Parsing

The Parsing Process

The role of the parsing process is to reconstruct a derivation by which a given Context Free Grammar can generate a given input string.

Equivalently, construct the parsing tree which represents the derivation.

We consider first an ad-hoc manual method, called Recursive Descent which attempts to emulate the derivation process.

Recursive Descent Parsing

- Top-down parsing builds tree from root symbol
- Each production corresponds to one recursive procedure
- Each procedure recognizes an instance of a non-terminal, consumes the corresponding part of the input, and returns tree fragment for the non-terminal

General Structure

- Each right-hand side of a production provides a body for a function
- Each non-terminal on the rhs is translated into a call to the function (procedure) that recognizes that non-terminal
- Each terminal in the rhs is translated into a call to the lexical scanner. Error if the reulting token is not the expected terminal
- Each recognizing function returns a tree fragment.

Example: Parsing a Declaration

- FULL_TYPE_DECLARATION $::=
  type DEFINING_IDENTIFIER is TYPE_DEFINITION;
- Translates into
  - get token type
  - Find a defining_identifier — function call
  - get token is
  - Recognize a type_definition — function call
  - get token semicolon
- In practice, we already know that the first token is type, this is why this procedure was called in the first place. predictive parsing is guided by the next token

Example: Parsing a Loop

- FOR_STATEMENT $::=
  ITERATION_SCHEME loop STATEMENTS end loop;

- Translates into

  Node1 $:= find_iteration_scheme — function call
  get token loop
  List1 $:= Sequence_of_statements — function call
  get token end
  get token loop
  get token semicolon
  Result $:= build loop_node with Node1 and List1
  return Result

- In case we fail to find any of the expected tokens or one of the called functions returns a failure, this function returns a failure indication.

Complications

- If there are multiple productions for a non-terminal, we need a mechanism to determine which production to use

  IF_STAT $::= if COND then Stats end if;

  IF_STAT $::= if COND then Stats ELSE_PART end if;

When next token is if, cannot tell which production to use.

Solution:Factor Grammar

- If several productions have the same prefix, rewrite as a single production:
- IF_STAT $::= if COND then Stats [ELSE_PART] end if;
- Problem now reduces to recognize whether an optional component (ELSE_PART) is present. With a single token lookahead this is possible — look for a token else.

Illustrate Solution

- Consider rule

  IF_STAT $::= if COND then STATS [else STATS] end if;

```
boolean function get_IF( )
if Token=if then Scan else return 0;
if get_COND( )=0 then return 0;
if Token=then then Scan else return 0;
if get_STATS( )=0 then return 0;
if Token=else then
        {Scan; if get_STATS( )=0 then return 0};
if Token=end_if then Scan else return 0;
return 1
End get_IF
```

Complication: Left Recursion

- Grammar cannot be left-recursive
- $E \quad ::= \quad E + T \quad | \quad T$
- Problem: to find an E, start by finding an E...
- Original scheme leads to an infinite loop: grammar is inappropriate for recursive descent.

Eliminating Left Recursion

- $E \quad ::= \quad E + T \quad | \quad T$ means that eventually E expands into

$$T + T + T \cdots$$

- Rewrite as

$$\begin{array}{lcl} E & ::= & T E' \\ E' & ::= & +T E' \quad | \quad \epsilon \end{array}$$

- Informally: E' is a possibly empty sequence of terms (T) each preceded by $+$.

Left Recursion Involving Several Non-Terminals

- The grammar

$$
\begin{array}{lcccc}
A & ::= & BC & \mid & D \\
B & ::= & AE & \mid & F
\end{array}
$$

Can be rewritten as

$$
A \quad ::= \quad AEC \quad \mid \quad FC \quad \mid \quad D
$$

and then apply previous method

The General Case

Arrange the non-terminals in some order $A_1, \ldots, A_n$

for $i \in [1 \ldots n]$ **do**

$$\left[\begin{array}{l} \textbf{for } j \in [1 \ldots i{-}1] \textbf{ do} \\ \quad \left[\begin{array}{l} \text{Replace each production of the form } A_i \rightarrow A_j\gamma \text{ by the} \\ \text{productions } A_i \rightarrow \delta_1\gamma \mid \cdots \mid \delta_k\gamma\text{, where} \\ A_j \rightarrow \delta_1 \mid \cdots \mid \delta_k \text{ are all current } A_j\text{-productions.} \end{array}\right] \\ \text{Eliminate the immediate left recursion among} \\ \text{the } A_i\text{-productions} \end{array}\right]$$

