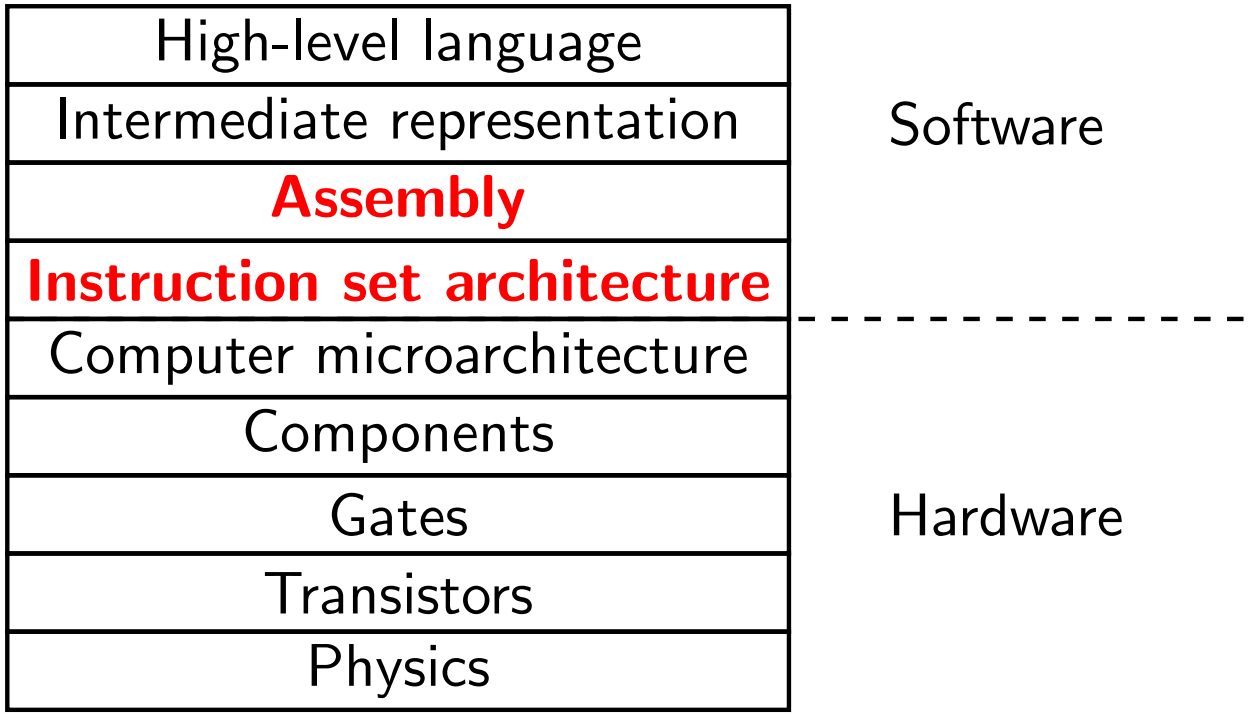
Compiler concepts: Lexing
COMSM1302 Overview of Computer Architecture

