Chapter 10

Compiler I: Parsing

These slides support chapter 10 of the book
The Elements of Computing Systems
By Noam Nisan and Shimon Schocken
MIT Press

Compilation (one-tier)

high-level program

```
/** Tests the Point class. */
class Main {
  function void main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();
     // Prints (4,6)
     do Output.println();
     do Output.printInt(p1.dist(p3));
     // Prints 5
     return;
  }
}
```

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax,
                        int ay) {
      let x = ax;
      let y = ay;
      let pointCount =
          pointCount + 1;
      return this;
  }
  // ... more Point methods
```

machine code

```
...
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

Compilation (two-tier)

high-level program

```
/** Tests the Point class. */
class Main {
  function void main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();
     // Prints (4,6)
     do Output.println();
     do Output.printInt(p1.dist(p3));
     // Prints 5
     return;
  }
}
```

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax,
                        int ay) {
      let x = ax;
      let y = ay;
      let pointCount =
          pointCount + 1;
      return this;
  }
  // ... more Point methods
```

compiler

VM code

```
...
push local 0
push constant 0
eq
not
push argument 1
neg
push
call
pop a
push
neg
push
push
push
add
push
push
...
```

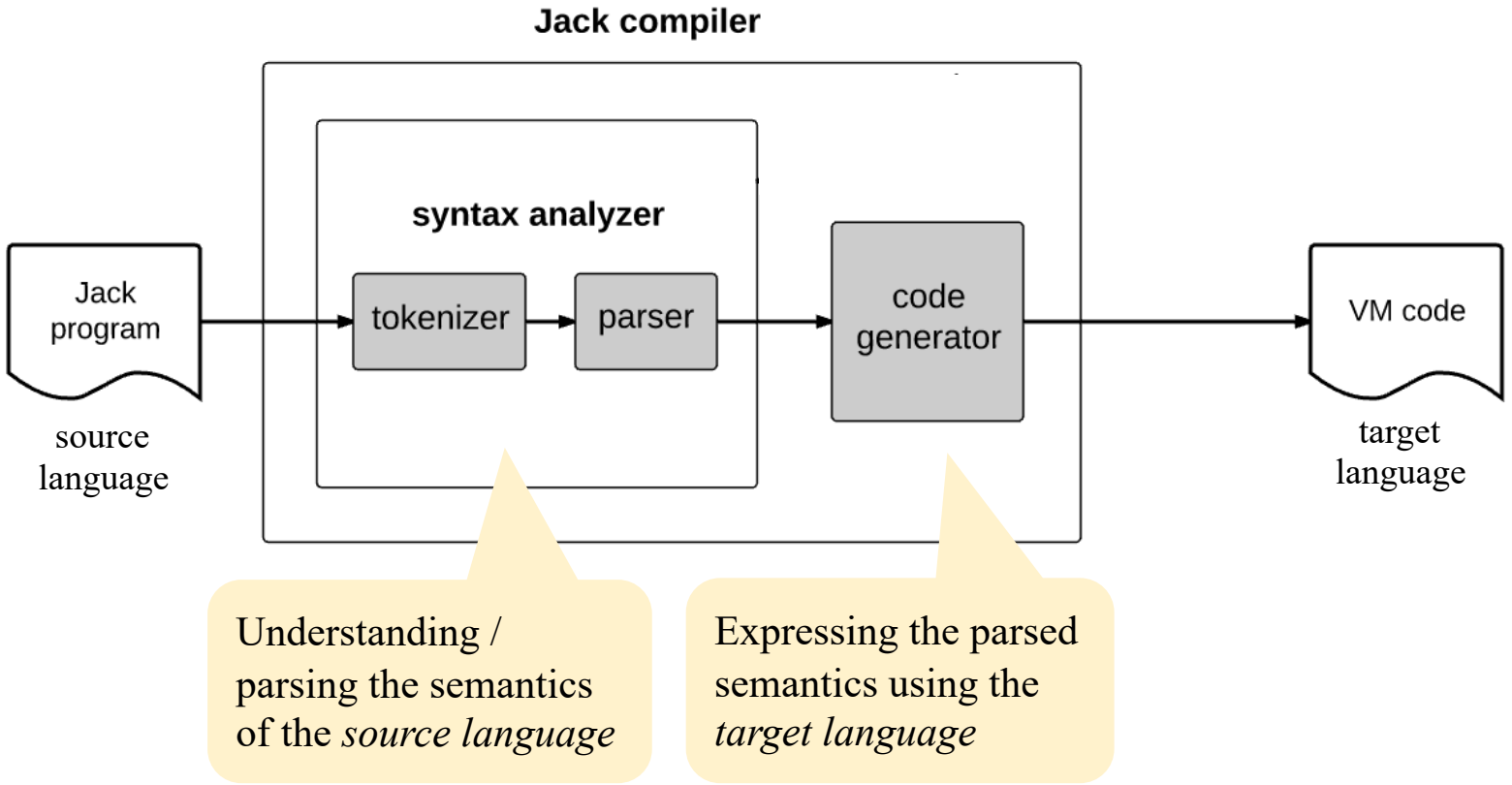
```
...
push constant 2
push local 0
pop argument 1
push local 0
push constant 0
not
push local 1
push local 0
push local 1
add
pop argument 3
push local 1
neg
push local 0
push local 1
push local 2
add
...
```

VM translator

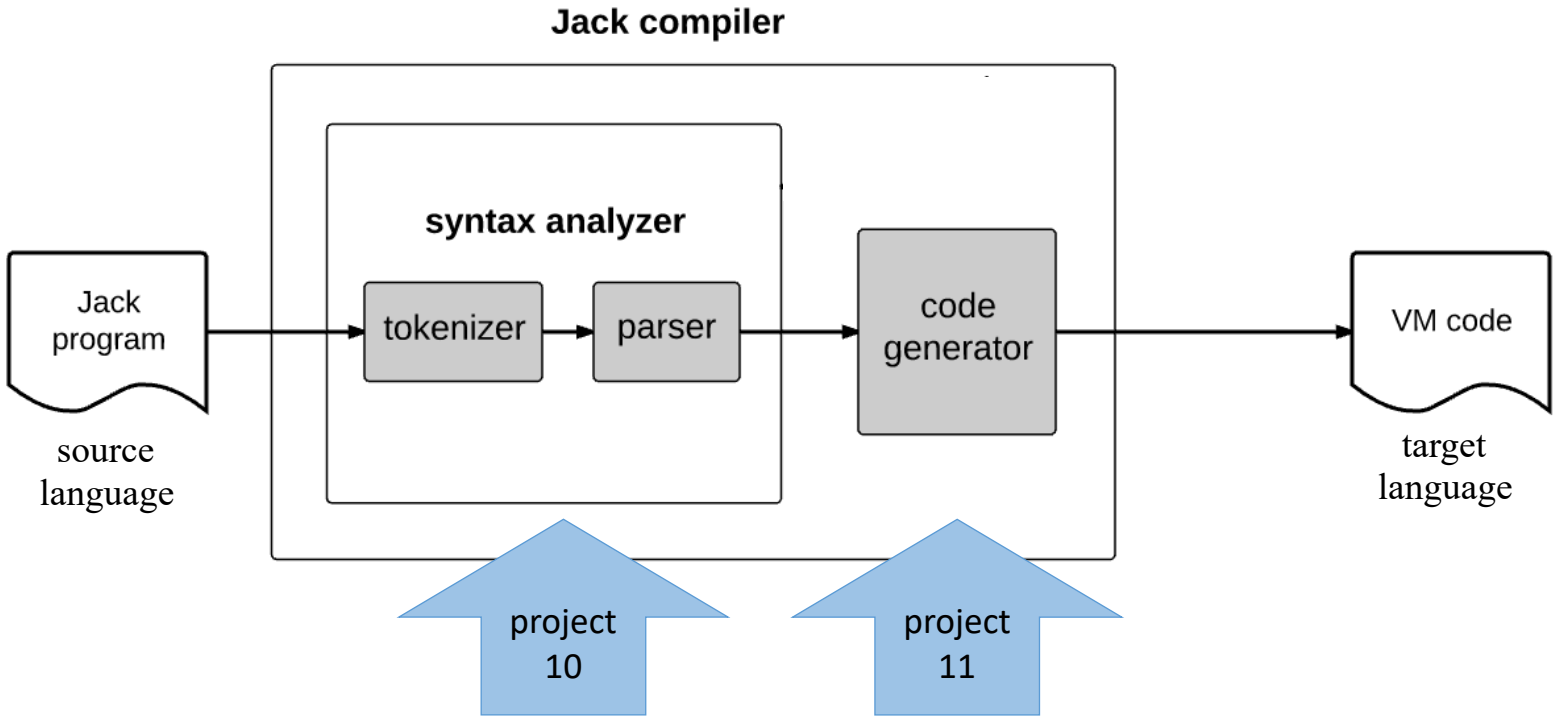
machine code

