

Compiling Jack

John Lapinskas, University of Bristol

Lexing Jack

Lexing Jack has no surprises and uses exactly the same techniques you've seen before. Here are the tokens:

- **Keywords:**

- 'class', 'constructor', 'function', 'method', 'field', 'static'.
- 'int', 'char', 'boolean', 'void'.
- 'var', 'let', 'do', 'if', 'else', 'while', 'return'.
- 'true', 'false', 'null', 'this'.

- **Symbols:**

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

- **Integer literals:** Any base-10 integer in the range 0...32767.

- **String literals:** Any sequence of characters enclosed in "s that doesn't include newlines or additional "s.

- **Identifiers:** Any sequence of letters, digits, and '_'s not starting with a digit and that's not a keyword.

Notice that identifiers are defined more restrictively than in Hack VM/assembly. This makes it possible to lex e.g. `myVar+4` unambiguously without spaces.

Also notice that newlines are not tokens in Jack! Like in C, they're just whitespace.

Parsing Jack: The grammar

Remember, in our EBNF dialect, $\{\}$ means “repeat any number of times (including zero)” and $[]$ means “this is optional”.

```

    <class> ::= 'class', identifier, '{', {<classVarDec>}, {<subroutineDec>}, '}';
    <classVarDec> ::= ('static' | 'field'), <type>, identifier, {'', identifier}, ';';
    <type> ::= 'int' | 'char' | 'boolean' | identifier;
    <subroutineDec> ::= ('constructor' | 'function' | 'method'), ('void' | <type>),
        identifier, '(', <parameterList>, ')', <subroutineBody>;
    <parameterList> ::= [(<type>, identifier, {'', <type>, identifier});
    <subroutineBody> ::= '{', {<varDec>}, <statements>, '}';
    <varDec> ::= 'var', <type>, identifier, {'', identifier}, ';';
```

Notice that all variables must be declared at the start of a class (in $\langle\text{classvarDec}\rangle$ s) or function (in $\langle\text{varDec}\rangle$ s).

This will make building symbol tables much easier. We have all the information we need about the variables before needing to generate any code involving them!

All we have left to define is $\langle\text{statements}\rangle$.

Parsing Jack: The grammar

Remember, in our EBNF dialect, $\{ \}$ means “repeat any number of times (including zero)” and $[]$ means “this is optional”.

```
 $\langle \text{statements} \rangle ::= \{ \langle \text{letStatement} \rangle \mid \langle \text{ifStatement} \rangle \mid \langle \text{whileStatement} \rangle \mid$   
 $\langle \text{returnStatement} \rangle \mid \langle \text{doStatement} \rangle \}$   
 $\langle \text{letStatement} \rangle ::= \text{'let'}$ , identifier,  $[ \text{'['}$ ,  $\langle \text{expression} \rangle$ ,  $\text{'}]'$ ,  $\text{'='}$ ,  $\langle \text{expression} \rangle$ ,  $\text{';'}$ ;  
 $\langle \text{ifStatement} \rangle ::= \text{'if'}$ ,  $\text{'('}$ ,  $\langle \text{expression} \rangle$ ,  $\text{'}'$ ,  $\text{'{'}$ ,  $\langle \text{statements} \rangle$ ,  $\text{'}'$ ,  $[\text{'else'}$ ,  $\text{'{'}$ ,  $\langle \text{statements} \rangle$ ,  $\text{'}'$ ];  
 $\langle \text{whileStatement} \rangle ::= \text{'while'}$ ,  $\text{'('}$ ,  $\langle \text{expression} \rangle$ ,  $\text{'}'$ ,  $\text{'{'}$ ,  $\langle \text{statements} \rangle$ ,  $\text{'}'$ ;  
 $\langle \text{doStatement} \rangle ::= \text{'do'}$ ,  $\langle \text{subroutineCall} \rangle$ ,  $\text{';'}$ ;  
 $\langle \text{returnStatement} \rangle ::= \text{'return'}$ ,  $[\langle \text{expression} \rangle]$ ,  $\text{';'}$ 
```

Notice the $\langle \text{statements} \rangle$ recursion in $\langle \text{ifStatement} \rangle$ s and $\langle \text{whileStatement} \rangle$ s. A $\langle \text{statements} \rangle$ can't contain any statements declaring new variables (which only appear in $\langle \text{varDec} \rangle$ s or $\langle \text{classVarDec} \rangle$ s), so we don't have to track separate scopes.

