

Non-recursive predictive parsing

Observation:

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

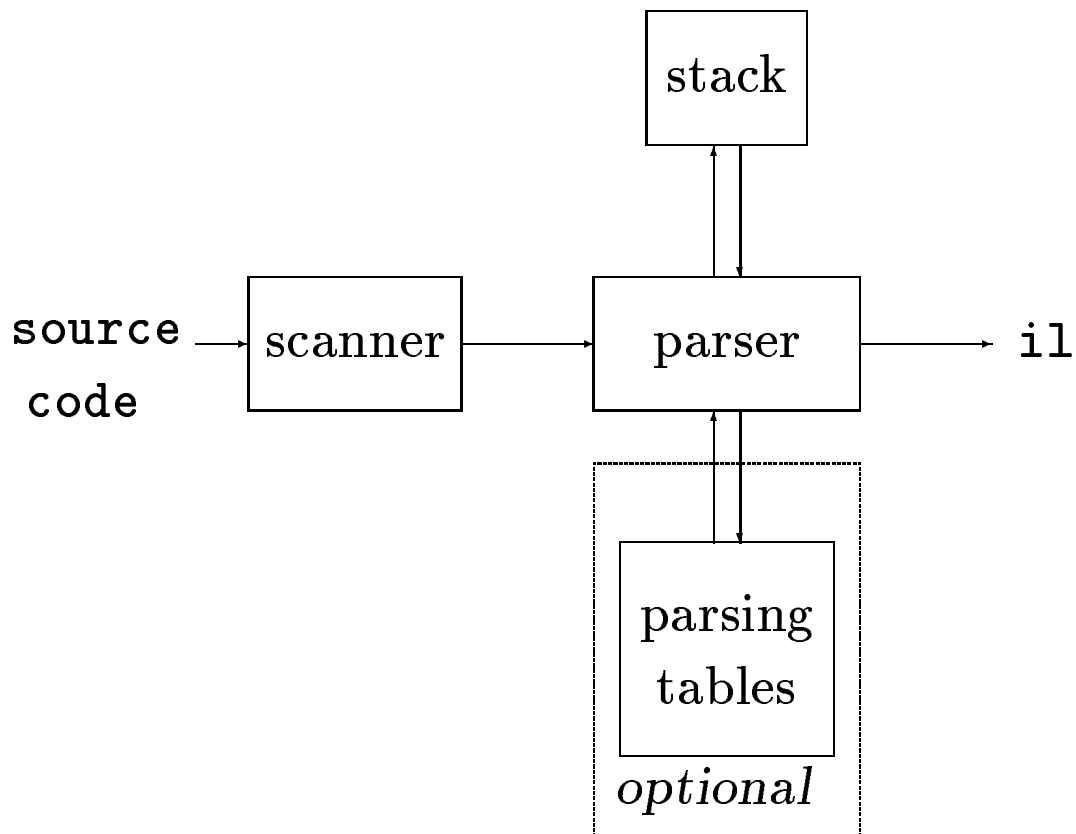
Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

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:

