A parser for Hack assembly
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

Tokens for Hack assembly (reminder)

- **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 $0 \dots 32767$.
- **Identifiers**: Any string containing no whitespace that's not a keyword and starts with a letter.
- **Newlines**.

EBNF for Hack assembly

There are many equivalent ways to formalise Hack in EBNF. Here's one.

$$\langle\text{instruction}\rangle ::= (\langle\text{aInstruction}\rangle \mid \langle\text{cInstruction}\rangle),\ \text{newline};$$

$$\langle\text{aInstruction}\rangle ::= \text{``}\texttt{@}\text{''}, (\text{integerLiteral} \mid \text{identifier} \mid \langle\text{memoryKeyword}\rangle);$$

$$\langle\text{memoryKeyword}\rangle ::= \text{``}\texttt{SCREEN}\text{''} \mid \text{``}\texttt{KBD}\text{''} \mid \text{``}\texttt{SP}\text{''} \mid \text{``}\texttt{LCL}\text{''} \mid \text{``}\texttt{ARG}\text{''} \mid \text{``}\texttt{THIS}\text{''} \mid \text{``}\texttt{THAT}\text{''};$$

EBNF for Hack assembly

There are many equivalent ways to formalise Hack in EBNF. Here's one.

$$\begin{aligned}
\langle\text{instruction}\rangle &::= (\langle\text{aInstruction}\rangle \mid \langle\text{cInstruction}\rangle),\ \text{newline};\\
\langle\text{aInstruction}\rangle &::= \text{“}\texttt{@}\text{”}, (\text{integerLiteral} \mid \text{identifier} \mid \langle\text{memoryKeyword}\rangle);\\
\langle\text{memoryKeyword}\rangle &::= \text{“}\texttt{SCREEN}\text{”} \mid \text{“}\texttt{KBD}\text{”} \mid \text{“}\texttt{SP}\text{”} \mid \text{“}\texttt{LCL}\text{”} \mid \text{“}\texttt{ARG}\text{”} \mid \text{“}\texttt{THIS}\text{”} \mid \text{“}\texttt{THAT}\text{”};\\
\langle\text{cInstruction}\rangle &::= [\langle\text{assignment}\rangle], \langle\text{computation}\rangle, [\langle\text{jump}\rangle];
\end{aligned}$$

EBNF for Hack assembly

There are many equivalent ways to formalise Hack in EBNF. Here's one.

$$
\begin{aligned}
\langle\text{instruction}\rangle &::= (\langle\text{aInstruction}\rangle \mid \langle\text{cInstruction}\rangle),\ \text{newline};\\
\langle\text{aInstruction}\rangle &::= \text{“}\texttt{@}\text{”}, (\text{integerLiteral} \mid \text{identifier} \mid \langle\text{memoryKeyword}\rangle);\\
\langle\text{memoryKeyword}\rangle &::= \text{“}\texttt{SCREEN}\text{”} \mid \text{“}\texttt{KBD}\text{”} \mid \text{“}\texttt{SP}\text{”} \mid \text{“}\texttt{LCL}\text{”} \mid \text{“}\texttt{ARG}\text{”} \mid \text{“}\texttt{THIS}\text{”} \mid \text{“}\texttt{THAT}\text{”};\\
\langle\text{cInstruction}\rangle &::= [\langle\text{assignment}\rangle], \langle\text{computation}\rangle, [\langle\text{jump}\rangle];\\
\langle\text{assignment}\rangle &::= (\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{D}\text{”} \mid \text{“}\texttt{M}\text{”} \mid (\text{“}\texttt{A}\text{”}, \text{“}\texttt{D}\text{”}) \mid (\text{“}\texttt{A}\text{”}, \text{“}\texttt{M}\text{”}) \mid (\text{“}\texttt{D}\text{”}, \text{“}\texttt{M}\text{”}) \mid \\
&\qquad (\text{“}\texttt{A}\text{”}, \text{“}\texttt{D}\text{”}, \text{“}\texttt{M}\text{”})), \text{“}\texttt{=}\text{”};
\end{aligned}
$$

EBNF for Hack assembly

There are many equivalent ways to formalise Hack in EBNF. Here's one.

