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.

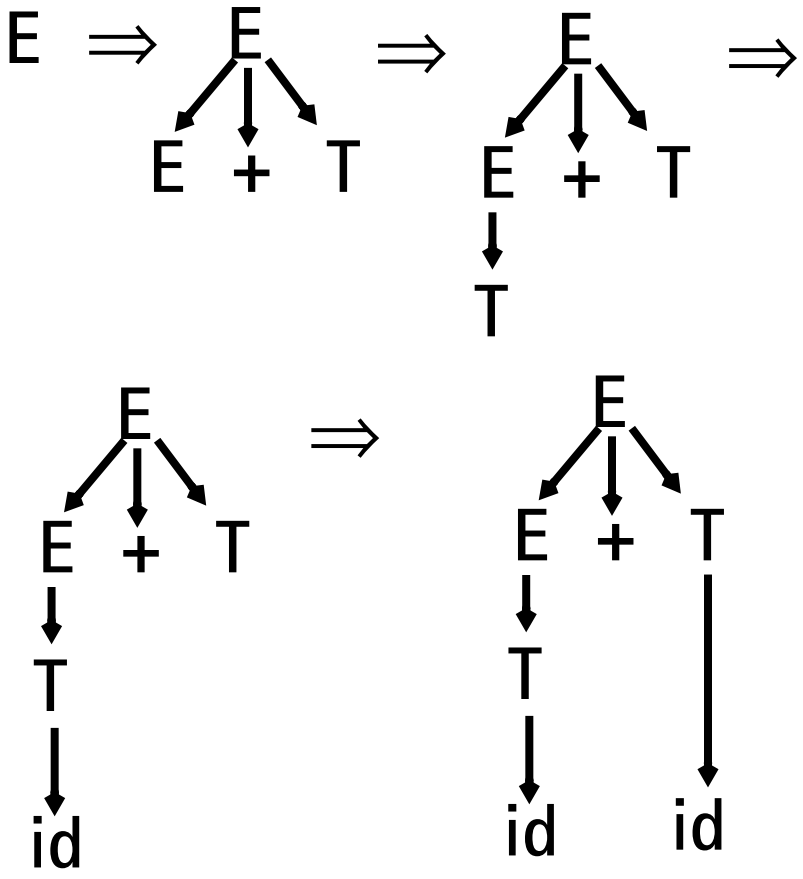
Top-down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins.

Consider

$\mathbf{E \rightarrow E + T \quad | \quad T}$

$\mathbf{T \rightarrow T * id \quad | \quad id}$

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.

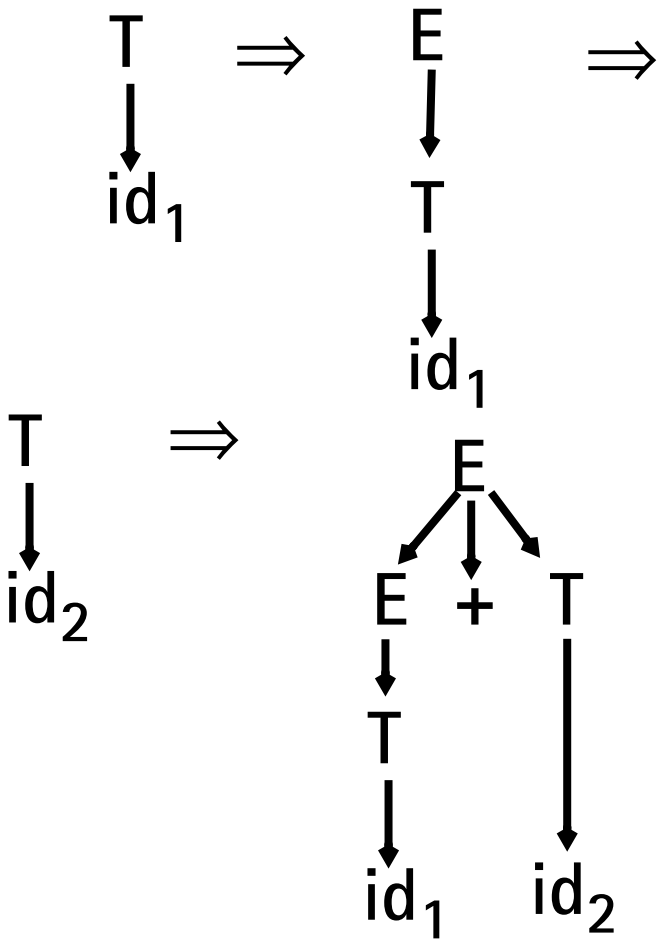
As the name suggests, bottom-up parsers discover the structure of a parse tree by beginning at its bottom (at the leaves of the tree which are terminal symbols) and determining the productions used to generate the leaves.

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!

LL(1) Grammars

A context-free grammar whose Predict sets are always disjoint (for the same non-terminal) is said to be *LL(1)*.

LL(1) grammars are ideally suited for top-down parsing because it is always possible to correctly predict the expansion of any non-terminal. No backup is ever needed.

