Compilation Principle
编 译 原 理

第8讲：语法分析(5)

张献伟

xianweiz.github.io

DCS290, 3/17/2022

Review Questions

- Q1: why do we prefer to use Predictive Parser?

  Requires no backtracking, more efficient.

- Q2: how to predict the next production to use?

  Current nonterminal being processed, next input symbol(s).

- Q3: can predictive parser handle E → E+T | int | int*T?

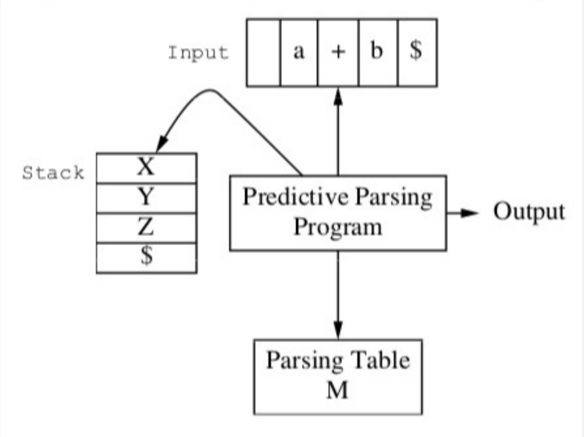
  NO. Left recursion, common prefix.

- Q4: 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



- Q5: what is the initial state of the parser?

  Input: input tokens followed by $

  Stack: start symbol followed by $

Use the Parse Table

- To recognize "int * int"

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

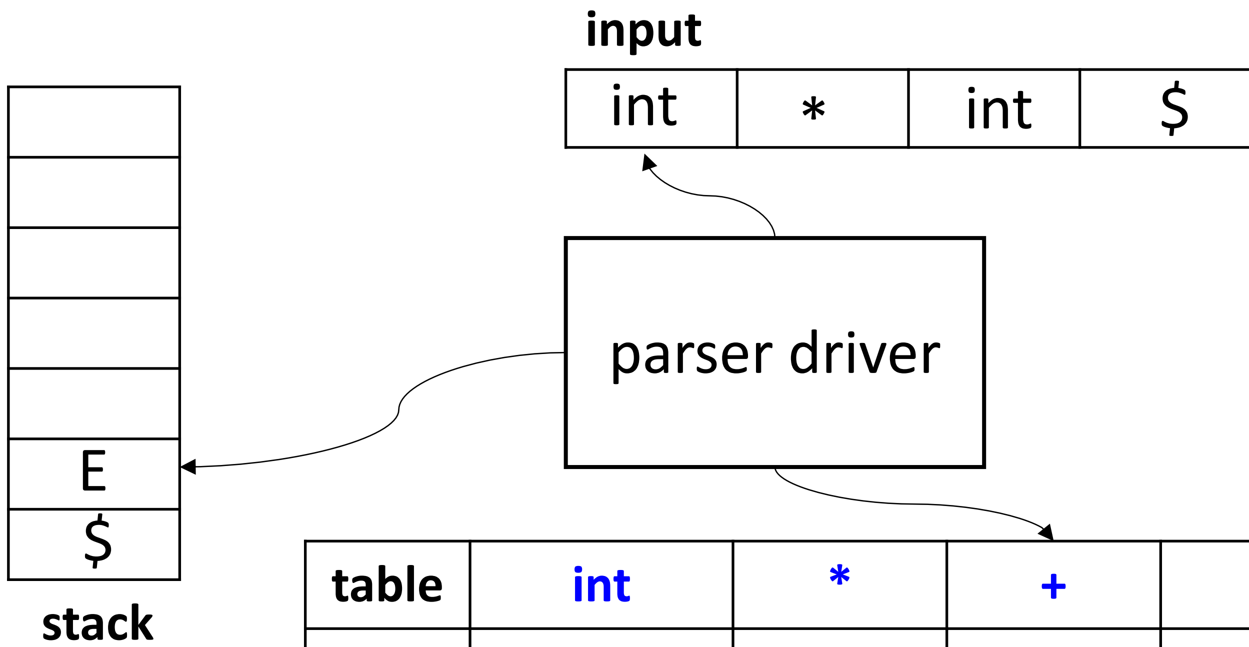
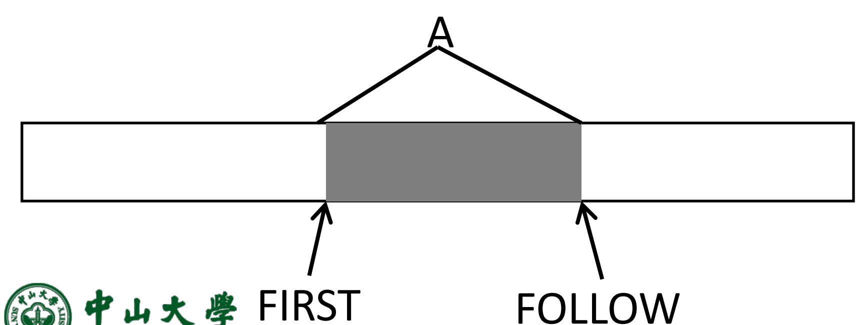
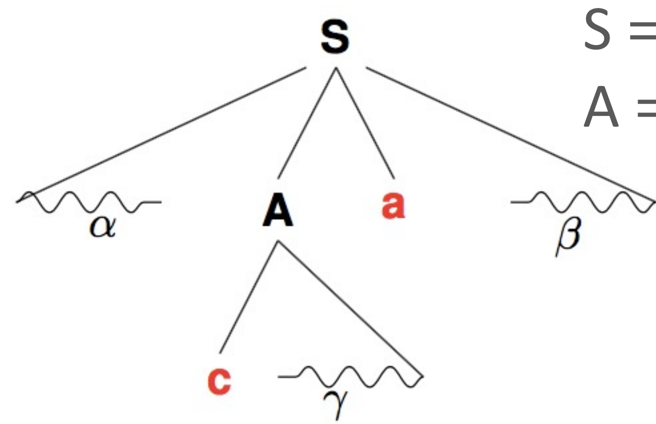