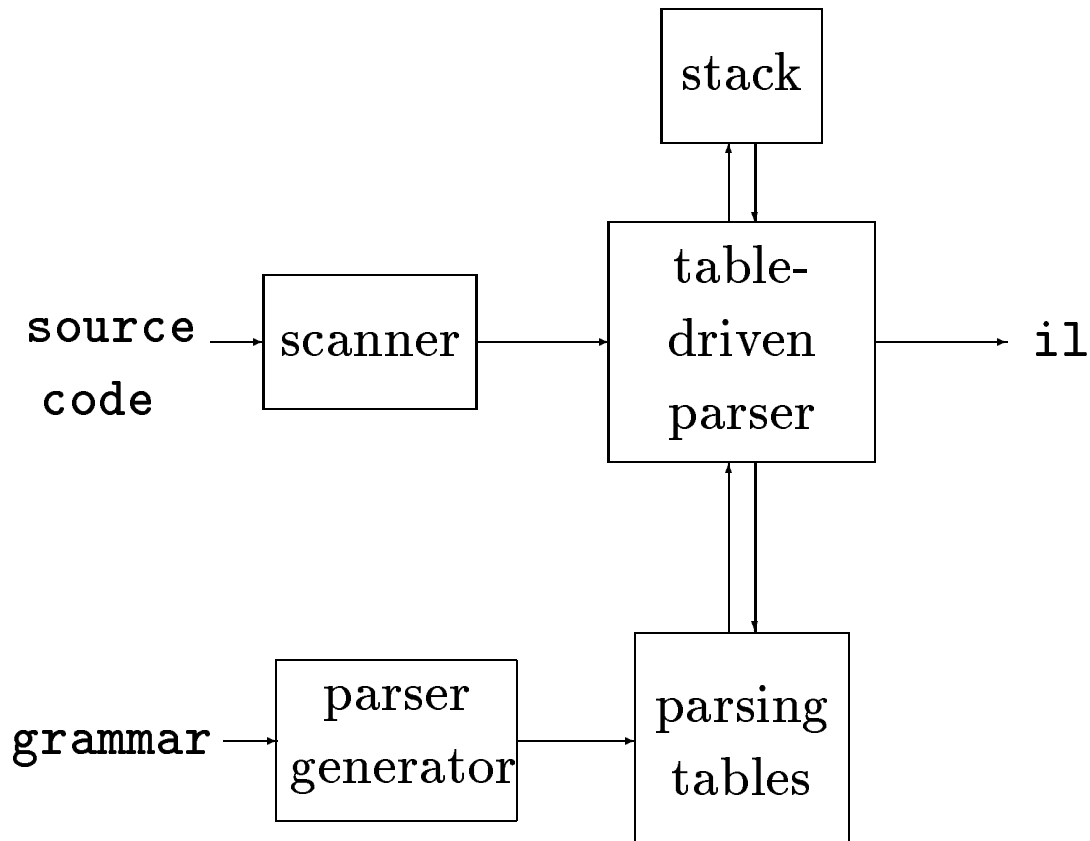


Rather than writing code, we build tables.

Building tables can be automated!

Table-driven parsers

A parser generator system often looks like:



This is true for both top down and bottom up parsers

$LL(1)$: left to right, leftmost derivation,
lookahead(1)

$LR(1)$: left to right, reverse rightmost derivation,
lookahead(1)

Table-driven parsing algorithm

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

```
tos ← 0
Stack[tos++] ← eof
Stack[tos++] ← Start Symbol
token ← next_token()

X ← Stack[tos]
repeat
    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  $M[X, token] = X \rightarrow Y_1 Y_2 \cdots Y_k$  then
            pop X
            push  $Y_k, Y_{k-1}, \cdots, Y_1$ 
        else error()

    X ← Stack[tos]
until X = eof
```

Aho, Sethi, and Ullman, Algorithm 4.3

The grammar and its table

Our long-suffering expression grammar

$$\begin{aligned}
 \langle \text{goal} \rangle & ::= \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle & ::= \langle \text{term} \rangle \langle \text{expr}' \rangle \\
 \langle \text{expr}' \rangle & ::= + \langle \text{expr} \rangle \\
 & \quad | - \langle \text{expr} \rangle \\
 & \quad | \epsilon \\
 \langle \text{term} \rangle & ::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\
 \langle \text{term}' \rangle & ::= * \langle \text{term} \rangle \\
 & \quad | / \langle \text{term} \rangle \\
 & \quad | \epsilon \\
 \langle \text{factor} \rangle & ::= \text{num} \\
 & \quad | \text{id}
 \end{aligned}$$

