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

The big picture: compilation

high-level program

```
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();
     // should print (4,6)
     do Output.println();
     do Output.printInt(p1.distance(p3));
     // should print 5
     return;
  }
}
```

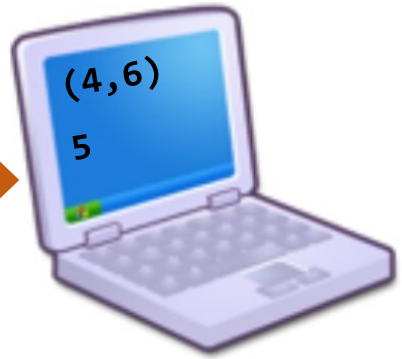
```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax,
                        int ay) {
      let x = ax;
      let y = ay;
      let pointCount =
          pointCount + 1;
      return this;
  }
 // ...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

Compiler

machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011
0000000000
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

execute



Two-tier compilation

high-level program

```
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();
     // should print (4,6)
     do Output.println();
     do Output.printInt(p1.distance(p3));
     // should print 5
     return;
  }
}
```

Compiler

```
/** 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
```

VM code

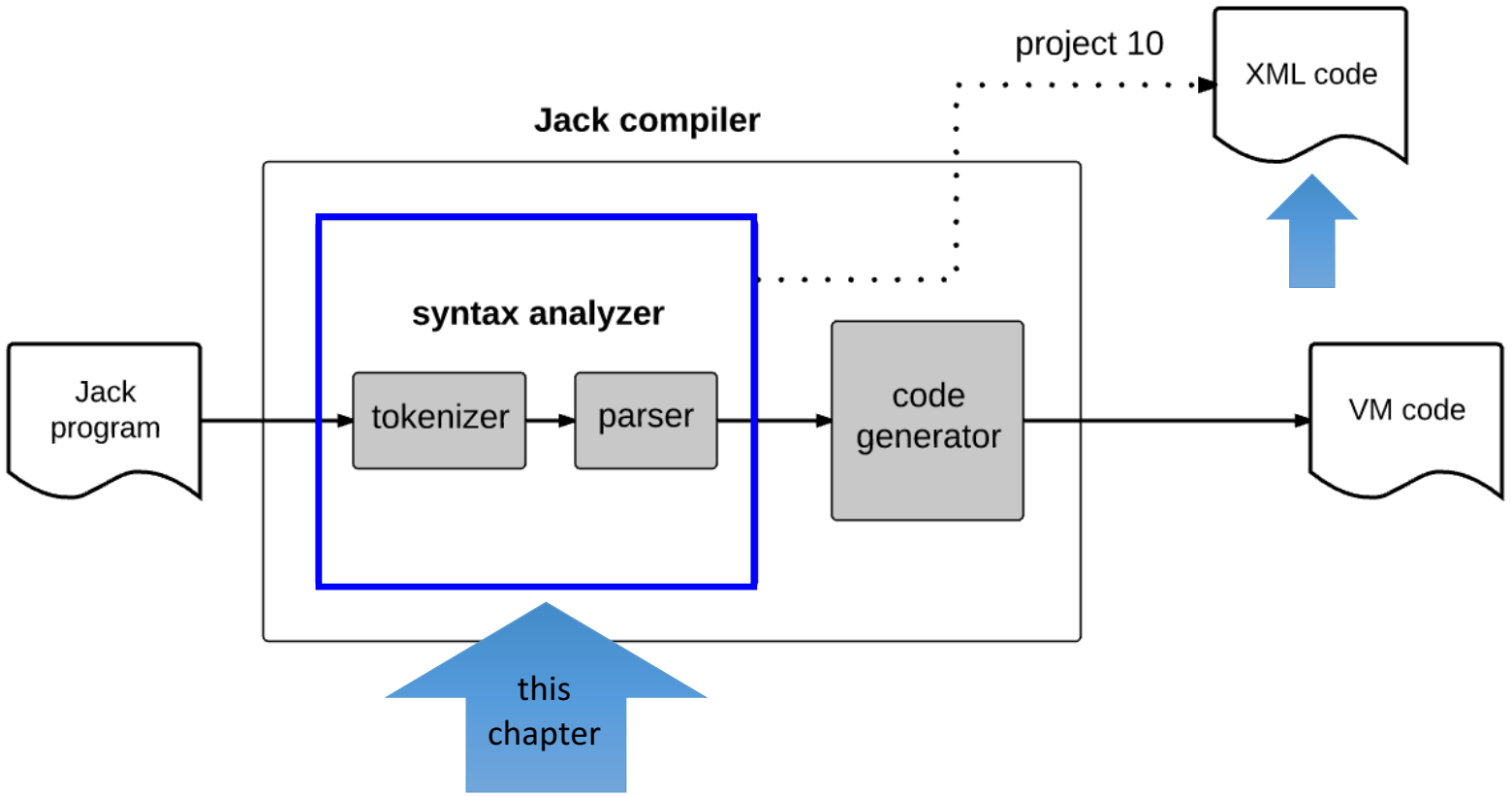
```
push local 0
push constant 0
eq
not
push argument 1
neg
push l
call d
pop argument 3
push local 1
neg
push local 0
push local 1
push local 2
add
push constant 2
push local 0
pop argument 1
push local 0
push constant 0
not
push local 1
push local 0
push local 1
add
...
```

VM translator

machine code

```
0000000000010000
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



Syntax analyzer implementation:

- Tokenizer
- Parser / compilation engine

Take home lessons

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

Compiler I / parsing: lecture plan

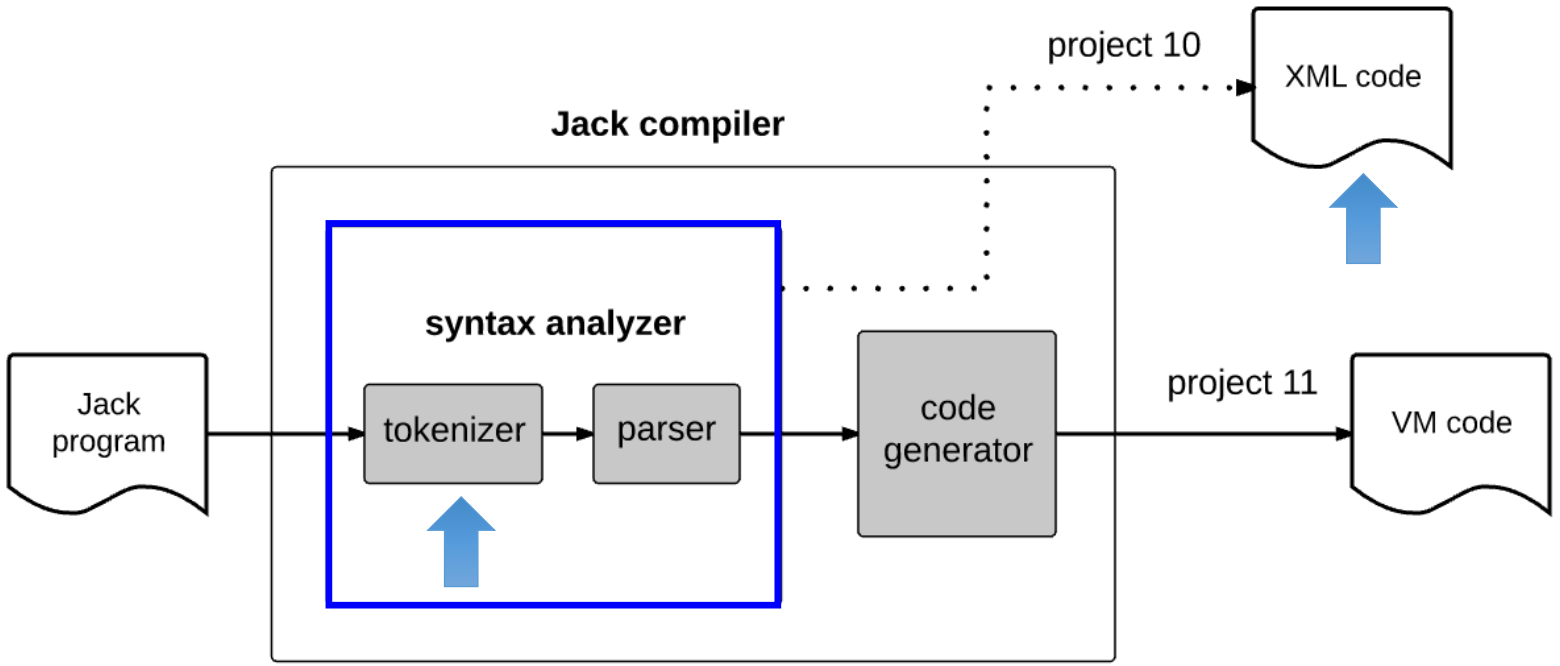
Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

