Compilation Principle
编 译 原 理

第6讲：语法分析(3)

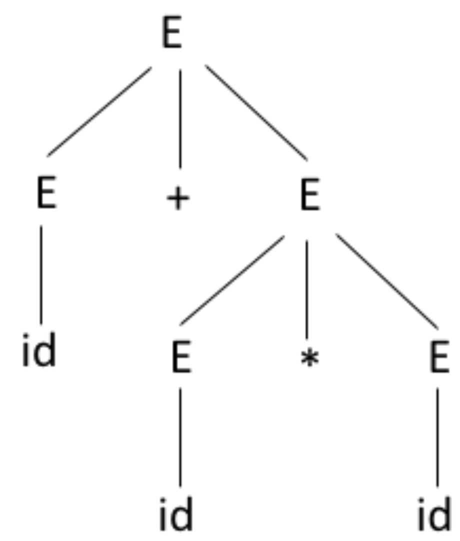
张献伟

xianweiz.github.io

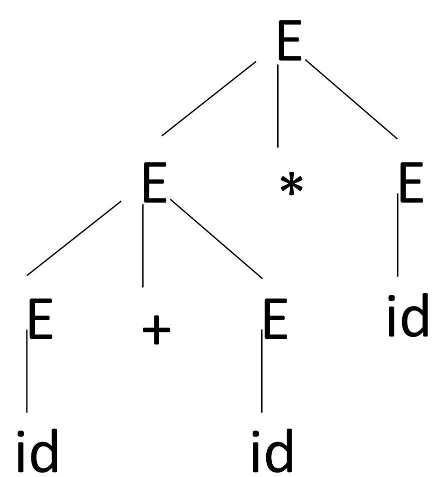
DCS290, 03/18/2021

Review: Ambiguous Grammar

- Grammar E→E*E | E+E | (E) | id
  - Ambiguous. Why?
  - Two distinct leftmost derivations for the sentence id + id * id



- Are the two trees have the same meaning?
  - Above: id + (id * id)
  - Below: (id + id) * id



- The deepest sub-tree is traversed first, thus higher precedence

Review: Ambiguity Removal

- How to remove the ambiguity?
- Specify precedence
  - The higher level of the production, the lower priority of operator
  - The lower level of the production, the higher priority of operator
- Specify associativity
  - If the operator is left associative, induce left recursion in its production
  - If the operator is right associative, induce right recursion in its production

$E \rightarrow E * E \mid E + E \mid (E) \mid id$ ⇒

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid id$

still possible to get
id + (id + id)
and
(id + id) + id
what if '-' (minus)?

⇒

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Now, can only have more '+' on left
E: sum of one or more terms (T)
T: product of one or more factors (F)
F: an identifier or a '()'ed expr

Review: Top-down and Bottom-up

- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

- This language has only one sentence: L(G) = {acbd}

Top-down (Leftmost Derivation)

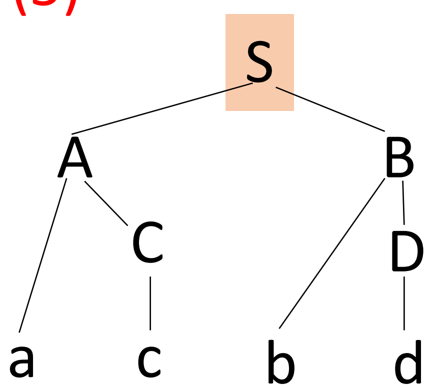
$S \Rightarrow AB$ (1)

$\Rightarrow aCB$ (2)

$\Rightarrow acB$ (3)

$\Rightarrow acbD$ (4)

$\Rightarrow acbd$ (5)



Bottom-up (reverse of rightmost derivation)

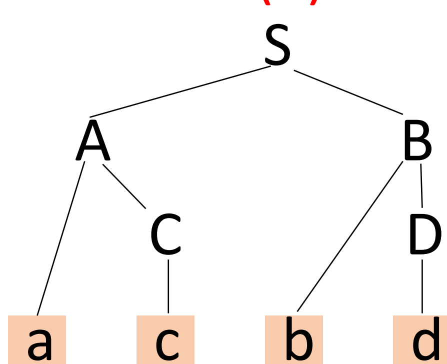
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbD$ (2)

$\Rightarrow acbd$ (1)



Preview: Bottom-up Steps

- Consider a CFG grammar G

S→AB A→aC B→bD D→d C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	
$A	bd$	Reduce
$Ab	d$	Shift
$Abd	$	Shift
$AbD	$	Reduce
$AB	$	Reduce
$S	$	Reduce

Bottom-up (reverse of rightmost derivation)

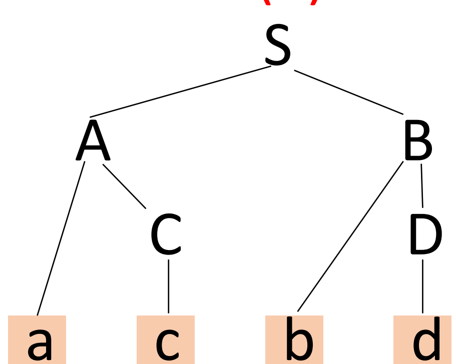
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

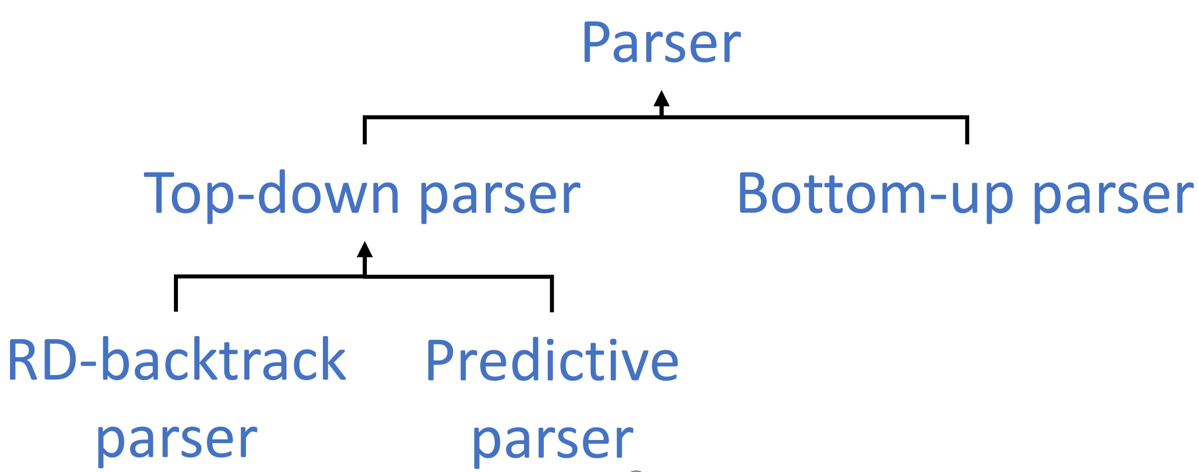
$\Rightarrow aCbD$ (2)