LL(1) parse table

	id	num	+	-	*	/	eof
$\langle \text{goal} \rangle$	$g \rightarrow e$	$g \rightarrow e$	-	-	-	-	-
$\langle \text{expr} \rangle$	$e \rightarrow te'$	$e \rightarrow te'$	-	-	-	-	-
$\langle \text{expr}' \rangle$	-	-	$e' \rightarrow +e$	$e' \rightarrow -e$	-	-	$e' \rightarrow \epsilon$
$\langle \text{term} \rangle$	$t \rightarrow ft'$	$t \rightarrow ft'$	-	-	-	-	-
$\langle \text{term}' \rangle$	-	-	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow *t$	$t' \rightarrow /t$	$t' \rightarrow \epsilon$
$\langle \text{factor} \rangle$	$f \rightarrow \text{id}$	$f \rightarrow \text{num}$	-	-	-	-	-

The FIRST set

For a string of grammar symbols α , define $\text{FIRST}(\alpha)$ as

- the set of terminal symbols that begin strings derived from α
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

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

To build $\text{FIRST}(X)$:

1. if X is a terminal, $\text{FIRST}(X)$ is $\{X\}$
2. if $X ::= \epsilon$, then $\epsilon \in \text{FIRST}(X)$
3. if $X ::= Y_1Y_2 \cdots Y_k$ then put $\text{FIRST}(Y_1)$ in $\text{FIRST}(X)$
4. if X is a non-terminal and $X ::= Y_1Y_2 \cdots Y_k$, then $a \in \text{FIRST}(X)$ if $a \in \text{FIRST}(Y_i)$ and $\epsilon \in \text{FIRST}(Y_j)$ for all $1 \leq j < i$
(If $\epsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$)

Our example grammar

1		$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2		$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle \langle \text{expr}' \rangle$
3		$\langle \text{expr}' \rangle$	$::=$	$+$ $\langle \text{expr} \rangle$
4				$-$ $\langle \text{expr} \rangle$
5				ϵ
6		$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle \langle \text{term}' \rangle$
7		$\langle \text{term}' \rangle$	$::=$	$*$ $\langle \text{term} \rangle$
8				$/$ $\langle \text{term} \rangle$
9				ϵ
10		$\langle \text{factor} \rangle$	$::=$	num
11				id

The FIRST construction

<i>rule</i>	1	2	3	4	<i>FIRST</i>
<i>goal</i>	–	–	num, id	–	{num, id}
<i>expr</i>	–	–	num, id	–	{num, id}
<i>expr'</i>	–	ϵ	+, -	–	{ ϵ , +, -}
<i>term</i>	–	–	num, id	–	{num, id}
<i>term'</i>	–	ϵ	*, /	–	{ ϵ , *, /}
<i>factor</i>	–	–	num, id	–	{num, id}
num	num	–	–	–	{num}
id	id	–	–	–	{id}
+	+	–	–	–	{+}
-	-	–	–	–	{-}
*	*	–	–	–	{*}
/	/	–	–	–	{/}

The FOLLOW set

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 $\text{FOLLOW}(X)$:

1. place **eof** in $\text{FOLLOW}(\langle\text{goal}\rangle)$
2. if $A ::= \alpha B \beta$, then put $\{\text{FIRST}(\beta) - \epsilon\}$ in $\text{FOLLOW}(B)$
3. if $A ::= \alpha B$ then put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$
4. if $A ::= \alpha B \beta$ and $\epsilon \in \text{FIRST}(\beta)$, then put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$

The FOLLOW construction

<i>rule</i>	1	2	3	4	<i>FOLLOW</i>
<i>goal</i>	eof	–	–	–	{eof}
<i>expr</i>	–	–	eof	–	{eof}
<i>expr'</i>	–	–	eof	–	{eof}
<i>term</i>	–	+, –	–	eof	{eof, +, –}
<i>term'</i>	–	–	eof, +, –	–	{eof, +, –}
<i>factor</i>	–	*, /	–	eof, +, –	{eof, +, –, *, /}

LL(1) parse table construction

Input: a grammar G

Method

1. \forall production $A ::= \alpha$, perform steps 2–4
2. \forall terminal a in $\text{FIRST}(\alpha)$, add $A ::= \alpha$ to $M[A, a]$
3. if $\epsilon \in \text{FIRST}(\alpha)$, add $A ::= \alpha$ to $M[A, b]$
 \forall terminal $b \in \text{FOLLOW}(A)$
4. if $\epsilon \in \text{FIRST}(\alpha)$ and $\text{eof} \in \text{FOLLOW}(A)$, add
 $A ::= \alpha$ to $M[A, \text{eof}]$
5. set each undefined entry of M to **error**