Remember from last video that an **expression** is anything that returns a value. We still have to define $\langle \text{expression} \rangle$ s and $\langle \text{subroutineCall} \rangle$ s.

Parsing Jack: The grammar

Remember, in our EBNF dialect, $\{\}$ means “repeat any number of times (including zero)” and $[]$ means “this is optional”.

```
 $\langle \text{expression} \rangle ::= \langle \text{term} \rangle, \{ ('+' | '-' | '*' | '/' | '\&' | '|' | '<' | '>' | '=') , \langle \text{term} \rangle \}$   
 $\langle \text{term} \rangle ::= \text{integer literal} | \text{string literal} | \text{'true'} | \text{'false'} | \text{'null'} | \text{'this'} |$   
                   $\text{identifier} , [ '[' , \langle \text{expression} \rangle , ']' ] | '(' , \langle \text{expression} \rangle , ') ' |$   
                   $(( '-' | '-' ) , \langle \text{term} \rangle) | \langle \text{subroutineCall} \rangle ;$   
 $\langle \text{subroutineCall} \rangle ::= \text{identifier} , [ '.' , \text{identifier} ] , '(' , \langle \text{expressionList} \rangle , ') ' ;$   
 $\langle \text{expressionList} \rangle ::= [ \langle \text{expression} \rangle , \{ ',' , \langle \text{expression} \rangle \} ] ;$ 
```

Notice how monstrously recursive expressions are! And this recursion is absolutely necessary to capture expressions like `Math.sqrt((x+y)*Main.fibonacci(z))`.

We've been putting it off for a long time, but we will absolutely need a parse tree to handle these, which means at last separating parsing from code generation...

Thankfully, Jack is an LL(2) language (and close to an LL(1) language) so this process won't be too bad.

Parsing Jack: Use of XML

We do need to decide how to store our parse tree between our parsing pass through the file and our subsequent pass(es) for code generation. Ideally, we want something that's:

- Human-readable enough to use for debugging.
- Standard enough that we can read it from/write it to a file without too much effort of our own.
- Flexible enough that we can navigate through the parse tree without having to store the whole thing in memory at once (which will be pretty slow and system-intensive for a large codebase).

The nand2tetris course correctly identifies **XML** as a good choice on all three counts — most languages can read and navigate it with standard libraries.

Of course, C is not most languages...

Parsing Jack: Use of XML

We do need to decide how to store our parse tree between our parsing pass through the file and our subsequent pass(es) for code generation. Ideally, we want something that's:

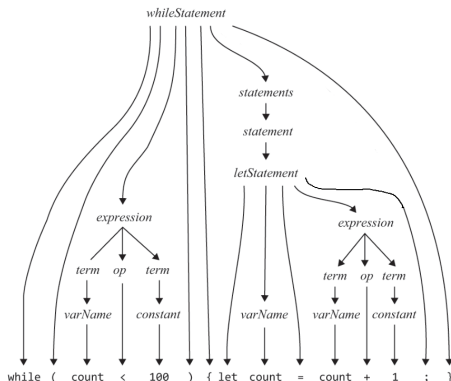
- Human-readable enough to use for debugging.
- Standard enough that we can read it from/write it to a file without too much effort of our own.
- Flexible enough that we can navigate through the parse tree without having to store the whole thing in memory at once (which will be pretty slow and system-intensive for a large codebase).

The nand2tetris course correctly identifies **XML** as a good choice on all three counts — most languages can read and navigate it with standard libraries.

Of course, C is not most languages... so we turned the `token` code from weeks 8–10 into our own budget XML-handling library and put it in the code skeleton.

