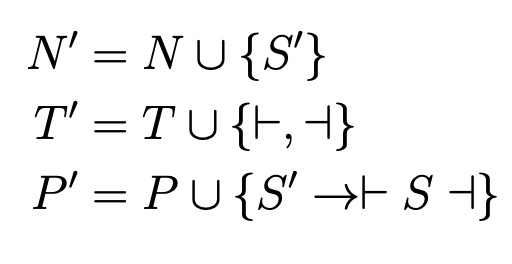
Augmented Grammars

* to simplify specification of parsing algorithms, make start symbol have only one production rule
  + if grammar doesn’t satisfy this condition, augment the grammar by creating G’ = (N’, T’, P’, S’) from our original CFG G = (N, T, P, S) where



* + - |- and -| symbols mark start and end of file

Informal Top-Down Parsing Algorithm with Example

* formally, the problem is given a CFG G = (N,Σ,P,S) and a terminal string w ∈ Σ\*, find the derivation (i.e. steps such that S => … => w or prove that w ∉ L(G))
* top-down parsing means start with S and try to get to w
  + begin at our start symbol S’ and choose a rule to apply
  + since we’re working with augmented grammars, there will be only one rule
  + apply the rule to get our first α, (i.e. a step in our derivation)
    - α1 is |- S -|
  + apply following loop:
    - look at symbols in current αi from left to right; if symbol is terminal, match it to terminal at same position in input string w; if symbol isn’t terminal, report parsing error and terminate
    - continue matching terminals until we hit first non-terminal in αi (left-most terminal)
    - choose an appropriate production rule for this non-terminal and apply it, (i.e. replace the non-terminal with the RHS of the chosen rule); this gives α(i+1)
    - repeat until there’s parse error or there’s no more non-terminals
* e.g. parsing example with input string s as |- a b y w x a -| and following augmented grammar:

A picture containing diagram

Description automatically generated

* + begin looking at symbols in α1 left to right; first symbol is |- which matches the first symbol in the string s
  + move to the next symbol and see that it is the non-terminal S; apply the rule S → AyB to obtain α2, |- AyB -|
  + |- from α2 matches the first symbol in the string s and then we encounter the leftmost nonterminal, A; algorithm chooses the appropriate rule by doing a lookahead at what the next unmatched symbol in the input is (a)
  + algorithm chooses to apply A → ab and gives us α3, |- abyB -|
  + the process repeats until there are no symbols left in the last αi and there are no unmatched symbols in the input string so the parse was successful
  + derivation obtained is:

Diagram

Description automatically generated

* optimization to above algorithm is to stop tracking prefix of input that’s already been matched every single time
  + when αi is xAβ and we match x to the input, we will only keep track of the remaining Aβ
  + then we apply the rule for A and this will produce γβ so in the next step, we will look at the first symbol in γβ and continue with our algorithm
* top-down parsing algorithm is:

Graphical user interface, text, application, email

Description automatically generated

* + at any given point, stack content rep truncated αi (i.e. current part of derivation with matched prefix removed)
  + leftmost symbol is on top of the stack
    - reason that we push the symbols in γ in reverse order
* using this algorithm with previous example’s grammar and input string s as |- a b y w x a -| (Predict table is shown)

A picture containing text, clock, screenshot

Description automatically generated

Table

Description automatically generated

* + can obtain αi by concatenating contents in Read and Stack columns
* algorithm is called LL(1) parsing
  + first L rep left-to-right scan of input
  + second L indicate algorithm produces leftmost derivations
  + 1 indicates we look ahead one symbol
* grammar is LL(1) if and only if each cell of the Predict table contains at most one rule
* LL(1) parsing algorithm can produce an error when:
  + TOS is terminal but it doesn’t match the next input symbol
  + algorithm queries Predict[A][a] for some A and a and finds either no rule, or more than one rule
* in practice, almost no compilers use LL(1) and instead use hand-written parsers (i.e. recursive descent parsing because each non-terminal is recursive function)
  + LL(1) has limitations and hand-written parsers bypass these limitations on an ad hoc (i.e. case-by-case) basis

