Compiler Design

Session - 4

Compiler Design

For the grammar below, a partial LL(1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E1, E2, and E3. [Tex]\\epsilon[/Tex] is the empty string, $ indicates end of input, and, | separates alternate right hand sides of productions.

S → a A b B | b A a B | ε
A → S
B → S

	a	b	$
S	**E1**, S→ε	**E2**, S→ε	S → ε
A	A → S	A → S	error
B	B → S	B → S	**E3**

E1: S→aAbB, S→ε

E2: bAaB, S→ε

E3: B→S

(A) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b, $}

(B) FIRST(A) = {a, b, $}
FIRST(B) = {a, b, ε}
FOLLOW(A) = {a, b}
FOLLOW(B) = {$}

(C) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = ∅

(D) FIRST(A) = {a, b} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

Compiler Design

Consider the date same as above question. The appropriate entries for E1, E2, and E3 are

(A) E1: S → aAbB, A → S
E2: S → bAaB, B → S
E3: B → S

(B) E1: S → aAbB, S → ε
E2: S → bAaB, S → ε
E3: S → ε

(C) E1: S → aAbB, S → ε
E2: S → bAaB, S → ε
E3: B → S

(D) E1: A → S, S → ε
E2: B → S, S → ε
E3: B → S

Compiler Design

Consider the following grammar:

```
S → FR
R → S | ε
F → id
```

In the predictive parser table, M, of the grammar the entries M[S, id] and M[R, $] respectively.

- [ ] {S → FR} and {R → ε }
- [ ] {S → FR} and { }
- [ ] {S → FR} and {R → *S}
- [ ] {F → id} and {R → ε}

Compiler Design

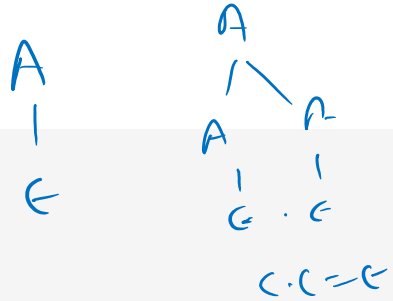
The grammar A → AA | (A) | ε is not suitable for predictive-parsing because the grammar is

- [x] ambiguous
- [x] left-recursive
- [ ] right-recursive
- [ ] an operator-grammar

LL(1) → (nonleftRecursive ∧ leftfactored ∧ UnAmbiguous)

✗ (nonleftRecursive ∧ leftfactor ∧ UnAmbiguous) → LL(1)

(leftRecursive ∨ leftfactored ∨ Ambiguous) → LL(1)



Compiler Design

Which one of the following grammars is free from left recursion?

(A)
$$S \rightarrow AB$$
$$A \rightarrow Aa \mid b$$
$$B \rightarrow c$$

(B)
$$S \rightarrow Ab \mid Bb \mid c$$
$$A \rightarrow Bd \mid \varepsilon$$
$$B \rightarrow e$$

Direct left Recursion

(C)
$$S \rightarrow Aa \mid B$$
$$A \rightarrow Bb \mid Sc \mid \varepsilon$$
$$B \rightarrow d$$

Indirect left Recursion

(D)
$$S \rightarrow Aa \mid Bb \mid c$$
$$A \rightarrow Bd \mid \varepsilon$$
$$B \rightarrow Ae \mid \varepsilon$$

Indirect left Recursion

Compiler Design

follow(S) = {a, $}

Properties of LL(1) Grammar:

1- A left Recursive Grammar is not LL(1): G: {S → Sa | b}

first(S) = {b}

	a	$
S	S → aSa	S → ε

2- A left factor Grammar is not LL(1)

G: S → a S | a | b

	a	b	$
S	S → aS S → ε		→ not LL(1)

3- An Ambiguous Grammar is not LL(1)

4- Every LL(1) Grammar is Unambiguous But every Unambiguous Grammar may not be LL(1)

5- If Any Production are of the form $A \to \alpha_1 | \alpha_2$ Then

$First(\alpha_1) \cap First(\alpha_2) \neq \phi \Rightarrow$ not LL(1)

$First(\alpha_1) \cap First(\alpha_2) = \phi \Rightarrow$ LL(1)

6- If Any Production of the form $A \to \alpha_1 | \alpha_2 | \alpha_3 | \dots | \alpha_n$ Then Any two Production having Common elements in firstset then it is not LL(1)