The lexer will likewise write tokens in XML format (using the same library), to make the code as consistent as possible.

Example XML-format parse tree



Prog.xml

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

Source: Nisan and Schocken, Figures 10.4a and 10.4b, with edit to fix an error in 10.4a. These figures contain several superfluous non-terminals, like `<op>` and `<varName>`, which are omitted from the definition above — see later.

Parsing Jack: A (fairly) general LL(2) parser

In parsing our list of tokens, we will maintain pointers to two tokens: `current` and `lookahead`. At all times, `current` will be the first token we haven't parsed and `lookahead` will be the second token.

Each non-terminal will get its own function, e.g. `parse_term` and `parse_statements`.

Our goal in the `parse_abc` function is:

- Suppose at the start of the function call that `current` is the first token of an `<abc>` non-terminal.
- Generate all the XML for that non-terminal (using the given `write_tag` and `copy_tag` functions).
- Return with `current` pointing to the first token past the end of the `<abc>`.

At each stage, we will be able to place `current` in the XML file without needing to advance further in the token list than `lookahead`.¹

To achieve this, recursion is our friend!

¹We'll usually only need `current`, but we'll need `lookahead` to parse `<term>`s. If the `<term>`'s first token is an identifier, it could be either an identifier within the `<term>` (i.e. a variable) or an identifier within a `<subroutineCall>` (i.e. a function).

An example of parsing Jack

```
→ <keyword> while </keyword>  
--> <symbol> ( </symbol>  
    <identifier> count </identifier>  
    <symbol> < </symbol>  
    <integerLiteral> 100 </integerLiteral>  
    <symbol> ) </symbol>  
    <symbol> { </symbol>  
    <keyword> let </keyword>  
    <identifier> count </identifier>  
    <symbol> = </symbol>  
    <identifier> count </identifier>  
    <symbol> + </symbol>  
    <integerLiteral> 1 </integerLiteral>  
    <symbol> ; </symbol>  
    <symbol> } </symbol>
```

Token list

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

Jack code

→ **current** --> **lookahead**

Starting `parse_while_statement` call, opening tag with `write_tag`.

An example of parsing Jack

```
→ <keyword> while </keyword>  
--> <symbol> ( </symbol>  
    <identifier> count </identifier>  
    <symbol> < </symbol>  
    <integerLiteral> 100 </integerLiteral>  
    <symbol> ) </symbol>  
    <symbol> { </symbol>  
    <keyword> let </keyword>  
    <identifier> count </identifier>  
    <symbol> = </symbol>  
    <identifier> count </identifier>  
    <symbol> + </symbol>  
    <integerLiteral> 1 </integerLiteral>  
    <symbol> ; </symbol>  
    <symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting `parse_while_statement` call, opening tag with `write_tag`.

```
<whileStatement>
```

An example of parsing Jack

```
→  
--> <keyword> while </keyword>  
<symbol> ( </symbol>  
<identifier> count </identifier>  
<symbol> < </symbol>  
<integerLiteral> 100 </integerLiteral>  
<symbol> ) </symbol>  
<symbol> { </symbol>  
<keyword> let </keyword>  
<identifier> count </identifier>  
<symbol> = </symbol>  
<identifier> count </identifier>  
<symbol> + </symbol>  
<integerLiteral> 1 </integerLiteral>  
<symbol> ; </symbol>  
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Inside `parse_while_statement`, copying tokens with `copy_tag`.

```
<whileStatement>
```

An example of parsing Jack



```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

Inside `parse_while_statement`, copying tokens with `copy_tag`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Inside `parse_while_statement`, copying tokens with `copy_tag`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_expression call, opening tag with write_tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_expression call, opening tag with write_tag.

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


An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_term call, opening tag with write_tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_term call, opening tag with write_tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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


Jack code

→ current --> lookahead

From lookahead we see this is not a
(subroutineCall), copying identifier tag.

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

An example of parsing Jack



```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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


Jack code

 current  lookahead

From lookahead we see this is not a
(subroutineCall), copying identifier tag.