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

To Construct Parse Table[构建解析表]

- The parsing table stores the actions the parser should take based on the <u>input token</u> and the <u>stack top</u>
- 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)
    - If $A \Rightarrow^* \varepsilon$, 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 symbol of a sentential form, $ is also in FOLLOW(A)





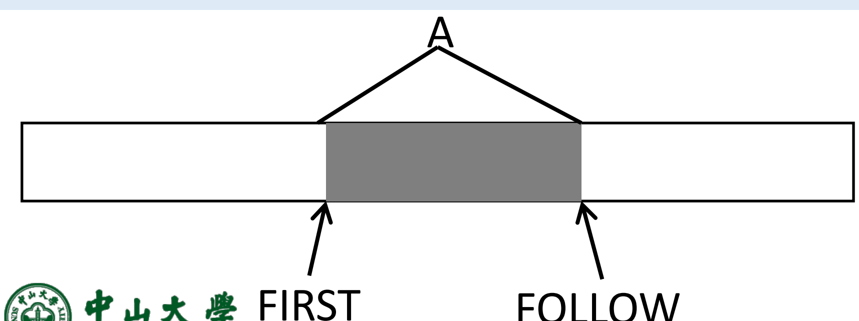
$S \Rightarrow^* \alpha A a \beta$

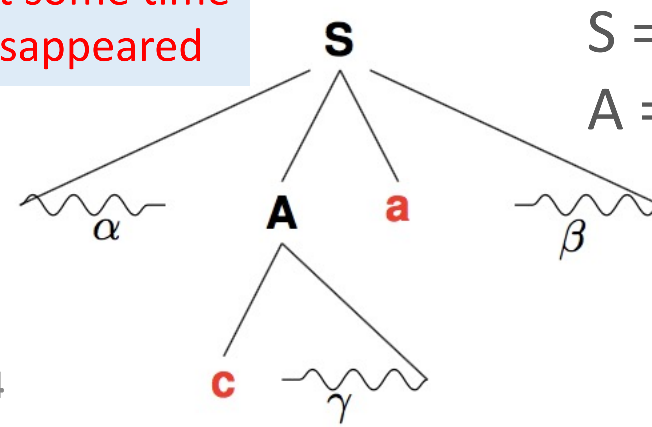
$A \Rightarrow^* c\gamma$

To Construct Parse Table[构建解析表]

- The parsing table stores the actions the parser should take based on the <u>input token</u> and the <u>stack top</u>
- 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)
    - If $A \Rightarrow^* \varepsilon$, 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 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





$S \Rightarrow^* \alpha A a \beta$

$A \Rightarrow^* c\gamma$

Use FIRST and FOLLOW

- Why do we need FIRST and FOLLOW in parsing?

