Lecture Overview

- **Context-Free Grammars**
- **Derivations and Parse Trees**
- **Parsing**
  - **Left-factoring**
  - **Predictive Parsing Example**
- **Making a Predictive Grammar**
  - **Removing Direct Left-recursion**

Chapter 3 in text

Context-Free Grammars

$G = (N, T, S, P)$

N: Set of nonterminals (often capital letters)

T: Set of terminals (often lower-case letters and symbols)

S: Start symbol (element of N)

P: Productions of the form

$A \rightarrow \alpha$

where α is a string on $N \cup T \cup \{\varepsilon\}$.

This is the BNF form of productions (Backus-Naur Form). The milestones use the EBNF (Extended BNF) form, where α can be a *regular expression* on $N \cup T \cup \{\varepsilon\}$. This is just a shorthand and does not change which languages have a grammar.

There are standard conventions for giving a grammar. We only list the productions. N is taken to be the set of all symbols on the left-hand side of any production. T is taken to be the set of all other symbols appearing in productions. S is taken to be the left-hand symbol of the first production.

For instance, in the grammar:

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow F * T$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow c$

$F \rightarrow i$

The nonterminals are the set {E, T, F}, the terminals are the set {+, *, (,), c, i}, and the start symbol is E.

This grammar models an arithmetic expression; E represents “expression”, T represents “term”, and F represents “factor”. c is for “constant”, and i for “identifier.”

We can also write this grammar as follows:

$$
\begin{aligned}
E &\longrightarrow T + E \\
&\quad\ \ T \\
T &\longrightarrow F * T \\
&\quad\ \ F \\
F &\longrightarrow (E) \\
&\quad\ \ c \\
&\quad\ \ i
\end{aligned}
$$

In this, the lines without a production LHS have the LHS of the most recent production with a LHS.

We can even write the grammar as

$$
\begin{aligned}
E &\longrightarrow T + E \mid T \\
T &\longrightarrow F * T \mid F \\
F &\longrightarrow (E) \mid c \mid i
\end{aligned}
$$

where the vertical bar is read as “or” (as in regular expressions).

Derivations and Parse Trees

A production $A \rightarrow \alpha$ is called a *production for A.*

A **derivation** in a grammar starts with α_0 being the start symbol, and in each step k it derives α_k from α_{k-1} by finding a nonterminal in α_{k-1} and replacing it with the RHS of a production for that nonterminal. The derivation halts if at some step k there are no nonterminals remaining in α_{k-1}.

$\underline{E} \Rightarrow T + \underline{E} \Rightarrow T + T + \underline{E} \Rightarrow \underline{T} + T + T \Rightarrow \underline{F} + T + T \Rightarrow$

$i + \underline{T} + T \Rightarrow i + \underline{F} * T + T \Rightarrow i + c * \underline{T} + T \Rightarrow$

$i + c * \underline{F} + T \Rightarrow i + c * i + \underline{T} \Rightarrow i + c * i + \underline{F} * T \Rightarrow$

