Non-recursive predictive parsing

Observation:

> *Our recursive descent parser encodes state information in its run-time stack, or call stack.*

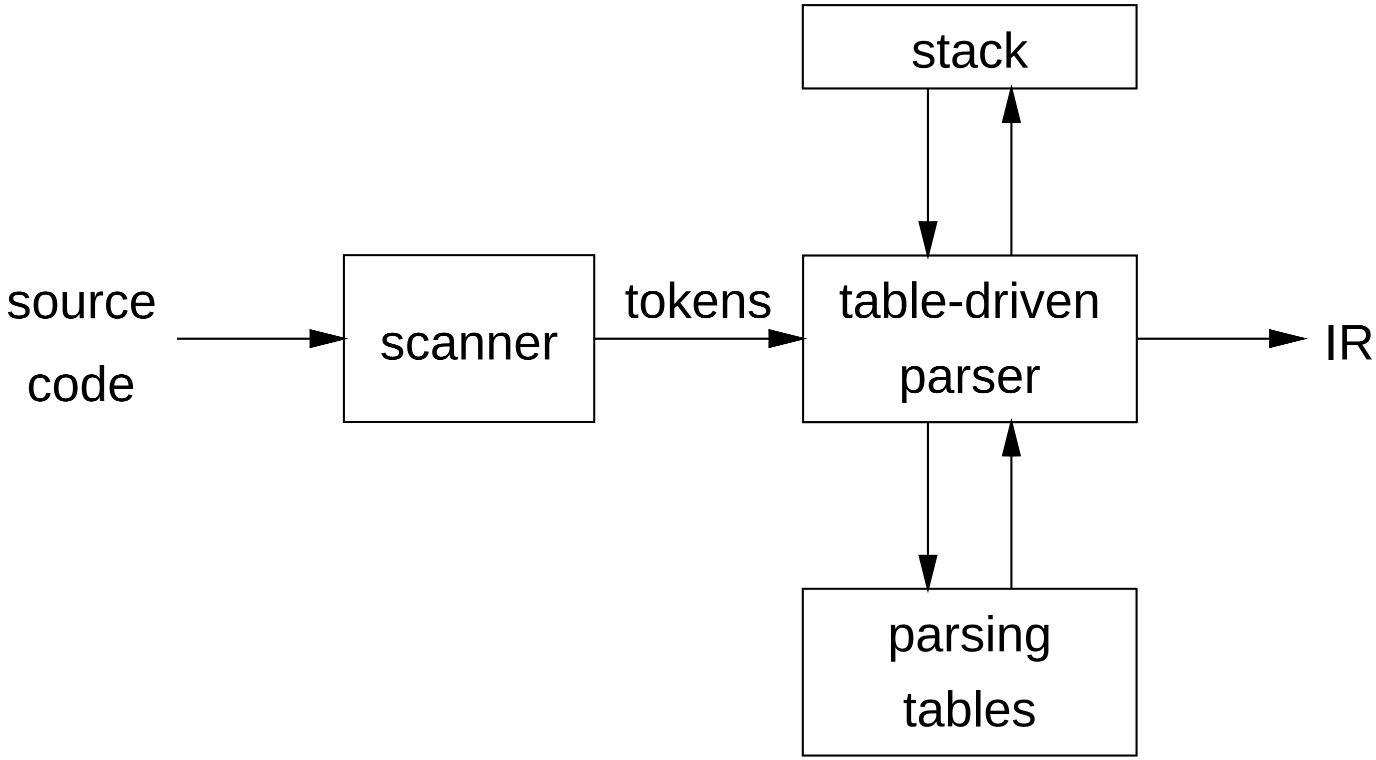
Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

This suggests other implementation methods:

- explicit stack, hand-coded parser
- stack-based, table-driven parser

Non-recursive predictive parsing

Now, a predictive parser looks like:

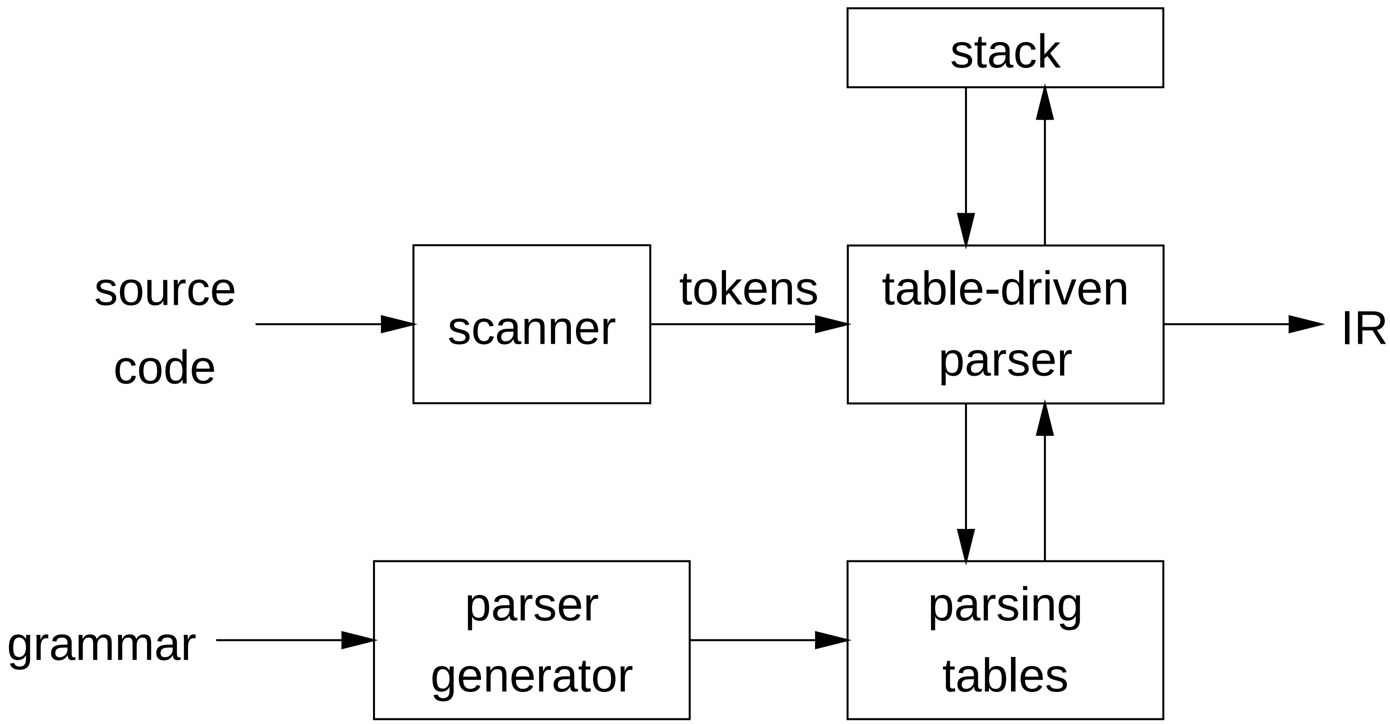


Rather than writing code, we build tables.

Building tables can be automated!

Table-driven parsers

A parser generator system often looks like:



This is true for both top-down (LL) and bottom-up (LR) parsers

Non-recursive predictive parsing

Input: a string w and a parsing table M for G

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         push Yk,Yk−1,···,Y1
      else error()
until X = EOF
```

Non-recursive predictive parsing

What we need now is a parsing table M.

Our expression grammar:

1	⟨goal⟩	::=	⟨expr⟩
2	⟨expr⟩	::=	⟨term⟩⟨expr′⟩
3	⟨expr′⟩	::=	+⟨expr⟩
4		|	−⟨expr⟩
5		|	ε
6	⟨term⟩	::=	⟨factor⟩⟨term′⟩
7	⟨term′⟩	::=	∗⟨term⟩
8		|	/⟨term⟩
9		|	ε
10	⟨factor⟩	::=	`num`
11		|	`id`

Its parse table:

	`id`	`num`	+	−	∗	/	$†
⟨goal⟩	1	1	–	–	–	–	–
⟨expr⟩	2	2	–	–	–	–	–
⟨expr′⟩	–	–	3	4	–	–	5
⟨term⟩	6	6	–	–	–	–	–
⟨term′⟩	–	–	9	9	7	8	9
⟨factor⟩	11	10	–	–	–	–	–

† we use $ to represent `EOF`

FIRST

For a string of grammar symbols α, define $\textsc{first}(\alpha)$ as:

- the set of terminal symbols that begin strings derived from α:
  $\{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in \textsc{first}(\alpha)$

$\textsc{first}(\alpha)$ contains the set of tokens valid in the initial position in α

To build $\textsc{first}(X)$:

1. If $X \in V_t$ then $\textsc{first}(X)$ is $\{X\}$
2. If $X \rightarrow \varepsilon$ then add ε to $\textsc{first}(X)$
3. If $X \rightarrow Y_1Y_2 \cdots Y_k$:
   (a) Put $\textsc{first}(Y_1) - \{\varepsilon\}$ in $\textsc{first}(X)$
   (b) $\forall i : 1 < i \leq k$, if $\varepsilon \in \textsc{first}(Y_1) \cap \cdots \cap \textsc{first}(Y_{i-1})$
   (i.e., $Y_1 \cdots Y_{i-1} \Rightarrow^* \varepsilon$)
   then put $\textsc{first}(Y_i) - \{\varepsilon\}$ in $\textsc{first}(X)$
   (c) If $\varepsilon \in \textsc{first}(Y_1) \cap \cdots \cap \textsc{first}(Y_k)$ then put ε in $\textsc{first}(X)$

   Repeat until no more additions can be made.

FOLLOW

For a non-terminal A, define $\text{FOLLOW}(A)$ as

> the set of terminals that can appear immediately to the right of A in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.

A terminal symbol has no FOLLOW set.

To build $\text{FOLLOW}(A)$:

1. Put \$ in $\text{FOLLOW}(\langle\text{goal}\rangle)$
2. If $A \rightarrow \alpha B \beta$:
   (a) Put $\text{FIRST}(\beta) - \{\varepsilon\}$ in $\text{FOLLOW}(B)$
   (b) If $\beta = \varepsilon$ (i.e., $A \rightarrow \alpha B$) or $\varepsilon \in \text{FIRST}(\beta)$ (i.e., $\beta \Rightarrow^* \varepsilon$) then put FOLLOW$(A)$ in FOLLOW(B)

   Repeat until no more additions can be made

LL(1) grammars

Previous definition

A grammar G is LL(1) iff. for all non-terminals A, each distinct pair of productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ satisfy the condition $\text{FIRST}(\beta) \bigcap \text{FIRST}(\gamma) = \phi$.

What if $A \Rightarrow^* \varepsilon$?

Revised definition

A grammar G is LL(1) iff. for each set of productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$:

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \ldots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint
2. If $\alpha_i \Rightarrow^* \varepsilon$ then $\text{FIRST}(\alpha_j) \bigcap \text{FOLLOW}(A) = \phi, \forall 1 \leq j \leq n, i \neq j$.

If G is ε-free, condition 1 is sufficient.

LL(1) grammars

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A ε–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example

- $S \rightarrow aS \mid a$ is not LL(1) because $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$
- $S \rightarrow aS'$
  $S' \rightarrow aS' \mid \varepsilon$
  accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. $\forall$ productions $A \rightarrow \alpha$:
   (a) $\forall a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
   (b) If $\varepsilon \in \text{FIRST}(\alpha)$:
      i. $\forall b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$
      ii. If $\$ \in \text{FOLLOW}(A)$ then add $A \rightarrow \alpha$ to $M[A, \$]$
2. Set each undefined entry of M to `error`

If $\exists M[A, a]$ with multiple entries then grammar is not LL(1).

Note: recall $a, b \in V_t$, so $a, b \neq \varepsilon$

Example

Our long-suffering expression grammar:

$$
\begin{array}{l|l}
S \rightarrow E & T \rightarrow FT' \\
E \rightarrow TE' & T' \rightarrow *T \mid /T \mid \varepsilon \\
E' \rightarrow +E \mid -E \mid \varepsilon & F \rightarrow \texttt{id} \mid \texttt{num}
\end{array}
$$

	FIRST	FOLLOW
S	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E'	$\{\varepsilon, +, -\}$	$\{\$\}$
T	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, \$\}$
T'	$\{\varepsilon, *, /\}$	$\{+, -, \$\}$
F	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, *, /, \$\}$
id	$\{\texttt{id}\}$	–
num	$\{\texttt{num}\}$	–
$*$	$\{*\}$	–
$/$	$\{/\}$	–
$+$	$\{+\}$	–
$-$	$\{-\}$	–

	id	num	$+$	$-$	$*$	$/$	$\$$
S	$S \rightarrow E$	$S \rightarrow E$	–	–	–	–	–
E	$E \rightarrow TE'$	$E \rightarrow TE'$	–	–	–	–	–
E'	–	–	$E' \rightarrow +E$	$E' \rightarrow -E$	–	–	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	$T \rightarrow FT'$	–	–	–	–	–
T'	–	–	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$	$T' \rightarrow *T$	$T' \rightarrow /T$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \texttt{id}$	$F \rightarrow \texttt{num}$	–	–	–	–	–

Building the tree

Again, we insert code at the right points:

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← root node
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
         pop and fill in node
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         pop node for X
         build node for each child and
         make it a child of node for X
         push nk,Yk,nk−1,Yk−1,···,n1,Y1
      else error()
until X = EOF
```

