بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

$\mathrm{LL}(1)$ *Grammars*

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called $\mathrm{LL}(1)$. The first "L" in $\mathrm{LL}(1)$ stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

The class of $\mathrm{LL}(1)$ grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be $\mathrm{LL}(1)$.

شرط لازم و کافی برای $\mathrm{LL}(1)$ بودن یک گرامر

A grammar G is $\mathrm{LL}(1)$ if and only if whenever $A \to \alpha|\beta$ are two distinct productions of G, the following conditions hold:

1. *For no terminal a do both α and β derive strings beginning with a.*
2. *At most one of α and β can derive the empty string.*
3. *If $\beta \Rightarrow^* \varepsilon$, then α does not derive any string beginning with a terminal in $\mathrm{FOLLOW}(A)$. Likewise, if $\alpha \Rightarrow^* \varepsilon$, then β does not derive any string beginning with a terminal in $\mathrm{FOLLOW}(A)$.*

The first two conditions are equivalent to the statement that $\mathrm{FIRST}(\alpha)$ and $\mathrm{FIRST}(\beta)$ are disjoint sets. The third condition is equivalent to stating that if ε is in $\mathrm{FIRST}(\beta)$, then $\mathrm{FIRST}(\alpha)$ and $\mathrm{FOLLOW}(A)$ are disjoint sets, and likewise if ε is in $\mathrm{FIRST}(\alpha)$.

Predictive parsers can be constructed for $\mathrm{LL}(1)$ ***grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol. Flow-of-control constructs, with their distinguishing keywords, generally satisfy the*** $\mathrm{LL}(1)$ ***constraints. For instance, if we have the productions***

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if} \ (\ expr \ ) \ stmt \ \textbf{else} \ stmt \\ & | & \textbf{while} \ (\ expr \ ) \ stmt \\ & | & \{ \ stmt\_list \ \} \end{array}$$

then the keywords if, while, and the symbol $\{$ ***tell us which alternative is the only one that could possibly succeed if we are to find a statement.***

Predictive Parsing Table

The next algorithm collects the information from FIRST *and* FOLLOW *sets into a* ***predictive parsing table*** $M[A, a]$, *a two-dimensional array, where* A *is a nonterminal, and* a *is a terminal or the symbol \$, the input endmarker.*

☞ *The algorithm is based on the following idea: the production* $A \to \alpha$ *is chosen if the next input symbol* a *is in* $\mathrm{FIRST}(\alpha)$.

☞ *The only complication occurs when* $\alpha = \varepsilon$ *or, more generally,* $\alpha \Rightarrow^* \varepsilon$. *In this case, we should again choose* $A \to \alpha$, *if the current input symbol is in* $\mathrm{FOLLOW}(A)$, *or if the \$ on the input has been reached and \$ is in* $\mathrm{FOLLOW}(A)$.

پروسهٔ ساخت جدول تجزیهٔ پیش‌بین

Algorithm 4.31 : Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Example 4.32:

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$

NON - TERMINAL	INPUT SYMBOL					
	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	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Consider production $E \rightarrow TE'$. Since

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \mathbf{id}\}$$

this production is added to $M[E, (]$ and $M[E, \mathbf{id}]$. Production $E' \rightarrow +TE'$ is added to $M[E', +]$ since $\text{FIRST}(+TE') = \{+\}$. Since $\text{FOLLOW}(E') = \{), \$\}$, production $E' \rightarrow \epsilon$ is added to $M[E',)]$ and $M[E', \$]$. $\Box$

جدول برای گرامرهای $\mathrm{LL}(1)$ واجد این ویژگی‌ست که درایه‌ها تنها یک عضو دارند

Algorithm 4.31 can be applied to any grammar G to produce a parsing table M. For every $\mathrm{LL}(1)$ grammar, each parsing-table entry uniquely identifies a production or signals an error. For some grammars, however, M may have some entries that are multiply defined. For example, if G is left-recursive or ambiguous, then M will have at least one multiply defined entry. Although left-recursion elimination and left factoring are easy to do, there are some grammars for which no amount of alteration will produce an $\mathrm{LL}(1)$ grammar.