$\Rightarrow acbd$ (1)



Recursive Descent[递归下降]

- **Recursive descent** is a simple and general parsing strategy
  - Try and backtrack
  - Left-recursion must be eliminated first
    - Can be eliminated automatically using some algorithm
- However it is not popular because of **backtracking**
  - Backtracking requires re-parsing the same string
  - Which is inefficient (can take exponential time)
  - Also undoing semantic actions may be difficult
    - E.g. removing already added nodes in parse tree



Predictive Parsers[预测分析]

- A parser with no backtracking: predict correct next production given next input terminal(s)
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking
  - If not unique, grammar cannot use predictive parsers

A→aBD | bBB
B→c | bce
D→d

parsing input "abced" requires no backtracking

Predictive Parsers (cont.)

- A predictive parser chooses the production to apply solely on the basis of
  - Next input symbols
  - Current nonterminal being processed
- Patterns in grammars that prevent predictive parsing
  - **Common prefix**[共同前缀]:

    $A \rightarrow \alpha\beta \mid \alpha\gamma$

    Given input terminal(s) α, cannot choose between two rules
  - **Left recursion**[左递归]:

    $A \rightarrow A\beta \mid \alpha$

    Given input terminal(s) α, cannot choose between two rules

    What is the language of the grammar? $\alpha\beta^*$

Rewrite Grammars for Prediction

- **Left factoring**[左公因子]: removes common left prefix
  - In previous example: $A \to \alpha\beta \mid \alpha\gamma$
  - can be changed to

    $A \to \alpha A'$

    $A' \to \beta \mid \gamma$
  - Given input α, A' can can choose between β or γ

  (Assuming β or γ do not start with α)
- **Left-recursion removal**: same as for recursive descent
  - In previous example: $A \to A\beta \mid \alpha$
  - can be changed to

    $A \to \alpha A'$

    $A' \to \beta A' \mid \varepsilon$
  - Given input α, A' can can choose between β or ε

  (Assuming β doesn't start with α or A' isn't followed by α)

LL(k) Parser / Grammar / Language

- **LL(k) Parser**
  - A predictive parser that uses *k* lookahead tokens
  - **L**: scans the input from left to right
  - **L**: produces a leftmost derivation
  - **k**: using *k* input symbols of lookahead at each step to decide
- **LL(k) Grammar**
  - A grammar that can be parsed using an LL(k) parser
  - LL(k) ⊂ CFG
    - Some CFGs are not LL(k): common prefix or left-recursion
- **LL(k) Language**
  - A language that can be expressed as an LL(k) grammar
- Many languages are LL(k) ... in fact many are LL(1)!

LL(k) Parser Implementation

- Implemented in a recursive or non-recursive fashion
  - Recursive: recursive descent (recursive function calls)
  - Non-recursive: explicit stack to keep track of recursion
- Recursive LL(1) parser for: $A \rightarrow B \mid C$, $B \rightarrow b$, $C \rightarrow c$
  - Parser consists of small functions, one for each non-terminal

```
int A() {
    int token = peekNext(); // lookahead token
    switch(token) {
        case ’b’: // ’B’ starts with ’b’
                return B();
        case ’c’: // ’C’ starts with ’c’
                return C();
        default: // Reject
                return 0;
}
```

LL(k) Parser Implementation (cont.)

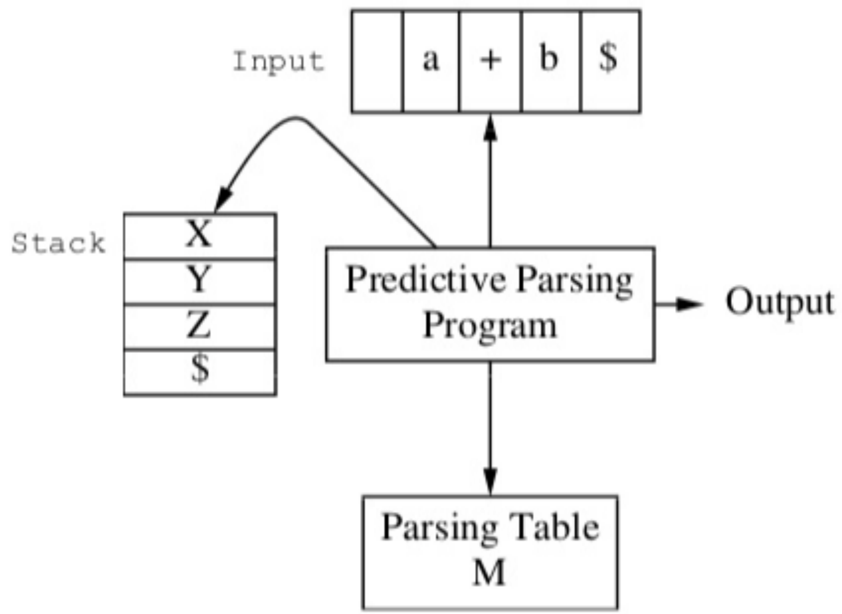
- Recursive LL(1) parser for: $A \rightarrow B \mid C,\ B \rightarrow b,\ C \rightarrow c$

```
int A() {
    int token = peekNext(); // lookahead token
    switch(token) {
        case ’b’: // ’B’ starts with ’b’
                return B();
        case ’c’: // ’C’ starts with ’c’
                return C();
        default: // Reject
                return 0;
}
```

- Is there a way to express above code more concisely?
  - Non-recursive LL(k) parsers use a **state transition table** (Just like finite automata)
  - Easier to automatically generate a non-recursive parser

Non-recursive LL(1) Parser

- Table-driven parser: amenable to automatic code generation (just like lexers)
  - **Input buffer**: contains the string to be parsed, followed by $
  - **Stack**: holds unmatched portion of derivation string
  - **Parse table** M[A, b]: an entry containing rule "A→…" or error
  - **Parser driver** (a.k.a., predictive parsing program): next action based on (stack top, current token)



LL(1) Parse Table: Example

Table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

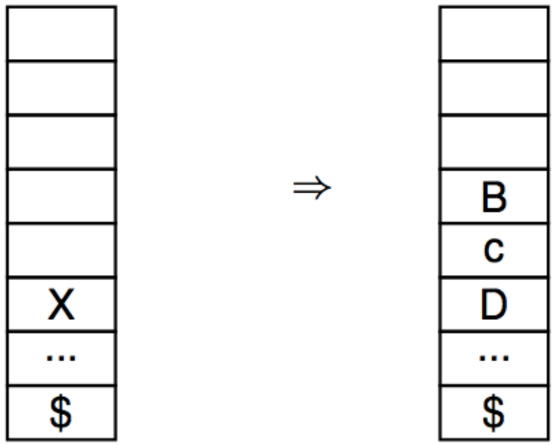
- Implementation with 2D parse table
  - **First column** lists all non-terminals in the grammar
  - **First row** lists all possible terminals in the grammar and $
  - A table entry contains one production
    - One action for each (non-terminal, input) combination
    - It “predicts” the correct action based on one lookahead
    - No backtracking required

