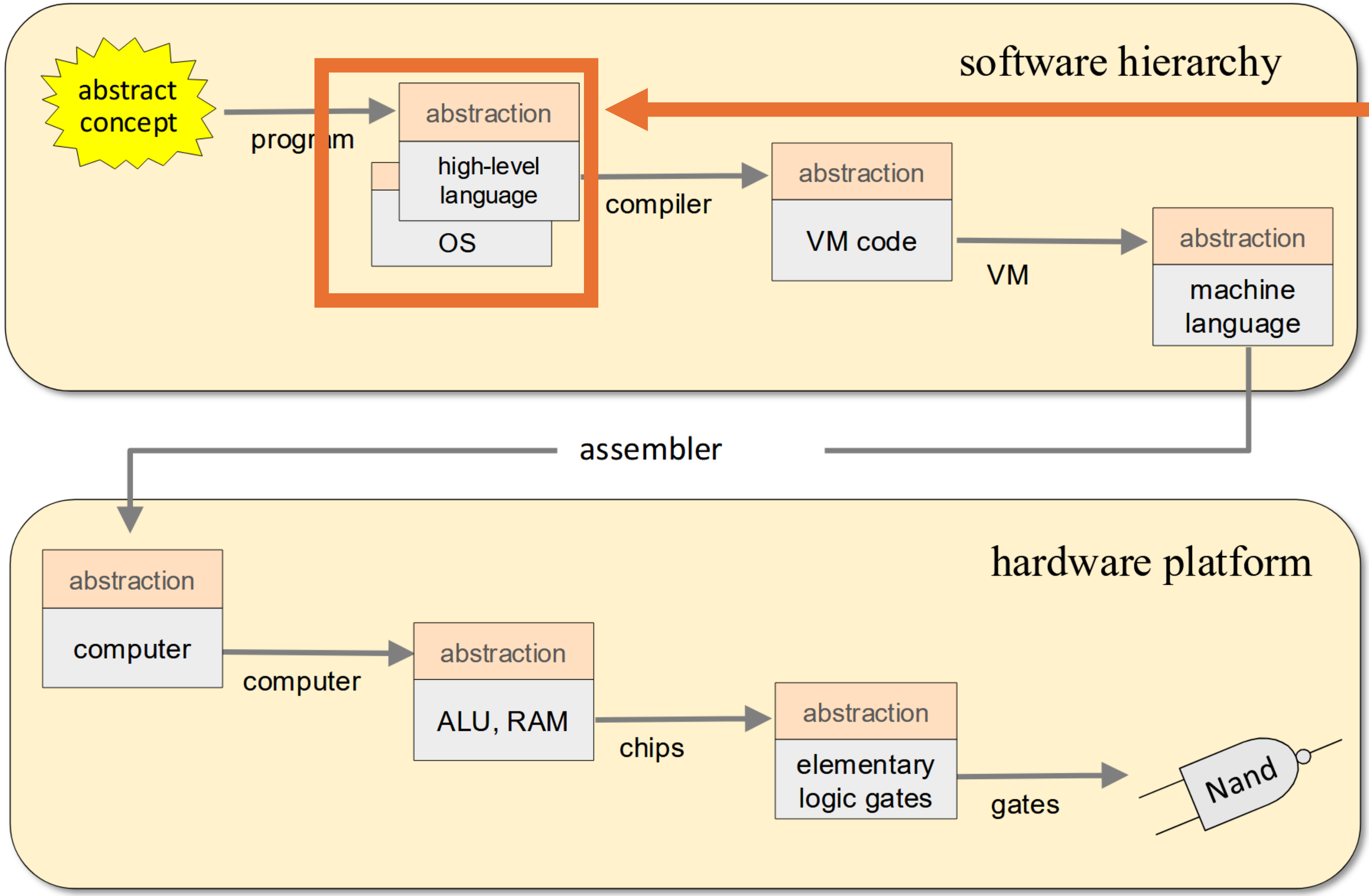
# Agenda

Compilation Part 1: Syntax Analyzer

- Tokenization
- Grammars
- Parsing
- (Aside) XML
- Recursive descent parsing
- Full Jack Grammar and parsing tips