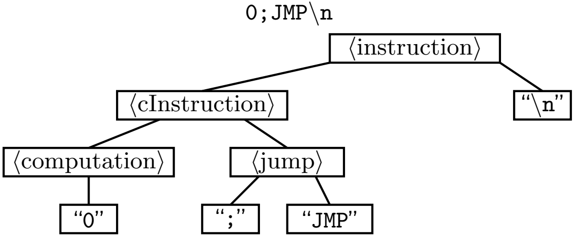
$$
\begin{aligned}
\langle\text{instruction}\rangle &::= (\langle\text{aInstruction}\rangle \mid \langle\text{cInstruction}\rangle),\ \text{newline};\\
\langle\text{aInstruction}\rangle &::= \text{“}\texttt{@}\text{”}, (\text{integerLiteral} \mid \text{identifier} \mid \langle\text{memoryKeyword}\rangle);\\
\langle\text{memoryKeyword}\rangle &::= \text{“}\texttt{SCREEN}\text{”} \mid \text{“}\texttt{KBD}\text{”} \mid \text{“}\texttt{SP}\text{”} \mid \text{“}\texttt{LCL}\text{”} \mid \text{“}\texttt{ARG}\text{”} \mid \text{“}\texttt{THIS}\text{”} \mid \text{“}\texttt{THAT}\text{”};\\
\langle\text{cInstruction}\rangle &::= [\langle\text{assignment}\rangle], \langle\text{computation}\rangle, [\langle\text{jump}\rangle];\\
\langle\text{assignment}\rangle &::= (\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{D}\text{”} \mid \text{“}\texttt{M}\text{”} \mid (\text{“}\texttt{A}\text{”}, \text{“}\texttt{D}\text{”}) \mid (\text{“}\texttt{A}\text{”}, \text{“}\texttt{M}\text{”}) \mid (\text{“}\texttt{D}\text{”}, \text{“}\texttt{M}\text{”}) \mid\\
&\qquad (\text{“}\texttt{A}\text{”}, \text{“}\texttt{D}\text{”}, \text{“}\texttt{M}\text{”})), \text{“}\texttt{=}\text{”};\\
\langle\text{jump}\rangle &::= \text{“}\texttt{;}\text{”}, (\text{“}\texttt{JMP}\text{”} \mid \text{“}\texttt{JGT}\text{”} \mid \text{“}\texttt{JEQ}\text{”} \mid \text{“}\texttt{JLT}\text{”} \mid \text{“}\texttt{JGE}\text{”} \mid \text{“}\texttt{JNE}\text{”} \mid \text{“}\texttt{JLE}\text{”});
\end{aligned}
$$

EBNF for Hack assembly

There are many equivalent ways to formalise Hack in EBNF. Here's one.

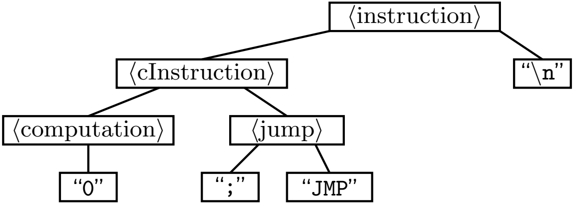
$$\begin{aligned}
\langle\text{instruction}\rangle &::= (\langle\text{aInstruction}\rangle \mid \langle\text{cInstruction}\rangle),\ \text{newline};\\
\langle\text{aInstruction}\rangle &::= \text{“}\texttt{@}\text{”}, (\text{integerLiteral} \mid \text{identifier} \mid \langle\text{memoryKeyword}\rangle);\\
\langle\text{memoryKeyword}\rangle &::= \text{“}\texttt{SCREEN}\text{”} \mid \text{“}\texttt{KBD}\text{”} \mid \text{“}\texttt{SP}\text{”} \mid \text{“}\texttt{LCL}\text{”} \mid \text{“}\texttt{ARG}\text{”} \mid \text{“}\texttt{THIS}\text{”} \mid \text{“}\texttt{THAT}\text{”};\\
\langle\text{cInstruction}\rangle &::= [\langle\text{assignment}\rangle], \langle\text{computation}\rangle, [\langle\text{jump}\rangle];\\
\langle\text{assignment}\rangle &::= (\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{D}\text{”} \mid \text{“}\texttt{M}\text{”} \mid (\text{“}\texttt{A}\text{”}, \text{“}\texttt{D}\text{”}) \mid (\text{“}\texttt{A}\text{”}, \text{“}\texttt{M}\text{”}) \mid (\text{“}\texttt{D}\text{”}, \text{“}\texttt{M}\text{”}) \mid\\
&\qquad (\text{“}\texttt{A}\text{”}, \text{“}\texttt{D}\text{”}, \text{“}\texttt{M}\text{”})), \text{“}\texttt{=}\text{”};\\
\langle\text{jump}\rangle &::= \text{“}\texttt{;}\text{”}, (\text{“}\texttt{JMP}\text{”} \mid \text{“}\texttt{JGT}\text{”} \mid \text{“}\texttt{JEQ}\text{”} \mid \text{“}\texttt{JLT}\text{”} \mid \text{“}\texttt{JGE}\text{”} \mid \text{“}\texttt{JNE}\text{”} \mid \text{“}\texttt{JLE}\text{”});\\
\langle\text{computation}\rangle &::= \text{“}\texttt{0}\text{”} \mid\\
&\quad ([\text{“}\texttt{-}\text{”}], \text{“}\texttt{1}\text{”}) \mid\\
&\quad ([\text{“}\texttt{-}\text{”} \mid \text{“}\texttt{!}\text{”}], (\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{D}\text{”} \mid \text{“}\texttt{M}\text{”})) \mid\\
&\quad ((\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{D}\text{”} \mid \text{“}\texttt{M}\text{”}), (\text{“}\texttt{+}\text{”} \mid \text{“}\texttt{-}\text{”}), \text{“}\texttt{1}\text{”}) \mid\\
&\quad ((\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{M}\text{”}), \langle\text{binaryOp}\rangle, \text{“}\texttt{D}\text{”}) \mid\\
&\quad (\text{“}\texttt{D}\text{”}, \langle\text{binaryOp}\rangle, (\text{“}\texttt{A}\text{”} \mid \text{“}\texttt{M}\text{”}));\\
\langle\text{binaryOp}\rangle &::= \text{“}\texttt{+}\text{”} \mid \text{“}\texttt{-}\text{”} \mid \text{“}\texttt{\&}\text{”} \mid \text{“}\texttt{|}\text{”};
\end{aligned}$$