7- If Any Production are of the form $A \to \alpha | \epsilon$ Then if $First(\alpha) \cap follow(A) \neq \phi$ not LL(1)

S → aSa | ε

Compiler Design

left factor { left factored / Not left factoring }

NOTE: not LL(1)

1) If Any entry of table is multiple defined then G is not LL(1)
2) If G is Ambiguous
3) If G is Left Recursi
4) If G is Left factor (Nondeterministic)
5) other case

Compiler Design

Not LL(1):

1- if any entry of table is multiple defined then G is not LL(1)

2- if G is Ambiguous

3- if G is Left Recursive

4- if G is Left factor

5- And some other cases also

$S \to aSa \mid \epsilon$

first(aSa) ∩ follow(S) ≠ ∅

= a ≠ ∅

Ex: G_1: $S \to AB \mid ab$

$A \to a$

$B \to b$

not LL(1) b/c Ambiguous Left factoring

G_2: $S \to Sa \mid b$

not LL(1) Left Recursive

G_3: $S \to a \mid ab$

not LL(1) left factor

G_4: $S \to aSa \mid \epsilon$

non left Recursive

Left factored

UnAmbiguous

not LL(1)

Compiler Design

Testing LL(1) Grammar:

Cases: $A \to \alpha_1 | \alpha_2 | \alpha_3$

$\in \notin first(\alpha_1)$

$\in \notin first(\alpha_2)$

$\in \notin first(\alpha_3)$

LL(1): $first(\alpha_1) \cap first(\alpha_2) = \phi$

$first(\alpha_1) \cap first(\alpha_3) = \phi$

$first(\alpha_2) \cap first(\alpha_3) = \phi$

Case2: $A \to \alpha_1 | \alpha_2 | \alpha_3$

$\in \notin first(\alpha_1)$

$\in \notin first(\alpha_2)$

$\in \in first(\alpha_3)$

LL(1): $first(\alpha_1) \cap first(\alpha_2) = \phi$

$first(\alpha_1) \cap [(first(\alpha_3) - \{\in\}) \cup follow(A)] = \phi$

$first(\alpha_2) \cap [(first(\alpha_3) - \{\in\}) \cup follow(A)] = \phi$

Case3: $A \to \alpha_1 | \alpha_2 | \alpha_3$

$\in \notin first(\alpha_1)$

$\in \in first(\alpha_2)$

$\in \in first(\alpha_3)$

$first(\alpha_1) \cap [(first(\alpha_2) - \in) \cup follow(A)] = \phi$

$first(\alpha_1) \cap [(first(\alpha_3) - \in) \cup follow(A)] = \phi$

$[(first(\alpha_2) - \{\in\}) \cup follow(A)] \cap [(first(\alpha_3) - \{\in\}) \cup follow(A)] \neq \phi$

not LL(1)

Compiler Design

$G = \{S \to aS \mid Ab,\ A \to aA \mid b\}$ ⇒ not LL(1)

$first(aS) \cap first(Ab) \neq \phi$

$a \cap \{a, b\} = a \neq \phi$

$G = \{S \to SaAb \mid bA,\ A \to bA \mid \in\}$

not LL(1)

$G = \begin{cases} S \to AaBb \mid \in \\ A \to Bb \mid b \\ B \to aB \mid \in \end{cases}$ ⇒

not LL(1)

$first(AaBb) \cap follow(S)$

$\{a, \in\} \cap \{\$\} = \phi$

$first(Bb) \cap first(b)$

$\{a, \in b\} \cap \{b\} \neq \phi$ b

⇒

$First(aB) \cap follow(B)$

$a \cap \{$

Compiler Design

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \in$

$E \rightarrow a$

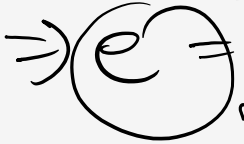
$follow(S) = \{e, \$\}$

$follow(S') = \{e, \$\}$

$\Rightarrow \phi$

$first(eS) \cap follow(S')$

$\{e\} \cap \{e, \$\} \neq \phi$

$\Rightarrow$  not LL(1)

Compiler Design

LL(K) :-

G_1: $S \to aA$, $A \to b$ — LL(0)