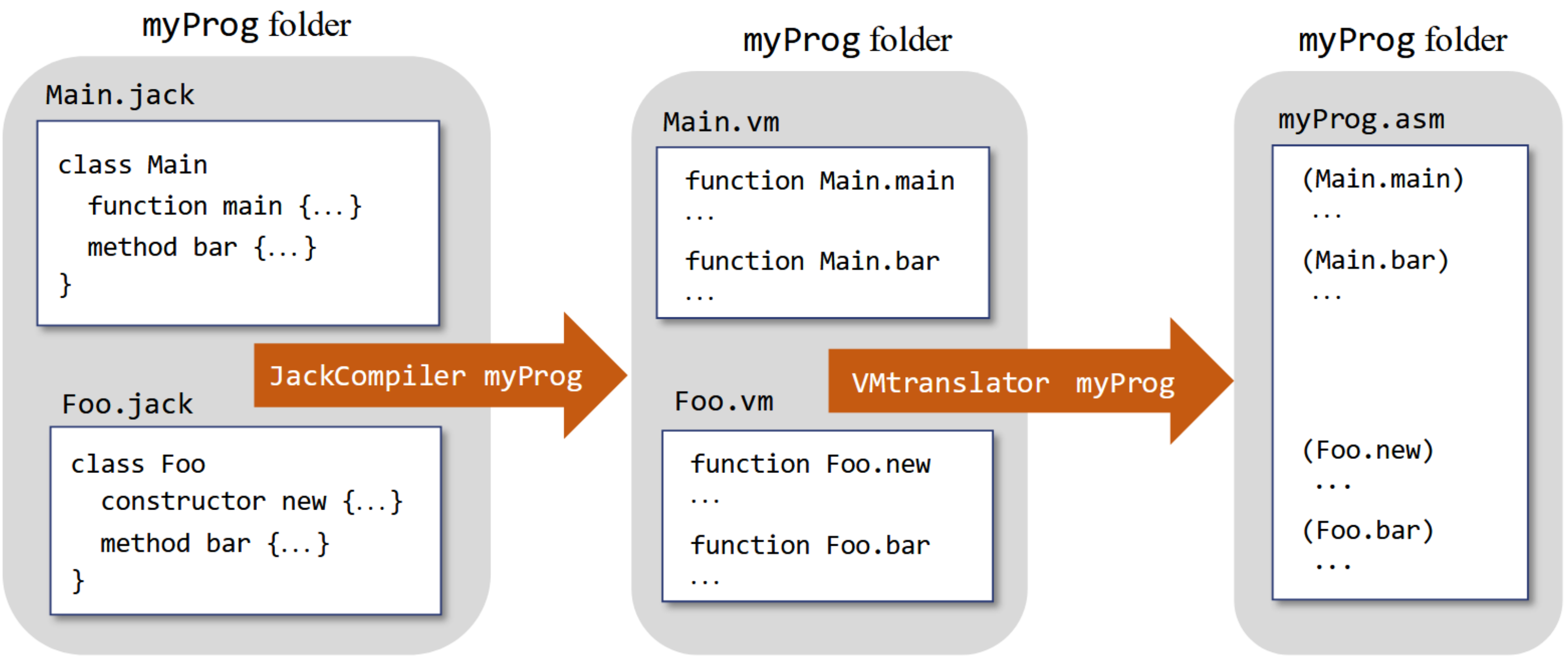
(with slides from nand2tetris)