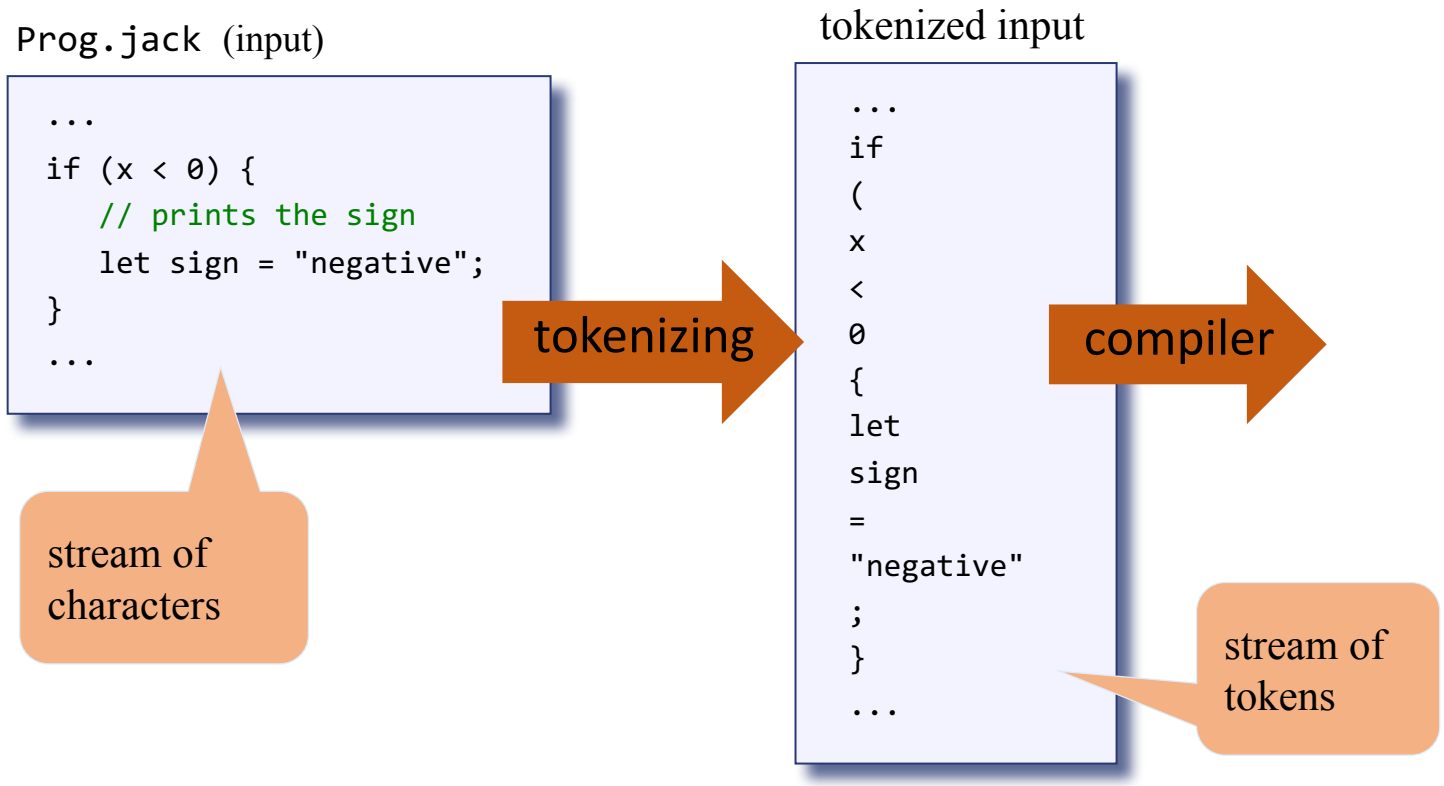
Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Compiler development roadmap



Tokenizing (first approximation)



Tokenizing = grouping characters into tokens

A *token* is a string of characters that has a meaning

A programming language specification must document (among other things) its allowable tokens.

Jack tokens

Prog.jack (input)

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

- keywords
- symbols
- integers
- strings
- identifiers

Jack tokens

`Prog.jack` (input)

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

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 tokenizer

Prog.jack

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

TokenizerTest

output

```
...
<keyword> if </keyword>
<symbol> ( </symbol>
<identifier> x </identifier>
<symbol> < </symbol>
<intConst> 0 </intCons>
<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:

- Handles the compiler's input
- Allows advancing the input
- Supplies the *current token's value* and *type*

  (complete API, later)

Jack tokenizer

Prog.jack

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

TokenizerTest

output

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

TokenizerTest (pseudo code)

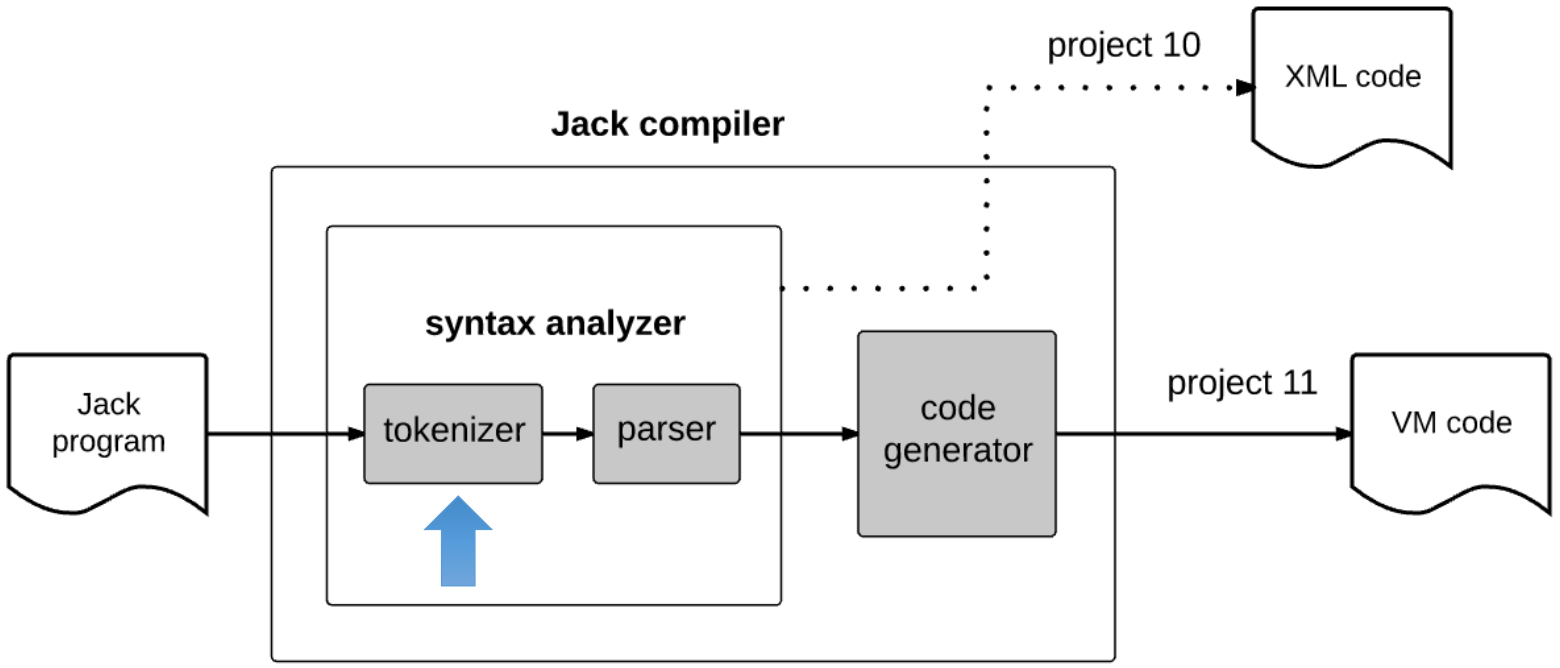
```
tknzr = new JackTokenizer("Prog.jack")
tknzr.advance();
while tknzr.hasMoreTokens() {
   tokenClassification = current token
                         classification
   print "<" + tokenClassification + ">"
   print the current token value
   print "</" + tokenClassification + ">"
   print newLine
   tknzr.advance();
}
```

Tokenizer:

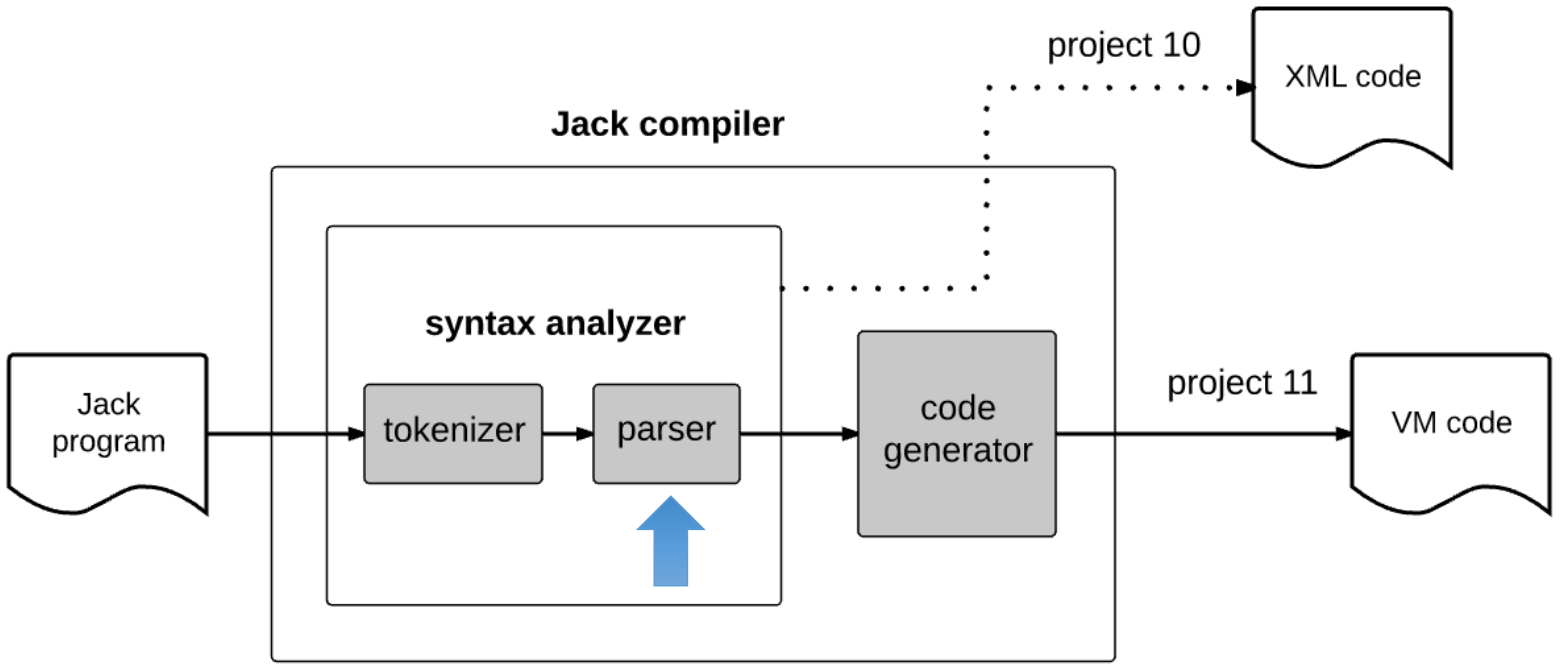
- Handles the compiler's input
- Allows advancing the input
- Supplies the *current token's value* and *type*

  (complete API, later)

Compiler development roadmap



Compiler development roadmap



Compiler I / parsing: lecture plan

Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Tokenizing (reminder)

source code

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

tokenized code