If this fails, the grammar is not $LL(1)$

Aho, Sethi, and Ullman, Algorithm 4.4

LL(1) parse table for example

	id	num	+	-	*	/	eof
$\langle \text{goal} \rangle$	$g \rightarrow e$	$g \rightarrow e$	-	-	-	-	-
$\langle \text{expr} \rangle$	$e \rightarrow te'$	$e \rightarrow te'$	-	-	-	-	-
$\langle \text{expr}' \rangle$	-	-	$e' \rightarrow +e$	$e' \rightarrow -e$	-	-	$e' \rightarrow \epsilon$
$\langle \text{term} \rangle$	$t \rightarrow ft'$	$t \rightarrow ft'$	-	-	-	-	-
$\langle \text{term}' \rangle$	-	-	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow *t$	$t' \rightarrow /t$	$t' \rightarrow \epsilon$
$\langle \text{factor} \rangle$	$f \rightarrow \text{id}$	$f \rightarrow \text{num}$	-	-	-	-	-

Symbol	FIRST	FOLLOW
$\langle \text{goal} \rangle$	{ id, number }	{ eof }
$\langle \text{expr} \rangle$	{ id, number }	{ eof }
$\langle \text{expr}' \rangle$	{ ϵ , +, - }	{ eof }
$\langle \text{term} \rangle$	{ id, number }	{ eof, +, - }
$\langle \text{term}' \rangle$	{ ϵ , *, / }	{ eof, +, - }
$\langle \text{factor} \rangle$	{ id, number }	{ eof, +, -, *, / }
+	{ + }	—
-	{ - }	—
*	{ * }	—
/	{ / }	—
id	{ id }	—
number	{ number }	—

Building the tree

Insert some code at the appropriate points

```
tos ← 0
Stack[tos++] ← eof
Stack[tos++] ← root node
Stack[tos++] ← Start Symbol
token ← next_token()

X ← Stack[tos]
repeat
  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  $M[X, token] = X \rightarrow Y_1 Y_2 \cdots Y_k$  then
      pop X
      pop node for X
      build node for each child and
      make it a child of node for X
      push  $n_k, Y_k, n_{k-1}, Y_{k-1}, \cdots, n_1, Y_1$ 
    else error()
until X = eof
```

LL(1) grammars

Features

- input parsed from left to right
- leftmost derivation
- one token lookahead

Definition

A grammar G is $LL(1)$ if and only if, for all non-terminals A , each distinct pair of productions $A ::= \beta$ and $A ::= \gamma$ satisfy the condition $\text{FIRST}(\beta) \cap \text{FIRST}(\gamma) = \emptyset$

A grammar G is $LL(1)$ if and only if for each set of productions $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \dots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint
2. if $\alpha_i \Rightarrow^* \epsilon$, then $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$, for all $1 \leq j \leq n, i \neq j$.

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

LL(1) grammars

Provable facts about *LL(1)* grammars:

- no left recursive grammar is *LL(1)*
- no ambiguous grammar is *LL(1)*
- *LL(1)* parsers operate in linear time
- an ϵ -free grammar where each alternative expansion for A begins with a distinct terminal is a *simple LL(1)* grammar

Not all grammars are *LL(1)*

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

LL grammars

LL(1) grammars

- may need to rewrite grammar
(left recursion, left factoring)
- resulting grammar larger, less maintainable

LL(k) grammars

- k -token lookahead
- more powerful than *LL(1)* grammars
- example:

$S ::= ac \mid abc$ is *LL(2)*

Not all grammars are *LL(k)*

- example:
 $S ::= a^i b^j$ where $i \geq j$
- equivalent to dangling else problem
- problem - must choose production after k tokens of lookahead

Bottom-up parsers avoid this problem