

Lecture 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, 2021

Compilation: One tier

high-level program

```
/** Creates and uses 2D points. */
class Main {
    function void main() {
        var Point p1, p2, p3;
        let p1 = Point.new(1,2);
        let p2 = Point.new(3,4);
        // Prints p1+p2
        let p3 = p1.plus(p2);
        do p3.print();
        do Output.println();
        // Prints distance(p1,p3)
        do Output.printInt(p1.distance(p3));
        return;
    }
}

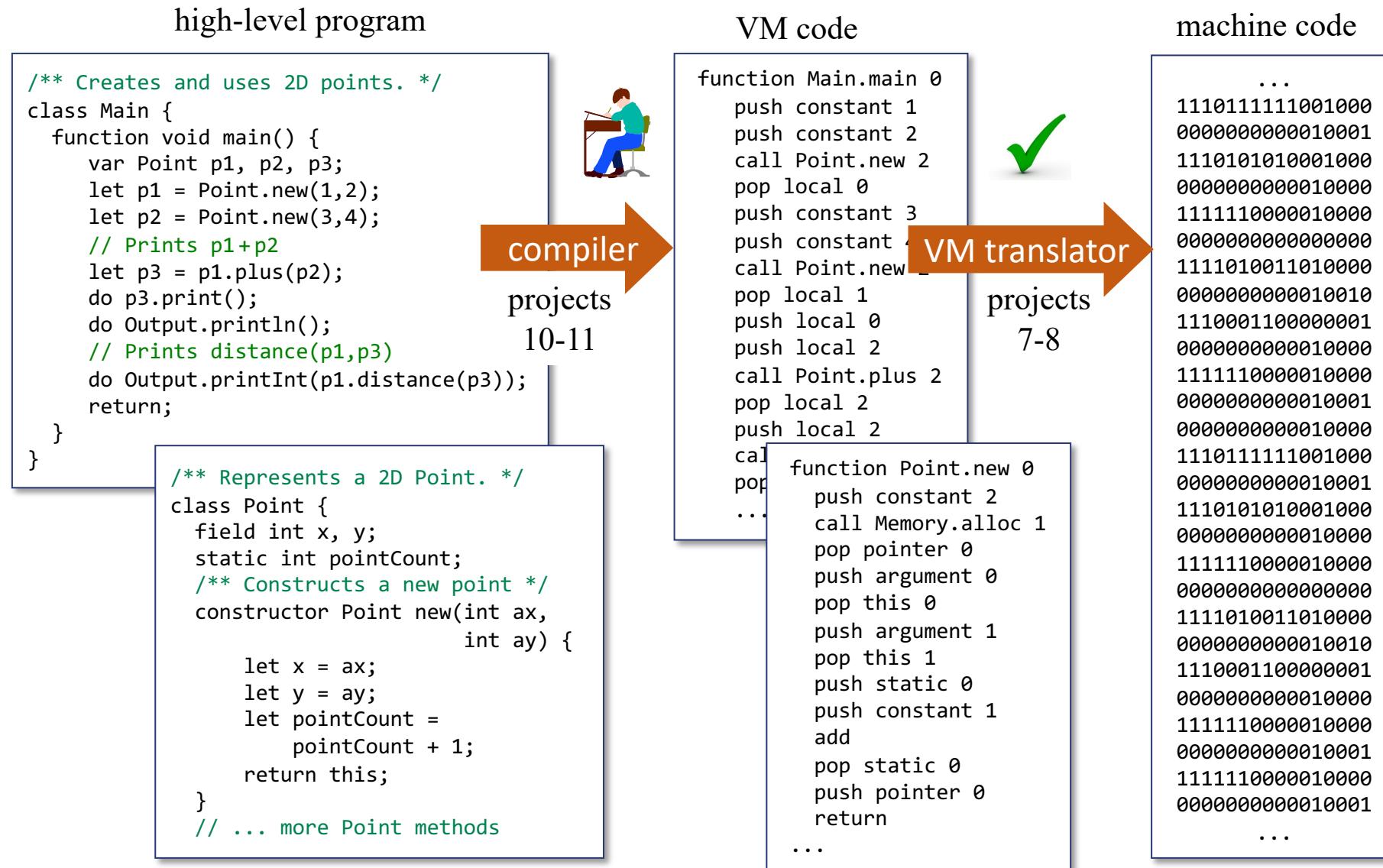
/** Represents a 2D Point. */
class Point {
    field int x, y;
    static int pointCount;
    /** Constructs a new point */
    constructor Point new(int ax
                           int ay
                           let x = ax;
                           let y = ay;
                           let pointCount =
                               pointCount + 1;
                           return this;
    }
    // ... more Point methods
}
```