LL(1) Parsing Algorithm

- Initial state
  - **Input** tape: input tokens followed by '$'
  - **Stack**: start symbol followed by '$' at bottom
- General idea: repeat one of two actions
  - Expand symbol at top of stack by applying a production
  - Match terminal symbol at top of stack with input token
- Step-by-step parsing based on (X,a)
  - X: symbol at the top of the stack
  - a: current input token
    - If X ∈ T, then
      - If X == a == $, parser halts with "success"
      - If X == a != $, successful match, pop X from stack and advance input head
      - If X != a, parser halts and input is rejected
    - if X ∈ N, then
      - if M[X,a] == 'X→RHS", pop X and push RHS to stack
      - if M[X,a] == empty, parser halts and input is rejected

Push RHS in Reverse Order

- For (X, a)
  - X: symbol at the top of the stack
  - a: current input token
- If M[X,a] = "$X \rightarrow BcD$"



- Performs the leftmost derivation: $\alpha\ X\ \beta \Rightarrow \alpha\ BcD\ \beta$
  - α: string that has already been matched with input
  - β: string yet to be matched, corresponding to the ... above

Applying LL(1) Parsing to Grammar

- Consider the grammar

  $E \rightarrow T+E \mid T$

  $T \rightarrow int{*}T \mid int \mid (E)$

  - No left recursion
  - But require left factoring
- After rewriting grammar, we have

  $E \rightarrow TE'$

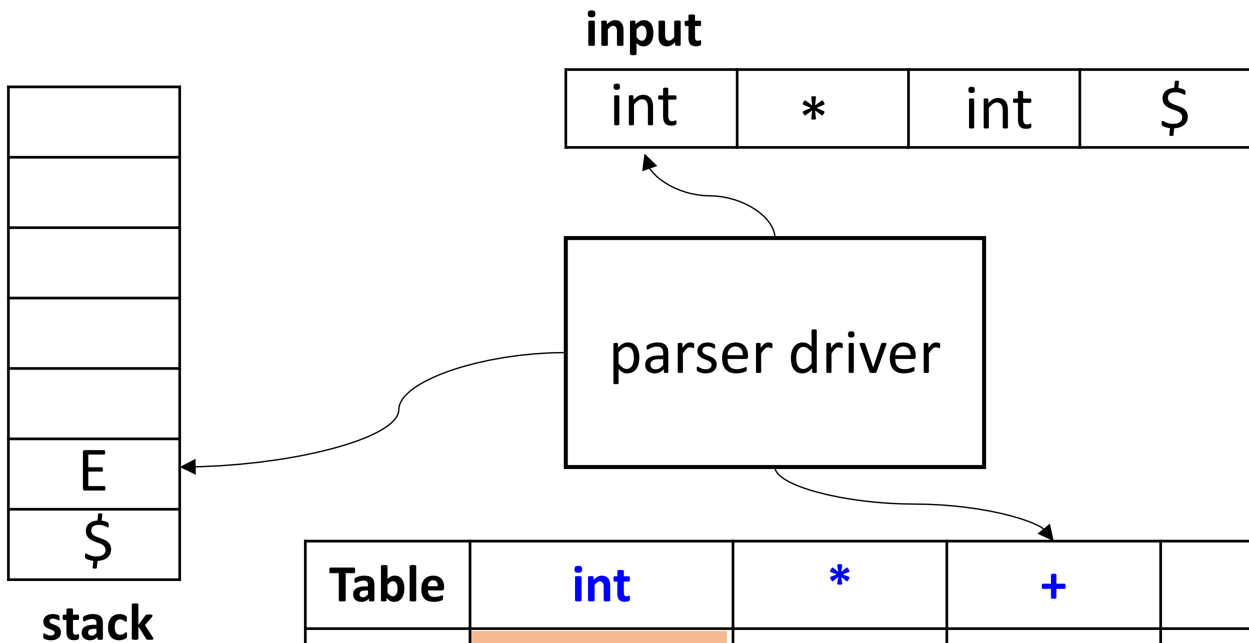
  $E' \rightarrow +E \mid \varepsilon$

  $T \rightarrow intT' \mid (E)$

  $T' \rightarrow {*}T \mid \varepsilon$

Using the Parse Table

- To recognize “int * int”

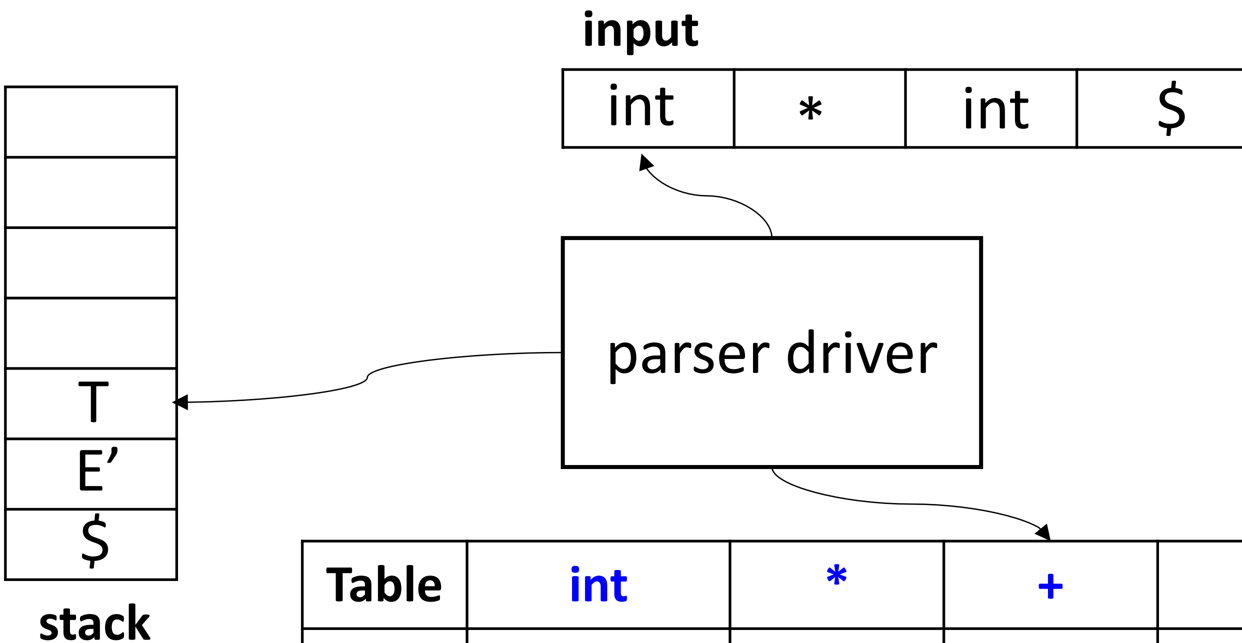


E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

Table	int	*	+	(	)	$
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

Using the Parse Table

- To recognize "int * int"