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

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
- FIRST[开始集]
  - FIRST(α): set of terminals that start strings derived from α
    - α: **any string** of grammar symbols
  - Consider $A \rightarrow \alpha | \beta$, where FIRST($\alpha$) and FIRST($\beta$) 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

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
- FIRST[开始集]
  - FIRST(α): set of terminals that start strings derived from α
    - α: **any string** of grammar symbols
  - Consider $A \rightarrow \alpha | \beta$, where FIRST($\alpha$) and FIRST($\beta$) 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

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
- FIRST[开始集]
  - FIRST(α): set of terminals that start strings derived from α
    - α: **<u>any string</u>** of grammar symbols
  - Consider $A \rightarrow \alpha|\beta$, where FIRST($\alpha$) and FIRST($\beta$) 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*: **<u>nonterminal</u>**
  - 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 $A \Rightarrow^* \varepsilon$

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
- FIRST[开始集]
  - FIRST(α): set of terminals that start strings derived from α
    - α: **<u>any string</u>** of grammar symbols
  - Consider $A \rightarrow \alpha | \beta$, where FIRST($\alpha$) and FIRST($\beta$) 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*: **<u>nonterminal</u>**
  - 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 $A \Rightarrow^* \varepsilon$

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

S

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

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:

$S \rightarrow aBC$

$B \rightarrow bC$

$B \rightarrow dB$

$B \rightarrow \varepsilon$

$C \rightarrow c$

$C \rightarrow a$

$D \rightarrow e$

Input: ada

S

$\Rightarrow aBC$

$\Rightarrow adBC$

Example

Grammar:

S → aBC

B → bC

B → dB

B → ε

C → c

C → a

D → e

Input: ada

S

⇒ aBC

⇒ adBC

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Example

Grammar:

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

Input: ada

S
⇒ aBC
⇒ adBC
⇒ adC

Example

Grammar:

$S \rightarrow aBC$

$B \rightarrow bC$

$B \rightarrow dB$

$B \rightarrow \varepsilon$

$C \rightarrow c$

$C \rightarrow a$

$D \rightarrow e$

Input: ada

S

$\Rightarrow aBC$

$\Rightarrow adBC$

$\Rightarrow adC$

Example

Grammar:

$S \rightarrow aBC$

$B \rightarrow bC$

$B \rightarrow dB$

$B \rightarrow \varepsilon$

$C \rightarrow c$

$C \rightarrow a$

$D \rightarrow e$

Input: ada

S

$\Rightarrow aBC$

$\Rightarrow adBC$

$\Rightarrow adC$

$\Rightarrow ada$

Example

Grammar:

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

Input: ada

S
⇒ aBC
⇒ adBC
⇒ adC
⇒ ada

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

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

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
⇒

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
⇒

Example

Grammar:

$S \rightarrow aBC$

$B \rightarrow bC$ **b ∈ FIRST(B)**

$B \rightarrow dB$ **d ∈ FIRST(B)**

$B \rightarrow \varepsilon$

$C \rightarrow c$ **c ∈ FOLLOW(B)**

$C \rightarrow a$ **a ∈ FOLLOW(B)**

$D \rightarrow e$

Input: ada

S

$\Rightarrow aBC$

$\Rightarrow adBC$

$\Rightarrow adC$

$\Rightarrow ada$

Input: ade

S

$\Rightarrow aBC$

$\Rightarrow adBC$

$\Rightarrow adC$

$\Rightarrow$

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[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., $Y_1...Y_{i-1} \Rightarrow^* \varepsilon$. 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 $Y_1 \Rightarrow^* \varepsilon$, 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

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> $\alpha = X_1X_2 \ldots X_n$[符号串]
  - Add FIRST(X_1) all non-ε symbols to FIRST(α)[当然！]
  - Add FIRST(X_i) − ε, $2 \leq i \leq k$, 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(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)
    - Namely, follow sets are defined in terms of follow sets
    - This step has to be repeated until follow sets converge

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

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}

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

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)

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

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[构建解析表]

- Goal: to put each production into the table entry

- To construct, rule $A \rightarrow \alpha$ 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 $A \rightarrow \alpha$ 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	*

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’						

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

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’	$\boxed{\text{int, (}}$
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				

SUN YAT-SEN UNIVERSITY

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				

中山大學 SUN YAT-SEN UNIVERSITY

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				

SUN YAT-SEN UNIVERSITY 中山大學

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

Use the Table [already examined]

- To recognize "int * int"