Example CSTs for Hack assembly

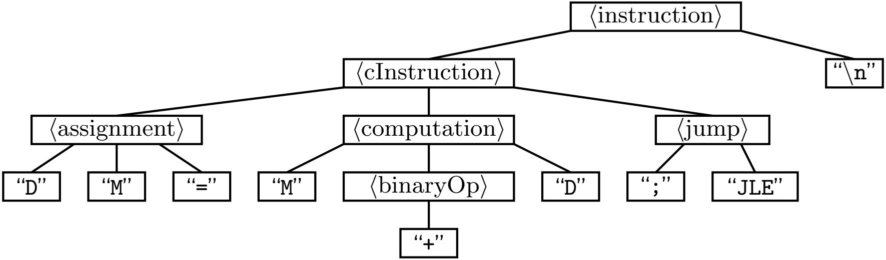


Example CSTs for Hack assembly

`0;JMP\n`



`DM=M+D;JLE\n`



How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

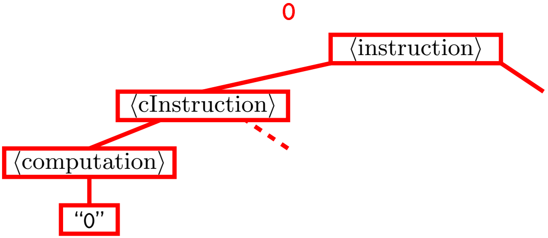
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

0

How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

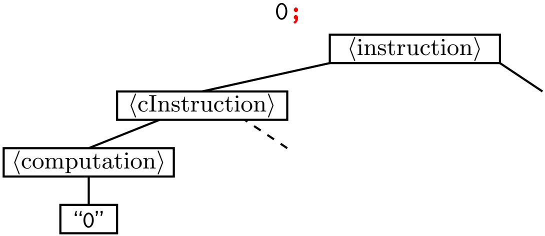


0 can arise only as an entire ⟨computation⟩, which must be in a ⟨cInstruction⟩, which must be in an ⟨instruction⟩.

We also know that this ⟨cInstruction⟩ has no ⟨assignment⟩, although it might have a ⟨jump⟩.

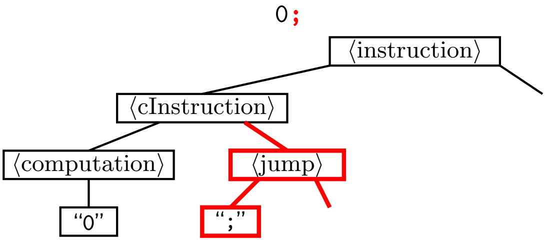
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

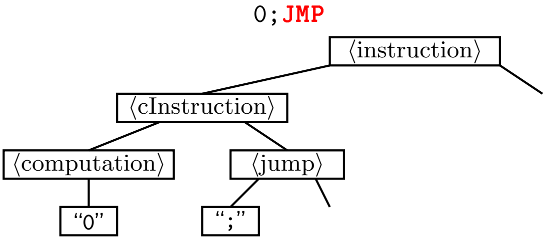
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



; can arise only as the first term of a ⟨jump⟩, which must be part of the ⟨cInstruction⟩ we already added.