Example 4.33: The grammar is ambiguous, and the corresponding language has no $\mathrm{LL}(1)$ *grammar at all.*

$$\begin{array}{lcl} S & \rightarrow & iEtSS' \mid a \\ S' & \rightarrow & eS \mid \epsilon \\ E & \rightarrow & b \end{array}$$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

یک گرامر که LL(1) نیست

Example **[A Grammar Which is Not an *LL*(1) Grammar]** Let us consider the grammar G whose axiom is S and whose productions are:

$$S \to \varepsilon \mid a\,b\,A$$
$$A \to S\,a\,a \mid b$$

We have that:

$First_1(\varepsilon) = \{\varepsilon\}$ $\quad$ $First_1(a\,bA) = \{a\}$ $\quad$ $First_1(S) = \{\varepsilon, a\}$

$First_1(S\,a\,a) = \{a\}$ $\quad$ $First_1(b) = \{b\}$

$Follow_1(S) = \{\$, a\}$ $\quad$ $Follow_1(A) = \{\$, a\}$

The parsing table is:

	a	b	$\$$
S	$S \to a\,bA$ $S \to \varepsilon$		$S \to \varepsilon$
A	$A \to S\,a\,a$	$A \to b$	$A \to S\,a\,a$

The given grammar is *not* $LL(1)$ because in this parsing table for the symbol S on the top of the stack and the input symbol a, there are two productions. □

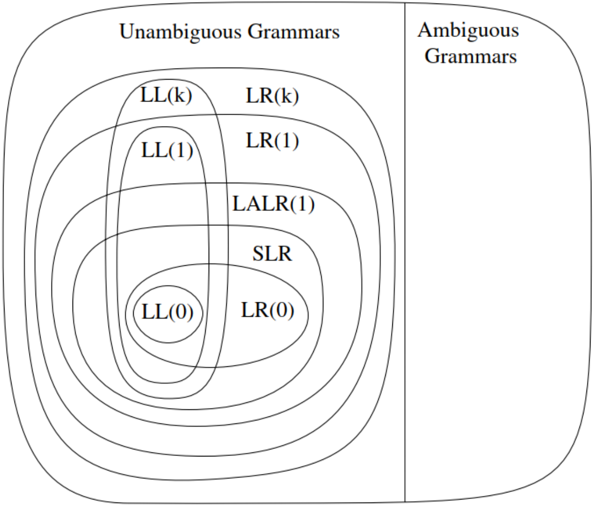
چند نکتهٔ درخور توجه

☞ *An ambiguous grammar will always lead to duplicate entries in a predictive parsing table.*

☞ *Grammars whose predictive parsing tables contain no duplicate entries are called* $\mathrm{LL}(1)$. *This stands for left-to-right parse, leftmost-derivation,* 1*-symbol lookahead.*

☞ *We can generalize the notion of* FIRST *sets to describe the first* k *tokens of a string, and to make an* $\mathrm{LL}(k)$ *parsing table whose rows are the nonterminals and columns are every sequence of* k *terminals. This is rarely done (because the tables are so large), but sometimes when you write a recursive-descent parser by hand you need to look more than one token ahead. Grammars parsable with* $\mathrm{LL}(2)$ *parsing tables are called* $\mathrm{LL}(2)$ *grammars, and similarly for* $\mathrm{LL}(3)$*, etc. Every* $\mathrm{LL}(1)$ *grammar is an* $\mathrm{LL}(2)$ *grammar, and so on. No ambiguous grammar is* $\mathrm{LL}(k)$ *for any* k*.*

