Why care about assemblers?

Because ...

- Assemblers employ some nifty tricks
- Assemblers are the first rung up the software hierarchy ladder
- An assembler is a translator of a simple language
- Writing an assembler is a good introduction for writing compilers.
Program translation

Source code

```
// Computes sum=1+...+100
00  i=1
01  sum=0
LOOP:
02  IF i=101 GOTO END
03  sum=sum+i
04  i=i+1
05  GOTO LOOP
END:
06  GOTO END
```

Target code

```
00  10001010110011010100011110111011101
01  00011110110111101000111011001100
02  00101101110011001110110111011001
03  10001101100110110111011111001101
04  00011101101100111011011101001100
05  10011011101101110111011101110110
06  000011101111110111001111000100
07  10001101100110110111011111001101
08  00011101101100111011011101001100
09  10011011101101110111011101110110
11  10001101100110110111011111001101
```

The program translation challenge

- Parse the source program, using the syntax rules of the source language
- Re-express the program's semantics using the syntax rules of the target language

Assembler = simple translator

- Translates each assembly command into one or more machine instructions
- Handles symbols (\( i \), \( \text{sum} \), \( \text{LOOP} \), \( \text{END} \), ...).

Symbol resolution

In low level languages, symbols are used to code:

- Variable names
- Destinations of goto commands (labels)
- Special memory locations

Code with Symbols

```
// Computes sum=1+...+100
00  i=1
01  sum=0
LOOP:
02  IF i=101 GOTO END
03  sum=sum+i
04  i=i+1
05  GOTO LOOP
END:
06  GOTO END
```

Symbol table

- \( i \): 1014
- \( \text{sum} \): 1025
- \( \text{LOOP} \): 2
- \( \text{END} \): 6

(assuming that variables are allocated to Memory[1024] onward)

Code with Symbols Resolved

```
00  M[1024]=1 // (in memory)
02  M[1025]=0
03  IF M[1024]=101 goto 6
05  goto 2
06  goto 6
```

(assuming that each symbolic command is translated into one word in memory)

The assembly process:

- First pass: construct a symbol table
- Second pass: translate the program, using the symbol table for symbols resolution.
This example is based on some simplifying assumptions:

- Largest possible program is 1024 commands long
- Each command fits into one memory location
- Each variable fits into one memory location

Every one of these assumptions can be relaxed rather easily.
Handling A-instructions

Symbolic: 0\text{value}  // Where value is either a non-negative decimal number  
// or a symbol referring to such number.

\text{value} (v = 0 \text{ or } 1)

<table>
<thead>
<tr>
<th>Binary:</th>
<th>0</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
</tr>
</thead>
</table>

Translation to binary:

- If \text{value} is a number: simple
- If \text{value} is a symbol: later.

Handling C-instruction

Symbolic: dest=comp;jump  // Either the dest or jump fields may be empty.
// If dest is empty, the "=" is omitted;
// If jump is empty, the ";" is omitted.

| Binary: | 1 | 1 | a | c1 | c2 | c3 | c4 | c5 | c6 | d1 | d2 | d3 | j1 | j2 | j3 |
|---------|---|---|---|----|----|----|----|----|----|----|----|----|----|----|

Translation to binary: simple!
The overall assembly logic

Assembly program (Prog.asm)

```
// Adds 1 + ... + 100
@i      // i=1
M=1     // sum=0
@sum    // sum=0
LOOP:
    @i
    M=M+1  // i=i+1
    @sum
    M=M+D   // sum=sum+i
    @i
    M=M+1  // i=i+1
    @LOOP
    0;JMP  // goto LOOP
@end:
    @END
    0;JMP  // infinite loop
```

For each (real) command

- Parse the command, i.e. break it into its constituent fields
- Replace each symbolic reference (if any) with the corresponding memory address (a binary number)
- For each field, generate the corresponding binary code
- Assemble the binary codes into a complete machine instruction.