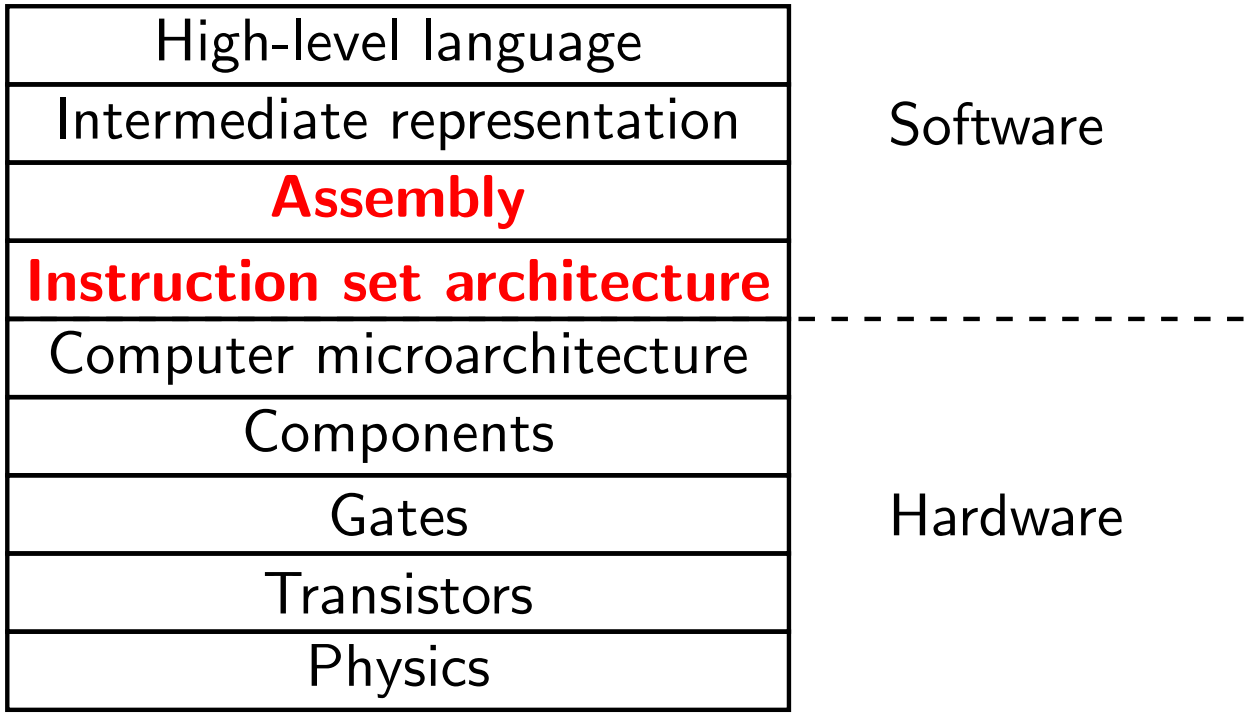
John Lapinskas, University of Bristol

Our next goal



We now understand both Hack assembly and the Hack ISA, and we can use an assembler to turn assembly into machine code.