# The big picture: Hack




Next: From high-level language to VM Code

# The big picture: Compilation steps




Each *Jack class* is a set of *methods*, *functions*, and *constructors*

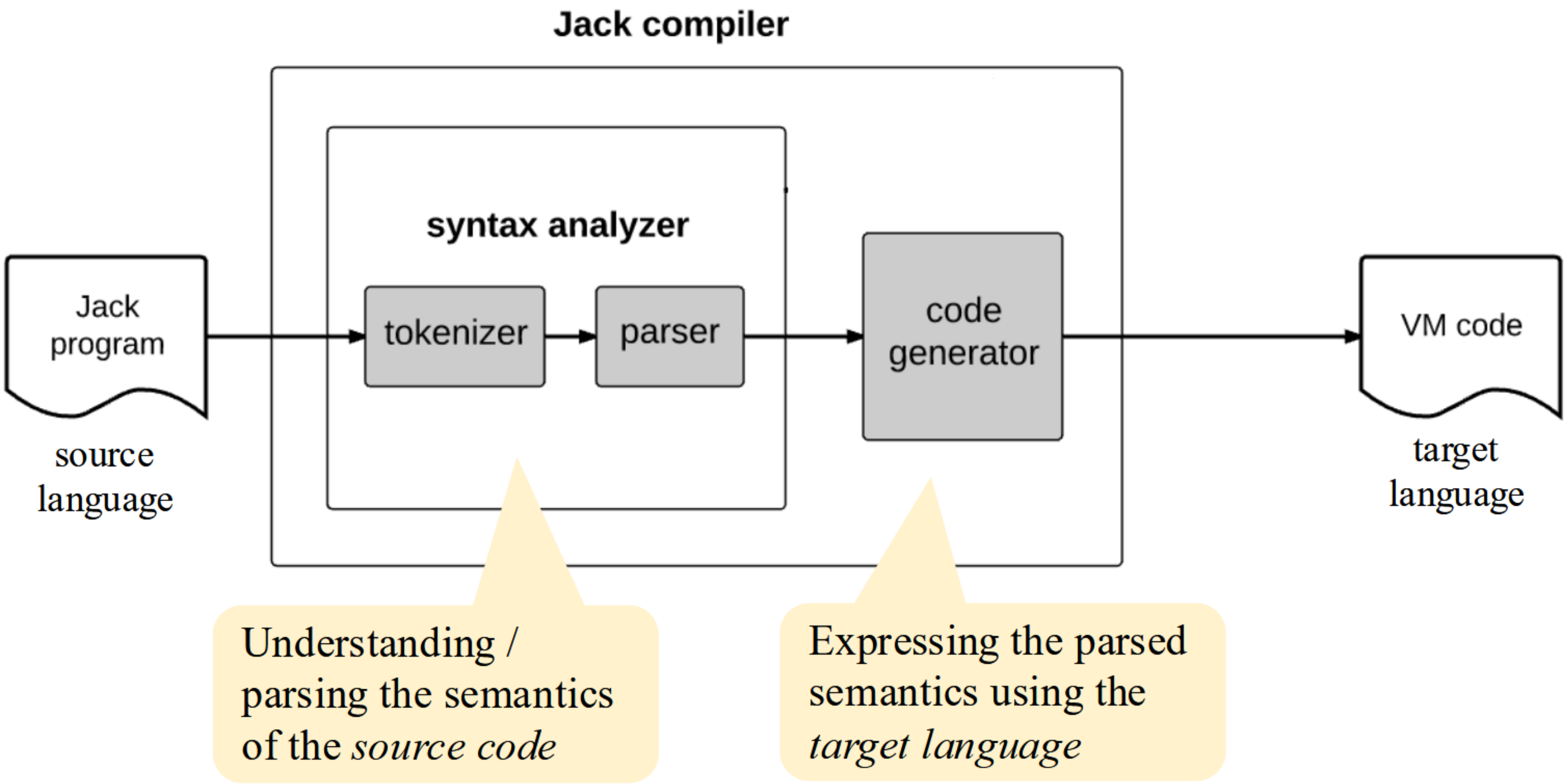
Every *method*, *function*, and *constructor* is translated into a *VM function*

All the VM functions from all the compiled class files are translated into a single assembly file

(the notion of multiple VM files melts away)