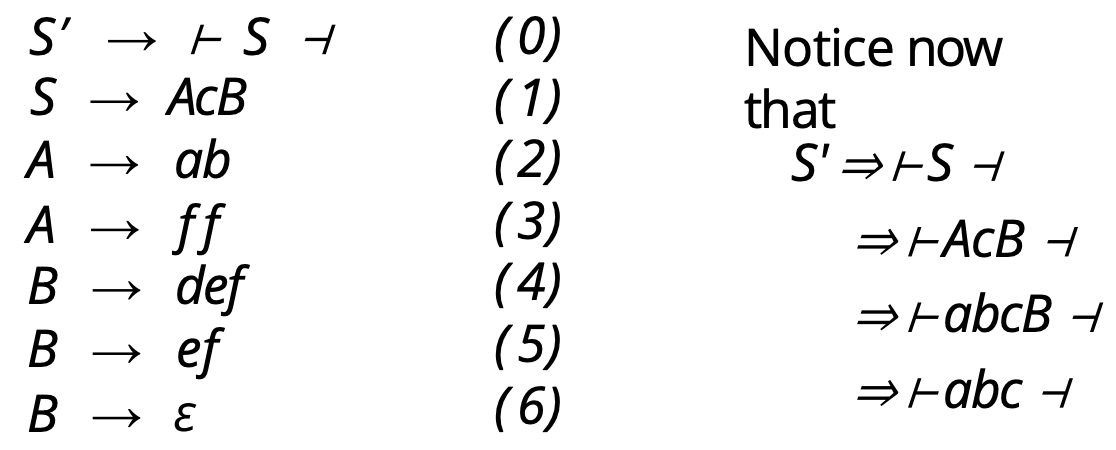
Constructing the Predict Table

* Predict table is 2D lookup table
  + in reality, it’s a function that produces a set of rules that can apply when A ∈ N’ (a non-terminal) is on the TOS and a ∈ T’ (a terminal) is the next input symbol
  + must know which rule to apply based on a single symbol
* definition of Predict is: Predict[A][a] = {A → β : a ∈ First(β)} ∪ {A → β : β is nullable and a ∈ Follow(A)}
  + First(β) = {a ∈ T’ : β ⇒\* aγ, for some γ ∈ V\*}
    - First(β) is the set of characters/terminals that can be the first symbol of a derivation starting from β ∈ V\*
    - i.e. if β can derive a string where the first symbol is an a then we could apply the rule A → β since we will be able to get to a

