中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第9讲：语法分析(5)

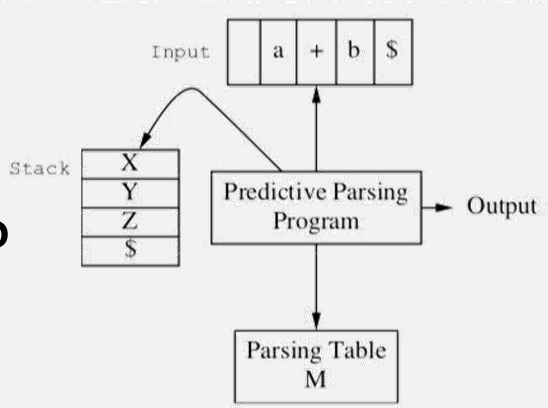
张献伟

xianweiz.github.io

DCS290, 3/28/2024

Review Questions

- Q1: 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
- Q2: is E → E+T | int | int*T a LL(1) grammar?
  NO. Left recursion, common prefix.
- Q3: for table-driven LL(1) parser, what are the components?
  Input buffer, stack, parse table, driver



- Q4: what's the structure of parse table?
  Row: each non-terminal, Col: each terminal + $
  Entry: one production rule, or empty
- Q5: how to predict the next production to use?
  Stack top (current non-terminal), input 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

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

- Implementation with 2D parse table
  - **First column** (each row) lists all non-terminals in the grammar
    - I.e., leftmost non-terminal in derivation
  - **First row** (each col) lists all possible terminals in the grammar and ‘$’
    - I.e., next input token
  - 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

Use the Parse Table

- To recognize "int * int"