<u>Program's entry point</u>: `Main.main()`

# Compiler Roadmap



# Syntax Analyzer - Tokenizer

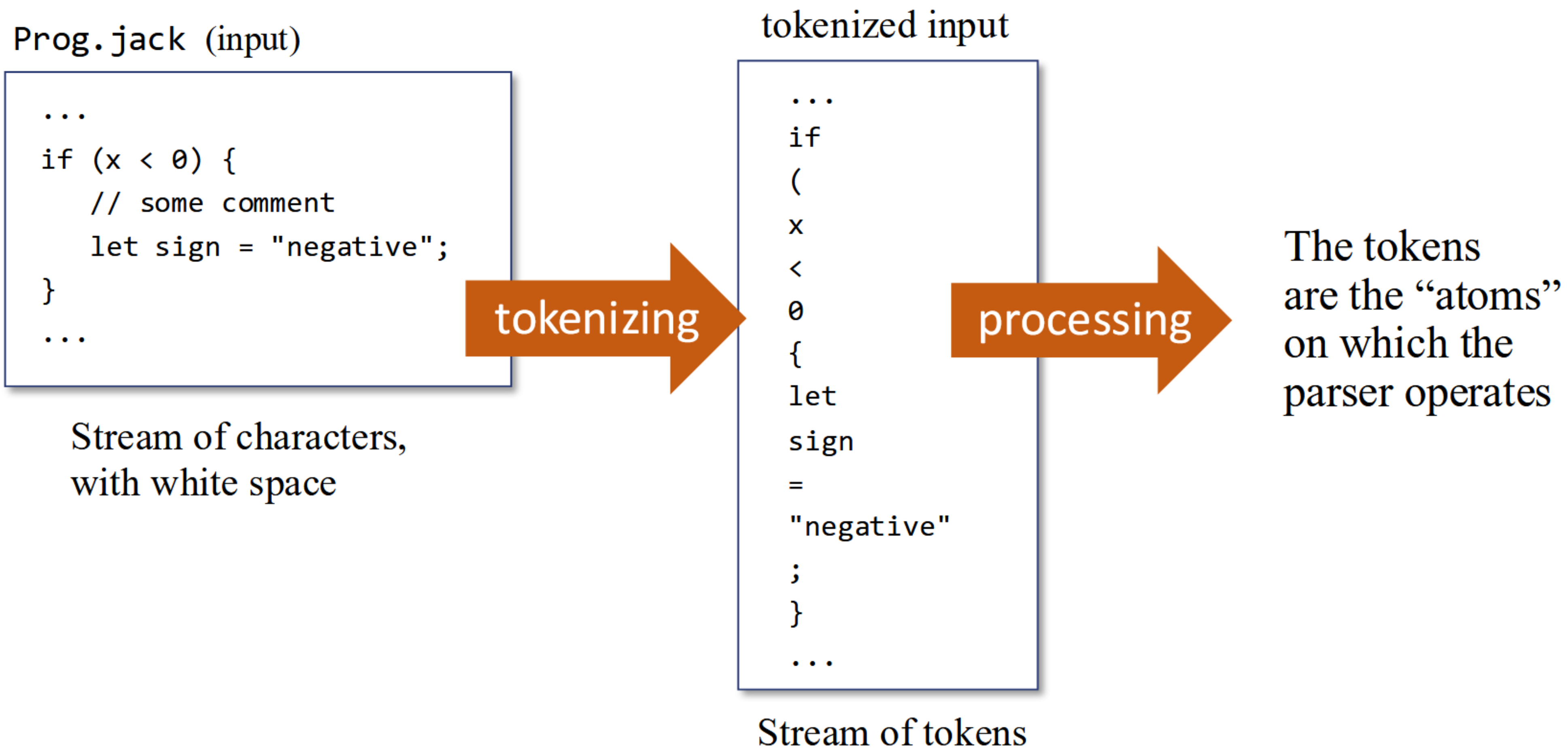
Goal: Divide input stream into a stream of tokens, or syntactic elements

A **lexicon** is the valid set of tokens. For jack, this corresponds to

- keywords (e.g. while, if)
- symbols (e.g. {,], etc)
- integers (e.g. 0,1,2,3, etc)
- strings (e.g. “apple”)
- identifiers (e.g. variable names)

**Tokenizing** is the process of producing tokens

# Tokenizing



Prog.jack (input)
...
if (x < 0) {
// some comment
let sign = "negative";
}
...
Stream of characters, with white space
tokenizing
tokenized input
...
if
(
x
<
0
{
let
sign
=
"negative"
;
}
...
Stream of tokens
processing
The tokens are the “atoms” on which the parser operates

# Syntactic elements

<table>
<tr><td rowspan="4">White Space and comments</td><td>Space characters, newlines, and comments are ignored</td></tr>
<tr><td>// Comment to end</td></tr>
<tr><td>/* comment until closing */</td></tr>
<tr><td>/** API doc command */</td></tr>
<tr><td>Symbols</td><td>() [] {} , ; = . +- */ & | - < ></td></tr>
<tr><td rowspan="5">Reserved Words (keywords)</td><td>class, constructor, method, function, this</td></tr>
<tr><td>int, Boolean, char, void</td></tr>
<tr><td>var, static, field</td></tr>
<tr><td>let, do, if, else, while, return</td></tr>
<tr><td>true, false, null</td></tr>
<tr><td rowspan="4">Constants</td><td>integers (e.g. 0..32707)</td></tr>
<tr><td>Strings (e.g. “a string constant is in quotes”</td></tr>
<tr><td>Boolean (true or false)</td></tr>
<tr><td>null</td></tr>
<tr><td>Identifiers</td><td>strings consisting of letters, digits, and _. Cannot start with a digit.</td></tr>
</table>