```
...
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

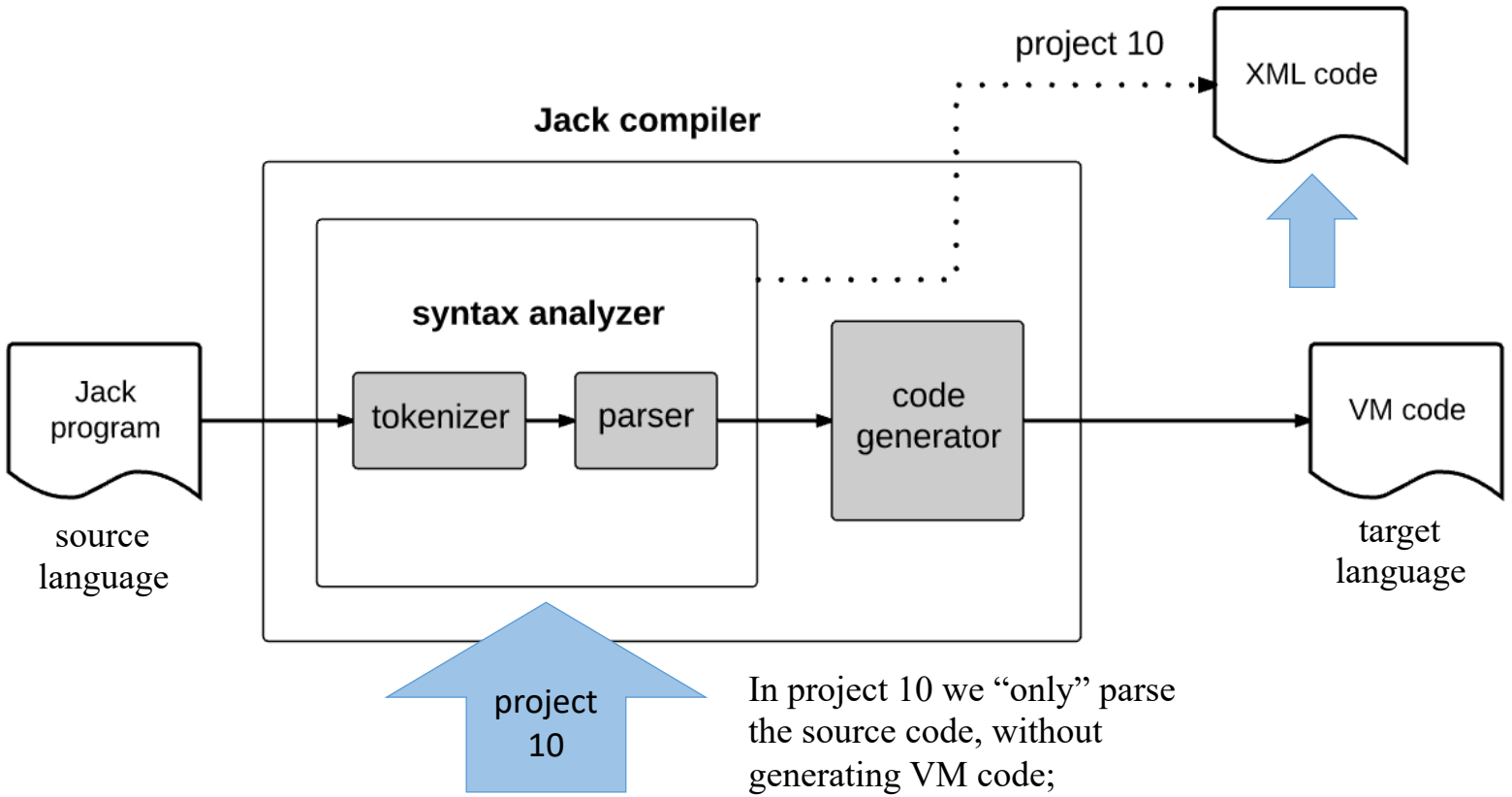
Compiler development roadmap



Compiler development roadmap



Compiler development roadmap



How can we tell that the syntax analyzer operates correctly?

We'll generate XML code, and use it to unit-test syntax analyzer.

Compiler I / Syntax Analysis

Take home lessons:

- Tokenizing
- Grammars
- Parsing
- Parse trees
- XML / mark-up
- Handling structured data files

Common techniques in numerous applications and R&D projects

Lecture plan

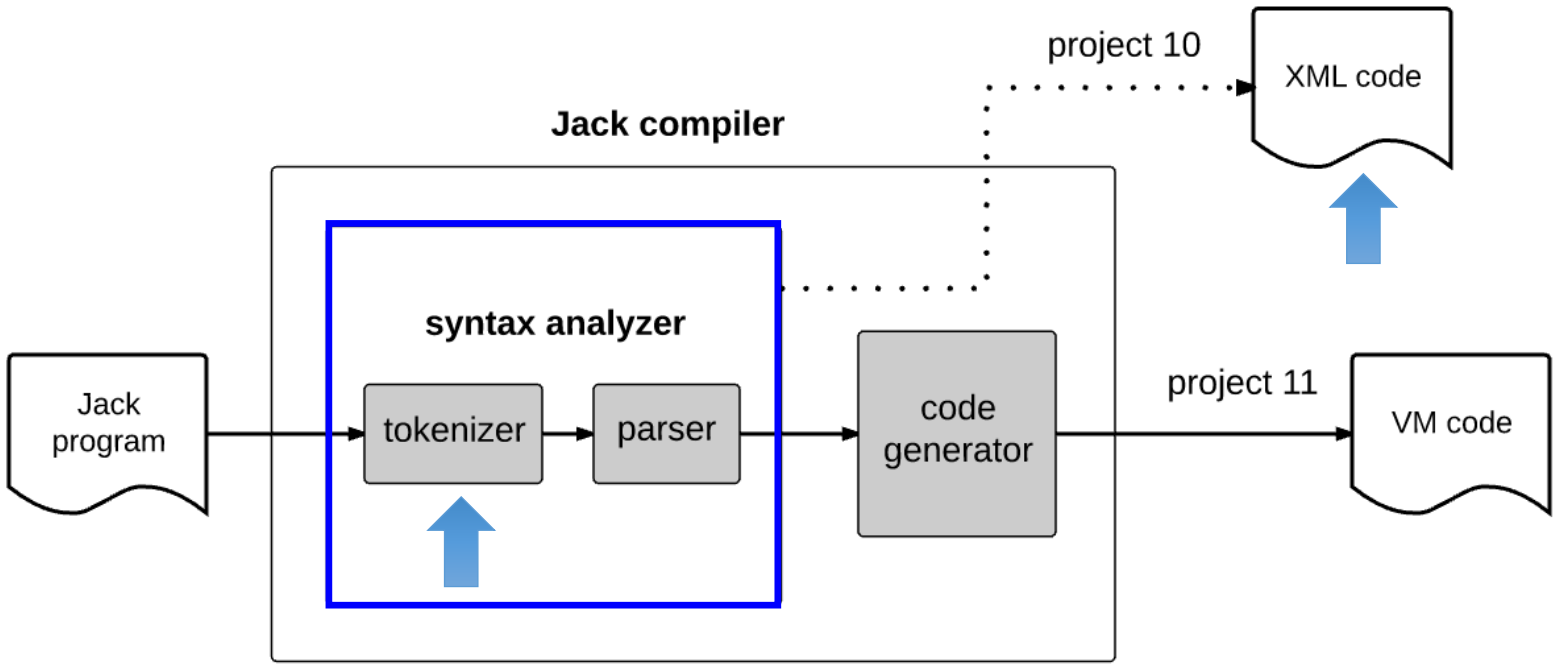
Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