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

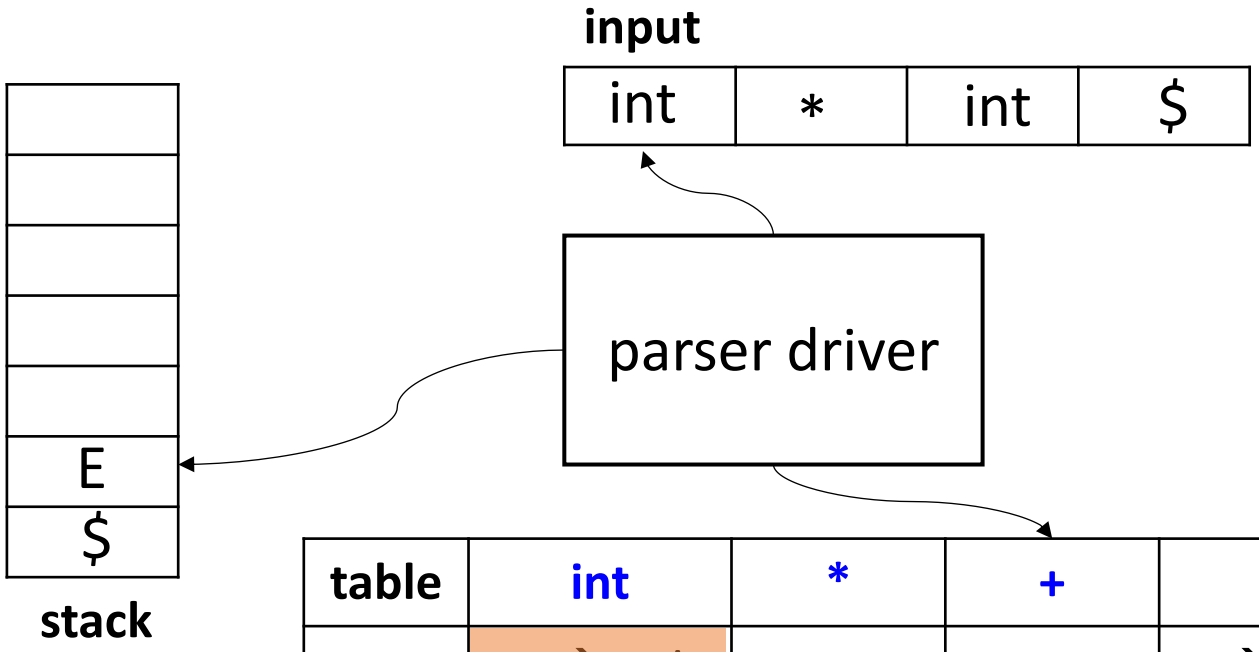


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

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

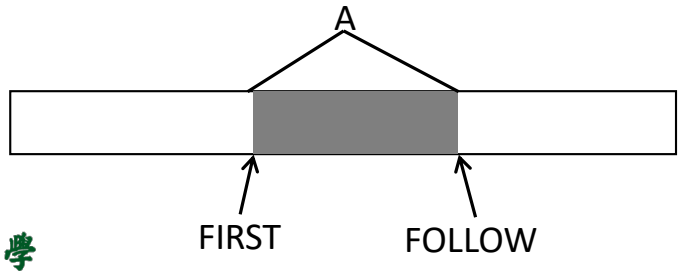
E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

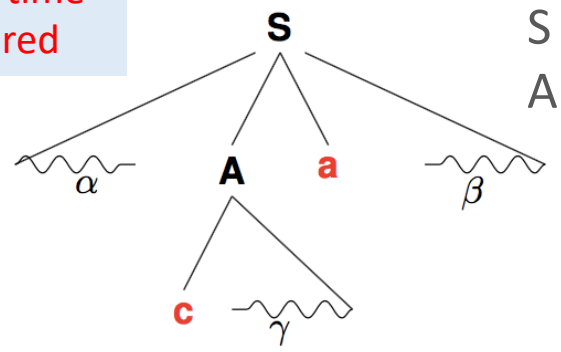
- Sentence int + int is valid
  - E → TE’ → intT’ E’ → int E’ → int +E → int + TE’ → int + intT’E’ → int + intE’ → int + int
  - … E → TE’ → intT’ E’ →
  - … → intT’ E’ → int E’ → int +E (T’ → ε)

To Construct Parse Table[构建解析表]

- 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(α)**: set of terminals that begin strings derived from α
    - E.g., c ∈ FIRST(A) or FIRST(Aaβ)[A推导出的串能以什么符号开始？]
    - 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)[A后能紧跟着什么符号？]
    - If *A* is rightmost symbol of a sentential form, $ is also in FOLLOW(A)

There may have been symbols between *A* and *a*, at some time during derivation, but if so, they derived ε and disappeared





S ⇒* αAaβ

A ⇒* cγ

Use FIRST and FOLLOW

- Why do we need FIRST and FOLLOW in parsing?
  - FIRST and FOLLOW allow to choose which production to apply, based on the next input symbol[提供下一个输入token信息供预测]
- FIRST[开始集]
  - FIRST(α): set of terminals that start strings derived from α
    - α: **any string** of grammar symbols
  - Consider A → α|β, 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

☞ lookahead

- FOLLOW[后继集]
  - FOLLOW(A): set of terminals that can appear right after *A*
    - *A*: **nonterminal**[不在A推导出串的FIRST集，但能紧跟在A后出现]
  - If there's a derivation of *A* that results in ε
    - In this case, *A* could be replaced 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 ⇒* ε

Non-terminal A disappears, without consuming any input symbol.

Example

Grammar:

S → aBC

B → bC

B → dB

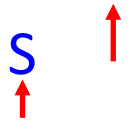
B → ε

C → c

C → a

D → e

Input: ada



Example

Grammar:

S → aBC

B → bC

B → dB

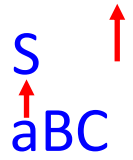
B → ε

C → c

C → a

D → e

Input: ada



Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

Is there a B's production to derive a?

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

Is there a B's production to derive a?

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

⇒ adC

Is there a B's production to derive a?

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

⇒ adC

Is there a B's production to derive a?

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

⇒ adC

⇒ ada

Is there a B's production to derive a?

Example

Grammar:

S → aBC
B → bC
B → dB
B → ε
C → c
C → a
D → e

Input: ada

S
⇒ aBC
⇒ adBC
⇒ adC
⇒ ada

Is there a B's production to derive a?

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

⇒ adC

⇒ ada

Input: ade

Is there a B's production to derive a?

Example

Grammar:

S → aBC
B → bC
B → dB
B → ε
C → c
C → a
D → e

Input: ada

S
⇒ aBC
⇒ adBC
⇒ adC
⇒ ada

Input: ade

S
⇒ aBC
⇒ adBC
⇒ adC
⇒

Is there a B's production to derive a?