# Example: Tokenize the code

```
while (count <= 100) {
  let count = count + 1;
}
let city = “Paris”;
```

Give the token and its type

# SyntaxAnalyzer - Parser

A **grammar** is a set of rules that define a set of valid strings

The output of the tokenizer will be passed to the **parser**

The parser tests whether a stream of token satisfies a grammar

# Grammar definitions

Grammar rules have the form *ruleName : ruleDefinition*

A rule definition can be one of

- ‘terminal_string’ ← a verbatim string
- nonterminal_string ← a lexicon element
- x y ← x then y
- (x y) ← grouping of x and y
- x|y ← either x or y
- x? ← x can appear 1 or more times
- x* ← x can appear 0 or more times

# Exercise: Connect the strings to their rules

- (ab)?
- ab(c)*
- a*|c*

- ab
- “” (empty string)
- abababab
- abc
- abcccccc
- aaa
- a
- c
- accccc
- aaaaaaaac

# Example: Grammar

## Tiny English grammar (subset)

| | |
|---|---|
| *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
```

# Exercise: Which phrases are valid for this grammar?

Tiny English grammar (subset)

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

the dog ate my school

the girl said homework

to school went to she

# Jack grammar examples

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

A **nonterminal** is a rule whose definition contains other rules

A **terminal** is a rule that cannot be expanded further

# Jack grammar examples

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

Example: let a = 3;

Example: let n = n + 1;

# Exercise: Jack 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*: | '**+**' \| '**-**' \| '**=**' \| '**>**' \| '**<**' |

Match the grammar rules for the following code

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

# Parsing

Goal: Tests whether a stream of token satisfies a grammar

English grammar (subset)

| | |
|---|---|
| *sentence*: | *nounPhrase verbPhrase* |
| *nounPhrase*: | *determiner noun* |
| *verbPhrase*: | *verb nounPhrase* |
| *noun*: | `'dog'`\|`'school'`\|`'girl'`\|<br>`'he'`\|`'she'`\|`'homework'`\| … |
| *verb*: | `'went'`\|`'ate'`\|`'said'`\| … |
| *determiner*: | `'the'`\|`'to'`\|`'my'`\| … |
| . . . | |

Input