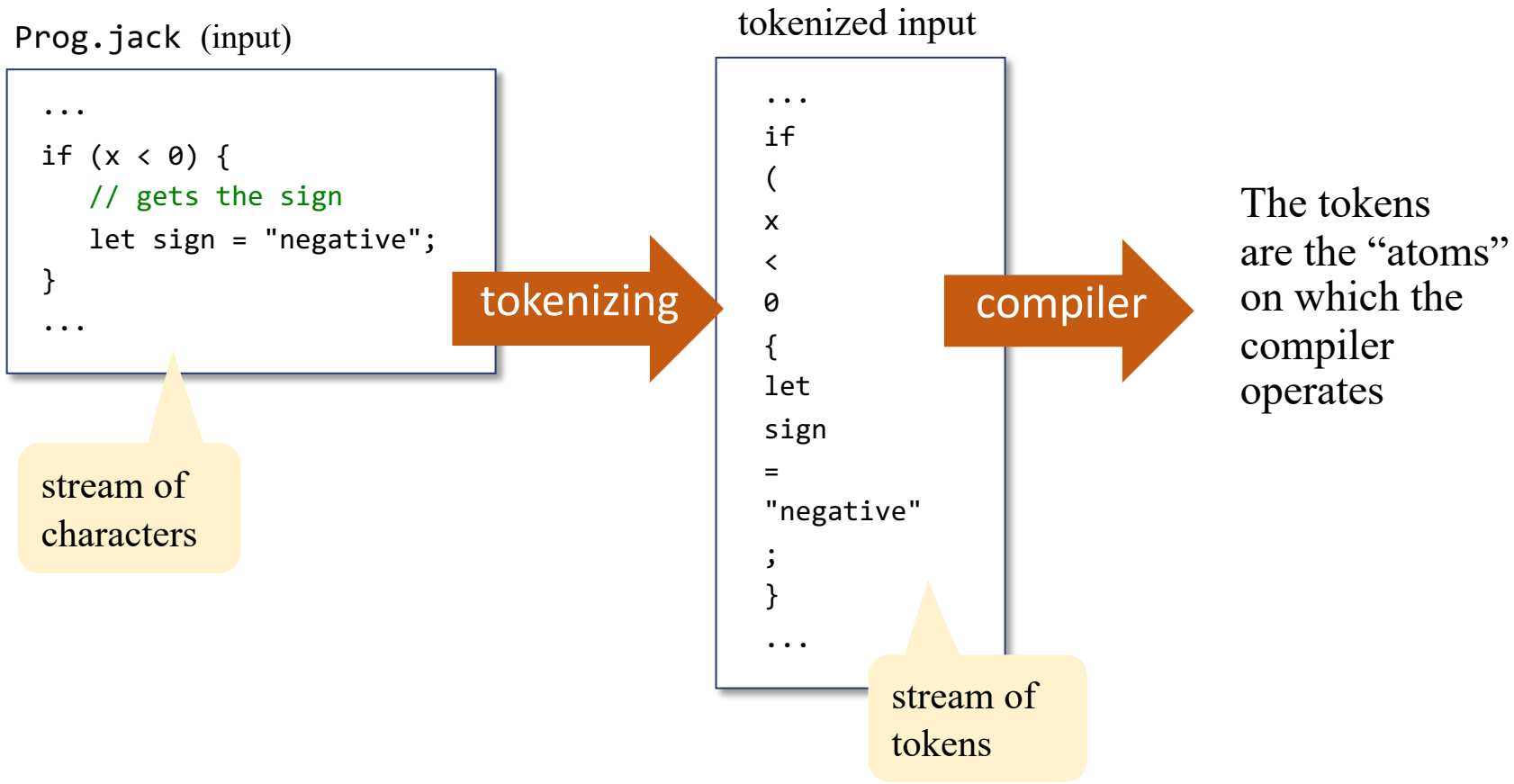
Tokenizing



Plan

- Develop a tokenizer
- Develop a program that tests the tokenizer by generating structured XML output that represents the source code's *tokens*

Tokenizing



- Lexicon = a group of valid tokens
- Each programming language specifies its lexicon.

Jack lexicon

`Prog.jack` (input)

```
...
if (x < 0) {
   // gets the sign
   let sign = "negative";
}
...
```

- keywords
- symbols
- integers
- strings
- identifiers

Jack tokenizer

Prog.jack (input)

```
...
if (x < 0) {
   // gets the sign
   let sign = "negative";
}
...
```

TokenizerTest

output

```
...
<keyword>     if         </keyword>
<symbol>      (          </symbol>
<identifier>  x          </identifier>
<symbol>      <          </symbol>
<intConst>    0          </intConst>
<symbol>      )          </symbol>
<symbol>      {          </symbol>
<keyword>     let        </keyword>
<identifier>  sign       </identifier>
<symbol>      =          </symbol>
<stringConst> negative   </stringConst>
<symbol>      ;          </symbol>
<symbol>      }          </symbol>
...
```

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.

Tokenizer services:

- Checks if there are more tokens in the input
- Gets the current token, and advances the input
- Gets the token's type

  (complete API, later)

Jack tokenizer

Prog.jack (input)

```
...
if (x < 0) {
   // gets the sign
   let sign = "negative";
}
...
```

TokenizerTest

output

```
...
<keyword>     if         </keyword>
<symbol>      (          </symbol>
<identifier>  x          </identifier>
<symbol>      <          </symbol>
<intConst>    0          </intConst>
<symbol>      )          </symbol>
<symbol>      {          </symbol>
<keyword>     let        </keyword>
<identifier>  sign       </identifier>
<symbol>      =          </symbol>
<stringConst> negative   </stringConst>
<symbol>      ;          </symbol>
<symbol>      }          </symbol>
...
```

TokenizerTest (pseudo code)

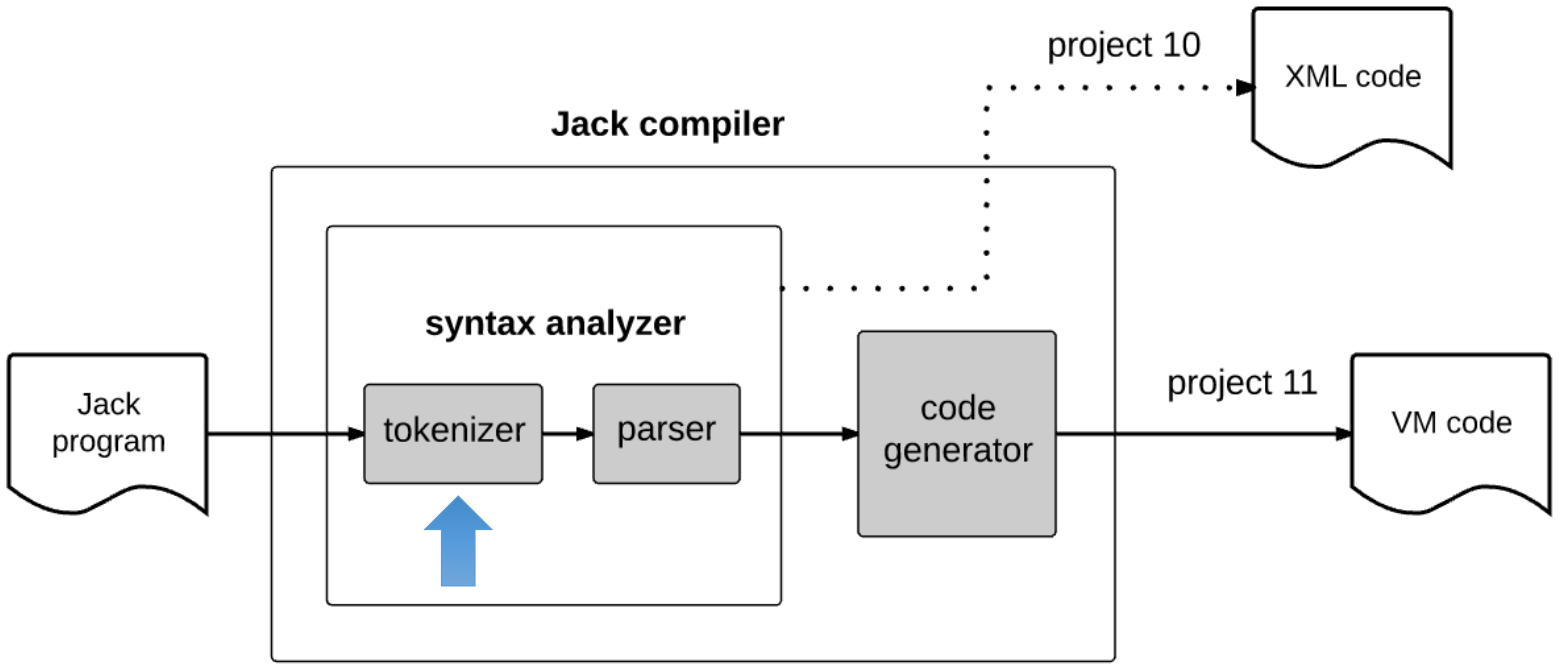
```
tknzr = new JackTokenizer("Prog.jack")
tknzr.advance(); // gets the first token
while tknzr.hasMoreTokens() {
   tokenType = type of the current token
   print "<" + tokenType + ">"
   print the current token
   print "</" + tokenType + ">"
   print newLine
   tknzr.advance();
}
```

Tokenizer services:

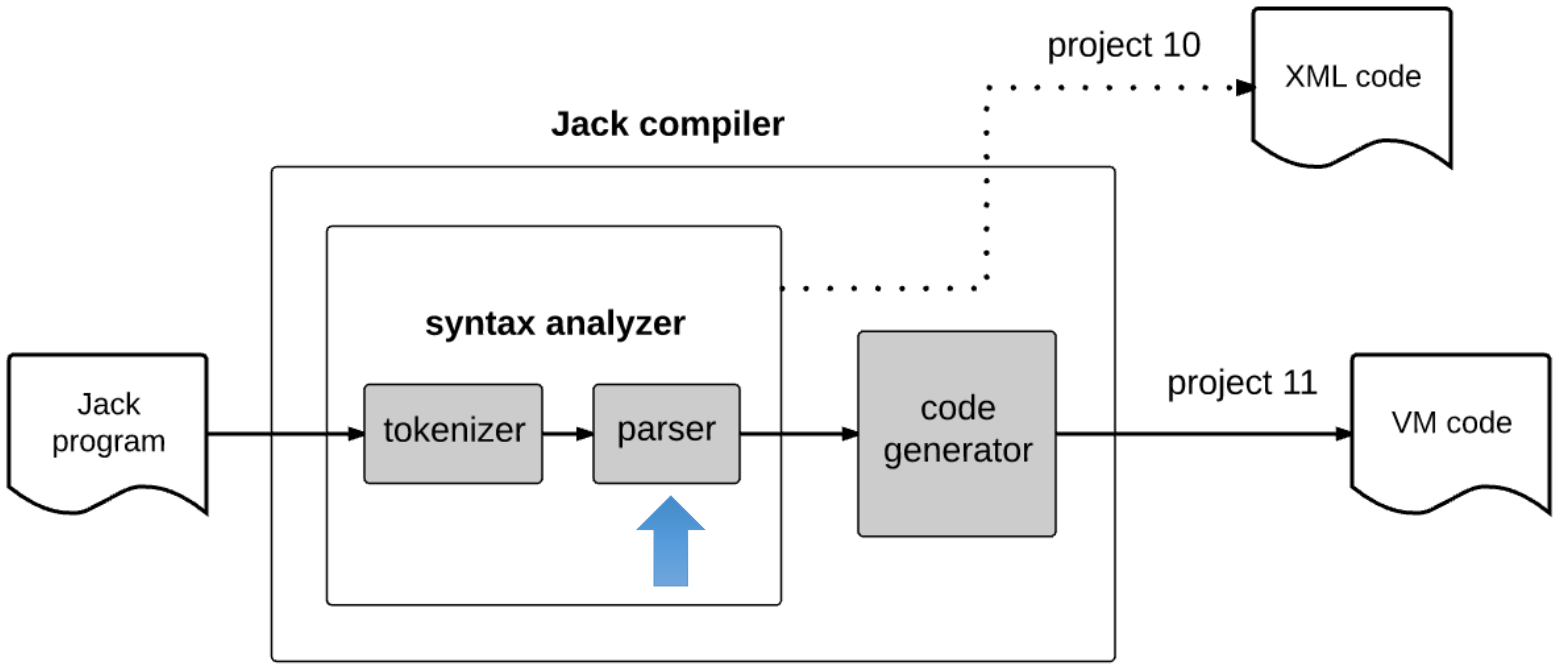
- Checks if there are more tokens in the input
- Gets the current token, and advances the input
- Gets the token's type

  (complete API, later)

Compiler development roadmap



Compiler development roadmap



Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Grammar

Lexicon: a set of valid words

Grammar: a set of *rules*, describing how words can be combined to form valid sentences

sentence: *nounPhrase verbPhrase*
nounPhrase: *determiner? noun*
verbPhrase: *verb nounPhrase*
noun: 'dog' | 'school' | 'dina' | 'he' | 'she' | 'homework' | ...
verb: 'went' | 'ate' | 'said' | ...
determiner: 'the' | 'to' | 'my' | ...
. . .

dina went to school

she said

the dog ate my homework

Each one of the three sentences is accepted (correctly parsed) by the grammar.

Grammar

Lexicon: a set of valid words

Grammar: a set of *rules*, describing how words can be combined to form valid sentences

```
    sentence:  nounPhrase  verbPhrase
nounPhrase:  determiner?  noun
verbPhrase:  verb  nounPhrase
        noun:  'dog' | 'school' | 'dina' | 'he' | 'she' | 'homework' | ...
        verb:  'went' | 'ate' | 'said' | ...
   determiner:  'the' | 'to' | 'my' | ...
         . . .
```

Each rule has a *left hand-side* and a *right hand-side*

LHS: The rule name

Terminal rules: Their RHS specify no rule names

Nonterminal rules: All the other rules.

Grammar

Jack grammar (subset)

statement: *letStatement* |
ifStatement |
whileStatement

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**'
'**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**'
'**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

*x**: *x* appears 0 or more times

x? : *x* appears 0 or 1 times

Grammar

Jack grammar (subset)

statement: *letStatement* |
ifStatement |
whileStatement

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**'
'**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**'
'**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Nonterminals

Terminals

Grammar

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Input examples

```
let x = 100;
```

Grammar

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Input examples

```
let x = 100;
```

```
let x = x + 1;
```

Grammar

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Input examples

```
let x = 100;
```

```
let x = x + 1;
```

```
while (n < lim)
   let n = n + 1;
}
```

Grammar

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Input examples

```
let x = 100;
```

```
let x = x + 1;
```

```
while (n < lim)
   let n = n + 1;
}
```

```
if (x = 1) {
   let x = 100;
   let x = x + 1;
}
```

Grammar

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Input examples

```
while (lim < 100) {
   if (x = 1) {
      let z = 100;
      while (z > 0) {
         let z = z - 1;
      }
   }
   let lim = lim + 10;
}
```

Parsing:

- Determining if a given input is accepted by the grammar
- In the process, uncovering the grammatical structure of the input.

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Parse tree

sentence: *nounPhrase verbPhrase*

nounPhrase: *determiner*? *noun*

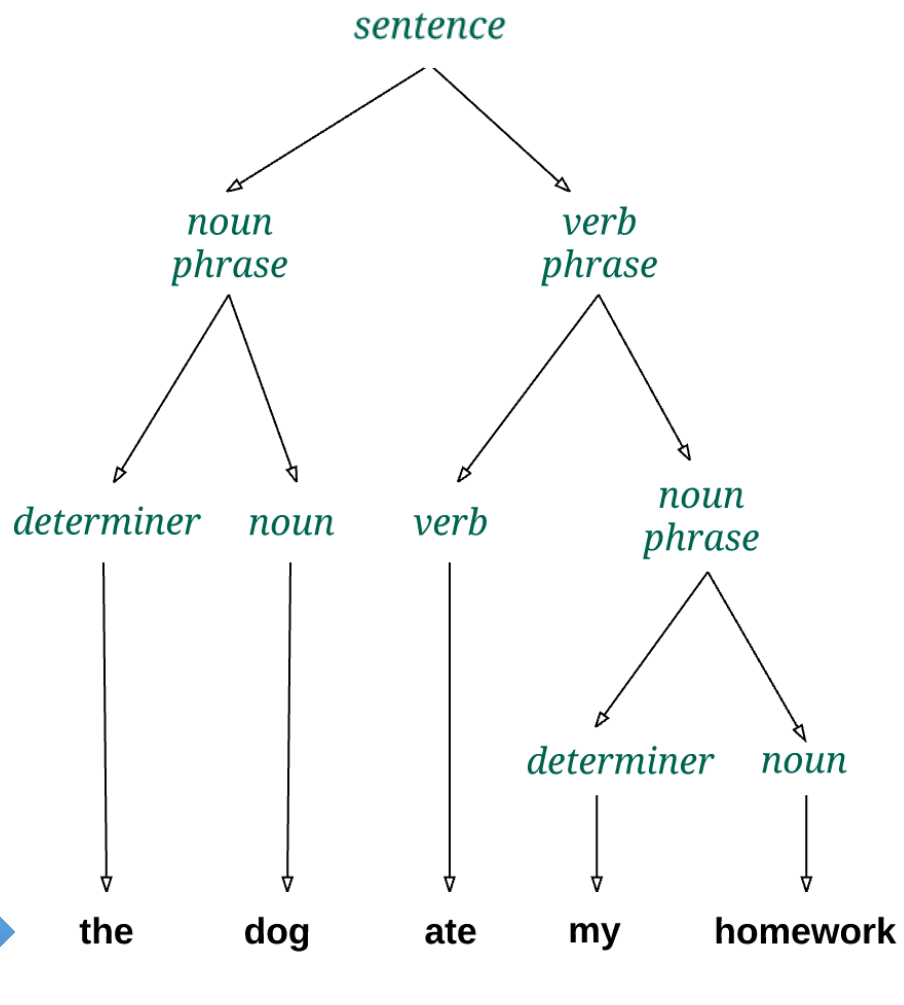
verbPhrase: *verb nounPhrase*

noun: 'dog' | 'school' | 'dina' | 'he' | 'she' | 'homework' | ...

verb: 'went' | 'ate' | 'said' | ...

determiner: 'the' | 'to' | 'my' | ...

. . .



