λ Derivations

When parsing, we'll sometimes need to know which non-terminals can derive λ. (λ is "invisible" and hence tricky to parse).

We can use the following marking algorithm to decide which non-terminals derive λ

(1) For each production $A \rightarrow \lambda$
    mark A

(2) Repeat
        If the entire righthand
        side of a production
        is marked
        Then mark the lefthand side
    Until no more non-terminals
        can be marked

As an example consider

$$\mathbf{S} \rightarrow \mathbf{A}\ \ \mathbf{B}\ \ \mathbf{C}$$

$$\mathbf{A} \rightarrow \mathbf{a}$$

$$\mathbf{B} \rightarrow \mathbf{C}\ \ \mathbf{D}$$

$$\mathbf{D} \rightarrow \mathbf{d}$$

$$\mid\ \lambda$$

$$\mathbf{C} \rightarrow \mathbf{c}$$

$$\mid\ \lambda$$

Recall that compilers prefer an unambiguous grammar because a unique parse tree structure can be guaranteed for all inputs.

Hence a unique translation, guided by the parse tree structure, will be obtained.

We would like an algorithm that checks if a grammar is ambiguous.

Unfortunately, it is undecidable whether a given CFG is ambiguous, so such an algorithm is impossible to create.

Fortunately for certain grammar classes, including those for which we can generate parsers, we can prove included grammars are unambiguous.

Potentially, the most serious flaw that a grammar might have is that it generates the “wrong language."

This is a subtle point as a grammar serves as the *definition* of a language.

For established languages (like C or Java) there is usually a suite of programs created to test and validate new compilers. An incorrect grammar will almost certainly lead to incorrect compilations of test programs, which can be automatically recognized.

For new languages, initial implementors must thoroughly test the parser to verify that inputs are scanned and parsed as expected.

Parsers and Recognizers

Given a sequence of tokens, we can ask:

"Is this input syntactically valid?"

(Is it generable from the grammar?).

A program that answers this question is a *recognizer*.

Alternatively, we can ask:

"Is this input valid and, if it is, what is its structure (parse tree)?"

A program that answers this more general question is termed a *parser*.

We plan to use language structure to drive compilers, so we will be especially interested in parsers.

Two general approaches to parsing exist.

The first approach is *top-down*.

A parser is top-down if it "discovers" the parse tree corresponding to a token sequence by starting at the top of the tree (the start symbol), and then expanding the tree (via predictions) in a depth-first manner.

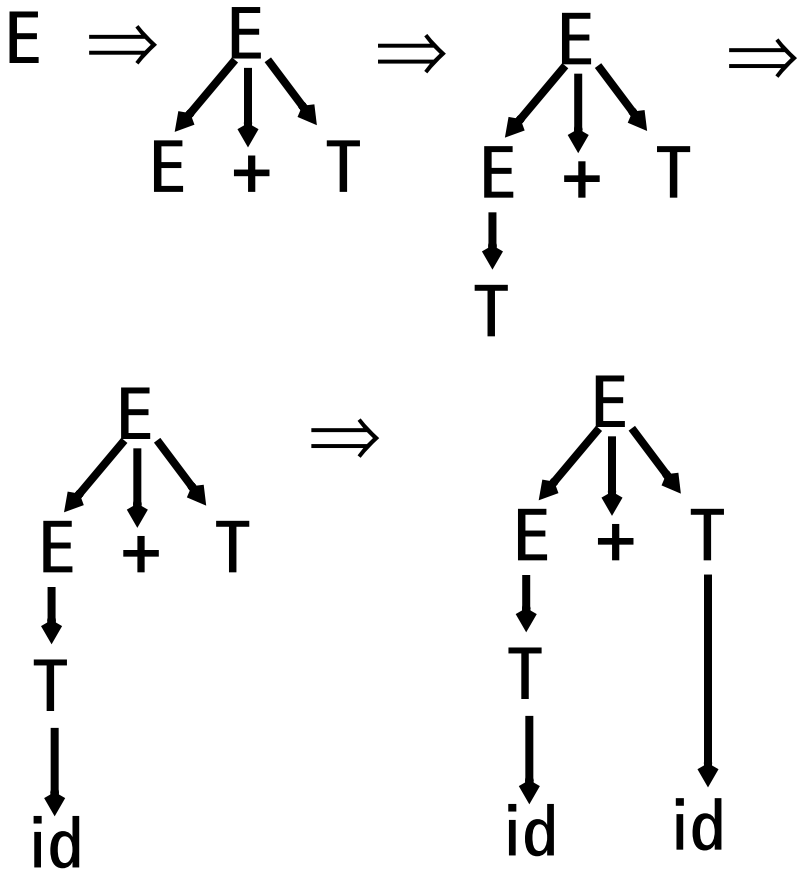
Top-down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins.

