In programming languages, building parse trees is a significant task because parse trees tell us the meaning of computer programs. A typical CFG for a programming language like java could have about 50-100 nonterminal symbols and 30-40 terminals. The terminals of a programming language, called *tokens*, are things like boolean and arithmetic operators (e.g., `=`, `+`, `&`, `==`), parentheses (e.g., `(`, `)`), special symbols (e.g., `.`, `,`, `;`), special keywords (e.g., `if`, `return`, `int`, `class`), as well as placeholders for identifiers (`ID`, which stands for variable, procedure, and class names) and literals (e.g., `INT_LIT` which stands for an integer literal like `5`).

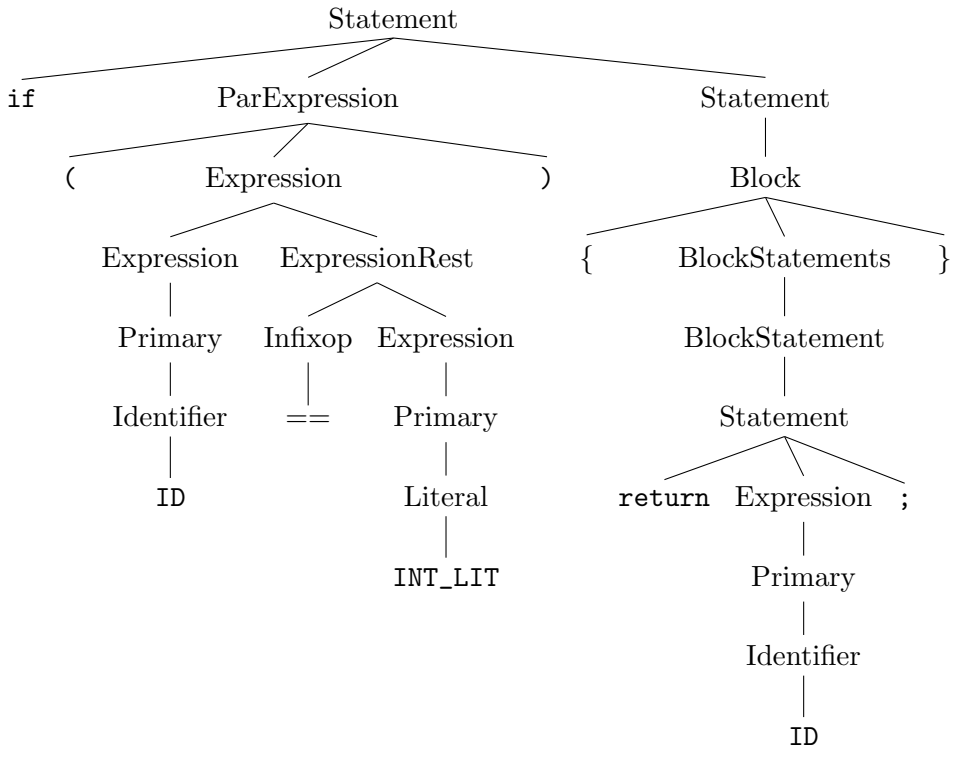
When the java compiler is invoked on a program, it first runs a *lexical analyzer* whose task is to identify the tokens. After running it, a line of code like

```
if ( n == 0 ) { return x; }
```

would be converted to

`if` `(` `ID` `==` `INT_LIT` `)` `{` `return` `ID` `;` `}`

Once we have the program in this form, the java parser creates a parse tree of the program. Here is the (simplified) tree that describes this line of code:



How does the java parser build this tree? One possibility is to use the Cocke-Younger-Kasami algorithm. However, this algorithm is too slow even for a moderate-size java program. The running time of the CYK algorithm is on the order of gn^3, where g is the number of symbols in the CFG and n is the length of the string to be parsed (i.e., the program). If the program has 1000 tokens – a reasonable size – the CYK parser could well take over a week to complete its task.

This is totally inadequate for computer programs, which need to be parsed much faster. However, the CYK parser is the fastest general-purpose parsing algorithm the we know of. Yet we know from experience that when we run the java compiler, our programs get parsed within a few milliseconds. How is this possible?

The answer is that the CFG of the java programming language (and other languages) are designed in a way that allows extremely quick parsing. The java CFG allows programs to be parsed in linear time. In fact, the parser only makes one left-to-right sweep of the input, at the end of which the parse tree is completely built.