Parsing

- Determining if the given input is accepted by the grammar
- In the process, constructing the grammatical structure of the input

Parse tree

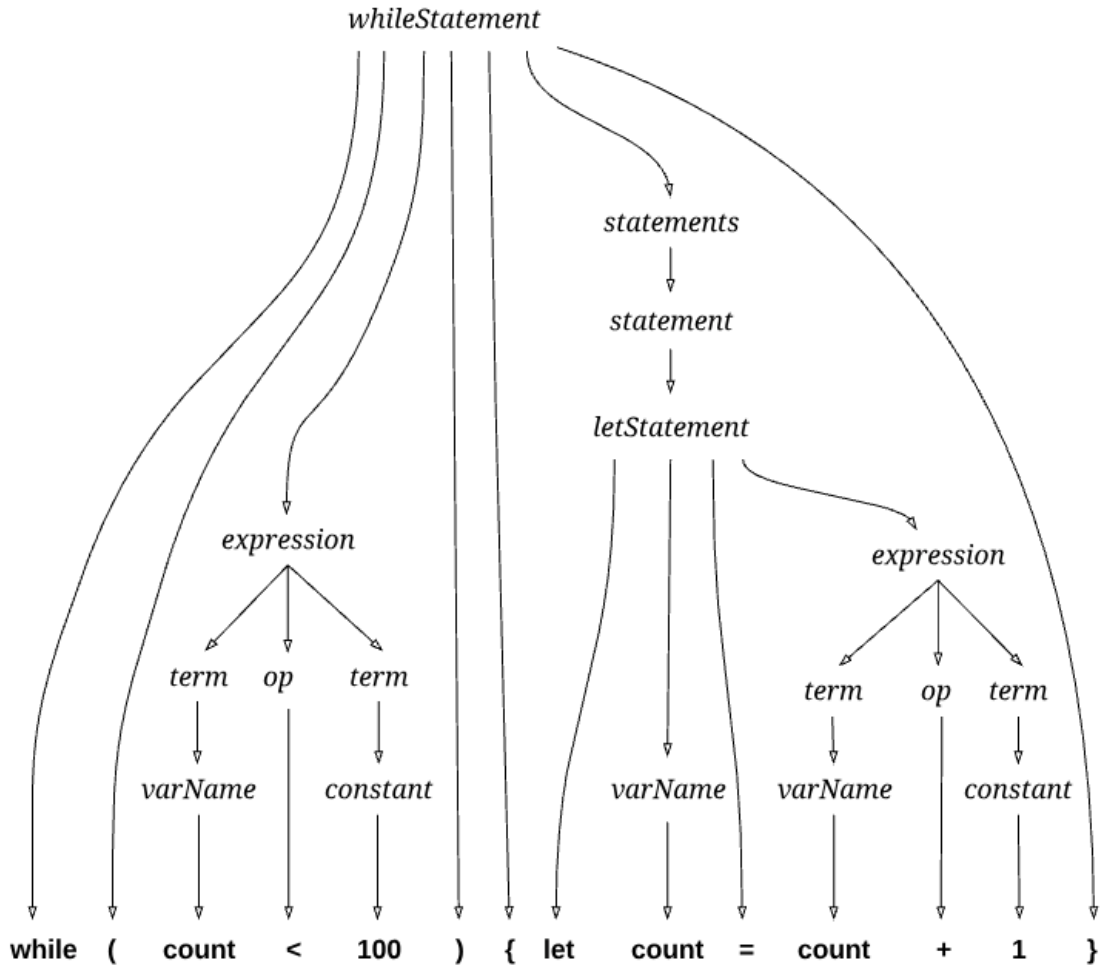
Jack grammar (subset)

statement:	*letStatement* | *ifStatement* | *whileStatement*
statements:	*statement**
letStatement:	'**let**' *varName* '**=**' *expression* '**;**'
ifStatement:	'**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'
whileStatement:	'**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'
expression:	*term* (*op term*)?
term:	*varName* | *constant*
varName:	a string not beginning with a digit
constant:	a decimal number
op:	'**+**' | '**-**' | '**=**' | '**>**' | '**<**'

`prog.jack` (input)

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

parser



Parse tree

Jack grammar (subset)

statement:	*letStatement* | *ifStatement* | *whileStatement*
statements:	*statement**
letStatement:	**'let'** *varName* **'='** *expression* **';'**
ifStatement:	**'if'** **'('** *expression* **')'** **'{'** *statements* **'}'**
whileStatement:	**'while'** **'('** *expression* **')'** **'{'** *statements* **'}'**
expression:	*term* (*op term*)?
term:	*varName* | *constant*
varName:	a string not beginning with a digit
constant:	a decimal number
op:	**'+'** | **'-'** | **'='** | **'>'** | **'<'**

`prog.jack` (input)

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

Same parse tree, expressed linearly using mark-up tags

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      <term> <constant> 100 </constant> </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <statement> <letStatement>
         <keyword> let </keyword>
         <varName> count </varName>
         <symbol> = </symbol>
         <expression>
            <term> <varName> count </varName> </term>
            <op> <symbol> + </symbol> </op>
            <term> <constant> 1 </constant> </term>
         </expression>
         <symbol> ; </symbol>
      </letStatement> </statement>
   </statements>
   <symbol> } </symbol>
</whileStatement>
...
```

The output that the JackAnalyzer is expected to generate (project 10)

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Syntax analyzer

`prog.jack` (input)

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

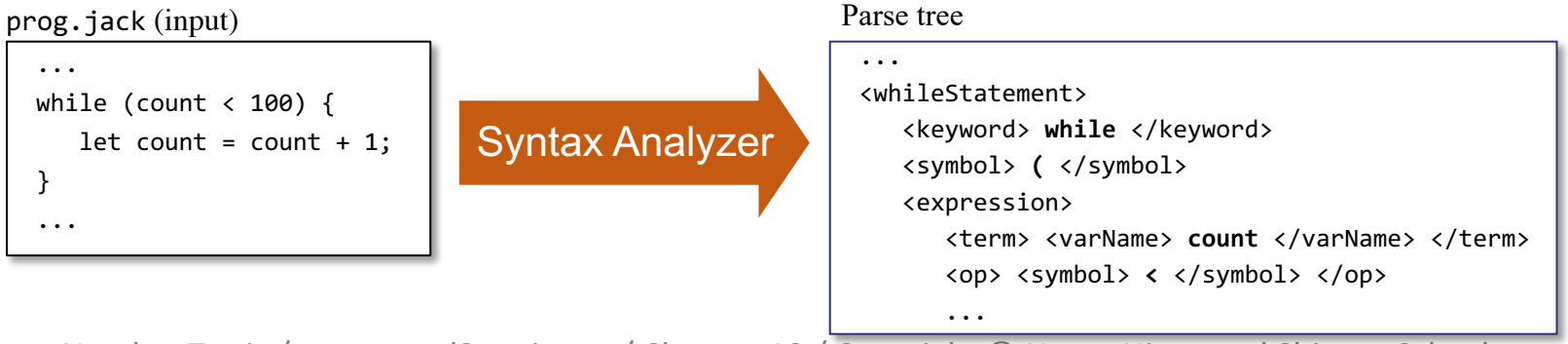
Parse tree

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      <term> <constant> 100 </constant> </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <statement> <letStatement>
         <keyword> let </keyword>
         <varName> count </varName>
         <symbol> = </symbol>
         <expression>
            <term> <varName> count </varName> </term>
            <op> <symbol> + </symbol> </op>
            <term> <constant> 1 </constant> </term>
         </expression>
         <symbol> ; </symbol>
      </letStatement> </statement>
   </statements>
   <symbol> } </symbol>
</whileStatement>
...
```

Syntax analyzer

The Syntax Analyzer

- A computer program
- Designed to accept / reject a given input (program)
- In the process, builds the program's parse tree (XML)
- ***Designed according to the language grammar***



Syntax analyzer design

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

The Syntax Analyzer

- A computer program
- Designed to accept / reject a given input (program)
- In the process, builds the program's parse tree (XML)
- ***Designed according to the language grammar***

`prog.jack` (input)

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

Syntax Analyzer

Parse tree

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      ...
```

Syntax analyzer design

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Syntax analyzer implementation

```
class CompilationEngine {
    ...
    compileLet() {
        // code for compiling a let statement
    }
    ...
    compileWhile() {
        // code for compiling a while statement
    }
    ...
    compileTerm() {
        // code for compiling a term
    }
    ...
}
```

We call the main module of the syntax analyzer

`CompilationEngine`

`prog.jack` (input)

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

Syntax Analyzer

Parse tree

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      ...
```

Syntax analyzer design

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Syntax analyzer implementation

```
class CompilationEngine {
    ...
    compileLet() {
        // code for compiling a let statement
    }
    ...
    compileWhile() {
        // code for compiling a while statement
    }
    ...
    compileTerm() {
        // code for compiling a term
    }
    ...
}
```

We call the main module of the syntax analyzer `CompilationEngine`

Consists of a set of `compile`*xxx* methods,

One for each grammar rule *xxx*

How to implement these parsing methods?

`prog.jack` (input)

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

Syntax Analyzer

Parse tree

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      ...
```

Syntax analyzer design

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Implementation of a typical parsing method:

```
// This method implements the rule whileStatement:
// 'while' '(' expression ')' '{' statements '}'
// It should be called if the current token is 'while'.
compileWhile()
   print("<whileStatement>")
   process("while")
   process("(")
   compileExpression()
   process(")")
   process("{")
   compileStatements()
   process("}")
   print("</whileStatement>")
```

(psuedo code)

```
// A helper method that handles
// the current token, and advances
// to get the next token.
process(str)
   if (currentToken == str)
      printXMLToken(str)
   else
      print("syntax error")
   currentToken =
      tokenizer.advance()
```

prog.jack (input)

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

Syntax Analyzer

Parse tree

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      ...
```

Syntax analyzer design

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

Implementation of a typical parsing method:

```
// This method implements the rule whileStatement:
// 'while' '(' expression ')' '{' statements '}'
// It should be called if the current token is 'while'.
compileWhile()
   print("<whileStatement>")
   process("while")
   process("(")
   compileExpression()
   process(")")
   process("{")
   compileStatements()
   process("}")
   print("</whileStatement>")
```

(psuedo code)

```
// A helper method that handles
// the current token, and advances
// to get the next token.
process(str)
   if (currentToken == str)
      printXMLToken(str)
   else
      print("syntax error")
   currentToken =
      tokenizer.advance()
```

Software engineering notes:

- The syntax analyzer is version 0 of the compiler
- The compiler is highly recursive: The `compile`*xxx* methods call each other, for their effect
- Each `compile`*xxx* method is responsible for advancing the input and handling its own part of the input.