Consider

E → E + T | T

T → T * id | id

To parse **id+id** in a top-down manner, a parser build a parse tree in the following steps:



A wide variety of parsing techniques take a different approach.

They belong to the class of *bottom-up* parsers.

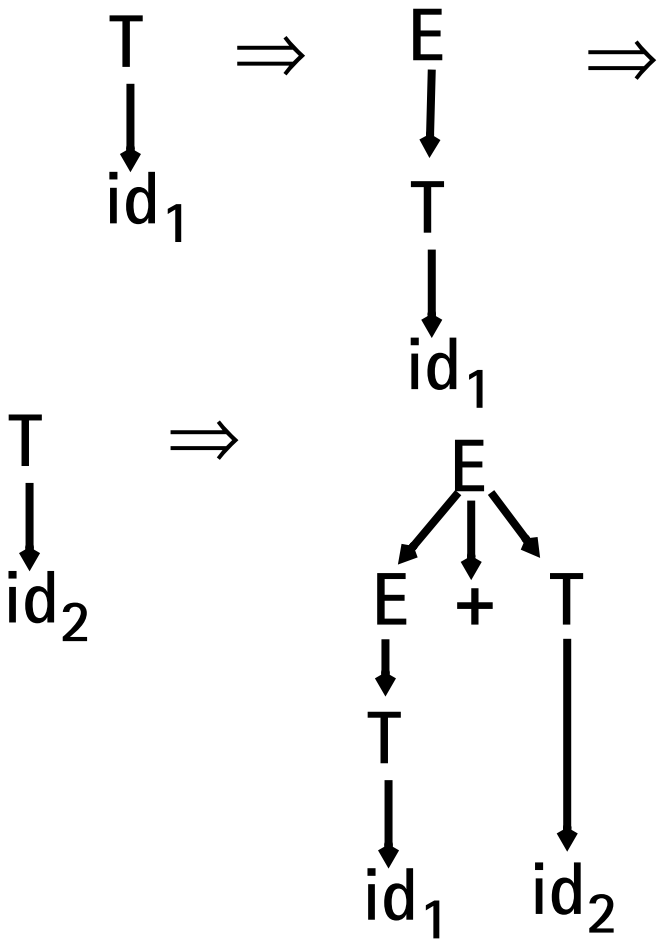
As the name suggests, bottom-up parsers discover the structure of a parse tree by beginning at its bottom (at the leaves of the tree which are terminal symbols) and determining the productions used to generate the leaves.

Then the productions used to generate the immediate parents of the leaves are discovered.

The parser continues until it reaches the production used to expand the start symbol.

At this point the entire parse tree has been determined.

A bottom-up parse of $\mathbf{id_1+id_2}$ would follow the following steps:



A Simple Top-Down Parser

We'll build a rudimentary top-down parser that simply tries each possible expansion of a non-terminal, in order of production definition.

If an expansion leads to a token sequence that doesn't match the current token being parsed, we *backup* and try the next possible production choice.

We stop when all the input tokens are correctly matched or when all possible production choices have been tried.

Example

Given the productions

S → **a**

| **(S)**

we try **a**, then **(a)**, then **((a))**, etc.

Let's next try an additional alternative:

S → **a**

| **(S)**

| **(S]**

Let's try to parse **a**, then **(a]**, then **((a]]**, etc.

We'll count the number of productions we try for each input.

- For input = **a**
  We try **S** → **a** which works.
  Matches needed = 1
- For input = **(a]**
  We try **S** → **a** which fails.
  We next try **S** → **(S)**.
  We expand the inner **S** three different ways; all fail.
  Finally, we try **S** → **(S]**.
  The inner **S** expands to **a**, which works.
  Total matches tried =
  1 + (1+3)+(1+1)= 7.
- For input = **((a]]**
  We try **S** → **a** which fails.
  We next try **S** → **(S)**.
  We match the inner **S** to **(a]** using 7 steps, then fail to match the last **]**.
  Finally, we try **S** → **(S]**.
  We match the inner **S** to **(a]** using 7

steps, then match the last **]**.
Total matches tried =
1 + (1+7)+(1+7)= 17.

- For input = **(((a]]]**
  We try **S** → **a** which fails.
  We next try **S** → **(S)**.
  We match the inner **S** to **((a]]** using 17 steps, then fail to match the last **]**.
  Finally, we try **S** → **(S]**.
  We match the inner **S** to **((a]]** using 17 steps, then match the last **]**.
  Total matches tried =
  1 + (1+17) + (1+17) = 37.