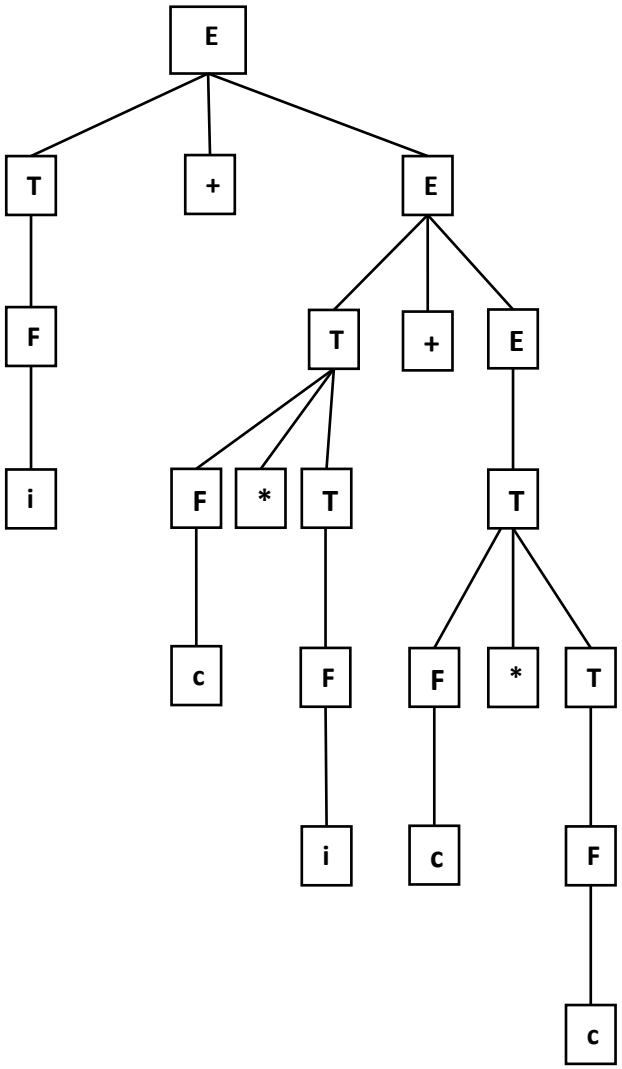
$i + c * i + c * \underline{T} \Rightarrow i + c * i + c * \underline{F} \Rightarrow i + c * i + c * c$

The last string in a terminating derivation is called a **sentence** or a **word** in the language of the grammar.

For instance, i + c*i + c*c is a word in the language of our grammar; i.e. it is a valid arithmetic expression.

The **language of a grammar** is all possible sentences that you can derive from the grammar. In most grammars we will be interested in, there are an infinite number of different derivations and therefore the language of the grammar has an infinite number of members.

A **parse tree** of a derivation is the tree one gets by placing the RHS of a production used as a step in the derivation below the LHS and as children of it. The parse tree corresponding to the previous derivation is:



Different derivations can lead to the same parse tree, but different parse trees must lead to different derivations.

Parsing

It is the job of the parser to look at an input stream (of tokens) and decide which derivation or parse tree of the language that input stream belongs to, if any. Languages are designed to make this task possible and efficient.

For instance, in many languages, each type of statement starts with its own keyword (including assignment statements).

stmt → **while (** cond **)** stmt

for (forControl **)** stmt

let target **=** expr ;

return expr ;

call ident **(** args **) ;**

…

This allows the parser to decide which alternative to use by examining the currently-being-processed token. Parsing becomes more difficult if two

alternatives for a nonterminal start with the same token.

In our sample grammar, we have two different alternatives for E that start with the same symbol. (We also have two different alternatives for T that start with the same symbol.) Here I let the bold parentheses stand for literal parentheses.

E → **T** + E | **T**

T → **F** * T | **F**

F → **(** E **)** | c | i

We can eliminate this problem through a process known as **left-factoring.** In EBNF, with the nonbold parentheses standing for EBNF grouping:

E → **T** (+ E | ε)

T → **F** (* T | ε)

F → **(** E **)** | c | i

Or, in BNF:

$$E \rightarrow T\ Y$$

$$Y \rightarrow +\ E \mid \varepsilon$$

$$T \rightarrow F\ Z$$

$$Z \rightarrow *\ T \mid \varepsilon$$

$$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid c \mid i$$

This grammar is now **predictive—**in parsing a sentence, you can always tell which production to use by what the next token is.