compiler

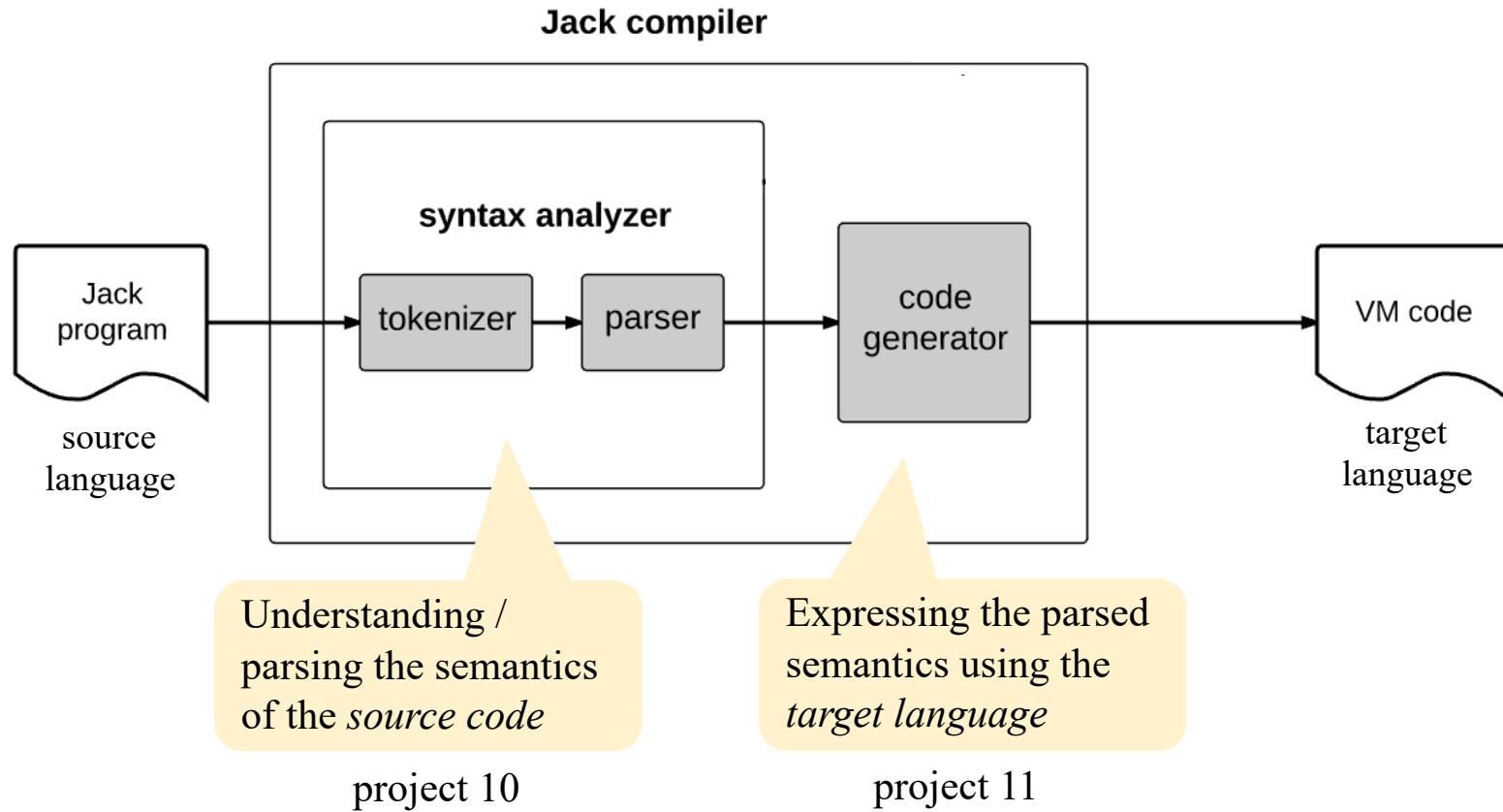
machine code

```
...  
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  
1111110000010000  
0000000000010001
```