Further Complications

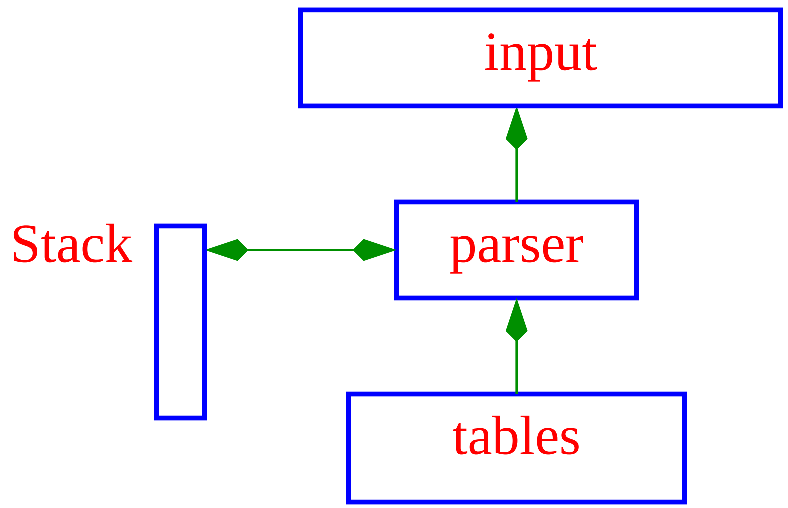
- Transformation does not preserve associativity.
- $E \quad ::= \quad E + T \quad | \quad T$
- Parses a+b+c as (a+b)+c
- $E \quad ::= \quad TE', \qquad E' \quad ::= \quad +TE' \quad | \quad \epsilon$
- Parses a+b+c as a+(b+c)
- Incorrect for a-b-c: must rewrite tree.
- In practice, treat as $E \ ::= \ T\{+T\}^*$

Use Loop to Parse Sequence of Terms

```
Node1 := P_Term;          -- call function that parses a term
loop
    exit when Token not in Token_Class_Binary_Addop;
    Node2 := New_Node(P_Binary_Adding_Operator);
    Node2↑.op := Token;
    Scan                                      -- past operator
    Node2↑.left := Node1;
    Node2↑.right := P_term;                   -- find next term
    Node1 := Node2;           -- operand for next operation
end loop;
```

Table-Driven Parsing

- Parsing performed by a finite-state machine augmented by a push-down stack
- FSM driven by table(s) generated automatically from grammar
- Language $\longrightarrow$ Generator $\longrightarrow$ Tables



Pushdown Automata

- A context-free language can be recognized by a finite state machine with a stack: a PDA.
- The PDA is defined by set of internal states and a transition table
- The PDA can read the input and read/write to the top of the stack
- Actions of the PDA are determined by the current state, the current symbol at the top of the stack and the current input character.
- Acceptance can be defined by either accepting state or reaching an empty stack

Formally

A PDA is defined by a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$, where

- Q — A finite set of States
- Σ — The input alphabet
- Γ — The stack alphabet
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$ — A non-deterministic transition function
- $q_0 \in Q$ — The initial state
- $Z_0 \in \Gamma$ — The initial stack symbol
- $F \subseteq Q$ — The set of accepting states