E → TE'
E' → +E | ε
T → intT' | (E)
T' → *T | ε

Table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Using the Parse Table

- To recognize “int * int”

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

input

int	*	int	$

parser driver

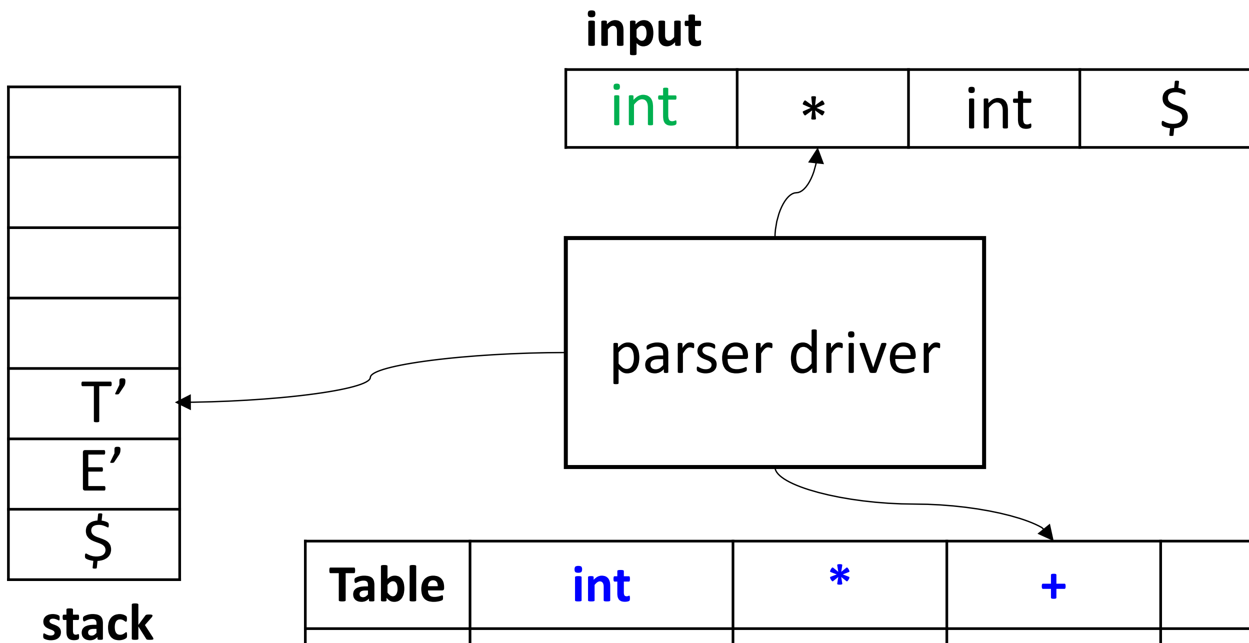
int
T’
E’
$

stack

Table	**int**	*****	**+**	**(**	**)**	**$**
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

Using the Parse Table

- To recognize "int * int"



E → TE'
E' → +E | ε
T → intT' | (E)
T' → *T | ε

Table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Using the Parse Table

- To recognize “int * int”

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

input

int	*	int	$

parser driver

*
T
E’
$

stack

Table	**int**	*****	**+**	**(**	**)**	**$**
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

Using the Parse Table

- To recognize “int * int”

$E \rightarrow TE'$
$E' \rightarrow +E \mid \varepsilon$
$T \rightarrow intT' \mid (E)$
$T' \rightarrow *T \mid \varepsilon$

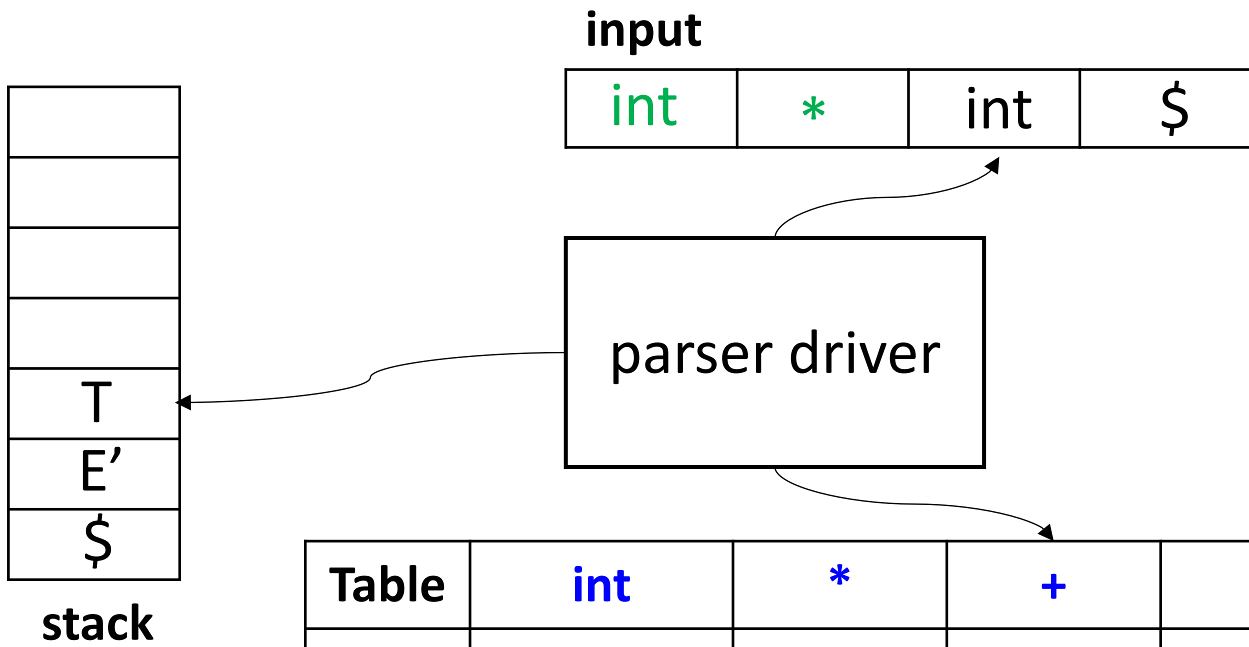


Table	int	*	+	(	)	$
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

Using the Parse Table

- To recognize "int * int"