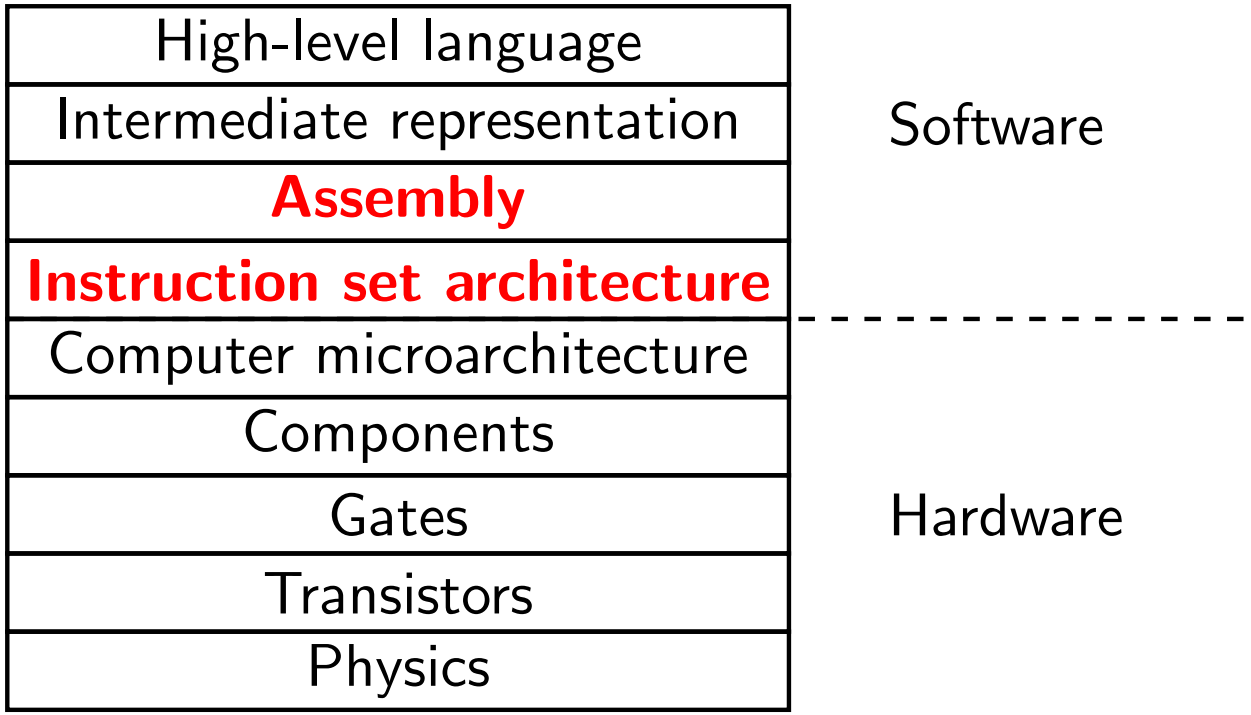
Our next goal



We now understand both Hack assembly and the Hack ISA, and we can use an assembler to turn assembly into machine code.

But how does the assembler work?

Our next goal



We now understand both Hack assembly and the Hack ISA, and we can use an assembler to turn assembly into machine code.

But how does the assembler work?

This week, our goal is to create a Hack assembler of our own in C!

Compilers: A big-picture view

Modern compilers usually work in discrete **phases**:

Preprocessing
Lexing
Parsing/syntax analysis
Semantic analysis
Intermediate representation
Optimisation
Code generation
More optimisation

Compilers: A big-picture view

Modern compilers usually work in discrete **phases**:

Preprocessing
Lexing
Parsing/syntax analysis
Semantic analysis
Intermediate representation
Optimisation
Code generation
More optimisation

Preprocessing is a set of language-specific steps to e.g. link code files together and execute compiler macros. In C, this includes reading the makefile and processing e.g. `#includes` and `#defines`. **Optimising** is the process of automatically turning inefficient code into efficient code.

Compilers: A big-picture view

Modern compilers usually work in discrete **phases**:

~~Preprocessing~~
Lexing
Parsing/syntax analysis
Semantic analysis
Intermediate representation
~~Optimisation~~
Code generation
~~More optimisation~~

Preprocessing is a set of language-specific steps to e.g. link code files together and execute compiler macros. In C, this includes reading the makefile and processing e.g. `#includes` and `#defines`. **Optimising** is the process of automatically turning inefficient code into efficient code.

Both preprocessing and optimisation are beyond the scope of this unit.

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
Semantic analysis
Intermediate representation
Code generation

Most compilers take the information they extracted from semantic analysis and put it into an **intermediate representation** (**IR**) of the code. They then optimise the IR rather than the original code.

IRs are intended for use by computers rather than humans, and are quite close to assembly. Optimising the IR is "the hard part" of optimisation.

Many languages can compile to a single IR — e.g. LLVM is used in compilers for C#, Java bytecode, Ruby and Rust.

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
Semantic analysis
~~Intermediate representation~~
Code generation

Most compilers take the information they extracted from semantic analysis and put it into an **intermediate representation** (**IR**) of the code. They then optimise the IR rather than the original code.

IRs are intended for use by computers rather than humans, and are quite close to assembly. Optimising the IR is "the hard part" of optimisation.

Many languages can compile to a single IR — e.g. LLVM is used in compilers for C#, Java bytecode, Ruby and Rust.

We'll see an IR for Hack starting next week, but a line of assembly already maps to one line of machine code, so for an assembler there's no point!

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
Semantic analysis
Code generation

Syntax refers to the logical structure of language or code. For example:

"The trophy wouldn't fit in the case because it was too big."

A syntax analysis would tell you that "trophy" and "case" are nouns, that the sentence breaks into two clauses, and generally anything you wanted to know about the sentence's grammar.

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
Semantic analysis
Code generation

Syntax refers to the logical structure of language or code. For example:

```
strcpy(x,"Test");
```

A syntax analysis would tell you that this is a call to a function called "strcpy" whose first argument is a variable called "x" and whose second argument is the string "Test".

Syntax analysis can get very complicated, but it's a very well-understood problem and there are many standard algorithms and approaches.

The process of syntax analysis is called **parsing**.

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
Semantic analysis
Code generation

Semantics refers to the wider meaning of language or code. For example:

"The trophy wouldn't fit in the case because it was too big."

A semantic analysis tells you that "it" refers to the trophy, not the case. The sentence is valid English either way, but there's only one sane interpretation.

