LL(1) grammars

Features

- input parsed from left to right — $\underline{\mathbf{L}}$L(1)
- leftmost derivation — L$\underline{\mathbf{L}}$(1)
- one token lookahead — LL($\underline{\mathbf{1}}$)

Definition

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. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \cdots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint, and
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 has no ϵ–productions, condition 1 is sufficient.

Note:

- LL(1) grammars allows predictive (deterministic) top–down parsing.
- LL(1) is a property of a grammar, not a language.

LL(1) grammars

Not all grammars are *LL(1)*

- $S ::= aS \mid a$

  is not *LL(1)*

  $\mathrm{FIRST}(aS) = \mathrm{FIRST}(a) = \{a\}$

- $$\begin{array}{l} S ::= aS' \\ S' ::= aS' \mid \epsilon \end{array}$$

  accepts the same language and is *LL(1)*
  (left factoring)

Provable facts about *LL(1)* grammars:

- no left recursive grammar is *LL(1)*
  Why? – Example: A := A a | b
- no ambiguous grammar is *LL(1)*
  Why?
- *LL(1)* parsers operate in linear time

LL(1) grammars

What to do if your grammar is not LL(1)?

- may need to rewrite grammar

  "bag of tricks":

  - left recursion removal
  - disambiguation
  - left factoring
- resulting grammar larger, less maintainable

Generality

Question:

> By *eliminating left recursion*, *disambiguation*, and *left factoring*, can we transform an arbitrary context free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

> Given a context free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many *context free languages* do not have an LL(k) grammar.

$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$

Example — Grammar with its parse table

Our long-suffering expression grammar

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

LL(1) parse table

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

The FIRST set

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

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

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

To build $\mathrm{FIRST}(X)$ for all grammar symbols X:

1. `if` X `is a terminal,` $\mathrm{FIRST}(X)$ `is` $\{X\}$
2. `if` $X ::= \epsilon$`, then` $\epsilon \in \mathrm{FIRST}(X)$

3. `iterate until no more terminals or` ϵ `can be added to any` $\mathrm{FIRST}(X)$`:`

`if` $X ::= Y_1 Y_2 \cdots Y_k$ `then`
$a \in \mathrm{FIRST}(X)$ `if` $a \in \mathrm{FIRST}(Y_i)$
`and` $\epsilon \in \mathrm{FIRST}(Y_j)$ `for all` $1 \leq j < i$
$\epsilon \in \mathrm{FIRST}(X)$ `if` $\epsilon \in \mathrm{FIRST}(Y_i)$ `for all` $1 \leq i \leq k$

`end iterate`

(If $\epsilon \notin \mathrm{FIRST}(Y_1)$, then $\mathrm{FIRST}(Y_i)$ 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		$\mid$	`-` $\langle\text{expr}\rangle$
5		$\mid$	ϵ
6	$\langle\text{term}\rangle$	$::=$	$\langle\text{factor}\rangle\ \langle\text{term}'\rangle$
7	$\langle\text{term}'\rangle$	$::=$	`*` $\langle\text{term}\rangle$
8		$\mid$	`/` $\langle\text{term}\rangle$
9		$\mid$	ϵ
10	$\langle\text{factor}\rangle$	$::=$	**`num`**
11		$\mid$	**`id`**

The FIRST construction

step	1	2	3	*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}
+	+	–	–	{+}
-	-	–	–	{-}
*	*	–	–	{*}
/	/	–	–	{/}

To build $\text{FIRST}(\alpha)$ for $\alpha = X_1X_2\cdots X_n$:

- $a \in \text{FIRST}(\alpha)$ `if` $a \in \text{FIRST}(X_i)$ `and` $\epsilon \in \text{FIRST}(X_j)$ `for all` $1 \leq j < i$
- $\epsilon \in \text{FIRST}(\alpha)$ `if` $\epsilon \in \text{FIRST}(X_i)$ `for all` $1 \leq i \leq n$