$E \to TE'$
$E' \to +E \mid \varepsilon$
$T \to intT' \mid (E)$
$T' \to *T \mid \varepsilon$

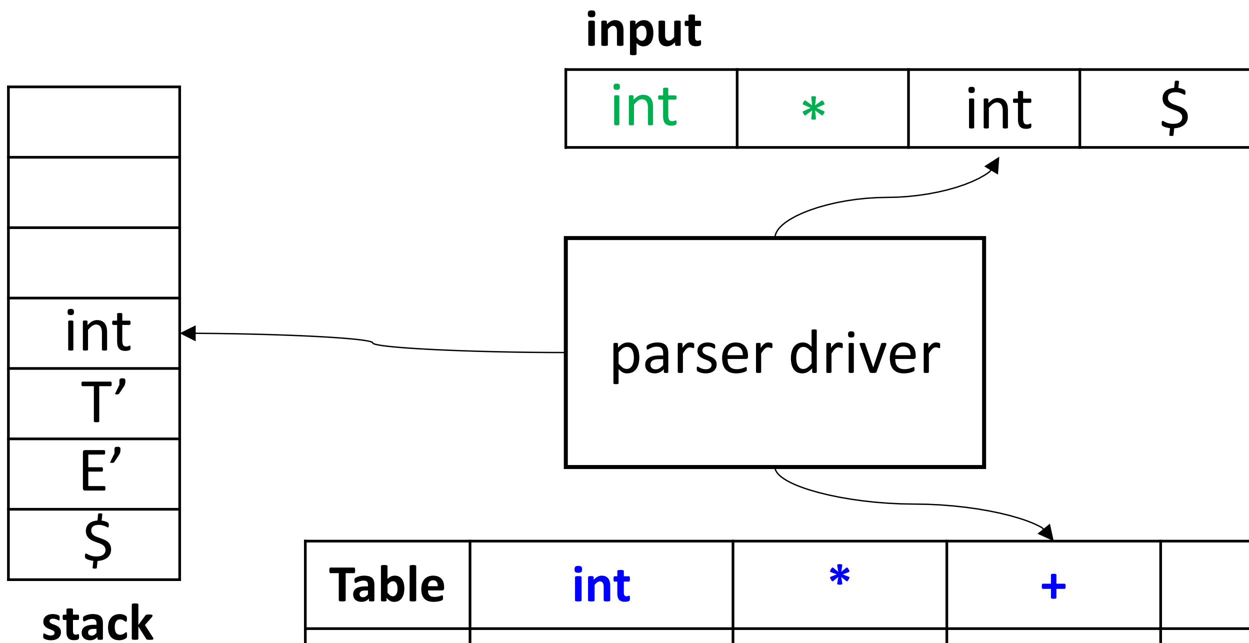


Table	int	*	+	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'			$E' \to +E$		$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to int\ T'$			$T \to (E)$		
T'		$T' \to *T$	$T' \to \varepsilon$		$T' \to \varepsilon$	$T' \to \varepsilon$

Using the Parse Table

- To recognize “int * int”

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

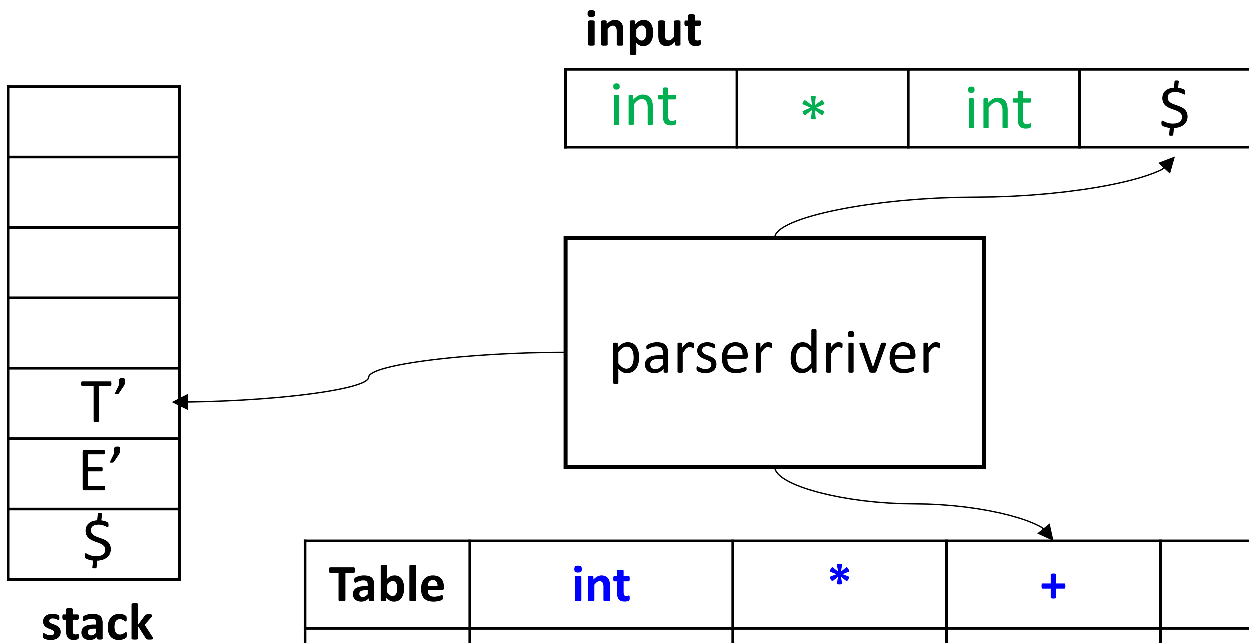
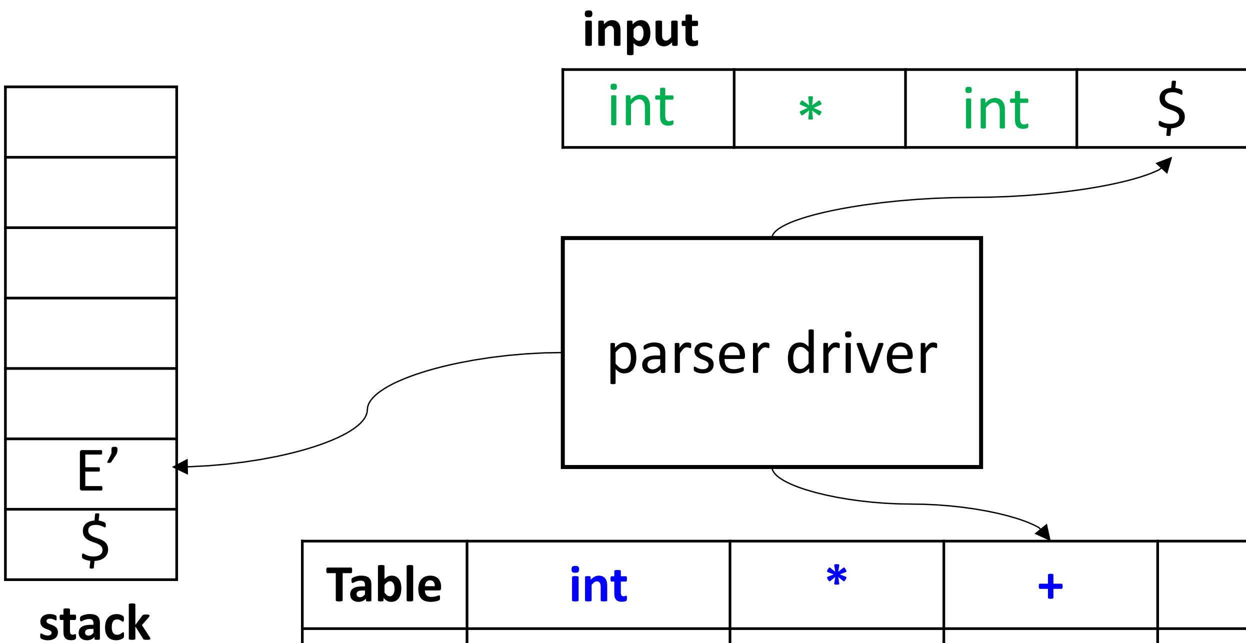