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

An example of parsing Jack



```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

 current  lookahead

Since current is not '[' the $\langle \text{term} \rangle$ is over. Closing tag with `write_tag` and returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Since current is not '[' the <term> is over. Closing tag with write_tag and returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming parse_expression call. Since current is '<' the (expression) isn't over. Copying token.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming parse_expression call. Since current is '`<`' the (expression) isn't over. Copying token.

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


An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_term call, opening tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_term call, opening tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

An integer literal must end the `<term>`. Copying token, then closing/returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

An integer literal must end the `<term>`. Copying token, then closing/returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

An integer literal must end the `<term>`. Copying token, then closing/returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming parse_expression call. Since current is '=', the <expression> is over. Closing token and returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming parse_expression call. Since current is '=', the <expression> is over. Closing token and returning.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming `parse_while_statement` call. Copy the next two tokens.

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


An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming `parse_while_statement` call. Copy the next two tokens.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming `parse_while_statement` call. Copy the next two tokens.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_statements call, opening tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Starting parse_statements call, opening tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Since current is 'let', this is a <letStatement>. Starting parse_let_statement call, opening tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Since current is 'let', this is a <letStatement>.
Starting parse_let_statement call, opening tag.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Copying first two tokens.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Copying first two tokens.

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


An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

Copying first two tokens.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

Since current is not '[', copy one more token then call parse_expression.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Since current is not '[', copy one more token then call parse_expression.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Since current is not '[', copy one more token then call parse_expression.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

This proceeds similarly to before until returning from `parse.expression`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

This proceeds similarly to before until returning from `parse.expression`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

This proceeds similarly to before until returning from `parse.expression`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

This proceeds similarly to before until returning from `parse.expression`.

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


An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

This proceeds similarly to before until returning from `parse.expression`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current -> lookahead

This proceeds similarly to before until returning from `parse.expression`.

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

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

This proceeds similarly to before until returning from `parse.expression`.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

This proceeds similarly to before until returning from `parse.expression`.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

This proceeds similarly to before until returning from `parse.expression`.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current - -> lookahead

Resuming parse_let_statement call. Last token is a
';', copy it and return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Resuming parse_let_statement call. Last token is a
';', copy it and return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current -> lookahead

Resuming parse_let_statement call. Last token is a
';', copy it and return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
</letStatement>
```


An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current -> lookahead

Resuming parse_statements call. Since current is not let, while, if, do or return, (statements) is over. Return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
</letStatement>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current -> lookahead

Resuming parse_statements call. Since current is not let, while, if, do or return, (statements) is over. Return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
</letStatement>
</statements>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current --> lookahead

Last token is always '}', copy it and (at last!) return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
</letStatement>
</statements>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current -> lookahead

Last token is always '}', copy it and (at last!) return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
</letStatement>
</statements>
<symbol> } </symbol>
```

An example of parsing Jack

```
<keyword> while </keyword>
<symbol> ( </symbol>
<identifier> count </identifier>
<symbol> < </symbol>
<integerLiteral> 100 </integerLiteral>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> count </identifier>
<symbol> = </symbol>
<identifier> count </identifier>
<symbol> + </symbol>
<integerLiteral> 1 </integerLiteral>
<symbol> ; </symbol>
<symbol> } </symbol>
```

Token list

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

Jack code

→ current -> lookahead

Last token is always '}', copy it and (at last!) return.