A bit of compilation theory

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

LL Grammars

- LL: Parsing the input from Left to right, performing Leftmost derivation of the input
- LL(*k*): Looking ahead *k* tokens is sufficient for knowing which parsing rule to invoke next
- LL(1) grammar: the first token informs which rule to invoke next
- LL(1) grammars can be handled simply and elegantly by recursive descent parsing algorithms, without backtracking

`prog.jack` (input)

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

Parse tree

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      ...
```

The Jack grammar is LL(1)

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

LL Grammars

- LL: Parsing the input from Left to right, performing Leftmost derivation of the input
- LL(*k*): Looking ahead *k* tokens is sufficient for knowing which parsing rule to invoke next
- LL(1) grammar: the first token informs which rule to invoke next
- LL(1) grammars can be handled simply and elegantly by recursive descent parsing algorithms, without backtracking

The complete Jack grammar is LL(1),
barring one exception that can be handled easily

(to be discussed soon)

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Grammar notation

Jack grammar (subset)

statement: *letStatement* | *ifStatement* | *whileStatement*

statements: *statement**

letStatement: '**let**' *varName* '**=**' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**' '**{**' *statements* '**}**'

expression: *term* (*op term*)?

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

op: '**+**' | '**-**' | '**=**' | '**>**' | '**<**'

'xxx'	:	quoted boldface is used to list language tokens that appear verbatim ("terminals");
xxx	:	Regular typeface represents names of non-terminals;
( )	:	used for grouping;
x | y	:	indicates that either x or y appear;
x y	:	indicates that x appears, and then y appears;
x?	:	indicates that x appears 0 or 1 times;
x*	:	indicates that x appears 0 or more times.

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

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'

Lexical elements (tokens)

Program structure

Statements

Expressions

Jack grammar: lexical elements

The Jack language features five categories 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.

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

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'

Lexical elements

Program structure

Statements

Expressions

Jack grammar: program structure

A Jack program is a collection of *classes*, each appearing in a separate file, and each compiled separately. Each class is structured as follows:

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

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

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'

Lexical elements

Program structure

Statements

Expressions

Jack grammar: statements

A Jack program includes *statements*, as follows:

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? '**;**'

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

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'

Lexical elements

Program structure

Statements

Expressions

Jack grammar: expressions

A Jack program can includes *expressions,* as follows:

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'

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Jack analyzer: usage

```
prompt> JackAnalyzer input
```

input: *fileName*`.jack`: name of a single source file, or

folderName: name of a folder containing one or more `.jack` source files

output: If the input is a single file: *fileName*`.xml`

If the input is a folder: one `.xml` file for every `.jack` file, stored in the given folder

Jack analyzer: usage

Example: `Point.jack`

```
/** Represents a Point. */
class Point {
   ...
   /** Returns the x value */
   method int getx() {
      return x;
   }
   ...
}
```

JackAnalyzer

`Point.vm`

```
<class>
   <keyword> class </keyword>
   <identifier> Point </identifier>
   <symbol> { </symbol>
   ...
   <subroutineDec>
      <keyword> method </keyword>
      <keyword> int </keyword>
      <identifier> getx </identifier>
      <symbol> ( </symbol>
      <parameterList>
      </parameterList>
      <symbol> ) </symbol>
      <subroutineBody>
         <symbol> { </symbol>
         <statements>
            <returnStatement>
               <keyword> return </keyword>
               <expression>
                  <term>
                     <identifier> x </identifier>
                  </term>
               </expression>
               <symbol> ; </symbol>
            </returnStatement>
         </statements>
         <symbol> } </symbol>
      </subroutineBody>
   </subroutineDec>
   ...
   <symbol> } </symbol>
<class>
```

The analyzer handles:

- tokens (terminals)
- rules (nonterminals)

Jack analyzer: handling tokens (terminals)

If the analyzer encounters a terminal element *xxx*

It generates the output:

<terminalType> xxx </terminalType>

where *terminalType* is:

`keyword`, `symbol`, `integerConstant`, `stringConstant`, `identifier`

Output examples:

```
<keyword> method </keyword>
<symbol> { </symbol>
<integerConstant> 42 </integerConstant>
<stringConstant> xkcd </stringConstant>
<symbol> { </symbol>
```

String constants are outputted without their enclosing double quotes

Jack analyzer: handling rules (nonterminals)

If the analyzer encounters one of the nonterminal elements:

class declaration, *class variable declaration*, *subroutine declaration*, *parameter list*, *subroutine body*, *variable declaration*, *statements*, `let` *statement,* `if` *statement,* `while` *statement,* `do` *statement,* `return` *statement, an expression, a term*, or an *expression list*

It generates the output:

```
<nonTerminal>
    Recursive output for the nonterminal body
</nonTerminal>
```

where *nonTerminal* is:

`class`, `classVarDec`, `subroutineDec`, `parameterList`, `subroutineBody`, `varDec`; `statements`, `LetStatement`, `ifStatement`, `whileStatement`, `doStatement`, `returnStatement`; `expression`, `term`, `expressionList`

Example:

```
// Jack grammar
...
returnStatement: 'return' expression? ';'
...
expression: term (op term)?
...
```

input example: `return x;`

Jack analyzer

Generated output:

```
<returnStatement>
   <keyword> return </keyword>
   <expression>
      <term>
         <identifier> x </identifier>
      </term>
   </expression>
   <symbol> ; </symbol>
</returnStatement>
```

The output of `compileExpression`, which was called by `compileReturn`

Jack analyzer: handling rules (nonterminals)

If the analyzer encounters one of the (simple) nonterminal elements:

type, *class name*, *subroutine name*, *var name*, *statement*, or *subroutine call*,

It handles the body of the nonterminal directly, without calling other methods
(because it simplifies the implementation).

Example:

```
// Jack grammar
...
letStatement:  'let'  varName  '='  expression ';'
varName:    identifier
identifier:    a sequence of letters, digits, ...,
               not starting with a digit
...
```

input example:

```
let x = 17;
```

Jack analyzer

Generated output:

```
<letStatement>
   <keyword> let </keyword>
   <identifier> x </identifier>
   <symbol> = </symbol>
   <expression>
      ...
```

There is no `varNamexxx` method.

The parsing of the *varName* is handled directly, by `compileLet`

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Jack Analyzer: Proposed implementation

Modules

- `JackTokenizer` : Handles the input
- `CompilationEngine`: Handles the parsing
- `JackAnalyzer`: Drives the translation process.

JackTokenizer

Function

Handles the compiler's input.

Provides services for:

- Ignoring white space
- Getting the current token and advancing the input just beyond it
- Getting the type of the current token

`JackTokenizer` API (first approximation)

```
class JackTokenizer {
   Constructor(inputFile)
   hasMoreTokens()
   advance()
   tokenType()
   ...
}
```

The implementation of the `tokenType` method is informed by the Jack lexicon:

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.

JackTokenizer API

JackTokeinizer: Ignores all comments and white space, gets the next token, and advances the input just beyond it

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	input file / stream	–	Opens the input `.jack` file / stream and gets ready to tokenize it.
`hasMoreTokens`	–	boolean	Are there more tokens in the input?
`advance`	–	–	Gets the next token from the input, and makes it the current token. This method should be called only 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, as a constant.
`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, as a constant. This method should be called only if `tokenType` is `KEYWORD`.
`symbol`	–	char	Returns the character which is the current token. Should be called only if `tokenType` is `SYMBOL`.
`identifier`	–	string	Returns the string which is the current token. Should be called only if `tokenType` is `IDENTIFIER`.
`intVal`	–	int	Returns the integer value of the current token. Should be called only if `tokenType` is `INT_CONST`.
`stringVal`	–	string	Returns the string value of the current token, without the opening and closing double quotes. Should be called only if `tokenType` is `STRING_CONST`.

Jack Analyzer: Proposed implementation

Modules

`JackTokenizer` : Handles the input

- `CompilationEngine`: Handles the parsing
- `JackAnalyzer`: Drives the translation process.

CompilationEngine

Grammar

```
...

statement:   ifStatement |
             whileStatement |
             letStatement | ...

statements:  statement*

ifStatement: 'if' '(' expression ')'
             '{' statements '}'

whileStatement: 'while' '(' expression ')'
             '{' statements '}'

...
```

`CompilationEngine` API (first approximation)

```
class CompilationEngine {

    // Constructor (code omitted)

    // Compiles statements
    compileStatements()

    // Compiles an if statement
    compileIf()

    // Compiles a while statement
    compileWhile()