Example

Grammar:

S → aBC
B → bC
B → dB
B → ε
C → c
C → a
D → e

Input: ada

S
⇒ aBC
⇒ adBC
⇒ adC
⇒ ada

Input: ade

S
⇒ aBC
⇒ adBC
⇒ adC
⇒

Is there a B's production to derive a?
Again, we choose B → ε. Unfortunately, not work

Example

Grammar:

S → aBC
B → bC
B → dB
B → ε
C → c
C → a
D → e

Input: ada

S
⇒ aBC
⇒ adBC
⇒ adC
⇒ ada

Input: ade

S
⇒ aBC
⇒ adBC
⇒ adC
⇒

Is there a B's production to derive a?
Again, we choose B → ε. Unfortunately, not work

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

⇒

Is there a B's production to derive a?

Again, we choose B → ε. Unfortunately, not work

☞ a is in FIRST(B)/FOLLOW(B), but e is not

- 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 ∈ T, then FIRST(X)={X}[终结符]
  - If X ∈ N and X→ ε exists, then add ε to FIRST(X)[非终结符, 空式]
  - If X ∈ N and X→ $Y_1Y_2Y_3...Y_k$, then[非终结符, 非空式]
    - 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., $Y_1...Y_{i-1}$ ⇒* ε. E.g., [穿透]
      - Everything in FIRST(Y_1) is surely in FIRST(X)
      - If Y_1 doesn't derive ε, then we add nothing more
      - But if Y_1 ⇒* ε, then we add FIRST(Y_2), and so on
    - Add ε to FIRST(X), if ε is in FIRST(Y_j) for all j=1, 2, …, k

产生式

如果X是非终结符，且有产生式形如 X → ABCdEF… (A,B,C均为非终结符且包含ε， d为终结符)，则需要把FIRST(A), FIRST(B), FIRST(C), d加入到FIRST(X)中

FIRST(cont.)

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

- Next, we can compute FIRST for <u>any string</u> $\alpha = X_1X_2...X_n$[符号串]
  - Add FIRST(X_1) all non-ε symbols to FIRST(α)[当然！]
  - Add FIRST(X_i) – ε, $2 \le i \le k$, to FIRST(α), if FIRST(X_1), ..., FIRST(X_{k-1}) all contain ε[前k-1个都透明]
    - 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[α自身可以透明]

FOLLOW

- To compute FOLLOW(X) to all non-terminals X, apply following rules until no terminal or ε can be added to any FOLLOW set
  - Place $ in FOLLOW(X), where X is the start symbol
  - If there is a production A → α*X*β, then everything in FIRST(β) except ε is in FOLLOW(X)
  - If there is a production A → α*X*, or a production A → α*X*β, where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(X)
    - Namely, follow sets are defined in terms of follow sets
    - This step has to be repeated until follow sets converge

右手边

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 (ow, consider E')
- 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 string 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[构建解析表]

- Goal: to put each production into the table entry
  - Each: LL(1)

- To construct, rule A→ α is added to M[A, *a*] if either:
  - For each terminal *a* in FIRST(α)[首先考虑RHS的FIRST集]
  - If ε is in FIRST(α), or α=ε, *a* is in FOLLOW(A) (ε-production)[有空产生式，同时考虑LHS的FOLLOW集]
    - If ε is in FIRST(α) and $ is in FOLLOW(A), add A → α 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	*

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

E → TE’

E’→ +E|ε

T → intT’ | (E)

T’→*T|ε

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

table	int	*	+	(	)	$
E						
E’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T	T → int T’					
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T	T → int T’			T → (E)		
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T	T → int T’			T → (E)		
T’		T’ → *T				

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

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			
T	T → int T’			T → (E)		
T’		T’ → *T				

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

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			
T	T → int T’			T → (E)		
T’		T’ → *T				

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

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				

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				

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				

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

Construct LL(1) Parse Table (cont.)

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

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

E → TE’

E’→ +E|ε

T → intT’ | (E)

T’→*T|ε

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

table	int	*	+	(	)	$
E						
E’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

E → TE’

E’→ +E|ε

T → intT’ | (E)

T’→*T|ε

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

table	int	*	+	(	)	$
E						
E’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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’						
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T						
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T	T → int T’					
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T	T → int T’			T → (E)		
T’						

Construct LL(1) Parse Table (cont.)