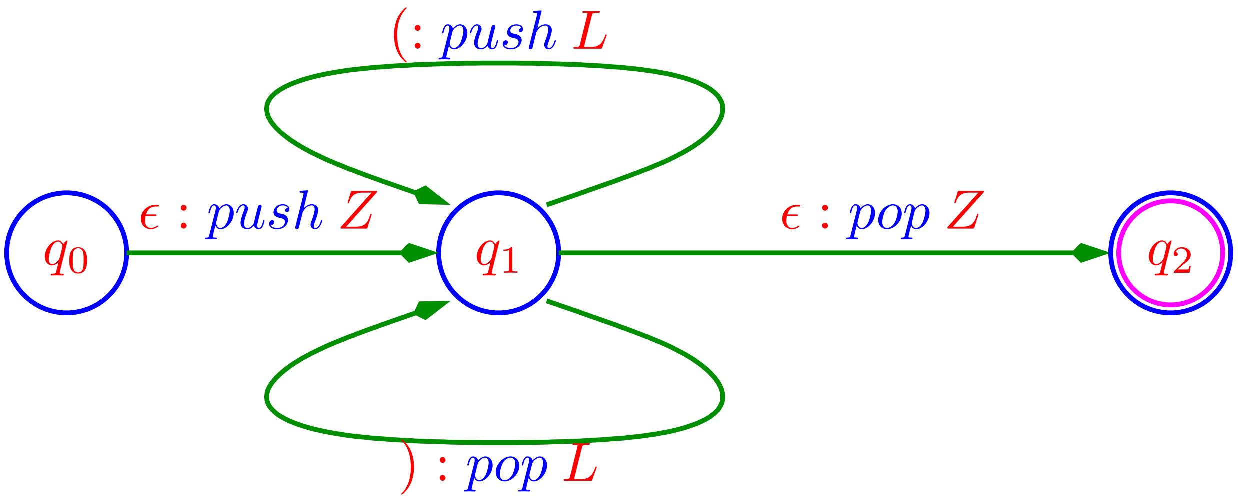
PDA's Can Accept Languages Beyond FSM's

For example, the language of balanced parentheses expressions.

This language can be generated by the following grammar:

$$S ::= (\ ) \quad | \quad (S) \quad | \quad SS$$

A PDA which accepts this language is given as follows:



The transition function for this automaton can be given by

$$\begin{array}{rcl} \delta(q_0, \epsilon, Z_0) & = & (q_1, ZZ_0) \\ \delta(q_1, ')', L) & = & (q_1, \epsilon) \end{array} \qquad \begin{array}{rcl} \delta(q_1, '(', X) & = & (q_1, LX) \\ \delta(q_1, \epsilon, Z) & = & (q_2, \epsilon) \end{array}$$

Runs and Acceptance

- An instantaneous description (*ID*) is a tuple (q, x, α), where q is a state, $x \in \Sigma^*$ is the input left to read, and $\alpha \in \Gamma^*$ is the current stack contents.
- For a PDA A,the *ID* $(q, x, \beta\alpha)$ is an A-successor of the *ID* $(p, ax, X\alpha)$, written $(p, ax, X\alpha) \vdash (q, x, \beta\alpha)$, if $(q, \beta) \in \delta_A(p, a, X)$.
- The reflexive-transitive closure of $\vdash$ is defined by the rules $I \vdash^* I$ and ($I \vdash^* J$ and $J \vdash K$ imply $I \vdash^* K$).
- The word $w \in \Sigma^*$ is accepted by PDA A if $(q_0, w, Z_0) \vdash^* (q, \epsilon, \gamma)$ for some $q \in F$ and $\gamma \in \Gamma^*$. The language recognized by A $L(A)$ is the set of all words accepted by A.
- An alternative definition is provided by the notion of the word $w \in \Sigma^*$ accepted by an empty stack if $(q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)$ for some $q \in Q$.

Properties of PDA's and CFL's

- A PDA is defined to be deterministic (DPDA) if it has no ϵ-moves, and for every $q \in Q$, $a \in \Sigma$, and $X \in \Gamma$, $|\delta(q, a, X)| = 1$.
- A language L is a CFL (can be generated by a CFG) iff L is recognizable by a (possibly non-deterministic) PDA.
- There exist CFL's which cannot be recognized by a DPDA (deterministic PDA). For example,

$$\{a^i b^i c \mid i > 1\} \cup \{a^i b^{2i} d \mid i > 1\}$$

Top-Down Parsing

- Parsing tree is synthesized from the root (sentence symbol).
- Stack contains symbols of RHS of most recent production and pending non-terminals.
- Automaton is trivial (no need for explicit states).
- Transition table indexed by stack's top symbol G and input symbol a. Entries in table are terminals or (set of) RHS of a production

