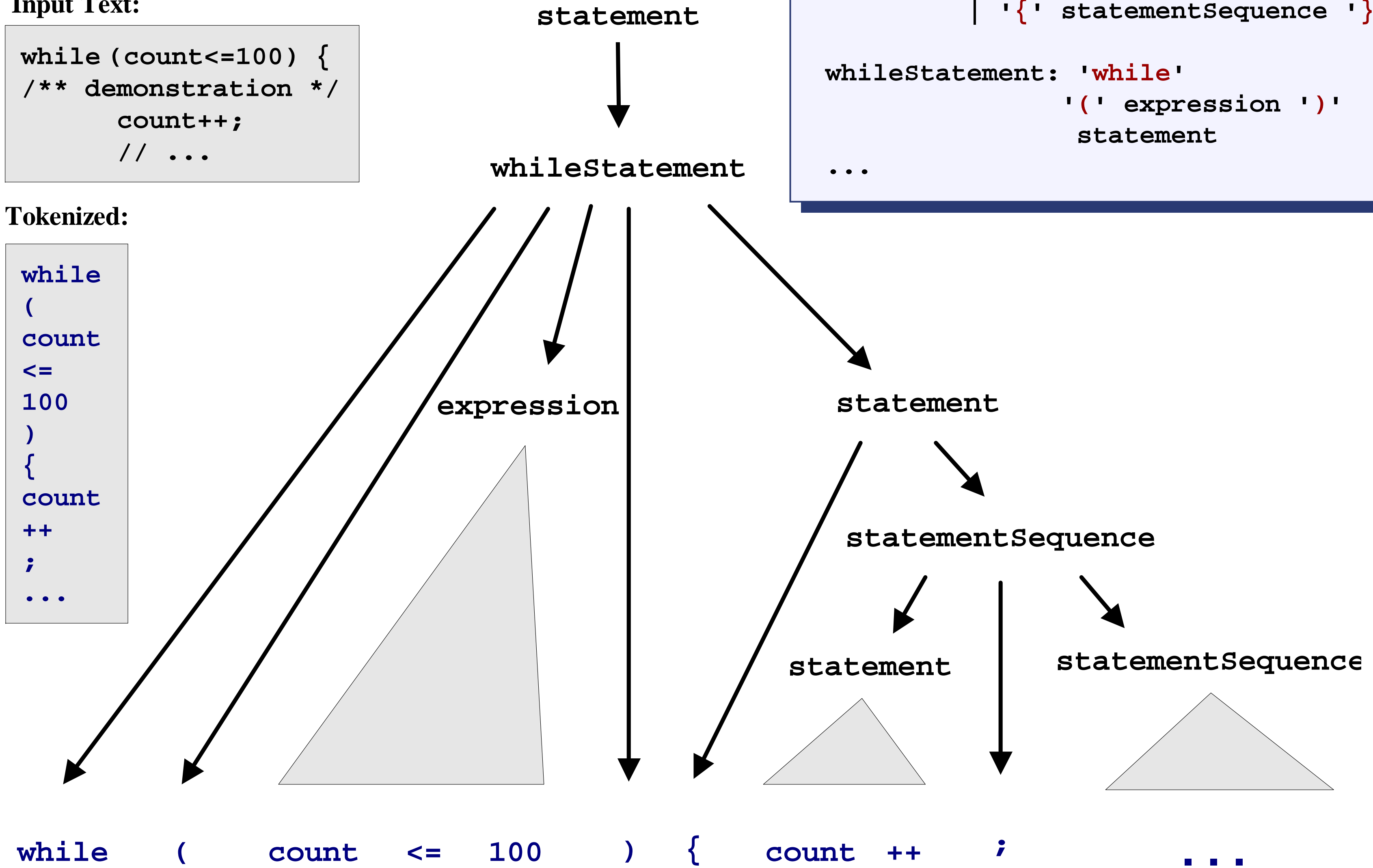
- Simple (terminal) forms / complex (non-terminal) forms
- Grammar = set of rules on how to construct
  complex forms from simpler forms
- Highly recursive.

