Non-recursive predictive parsing

Observation:

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

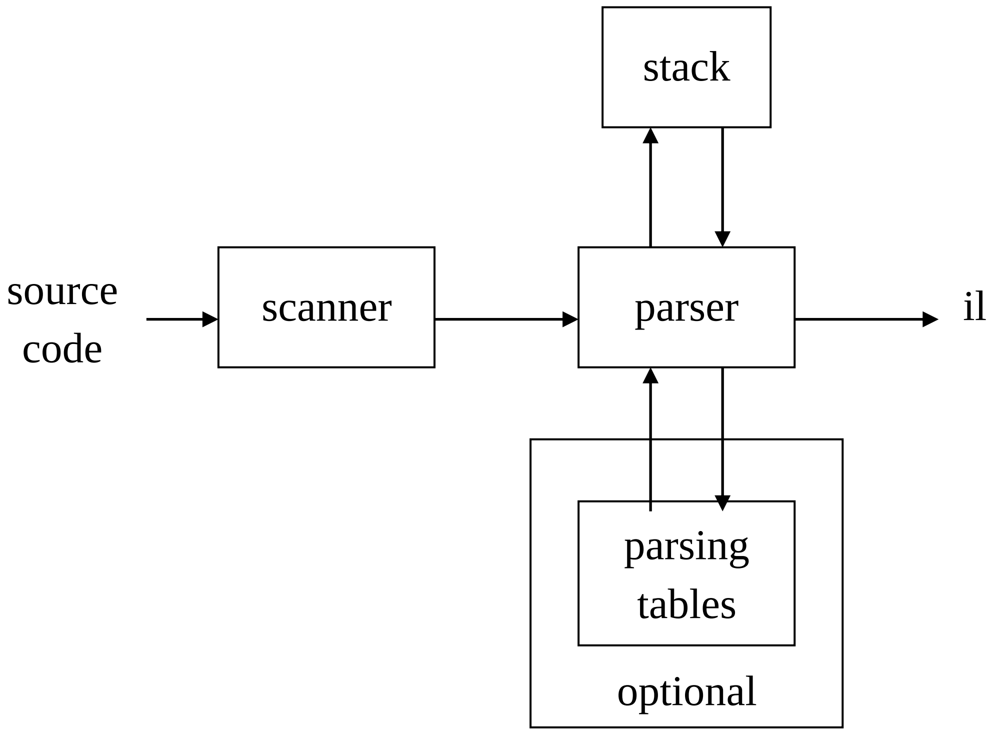
Using recursive procedure calls to implement a stack abstraction may not particularly efficient.

This suggests other implementation methods.

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

Non-recursive predictive parsing

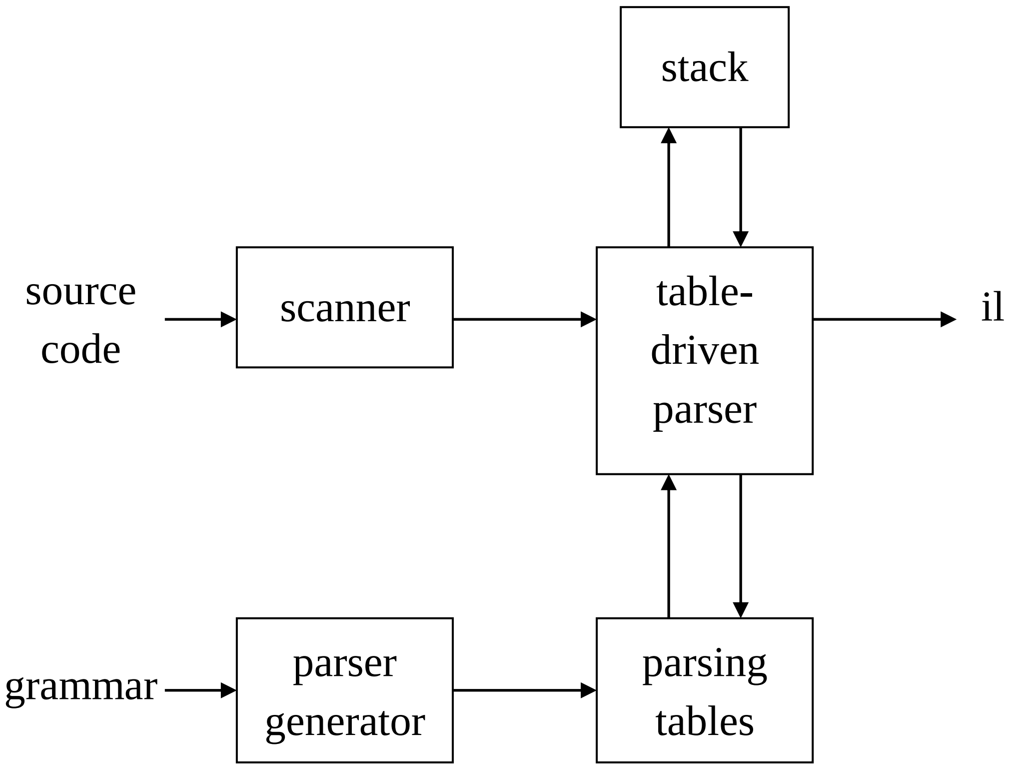
Now, a predictive parser looks like:



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 scan, leftmost derivation, one lookahead symbol

LR(1): left to right scan, rightmost derivation, one lookahead symbol

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 → Y1Y2…Yk then
         pop X
         push Yk, Yk-1, …, Y1
      else error()
   X ← Stack[tos]
until X = eof
```

Aho, Sethi, and Ullman, Algorithm 4.34

The grammar and its table

Our long-suffering expression grammar

<goal> ::= <expr>

<expr> ::= <term> <expr'>

<expr'> ::= + <term> <expr'>

| – <term> <expr'>

| ε

<term> ::= <factor> <term'>

<term'> ::= * <term'>

| / <term'>

| ε

<factor> ::= number

| id

LL(1) parsing table

	id	num	+	–	*	/	eof
<goal>	$g \to e$	$g \to e$	-	-	-	-	-
<expr>	$e \to te'$	$e \to te'$	-	-	-	-	-
<expr'>	-	-	$e' \to +e$	$e' \to -e$	-	-	$e' \to e$
<term>	$t \to f\,t'$	$t \to f\,t'$	-	-	-	-	-
<term'>	-	-	$t' \to \varepsilon$	$t' \to \varepsilon$	$t' \to *t$	$t' \to /t$	$t' \to \varepsilon$
<factor>	$f \to id$	$f \to num$					

The FIRST set

For a string of grammar symbols α, define FIRST(α) as

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

FIRST(α) contains the set of tokens valid in the first position of α

To build FIRST(X):

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

The FIRST construction

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

The FOLLOW set

For a non-terminal A, define 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 legally appear after it

A terminal symbol has no FOLLOW set

To build FOLLOW(X)

1. place eof in FOLLOW(<goal>)
2. if A ::= $\alpha B\beta$, then put {FIRST(β) - ε} in FOLLOW(B)
3. if A::= αB then put FOLLOW(A) in FOLLOW(B)
4. if A::= $\alpha B\beta$ and $\varepsilon \in$ FIRST(β), then put FOLLOW(A) in FOLLOW(B)

The FOLLOW construction

rule	1	2	3	4	FOLLOW
goal	eof	-	-	-	{eof}
expr	-	-	eof	-	{eof}
expr’	-	-	eof	-	{eof}
term	-	+,–	-	eof	{eof, +,–}
term’	-	-	eof, +,–	-	{ε,*, /}
factor	-	*,/	-	eof, +,–	{ eof, +,–,*,/}

LL(1) parsing table construction

Input: a grammar G

Method

1. $\forall$ production A ::= α, perform steps 2-4
2. $\forall$ terminal *a* in FIRST(α), add A ::= α to M[A, a]
3. if $\varepsilon \in$ FIRST(α), for any terminal b $\in$ FOLLOW(A)
   add A ::= α to M[A, b]
4. if $\varepsilon \in$ FIRST(α), and eof $\in$ FOLLOW(A),
   add A ::= α to M[A, eof]
5. set each undefined entry of M to error

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

Aho, Sethi, and Ullman, Algorithm 4.31

LL(1) parsing table for example

	id	num	+	–	*	/	eof
<goal>	$g \rightarrow e$	$g \rightarrow e$	-	-	-	-	-
<expr>	$e \rightarrow te'$	$e \rightarrow te'$	-	-	-	-	-
<expr'>	-	-	$e' \rightarrow +e$	$e' \rightarrow -e$	-	-	$e' \rightarrow e$
<term>	$t \rightarrow f\,t'$	$t \rightarrow f\,t'$	-	-	-	-	-
<term'>	-	-	$t' \rightarrow \varepsilon$	$t' \rightarrow \varepsilon$	$t' \rightarrow *t$	$t' \rightarrow /t$	$t' \rightarrow \varepsilon$
<factor>	$f \rightarrow id$	$f \rightarrow num$					

Symbol	FIRST	FOLLOW
<goal>	{id, number}	{eof}
<expr>	{id, number}	{eof}
<expr'>	{ε, +, –}	{eof}
<term>	{id, number}	{eof, +, –}
<term'>	{ε, *, /}	{eof, +, –}
<factor>	{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 → 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()
   X ← Stack[tos]
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 ::= β and A ::= Υ satisfy the condition FIRST(β) $\cap$ FIRST(Υ) = $\emptyset$

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

1. FIRST(α_1), FIRST(α_2), …, FIRST(α_n) are all pairwise disjoint
2. if $\alpha_i \Rightarrow^* \varepsilon$, then FIRST($\alpha_j$) $\cap$ FOLLOW(A) = $\emptyset$, for all $1 \le j \le n$, $i \ne j$.

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 | a
  is not LL(1)
  FIRST(aS) = FIRST(a) = {a}

- S ::= aS'
  S' ::= aS' | ε
  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) gammars
- example:
  S ::= ac | 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