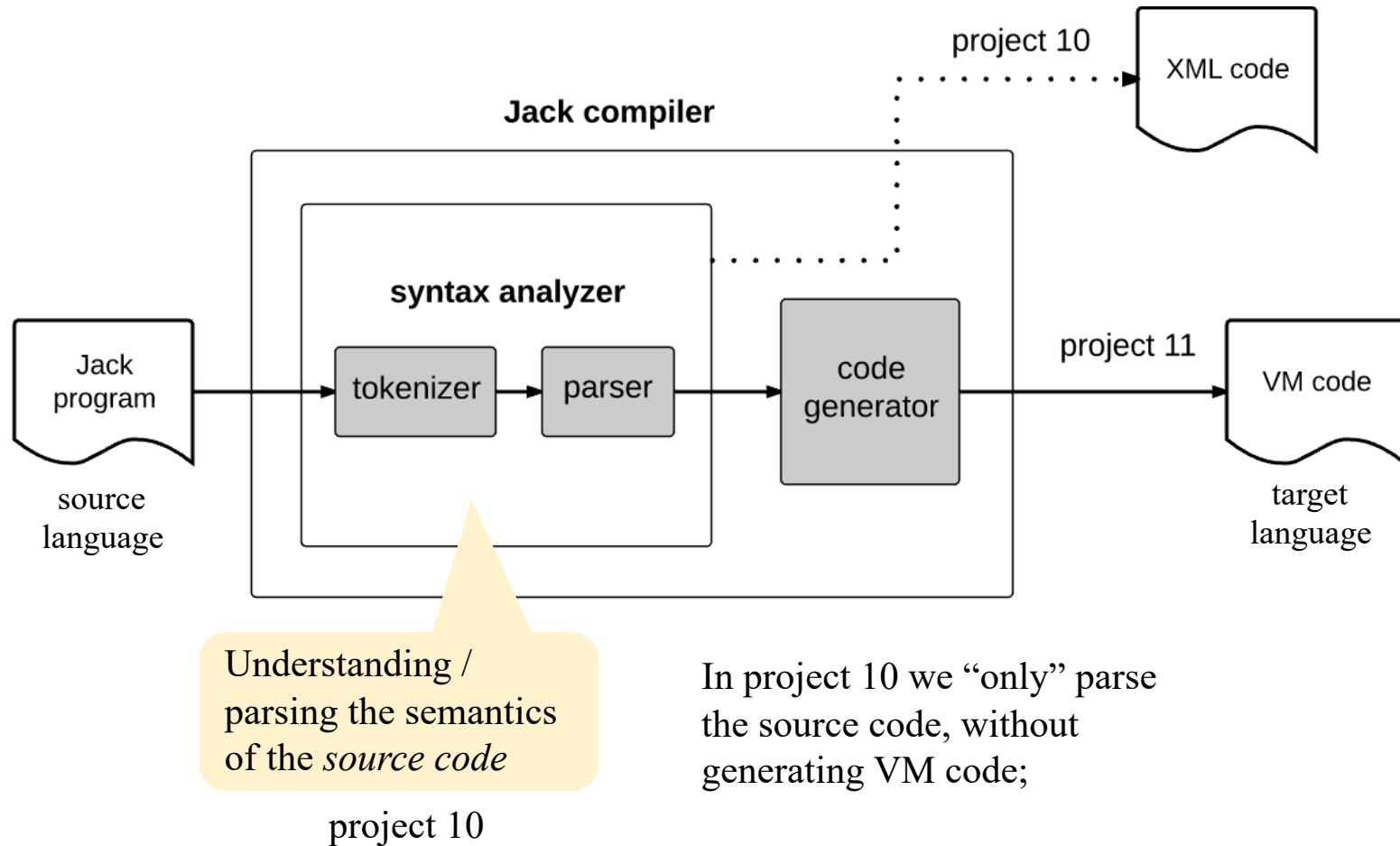
Compilation: Two tier



Compiler development roadmap



Compiler development roadmap



To unit-test that the syntax analyzer parses correctly, we’ll have it generate XML code that represents the source code semantics.

Compiler I / Syntax Analysis

Take home lessons

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

Applications

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

Lecture plan

Syntax analysis

- Overview
- Tokenizing
- Grammar
- Parsing

Building a syntax analyzer

- Software design
- The Jack grammar
- The Jack analyzer

Overview

Implementation

Project 10

Tokenizing

Prog.jack (input)

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

tokenizing

Stream of characters,
with white space

tokenized input

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

processing

Stream of tokens

The tokens
are the “atoms”
on which the
parser operates

Tokenizing: facilitates access to an input stream, one token at a time

Lexicon: set of valid tokens.

Tokenizing

Prog.jack (input)

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

Jack lexicon:

- keywords
- symbols
- integers
- strings
- identifiers

Tokenizing

Prog.jack (input)

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

Jack lexicon:

keyword: 'class' | 'constructor' | 'function'
| 'method' | 'field' | 'static' | 'var' | 'int'
| 'char' | 'boolean' | 'void' | 'true' | 'false'
| 'null' | 'this' | 'let' | 'do' | 'if' | 'else'
| 'while' | 'return'

symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' |
'|' | '&' | '<' | '>' | '=' | '~'

integerConstant: a decimal number in the range 0 ... 32767

StringConstant: "" a sequence of characters ""
(not including double quote or newline)

identifier: a sequence of letters, digits, and
underscore ('_') not starting with a digit.

Tokenizing

Prog.jack (input)

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

JackAnalyzer (v.0)

output

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

Jack lexicon:

keyword: 'class' | 'constructor' | 'function'
'method' | 'field' | 'static' | 'var' | 'int'
'char' | 'boolean' | 'void' | 'true' | 'false'
'null' | 'this' | 'let' | 'do' | 'if' | 'else'
'while' | 'return'

symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' |
'|' | '&' | '[' | ']' | '=' | '^' | '~'

integerConstant: a decimal number in the range 0 ... 32767

StringConstant: "" a sequence of characters ""
(not including double quote or newline)

identifier: a sequence of letters, digits, and
underscore ('_') not starting with a digit.

JackAnalyzer (version 0):

Using the services of a JackTokenizer,
prints the list of tokens in a given input
file, and their types.

Tokenizing

Prog.jack (input)

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

JackAnalyzer (v.0)

output

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

```
JackTokenizer { // API (details: later)  
  
    // Constructs a tokenizer  
    JackTokenizer(filename)  
  
    // Checks if there are more lines to process  
    boolean hasMoreTokens()  
  
    // Gets the current token and advances the input  
    void advance()  
  
    // Returns the type of the current token  
    String tokenType()  
    ...  
}
```

```
// Version 0: Designed to test the JackTokenizer  
JackAnalyzer {  
    tknzs = new JackTokenizer("Prog.jack")  
    tknzs.advance(); // gets the first token  
    while tknzs.hasMoreTokens() {  
        tokenType = type of the current token  
        print "<" + tokenType + ">"  
        print the current token  
        print "</" + tokenType + ">"  
        print newLine  
        tknzs.advance();  
    }  
} (pseudo code)
```

Lecture plan

Syntax analysis

- Overview
- Tokenizing
- Grammar
- Parsing

Building a syntax analyzer

- Software design
- The Jack grammar
- The Jack analyzer

Overview

Implementation

Project 10

Grammar

Tiny English grammar

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

valid sentences (examples)

```
the girl went to school  
she said  
the dog ate my homework
```

Grammar

A set of rules of the form *ruleName*: *ruleDefinition*

Each rule specifies how valid tokens can be arranged to form valid language patterns.