```
<keyword> class </keyword>
<identifier> Point </identifier>
<symbol> { </symbol>
...
<keyword> constructor </keyword>
<identifier> Point </identifier>
<identifier> new </identifier>
<symbol> ( </symbol>
<keyword> int </keyword>
<identifier> ax </identifier>
<symbol> , </symbol>
<keyword> int </keyword>
<identifier> ay </identifier>
<symbol> ) </symbol>
<symbol> { </symbol>
... // tokenizing the next two statements (omitted)
<keyword> let </keyword>
<identifier> pointCount </identifier>
<symbol> = </symbol>
<identifier> pointCount </identifier>
<symbol> + </symbol>
<integerConstant> 1 </integerConstant>
<symbol> ; </symbol>
...
```

Grammar

A *grammar* is a set of rules, describing how tokens can be combined to create valid language constructs

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 side* and a *right side*

Dina went to school

She said

The dog ate my homework

Terminal rule: right-hand side includes constants only

Non-terminal rule: all other rules

Grammar

Jack grammar (subset)

```
statement:  ifStatement |
            whileStatement |
            letStatement
```

Grammar

Jack grammar (subset)

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

Input examples

Expressions

```
17
x
x + 17
x - y
```

Grammar

Jack grammar (subset)

statement:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	'if' '(' expression ')' '{' statements '}'
whileStatement:	'while' '(' expression ')' '{' statements '}'
letStatement:	'let' varName '=' expression ';'
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:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	'**if**' '**(**' expression '**)**' '**{**' statements '**}**'
whileStatement:	'**while**' '**(**' expression '**)**' '**{**' statements '**}**'
letStatement:	'**let**' varName '**=**' expression '**;**'
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: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

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: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

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:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	'if' '(' expression ')' '{' statements '}'
whileStatement:	'while' '(' expression ')' '{' statements '}'
letStatement:	'let' varName '=' expression ';'
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 conforms to a grammar

In the process, uncovering the grammatical structure of the given input.

Compiler I / parsing: lecture plan

Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Parse tree

sentence: *nounPhrase verbPhrase*

nounPhrase: *determiner*? *noun*

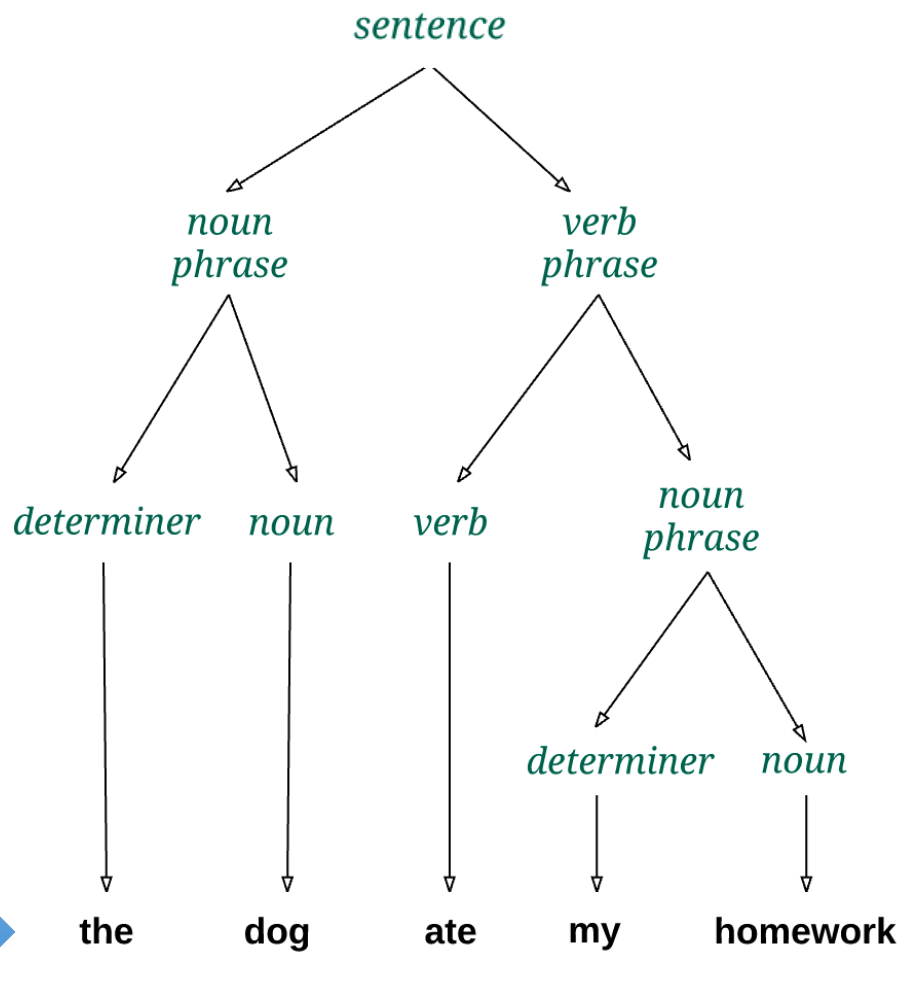
verbPhrase: *verb nounPhrase*

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

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

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

. . .



Parsing

- Determine if the given input conforms to the grammar
- In the process, construct the grammatical structure of the input

Parse tree

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

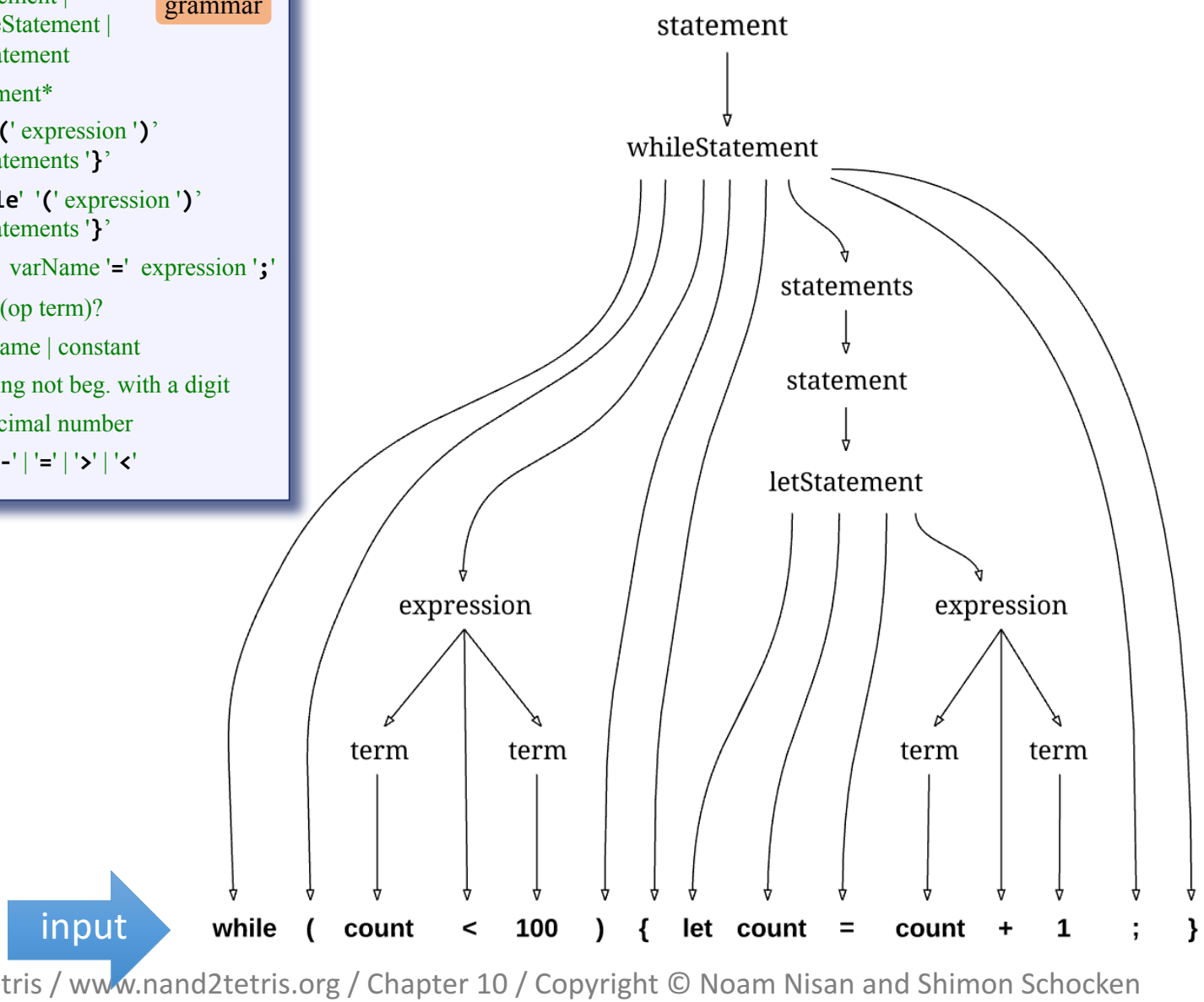
expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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



Parse tree

grammar

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

Same parse tree, in XML

parser output

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
         <symbol> = </symbol>
         <expression>
            <term> <identifier> count </identifier> </term>
            <symbol> + </symbol>
            <term> <intConstant> 1 </intConstant> </term>
         </expression>
         <symbol> ; </symbol>
      </letStatement>
   </statements>
   <symbol> } </symbol>
</whileStatement>
```