A grammar that is not LL(1)

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmt} \rangle \\ & | & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmt} \rangle \texttt{ else } \langle \text{stmt} \rangle \\ & | & \ldots \end{array}$$

Left-factored:

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmt} \rangle \ \langle \text{stmt}' \rangle \mid \ldots \\ \langle \text{stmt}' \rangle & ::= & \texttt{else } \langle \text{stmt} \rangle \mid \varepsilon \end{array}$$

Now, $\text{FIRST}(\langle \text{stmt}' \rangle) = \{\varepsilon, \texttt{else}\}$

Also, $\text{FOLLOW}(\langle \text{stmt}' \rangle) = \{\texttt{else}, \$\}$

But, $\text{FIRST}(\langle \text{stmt}' \rangle) \bigcap \text{FOLLOW}(\langle \text{stmt}' \rangle) = \{\texttt{else}\} \neq \phi$

On seeing `else`, conflict between choosing

$$\langle \text{stmt}' \rangle \ ::= \ \texttt{else } \langle \text{stmt} \rangle \quad \text{and} \quad \langle \text{stmt}' \rangle \ ::= \ \varepsilon$$

$\Rightarrow$ grammar is not LL(1)!

The fix:

Put priority on $\langle \text{stmt}' \rangle \ ::= \ \texttt{else } \langle \text{stmt} \rangle$ to associate `else` with closest previous `then`.

Error recovery

Key notion:

- For each non-terminal, construct a set of terminals on which the parser can synchronize
- When an error occurs looking for A, scan until an element of $\text{SYNCH}(A)$ is found

Building SYNCH:

1. $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNCH}(A)$
2. place keywords that start statements in $\text{SYNCH}(A)$
3. add symbols in $\text{FIRST}(A)$ to $\text{SYNCH}(A)$

If we can't match a terminal on top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

(i.e., $\text{SYNCH}(a) = V_t - \{a\}$)