A hierarchy of grammar classes



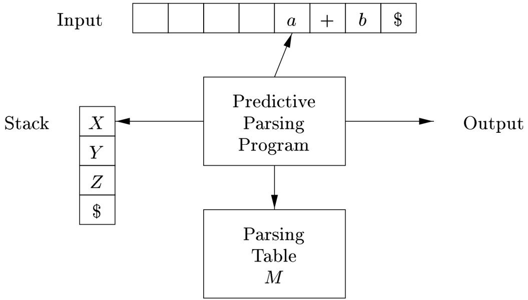
Nonrecursive Predictive Parsing

*A nonrecursive predictive parser can be built by maintaining a **stack** explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that*

$$S \Rightarrow^*_{lm} w\alpha$$

The table-driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm 4.31, and an output stream. The input buffer contains the string to be parsed, followed by the endmarker \$. We reuse the symbol \$ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of \$.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol. If X is a nonterminal, the parser chooses an X-production by consulting entry $M[X, a]$ of the parsing table M. (Additional code could be executed here, for example, code to construct a node in a parse tree.) Otherwise, it checks for a match between the terminal X and current input symbol a.



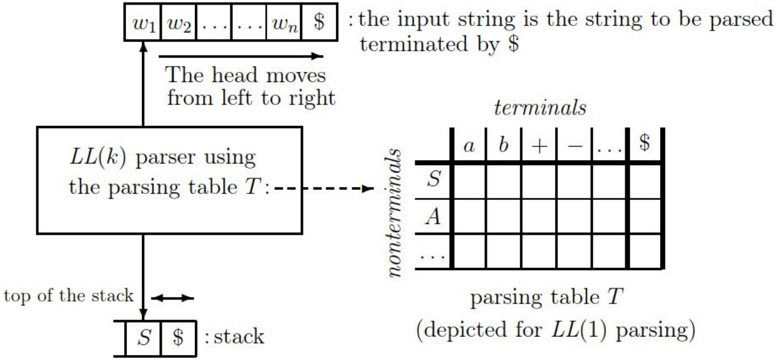


FIGURE A deterministic pushdown automaton for $LL(k)$ parsing, with $k \geq 1$. The string to be parsed is $w_1 w_2 \ldots w_n$. Initially, the stack has two symbols only: (i) S on top of the stack, and (ii) \$ at the bottom of the stack. The input string is the string to be parsed with the extra rightmost symbol \$. We have depicted the parsing table T for the $LL(1)$ parsers. For the $LL(k)$ parsers, with $k > 1$, different tables should be used.

Table-driven predictive parser

Algorithm 4.34: Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X = a ) pop the stack and let a be the next symbol of w;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```

Chop move and expand move

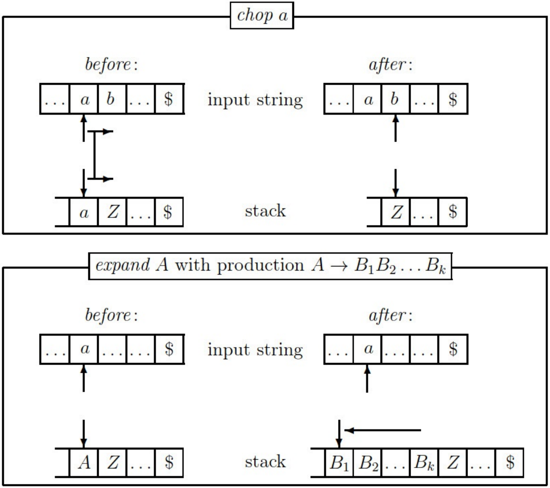
chop move:

if the input head is pointing at a terminal symbol, say a, and the same symbol a is at the top of the stack, then the input head is moved one cell to the right and the stack is popped;

expand move:

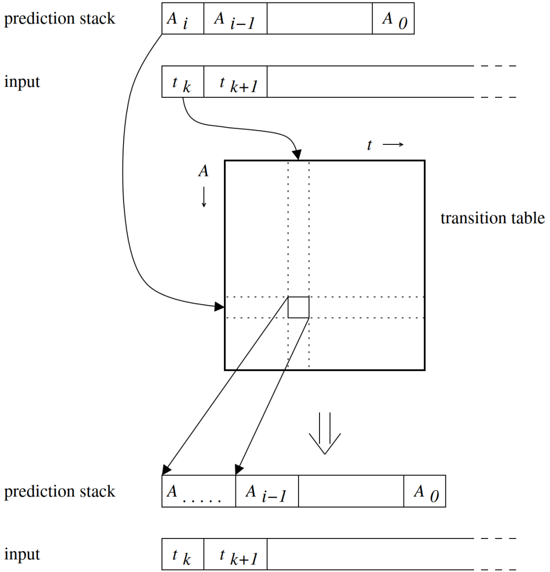
if the input head is pointing at a terminal symbol, say a, and the top of the stack is a nonterminal symbol, say A, then the stack is popped and a new string $\alpha_1\alpha_2\ldots\alpha_n$, with $\alpha_i \in V_T \cup V_N$, for $i = 1, \ldots, n$, is pushed onto the stack if the production $A \to \alpha_1\alpha_2\ldots\alpha_n$ is at the entry (A, a) of the parsing table T (thus, after this move the new top symbol of the stack will be α_1).

Chop move and expand move

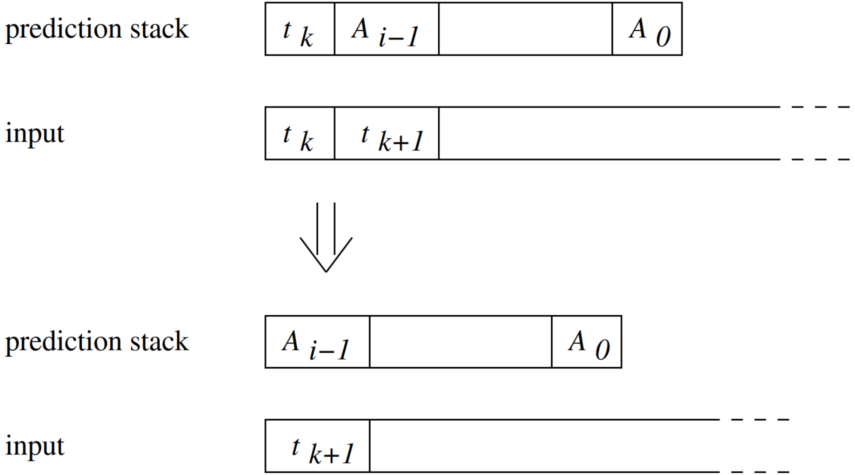


The *chop* move and the *expand* move of an $LL(1)$ parser.
a and b are symbols in V_T and Z is a symbol in $V_T \cup V_N \cup \{\$\}$.

Prediction move in an $\mathrm{LL}(1)$ push-down automaton



Match move in an $\mathrm{LL}(1)$ *push-down automaton*



The sequence of moves on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow +\ TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow *\ FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

https://github.com/javacc/javacc

- JavaCC generates top-down (recursive descent) parsers as opposed to bottom-up parsers generated by YACC-like tools. This allows the use of more general grammars, although left-recursion is disallowed. Top-down parsers have a number of other advantages (besides more general grammars) such as being easier to debug, having the ability to parse to any non-terminal in the grammar, and also having the ability to pass values (attributes) both up and down the parse tree during parsing.

- By default, JavaCC generates an `LL(1)` parser. However, there may be portions of grammar that are not `LL(1)`. JavaCC offers the capabilities of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points. For example, the parser is `LL(k)` only at such points, but remains `LL(1)` everywhere else for better performance. Shift-reduce and reduce-reduce conflicts are not an issue for top-down parsers.