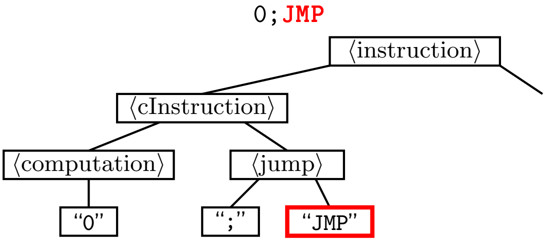
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

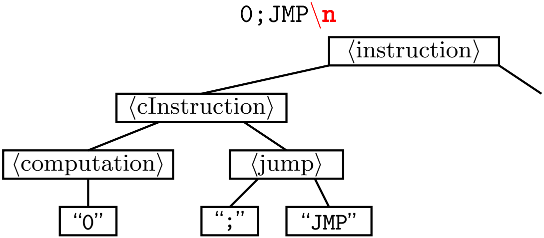
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



JMP must be the continuation of that ⟨jump⟩.

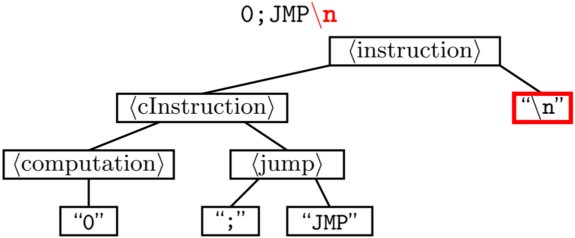
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

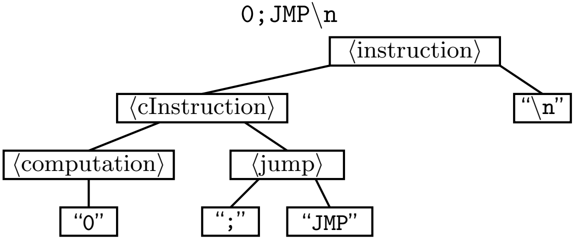
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



Finally, \n must be the end of the ⟨instruction⟩.

How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



In each case, if there had been no way to fit the next token into our existing CST (e.g. on parsing the 1 in 01;JMP), we would return an error.

How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

`D`

How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

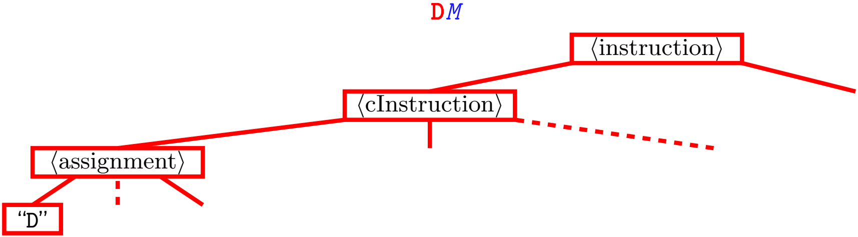
D*M*

We can't tell immediately how `D` should fit into the CST. It could be part of either an $\langle\text{assignment}\rangle$ or a $\langle\text{computation}\rangle$.

We can check which by looking ahead to the *second* token. If it's `M` or `=`, then we must be in an $\langle\text{assignment}\rangle$. If it's `+`, `-`, `&`, `|`, `;` or `\n` we must be in a $\langle\text{computation}\rangle$.

How do we build a Hack CST?

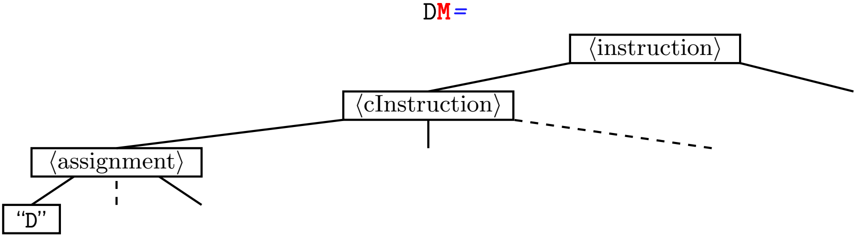
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



In this case, the next token is `M`. So `D` must be the start of an ⟨assignment⟩, which might also contain an `M` and will contain an `=`.