Here's another example. In this grammar, nonterminals are words/identifiers and terminals are written in bold.

doStatement → **do** statement **while (** condition **)**

doStatementVariant

doStatementVariant →

do statement **until (** condition **)**

This is **not** left-factored, as the two alternatives for doStatement can both start with the token **do**. (This holds even though the second option doesn't explicitly have the **do** in the RHS.) So first we have to substitute up until both options do explicitly have the **do**. This is simple in this case.

doStatement → **do** statement **while (** condition **)**

do statement **until (** condition **)**

Now we factor out the common prefix:

doStatement → **do** statement doStatementEnd

doStatementEnd → **while (** condition **)**

until (condition **)**

Predictive Parsing Example

For instance, suppose we have the sentence 3 + 4*p and the expression grammar shown above.

The parser starts with the start symbol E. The current token is the 3, an instance of **c**. There's only one production to take, so it takes the production (does a step in the derivation):

$E \Rightarrow T\ Y$

The first nonterminal in the current string is T, and the current token is still **c.** There's again only one production to take, so we do the step:

$TY \Rightarrow FZY$

The first nonterminal in the current string is now F, and the current token still **c**. The three alternatives for F start with (, c, and i—we take the one that matches the current token.

$FZY \Rightarrow cZY$

This match consumes the current token and so the current token becomes **+**. The first nonterminal in the string is now Z, and no nonempty alternative for Z starts with +, so we take the empty alternative.

$cZY \Rightarrow cY \qquad (= c\varepsilon Y)$

Now the first nonterminal is Y, and the current token is **+**, so we take the alternative for Y that starts with **+**.

$cY \Rightarrow c + E$

This matches the plus, so we read the next token, which is **4**, and instance of **c**. The first nonterminal is E, and it has only one RHS. T also has only one RHS:

$c + E \Rightarrow c + T\ Y \Rightarrow c + F\ Z\ Y$

The first nonterminal in the current string is now F, and the current token still **c**. We take the alternative for F that matches the current token.

$c + F\ Z\ Y \Rightarrow c + c\ Z\ Y$

Our **c** is now matched, so we get the next token, which is *. The first nonterminal is Z, so we take the alternative for Z that starts with *:

c + c Z Y ⇒ c + c * T Y

This matches the *, so we get the next token, which is **p**, an instance of **i.** The first nonterminal is T, for which there is only one alternative:

c + c * T Y ⇒ c + c * F Z Y

The leftmost nonterminal is F, so we take the alternative for F that matches the current token, **i**.

c + c * F Z Y ⇒ c + c * i Z Y

We advance the current token to **end-of-input.** Z is the first nonterminal, and we do not match the first alternative so we take the empty alternative.

c + c * i Z Y ⇒ c + c * i Y

Now Y is the first nonterminal, and again we do not match its first alternative, so we take the empty alternative.

c + c * i Y ⇒ c + c*i

Since we now have all terminals, we are done; we have successfully parsed 3 + 4 * p.

Making a Predictive Grammar

A predictive grammar has the following two properties. We assume the grammar is in BNF.

1) No two alternatives for any production can possibly start with the same terminal. (If this occurs, then **left-factor** the grammar.)
2) There is no **left-recursion**. This is a situation where some

   $A\ \alpha \Rightarrow \ldots \Rightarrow A\ \beta$

   when always expanding the leftmost nonterminal.

For example, $E \rightarrow E + E$ is a single production that has left recursion (this is called **direct** left-recursion). Another example (this time of **indirect** left-recursion) is:

$E \rightarrow T\ **\ E$

$T \rightarrow E + E$

because

$$\mathbf{E} \Rightarrow T ** E \Rightarrow \mathbf{E} + E ** E$$

Removing direct left-recursion

To remove **direct left-recursion**, write all of the productions as two (left-factored) productions of the form:

$$A \rightarrow A\ \alpha$$

$$A \rightarrow \beta$$

where α and β can be regular expressions. Clearly, A will derive a bunch of α's at the end of its string (0 or more) and finally a β at the beginning. So, in regular expressions, A derives β α*. We can write this in BNF as:

$$A \rightarrow \beta\ A'$$

$$A' \rightarrow \alpha\ A'$$

$$A' \rightarrow \varepsilon$$

For example, if you had:

$E \rightarrow E + E$

$E * E$

(E)

i

c

You could make this the two productions

$E \rightarrow E (+ E \mid * E)$

$E \rightarrow (E) \mid i \mid c$

Here $\alpha = (+ E \mid * E)$ and $\beta = (E) \mid i \mid c$

So we could write

$E \rightarrow ((E) \mid i \mid c) E'$

$E' \rightarrow (+ E \mid * E) E' \mid \varepsilon$

and this is the equivalent grammar without left-recursion.