Grammar

Tiny English grammar

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

Grammar conventions (rules' right-hand side)

'x' : text that appears verbatim
 x : lexical element
 $x y$: x appears, then y appears
 $x | y$: either x or y appear
 $x?$: x appears 0 or 1 times
 x^* : x appears 0 or more times
 $(x y)$: grouping of x and y

Grammar

A set of rules of the form $\textit{ruleName} : \textit{ruleDefinition}$

Each rule specifies how valid tokens can be arranged to form valid language patterns.

Grammar

Jack grammar (subset)

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

Grammar conventions (rules' right-hand side)

'x' : text that appears verbatim
x : lexical element
x y : x appears, then y appears
x | y : either x or y appear
x? : x appears 0 or 1 times
x* : x appears 0 or more times
(x y) : grouping of x and y

Grammar

Jack grammar (subset)

statement: *letStatement* |
ifStatement |
whileStatement

statements: *statement**

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

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

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

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

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

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

input examples

let x = 100;



Grammar

Jack grammar (subset)

statement: *letStatement* |
ifStatement |
whileStatement

statements: *statement**

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

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

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

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

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

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

input examples

let x = 100;



let n = n + 1;



Grammar

Jack grammar (subset)

statement: *letStatement* |
ifStatement |
whileStatement

statements: *statement**

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

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

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

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

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

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

input examples

let x = 100;



let n = n + 1;



if (n < lim)
let n = n + 1;
}



Grammar

Jack grammar (subset)

statement: *letStatement* |
ifStatement |
whileStatement

statements: *statement**

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

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

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

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

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a decimal number

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

input example

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



Grammar

Jack grammar (subset)

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

Grammars consist of
two kinds of rules:

“nonterminals”

(rules whose definitions
mention other rules)

“terminals”

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar



Parsing

Building a syntax analyzer

- Software design
- The Jack grammar
- The Jack analyzer

Overview

Implementation

Project 10

Parsing

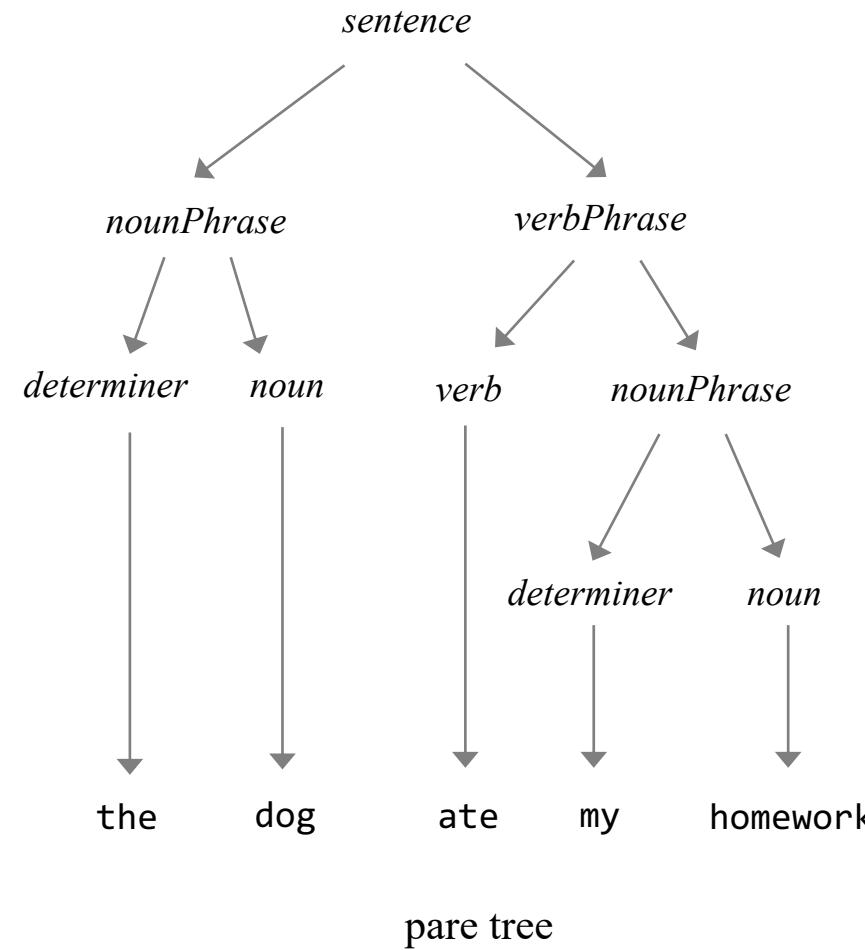
Tiny English grammar

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

Input

```
The dog ate my homework
```

parse



Parsing:

Checking if a given input is accepted by the grammar

In the process, uncovering the input's grammatical structure

Parsing

Jack grammar (subset)

```

statement: letStatement |  

           ifStatement |  

           whileStatement  

statements: statement *  

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

ifStatement: 'if' '(' expression ')' |  

           '{' statements '}' |  

whileStatement: 'while' '(' expression ')' |  

              '{' statements '}' |  

expression: term (op term)?  

term: varName | constant  

varName: a string not beginning with a digit  

constant: a decimal number  

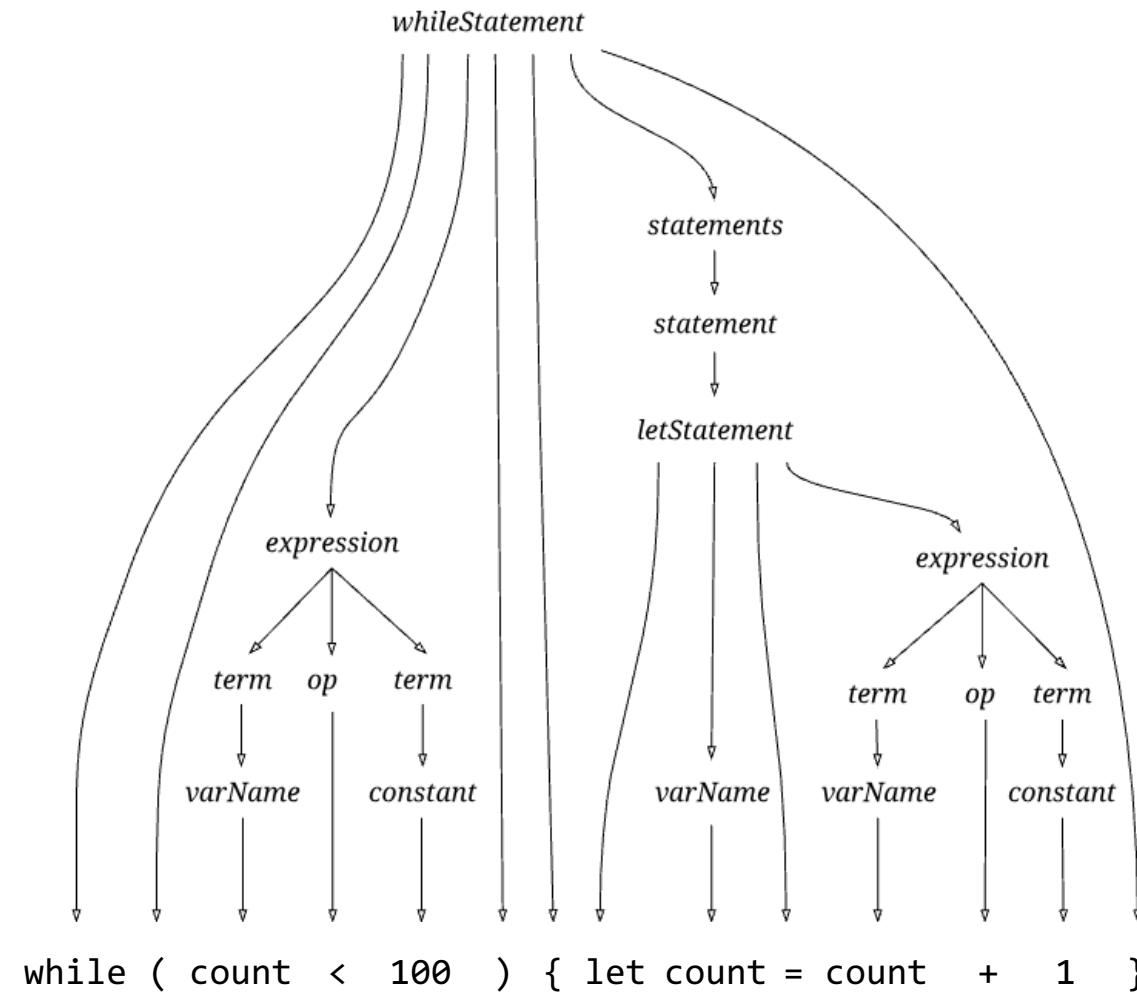
op: '+' | '-' | '=' | '>' | '<'  


```

Input

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

parse



Parsing

Jack grammar (subset)

```
statement: letStatement |
           ifStatement |
           whileStatement

statements: statement *

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

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

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

expression: term (op term)? 

term: varName | constant

varName: a string not beginning with a digit

constant: a decimal number

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

Input

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

parse

Same parse tree, expressed linearly using mark-up tags

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

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parsing

Building a syntax analyzer

- 
- Software design
 - The Jack grammar
 - The Jack analyzer
 - Overview
 - Implementation
 - Project 10

Syntax analyzer

Jack grammar (subset)

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

Syntax Analyzer (also known as *parser*)

Accepts / rejects a given input (program);

In the process, builds the program's parse tree.

prog.jack (input)

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

JackAnalyzer

Parse tree (output)

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

Syntax analyzer design

Jack grammar (subset)

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

Syntax analyzer (main module)

```
CompilationEngine {  
    ...  
    compileLet() {  
        // parses a let statement  
    }  
    ...  
    compileWhile() {  
        // parses a while statement  
    }  
    ...  
    compileTerm() {  
        // parses a term  
    }  
    ...  
}
```

The Parser's software design:
Informed by the language's grammar.

prog.jack (input)

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

JackAnalyzer

Parse tree (output)

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

Syntax analyzer design

Jack grammar (subset)

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

Syntax analyzer (main module)