Formally, let

First($X_1 ... X_n$) =

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

Follow(A) = $\{a \text{ in } V_t \mid S \Rightarrow^+ ... Aa ...\}$

Predict($A \rightarrow X_1 ... X_n$) =

If $X_1 ... X_n \Rightarrow^* \lambda$

Then First($X_1 ... X_n$) U Follow(A)

Else First($X_1 ... X_n$)

If some CFG, G, has the property that for all pairs of distinct productions with the same lefthand side,
$A \rightarrow X_1 ... X_n$ and $A \rightarrow Y_1 ... Y_m$
it is the case that

Predict($A \rightarrow X_1 ... X_n$) $\cap$

Predict($A \rightarrow Y_1 ... Y_m$) = ϕ

then G is LL(1).

LL(1) grammars are easy to parse in a top-down manner since predictions are always correct.

Example

Production	Predict Set
$S \rightarrow A\ a$	{b,d,a}
$A \rightarrow B\ D$	{b, d, a}
$B \rightarrow b$	{ b }
$B \rightarrow \lambda$	{d, a}
$D \rightarrow d$	{ d }
$D \rightarrow \lambda$	{ a }

Since the predict sets of both B productions and both D productions are disjoint, this grammar is LL(1).

Recursive Descent Parsers

An early implementation of top-down (LL(1)) parsing was recursive descent.

A parser was organized as a set of *parsing procedures*, one for each non-terminal. Each parsing procedure was responsible for parsing a sequence of tokens derivable from its non-terminal.

For example, a parsing procedure, A, when called, would call the scanner and match a token sequence derivable from A.

Starting with the start symbol's parsing procedure, we would then match the entire input, which must be derivable from the start symbol.

This approach is called recursive descent because the parsing procedures were typically *recursive*, and they *descended* down the input's parse tree (as top-down parsers always do).

Building A Recursive Descent Parser

We start with a procedure **Match**, that matches the current input token against a predicted token:

```
void Match(Terminal a) {
  if (a == currentToken)
     currentToken = Scanner();
  else SyntaxErrror();}
```

To build a parsing procedure for a non-terminal A, we look at all productions with A on the lefthand side:

$A \rightarrow X_1 ... X_n \mid A \rightarrow Y_1 ... Y_m \mid ...$

We use predict sets to decide which production to match (LL(1) grammars always have disjoint predict sets).

We match a production's righthand side by calling **Match** to

match terminals, and calling parsing procedures to match non-terminals.

The general form of a parsing procedure for

$A \rightarrow X_1 \ldots X_n \mid A \rightarrow Y_1 \ldots Y_m \mid \ldots$ is

```
void A() {
 if (currentToken in Predict(A→X1…Xn))
   for(i=1;i<=n;i++)
     if (X[i] is a terminal)
         Match(X[i]);
     else X[i]();
 else
 if (currentToken in Predict(A→Y1…Ym))
   for(i=1;i<=m;i++)
     if (Y[i] is a terminal)
         Match(Y[i]);
     else Y[i]();
 else
      // Handle other A →… productions
 else // No production predicted
     SyntaxError();
}
```

Usually this general form isn't used.

Instead, each production is "macro-expanded" into a sequence of **`Match`** and parsing procedure calls.

Example: CSX-Lite

Production	Predict Set
Prog → { Stmts } Eof	{
Stmts → Stmt Stmts	id if
Stmts → λ	}
Stmt → id = Expr ;	id
Stmt → if (Expr) Stmt	if
Expr → id Etail	id
Etail → + Expr	+
Etail → - Expr	-
Etail → λ	) ;

CSX-Lite Parsing Procedures

```
void Prog() {
  Match("{");
  Stmts();
  Match("}");
  Match(Eof);
}


void Stmts() {
 if (currentToken == id ||
     currentToken == if){
     Stmt();
     Stmts();
 } else {
     /* null */
}}

void Stmt() {
 if (currentToken == id){
     Match(id);
     Match("=");
     Expr();
     Match(";");
 } else {
     Match(if);
     Match("(");
     Expr();
     Match(")");
     Stmt();
}}
```

```
void Expr() {
  Match(id);
  Etail();
}


void Etail() {
 if (currentToken == "+") {
     Match("+");
     Expr();
 } else if (currentToken == "-"){
     Match("-");
     Expr();
 } else {
     /* null */
}}
```

Let's use recursive descent to parse

`{ a = b + c; } Eof`

We start by calling **`Prog()`** since this represents the start symbol.

Calls Pending	Remaining Input
`Prog()`	`{ a = b + c; } Eof`

`Match("{");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`{ a = b + c; } Eof`

`Stmts();`
`Match("}");`
`Match(Eof);`
	`a = b + c; } Eof`

`Stmt();`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`a = b + c; } Eof`

`Match(id);`
`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`a = b + c; } Eof`

Calls Pending	Remaining Input

`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`= b + c; } Eof`

`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c; } Eof`

`Match(id);`
`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c; } Eof`

`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`+ c; } Eof`

Calls Pending	Remaining Input

`Match("+");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`+ c; } Eof`

`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`c; } Eof`

`Match(id);`
`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`c; } Eof`

`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`; } Eof`

`/* null */`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`; } Eof`

Calls Pending	Remaining Input

`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`; } Eof`

`Stmts();`
`Match("}");`
`Match(Eof);`
	`} Eof`

`/* null */`
`Match("}");`
`Match(Eof);`
	`} Eof`

`Match("}");` `Match(Eof);`	`} Eof`
`Match(Eof);`	`Eof`
Done!	All input matched