For eliminating **indirect left-recursion**, use the following algorithm:

Assume nonterminals are A_1, A_2, A_3, … ,A_n.

```
for i = 1 to n
    for j = 1 to i-1
        if ∃ a production A_i → A_j γ
            replace that production by expanding A_j
            with all alternatives for A_j
    rewrite any direct left recursion on A_i.
```

The first iteration of the i loop eliminates direct left-recursion from A_1. The second iteration first replaces any $A_2 \rightarrow A_1 \gamma$ productions and then eliminates direct left-recursion from A_2. Now A_1 and A_2 have no direct left recursion, and A_2 has no production starting with A_1.

The third iteration of the i loop replaces $A_3 \rightarrow A_1 \gamma$ productions (which could introduce $A_3 \rightarrow A_2 \gamma$ productions) then replaces $A_3 \rightarrow A_2 \gamma$ productions

(which can't introduce any $A_3 \rightarrow A_1 \gamma$ productions), and then eliminates left recursion from A_3. Now A_3 has no direct left recursion and no production starting with A_1 or A_2.

And so on. Iteration i makes it so that A_i has no production starting with A_k for any $k < I$, and clears left-recursion from A_i.

At the end, all productions are left-recursion-free.

$A \rightarrow Ax \mid ABy \mid Bz \mid x$

$B \rightarrow Ax \mid Cw \mid Cx$

$C \rightarrow Az \mid Bz \mid Cy \mid v$

i=1: eliminate direct left-recursion from A.

$A \rightarrow A(x \mid By) \mid (Bz \mid x)$ $\qquad$ $A \rightarrow A\,\alpha \mid \beta$

$A \rightarrow (Bz \mid x)\, A'$ $\qquad$ $A \rightarrow \beta\, A'$

$A' \rightarrow (x \mid By)\, A' \mid \varepsilon$ $\qquad$ $A' \rightarrow \alpha\, A' \mid \varepsilon$

This leaves us with the grammar:

A ⟶ (Bz | x) A’
A’ ⟶ (x | By) A’ | ε
B ⟶ Ax | Cw | Cx
C ⟶ Az | Bz | Cy | v

i=2: j=1: replace productions for second symbol (B) that start with the first (A):

B ⟶ Ax | Cw | Cx

B ⟶ (Bz | x) A’ x | Cw | Cx

We now have:

A ⟶ (Bz | x) A’
A’ ⟶ (x | By) A’ | ε
B ⟶ (Bz | x) A’ x | Cw | Cx
C ⟶ Az | Bz | Cy | v

i=2: eliminate direct left-recursion from B.

B → Bz A’ x | (x A’ x| Cw | Cx) B → B α |β

B → (x A’ x| Cw | Cx) B’ B → β B’

B’ → z A’ x B’ | ε B’ → α B’| ε

So we now have:

A → (Bz | x) A’

A’ → (x | By) A’ | ε

B → (x A’ x| Cw | Cx) B’

B’ → z A’ x B’ | ε

C → Az | Bz | Cy | v

i=3: j=1: replace productions for third symbol (C) that start with the first (A):

C → (Bz | x) A’ z | Bz | Cy | v

or

C ⟶ Bz A’ z| x A’ z | Bz | Cy | v

i=3: j=2: replace productions for third symbol (C) that start with the second (B):

C ⟶ (x A’ x| Cw | Cx) B’ z A’ z| x A’ z |
(x A’ x| Cw | Cx) B’ z | Cy | v

we can rewrite this as:

C ⟶ xA’xB’zA’z | CwB’zA’z | CxB’zA’z | xA’z|
xA’xB’z | CwB’z | CxB’z | Cy | v

and change the order of the terms and regroup to get:

C ⟶ C(wB’zA’z | xB’zA’z | wB’z | xB’z | y) |
(A’xB’zA’z | xA’z| xA’xB’z | v)

C ⟶ C α |β

i=3: eliminate direct left-recursion from C.