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

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			
T	T → int T’			T → (E)		
T’		T’ → *T				

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

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			
T	T → int T’			T → (E)		
T’		T’ → *T				

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

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			
T	T → int T’			T → (E)		
T’		T’ → *T				

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

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				

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				

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				

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

Construct LL(1) Parse Table (cont.)

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

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

What we have used↗

What we just constructed↘

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

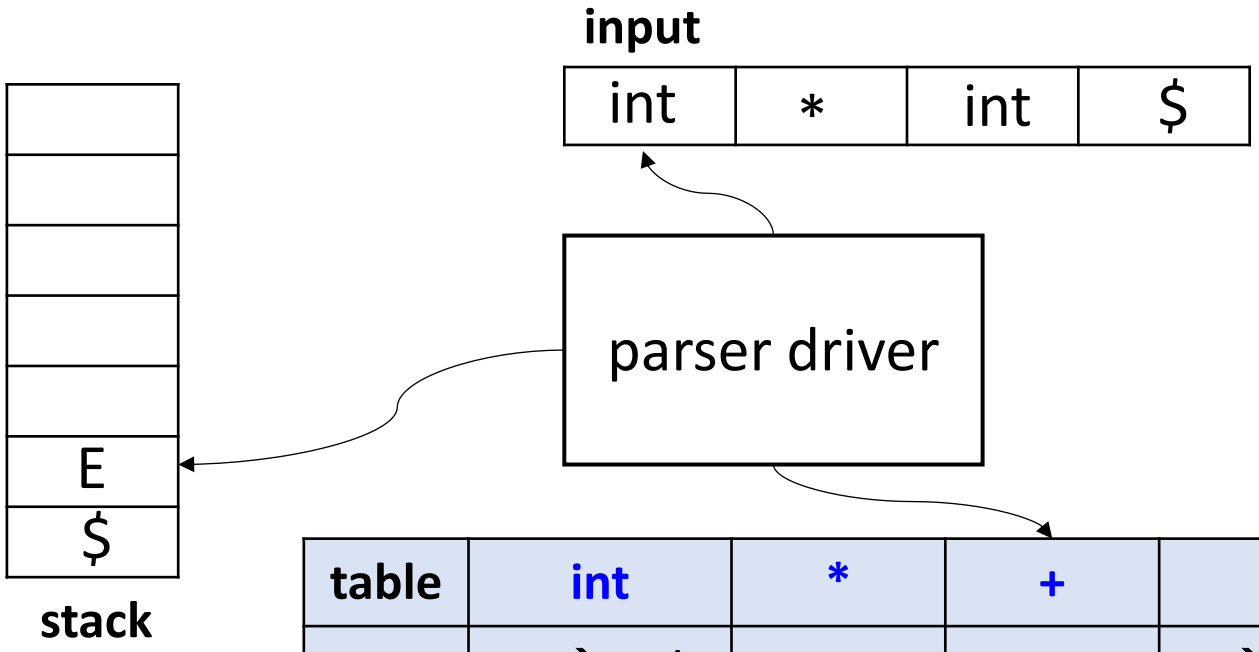


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Determine If Grammar is LL(1)[判断]

- Observation[直观依据]
  - 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)

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

- 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
  - Check each rule as if the table is getting constructed

    G is LL(1) iff for a rule $A \rightarrow \alpha | \beta$
    - FIRST(α) ∩ FIRST(β) = ϕ
    - At most one of α and β can derive ε
    - If β derives ε, then FIRST(α) ∩ FOLLOW(A) = ϕ

保证预测的唯一性

Non-LL(1) Grammars[非LL(1)文法]