```
<whileStatement>
<keyword> while </keyword>
<symbol> ( </symbol>
<expression>
<term>
<identifier> count </identifier>
</term>
<symbol> < </symbol>
<term>
<integerLiteral> 100 </integerLiteral>
</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>
<integerLiteral> 1 </integerLiteral>
</term>
</expression>
<symbol> ; </symbol>
</letStatement>
</statements>
<symbol> } </symbol>
</whileStatement>
```

Parsing Jack: Building an AST

We've already removed a lot of superfluous classes from the grammar in Nisan and Schocken to make the parse tree nicer.

In addition, you should remove the `<type>` class in building your parse tree — replace it with its definition everywhere it appears in the grammar.²

²Nisan and Schocken also recommend removing `<subroutineCall>` from the grammar in parsing, but this is because they're assuming most people will use a janky self-built parser which treats lookahead as an ugly special case. The method used in our skeleton will work nicely for any LL(2) language, so this isn't needed, and indeed having access to the `<subroutineCall>` tags is useful in the code generation step. They recommend removing some other non-terminals, which we've already done in our definition. The original can be found in Nisan and Schocken Figure 10.5 if you're curious.

Parsing Jack: Building an AST

We've already removed a lot of superfluous classes from the grammar in Nisan and Schocken to make the parse tree nicer.

In addition, you should remove the `<type>` class in building your parse tree — replace it with its definition everywhere it appears in the grammar.²

You'll probably find yourself wanting to remove other parts of the parse tree too. Things like `'if'` and `{}`s and `()`s are useful in turning a list of tokens into a parsed `<ifStatement>`, but they're the same between all `<ifStatement>`. So do you really need them as parse tree nodes to turn that `<ifStatement>` into code?

²Nisan and Schocken also recommend removing `<subroutineCall>` from the grammar in parsing, but this is because they're assuming most people will use a janky self-built parser which treats lookahead as an ugly special case. The method used in our skeleton will work nicely for any LL(2) language, so this isn't needed, and indeed having access to the `<subroutineCall>` tags is useful in the code generation step. They recommend removing some other non-terminals, which we've already done in our definition. The original can be found in Nisan and Schocken Figure 10.5 if you're curious.

Parsing Jack: Building an AST

We've already removed a lot of superfluous classes from the grammar in Nisan and Schocken to make the parse tree nicer.

In addition, you should remove the `<type>` class in building your parse tree — replace it with its definition everywhere it appears in the grammar.²

You'll probably find yourself wanting to remove other parts of the parse tree too. Things like `'if'` and `{}`s and `()`s are useful in turning a list of tokens into a parsed `<ifStatement>`, but they're the same between all `<ifStatement>`. So do you really need them as parse tree nodes to turn that `<ifStatement>` into code?

This is exactly what an AST is for, and there's no single right choice here. We recommend you:

- Write code which includes every node of the parse tree.
- Test it against the XML outputs provided with `fc` or `diff`.
- Consider what you'd like to leave out of the AST, and replace calls to `copy_tag` with calls to `advance_tag` accordingly.

²Nisan and Schocken also recommend removing `<subroutineCall>` from the grammar in parsing, but this is because they're assuming most people will use a janky self-built parser which treats lookahead as an ugly special case. The method used in our skeleton will work nicely for any LL(2) language, so this isn't needed, and indeed having access to the `<subroutineCall>` tags is useful in the code generation step. They recommend removing some other non-terminals, which we've already done in our definition. The original can be found in Nisan and Schocken Figure 10.5 if you're curious.

Semantic analysis: Mapping variables to Hack VM memory

We store variables in Hack VM memory as follows.

- Variables declared with `var` are stored in the `local` segment. These are all declared in the `<varDec>`s at the start of the `<subroutineBody>`.
- Function arguments are stored in the `argument` segment. These are all declared in the `<parameterList>` in the `<subroutineDec>`.
- Variables declared with `static` are stored in the `static` segment. These are all declared in the `<classVarDec>`s at the start of the `<class>`.
- Variables declared with `field` are stored in the `this` segment (see next video). These are all declared in the `<classVarDec>`s at the start of the `<class>`.
- The `that` segment allows for smooth compilation of expressions involving both `[]`s and `field` variables, using `this` for the field variables and `that` for the `[]`s.
- The `temp` segment allows for temporary storage while compiling a single statement.
- The purposes of the `pointer` and `constant` “segments” are already clear.

Semantic analysis: Creating symbol tables

Jack was designed very carefully to avoid needing a separate pass through the file to generate symbol tables or handle other semantic analysis.

After parsing, we move directly to code generation, scanning through the file from start to finish and converting XML into code as we read it. To do this for a `<subroutineDec>`, on reaching the opening XML tag:

- Create a new symbol table for the subroutine.
- If this is a `method`, add one `argument` entry for `this` with offset 0. (You'll understand why next video!)
- Add one `argument` entry for each variable in the `<parameterList>`.
- Add one `local` entry for each `<varDec>` in the `<subroutineBody>`.
- Use your symbol table in generating code for the `<statements>` of the `<subroutineBody>`.
- On reaching the end of the `<subroutineBody>`, free your symbol table (which will no longer be needed).

Next video, we'll handle `field` and `static` variables very similarly in a `<class>`.

Semantic analysis: An example symbol table

High-level (Jack) code