Top-Down Parsing

- Actions
  - Initially, stack contains Grammar start symbol S.
  - At each step, let T be the symbol at top of the stack and a be the next input token.
  - If T is a terminal symbol, then T must equal a. Pop stack and consume input. This is called a match action.
  - Otherwise, choose a grammar production $T \to \alpha$, replace T by α. Do not consume a.
  - If stack and input string are both empty, then accept.
- Semantic Action when choosing a production, build tree node for non-terminal, attach to parent T.

Example: Top-Down Parsing

Consider the grammar

$$S ::= (\,) \quad | \quad (S) \quad | \quad SS$$

and the leftmost derivation

$$S \quad \Longrightarrow \quad (S) \quad \Longrightarrow \quad (SS) \quad \Longrightarrow \quad ((\,)S) \quad \Longrightarrow \quad ((\,)(\,))$$

A top-down recognition by a PDA is given by:

Stack	*Input*	*Action*	
S	$((\,)(\,))$	*Expand*	$S ::= (S)$
(S)	$((\,)(\,))$	*Match*	
$S)$	$(\,)(\,))$	*Expand*	$S ::= SS$
$SS)$	$(\,)(\,))$	*Expand*	$S ::= (\,)$
$(\,)S)$	$(\,)(\,))$	*Match*	
$)S)$	$)(\,))$	*Match*	
$S)$	$(\,))$	*Expand*	$S ::= (\,)$
$(\,))$	$(\,))$	*Match*	
$))$	$))$	*Match*	
$)$	$)$	*Match*	
ϵ	ϵ	*Accept*	

A CFG Corresponds to a PDA

Claim 3. *For every* CFG $\mathcal{G}$ *there exists a* PDA $\mathcal{A}$ *which accepts by top-down parsing the language* $L(\mathcal{G})$.

Proof: Let $\mathcal{G} : \langle T, N, P, S\rangle$ be a CFG. Construct the PDA $\mathcal{A} : \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$, where

- $Q = \{q\}$
- $\Sigma = T$
- $\Gamma = N \cup T$
- $q_0 = q$
- $Z_0 = S$ The grammar start symbol
- $F = \emptyset$ Acceptance is by empty stack

The transition function is defined as the smallest relation satisfying

- For every terminal $a \in T$, $\delta(q, a, a)$ includes (q, ϵ) (a match action).
- For every production $(A \to \alpha) \in P$, $\delta(q, A, \epsilon)$ includes (q, α) (an expand action).

Note that for every leftmost derivation $S \stackrel{*}{\Longrightarrow} xA\alpha \stackrel{*}{\Longrightarrow} xy$, there exists an automaton run $(q, xy, S) \vdash^* (q, y, A\alpha)$. ■

Correspondence of PDA's to CFG's

Claim 4. *For every single-state* PDA $\mathcal{A}$ *there exists a* CFG $\mathcal{G}$ *which generates the language recognized by* $\mathcal{A}$ *with empty stack.*

Proof: Let $\mathcal{A} : \langle\{q\}, \Sigma, \Gamma, \delta, q, Z_0, \emptyset\rangle$ be a single-state PDA. Construct the grammar $\mathcal{G} : \langle (T{=})\Sigma, (N{=})\Gamma, P, Z_0\rangle$, where the production set P contains the rule $X \to a\gamma$ for each $(q, \gamma) \in \delta(q, a, X)$.

We can show by induction on the length of the run that, whenever $(q, x, X) \vdash^* (q, \epsilon, \gamma)$, there exists a leftmost derivation of $\mathcal{G}$ of the form $X \stackrel{*}{\Longrightarrow} x\gamma$. It follows that if the string w is accepted by $\mathcal{A}$ with empty stack, then $Z_0 \stackrel{*}{\Longrightarrow} w$. ▪

What about multi-state PDA's?

Claim 5. *Every* PDA *is equivalent to a single-state* PDA.