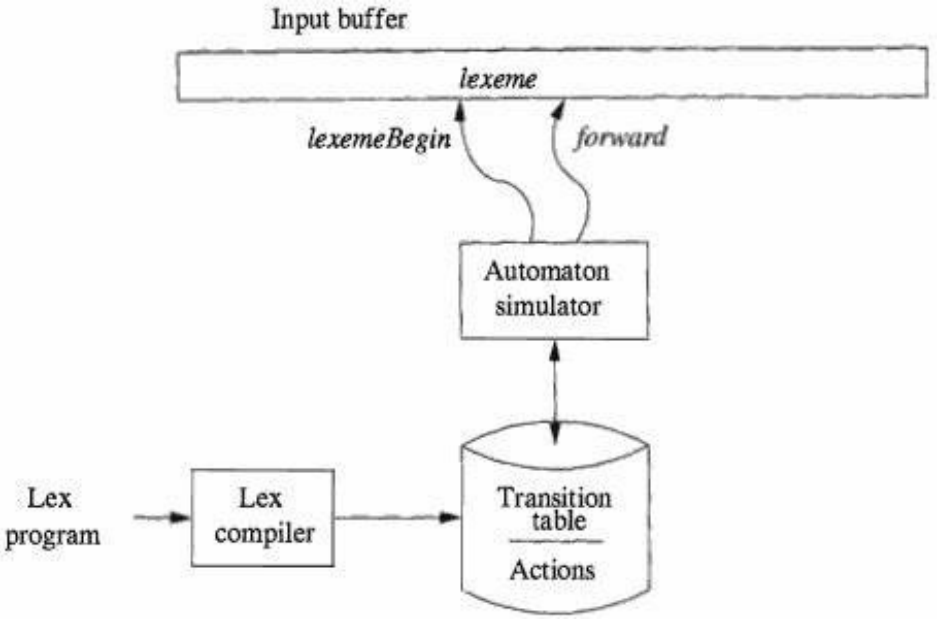
- Suppose a grammar is not LL(1). What then?
- Case-1: the language may still be LL(1)
  - Try to rewrite grammar to LL(1) grammar:
    - Apply left-factoring
    - Apply left-recursion removal
  - Try to remove ambiguity in grammar:
    - Encode precedence into rules
    - Encode associativity into 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))

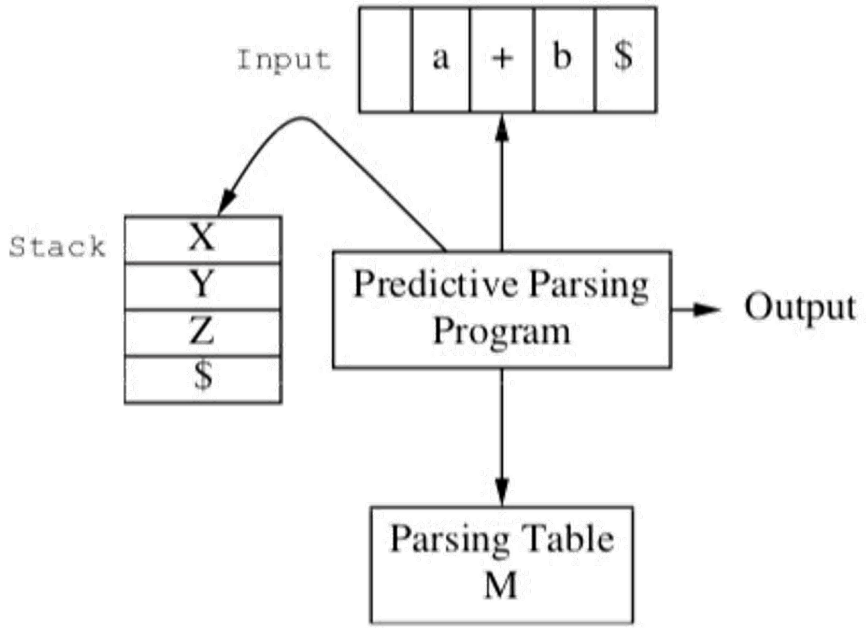
LL(1) Time and Space Complexity[复杂度]

- **Linear** time and space relative to length of input[线性]
- Time: each input symbol is consumed within a constant number of steps
  - If symbol at top of stack is a terminal:[栈顶是终结符，比对]
    - Matched immediately in one step
  - If symbol at top of stack is a non-terminal:[栈顶是非终结符，展开待比对]
    - 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 → ε)
  - RHS is always longer or equal to LHS
    - Derivation string expands monotonically
    - Derivation string is always shorter than input string[input即最终扩展态]
  - 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





Some Thoughts … (cont.)

- 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

Consider a CFG grammar G

S→AB A→aC B→bD D→d C→c

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

Summary: Predictive Parser[小结]

- **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 → α where *t* ∈ FIRST(α)

    OR
  - For non-terminal *A* and lookahead *t*, use the production rule *A* → *α* where ε ∈ FIRST(α) and *t* ∈ FOLLOW(A)
  - There can only be ONE such rule
    - Otherwise, the grammar is not LL(1)