Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1403-1404

Error Recovery in Predictive Parsing

- An error is detected during predictive parsing when
  1. The terminal on top of the stack does not match the next input symbol
  2. Nonterminal A is on top of the stack, a is the next input symbol, and $M[A, a]$ is error (i.e., the parsing-table entry is empty)

- **Panic Mode**
  - ***Panic-mode error recovery*** is based on the idea of skipping over symbols on the input until a token in a selected set of **synchronizing tokens** appears
  - Its effectiveness depends on the choice of synchronizing set

Error Recovery in Predictive Parsing

- **Panic Mode**
  1. As a starting point, place all symbols in $FOLLOW(A)$ into the synchronizing set for nonterminal A
     - If we skip tokens until an element of $FOLLOW(A)$ is seen and pop A from the stack, it is likely that parsing can continue
  2. If we add symbols in $FIRST(A)$ to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in $FIRST(A)$ appears in the input
  3. If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default
     - Doing so may postpone some error detection

Error Recovery in Predictive Parsing

- **Panic Mode**
  4. If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal
  5. Often, there is a hierarchical structure on constructs in a language; for example, expressions appear within statements, which appear within blocks, and so on. We can add to the synchronizing set of a lower-level construct the symbols that begin higher-level constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions

Error Recovery in Predictive Parsing

- **Example**
  - In this example, "synch" indicating synchronizing tokens obtained from the FOLLOW set of the nonterminal
    - If the parser looks up entry M[A, a] and finds that it is blank, then the input symbol *a* is skipped
    - If the entry is "synch" then the nonterminal on top of the stack is popped in an attempt to resume parsing
    - If a token on top of the stack does not match the input symbol, then we pop the token from the stack

در صورتی‌که با پاپ‌کردن پشته خالی شود، به جای پاپ‌کردن توکن ورودی نادیده گرفته می‌شود.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E \rightarrow +TE'$			$E \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**	synch	synch	$F \rightarrow (E)$	synch	synch