The java CFG is a special type of CFG called LR(1).[1] LR(1) grammars do not cover all possible context-free languages, but they are rich enough to describe the semantics of most programming languages, while allowing for extremely fast parsing.

Before we describe how LR(1) grammars work, we start with an even simpler kind of grammar called LR(0). While LR(0) grammars are much less useful in practice, they will allow us to introduce the main ideas behind LR(1) parsing while avoiding some confusing complications.

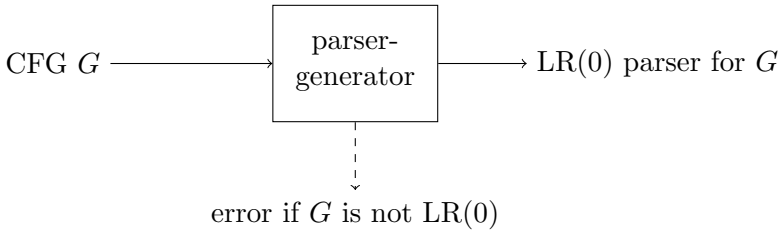
1 LR(0) parsing

At this point, you may expect a definition saying "An LR(0) grammar is [something]." However unlike in previous lectures, today we will do things a bit differently. Instead of giving a (syntactic) definition of LR(0) grammars, we will begin by describing a parsing algorithm. Unlike the CYK parsing algorithm, the algorithm we show today will not work for *every* CFG. The CFGs for which the algorithm happens to work correctly are the LR(0) grammars.

But given a grammar G, how do we know if the parsing algorithm will work or not? Unlike the CYK algorithm, which is a single (general-purpose) parsing algorithm which works for *any* CFG, LR(0) parsing algorithms are specifically tailored for the CFG we want to parse. What we will describe is a process that, given any CFG G, attempts to build a parsing algorithm tailored to G. At some point in the construction of this algorithm, we may observe that this algorithm has a certain kind of "bug"; if the bug does not occur, we will say that G is an LR(0) grammar.

This process, which takes a grammar G and outputs a fast algorithm for parsing strings in G (or outputs an error if the algorithm has a bug) is called a *parser generator*. The parser generator can itself be viewed as an algorithm: It is an algorithm which takes as input the description of a CFG G and produces as output the code of an algorithm for parsing strings in G.

[1] LR(1) stands for **L**eft-to-right read, **R**ightmost derivation, with **1** symbol of lookahead.



2 An example

The intuition behind the LR(0) algorithm is very natural. To explain it, suppose you have the following CFG G

$$A \to \mathtt{a}A\mathtt{b} \mid \mathtt{ab} \mid \mathtt{c}$$

and you want to parse the string `aabb`. How would you go about this task in a systematic way?

Imagine that you are a DFA. You are standing to the left of the input and you have not seen any of the input symbols yet, like this:

• a a b b

Now you see that the first symbol coming at you is an `a`. Looking at the productions of your CFG, you notice that there are two possibilities: The top of the parse tree must be either $A \to \mathtt{a}A\mathtt{b}$ or $A \to \mathtt{ab}$, but certainly not $A to \mathtt{c}$. You want to keep both of the first two possibilities in mind. So you consume the first `a`, move one spot to the right and find yourself in this situation:

a • a b b

You want to remember that at this stage, after you have seen the first symbol, the derivation you are looking at must either start with $A \to \mathtt{a}A\mathtt{b}$ or $A \to \mathtt{ab}$. You have already consumed the first A, so you must be in one of these two situations: $\boxed{A \to \mathtt{a} \bullet A\mathtt{b} \text{ or } A \to \mathtt{a} \bullet \mathtt{b}}$. To keep track of this, you set up a "registry" where you store both of these options as possibilities.