Compiler I / parsing: lecture plan

Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Parser design (sneak preview)

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

Parser

```
class CompilationEngine {
   compileStatements() {
      // code for compiling statements
   }
   compileIfStatement() {
      // code for compiling an if statement
   }

   compileWhileStatement() {
      // code for compiling an while statement
   }
   ...
   compileTerm() {
      // code for compiling a term
   }
}
```

The parser conists of a set of `compile`*xxx* methods;

each `compile`*xxx* method implements the right hand side of the grammar rule describing *xxx*

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

Parsing process

- Follow the right-hand side of the rule, and parse the input accordingly
- If the right-hand side specifies a non-terminal rule *xxx*, call `compileXxx`
- Do this recursively.

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

Parsing process

- Follow the right-hand side of the rule, and parse the input accordingly
- If the right-hand side specifies a non-terminal rule *xxx*, call `compilexxx`
- Do this recursively.

Example:

input

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

Parsing process

grammar

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

Parsing process

- Follow the right-hand side of the rule, and parse the input accordingly
- If the right-hand side specifies a non-terminal rule *xxx*, call `compilexxx`
- Do this recursively.

Example:

Let us assume that we read `while`,

so we know that we have to call the method `compileWhileStatement`

input

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
    <keyword> while </keyword>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
    <keyword> while </keyword>
```

input

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

Parsing process

grammar

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
```

compileExpression

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
```

compileExpression

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

compileWhileStatement

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
```

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
```

compileExpression

compileTerm

compileTerm

compileStatements

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
    <letStatement>
```

compileWhileStatement

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
         <symbol> = </symbol>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
         <symbol> = </symbol>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
    <letStatement>
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

compileExpression

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                          compileExpression
    <term>                              compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                              compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                          compileStatements
    <letStatement>                      compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>
      <expression>                      compileExpression
        <term> <identifier> count </identifier> </term>   compileTerm
        <symbol> + </symbol>
        <term> <intConstant> 1 </intConstant> </term>     compileTerm
      </expression>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
    <letStatement>
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>
      <expression>
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
    <letStatement>
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>
      <expression>
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term>
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
    <letStatement>
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>
      <expression>
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
```

compileExpression

compileTerm

compileTerm

compileStatements

compileLetStatement

compileExpression

compileTerm

compileTerm

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                          compileExpression
    <term>                              compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                              compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                          compileStatements
    <letStatement>                      compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>              compileExpression
      <expression>                      compileTerm
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>            compileTerm
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
    </letStatement>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                          compileExpression
    <term>                              compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                              compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                          compileStatements
    <letStatement>                      compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>              compileExpression
      <expression>                      compileTerm
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>            compileTerm
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
    </letStatement>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                          compileExpression
    <term>                              compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                              compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                          compileStatements
    <letStatement>                      compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>              compileExpression
      <expression>                      compileTerm
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>            compileTerm
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
    </letStatement>
  </statements>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                          compileExpression
    <term>                              compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                              compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                          compileStatements
    <letStatement>                      compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>              compileExpression
      <expression>                      compileTerm
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>            compileTerm
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
    </letStatement>
  </statements>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                          compileExpression
    <term>                              compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                              compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                          compileStatements
    <letStatement>                      compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>              compileExpression
      <expression>                      compileTerm
        <term> <identifier> count </identifier> </term>
        <symbol> + </symbol>            compileTerm
        <term> <intConstant> 1 </intConstant> </term>
      </expression>
      <symbol> ; </symbol>
    </letStatement>
  </statements>
  <symbol> } </symbol>
```

input

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

Parsing process

grammar

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

parser output

compileWhileStatement

```
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>                                  compileExpression
    <term>                                      compileTerm
      <identifier> count </identifier>
    </term>
    <symbol> < </symbol>
    <term>                                      compileTerm
      <intConstant> 100 </intConstant>
    </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>                                  compileStatements
    <letStatement>                              compileLetStatement
      <keyword> let </keyword>
      <identifier> count </identifier>
      <symbol> = </symbol>
      <expression>                              compileExpression
        <term> <identifier> count </identifier> </term>   compileTerm
        <symbol> + </symbol>
        <term> <intConstant> 1 </intConstant> </term>     compileTerm
      </expression>
      <symbol> ; </symbol>
    </letStatement>
  </statements>
  <symbol> } </symbol>
</whileStatement>
```

input

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

Parsing process

grammar

statement: ifStatement | whileStatement | letStatement

statements: statement*

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

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

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

expression: term (op term)?

term: varName | constant

varName: a string not beg. with a digit

constant: a decimal number

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

parser output

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
         <symbol> = </symbol>
         <expression>
            <term> <identifier> count </identifier> </term>
            <symbol> + </symbol>
            <term> <intConstant> 1 </intConstant> </term>
         </expression>
         <symbol> ; </symbol>
      </letStatement>
   </statements>
   <symbol> } </symbol>
</whileStatement>
```

input

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

Parser design

Grammar

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

Parser

```
class CompilationEngine {
    compileStatements() {
        // code for compiling statements
    }
    compileIfStatement() {
        // code for compiling an if statement
    }

    compileWhileStatement() {
        // code for compiling an while statement
    }
    ...
    compileTerm() {
        // code for compiling a term
    }
}
```

Parser's design:

- A set of `compile`*xxx* methods;
- each `compile`*xxx* method implements the right hand side of the grammar rule describing *xxx*

Parser design

Grammar

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

Parsing process

```
class CompilationEngine {
   ...
   compileWhileStatement() {
      eat('while'); code to handle 'while';
      eat('('); code to handle '(';
      compileExpression();
      eat(')'); code to handle ')';
      ...
```

```
eat(string) {
   if (currentToken <> string)
      error...
   else
      advance...
}
```

- The code of each `compile`*xxx* method follows the right-hand side of the rule *xxx*
- Each `compile`*xxx* method is responsible for advancing and handling its own part of the input.

LL grammar

Grammar

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

- LL grammar: can be parsed by a recursive descent parser without backtracking
- LL(*k*) parser: a parser that needs to look ahead at most *k* tokens in order to determine which rule is applicable
- The grammar that we saw so far is LL(1).

Compiler I / parsing: lecture plan

Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Grammar notation

statement:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	**'if'** **'('** expression **')'** **'{'** statements **'}'**
whileStatement:	**'while'** **'('** expression **')'** **'{'** statements **'}'**
letStatement:	**'let'** varName **'='** expression **';'**
expression:	term (op term)?
term:	varName | constant
varName:	a string not beg. 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'**

Jack grammar: lexical elements

The Jack language includes 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: 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: 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: expressions

A Jack program 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'

End note: parsing

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'**

Parser's design

- A set of `compile`*xxx* methods, structured according to the grammar rules defining *xxx*
- Each method outputs some of the parse tree (XML), and advances the input
- The parsing logic of each method follows the right-hand side of the rule that it implements.

End note: parsing expressions

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'

Parser's design

- A set of `compile`*xxx* methods, structured according to the grammar rules defining *xxx*
- Each method outputs some of the parse tree (XML), and advances the input
- The parsing logic of each method follows the right-hand side of the rule that it implements.

End note: parsing 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'

When 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)
bar(expressionList)
```

To resolve which possibility we are in, the parser should "look ahead":

- save the current token, and
- advance to get the next one.