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 $\text{FOLLOW}(X)$ for non-terminal X:

1. `place eof in` FOLLOW(`⟨goal⟩`)

  <u>`iterate`</u> <u>`until`</u> `no more terminals or` ϵ `can be added to any` FOLLOW(X)`:`

2. `if` $A ::= \alpha B \beta$ `then` `put` $\{\text{FIRST}(\beta) - \epsilon\}$ `in` FOLLOW(B)

3. `if` $A ::= \alpha B$ `then` `put` FOLLOW(A) `in` FOLLOW(B)

4. `if` $A ::= \alpha B \beta$ `and` $\epsilon \in$ FIRST(β) `then` `put` FOLLOW(A) `in` FOLLOW(B)

  <u>`end`</u> <u>`iterate`</u>

step	1	2	3	4	*FOLLOW*
goal	`eof`	–	–	–	{`eof`}
expr	–	–	`eof`	–	{`eof`}
expr′	–	–	`eof`	–	{`eof`}
term	–	`+,-`	–	`eof`	{`eof,+,-`}
term′	–	–	`eof,+,-`	–	{`eof,+,-`}
factor	–	`*,/`	–	`eof,+,-`	{`eof,+,-,*,/`}

Recall: Predictive LL(1) Parsing

Key Property:

Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$, and
- if $\alpha \Rightarrow^* \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Analogue case for $\beta \Rightarrow^* \epsilon$. Note: due to first condition, at most one of α or β can derive ϵ.

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 $eof \in \text{FOLLOW}(A)$, add $A ::= \alpha$ to $M[A, \texttt{eof}]$
5. set each undefined entry of M to `error`

Construction fails if any $M[A, a]$ or $M[A, \texttt{eof}]$ has multiple entries. In that case, the grammar is not *LL(1)*.

Aho, Sethi, and Ullman, Algorithm 4.4

LL(1) parse table for example

	`id`	`num`	`+`	`-`	`*`	`/`	`eof`
⟨goal⟩	$g \rightarrow e$	$g \rightarrow e$	–	–	–	–	–
⟨expr⟩	$e \rightarrow te'$	$e \rightarrow te'$	–	–	–	–	–
⟨expr′⟩	–	–	$e' \rightarrow \texttt{+}e$	$e' \rightarrow \texttt{-}e$	–	–	$e' \rightarrow \epsilon$
⟨term⟩	$t \rightarrow ft'$	$t \rightarrow ft'$	–	–	–	–	–
⟨term′⟩	–	–	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow \texttt{*}t$	$t' \rightarrow \texttt{/}t$	$t' \rightarrow \epsilon$
⟨factor⟩	$f \rightarrow \texttt{id}$	$f \rightarrow \texttt{num}$	–	–	–	–	–

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` }	—

A parser generator system often looks like:

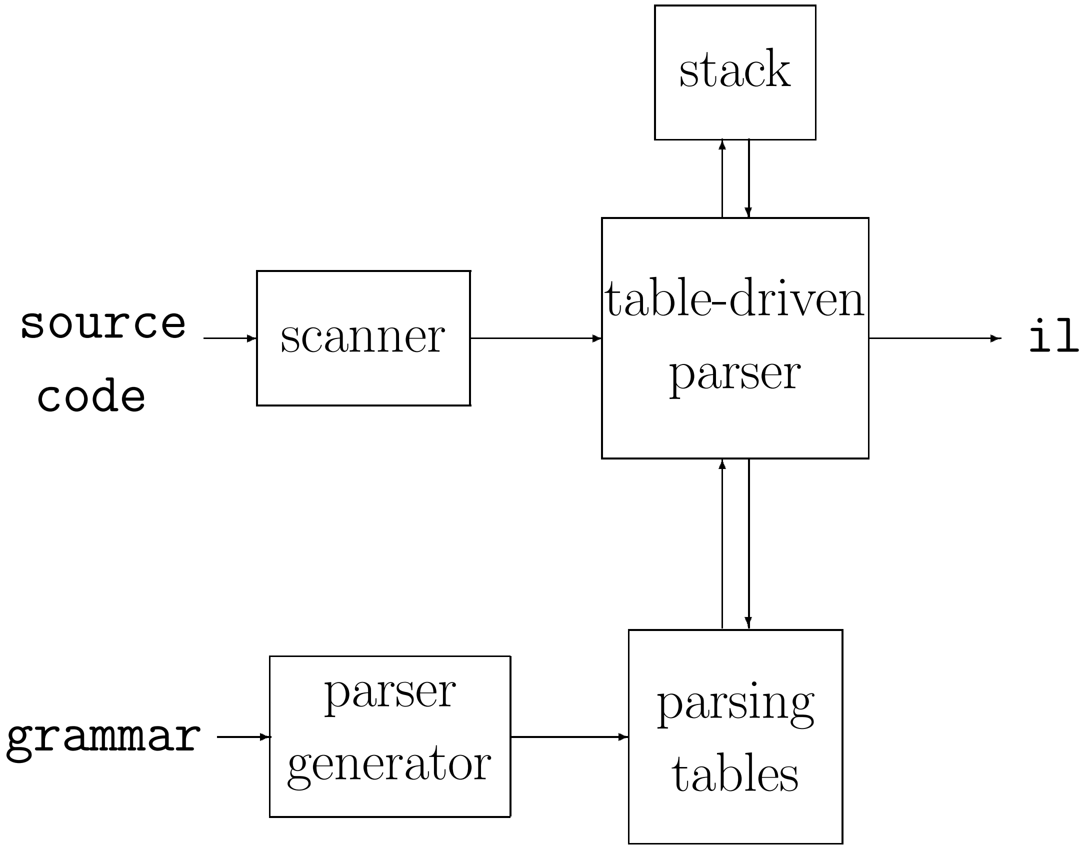

Table-driven predictive parsing algorithm

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

```
push eof
push Start Symbol
token ← next_token()

X ← top-of-stack
repeat
   if X is a terminal 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 ← top-of-stack
until X = eof

if token ≠ eof then error()
```

Aho, Sethi, and Ullman, Algorithm 4.3

Recursive Descent Parsing

Now, we can produce a simple recursive descent parser from our favorite LL(1) expression grammar.

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated **parsing procedure** that can recognize any sequence of tokens generated by that non–terminal.
- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful "computation" (syntax directed translation).

Recursive Descent Parsing

Our favorite example grammar

```
goal:
  token ← next_token();
  if (expr() = ERROR | token ≠ EOF) then
    return ERROR;
  else return OK;

expr:
  if (term() = ERROR) then
    return ERROR;
  else return expr_prime();

expr_prime:
  if (token = PLUS) then
    token ← next_token();
    return expr();
  else if (token = MINUS) then
    token ← next_token();
    return expr();
  else return OK;
```

Recursive Descent Parsing (Cont.)

```
term:
  if (factor() = ERROR) then
    return ERROR;
  else return term_prime();

term_prime:
  if (token = MULT) then
    token ← next_token();
    return term();
  else if (token = DIV) then
    token ← next_token();
    return term();
  else return OK;

factor:
  if (token = NUM) then
    token ← next_token();
    return OK;
  else if (token = ID) then
    token ← next_token();
    return OK;
  else return ERROR;
```

Error recovery

Panic-mode error recovery:

- for each non-terminal, construct a set of terminals on which the parser can synchronize.
- When an error occurs with non-terminal A on top of stack, scan until an element of $\text{SYNCH}(A)$ is found, then pop A and continue.

Heuristic for building SYNCH:

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

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

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

LL(k) grammars

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)*

- problem - must choose production after k tokens of lookahead
- example:

  Set of productions of form: $S ::= a^i b^j \quad \text{for } i \geq j$

Bottom-up parsers avoid this problem