Table	int	*	+	(	)	$
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

Using the Parse Table

- To recognize “int * int”

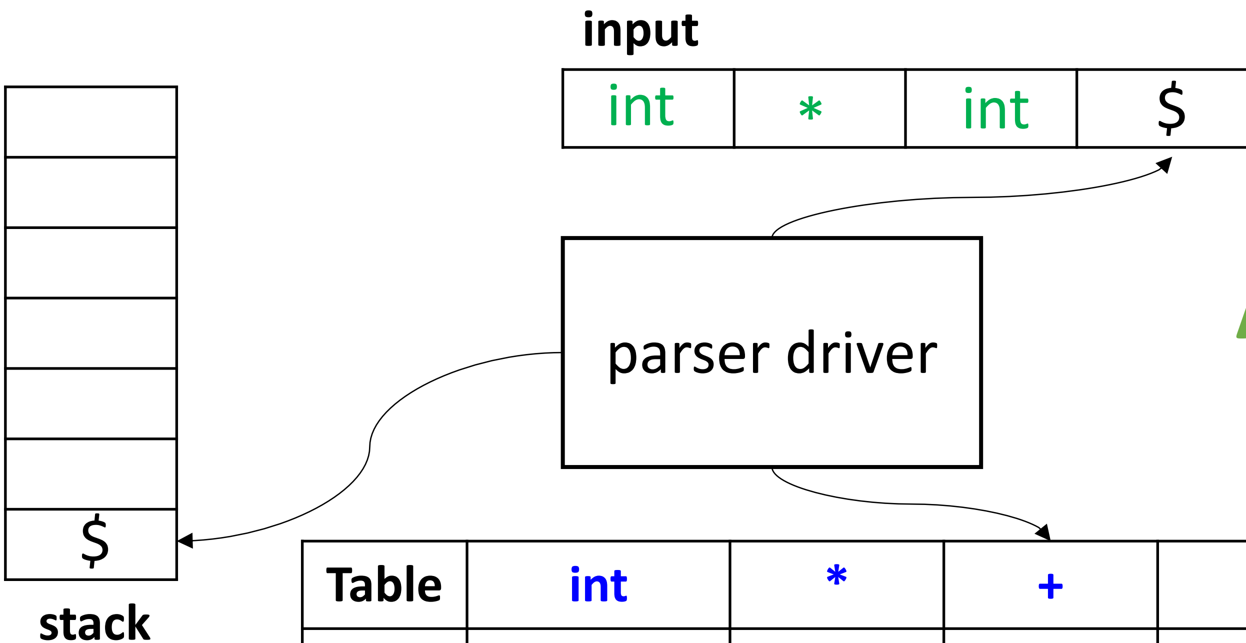


E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

Table	int	*	+	(	)	$
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

Using the Parse Table

- To recognize "int * int"



E → TE'
E' → +E | ε
T → intT' | (E)
T' → *T | ε

ACCEPT!

Table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Recognizing Sequence

Stack	Input	Action
E $	int * int $	E → TE’
T E’ $	int * int $	T → int T’
int T’ E’ $	int * int $	match
T’ E’ $	* int $	T’ → *T
* T E’ $	* int $	match
T E’ $	int $	T → int T’
int T’ E’ $	int $	match
T’ E’ $	$	T’ → ε
E’ $	$	E’ → ε
$	$	Halt and accept

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

- Contents of stack correspond to remaining input
- Actions correspond to productions in leftmost derivation

Review Questions (1)

- What is Recursive Descent?

  Parsing by trying and backtracking to produce the leftmost derivation

- Why do we prefer to use Predictive Parser?

  Requires no backtracking, more efficient

- How to predict the next production to use?

  Next input symbol, current nonterminal being processed

- What are the grammar requirements of predictive parse?

  No common prefix, no left recursion [唯一性]

- What does LL(k) mean?

  L: scans the input from left to right

  L: produces a leftmost derivation

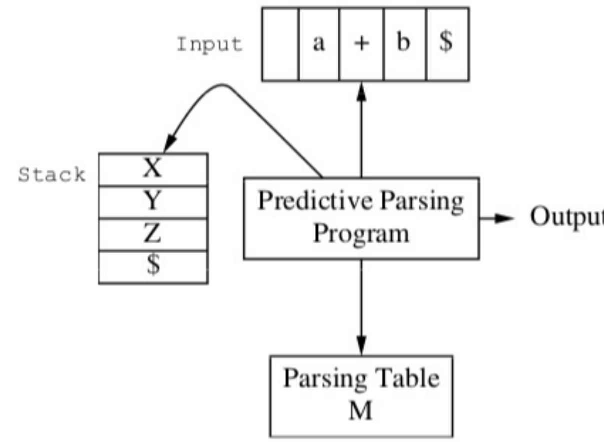
  K: using k input symbols of lookahead

Review Questions (2)

- What is the initial state of the parser?

  Input: input tokens followed by $

  Stack: start symbol followed by $



- General idea of the table-driven parse?

  Expand on non-terminal, match on terminal

- How do we expand?