Symbols handling (in the Hack language)

Program example

```
// Adds 1 + ... + 100
@i      // i=1
M=1     // sum=0
@sum    // sum=0
LOOP:
    @i
    D=M    // D=i
    @100
    D=D-A  // D=i-100
    @END
    D;JGT // if (i-100)>0 goto END
    @i
    D=M    // D=i
    @sum
    M=M+D   // sum=sum+i
    @END
    @END
    0;JMP  // infinite loop
```

- **Predefined symbols:** (don't appear in this example)
  - Label | RAM address
  - SP     | 0
  - LCL    | 1
  - ARG    | 2
  - THIS   | 3
  - THAT   | 4
  - @i-5   | 5
  - @i-15  | 16
  - END    | 2476

- **Label symbols:** The pseudo-command "(xxx)" declares that the user-defined symbol xxx should refer to the memory location holding the next command in the program

- **Variable symbols:** Any symbol xxx appearing in an assembly program that is not predefined and is not defined elsewhere using the "(xxx)" pseudo command is treated as a variable

  Variables are mapped to consecutive memory locations starting at RAM address 16.
### Example

<table>
<thead>
<tr>
<th>Assembly code (Prog.asm)</th>
<th>Binary code (Prog.hack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@i&lt;br&gt;M=1 // i=1&lt;br&gt;@sum&lt;br&gt;M=0 // sum=0&lt;br&gt;(LOOP)&lt;br&gt;@i&lt;br&gt;D=M // D=i&lt;br&gt;@100&lt;br&gt;D=D-A // D=i-100&lt;br&gt;@END&lt;br&gt;D;JGT // if (i-100)&gt;0 goto END&lt;br&gt;@i&lt;br&gt;D=M // D=i&lt;br&gt;@sum&lt;br&gt;M=D+M // sum=sum+i&lt;br&gt;@i&lt;br&gt;M=M+1 // i=i+1&lt;br&gt;(LOOP)&lt;br&gt;0;JMP // goto LOOP&lt;br&gt;(END)&lt;br&gt;@END&lt;br&gt;0;JMP // infinite loop</td>
<td>(this line should be erased)&lt;br&gt;0000 0000 0001 0000&lt;br&gt;1110 1111 1100 1000&lt;br&gt;0000 0000 0001 0001&lt;br&gt;1110 1010 1000 1000&lt;br&gt;(this line should be erased)&lt;br&gt;0000 0000 0001 0000&lt;br&gt;1111 1100 0001 0000&lt;br&gt;0000 0000 0110 0100&lt;br&gt;1110 0100 1101 0000&lt;br&gt;0000 0000 0001 0010&lt;br&gt;1110 0011 0000 0001&lt;br&gt;0000 0000 0001 0000&lt;br&gt;1111 1100 0001 0000&lt;br&gt;0000 0000 0001 0001&lt;br&gt;1111 0000 1000 1000&lt;br&gt;0000 0000 0000 0001&lt;br&gt;1111 1011 1100 1000&lt;br&gt;0000 0000 0000 0000&lt;br&gt;1110 0101 1000 0111&lt;br&gt;0000 0000 0001 0010&lt;br&gt;1110 1010 1000 0111&lt;br&gt;1111 1010 1000 0111</td>
</tr>
</tbody>
</table>

### Proposed implementation

An assembler program can be implemented (in any language) as follows.

**Software modules:**

- **Parser:** Unpacks each command into its underlying fields
- **Code:** Translates each field into its corresponding binary value
- **SymbolTable:** Manages the symbol table
- **Main:** Initializes I/O files and drives the show.

**Proposed implementation steps**

- **Stage I:** Build a basic assembler for programs with no symbols
- **Stage II:** Extend the basic assembler with symbol handling capabilities.
## Parser module

**Parser**: Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor / initializer</td>
<td>Input file / stream</td>
<td>--</td>
<td>Opens the input file/stream and gets ready to parse it</td>
</tr>
<tr>
<td>hasMoreCommands</td>
<td>--</td>
<td>Boolean</td>
<td>Are there more commands in the input?</td>
</tr>
<tr>
<td>advance</td>
<td>--</td>
<td>--</td>
<td>Reads the next command from the input and makes it the current command. Should be called only if hasMoreCommands() is true. Initially there is no current command.</td>
</tr>
<tr>
<td>commandType</td>
<td>--</td>
<td>A_COMMAND, C_COMMAND, L_COMMAND</td>
<td>Returns the type of the current command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A_COMMAND for (Xxx where Xxx is either a symbol or a decimal number)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C_COMMAND for dest=comp; jump</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L_COMMAND (actually, pseudo-command) for (Xxx) where Xxx is a symbol</td>
</tr>
</tbody>
</table>