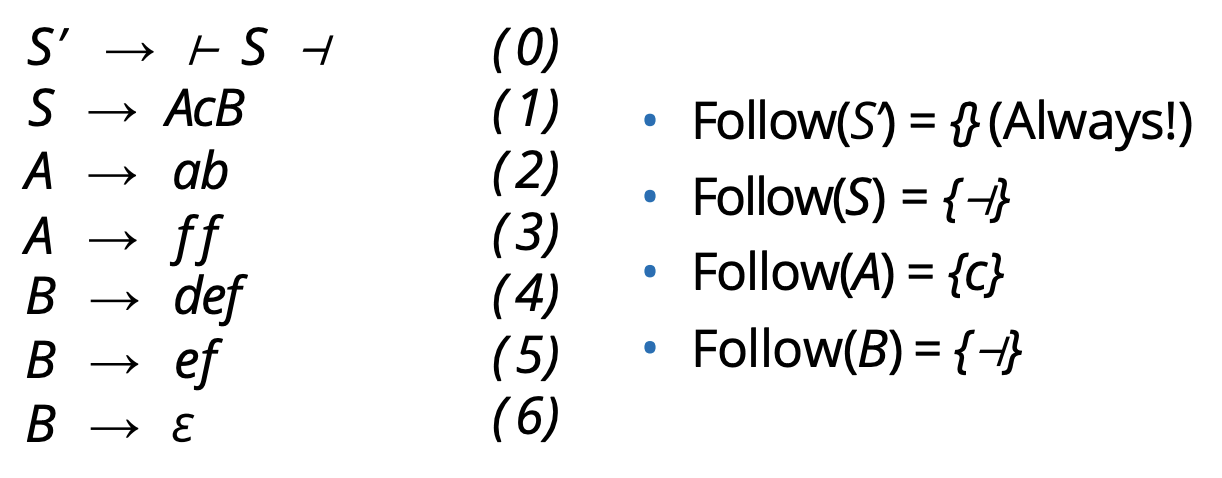
Text

Description automatically generated

* + - to compute First(L M), needed First(L) first because the only transition function for L leads to another non-terminal
  + if there’s a rule of the form A → ε, then a does not need to be derived from A but from some symbol that follows A
    - define Nullable(β) = true iff β ⇒\* ε and false otherwise
    - define Follow(A) = {b ∈ T’ : S’ ⇒\* αAbβ for some α, β ∈ V\*}
      * Follow(A) is a set of terminals that can come immediately after A in a derivation starting at the start symbol S’
* without Nullable and Follow, Predict for the below example would lead to an error:



* + when we reach ⊢abcB ⊣, the stack is ⊣B and remaining input is ⊣
  + Predict(B, ⊣) is empty since ⊣∉ First(B) so it gives an error
  + since Nullable(B) = true and Follow(B) = {⊣}
* examples of Follow:

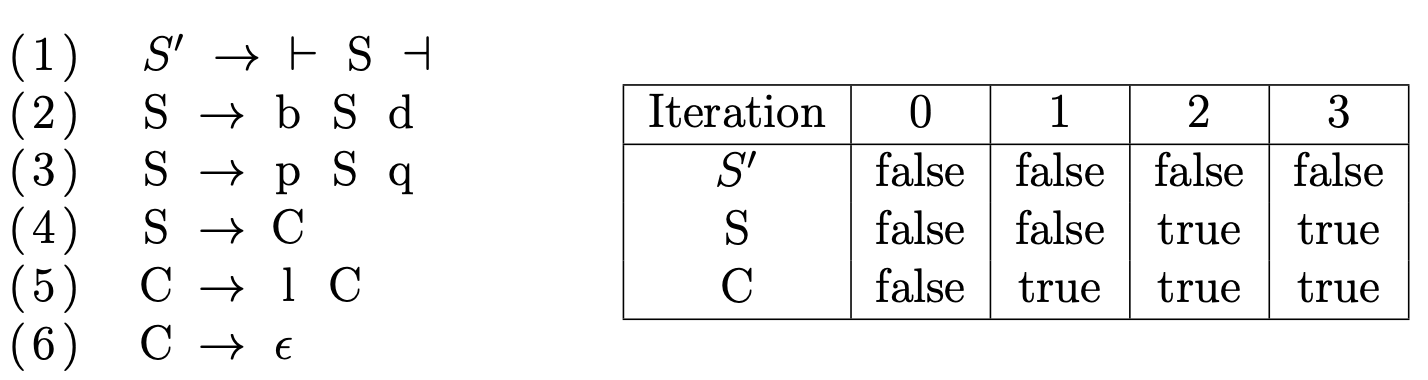


* if Nullable(A) = false, then Follow(A) is still the same set of terminals but they won’t be relevant since we’ll use First(A)
* properties of Nullable:
  + Nullable(ε) = true by definition
  + Nullable(β) = false whenever β contains a terminal symbol
  + Nullable(AB) = Nullable(A) ∧ Nullable(B)

Text

Description automatically generated with medium confidence

* + i.e. non-terminal A is nullable if it directly derives ε or has a rule of the form A → B1 · · · Bk where each of B1 through Bk are nullable
  + identification of a new nullable non-terminal may lead to other non-terminals to be nullable so algorithm must continue until a complete pass through all the rules don’t identify a new nullable non-terminal
  + e.g.