Before moving on, you notice that the first option on your registry says that what is coming next is a derivation starting from A. To prepare for this possible derivation, you add the following three items to your registry: $\boxed{A \to \bullet\mathtt{a}A\mathtt{b},\ A \to \bullet\mathtt{ab},\ A \to \bullet\mathtt{c}}$. At this point, the registry looks like this:

$A \to \mathtt{a} \bullet A\mathtt{b}$	$A \to \mathtt{a} \bullet \mathtt{b}$	
$A \to \bullet\mathtt{a}A\mathtt{b}$	$A \to \bullet\mathtt{ab}$	$A \to \bullet\mathtt{c}$

You now peek ahead again and notice that the next symbol is an `a`. This immediately rules out the second and fifth production from the registry. The first production is a bit tricky; the symbol after the • is non-terminal, so what should you do with it? Let's remember (in a separate place) that this is the state of the production after reading the first A, and take out this item from the registry too. What is left inside is $\boxed{A \to \bullet\mathtt{a}A\mathtt{b}, A \to \bullet\mathtt{ab}}$. After moving past the second `a`,

a a • b b

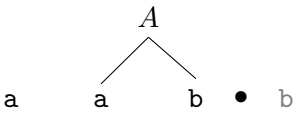
the registry is updated to $\boxed{A \to \texttt{a} \bullet A\texttt{b}, A \to \texttt{a} \bullet \texttt{b}}$. Since the first possibility again indicates that what may be coming is a derivation for A, we add the items $A \to \bullet\texttt{a}A\texttt{b}$, $A \to \bullet\texttt{ab}$, $A \to \bullet\texttt{c}$ and get the following registry:

$$\boxed{\begin{array}{lll} A \to \texttt{a} \bullet A\texttt{b} & A \to \texttt{a} \bullet \texttt{b} & \\ A \to \bullet\texttt{a}A\texttt{b} & A \to \bullet\texttt{ab} & A \to \bullet\texttt{c} \end{array}}$$

Peeking again, we notice the next input symbol is a `b`, meaning that the only viable item in our registry is $A \to \texttt{a} \bullet \texttt{b}$. After moving past this `b`, the registry is updated to a single item:

$$\boxed{A \to \texttt{ab}\bullet}$$

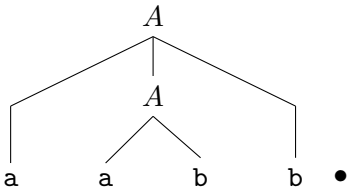
This means that we must have just completed part of the derivation, and we can construct the corresponding portion of the parse tree:



To continue from here, we must backtrack a bit. We remember that the sub-tree we just constructed came from the item $A \to \texttt{a} \bullet A\texttt{b}$ which we put aside after we read the second `a` in the input. So we must now be in the situation $A \to \texttt{a}A \bullet \texttt{b}$. So we clear the registry and put this item inside:

$$\boxed{A \to \texttt{a}A \bullet \texttt{b}}$$

Moving along, we notice the next (and last) input symbol is a `b`, which corresponds exactly to what is predicted by the only item in the registry. We read this last `b` and update the registry to $\boxed{A \to \texttt{ab}\bullet}$. The unique item in the registry now tells us how to complete the parse tree:



3 Parsing with the help of a DFA

Let us now go back to the example we just did and try to pin down the strategy that we used in producing the parse tree. The main idea was to keep track of all possible "items" that represent the current state of the parse tree.

Before we go on, let's introduce some notation. As usual, will capital letters like A, B, C to denote variables and lowercase letters like a, b, c for terminals. The letter X denotes a symbol that is either a variable or a terminal. We use greek letters α, β, γ for a string consisting of variables and terminals.

Definition 1. An *item* of a CFG G is a string of the form $A \to \alpha \bullet \beta$, where $A \to \alpha\beta$ is a production in G.

At each step of the process, we used a registry to keep track of all possible *valid items*, which we updated along the way based on what we saw in the input. Let us try to formalize the rules we used for updating items. Initially, the what we have in the registry are all items of the form $S \to \bullet\alpha$, where S is the start variable. We then updated the items according to the following rules:

```
1   If the registry does not contain any item of the form A → X1X2...Xk•:
2       Read the next input symbol a.
3       Replace every item of the form B → α • aβ with B → αa • β.
4       Discard every item of the form B → α • Xβ where X ≠ a.
5       For every item of the form B → α • Cβ, add all items of the form C → •δ.
6   If the registry contains a single item of the form A → X1X2...Xk•:
7       Discard this item.
8       For every item of the form B → α • Aβ that was discarded k steps ago,
9           add the item B → αA • β to the registry.
11      Build a portion of the parse tree.
12      For every item of the form B → α • Cβ, add all items of the form C → •δ.
13  If there are no items left in the registry and the whole input has been traversed, accept.
```