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

Compiler I / parsing: lecture plan

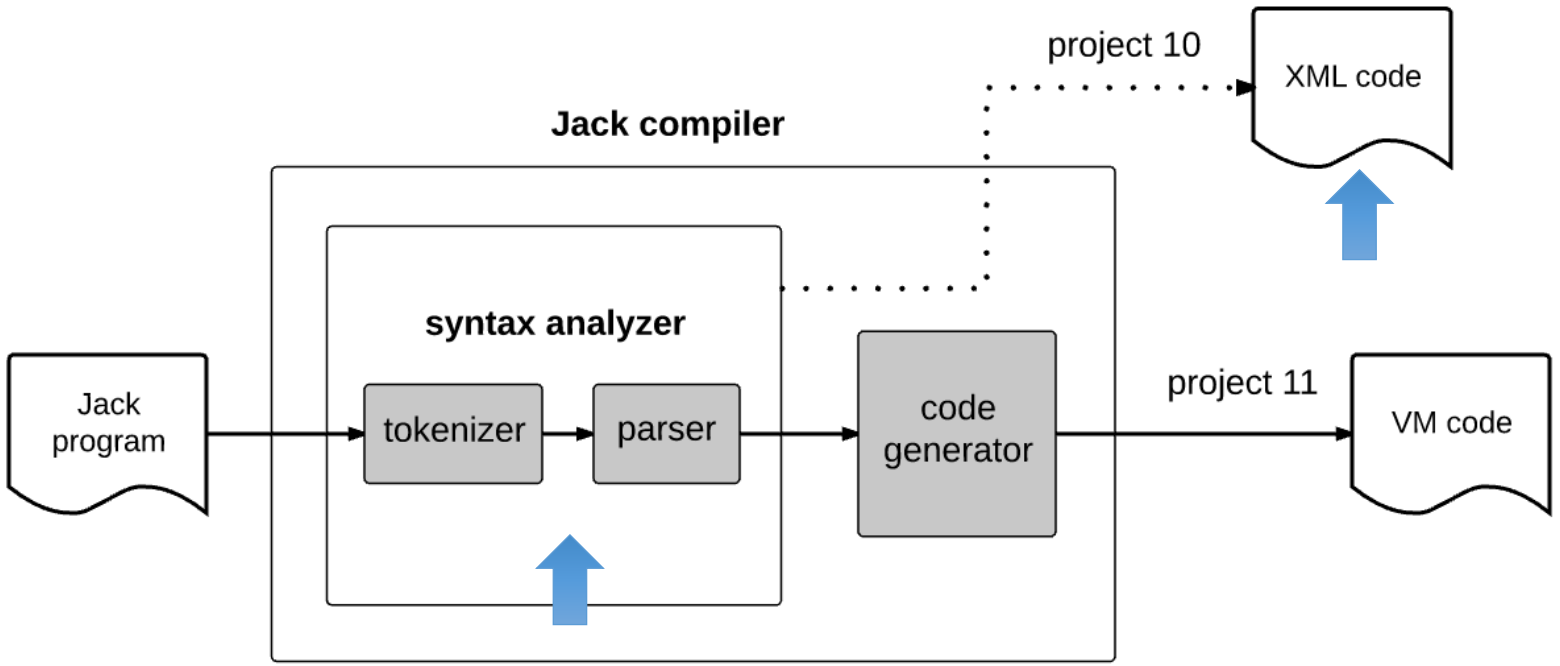
Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Compiler development roadmap



- How can we unit-test that the analyzer "understands" the source code?
- We can have it output the the source code in some structured way

Jack analyzer

input

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

parser

XML output

```
<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 parser generates marked-up output
- The mark up creates a textual parse tree
- Generated according to the Jack grammar

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'**

Jack analyzer: handling inputs that correspond to terminal rules

If the parser encounters a *terminalElement* *xxx* of type
keyword, *symbol*, *integer constant*, *string constant*, or *identifier*,

the parser generates the output:

```
<terminalElement>
    xxx
</terminalElement>
```

where *terminalElement* is:
`keyword`,
`symbol`,
`integerConstant`,
`stringConstant`,
`identifier`

Examples:

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

Jack analyzer: handling inputs that correspond to non-terminal rules

If the parser encounter a *nonTerminal* element of type *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*,

the parser generates the output:

```
<nonTerminal>
    Recursive output for the non-terminal body
</nonTerminal>
```

where *nonTerminal* is:
`class`, `classVarDec`, `subroutineDec`, `parameterList`, `subroutineBody`, `varDec`; `statements`, `LetStatement`, `ifStatement`, `whileStatement`, `doStatement`, `returnStatement`; `expression`, `term`, `expressionList`

Example: if the input is `return x;`

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

Jack analyzer: handling inputs that correspond to non-terminal rules

If the parser encounter a *nonTerminal* element of type *type*, *class name*, *subroutine name*, *variable name*, *statement*, or *subroutine call*,

The parser handles it directly, without recursing.

Example:

Jack grammar:

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

Suppose the input is `let x = 17;`

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

varName generates no mark-up

varName is handled directly

Compiler I / parsing: lecture plan

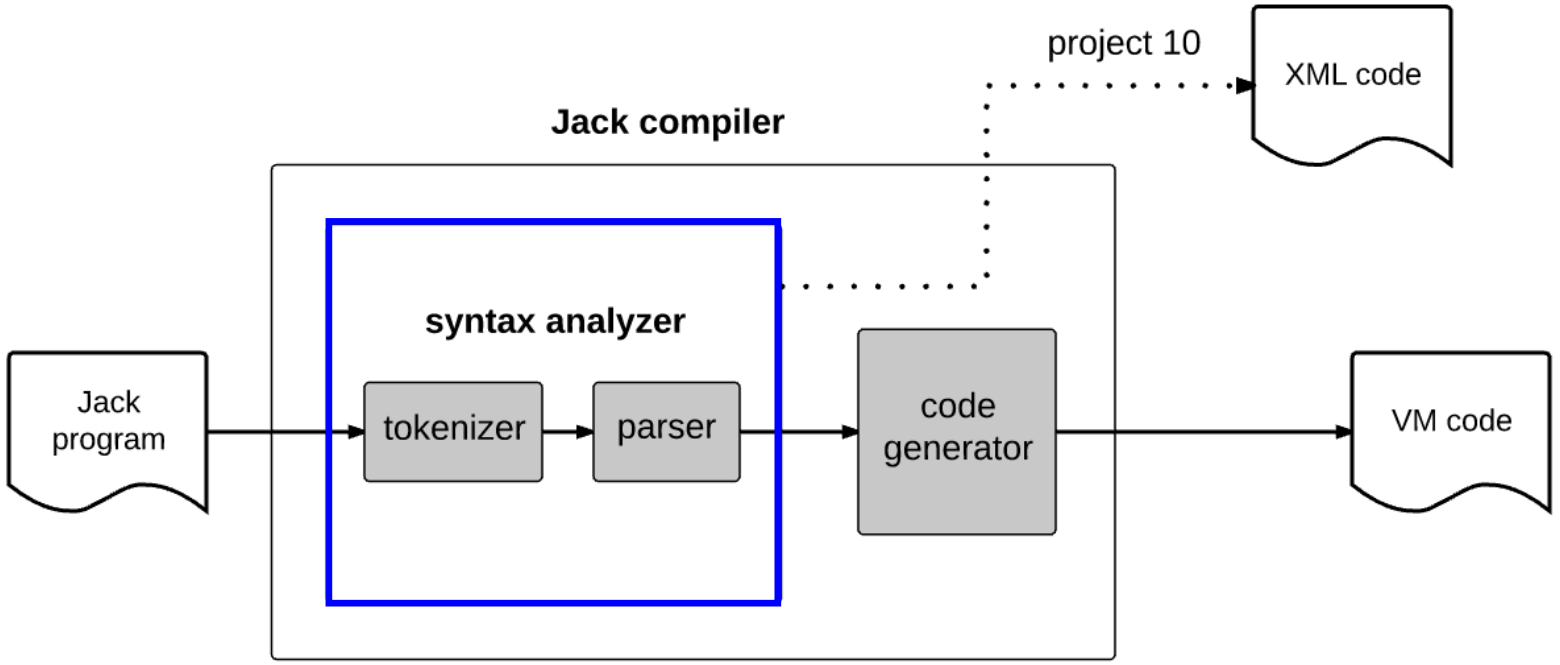
Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Compiler development roadmap



Implementation plan:

- `JackTokenizer`
- `CompilationEngine`
- `JackAnalyzer` (top-most / main module)

JackAnalyzer: usage