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

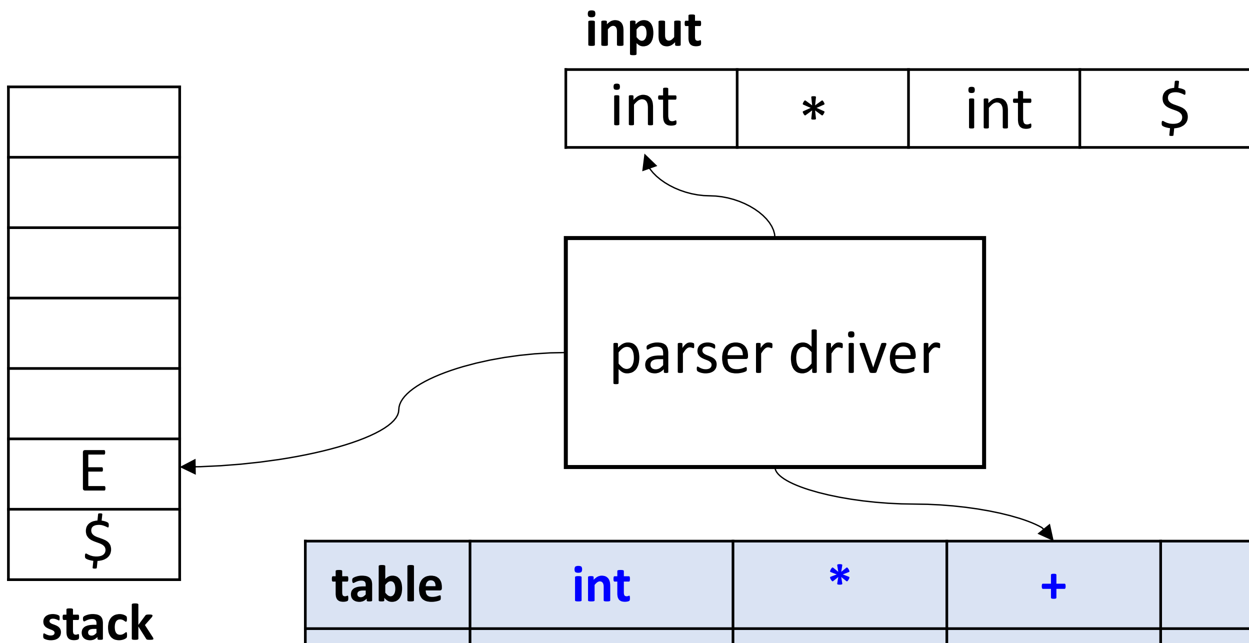


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

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 → α|β

    - 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

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 $A \to \alpha|\beta$

- $FIRST(\alpha) \cap FIRST(\beta) = \phi$
- 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[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:
    - 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

Some Thoughts …

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

- We have studied LL(1), what about LL(0), LL(2) or LL(k)?

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?

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

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

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

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?

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

Summary: Predictive Parser[小结]

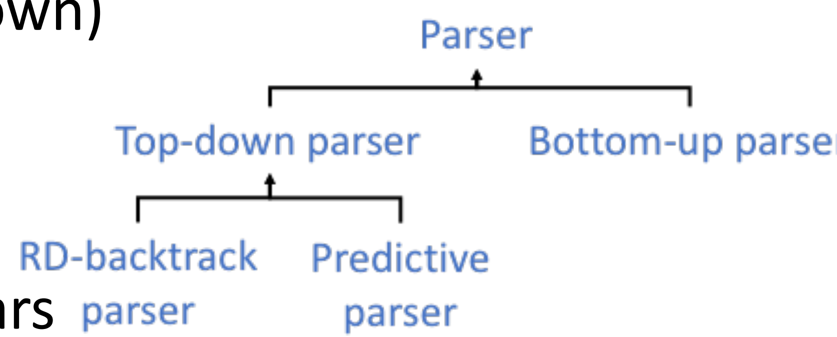
- **FIRST** and **FOLLOW** sets are used to construct **predictive parsing tables**

- 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)

Bottom-up Parsing[自底向上]

- Begins at leaves and works to the top[叶子到根]
  - Bottom-up: **reduce**s[归约] input string to start symbol
  - In the <u>opposite direction</u> from top-down
    - Top-down: expands start symbol to input string
  - In <u>reverse order of rightmost derivation</u> (In effect, builds tree from left to right, just like top-down)



- More powerful than top down
  - Don’t need left factored grammars
  - Can handle left recursion
  - Can express a larger set of languages
  - And just as efficient