    ...
}
```

- Gets its input from a `JackTokenizer`, and emits its output to an output file
- The output is generated by a series of `compile`*xxx* routines, structured according to the grammar rules that define *xxx*
- Each `compile`*xxx* routine is responsible for handling all the tokens that make up *xxx*, advancing the tokenizer exactly beyond these tokens, and outputing the parsing of *xxx*.

CompilationEngine API

CompilationEngine: Generates the compiler's output (API continues next slide):

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Input stream/file Output stream/file	–	Creates a new compilation engine with the given input and output. The next routine called (by the `JackAnalyzer` module) must be `compileClass`.
`compileClass`	–	–	Compiles a complete class.
`compileClassVarDec`	–	–	Compiles a static variable declaration, or a field declaration.
`compileSubroutine`	–	–	Compiles a complete method, function, or constructor.
`compileParameterList`	–	–	Compiles a (possibly empty) parameter list. Does not handle the enclosing parentheses tokens (and).
`compileSubroutineBody`	–	–	Compiles a subroutine's body.
`compileVarDec`	–	–	Compiles a `var` declaration.
`compileStatements`	–	–	Compiles a sequence of statements. Does not handle the enclosing curly bracket tokens { and }.

CompilationEngine API

CompilationEngine: Generates the compiler's output (API continues next slide):

Routine	*Arguments*	*Returns*	*Function*
`compileLet`	–	–	Compiles a `let` statement.
`compileIf`	–	–	Compiles an `if` statement, possibly with a trailing `else` clause.
`compileWhile`	–	–	Compiles a `while` statement.
`compileDo`	–	–	Compiles a `do` statement.
`compileReturn`	–	–	Compiles a `return` statement.

CompilationEngine API

CompilationEngine: Generates the compiler's output (API continues next slide):

Routine	*Arguments*	*Returns*	*Function*
`compileExpression`	–	–	Compiles an expression.
`compileTerm`	–	–	Compiles a *term*. If the current token is an *identifier*, the routine must resolve it into a *variable*, an *array entry*, or a *subroutine call*. A single lookahead token, which may be `[`, `(`, or `.`, suffices to distinguish between the possibilities. Any other token is not part of this term and should not be advanced over.
`compileExpressionList`	–	int	Compiles a (possibly empty) comma-separated list of expressions. Returns the number of expressions in the list.

Reminder:

The following Jack grammar rules are handled directly, and have no corresponding `compile`*xxx* methods:

type, className, *subroutineName*, *variableName*, *statement*, *subroutineCall*

Jack Analyzer: Proposed implementation

Modules

- ✓ `JackTokenizer` : Handles the input
- ✓ `CompilationEngine`: Handles the parsing
- `JackAnalyzer`: Drives the translation process.

JackAnalyzer

- The top-most / main module
- Input: a single fileName`.jack`, or a folder containing 0 or more such files
- For each file:

  1. Creates a `JackTokenizer` from *fileName*`.jack`
  2. Creates an output file named *fileName*`.xml`
  3. Creates a `CompilationEngine`, and calls the `compileClass` method.

Jack Analyzer: Proposed implementation

Modules

- ✓ `JackTokenizer` : Handles the input
- ✓ `CompilationEngine`: Handles the parsing
- ✓ `JackAnalyzer`: Drives the translation process.

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parse trees

Building a Jack analyzer

- The analyzer's design
- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed API
  - Project 10

Project 10: Building a Jack analyzer

Contract:

- Implement a syntax analyzer for the Jack language
- Test it by parsing the supplied test `.jack` class files
- For each test `.jack` file, your analyzer should generate an `.xml` output file, identical to the supplied compare file.

Tools and resources:

- Test programs and compare files: `nand2tetris/projects/10`
- TextComparer: `nand2tetris/tools`
- XML file viewer: Use a web browser or a text editor
- Programming language: Java, Python, ...
- Reference: chapter 10 in *The Elements of Computing Systems*.

Implementation plan

- Build a Jack Tokenizer
- Build a compilation engine
  (a Jack analyzer that uses the Tokenizer's services):
  - Basic version
  - Complete version.

Jack tokenizer

TestClass.jack

```
...
if (x < 0) {
   let sign = "negative";
}
...
```

TokenizerTest

TestClassT.xml

```
<tokens>
   <keyword> if </keyword>
   <symbol> ( </symbol>
   <identifier> x </identifier>
   <symbol> &lt; </symbol>
   <integerConstant> 0 </integerConstant>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <keyword> let </keyword>
   <identifier> sign </identifier>
   <symbol> = </symbol>
   <stringConstant> negative </stringConstant>
   <symbol> ; </symbol>
   <symbol> } </symbol>
</tokens>
```

string constants are outputted without the double-quotes

<, >, ", and & are outputted as <, >, ", and &

- Implement the `JackTokenizer` API, and write a `TokenizerTest` program
- Note: The `TokenizerTest` will not be used in the `JackAnalyzer`; Its only purpose is unit-testing the tokenizer.

Jack tokenizer

TestClass.jack

```
...
if (x < 0) {
   let sign = "negative";
}
...
```

TokenizerTest

TestClassT.xml

```
<tokens>
   <keyword> if </keyword>
   <symbol> ( </symbol>
   <identifier> x </identifier>
   <symbol> &lt; </symbol>
   <integerConstant> 0 </integerConstant>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <keyword> let </keyword>
   <identifier> sign </identifier>
   <symbol> = </symbol>
   <stringConstant> negative </stringConstant>
   <symbol> ; </symbol>
   <symbol> } </symbol>
</tokens>
```

Implementation tips:

Use your implementation language services for

- File processing
- String processing
- Tokenizing (if you want)

Jack analyzer

TestClass.jack

```
...
let x = x * (x + 1);
...
```

JackAnalyzer

TestClass.xml

```
...
<letStatement>
  <keyword> let </keyword>
  <identifier> x </identifier>
  <symbol> = </symbol>
  <expression>
    <term>
      <identifier> x </identifier>
    </term>
    <symbol> * </symbol>
    <term>
      <symbol> ( </symbol>
      <expression>
        <term>
          <identifier> x </identifier>
        </term>
        <symbol> * </symbol>
        <term>
          <integerConstant> 1 </integerConstant>
        </term>
      </expression>
      <symbol> ) </symbol>
    </term>
  </expression>
  <symbol> ; </symbol>
</letStatement>
...
```

Implementation:

- Build a basic `CompilationEngine` that handles everything except expressions
- Add the handling of expressions later

  (we supply test files for unit-testing both stages).

Sample test file

projects/10/Square/Square.jack

```
class Square {

   field int x, y;   // location on the screen
   field int size;   // The size of the square
   ...
   // Increments the square size by 2.
   method void incSize() {
      if (((y + size) < 254) & ((x + size) < 510)) {
         do erase();
         let size = size + 2;
         do draw();
      }
      return;
   }
   ...
   // Moves the square up by 2 pixels.
   method void moveUp() {
      if (y > 1) {
         do Screen.setColor(false);
         do Screen.drawRectangle(x, (y + size) - 1, x + size, y + size);
         let y = y - 2;
         ...
```

Sample test file (expressions highlighted)

projects/10/Square/Square.jack

```
class Square {

   field int x, y;   // location on the screen
   field int size;   // The size of the square
   ...
   // Increments the square size by 2.
   method void incSize() {
      if (((y + size) < 254) & ((x + size) < 510)) {
         do erase();
         let size = size + 2;
         do draw();
      }
      return;
   }
   ...
   // Moves the square up by 2 pixels.
   method void moveUp() {
      if (y > 1) {
         do Screen.setColor(false);
         do Screen.drawRectangle(x, (y + size) - 1, x + size, y + size);
         let y = y - 2;
         ...
```

Sample test file (expressionless version)

projects/10/Square/Square.jack

```
class Square {

   field int x, y;   // location on the
   field int size;   // The size of the
   ...
   // Increments the square size by 2.
   method void incSize() {
      if (((y + size) < 254) & ((x + si
         do erase();
         let size = size + 2;
         do draw();
      }
      return;
   }
   ...
   // Moves the square up by 2 pixels.
   method void moveUp() {
      if (y > 1) {
         do Screen.setColor(false);
         do Screen.drawRectangle(x, (y -
         let y = y - 2;
         ...
```

projects/10/ExpressionLessSquare/Square.jack