Example: Converting a PDA to a CFG

Consider the PDA

$\mathcal{A} : \langle \{q\}, \Sigma{:}\{'(', ')'\}, \Gamma{:}\{S, T\}, \delta, q, S, \emptyset \rangle$, where the transition function is given by the following table:

$X \in \Gamma$	(	)	ϵ
S	STS	$\emptyset$	ϵ
T	$\emptyset$	ϵ	$\emptyset$

Following the recipe of Claim 4 we obtain the grammar $\mathcal{G} : \langle T{:}\{'(', ')'\}, N{:}\{S, T\}, P, S \rangle$, where the productions P include the following rules:

$$
\begin{array}{lcll}
S & \rightarrow & (STS & \mid \; \epsilon \\
T & \rightarrow &) &
\end{array}
$$

We Need Deterministic Parsing

The results presented so far allowed non-deterministic PDA's. Unlike finite-state automata, deterministic push down automata (DPDA) are strictly less expressive then general PDA's.

For example, the context-free language $\{a^n b^n c\} \cup \{a^n b^{2n} d\}$ cannot be recognized by a DPDA.

When doing parsing, we are interested only in a parsing process which is based on a deterministic process.

$LL(k)$ Grammars

Let $k > 0$ be a positive integer. The grammar $\mathcal{G}$ is called an $LL(k)$ grammar if, for every leftmost derivation

$$S \stackrel{*}{\Longrightarrow} xA\alpha \Longrightarrow x\beta\alpha \stackrel{*}{\Longrightarrow} xy$$

the production $A \to \beta$ is uniquely determined by A, and the k first characters of y.

The name is based on the fact that parsing according to such a grammar reads the input from left to right while constructing a leftmost derivation with a lookahead of k characters.

The unique determination means that if we have two derivations of the form

$$S \stackrel{*}{\Longrightarrow} x_1A\alpha_1 \Longrightarrow x_1\beta_1\alpha_1 \stackrel{*}{\Longrightarrow} x_1y_1$$
$$S \stackrel{*}{\Longrightarrow} x_2A\alpha_2 \Longrightarrow x_2\beta_2\alpha_2 \stackrel{*}{\Longrightarrow} x_2y_2$$

such that $y_1[1..k] = y_2[1..k]$, then $\beta_1 = \beta_2$.

Examples

- The following grammar for balanced parentheses expressions is not $LL(k)$ for any k:

$$S \quad ::= \quad (\,) \quad | \quad (S) \quad | \quad SS$$

A 1-lookahead is sufficient in order to distinguish between $(\,)$ and (S). However, no bounded lookahead is sufficient in order to distinguish between (S) and SS.

- The following grammar is $LL(1)$:

$$S \quad ::= \quad (S)S \quad | \quad \epsilon$$

If the next input character is $($ we choose $(S)S$. Otherwise we choose ϵ.

Constructing $LL(1)$ Tables

- Define two functions on the symbols of the grammar: *FIRST* and *FOLLOW*.
- For a non-terminal $A \in N$, *FIRST*(A) is the set of terminals that can appear as the first character in a string derived from N.

$$\mathit{FIRST}(A) = \{a \in T \mid A \stackrel{*}{\Longrightarrow} ax\} \quad \cup \quad \{\epsilon \mid A \stackrel{*}{\Longrightarrow} \epsilon\}$$

- *FOLLOW*(A) is the set of terminals that can appear **after** a string derived from A.

$$\mathit{FOLLOW}(A) = \{a \in T \mid S \stackrel{*}{\Longrightarrow} xA\alpha \stackrel{*}{\Longrightarrow} xy\alpha \Longrightarrow xya\beta\}$$

Computing *FIRST* (X)

- If X is terminal, then *FIRST* $(X) = \{X\}$.
- For each non-terminal X and production $X \rightarrow Y_1 Y_2 \cdots Y_k$,
  - Add a to *FIRST* (X) if, for some $i \in [1..k]$, $a \in$ *FIRST* (Y_i) and $\epsilon \in$ *FIRST* (Y_j), for all $j \in [1..i{-}1]$.
  - Add ϵ to *FIRST* (X) if $\epsilon \in$ *FIRST* (Y_i) for all $i \in [1..k]$.
- If $X \rightarrow \epsilon$ is a production, add ϵ to *FIRST* (X).