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
Semantic analysis
Code generation

Semantics refers to the meaning of language or code. For example:

```
strcpy(x,"Test");
```

A semantic analysis tells you that `strcpy` isn't defined (because the programmer forgot to `#include <string.h>`) and that `x` is a `char *` variable, but that there's a bug which makes it point to unallocated memory if the system time is exactly midnight.

Semantic analysis is really hard! Some aspects (like type-checking) are well-understood, but automatic bug-checking ("verification") is a research area.

Compilers: A big-picture view

Lexing
Parsing/syntax analysis
~~Semantic analysis~~
Code generation

For an assembler, though, syntax and semantics are almost identical, and it's common to fold semantic analysis into lexing and parsing.

Compilers: A big-picture view

Lexing
Parsing and code generation
~~Semantic analysis~~
~~Code generation~~

For an assembler, though, syntax and semantics are almost identical, and it's common to fold semantic analysis into lexing and parsing.

And without complicated semantics, there's no need to separate code generation from parsing — we can just go line-by-line.

Compilers: A big-picture view

Lexing
Parsing and code generation
~~Semantic analysis~~
~~Code generation~~

For an assembler, though, syntax and semantics are almost identical, and it's common to fold semantic analysis into lexing and parsing.

And without complicated semantics, there's no need to separate code generation from parsing — we can just go line-by-line.

Our Hack assembler will have just two steps: lexing and parsing.

Even this will be overkill for Hack assembly! But we'll do it in detail to establish the principles involved.

The Joy of Lex

A **token** (a.k.a. **lexeme** or **lexical element**) is an "indivisible" piece of code that can't be broken down further without losing its meaning.

In English, the closest analogy is to words and punctuation:

Never | gonna | give | you | up | , | never | gonna | let | you | down | .

The Joy of Lex

A **token** (a.k.a. **lexeme** or **lexical element**) is an "indivisible" piece of code that can't be broken down further without losing its meaning.

In English, the closest analogy is to words and punctuation:

$$\text{Never} \mid \text{gonna} \mid \text{give} \mid \text{you} \mid \text{up} \mid , \mid \text{never} \mid \text{gonna} \mid \text{let} \mid \text{you} \mid \text{down} \mid .$$

Lexing is the process of converting a string of code into a list of tokens. Normally this just involves removing whitespace and comparing strings.

The Joy of Lex

A **token** (a.k.a. **lexeme** or **lexical element**) is an "indivisible" piece of code that can't be broken down further without losing its meaning.

In English, the closest analogy is to words and punctuation:

$$\text{Never} \mid \text{gonna} \mid \text{give} \mid \text{you} \mid \text{up} \mid , \mid \text{never} \mid \text{gonna} \mid \text{let} \mid \text{you} \mid \text{down} \mid .$$

Lexing is the process of converting a string of code into a list of tokens. Normally this just involves removing whitespace and comparing strings.

The list of possible tokens is given in the **specification** of a programming language, along with the **grammar** for putting them together to form statements (see later this week). C's specification is available here.

What counts as a token is a *choice* by the language creators, not a universal standard! E.g. in Java comments and whitespace are tokens, but in C they're not.

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

- **Keywords**:
  - 'A', 'D', 'M',
  - 'JGT', 'JEQ', 'JLT', 'JGE', 'JNE', 'JLE', 'JMP',
  - 'SCREEN', 'KBD', 'SP', 'LCL', 'ARG', 'THIS', 'THAT',
  - and 'R0' through 'R15'.

Notice e.g. 'A' and 'ARG' are different keywords — we need to be a little careful distinguishing between them. In general, a **keyword** is a string with special meaning intrinsic to the language that usually can't be redefined. C's keywords include e.g. `const`, `int`, `for`, and `return`, but not e.g. `printf`.

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

- **Keywords**:
  - 'A', 'D', 'M',
  - 'JGT', 'JEQ', 'JLT', 'JGE', 'JNE', 'JLE', 'JMP',
  - 'SCREEN', 'KBD', 'SP', 'LCL', 'ARG', 'THIS', 'THAT',
  - and 'R0' through 'R15'.
- **Symbols**: '@', '+', '-', '&', '|', '=', ';', and '!'.

C calls these **punctuators** and **operators**, and includes several with multiple symbols like '>=', '==', '%:' and '%:%:'. (The latter two are synonyms of '#' and '##' for historical reasons.)