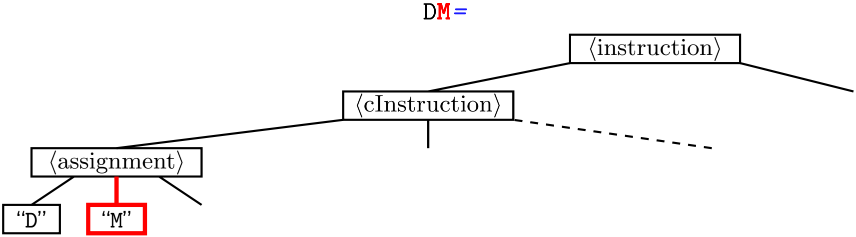
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

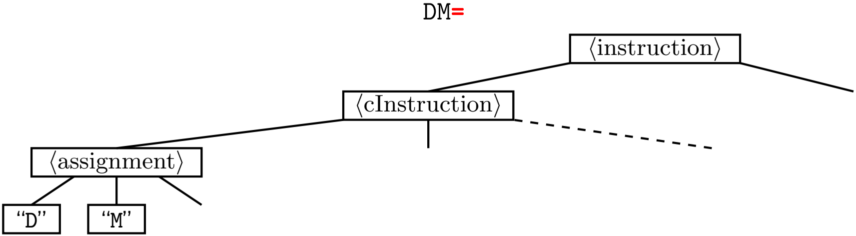


Again, looking ahead from `M` we see =, so we know is must be in an ⟨assignment⟩ as expected. (We could also use the fact that we know the next term must be part of an ⟨assignment⟩ to fit into the existing CST.)

The ⟨assignment⟩ must be part of a ⟨cInstruction⟩, which must be in an ⟨instruction⟩.

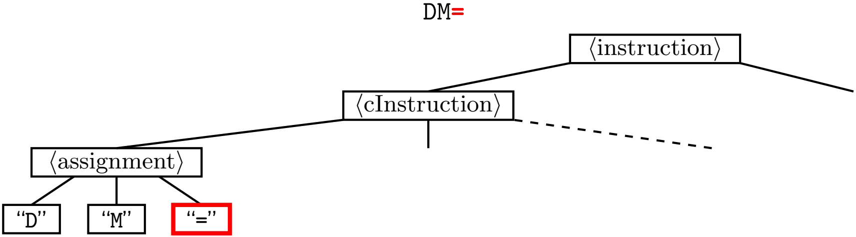
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

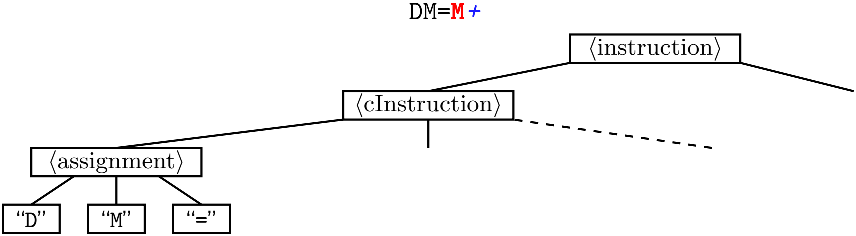
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



= can only fit into that ⟨assignment⟩.

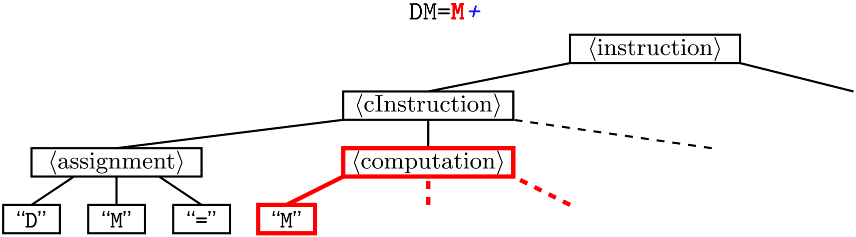
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

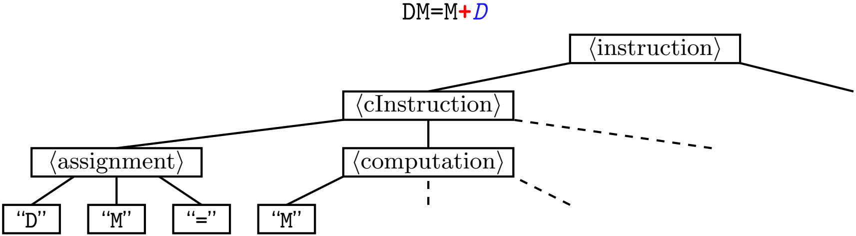


For this M, we again look ahead to see that the next character is +, so we must be in a ⟨computation⟩ as expected.