C ⟶ (A'xB'zA'z | xA'z| xA'xB'z | v)C'
C' ⟶ (wB'zA'z | xB'zA'z | wB'z | xB'z | y)C'| ε

C ⟶ β C'
C' ⟶ α C'| ε

We end up with the grammar:

A ⟶ (Bz | x) A'
A' ⟶ (x | By) A' | ε
B ⟶ (x A' x| Cw | Cx) B'
B' ⟶ z A' x B' | ε
C ⟶ (A'xB'zA'z | xA'z| xA'xB'z | v)C'
C' ⟶ (wB'zA'z | xB'zA'z | wB'z | xB'z | y)C'| ε

This has no left-recursion, which was the goal.

So, as you can see, this algorithmic process can get quite messy. Hopefully one doesn't have a grammar that gives this much mess, but often this process is done by machine and not by hand.

Suppose we had:

E → E + E	low precedence
E – E	also low precedence
E * E	medium precedence
-E	high precedence
(E)	**highest precedence**
i	
c	

To parse this, we will construct a grammar from high to low precedence. Consider i and c as super-high precedence. So we'll call them literals, and have a nonterminal LE (Literal expression) for them:

LE → i | c

The next highest precedence level is parentheses; we'll introduce a nonterminal PE (Parenthesized expression) for them. The kind of expression we want in a parenthesized expression is a full-blown expression (E). We also allow a literal to be a parenthesized expression:

PE → **(** E **)** | LE

LE → i | c

The next highest level is unary minus; we'll introduce UE (Unary expression) for them. We allow a parenthesized expression to be a unary expression. If we allow just one unary operator to occur before the expression, we get:

UE → -PE | PE

Note that we're using the next highest level for the operand of the unary and for skipping the unary. If instead we want more than one unary operator to be allowed (e.g. ---14), then we should use UE itself for the operand.

UE → -UE | PE

PE → **(** E **)** | LE

LE → i | c

The next highest level is multiplication. Here we use UE as the first argument and the skip option.

ME → UE * ME |UE

ME is used as the second argument so we can have expressions with three or more things multiplied together (e.g. 3 * 4 * n).

The lowest level is addition and subtraction. We'll call the nonterminal AE (additive expression). We use the next level up (ME) as the first argument and the skip option.

AE → ME + AE |ME – AE | ME

ME → UE * ME |UE

UE → -UE | PE

PE → **(** E **)** | LE

LE → i | c

Let's left-factor the productions:

AE → ME (+ AE |– AE | ε)

ME → UE (* ME | ε)

UE → -UE | PE

PE → **(** E **)** | LE

LE → i | c

Let's work on the first production.

AE → ME ((+|-) AE | ε) *remove recursion*

AE → ME ((+|-) ME)*

Now the second:

ME → UE (* UE)*

ME → UE ((*|/|%) UE)*

The third:

UE → (-)*PE

UE → (-|+)*PE

The fourth and fifth are about what we want, so we get:

E → AE

AE → ME ((+|-) ME)*

ME → UE ((*|/) UE)*

UE → (-|+)*PE

PE → **(** E **)** | LE

LE → i | c

This grammar is much like what you see in the Tan parser.

Suppose I put in binary operators @, $ that are even lower precedence than the additive operators. Then we'd get:

E → LPE

LPE = AE ((@|$) AE)*

…