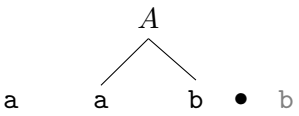
An item of the form $A \to X_1X_2 \dots X_k\bullet$ is called a *complete item.*

An NFA for the registry updates It will be convenient to represent some of the above steps using an NFA N_G. The purpose of this NFA is a bit different from the NFAs we saw in class so far: It will be used not merely to decide if its input is in G or not, but also help build a parse tree. The NFA will serve as a tool in constructing a parser for G.

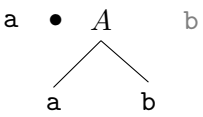
The NFA N_G has one state for every item of G, plus a start state q_0. Let us not worry about final states for now. We will attempt to represent the valid items registry after t steps by the collection of states that the NFA can be in at the same stage. Indeed, the rules for updating items in the registry look much like transitions of an NFA. Let's try to state them in NFA language:

> **Initial setup**: For every production of the form $S \to \alpha$ in G, where S is the start variable, put an ε-transition from q_0 to the state $S \to \bullet\alpha$ in N_G.
>
> **NFA update rule 1** (line 3) For every terminal a and every rule of the form $B \to \alpha a \beta$ in G, put a transition from $B \to \alpha \bullet a\beta$ to $B \to \alpha a \bullet \beta$ with label a.
>
> **NFA update rule 2** (lines 5 and 12) For every item of the form $B \to \alpha \bullet C\beta$, add an ε-transition from $B \to \alpha \bullet C\beta$ to $C \to \bullet\delta$.
>
> **NFA update rule 3** (lines 8-9) For every variable A and every rule of the form $B \to \alpha A \beta$ in G, put a transition from $B \to \alpha \bullet A\beta$ to $B \to \alpha A \bullet \beta$ with label A.

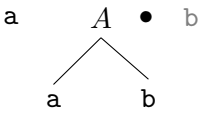
The last update rule does not quite make sense, because the variable A will never occur in the input. However, this variable comes up implicitly at some stage of the parsing. To explain how, let us recall what happened at this stage in the example we just did:



At this point, we had to backtrack and figure out which item the last portion of the parse tree came from. To do so, let us go back to the stage we must have been at just before we built the portion of the parse tree rooted at A:



At that stage there was a single item in the registry of the form $A \to \alpha \bullet A\beta$, namely the item $A \to \mathtt{a} \bullet A\mathtt{b}$. Using NFA update rule 3, upon seeing the next A, we update this item to $A \to \mathtt{a}A \bullet \mathtt{b}$ as the input becomes:

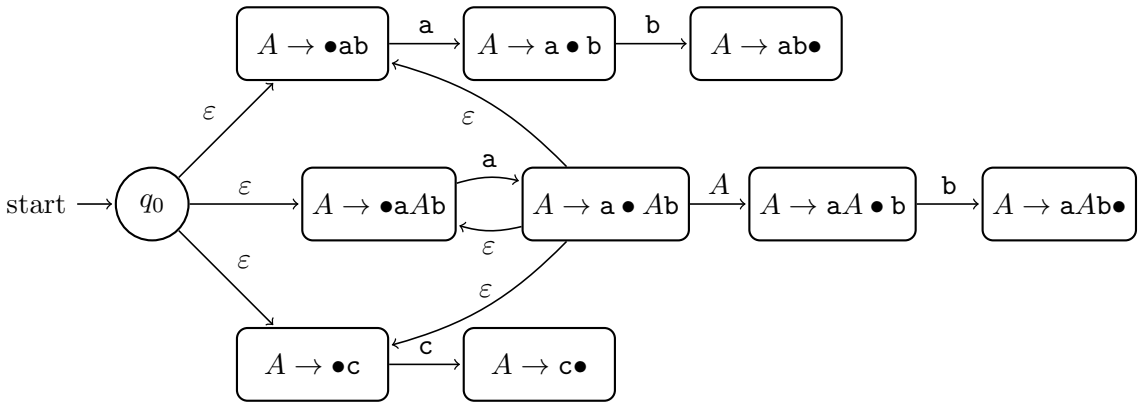


So although the variable A never occurs as an input symbol, when we discover a parse tree rooted at A, it is helpful to replace the corresponding portion of the input by the symbol A and move the $\bullet$ back in front. At this point we can apply one of the transitions described in NFA update rule 3.