```
class Point {  
  field int x, y;  
  static int pointCount;  
  ...  
  method int distance(Point other) {  
    var int dx, dy;  
    let dx = x - other.getx();  
    let dy = y - other.gety();  
    return Math.sqrt((dx*dx) + (dy*dy));  
  }  
  ...  
}
```

name	type	kind	#
x	int	field	0
y	int	field	1
pointCount	int	static	0

Class-level
symbol table

name	type	kind	#
this	Point	arg	0
other	Point	arg	1
dx	int	var	0
dy	int	var	1

Subroutine-level
symbol table

Source: Nisan and Schocken Figure 11.2.

Each symbol table entry should contain the variable name (which can be used as a lookup as with the assembler), its type (as a string), its “kind” (e.g. local or argument), and its offset (which can be mapped greedily).

For example, the table above maps `this` to argument 0, `other` to argument 1, `dx` to local 0 and `dy` to local 1.

Generating code: The true power of recursion

To actually use your symbol tables and convert the parsed Jack code into Hack VM code, you'll use a very similar recursive structure as for the parsing itself.

For example, recall that

```
⟨whileStatement⟩ ::= 'while', '(', ⟨expression⟩, ')', '{', ⟨statements⟩, '}'.
```

And you've known how to create a `while` loop with `gotos` and `labels` since week 5! So in the `compile_while` function, you might:

- Generate two unused Hack VM labels, say `while_start_4` and `while_end_4`.
- Output Hack VM code `label while_start_4`.
- Navigate to the `⟨expression⟩` with `advance_tag` and call `compile_expression` to output Hack VM code that pushes the result of the loop condition onto the stack.
- Output Hack VM code `not and if-goto while_end_4`.
- Navigate to the `⟨statements⟩` of the loop body with `advance_tag` and call `compile_statements` to output Hack VM code that runs the loop body.
- Output Hack VM code `goto while_start_4` and `label while_end_4`.
- Navigate past the `'}'` and `</whileStatement>` tag with `advance_tag` and return from `compile_while`.

Generating code: The true power of recursion

As another example, recall that

$$\langle \text{expression} \rangle ::= \langle \text{term} \rangle, \{ ('+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=') , \langle \text{term} \rangle \}.$$

So in the `compile_expression` function, you might:

- Call `compile_term` to output Hack VM code that pushes the result of the first `<term>` onto the stack.
- While there are `<term>`s left in the `<expression>`:
 - Store a copy of the next operation.
 - Navigate to the next `<term>` with `advance_tag`, then call `compile_term` to output Hack VM code that pushes the result of the `<term>` onto the stack.
 - Generate and output Hack code that carries out the last specified operation on the two top values of the stack. For example, for `'+'` this would be `add`, and for `'*'` this would be `'call Math.multiply 2'`.
- Return from `compile_expression`.

(Remember that operator precedence doesn't exist in Jack without `()`s, so this approach can't break it. This is why `()`s only appear as part of a `<term>`.)

You now have *almost* everything you need for a full Jack-to-Hack compiler. The only pieces missing are how to generate code that deals with `class`-type variables and methods, which we'll cover next video!