```
class Square {

   field int x, y;   // location on the screen
   field int size;   // The size of the square
   ...
   // Increments the square size by 2.
   method void incSize() {
      if (x) {
         do erase();
         let size = size;
         do draw();
      }
      return;
   }
   ...
   // Moves the square up by 2 pixels.
   method void moveUp() {
      if (y) {
         do Screen.setColor(false);
         do Screen.drawRectangle(x, y, x, y);
         let y = y;
         ...
```

The expressionless code is meaningless, but is grammatically valid

Purpose: Unit-test the analyzer's ability to parse everything except for expressions.

Implementation plan

- Build a Jack Tokenizer

- Build a compilation engine
  (a Jack analyzer that makes use of the Tokenizer's services):
  - Basic version (handles everything except expressions)
  - Complete version (handles everything, including expressions)

Handling 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'

Implementation note 1

If the current token is a *varName* (some identifier),
it can be the first token in any one of these possibilities:

- `foo`
- `foo[`*expression*`]`
- `foo.bar(`*expressionList*`)`
- `Foo.bar(`*expressionList*`)`
- `foo(`*expressionList*`)`

To resolve which possibility we're in, the syntax analyzer must *look ahead*:

- Save the current token, and
- Advance to get the next one

This is the only case in which the Jack grammar becomes LL(2) rather than LL(1).

Handling 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'

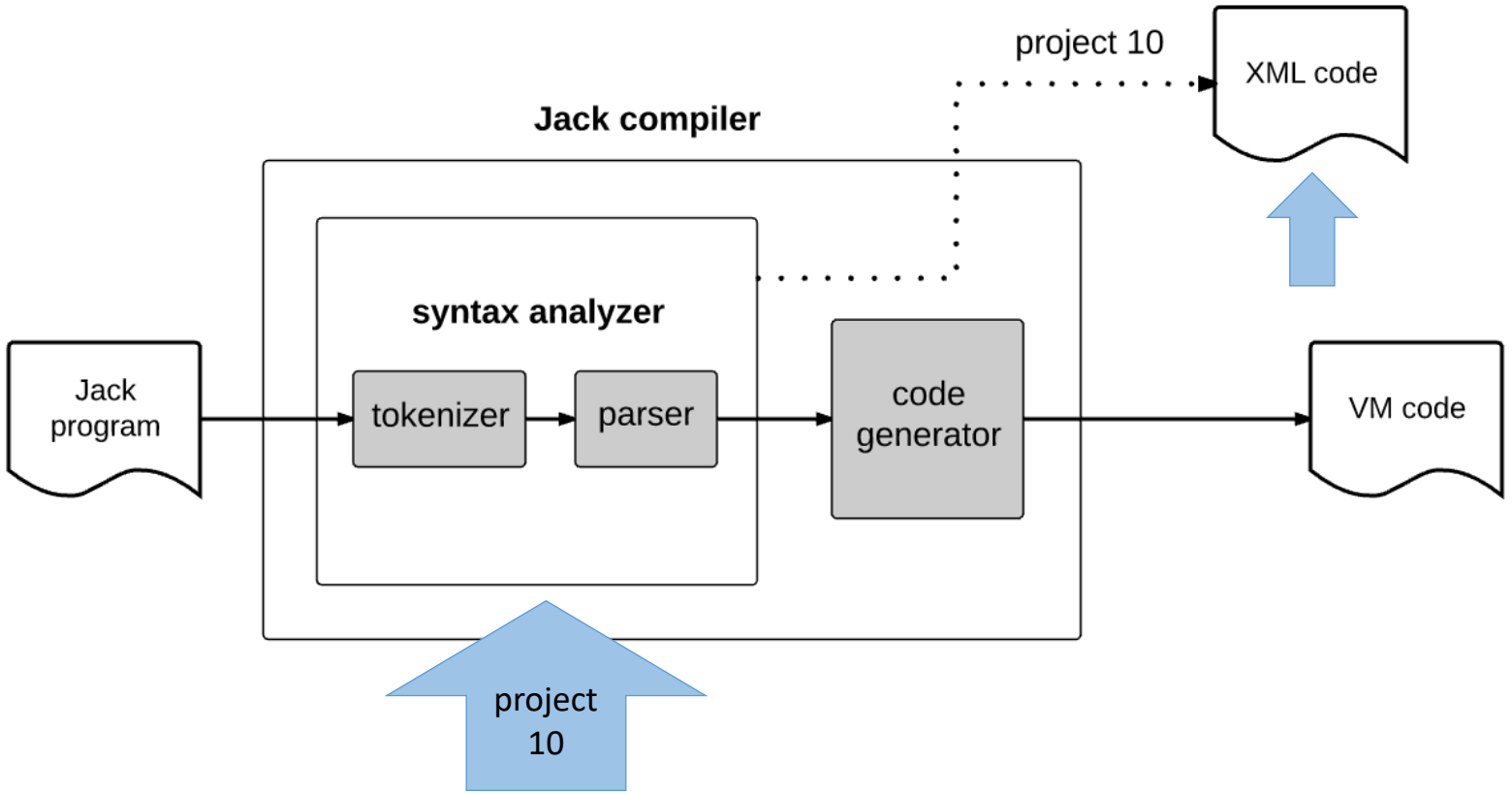
Implementation note 2

There is no `compileSubroutineCall` method

Rather, the subroutine call logic is handled in `compileTerm`

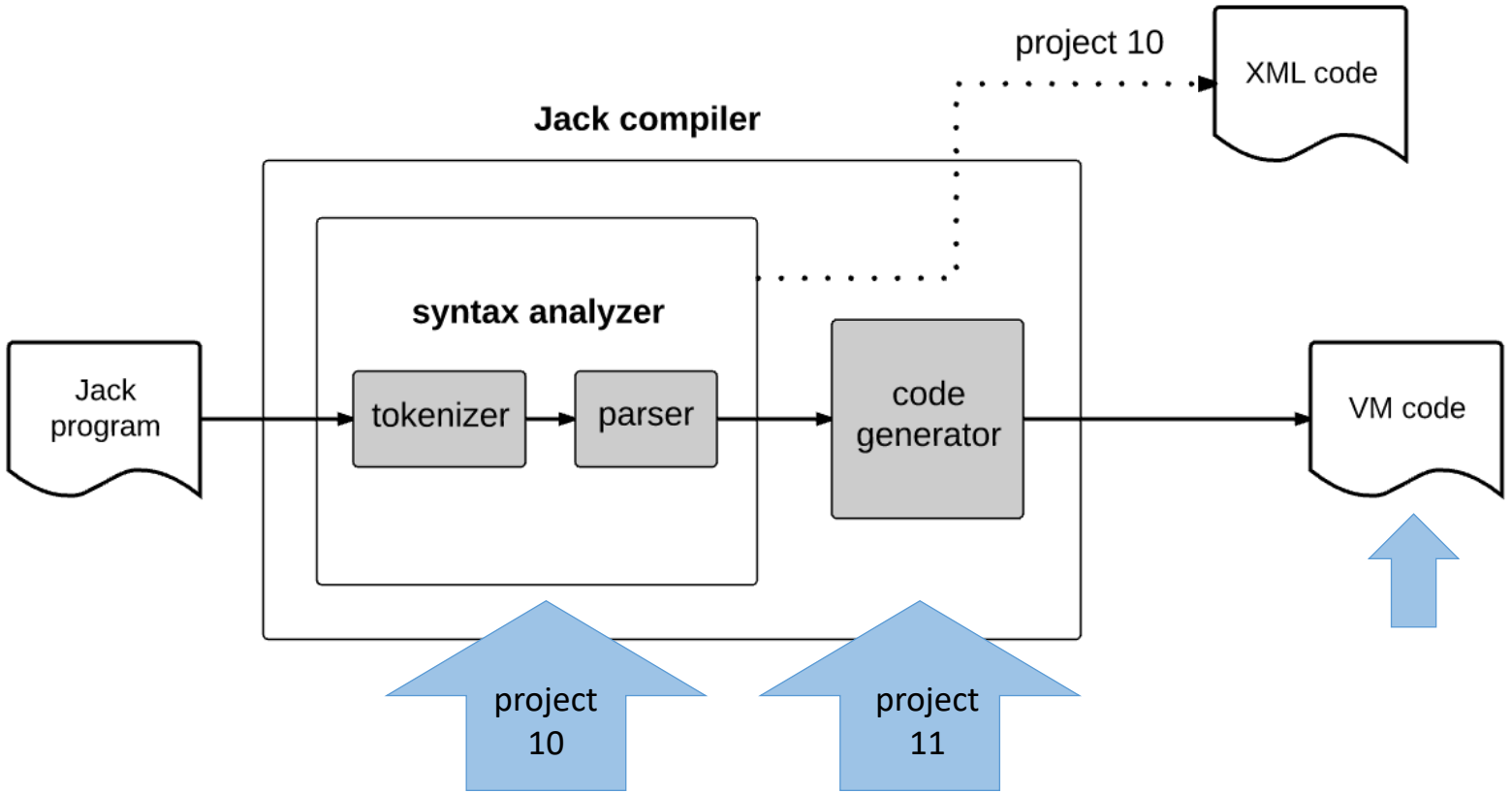
(Makes the implementation simpler).

Compiler development roadmap



Project 10: Generates XML code, for unit-testing the syntax analyzer

Compiler development roadmap



Project 10: Generates XML code, for unit-testing the syntax analyzer

Project 11: We'll replace the logic that generates passive XML code with logic that generates executable VM code.

Perspective: Error Handling

The Jack analyzer does not handle errors

Error handling

- Detection
- Handling
- Reporting.

Perspective: Tools

Parsing tools

- Since the design of a syntax analyzer is informed by the language grammar, it can written automatically
- LEX / YACC
- Many similar tools

Perspective: Applications

Parsing applications

- Compilation
- Computational linguistics
- Natural language processing
- Communications
- Bioinformatics
- Fintech
- Blockchain
- ...

Chapter 10

Compiler I: Parsing

These slides support chapter 10 of the book
The Elements of Computing Systems
By Noam Nisan and Shimon Schocken
MIT Press