Parse tree

```
program:   statement;

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

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



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 (some expression) {
  some statement;
  some statement;
     while (some expression) {
        while (some expression)
               some statement;
        some 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).

- `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
- The syntax analyzer takes a source text file and attempts to match it on the language grammar
- If successful, it generates a parse tree in some structured format, e.g. XML.

This syntax analyzer's algorithm:

- If xxx is non-terminal, output:

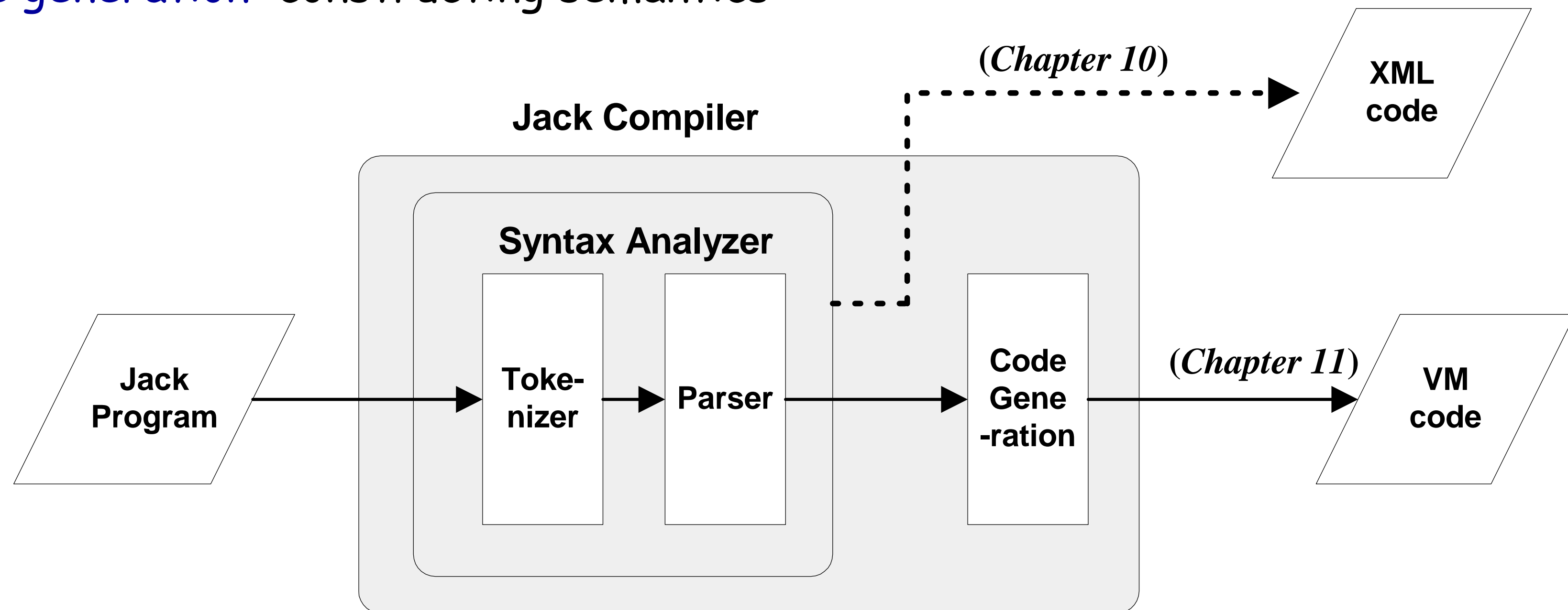
```
<xxx>
   Recursive code for the body of xxx
</xxx>
```

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

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

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
- Bottom up compilation is harder and more powerful
- Syntax analyzers are typically built using tools like:
  - `Lex` for tokenizing
  - `Yacc` for parsing
- The Jack language is intentionally simple:
  - Statement prefixes: `let`, `do`, ...
  - No operator priority
  - No error checking
  - Basic data types, etc.
- Typical languages are richer, requiring more powerful compilers
- The Jack compiler: designed to illustrate the key ideas that underlie modern compilers, leaving advanced features to more advanced courses.