Adding one extra **(...]** pair *doubles* the number of matches we need to do the parse.

In fact to parse $\mathbf{(^i a]^i}$ takes $5*2^i-3$ matches. This is *exponential* growth!

With a more effective *dynamic programming* approach, in which results of intermediate parsing steps are cached, we can reduce the number of matches needed to $\mathbf{n}^3$ for an input with **n** tokens.

Is this acceptable?

No!

Typical source programs have at least 1000 tokens, and $1000^3 = 10^9$ is a lot of steps, even for a fast modern computer.

The solution?

—Smarter selection in the choice of productions we try.

Reading Assignment

Read Chapter 5 of

Crafting a Compiler, Second Edition.

Prediction

We want to avoid trying productions that can't possibly work.

For example, if the current token to be parsed is an identifier, it is useless to try a production that begins with an integer literal.

Before we try a production, we'll consider the set of terminals it might initially produce. If the current token is in this set, we'll try the production.

If it isn't, there is no way the production being considered could be part of the parse, so we'll ignore it.

A *predict function* tells us the set of tokens that might be initially generated from any production.

For $A \rightarrow X_1 \ldots X_n$, Predict($A \rightarrow X_1 \ldots X_n$) = Set of all initial (first) tokens derivable from $A \rightarrow X_1 \ldots X_n$

$= \{a \text{ in } V_t \mid A \rightarrow X_1 \ldots X_n \Rightarrow^* a \ldots\}$

For example, given

Stmt → Label id = Expr ;
| Label if Expr then Stmt ;
| Label read (IdList) ;
| Label id (Args) ;

Label → intlit :
| λ

Production	Predict Set
Stmt → Label id = Expr ;	**{id, intlit}**
Stmt → Label if Expr then Stmt ;	**{if, intlit}**
Stmt → Label read (IdList) ;	**{read, intlit}**
Stmt → Label id (Args) ;	**{id, intlit}**

We now will match a production p only if the next unmatched token is in p's predict set. We'll avoid trying productions that clearly won't work, so parsing will be faster.

But what is the predict set of a λ-production?

It can't be what's generated by λ (which is nothing!), so we'll define it as the tokens that can *follow* the use of a λ-production.

That is, Predict($A \rightarrow \lambda$) = Follow(A)

where (by definition)

Follow(A) = {a in V_t | $S \Rightarrow^+ \ldots Aa \ldots$}

In our example,
Follow(Label $\rightarrow \lambda$) ={ id, if, read }

(since these terminals can immediately follow uses of Label in the given productions).

Now let's parse

id (intlit) ;

Our start symbol is **Stmt** and the initial token is **id**.

id can predict

Stmt $\rightarrow$ **Label id = Expr ;**

id then predicts **Label** $\rightarrow \lambda$

The **id** is matched, but "**(**" doesn't match "**=**" so we backup and try a different production for **Stmt**.

id also predicts

Stmt $\rightarrow$ **Label id (Args) ;**

Again, **Label** $\rightarrow \lambda$ is predicted and used, and the input tokens can match the rest of the remaining production.

We had only one misprediction, which is better than before.

Now we'll rewrite the productions a bit to make predictions easier.

We remove the **Label** prefix from all the statement productions (now **intlit** won't predict all four productions).

We now have

Stmt $\rightarrow$ **Label BasicStmt**

BasicStmt $\rightarrow$ **id = Expr ;**

| **if Expr then Stmt ;**

| **read (IdList) ;**

| **id (Args) ;**

Label $\rightarrow$ **intlit :**

| λ

Now **id** predicts two different **BasicStmt** productions. If we rewrite these two productions into

BasicStmt $\rightarrow$ **id StmtSuffix**

StmtSuffix $\rightarrow$ **= Expr ;**

| **(Args) ;**

we no longer have any doubt over which production id predicts.

We now have

Production	Predict Set
Stmt → Label BasicStmt	**Not needed!**
BasicStmt → id StmtSuffix	**{id}**
BasicStmt → if Expr then Stmt ;	**{if}**
BasicStmt → read (IdList) ;	**{read}**
StmtSuffix → (Args) ;	**{ (}**
StmtSuffix → = Expr ;	**{ = }**
Label → intlit :	**{intlit}**
Label → λ	**{if, id, read}**

This grammar generates the same statements as our original grammar did, but now prediction never fails!

Whenever we must decide what production to use, the predict sets for productions with the same lefthand side are always disjoint.

Any input token will predict a unique production or no production at all (indicating a syntax error).

If we never mispredict a production, we never backup, so parsing will be fast and absolutely accurate!