```
prompt> JackAnalyzer input
```

Input:
- *fileName*`.jack`: name of a single source file, or
- *directoryName*: name of a directory containing one or more `.jack` source files

Output:
- if the input is a single file: *fileName*`.xml`
- if the input is a directory: one `.xml` file for every `.jack` file, stored in the same directory

JackAnalyzer in action

Point.jack

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

JackAnalyzer

Point.xml

```
<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 JackAnalyzer

- Uses the services of a JackTokenizer
- Written according to the Jack grammar

Jack grammar (guideline for writing the JackAnalyzer)

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'

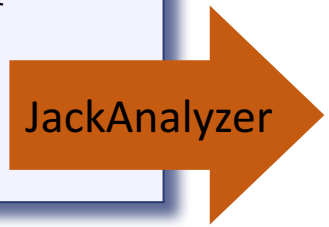
JackTokenizer

CompilationEngine

JackAnalyzer

input

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



XML output

```
<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>
```

JackTokenizer

input

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

JackAnalyzer

XML output

```
<keyword> class </keyword>
<identifier> Point </identifier>
<symbol> { </symbol>
...

   <keyword> method </keyword>
   <keyword> int </keyword>
   <identifier> getx </identifier>
   <symbol> ( </symbol>


   <symbol> ) </symbol>

      <symbol> { </symbol>


            <keyword> return </keyword>


                  <identifier> x </identifier>


            <symbol> ; </symbol>


      <symbol> } </symbol>


...
<symbol> } </symbol>
```

tokens only

JackTokenizer

input

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

JackAnalyzer

XML output

```
<keyword> class </keyword>
<identifier> Point </identifier>
<symbol> { </symbol>
...

   <keyword> method </keyword>
   <keyword> int </keyword>
   <identifier> getx </identifier>
   <symbol> ( </symbol>
   ...
```

tokens only

Jack 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.

JackTokenizer:

Handles the compiler's input.

Allows:

- Ignoring white space
- Advancing the input, one token at a time
- Getting the *value* and *type* of the *current token*

JackTokenizer

input

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

JackAnalyzer

XML output

```
<keyword> class </keyword>
<identifier> Point </identifier>
<symbol> { </symbol>
...

   <keyword> method </keyword>
   <keyword> int </keyword>
   <identifier> getx </identifier>
   <symbol> ( </symbol>
   ...
```

tokens only

Jack 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.

JackTokenizer

```
class JackTokenizer {

   // Constructor (code omitted)

   hasMoreTokens()

   advance()

   tokenType()

   ...

}
```

JackTokenizer API

JackTokenizer: Ignores all comments and white space in the input stream, and serializes it into Jack-language tokens. The token types are specified according to the Jack grammar.

Routine	Arguments	Returns	Function
Constructor	input file / stream		Opens the input `.jack` file 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.

JackTokenizer API

JackTokenizer: Ignores all comments and white space in the input stream, and serializes it into Jack-language tokens. The token types are specified according to the Jack grammar.

Routine	Arguments	Returns	Function
`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 identifier 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 two enclosing double quotes. Should be called only if `tokenType` is `STRING_CONST`.

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.

Tokenizer ✓

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'**

CompilationEngine

CompilationEngine design

Grammar

```
...

statement:   ifStatement |
             whileStatement |
             letStatement | ...

statements:  statement*

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

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

...
```

- Gets its input from a `JackTokenizer`, and emits its output to an output file

CompilationEngine

```
class CompilationEngine {
```

CompilationEngine design

Grammar

```
...

statement:   ifStatement |
             whileStatement |
             letStatement | ...

statements:  statement*

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

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

...
```

- Gets its input from a `JackTokenizer`, and emits its output to an output file

CompilationEngine

```
class CompilationEngine {
    // Constructor (code omitted)
```

CompilationEngine design

Grammar

```
...

statement:   ifStatement |
             whileStatement |
             letStatement | ...

statements:  statement*

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

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

...
```

CompilationEngine

```
class CompilationEngine {

    // Constructor (code omitted)

    compileStatements() {
        // code for compiling statements
    }

    compileIfStatement() {
        // code for compiling an if statement
    }

    compileWhileStatement() {
        // code for compiling an while statement
    }

    ...
}
```

- 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.

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 variable declaration, or a field declaration.
`CompileSubroutineDec`	—	—	Compiles a complete method, function, or constructor.
`compileParameterList`	—	—	Compiles a (possibly empty) parameter list. Does not handle the enclosing "`()`".
`compileSubroutineBody`	—	—	Compiles a subroutine's body.
`compileVarDec`	—	—	Compiles a `var` declaration.
`compileStatements`	—	—	Compiles a sequence of statements. Does not handle the enclosing "`{}`".

need 2 loops to handle two Class compilation

CompilationEngine API

`CompilationEngine`: generates the compiler's output.

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.

Routine	Arguments	Returns	Function
`CompileExpression`	--	--	Compiles an expression.
`CompileTerm`	--	--	Compiles a *term*. If the current token is an *identifier*, the routine must distinguish between a *variable*, an *array entry*, or a *subroutine call*. A single look-ahead token, which may be one of "[", "(", or ".", suffices to distinguish between the 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.

CompilationEngine (end-note)

The following rules in the Jack grammar have no corresponding `compile`*xxx* methods:

- *type*
- *className*
- *subroutineName*
- *variableName*
- *statement*
- *subroutineCall*

the parsing logic of these rules is handled by the rules that invoke them

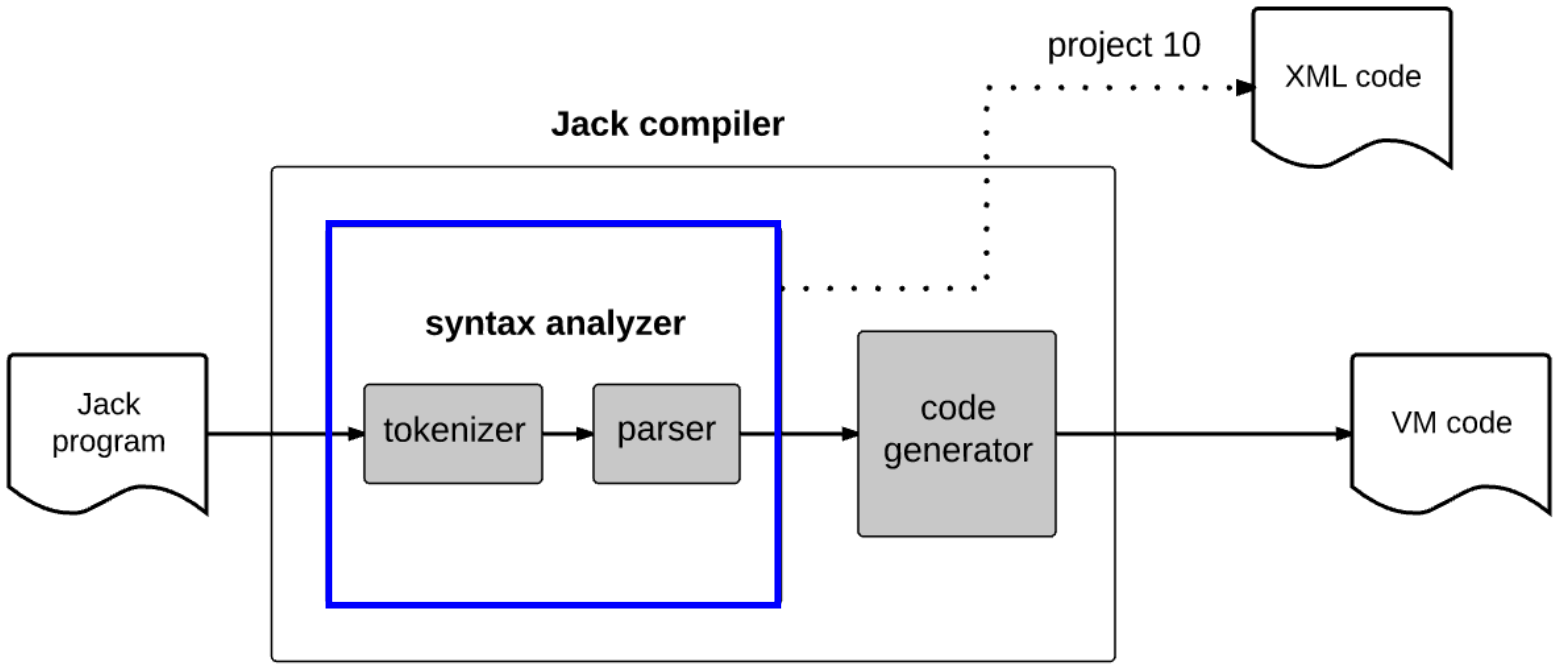
Example:

```
...
statement:    ifStatement |
              whileStatement |
              letStatement | ...
statements:   statement*
ifStatement:  'if' '(' expression ')'
              '{' statements '}'
whileStatement: 'while' '(' expression ')'
              '{' statements '}'
...
```