```
CompilationEngine {  
    ...  
    compileLet() {  
        // parses a let statement  
    }  
    ...  
    compileWhile() {  
        // parses a while statement  
    }  
    ...  
    compileTerm() {  
        // parses a term  
    }  
    ...  
}
```

The Parser's software design:
Informed by the language's grammar.

Grammar rules are realized
by parsing methods

prog.jack (input)

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

JackAnalyzer

Parse tree (output)

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

Syntax analyzer design

Jack grammar (subset)

```
statement: letStatement |
           ifStatement |
           whileStatement

statements: statement*

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

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

whileStatement: 'while' '(' expression ')'
                '{' statements '}'  
  
expression: term (op term)?  
  
term: varName | constant  
  
varName: a string not beginning with a digit  
  
constant: a decimal number  
  
op: '+' | '-' | '=' | '>' | '<'
```

Parsing method (typical)

```
// Parses a while statement:  
// 'while' '(' expression ')' '{' statements '}'  
// Should be called if the current token is 'while'.  
compileWhile()
```

pseudo code

prog.jack (input)

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

JackAnalyzer

Parse tree (output)

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

Syntax analyzer design

Jack grammar (subset)

```
statement: letStatement |
           ifStatement |
           whileStatement

statements: statement*

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

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

whileStatement: 'while' '(' expression ')' '{' statements '}'  
  
expression: term (op term)?  
  
term: varName | constant  
  
varName: a string not beginning with a digit  
  
constant: a decimal number  
  
op: '+' | '-' | '=' | '>' | '<'
```

prog.jack (input)

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

Parsing method (typical)

```
// Parses a while statement:  
// 'while' '(' expression ')' '{' statements '}'  
// Should be called if the current token is 'while'.  
  
compileWhile()  
    print("<whileStatement>")  
    process("while")  
    process("(")  
    compileExpression()  
    process(")")  
    process("{")  
    compileStatements()  
    process("}")  
    print("</whileStatement>")  
  
// Helper method:  
// Handles the current input token,  
// and advances the input.  
  
process(str)  
    if (currentToken == str)  
        printXMLToken(str)  
    else  
        print("syntax error")  
    currentToken =  
        tokenizer.advance()
```

pseudo code

JackAnalyzer

Parse tree (output)

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

Syntax analyzer design

Jack grammar (subset)

```
statement: letStatement |
           ifStatement |
           whileStatement

statements: statement*

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

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

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

expression: term (op term)? 

term: varName | constant

varName: a string not beginning with a digit

constant: a decimal number

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

Parsing method (typical)

```
// Parses a while statement:
// 'while' '(' expression ')' '{' statements '}'
// Should be called if the current token is 'while'.

compileWhile()
    print("<whileStatement>")
    process("while")
    process("(")
    compileExpression()
    process(")")
    process("{")
    compileStatements()
    process("}")
    print("</whileStatement>")
```

pseudo
code

```
// Helper method:
// Handles the current input token,
// and advances the input.

process(str)
    if (currentToken == str)
        printXMLToken(str)
    else
        print("syntax error")
    currentToken =
        tokenizer.advance()
```

Recursive descent parsing

The parser is designed as a set of `compilexxx` methods (of which `compileWhile` is one);

The design of each `compilexxx` method is informed by a corresponding grammar rule;

Each `compilexxx` method is responsible for advancing, and handling, its own part of the input;

The `compilexxx` methods call each other, recursively.

How do we know which rule to invoke at each step?

Syntax analyzer design

Jack grammar (subset)

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

LL Grammars (a bit of theory)

LL: Parsing the input from Left to right, performing Leftmost derivation of the input

LL(k): Looking ahead k tokens is sufficient for knowing which parsing rule to invoke next

LL(1) grammar: The first token informs which rule to invoke next

LL(1) grammars can be handled simply and elegantly by recursive descent parsing algorithms, without backtracking.

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

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parsing

Building a syntax analyzer

- Software design
- 
- The Jack grammar
 - The Jack analyzer

Overview

Implementation

Project 10

Jack grammar: Tokens

The Jack language has 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.

Parsing tip 1

This part of the Jack grammar informs the implementation of the Jack Tokenizer;
(informs how to group characters into tokens).

Jack grammar: Program structure

A Jack class declaration is a set of variable and subroutine declarations:

```
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
```

Parsing tip 2

Some rules specify a sequence of elements whose length is not known;

For example, the *class* rule specifies a sequence of zero or more *classVarDec* elements;

To handle, the corresponding parsing methods can use iteration;

For example, `compileClass` can use a loop to parse as many *classVarDec* elements as there are in the input.

Jack grammar: Program structure

A Jack class declaration is a set of variable and subroutine declarations:

```
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
```

Parsing tip 3

Some rules specify two or more parsing possibilities;

For example, the first token of a *classVarDec* is either *static*, or *field*;

The process method shown before can be refined to accommodate such choices;

For example, we can change its parameter to a list/set of possible valid tokens.