The ⟨computation⟩ may or may not contain a ⟨binaryOp⟩ and a second term.

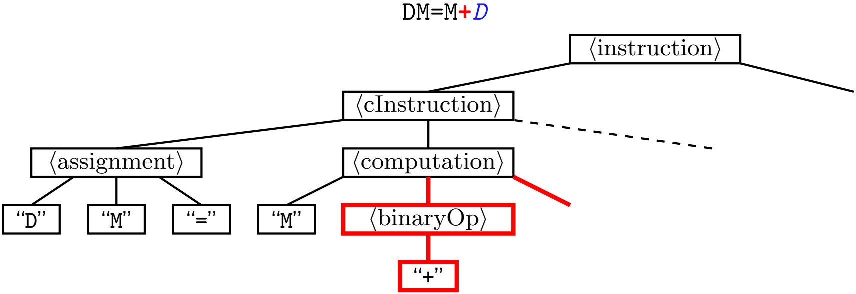
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.

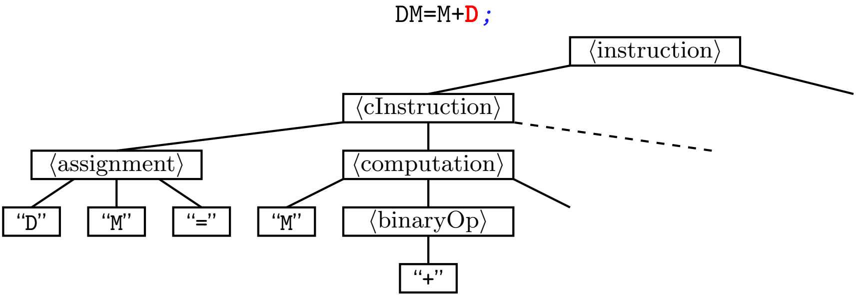


+ must be part of a ⟨computation⟩, but might be of the `M+1` form or of the `M+D` form (which leads to a different CST since the + is a ⟨binaryOp⟩).

Again, by looking ahead another token, we can tell that it is a ⟨binaryOp⟩. This also tells us we should expect the next term of the ⟨computation⟩ to be a `D`.

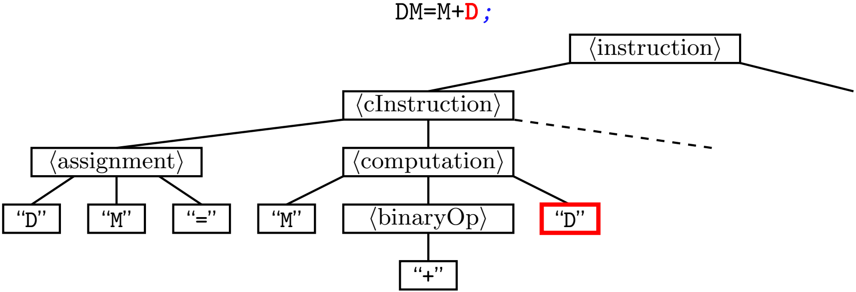
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

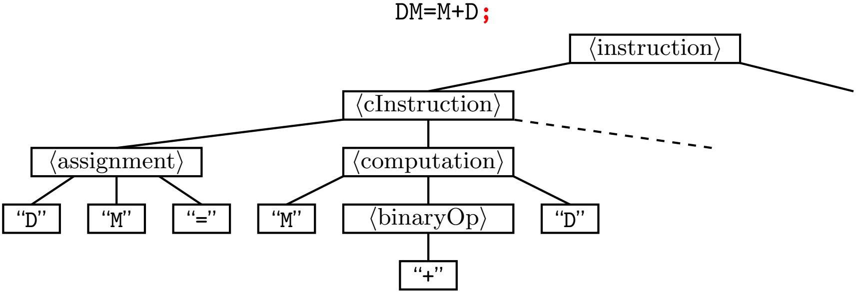
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



As expected, we see a `D`. Looking ahead we see a `;`, so this is part of a ⟨computation⟩ as expected.