* to compute First:

Text

Description automatically generated

Text

Description automatically generated with medium confidence

* + main idea is to keep processing B1B2 ...Bk from a production rule until you encounter a terminal or a symbol that is not nullable and then go to the next rule; repeat until no changes are made during the processing
    - ignore trivial productions of the form A → ε because ε isn’t a real symbol and can’t be in the First set
  + first, we compute First(A) for all A ∈ N using algorithm 3 and then we compute First(β) for all relevant β ∈ V\* using algorithm 4
  + e.g.

Table

Description automatically generated

* to compute Follow:

Text, letter

Description automatically generated

* + algorithm goes through each rule A → B1 · · · Bk and updates Follow(Bi) to contain First(Bi+1 · · · Bk) since First(Bi+1 · · · Bk) is the set of terminals that can occur as the first symbol in a derivation starting with Bi+1 · · · Bk
  + if Bi is the last symbol in the rule or if all symbols after Bi are nullable, Follow(Bi) should contain Follow(A)
  + e.g.

Table

Description automatically generated

* + - on first iteration, the symbols a, d and q are added to Follow(S) since, on the right-hand side of Rules 1, 2 and 3 respectively, S is followed by these three symbols
    - for Follow©, C is not followed by any symbol on the right-hand side of rule 4 so we must add Follow(S) to Follow(C)
* to compute Predict:

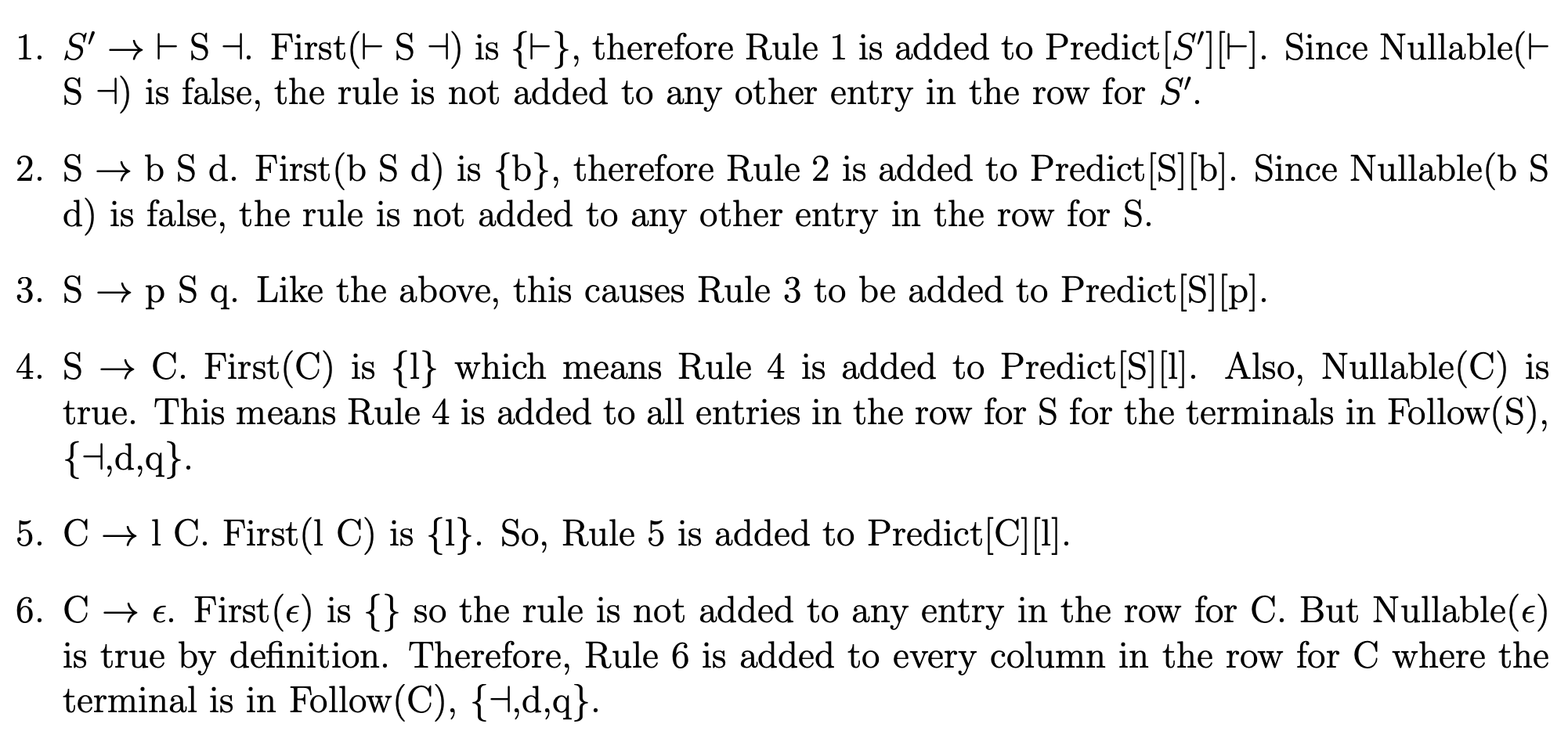
Text

Description automatically generated

* + e.g.

Table

Description automatically generated



* + - above is an LL(1) grammar since each entry in the Predict table has at most one rule
* summary/cheat sheet:

Text

Description automatically generated

Parse Trees

* top-down parsing is a way of obtaining a leftmost derivation for an input string with the eventual goal of generating the parse tree that represents the structure of the input
* e.g.

A picture containing text, receipt

Description automatically generated

* + in the above derivation, we see the input is a word in the language because the algorithm accepts the input
  + a parser can use the derivation to generate a parse tree, as shown below:

A picture containing diagram

Description automatically generated

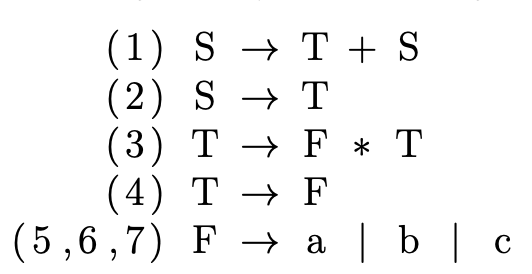
Limitations of LL(1) Grammars

* a grammar is LL(1) if and only if
  + no two distinct rules with the same LHS can generate the same first terminal
  + no nullable symbol A has the same terminal a in both its first and follow sets
  + there is only one way to derive ε from a nullable symbol
* e.g. the below grammar (used to respect BEDMAS) is not LL(1):

Text

Description automatically generated

* + the derivation for a would be S => T => F => a and the derivation for a + b would be S => S + T => T + T => F + T => a + T => a + F => a + b
    - the two strings have the same first character but require different starting rules (i.e. {1,2}⊆ Predict(S, a))
    - this means the (S, a) cell of the Predict table contains two rules
* left recursive grammars are never LL(1) because the recursion on a non-terminal happens on the left within the RHS
* a grammar with two or more rules for the same non-terminal with a common left prefix of length k, cannot be LL(k)
  + e.g. we could switch the previous grammar to right recursion:



* + - given S on the stack and the lookahead of just a, the algorithm cannot decide whether it should use S → T or S → T + S (since it doesn’t know what comes after a)
    - the common left prefix is of length 1 and therefore the grammar is still not LL(1)
    - one solution is to increase lookahead so above grammar is LL(2)
* can use left factorization to make a right recursive grammar LL(1)
  + if a grammar has rules A → αβ1|...|αβn|γ where α ≠ ε and γ is representative of other productions that do not begin with α, then we can change this to the following equivalent grammar by left factoring: A
    - A → αB|γ
    - B → β1| . . . |βn
  + e.g.

A picture containing text

Description automatically generated

* + - this right recursive and factored grammar is LL(1), as shown by the Predict table:

Diagram, table

Description automatically generated

* top-down parsing, LL(1), is not compatible with left-associative grammars
  + this can be an issue when dealing grammars that involve math (which is left-associative)