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

- **Keywords**:
  - 'A', 'D', 'M',
  - 'JGT', 'JEQ', 'JLT', 'JGE', 'JNE', 'JLE', 'JMP',
  - 'SCREEN', 'KBD', 'SP', 'LCL', 'ARG', 'THIS', 'THAT',
  - and 'R0' through 'R15'.
- **Symbols**: '@', '+', '-', '&', '|', '=', ';', and '!'.
- **Integer literals**: Any base-10 integer in the range $0 \ldots 32767$.

C calls these **constants**, and also has tokens for constant strings, floats, enums and chars.

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

- **Keywords**:
  - `'A'`, `'D'`, `'M'`,
  - `'JGT'`, `'JEQ'`, `'JLT'`, `'JGE'`, `'JNE'`, `'JLE'`, `'JMP'`,
  - `'SCREEN'`, `'KBD'`, `'SP'`, `'LCL'`, `'ARG'`, `'THIS'`, `'THAT'`,
  - and `'R0'` through `'R15'`.
- **Symbols**: `'@'`, `'+'`, `'-'`, `'&'`, `'|'`, `'='`, `';'`, and `'!'`.
- **Integer literals**: Any base-10 integer in the range $0 \ldots 32767$.
- **Identifiers**: Any string containing no whitespace that's not a keyword and starts with a letter.

C has these too. Notice that we *don't* distinguish between variables and labels! In general, an **identifier** is a token representing an entity defined in the code rather than the language, e.g. a specific variable, function, or struct type. In C, `printf` is an identifier defined in the stdio library.

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

- **Keywords**:
  - `'A'`, `'D'`, `'M'`,
  - `'JGT'`, `'JEQ'`, `'JLT'`, `'JGE'`, `'JNE'`, `'JLE'`, `'JMP'`,
  - `'SCREEN'`, `'KBD'`, `'SP'`, `'LCL'`, `'ARG'`, `'THIS'`, `'THAT'`,
  - and `'R0'` through `'R15'`.
- **Symbols**: `'@'`, `'+'`, `'-'`, `'&'`, `'|'`, `'='`, `';'`, and `'!'`.
- **Integer literals**: Any base-10 integer in the range $0 \ldots 32767$.
- **Identifiers**: Any string containing no whitespace that's not a keyword and starts with a letter.
- **Newlines**.

C ignores newlines completely, but we need them to be tokens — otherwise we can't distinguish e.g. `A` followed by `D=M` from `AD=M`.

Tokens in Hack assembly

Nisan and Schocken don't give a grammar for Hack assembly, but we can create one. We take as tokens:

- **Keywords**:
  - 'A', 'D', 'M',
  - 'JGT', 'JEQ', 'JLT', 'JGE', 'JNE', 'JLE', 'JMP',
  - 'SCREEN', 'KBD', 'SP', 'LCL', 'ARG', 'THIS', 'THAT',
  - and 'R0' through 'R15'.
- **Symbols**: '@', '+', '-', '&', '|', '=', ';', and '!'.
- **Integer literals**: Any base-10 integer in the range $0 \ldots 32767$.
- **Identifiers**: Any string containing no whitespace that's not a keyword and starts with a letter.
- **Newlines**.