  If M[X, a] = “X → RHS”, pop X and push RHS to stack

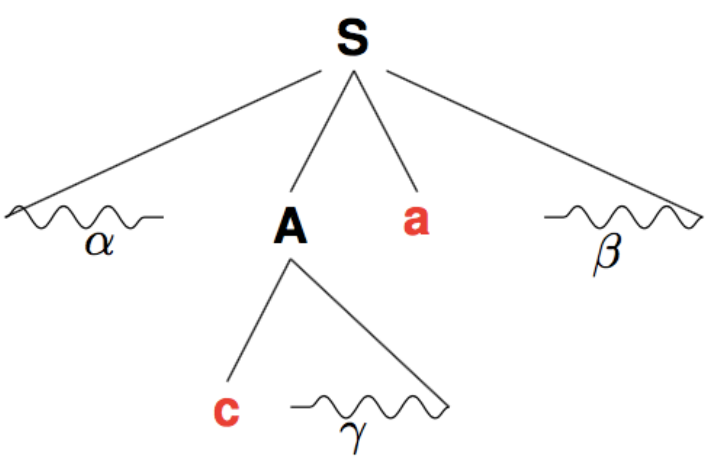
- What are stored in the parsing table?

  Actions the parser should take based on input token and stack top

Table	int	*	+	(	)	$
E	E → TE’			E → TE’		
E’			E’ → +E		E’ → ε	E’ → ε
T	T → int T’			T → (E)		
T’		T’ → *T	T’ → ε		T’ → ε	T’ → ε

To Construct Parsing Table

- The parsing table stores the actions the parser should take based on the input token and the stack top
- The parsing table can be constructed using two sets
  - **FIRST(A)**: set of terminals that begin strings derived from A
    - E.g., c ∈ FIRST(A)
    - If A ⇒* ε, then ε is also in FIRST(A)
  - **FOLLOW(A)**: set of terminals that can appear following A
    - E.g., a ∈ FOLLOW(A)
    - If A is rightmost of a sentential form, then $ is also in FOLLOW(A)



Use FIRST and FOLLOW

- Why do we need FIRST and FOLLOW in parsing?
- FIRST[开始集]
  - FIRST(α): set of terminals that start strings derived from α
  - Consider $A \rightarrow \alpha | \beta$, where FIRST(α) and FIRST(β) are disjoint sets
  - We can then choose by looking at the next input symbol *a*
    - since *a* can be in at most FIRST(α) or FIRST(β), not both
- FOLLOW[后继集]
  - FOLLOW(A): set of terminals that can appear right after *A*
  - If there's a derivation of *A* that results in ε
    - In this case, *A* could be replace by nothing and the next token would be the first token of the symbol following *A* in the sentence being parsed
    - Thus, parser needs to consider to choose the path $A \Rightarrow^* \varepsilon$

Example

Grammar:

S → aBC

B → bC b ∈ FIRST(B)

B → dB d ∈ FIRST(B)

B → ε

C → c c ∈ FOLLOW(B)

C → a a ∈ FOLLOW(B)

D → e

Input: ada

S

⇒ aBC

⇒ adBC

⇒ adC

⇒ ada

Input: ade

S

⇒ aBC

⇒ adBC

⇒ adC

⇒

- Both FIRST and FOLLOW should be used to construct the parsing table

FIRST

- Compute FIRST(X) for all grammar symbols X, apply the following rules until no terminal or ε can be added to any FIRST set
  - If $X \in T$, then FIRST(X)={X} [终结符]
  - If $X \in N$ and $X \rightarrow \varepsilon$ exists, then add ε to FIRST(X) [非终结符，空式]
  - If $X \in N$ and $X \rightarrow Y_1Y_2Y_3...Y_k$, then
    - Add *a* to FIRST(X), if for some *i*, *a* is in FIRST(Y_i), and ε is in all of FIRST(Y_1), …, FIRST(Y_{i-1}), i.e., $Y_1...Y_{i-1} \Rightarrow^* \varepsilon$. E.g.,
      - Everything in FIRST(Y_1) is surely in FIRST(X)
      - If Y_1 doesn't derive ε, then we add nothing more
      - But if $Y1 \Rightarrow^* \varepsilon$, then we add FIRST(Y2), and so on
    - Add ε to FIRST(X), if ε is in FIRST(Y_j) for all j=1,2,…k

FIRST(cont.)

- Compute FIRST(X) for all grammar symbols X [符号]

- Next, we can compute FIRST for any string $\alpha = X_1X_2...X_n$[符号串]
  - Add FIRST(X_1) all non-ε symbols to FIRST(α)
  - Add FIRST(X_i) − ε), 2≤i≤k, to FIRST(α), if FIRST(X_1), ..., FIRST(X_{k-1}) all contain ε
    - Add non-ε symbols of FIRST(X_2), if ε is in FIRST(X_1)
    - Add non-ε symbols of FIRST(X_3), if ε is in FIRST(X_1) and FIRST(X_2)
    - ...
  - Add ε to FIRST(α), if FIRST(X_1), ..., FIRST(X_k) all contain ε

FOLLOW

- To compute FOLLOW(A) to all non-terminals A, apply following rules until no terminal or ε can be added to any FOLLOW set
  - Place \$ in FOLLOW(S), where S is the start symbol
  - If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is in FOLLOW(B)
  - If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$, where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Example: FIRST and FOLLOW

- FIRST(T) = FIRST(E) = {int, (}
  - E has only one production, and its body starts with T
  - T doesn't derive ε, E is same with T
- FIRST(E') = {+, ε}
- FIRST(T') = {*, ε}
- FOLLOW(E) = FOLLOW(E') = {), $}
  - E is start symbol, thus $ must be contained; production body (E)
  - E' appears at the ends of E-productions, same as FOLLOW(E)
- FOLLOW(T) = FOLLOW(T') = {+,), $}
  - +: T appears in bodies only followed by E', thus FIRST(E')- ε
  -), $: FIRST(E') contains ε, and E' is the entire str following T, so FOLLOW(E') is in FOLLOW(T)
  - T' is only at ends of T-productions, FOLLOW(T')=FOLLOW(T)

E → TE'
E'→ +E|ε
T → intT' | (E)
T'→*T|ε

Example: FIRST and FOLLOW (cont.)

Symbol	FIRST	FOLLOW
E	int, (	), $
E’	+, ε	), $
T	int, (	+,), $
T’	*, ε	+,), $

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→*T | ε

A → α (RHS)	FIRST
E → TE’	int, (
E’ → +E	+
T → intT’	int
T → (+E)	(
T’ → *T	*

Construct LL(1) Parse Table

- To construct, rule $A \rightarrow \alpha$ is added to $M[A, a]$ if either:
  - For each terminal a in FIRST(α)
  - If ε is in FIRST(α), or α=ε, a is in FOLLOW(A) (Epsilon production)

- If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ 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 is normally represented by an empty entry in the table

Construct LL(1) Parse Table (cont.)