`compileStatements:`

- Uses a loop to handle 0 or more *statement* instances, according to the left-most token
- If the left-most token is `"if"`, `"while"`, ... it invokes `compileIf`, `compileWhile`, ...
- There is no `compileStatement` method

Recap



Implementation plan:

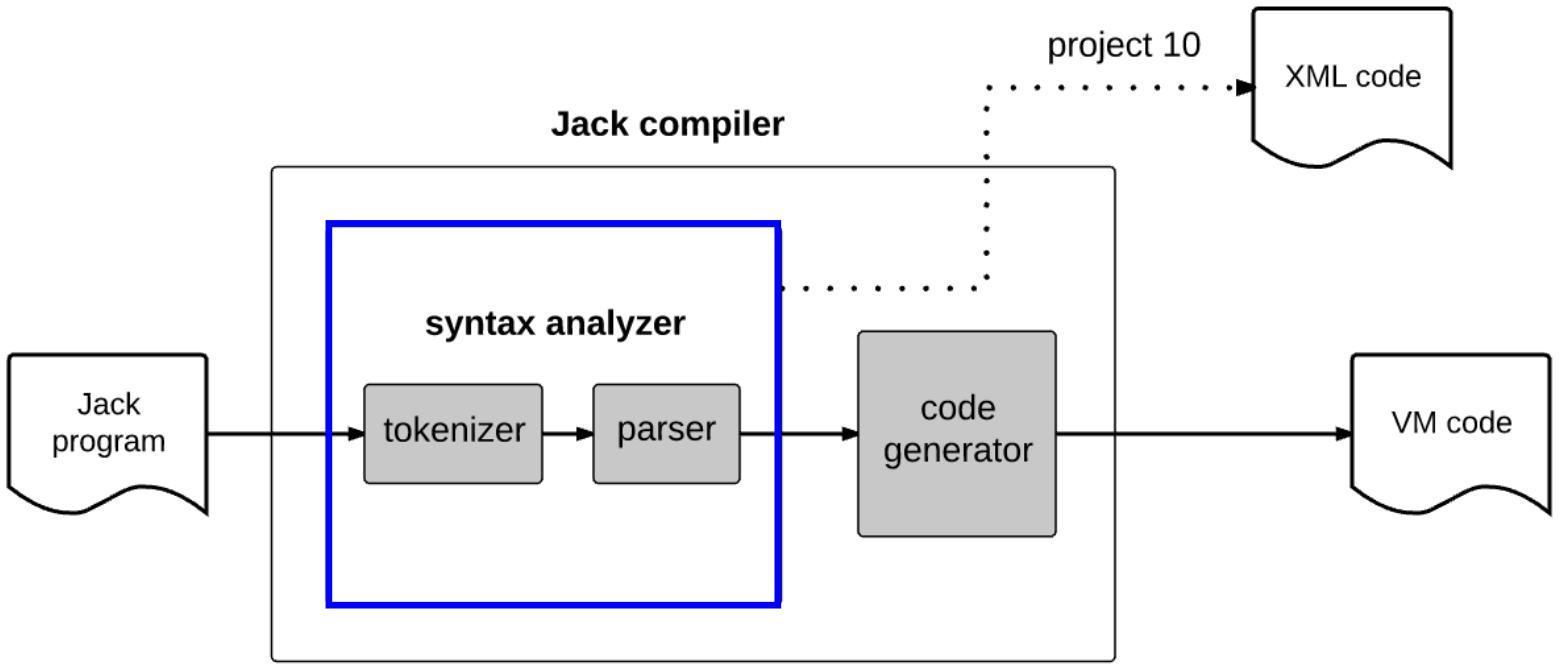
- ✓ `JackTokenizer`
- ✓ `CompilationEngine`
- `JackAnalyzer` (top-most module)

JackAnalyzer

- The top-most / main module
- Input: a single fileName`.jack`, or a directory containing 0 or more such files
- For each file, goes through the following logic:

1. Creates a `JackTokenizer` from *fileName*`.jack`
2. Creates an output file named *fileName*`.xml`
   and prepares it for writing
3. Creates and uses a `CompilationEngine` to compile the input `JackTokenizer`
   into the output file.

Recap



Implementation plan:

- ✓ `JackTokenizer`
- ✓ `CompilationEngine`
- ✓ `JackAnalyzer` (top-most module)

Compiler I / parsing: lecture plan

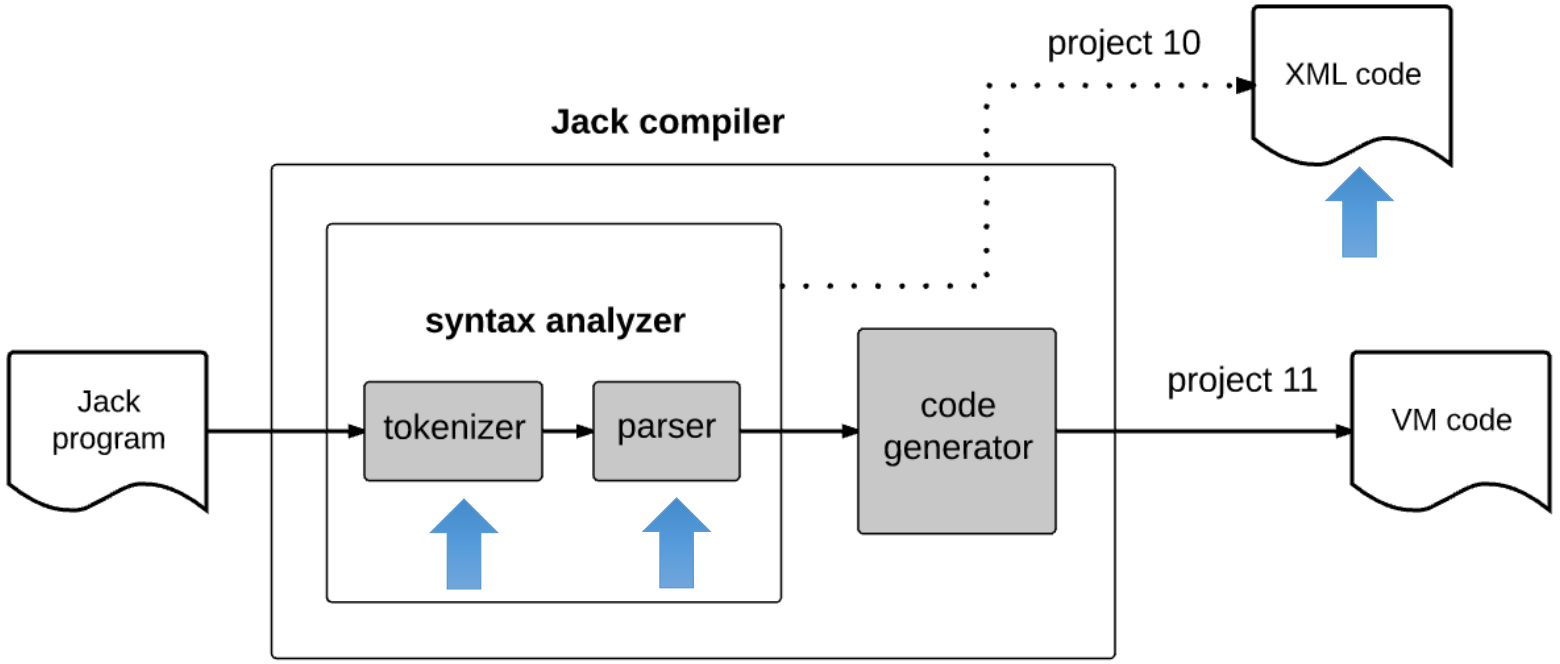
Parsing:

- Overview
- Tokenizer
- Grammar
- Parse trees
- Parsing process

Parsing Jack programs

- The Jack grammar
- The Jack analyzer
  - Overview
  - Proposed implementation
  - Building the Jack Analyzer

Goal: develop a syntax analyzer for the Jack language



Goal: develop a syntax analyzer for the Jack language

Contract:

- Implement a syntax analyzer for the Jack language
- Use it to parse all 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: browser, 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 makes use of the Tokenizer's services):
  - Basic version
  - Complete version

Jack tokenizer

TestClass.jack

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

tokenizer

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 &

Implementation tips:

Implement the Tokenizer API (described earlier)

You may use class library services (e.g. for handling strings etc.)