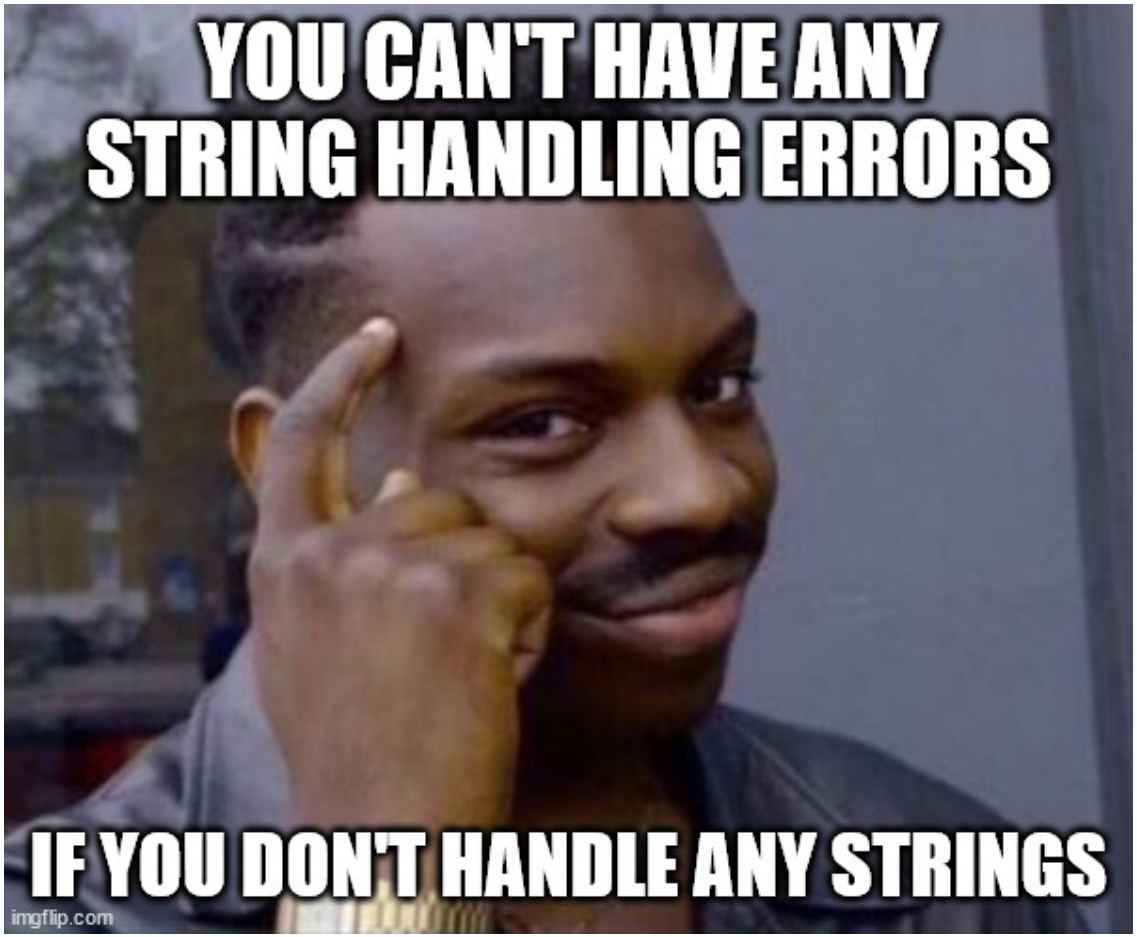
Notice what's missing! We throw whitespace and comments away, so they don't get tokens. We will also throw labels away without turning them into tokens (see next video), so we don't include '(' or ')' as symbols.

The benefits of lexing

Why bother with lexing?

The benefits of lexing

Why bother with lexing?



Source: Generated with imgflip (here).

We can store tokens as nice clean structs and never have to fight with C's awful string handling again! (At least until code generation...)

Example lexer output

Consider this Hack code from one of the test cases:

```
D;JGT                    // if D>0 goto output_first
```

Example lexer output

Consider this Hack code from one of the test cases:

```
D;JGT                    // if D>0 goto output_first
```

Your lexer will convert this to four `Token` structs:

Token 1: `(type=Keyword, value=D)`
Token 2: `(type=Symbol, value=;)`
Token 3: `(type=Keyword, value=JGT)`
Token 4: `(type=Newline, value=None)`

in which D and JGT are stored as enums and ';' is stored as a char, then call a `write_token` function to store those tokens in a temporary file.

Your parser will then process that file instead of the original code, using a `read_token` function to read `Token` structs rather than strings.

Example lexer output

Consider this Hack code from one of the test cases:

```
D;JGT                    // if D>0 goto output_first
```

Your lexer will convert this to four `Token` structs:

Token 1: `(type=Keyword, value=D)`
Token 2: `(type=Symbol, value=;)`
Token 3: `(type=Keyword, value=JGT)`
Token 4: `(type=Newline, value=None)`

in which D and JGT are stored as enums and ';' is stored as a char, then call a `write_token` function to store those tokens in a temporary file.

Your parser will then process that file instead of the original code, using a `read_token` function to read `Token` structs rather than strings.

Your lexer will also handle labels, which would normally be part of semantic analysis — see next video!