```
The dog ate my homework
```

parse

# Exercise: Derive the parse tree

Jack grammar (subset)

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

*statements*: *statement* *

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

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

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

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

*term*: *varName* | *constant*

*varName*: a string not beginning with a digit

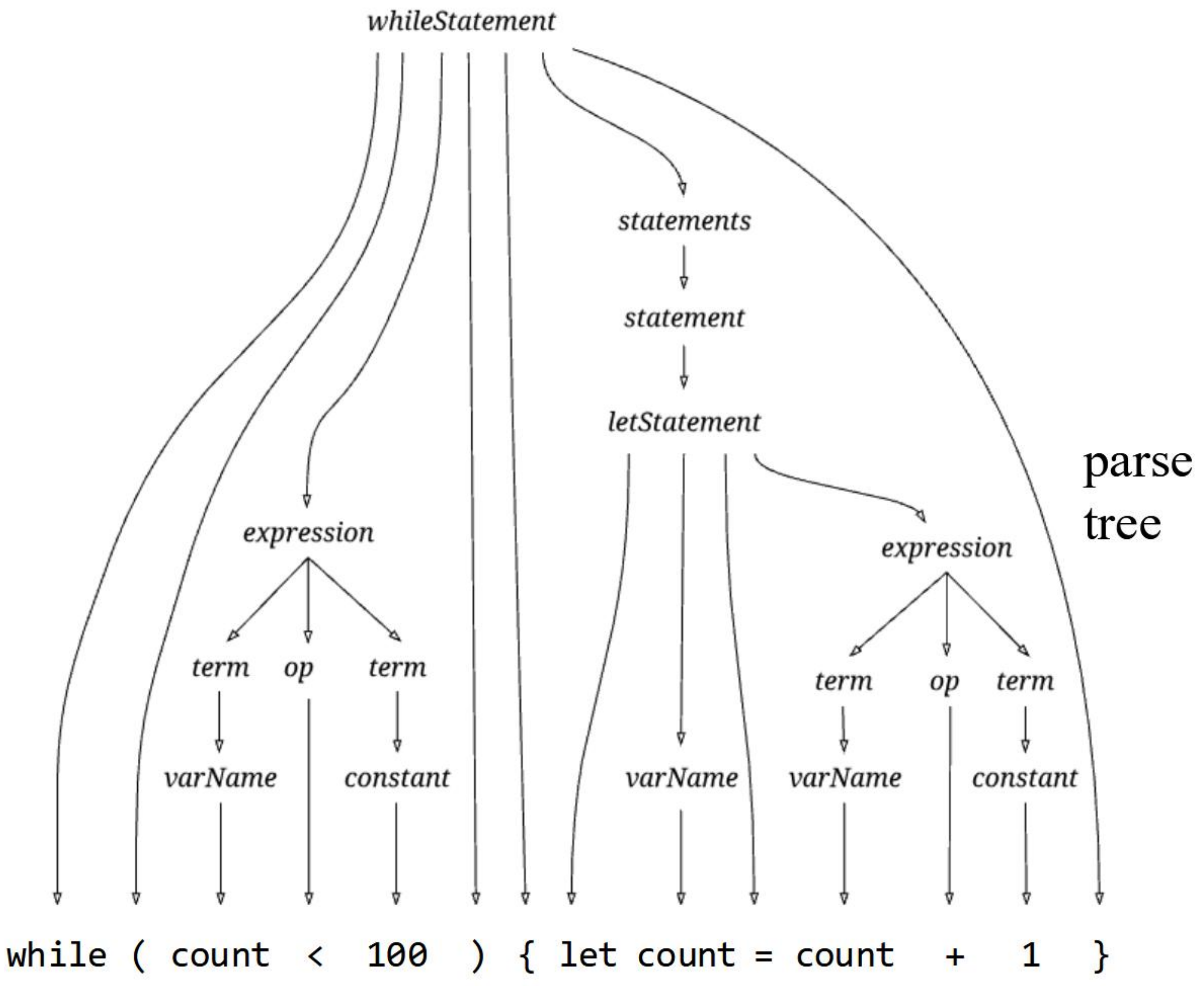
*constant*: a decimal number

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

Input

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

parse



# (Aside) XML

Extensible Markup language

Data is stored between tags with the following structure

<tag> data </tag>

We will use XML to output tokenized and parsed output for debugging our compiler

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

```
...
<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>
...
```

# Recursive descent parsing

## Idea: Recursively compile program using grammar rules

```
// 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")
}
```

# LL grammars

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

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

LL(1) grammar: The first token is sufficient. Jack is a LL(1) grammar with one exception (later)

=> parsing LL(1) grammars is straight-forward

# Full 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.

<u>Parsing tip 1</u>

This part of the Jack grammar informs the implementation of the Jack Tokenizer;

(informs how to group characters into tokens).

# Full 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 grammar rules specify *element** , which is a sequence of elements of any length;

To handle, the corresponding parsing methods can use loops.

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

# Full 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 grammar rules specify two or more parsing possibilities
(for example: the first token of a *classVarDec* is either `static`, or `field`);

To handle: The `process(String)` helper method shown before can be refined to get a list/set of possible strings as a parameter.

# Full 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*

Therefore, 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.

# Full 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 5

A *subroutineCall* is an *expression*

(expressions are discussed next);

Therefore, it is recommended to parse `do` *subroutineCall* statements as if they were `do` *expression* statements.

# Full 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

A *subroutineCall* (for example, `calc(x)`) occurs in one place only: when parsing a *term*

To handle: Instead of writing a separate `compileSubroutineCall` method, have your `compileTerm` method parse subroutine calls directly.

# Full 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' |

Lexicon reminder:

| | |
|---|---|
| varName | identifier |
| className: | idnetifier |
| subroutineName: | identifier |
| identifier: | a sequence of letters, digits, and underscore ('_'), not starting with a digit |

Parsing scenario: Suppose that we are parsing the expression...

```
y + arr[5] - p.get(row) * count() - Math.sqrt(dist)
```

... and *currentToken* is `y`, `arr`, `p`, `count`, or `Math`

In each case we know that we have a *term* that begins with an *identifier*;

Question: Which parsing possibility to follow next?

Answer: The *next token* is sufficient to resolve which option we are in.

## Parsing tip 7

When parsing a *term* that begins with an *identifier*:

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

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

# 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>
```

For your final project, milestone 1

The analyzer handles:

- Terminals (tokens)
- Nonterminals

# 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 handles:

- Terminals (tokens)
- Nonterminals

For your final project, milestone 1