Jack grammar: Statements

A Jack subroutine's body is a sequence of 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? ';'
```

Parsing tip 4

The *letStatement* rule accepts patterns like *let x = expression* as well as *let x[expression] = expression*

This implies that in the corresponding parsing method `compileLet`, after handling the *varName*, we should expect to see either the token `=`, or the token `[`, and continue the parsing accordingly;

To support such choices (in this and in other rules), see "Parsing Tip 3".

Jack grammar: Expressions

Jack statements can include 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'
```

Parsing tip 5

A *subroutine call* occurs in two places only:

when parsing a *term* in an *expression*. For example: `x + calc(...)` + y

when parsing a do statement. For example: `do play(...)`

To simply things, instead of writing a separate `compileSubroutineCall` method:

- Have your `compileTerm` method parse subroutine calls directly;
- Parse do *subroutineCall* statements as if they were do *expression*.

Jack grammar: Expressions

Jack statements can include 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'

```

Parsing tip 6

According to the definitions of *term* and *subroutineCall*, if the current token is a *varName*, say *foo*, it can be one of 5 possible things: *foo*, *foo[expression]*, *foo(expressionList)*, *Foo.bar(expressionList)*, or *foo.bar(expressionList)*.

Notice that the token following the *varName* (in this example, `foo`) is sufficient to resolve which option we are in. Therefore, if the current token is a *varName*, the syntax analyzer must:

1. save the current token, and
 2. advance to get the next token

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

Jack grammar: Complete

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'

tokens

program structure

statements

expressions

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parsing

Building a syntax analyzer

- Software design
 - The Jack grammar
- The Jack analyzer
- Overview
- Implementation
- Project 10

Jack analyzer

Example: Point.jack

```
/** Represents a 2D Point. */
class Point {
    ...
    method int getx() {
        // some comment
        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 analyzer outputs a parse tree,
represented by XML markup

Jack analyzer

Example: Point.jack

```
/** Represents a 2D Point. */
class Point {
    ...
    method int getx() {
        // some comment
        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 analyzer outputs a parse tree,
represented by XML markup

The analyzer handles:

- Terminals (tokens)
- Nonterminals

Jack analyzer: Terminals

Terminal elements *xxx* generate the output:

```
<tokenType> xxx </tokenType>
```

where *tokenType* is:

keyword, symbol, integerConstant,
stringConstant, identifier

Output examples

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

Jack analyzer: Nonterminals

Nonterminal element *xxx* generates the output:

```
<nonTerminal>  
    Recursive output from parsing xxx  
</nonTerminal>
```

where *nonTerminal* is:

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

Output example: parsing return x;

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

Jack analyzer: Nonterminals

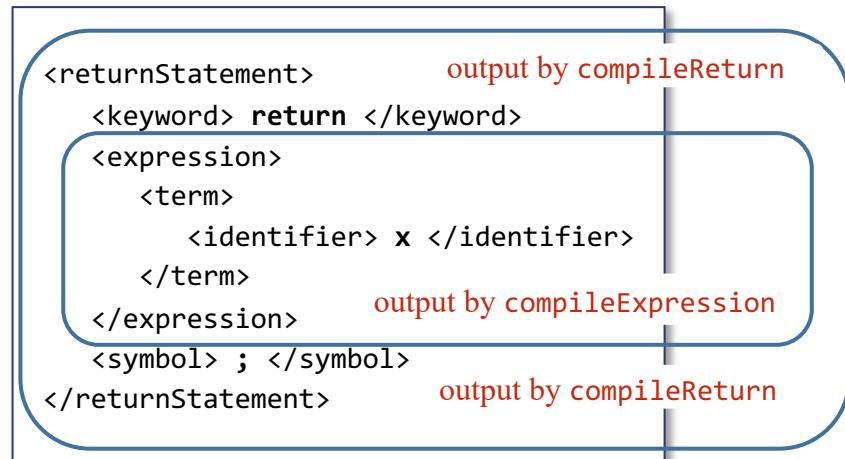
Nonterminal element *xxx* generates the output:

```
<nonTerminal>  
    Recursive output from parsing xxx  
</nonTerminal>
```

where *nonTerminal* is:

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

Output example: parsing `return x;`



Produced by:

```
// Parses a return statement. Implements the rule:  
// 'return' expression ';'  
compileReturn()  
    print("<returnStatement>")  
    process("return")  
    compileExpression()  
    process(";")  
    print("</returnStatement>")
```

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parsing

Building a syntax analyzer

- Software design
- The Jack grammar
- The Jack analyzer

Overview

→ Implementation

Project 10

Proposed implementation

Modules

→ **JackTokenizer** : Handles the input

CompilationEngine: Handles the parsing

JackAnalyzer: Drives the process.

JackTokenizer API

JackTokenizer: Provides routines that skip comments and white space, get the next token, and advance the input exactly beyond it. Other routines return the type of the current token, and its value.

Routine	Arguments	Returns	Function
Constructor / initializer	input file / stream	–	Opens the input .jack file / stream and gets ready to tokenize it.
hasMoreTokens	–	boolean	Are there more tokens in the input?
advance	–	–	Gets the next token from the input, and makes it the current token. This method should be called only if hasMoreTokens is true. Initially there is no current token.
tokenType	–	KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST	Returns the type of the current token, as a constant.
keyword	–	CLASS, METHOD, FUNCTION, CONSTRUCTOR, INT, BOOLEAN, CHAR, VOID, VAR, STATIC, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS	Returns the keyword which is the current token, as a constant. This method should be called only if tokenType is KEYWORD.
symbol	–	char	Returns the character which is the current token. Should be called only if tokenType is SYMBOL.
identifier	–	string	Returns the string which is the current token. Should be called only if tokenType is IDENTIFIER.
intval	–	int	Returns the integer value of the current token. Should be called only if tokenType is INT_CONST.
stringVal	–	string	Returns the string value of the current token, without the opening and closing double quotes. Should be called only if tokenType is STRING_CONST.

Proposed implementation

Modules

JackTokenizer : Handles the input

→ CompilationEngine: Handles the parsing

JackAnalyzer: Drives the process.

CompilationEngine API

CompilationEngine: Gets its input from a JackTokenizer and emits its output to an output file, using a set of parsing routines. Each `compilexxx` routine is responsible for handling all the tokens that make up `xxx`, advancing the tokenizer exactly beyond these tokens, and outputting the parsing of `xxx`.

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

CompilationEngine API

CompilationEngine: Gets its input from a JackTokenizer and emits its output to an output file, using a set of parsing routines. Each `compilexxx` routine is responsible for handling all the tokens that make up `xxx`, advancing the tokenizer exactly beyond these tokens, and outputting the parsing of `xxx`.

<i>Routine</i>	<i>Arguments</i>	<i>Returns</i>	<i>Function</i>
<code>compileLet</code>	–	–	Compiles a <code>let</code> statement.
<code>compileIf</code>	–	–	Compiles an <code>if</code> statement, possibly with a trailing <code>else</code> clause.
<code>compileWhile</code>	–	–	Compiles a <code>while</code> statement.
<code>compileDo</code>	–	–	Compiles a <code>do</code> statement.
<code>compileReturn</code>	–	–	Compiles a <code>return</code> statement.

API
continues in
next slide

CompilationEngine API

CompilationEngine: Gets its input from a JackTokenizer and emits its output to an output file, using a set of parsing routines. Each `compilexxx` routine is responsible for handling all the tokens that make up `xxx`, advancing the tokenizer exactly beyond these tokens, and outputting the parsing of `xxx`.

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

Parsing tip 7

The nonterminal elements *subroutineCall*, *subroutineName*, *variableName*, *className*, *type*, and *statement* are parsed directly, by other parsing methods that mention them;

Therefore, these rules have no corresponding `compilexxx` methods;

This makes the analyzer's implementation simpler.

Proposed implementation

Modules

JackTokenizer : Handles the input

CompilationEngine: Handles the parsing

→ JackAnalyzer: Drives the process.

Jack analyzer

Usage

% JackAnalyzer *input*

input: *fileName.jack*: name of a single file containing a Jack class, or

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

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

 If the input is a folder: One .xml file for every .jack file, stored in that folder

Implementation

JackAnalyzer is the main module. For each file in the input, it:

1. Creates a JackTokenizer from *fileName.jack*
2. Creates an output file named *fileName.xml*
3. Creates a CompilationEngine, and calls the compileClass method
(compileClass will then do the rest of the parsing, recursively)
4. Closes the output file.

No API is given, implement your own design.

Lecture plan

Syntax analysis

- Overview
- Tokenizer
- Grammar
- Parsing

Building a syntax analyzer

- Software design
- The Jack grammar
- The Jack analyzer

Overview

Implementation

→ Project 10

Project 10

Implementation stages

- Build a Jack tokenizer, and a basic Jack analyzer (V.0) that tests it
- Complete the Jack analyzer:

 Version 1 (handles no expressions and no array syntax)

 Version 2 (complete)

Jack tokenizer

Prog.jack

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

JackAnalyzer (v.0)

Unit-tests the
JackTokenizer:
Prints a list of tokens

ProgT.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>
```

Technical notes

- String constants (like "negative") are outputted without the double-quotes
- The symbols <, >, ", and & are outputted as <, >, ", and &
- The begin and end tags (`tokens`) are needed for XML integrity.

Jack tokenizer

Prog.jack

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

JackAnalyzer (v.0)

Unit-tests the
JackTokenizer:
Prints a list of tokens

ProgT.xml

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

Implementation plan

- Write a JackTokenizer (implement the JackTokenizer API)
- Write a JackAnalyzer (version 0) that uses a loop to iterate the input file and print one token at a time, using the services of a JackTokenizer

Project 10

Implementation stages

- Build a Jack tokenizer, and a basic Jack analyzer (V.0) that tests it

→ Complete the Jack analyzer:

 Version 1 (handles no expressions and no array syntax)

 Version 2 (complete)

Jack analyzer

Prog.jack

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

JackAnalyzer

Uses the services of a
JackTokenizer, and a
CompilationEngine

Stages

1. Build a `CompilationEngine` that handles everything except for expressions and array syntax
2. Add the handling of expressions and array syntax

We supply Jack files for unit-testing each stage.

Prog.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>  
...
```

Test file example

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;  
            ...  
        }  
    }  
}
```

Test file example (with and without expressions)

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) - 2, size, size);  
        let y = y - 2;  
        ...  
    }  
}
```

projects/10/ExpressionLessSquare/Square.jack

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

All expressions were replaced with variables in scope;
The resulting code is meaningless, but grammatically valid.

The expressionless files supports unit-testing the analyzer's ability to parse everything, except for expressions and array syntax.

What's Next: Code Generation