Jack analyzer

TestClass.jack

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

tokenizer

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

  (we supply test files to unit-test both stages)

Sample test class 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;
         do Screen.setColor(true);
         do Screen.drawRectangle(x, y, x + size, y + 1);
      }
      return;
   }
   // More Square methods
}
```

Sample test class 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;
         do Screen.setColor(true);
         do Screen.drawRectangle(x, y, x + size, y + 1);
      }
      return;
   }
   // More Square methods
}
```

Sample test class file (expressionless version)

projects/10/Square/Square.jack

```
class Square {

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

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(x);
         do Screen.drawRectangle(x, y, x, y);
         let y = y;
         do Screen.setColor(x);
         do Screen.drawRectangle(x, y, x, y);
      }
      return;
   }
   // More Square methods
}
```

Purpose:
tests the analyzer's ability to parse everything except expressions.

Basic Jack analyzer

Main.jack (expressionless)

```
...
let x = x;
...
```

tokenizer

Main.xml

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

The expressionless code is meaningless, but is grammatically valid

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)

Reminder: 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'

- When the current token is a *varName* (some identifier), it can be either a variable name, an array entry, or a subroutine call
- To resolve which possibility we are in, the parser should "look ahead": save the current token, and advance to get the next one.

Reminder: 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'

- When the current token is a *varName* (some identifier), it can be either a variable name, an array entry, or a subroutine call
- To resolve which possibility we are in, the parser should "look ahead": save the current token, and advance to get the next one
- There is no `compileSubroutineCall` method; rather, the subroutine call logic is handled in `compileTerm`

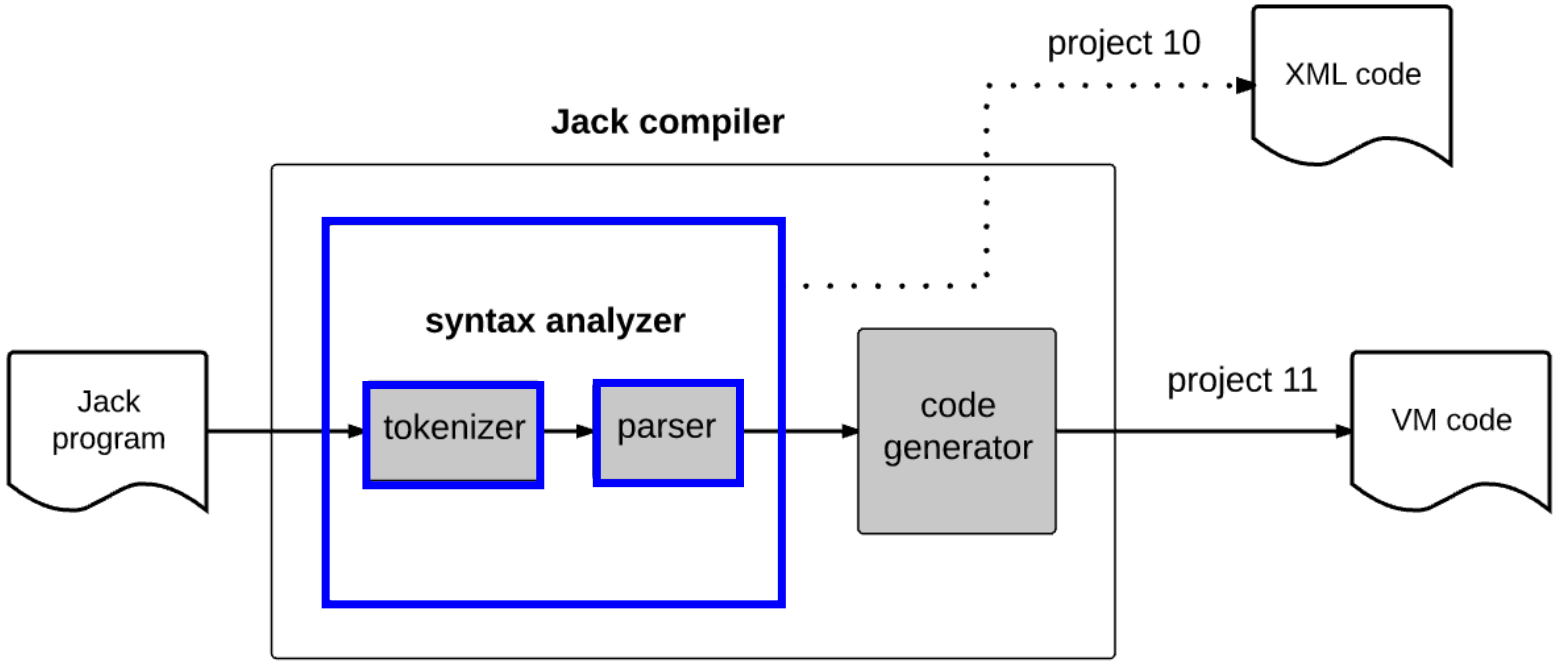
The ArrayTest class

Main.jack

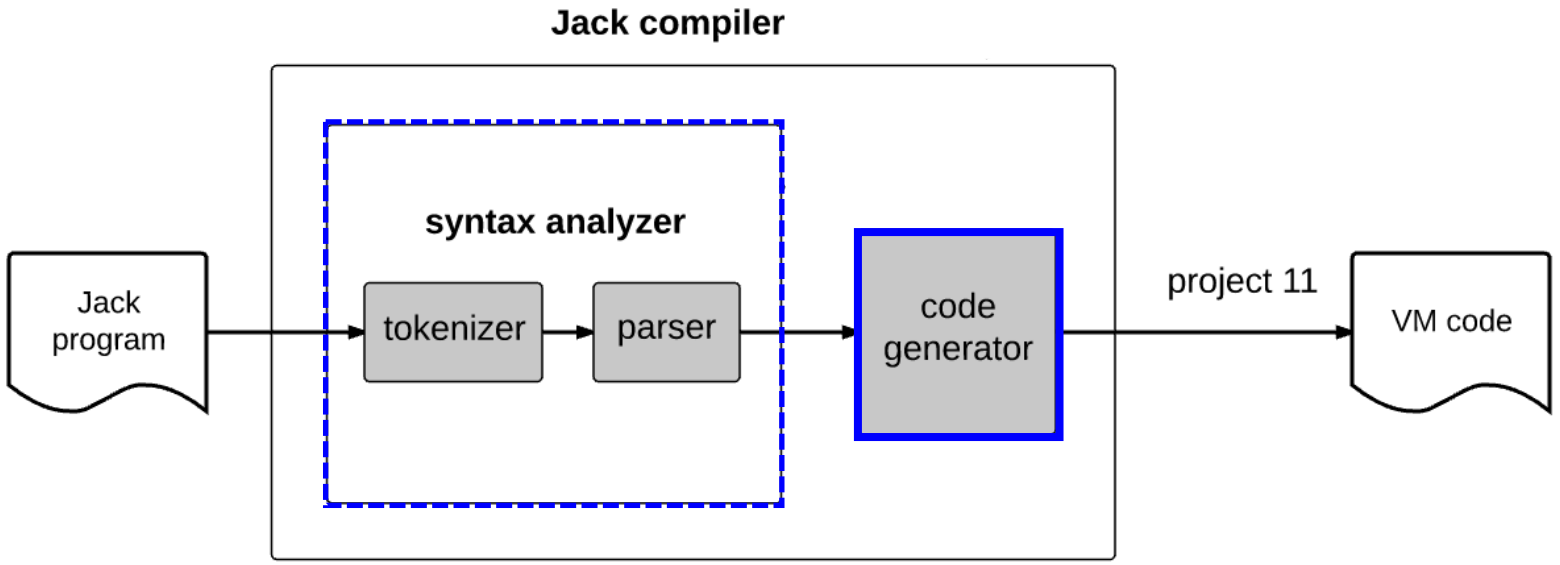
```
/** Computes the average of a sequence of integers. */
class Main {
   function void main() {
      var Array a;
      var int length;
      var int i, sum;

      let length = Keyboard.readInt("HOW MANY NUMBERS? ");
      let a = Array.new(length);
      let i = 0;

      while (i < length) {
         let a[i] = Keyboard.readInt("ENTER THE NEXT NUMBER: ");
         let i = i + 1;
      }

      let i = 0;  let sum = 0;

      while (i < length) {
         let sum = sum + a[i];
         let i = i + 1;
      }

      do Output.printString("THE AVERAGE IS: ");
      do Output.printInt(sum / length);
      do Output.println();
      return;
   }
}
```

Purpose:
Tests the analyzer's ability to handle array processing code.

Compiler's development roadmap (project 10)



Compiler's development roadmap (project 11)



Perspective

Errors

- Detection
- Handling
- Reporting

Parsing applications

- Compilation
- Linguistics
- Natural language processing
- Bioinformatics
- Financial services
- ...

Parsing strategies

- Top-down
- Bottom-up

Parsing tools

- LEX / YACC
- Why didn't we use them?