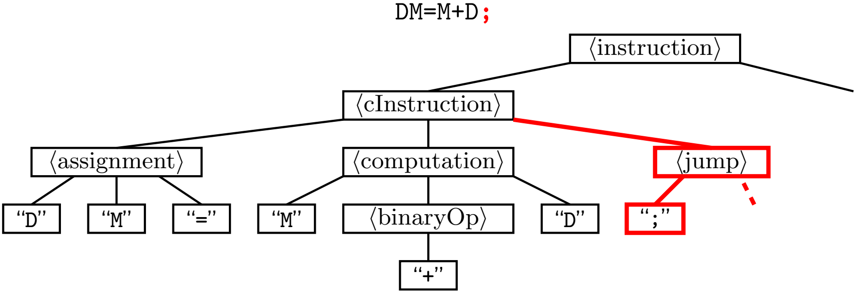
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

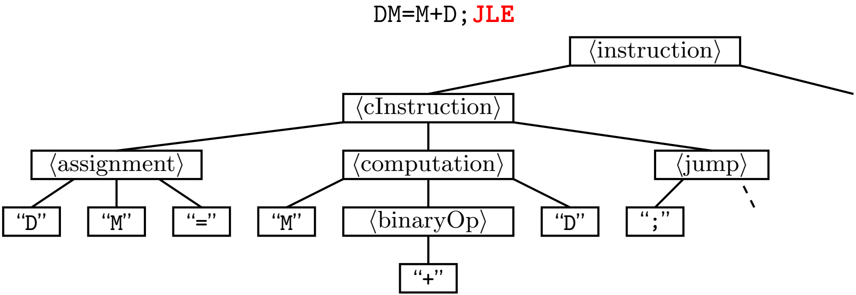
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



As in the previous example, ; must be part of a ⟨jump⟩...

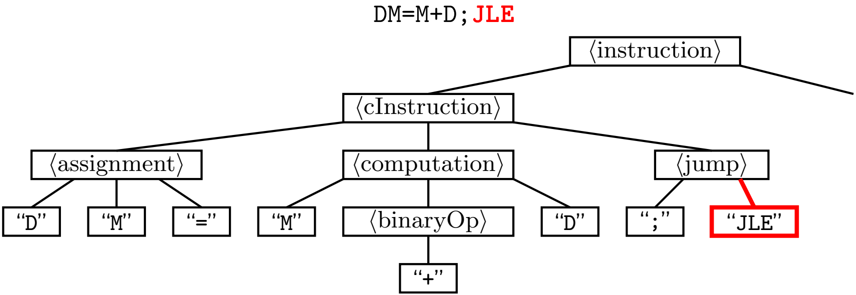
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

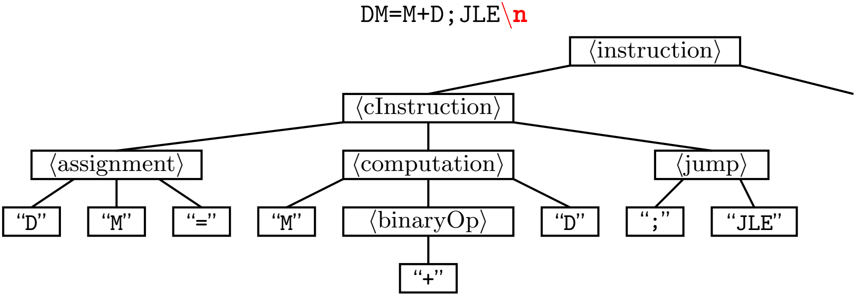
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



which is completed by this JLE...

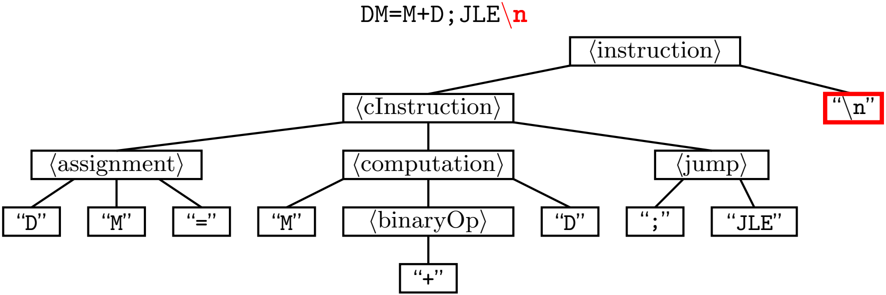
How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



How do we build a Hack CST?

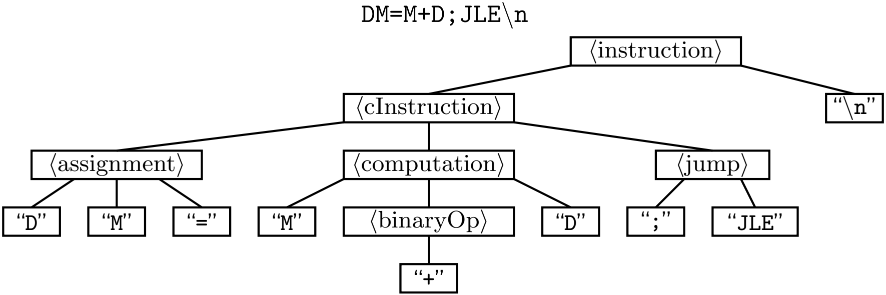
In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



leaving only the final newline.