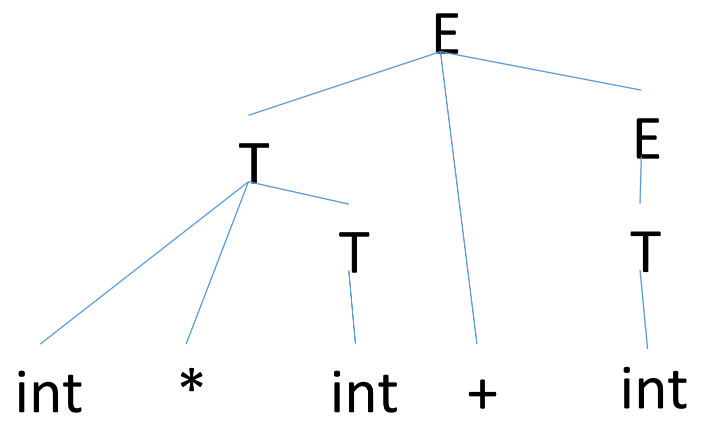
Example

- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String: int * int + int



- The rightmost derivation of the parse tree
  - $E \Rightarrow T + E \Rightarrow T + T \Rightarrow T + int \Rightarrow int * T + int \Rightarrow int * int + int$

- To recognize the string via bottom-up parsing
  - $int * int + int \Rightarrow int * T + int \Rightarrow T + int \Rightarrow T + T \Rightarrow T + E \Rightarrow E$

Bottom-up: Overview

- An important fact:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \to \beta$
  - Then ω is a string of terminals [i.e., 句子]

Bottom-up: Overview

- An important fact:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \rightarrow \beta$
  - Then ω is a string of terminals [i.e., 句子]
- Why?

Bottom-up: Overview

- An important fact:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \rightarrow \beta$
  - Then ω is a string of terminals [i.e., 句子]
- Why? $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a rightmost derivation

Bottom-up: Overview

- An important fact:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \rightarrow \beta$
  - Then ω is a string of terminals [i.e., 句子]
- Why? $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a rightmost derivation
- **Idea**: split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)[右侧尚未处理]
  - Left substring has terminals and non-terminals[左侧已有处理]
- The dividing point is marked by a #
  - The # is not part of the string
  - Initially, all input is unexamined $\#x_1x_2 \ldots x_n$

Bottom-up: Shift-Reduce[移入-归约]

- Bottom-up parsing is also known as **Shift-Reduce** parsing
  - Involves two types of operations: shift and reduce

- **Shift**[移入]: move # one place to the right
  - Shifts a terminal to the left string[向左侧移入终结符]

    ABC#xyz ⇒ ABCx#yz

- **Reduce**[归约]: apply an inverse production at the right end of the left string[左侧的右端进行规约]
  - If E → Cx is a production, then

    ABCx#yz ⇒ ABE#yz

The Example

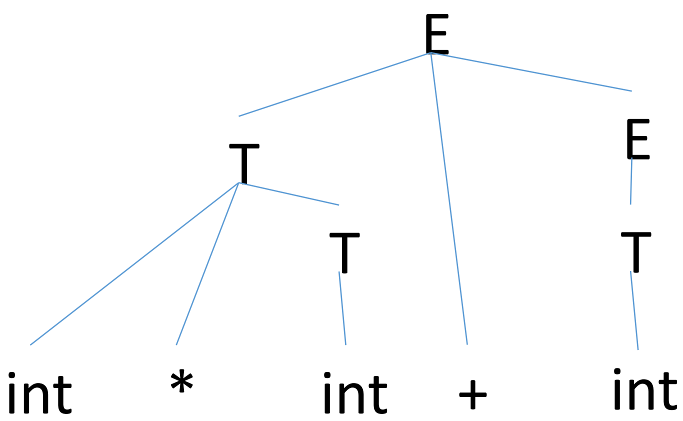
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Shift
T + T #	Reduce T → int
T + E #	Reduce E → T
E #	Reduce E → T+E

Stack[栈]

- Left string can be stored into a **stack**
  - Top of the stack is the #

- **Shift** pushes a terminal on the stack

- **Reduce** does the following:
  - pops zero or more symbols off of the stack
    - production rhs[pop出了产生式RHS]
  - pushes a non-terminal on the stack
    - production lhs[push进了产生式LHS]
  - just reverts production (LHS ← RHS)

Step
#int * int + int
int# * int + int
int * #int + int
int * int # + int
int * T # + int
T # + int
T + # int
T + int #
T + T #
T + E #
E #

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Key Issue[一个关键问题]

- How to decide when to shift or reduce?
  - Example grammar:

    E → T+E | T

    T → int*T | int | (E)

#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
… … …	

#int * int + int	Shift
int# * int + int	Reduce T → int
T # * int + int	Shift
… … …	

  - Consider the step int # * int + int
  - We could reduce by T → int giving T#*int + int
    - **A fatal mistake**: no way to reduce to the start symbol E

- Intuition: want to reduce only if the result can still be reduced to the start symbol[必须在对的方向上]