Now notice that NFA update rules 1 and 3 are very similar. We can merge them into the following single rule:

> **NFA update rule 1 and 3** For every terminal or variable X and every rule of the form $B \to \alpha X \beta$ in G, put a transition from $B \to \alpha \bullet X\beta$ to $B \to \alpha X \bullet \beta$ with label a.

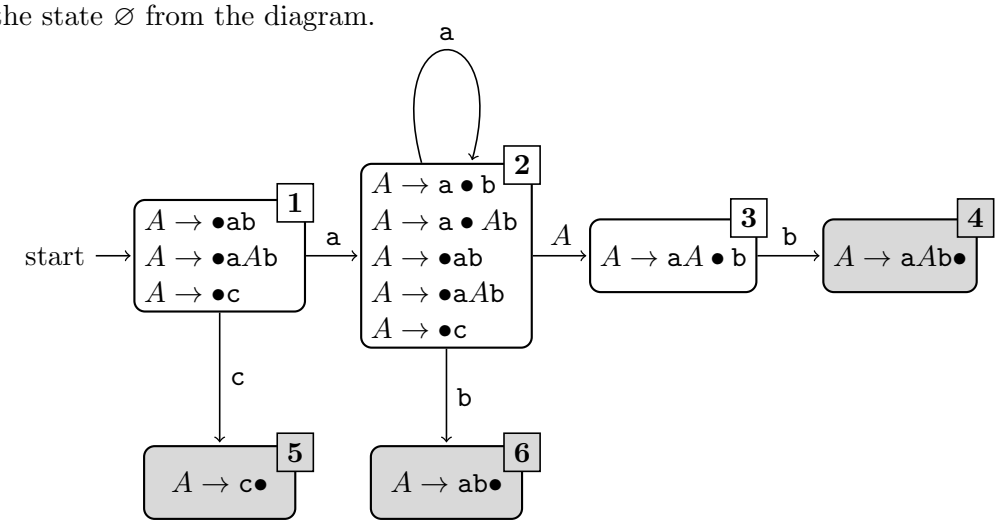
Applying the construction to the CFG G, we obtain the following NFA N_G:



The NFA we just described implements all the steps we took in the parsing example, except for the step in line 9: How can the NFA know which items were discarded k steps ago? We will take care of this later by augmenting the NFA with a stack that will implicitly keep track of the registry at different stages of the parsing process. But before we do so, there is another problem we need to solve. Eventually, to produce the parse tree, we will need to run the NFA N_G. However, an NFA is difficult to run because it can be in several states at once. Let us first convert it to a DFA.

Converting the NFA to a DFA: Shift and reduce states Recall that when we convert an NFA to a DFA, the states of the DFA are subsets of the states in the NFA. As the states of the NFAN_G correspond to items of G, the states of the corresponding DFA D_G will represent collections of items. Each such collection of items is a possible state of the registry we used in our example. The state $\varnothing$ of D_G has a special meaning – it means that the registry is empty. If an input ends up in this state, it is not in the language of G, and so it cannot be parsed.

Applying the NFA to DFA conversion rules to N_G, we obtain the following DFA D_G. For clarity we omit the state $\varnothing$ from the diagram.



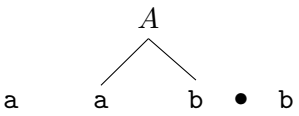
This DFA has two kinds of states: States 1, 2, and 3 all contain only items that are not complete. Such states are called *shift states*. States 4, 5, and 6 only contain a single item that is complete. Such states are called *reduce states*. Our parsing strategy crucially relies on the fact that there are no other kinds of states: If we had a shift item and a reduce item inside a single state, we would not know whether our following action should be to read the next input symbol corresponding to the shift item or build a portion of the parse tree corresponding to the reduce item.

Definition 2. A CFG G is an *LR(0) grammar* if all states in the DFA D_G are either shift states or reduce states.