A → α (RHS)	FIRST
E → TE'	int, (
E' → +E	+
T → intT'	int
T → (E)	(
T' → *T	*
E' → ε	FOLLOW
T' → ε	FOLLOW

Symbol	FIRST	FOLLOW
E	int, (	), $
E'	+, ε	), $
T	int, (	+,), $
T'	*, ε	+,), $

E → TE'
E' → +E | ε
T → intT' | (E)
T' → *T | ε

Table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Table [already examined]

- To recognize "int * int"

$E \rightarrow TE'$

$E' \rightarrow +E \mid \varepsilon$

$T \rightarrow intT' \mid (E)$

$T' \rightarrow *T \mid \varepsilon$

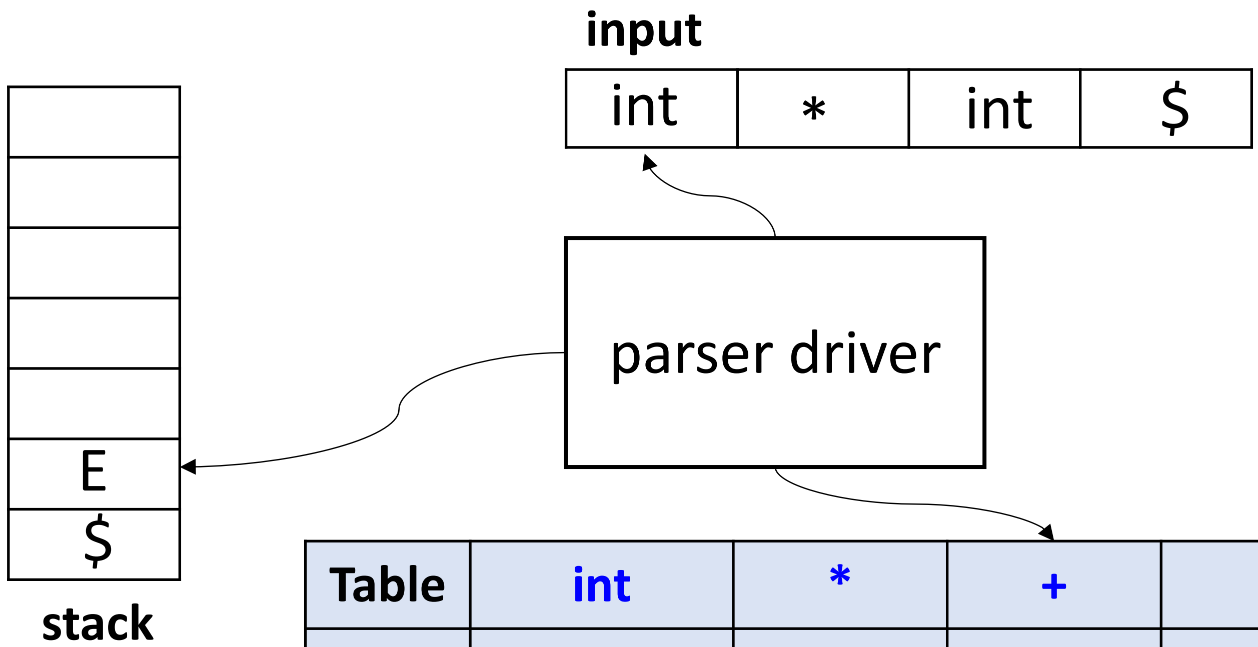


Table	int	*	+	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'			$E' \rightarrow +E$		$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow int\ T'$			$T \rightarrow (E)$		
T'		$T' \rightarrow *T$	$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$

Determine if Grammar is LL(1)

- Observation
  - If a grammar is LL(1), then each of its LL(1) table entry contains at most one rule
  - Otherwise, it is not LL(1).

- Two methods to determine if a grammar is LL(1) or not
  - Construct LL(1) table, and check if there is a multi-rule entry
  - Checking each rule as if the table is getting constructed.

G is LL(1) iff for a rule $A \rightarrow \alpha | \beta$

- $FIRST(\alpha) \cap FIRST(\beta) = \phi$
- At most one of α and β can derive ε
- If β derives ε, then $FIRST(\alpha) \cap FOLLOW(A) = \phi$

保证预测的唯一性

Non-LL(1) Grammars

- Suppose a grammar is not LL(1). What then?
- Case-1: the language may still be LL(1).
  - Try to rewrite grammar to LL(1) grammar:
    - Apply left-factoring
    - Apply left-recursion removal
  - Try to remove ambiguity in grammar:
    - Encode precedence into rules
    - Encode associativity into rules
- Case-2: If Case-1 fails, language may not be LL(1)
  - Impossible to resolve conflict at the grammar level
  - Programmer chooses which rule to use for conflicting entry (if choosing that rule is always semantically correct)
  - Otherwise, use a more powerful parser (e.g. LL(k), LR(1))

LL(1) Time and Space Complexity

- **Linear** time and space relative to length of input
- Time: each input symbol is consumed within a constant number of steps
  - If symbol at top of stack is a terminal:
    - Matched immediately in one step
  - If symbol at top of stack is a non-terminal:
    - Matched in at most *N* steps, where *N* = number of rules
    - Since no left-recursion, cannot apply same rule twice without consuming input
- Space: smaller than input (after removing $X \rightarrow \varepsilon$)
  - RHS is always longer or equal to LHS
    - Derivation string expands monotonically
    - Derivation string is always shorter than final input string
  - Stack is a subset of derivation string (unmatched portion)

Some Thoughts ...

- LL(1) table-driven parser is basically DFA + Stack
  - Capable to count ⇒ CFG is more powerful than RE

- We have studied LL(1), what about LL(0), LL(2) or LL(k)?
- Is **LL(0)** useful at all?
  - Grammar where rules can be predicted with no lookahead
  - ⇒ That means, there can only be one rule per non-terminal
  - ⇒ That means, this language can have only one string
- What would prevent LL(2) ... LL(k) from wide usage?
  - Size of parse table = $O(|N| * |T|^k)$
    - where N = set of non-terminals, T = set of terminals

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Summary: Predictive Parser

- **FIRST** and **FOLLOW** sets are used to construct **predictive parsing tables**

- Intuitively, FIRST and FOLLOW sets guide the choice of rules
  - For non-terminal *A* and lookahead *t*, use the production rule $A \rightarrow \alpha$ where $t \in \text{FIRST}(\alpha)$

    OR
  - For non-terminal *A* and lookahead *t*, use the production rule $A \rightarrow \alpha$ where $\varepsilon \in \text{FIRST}(\alpha)$ and $t \in \text{FOLLOW}(A)$
  - There can only be ONE such rule
    - Otherwise, the grammar is not LL(1)