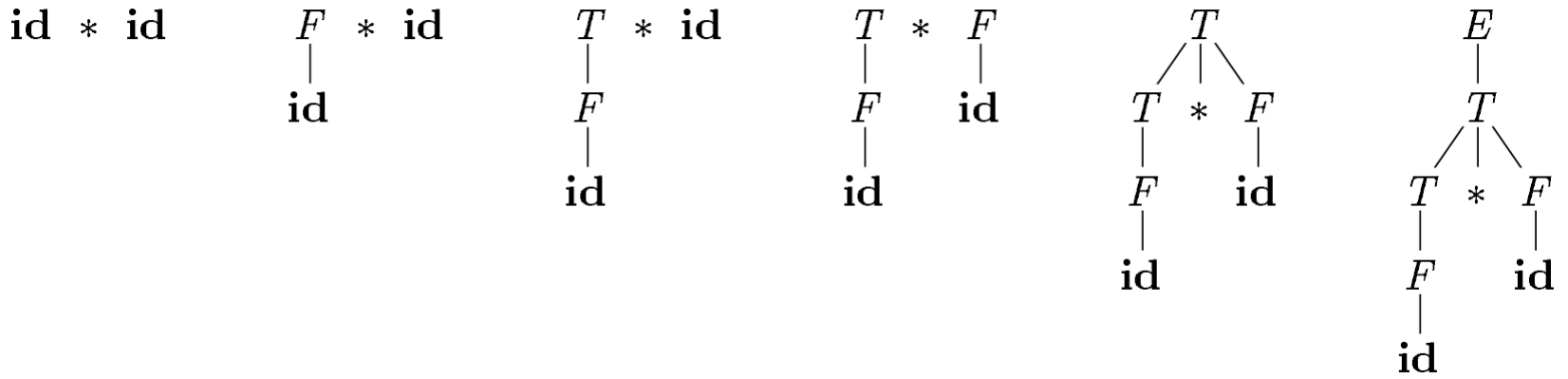
Error Recovery in Predictive Parsing

- **Example**

STACK	INPUT	REMARK
$E\ \$$	$)\ \mathbf{id} * + \mathbf{id}\ \$$	error, skip)
$E\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	**id** is in FIRST(E)
$TE'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$FT'E'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$T'E'\ \$$	$* + \mathbf{id}\ \$$	
$* FT'E'\ \$$	$* + \mathbf{id}\ \$$	
$FT'E'\ \$$	$+ \mathbf{id}\ \$$	error, $M[F, +] =$ synch F has been popped
$T'E'\ \$$	$+ \mathbf{id}\ \$$	
$E'\ \$$	$+ \mathbf{id}\ \$$	
$+ TE'\ \$$	$+ \mathbf{id}\ \$$	
$TE'\ \$$	$\mathbf{id}\ \$$	
$FT'E'\ \$$	$\mathbf{id}\ \$$	
$\mathbf{id}\ T'E'\ \$$	$\mathbf{id}\ \$$	
$T'E'\ \$$	$\$$	
$E'\ \$$	$\$$	
$\$$	$\$$	

Bottom-Up Parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)
- **Example**



Bottom-Up Parsing

- **Reductions**
  - We can think of bottom-up parsing as the process of ***reducing*** a string w to the start symbol of the grammar
  - The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds
  - **Example**
    - The sequence of reductions in the previous example:
      - $id * id,\ F * id,\ T * id,\ T * F,\ T,\ E$
  - A reduction is the reverse of a step in a derivation
  - The goal of bottom-up parsing is therefore to construct a derivation in reverse
  - **In previous example:** $E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$
    - ***This derivation is in fact a rightmost derivation***

Bottom-Up Parsing

- **Handle Pruning**
  - Bottom-up parsing during a left-to-right scan of the input constructs a ***rightmost derivation in reverse***
  - A ***handle*** is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation
  - **Example**

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

Bottom-Up Parsing

- **Shift-Reduce Parsing**
  - **Shift-reduce parsing is a form of bottom-up parsing** in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed
  - The **handle always appears at the top of the stack** just before it is identified as the handle
  - We use $ to mark the bottom of the stack and also the right end of the input
  - In bottom-up parsing, we show **the top of the stack on the right**, rather than on the left as we did for top-down parsing

Bottom-Up Parsing

- **Shift-Reduce Parsing**
  - Initially, the stack is empty, and the string *w* is on the input

STACK	INPUT
$	*w* $

  - If parser enters the following configuration announces successful completion of parsing

STACK	INPUT
$ *S*	$

Bottom-Up Parsing

- **Shift-Reduce Parsing**
  - **Example**

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \to \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \to F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \to \mathbf{id}$
$ $T * F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	accept

Bottom-Up Parsing

- **Shift-Reduce Parsing**
  - Possible actions a shift-reduce parser can make
    1. **Shift**
       - Shift the next input symbol onto the top of the stack
    2. **Reduce**
       - The right end of the string to be reduced must be at the top of the stack
       - Locate the left end of the string within the stack and decide with what nonterminal to replace the string
    3. **Accept**
       - Announce successful completion of parsing
    4. **Error**
       - Discover a syntax error and call an error recovery routine

Bottom-Up Parsing

- Conflicts During Shift-Reduce Parsing
  - **Shift/Reduce conflict**

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	**other**

STACK	INPUT
··· **if** *expr* **then** *stmt*	**else** ··· $

  - **Reduce/Reduce conflict**

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

STACK	INPUT
··· **id** (**id**	, **id**) ···

Bottom-Up Parsing

- **Example:** Grammer $S \rightarrow 0\ S\ 1 \mid 0\ 1$ **Input 000111**

Stack	Input	Handle	Action
$	000111$		Shift
$0	00111$		Shift
$00	0111$		Shift
$000	111$		Shift
$0001	11$	01	Reduce: S -> 01
$00S	11$		Shift
$00S1	1$	0S1	Reduce : S -> 0S1
$0S	1$		Shift
$0S1	$	0S1	Reduce : S -> 0S1
$S	$		Accept