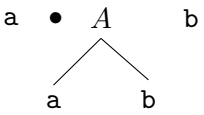
4 Tracking previous items in a stack

Let us now attempt to use the DFA D_G to parse the string `aabb`. Using the • marker to denote our position in the input, we are initially looking at •`aabb`, and we are in state 1, which is a shift

state. After reading the input `a`, we move the state 2, also a shift state, and the state of the input becomes `a` • `abb`. We read the next `a`, loop to state 2, and the input becomes `aa` • `bb`. On the next `b`, we move to state 6, which is a reduce state. We can now build the portion of the parse tree:



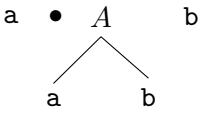
We would now like to go back to state 2, with the following picture in mind:



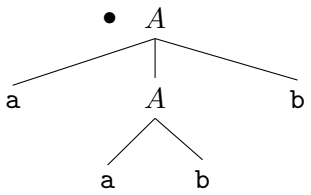
But how is the DFA D_G know that it is supposed to go to state 2? We would like to send D_G back to whichever state it was in before we processed the last two symbols `ab`. The DFA itself cannot remember this state, but we can help it store this information by using a stack.

Once we augment the DFA with a stack, what we have is no more a DFA, but a PDA P_G. The stack P_G will keep track of the sequence of states that the parsing process goes through. As usual, the stack will be initialized to `$`. After shifting the first `a`, we push the symbol `1` onto the stack, in order to remember where we were before we processed this `a`. After the second `a`, we push a `2`. After the first `b`, we push another `2`. We now find ourselves in reduce state 6 with input state `aab` • `b` and stack contents. `$122`.

We now want to replace the pattern `ab`• in the input with the pattern •A. But we must also figure out which state of the DFA to go back to before we continue the processing. To do so, we look at the state 6 item and notice that it comes from a production with two symbols on the right. To figure out where we were before we processed these two symbols, we therefore need to pop two items from the stack, and go to the state described by the last item we popped. Since the state 6 production has two symbols on the right and the first one is a `2`, after popping these two symbols we end up in the following situation: We are in state 2, the contents of the stack are `$1`, and the input is now `a` • A`b`, representing the following partial parse tree:



We are now back into a shift state, so peek at the next input symbol which is now an A. We move to state 3, the input becomes `a`A • `b`, and the stack contents are now `$12`. This is again a shift state, so we process the next input symbol. The input becomes `a`A`b`•, the stack becomes `$123`, and we are now in reduce state 4. State 4 has three symbols on the right. After reading off three symbols from the stack and building the parse tree that corresponds to this item, we are sent back to state 1 and the input is now



The stack now only contains the bottom of stack symbol \$, so we have finished parsing and are ready to accept the input.

A formal description of the PDA We now give a formal description of the PDA P_G for a general LR(0) grammar G.[2]

States: The states of P_G are the same as the states of D_G. In addition, P_G has a special start state $\bullet S$ and a special final state $S\bullet$. There are no other final states.

Transitions: The PDA D_G has the following transitions:

- For every shift state q and every transition $q \xrightarrow{X} q'$ of D_G (except those going to the dead state of D_G), P_G has a transition from q to q' with the action `read` X, `push` q.
- If q is a reduce state with item $A \to X_1 \dots X_k\bullet$, P_G has transitions going out of q with the following sequence of actions: `pop` q_k, ..., `pop` q_1, `go to state` q_1. Replace the portion of the input containing the pattern $X_1 \dots X_k\bullet$ with $\bullet A$ and construct part of the parse tree by connecting A to $X_1, \dots, X_k$.
- In addition, P_G has the special transitions `push $` from state $\bullet S$ to the start state of D_G and `pop $` from the start state of D_G to the final state $S\bullet$.

5 The parser generator

We now have all the ingredients necessary to describe the parser generator for LR(0) grammars. Recall that on input G, the output of the parser generator is an algorithm for parsing G if G is an LR(0) grammar. If G is not LR(0), the parser generator produces an error message.

On input G, where G is a CFG:
  Construct the NFA N_G.
  Convert N_G to a DFA D_G.
  If D_G has states that are neither shift or reduce states,
    Output the error message "G is not an LR(0) grammar" and halt.
  Otherwise, construct and output the PDA P_G.

[2]Technically, this is not a PDA like the ones we saw in previous lectures because in addition to deciding if the input is in the language of G, our PDA needs to build an actual parse tree.