For a string $X_1 \cdots X_k$ and terminal a, we say that $a \in$ *FIRST* $(X_1 \cdots X_k)$ if $a \in$ *FIRST* (X_i), for some $i \in [1..k]$, and $\epsilon \in$ *FIRST* (X_j), for all $j \in [1..i{-}1]$. Also $\epsilon \in$ *FIRST* $(X_1 \cdots X_k)$ if $\epsilon \in$ *FIRST* (X_i) for all $i \in [1..k]$.

Computing *FOLLOW* (X)

- Place $\$$ in *FOLLOW* (S), where S is the start symbol, and $\$$ is the input end-marker.
- If there is a production $A \rightarrow \alpha B \beta$, then add all symbols in *FIRST* $(\beta) - \{\epsilon\}$ to *FOLLOW* (B).
- If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$, where $\epsilon \in$ *FIRST* (β), then add all symbols in *FOLLOW* (A) to *FOLLOW* (B).

Constructing an $LL(1)$ Parsing Table

This is a table $M[A, a]$ which, for each non-terminal $A \in N$ and next input character $a \in T$, tells us which production should next be taken.

- For each production $A \to \alpha$ of the grammar, do the following:
  - For each terminal $a \in \mathit{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$.
  - If $\epsilon \in \mathit{FIRST}(\alpha)$ then, for each terminal $b \in \mathit{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, b]$. If $\epsilon \in \mathit{FIRST}(\alpha)$ and $\$ \in \mathit{FOLLOW}(A)$, then add $A \to \alpha$ to $M[A, \$]$ as well.

Example: Constructing $LL(1)$ Tables

Starting with the grammar
$$\left\{\begin{array}{lllllll} E & \rightarrow & TE' & E' & \rightarrow & +TE' & \mid \epsilon \\ T & \rightarrow & FT' & T' & \rightarrow & *FT' & \mid \epsilon \\ & & & F & \rightarrow & (E) & \mid id \end{array}\right.$$

we construct the FIRST/FOLLOW tables:

Non-Terminal	FIRST	FOLLOW
E	$(, id$	$), \$$
E'	$+, \epsilon$	$), \$$
T	$(, id$	$+,), \$$
T'	$*, \epsilon$	$+,), \$$
F	$(, id$	$+, *,), \$$

leading to the following parsing table:

	id	$+$	$*$	$($	$)$	$\$$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

$LL(1)$ Parsing of Arithmetical Expressions

We can parse

Stack	*Input*	*Action*
E	$id + id * id\$$	$E \to TE'$
TE'	$id + id * id\$$	$T \to FT'$
$FT'E'$	$id + id * id\$$	$F \to id$
$idT'E'$	$id + id * id\$$	
$T'E'$	$+id * id\$$	$T' \to \epsilon$
E'	$+id * id\$$	$E' \to +TE'$
$+TE'$	$+id * id\$$	
TE'	$id * id\$$	$T \to FT'$
$FT'E'$	$id * id\$$	$F \to id$
$idT'E'$	$id * id\$$	
$T'E'$	$*id\$$	$T' \to *FT'$
$*FT'E'$	$*id\$$	
$FT'E'$	$id\$$	$F \to id$
$idT'E'$	$id\$$	
$T'E'$	$\$$	$T' \to \epsilon$
E'	$\$$	$E' \to \epsilon$
ϵ	$\$$	*Accept*

with table

	id	$+$	$\$$
E	$E \to TE'$		
E'		$E' \to +TE'$	$E' \to \epsilon$
T	$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$		

	$($	$)$	$*$
E	$E \to TE'$		
E'		$E' \to \epsilon$	
T	$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$
F	$F \to (E)$		

Correctness of the Construction

Claim 6. *A grammar* $\mathcal{G}$ *is an* $LL(1)$ *grammar iff the parsing table* $M[A, a]$ *contains at most one production in each entry.*