## Parser module (cont.)

<table>
<thead>
<tr>
<th>symbol</th>
<th>--</th>
<th>string</th>
<th>Returns the symbol or decimal Xxx of the current command $Xxx or (Xxx). Should be called only when commandType() is A_COMMAND or L_COMMAND.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>--</td>
<td>string</td>
<td>Returns the dest mnemonic in the current C-command (2 possibilities). Should be called only when commandType() is C_COMMAND.</td>
</tr>
<tr>
<td>comp</td>
<td>--</td>
<td>string</td>
<td>Returns the comp mnemonic in the current C-command (28 possibilities). Should be called only when commandType() is C_COMMAND.</td>
</tr>
<tr>
<td>jump</td>
<td>--</td>
<td>string</td>
<td>Returns the jump mnemonic in the current C-command (8 possibilities). Should be called only when commandType() is C_COMMAND.</td>
</tr>
</tbody>
</table>
Building the final assembler

- **Initialization**: create the symbol table and initialize it with the pre-defined symbols.

- **First pass**: march through the program without generating any code. For each label declaration "(Xxx)", add the pair <Xxx, n> to the symbol table.

- **Second pass**: march again through the program, and translate each line:
  - If the line is a C-instruction, simple.
  - If the line is "@Xxx" where Xxx is a number, simple.
  - If the line is "@Xxx" where Xxx is a symbol, look it up in the symbol table:
    - If the symbol is found, replace it with its numeric meaning and complete the command’s translation.
    - If the symbol is not found, then it must represent a new variable: add the pair <Xxx, n> to the symbol table, where n is the next available RAM address, and complete the command’s translation.

(The allocated RAM addresses are running, starting at address 16.)
Symbol Table

Symbol Table: A symbol table that keeps a correspondence between symbolic labels and numeric addresses.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>--</td>
<td>--</td>
<td>Creates a new empty symbol table.</td>
</tr>
<tr>
<td>addEntry</td>
<td>symbol(string),</td>
<td>--</td>
<td>Adds the pair (symbol, address) to the</td>
</tr>
<tr>
<td></td>
<td>address(nt)</td>
<td></td>
<td>table</td>
</tr>
<tr>
<td>contains</td>
<td>symbol(string)</td>
<td>boolean</td>
<td>Does the symbol table contain the given</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>symbol?</td>
</tr>
<tr>
<td>GetAddress</td>
<td>symbol(string)</td>
<td>int</td>
<td>Returns the address associated with the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>symbol</td>
</tr>
</tbody>
</table>

Perspective

- Simple machine language, simple assembler
- Most assemblers are not stand-alone, but rather encapsulated in a translator of a higher order
- Low-level programming (e.g. a C-based real-time system) may involve some assembly programming (e.g. for optimization)
- Macro assemblers:

```plaintext
// Computes sum=1...100
00  i=1
01  sum=0
02  LOOP: IF i<=100 GOTO END
03  sum=sum+i
04  i=i+1
05  GOTO LOOP
06  END:
07  GOTO END
```
Endnote I: Turing machine (1935)

Informal description:

- A tape, divided into cells, each containing a symbol
- A head that can move over the tape left and right and read and write symbols
- A state register that stores the machine's state
- An action table (transition function):
  If the current state is S, and the current symbol is s, then move the tape n positions right/left, write a symbol s', and enter state S'.
- Key conjecture: for any program, in any language, running on any computer, there exists an equivalent TM that achieves the same results (proof?).

The Halting Problem

- Program = data: a TM program can be written on the tape of another TM, becoming its input
- The halting detection program:
  A program H that, for any given program p, prints 1 if p halts on any input, and 0 otherwise
- The halting theorem: H does not exist
- Theoretical significance: If H existed, it would simple to prove/disprove many propositions automatically. Example:

```c
// Goldbach conjecture: every even number greater than 2 is the sum of two primes.
function goldbach() {
    i = 4
    while true {
        if i = sum of two primes {
            i = i + 2
        } else {
            print("the conjecture is false. Counter example: ",i)
            return
        }
    }
}
```

If H existed, we could apply it to the goldbach() function, thus proving or disproving the Goldbach conjecture.
Historical perspective

- **Hilbert's challenge** (1928): Can we devise a mechanical procedure (algorithm) which could, in principle, prove or disprove any given mathematical proposition?

- **Alan Turing** (1935): NO. 
  Proof: uncomputability of the halting problem

- **Kurt Godel** (1931): NO. 
  Proof: Incompleteness theorem (any system containing the arithmetic of natural numbers is either incomplete or inconsistent)

- Philosophical implications.

---

Endnote II: The Enigma

- **Recommended reading:** 