How do we build a Hack CST?

In this unit, we will only consider **LL parsing**. We go through tokens from left to right, building the CST from the top down by looking at only the next few tokens.



Grammars which can be LL-parsed looking by looking k tokens ahead are called **LL(k)**. For example, Hack is LL(2) (because of e.g. `DM=M+D`) but not LL(1).

Warning: I'm sweeping some very deep rabbit-holes under the rug here. Not all grammars are LL(k), and this isn't even the formal definition of LL(k). But for a quick and dirty parser implementation it's good enough!

How do we actually parse Hack?

CSTs are extremely useful for graceful error-handling and for compiling complex expressions like `i=(k+j++)/m`. But here, they're overkill.

What we actually need to know:

- Is the statement an A-instruction or a C-instruction?
- If it's an A-instruction, what value should we load into A?
- If it's a C-instruction, what are the values of `dest`, `comp`, and `jump`?

How do we actually parse Hack?

CSTs are extremely useful for graceful error-handling and for compiling complex expressions like `i=(k+j++)/m`. But here, they're overkill.

What we actually need to know:

- Is the statement an A-instruction or a C-instruction?
  - Is the first token an `@`?
- If it's an A-instruction, what value should we load into A?
- If it's a C-instruction, what are the values of `dest`, `comp`, and `jump`?

How do we actually parse Hack?

CSTs are extremely useful for graceful error-handling and for compiling complex expressions like `i=(k+j++)/m`. But here, they're overkill.

What we actually need to know:

- Is the statement an A-instruction or a C-instruction?
    - Is the first token an `@`?
- If it's an A-instruction, what value should we load into A?
    - What comes after the `@`?
    - If it's an identifier, what RAM/ROM address is it? (Via symbol tables.)
- If it's a C-instruction, what are the values of `dest`, `comp`, and `jump`?

How do we actually parse Hack?

CSTs are extremely useful for graceful error-handling and for compiling complex expressions like `i=(k+j++)/m`. But here, they're overkill.

What we actually need to know:

- Is the statement an A-instruction or a C-instruction?
  - Is the first token an `@`?
- If it's an A-instruction, what value should we load into A?
  - What comes after the `@`?
  - If it's an identifier, what RAM/ROM address is it? (Via symbol tables.)
- If it's a C-instruction, what are the values of `dest`, `comp`, and `jump`?
  - Does = appear, and which of `A`, `D` and `M` appear to the left of it?
  - Does `;` appear, and which jump instruction appears to the right of it?
  - What's to the right of the = and the left of the `;`? (30-ish possibilities.)

How do we actually parse Hack?

CSTs are extremely useful for graceful error-handling and for compiling complex expressions like `i=(k+j++)/m`. But here, they're overkill.

What we actually need to know:

- Is the statement an A-instruction or a C-instruction?
    - Is the first token an `@`?
- If it's an A-instruction, what value should we load into A?
    - What comes after the `@`?
    - If it's an identifier, what RAM/ROM address is it? (Via symbol tables.)
- If it's a C-instruction, what are the values of `dest`, `comp`, and `jump`?
    - Does = appear, and which of `A`, `D` and `M` appear to the left of it?
    - Does `;` appear, and which jump instruction appears to the right of it?
    - What's to the right of the = and the left of the `;`? (30-ish possibilities.)

We can answer all these questions with simple logic on tokens.

(Making the CST is still a good exercise if you have extra time, though!)

The Hack assembler: A summary

Pass 1: Lexing. For each line:

- Remove any comments and whitespace.
- If the line is empty, skip it.
- If the line is a label, add it to the symbol table along with the ROM address corresponding to the current line.
- Otherwise, break the line into tokens and output to a temporary file.

Pass 2: Parsing. For each ⟨instruction⟩ (separated by newline tokens):

- If the ⟨instruction⟩ starts with an '`@`' token:
  - If it uses a new variable, allocate RAM and add it to the symbol table.
  - If it uses an existing variable or label, retrieve the RAM/ROM address for it from the symbol table.
  - Generate and output the corresponding A-instruction.
- Otherwise:
  - Break it down into an assignment, a computation, and a condition.
  - Map these to appropriate values of `dest`, `comp` and `jump` respectively.
  - Generate and output the corresponding C-instruction.