G_2: $S \to abaA \mid bbA$, $A \to c \mid d$ — $\neq LL(0)$, LL(1)

G_3: $S \to abA \mid aaA$, $A \to c \mid d$ — $\neq LL(0)$, $\neq LL(1)$, LL(2)

G_4: $S \to abaA \mid abbA$, $A \to c \mid d$ — LL(3), $\neq LL(0)$, $\neq LL(1)$, $\neq LL(2)$

$LL(0) \Rightarrow LL(1) \Rightarrow LL(2) \Rightarrow LL(3) \Rightarrow LL(4) \Rightarrow \cdots \cdots LL(K)$

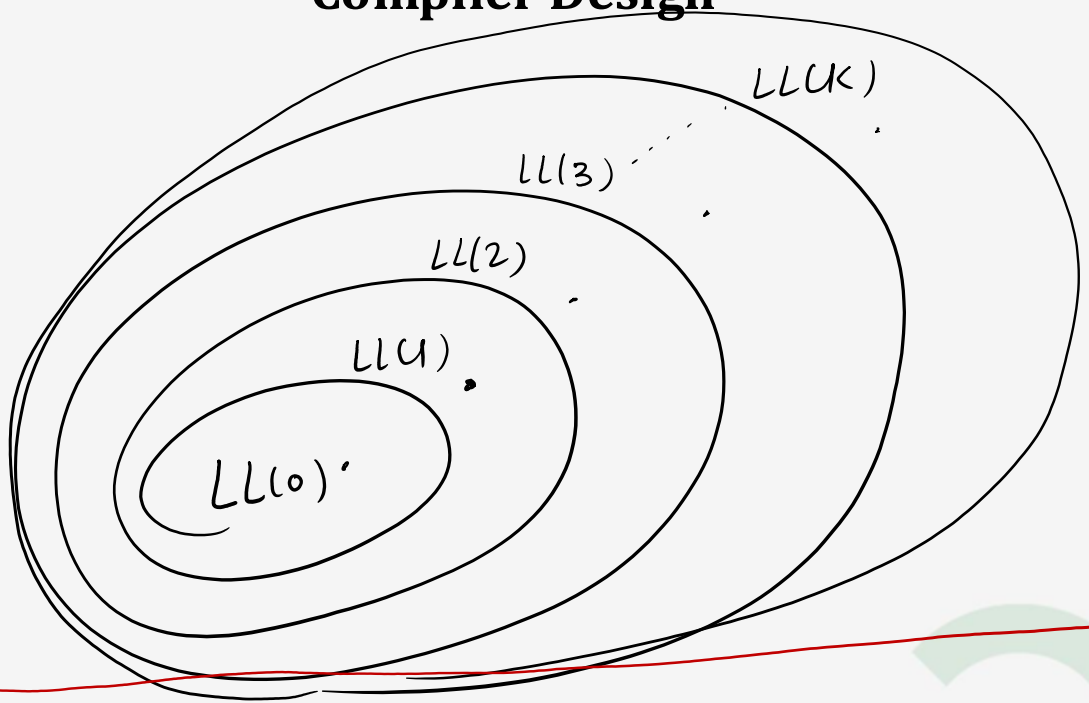
$LL(K) \Rightarrow LL(K+1)$

$LL(K+1) \Rightarrow LL(K)$

~~$LL(K+1) \Rightarrow LL(K)$~~

$x \to y \equiv \sim y \to \sim x$

Compiler Design



Every: $LL(0) \subset LL(1) \subset LL(2) \subset LL(3) \subset LL(4) \cdots \subset LL(K)$

Compiler Design

Question:- Which of the following grammar is LL(3) but not LL(2) and not LL(1)

a) $\{S \to aS \mid b\}$ $\Rightarrow LL(1) = \neq LL(0)$

b) $\{S \to aaS \mid ab \mid b\}$ $\Rightarrow LL(2) \neq LL(1)$

c) $\{S \to aaaS \mid aab \mid ab \mid b\}$ $\Rightarrow LL(3)$
$\neq LL(2)$
$\neq LL(1)$
$\neq LL(2)$

d) None

as b

LL(1)
LL(2)

Compiler Design

Question: The following Grammar G = {S → aA | b, A → aB | a, B → bB | a}

LL(2)

a) LL(1) but not LL(2)

b) LL(2) but not LL(1)

c) Both LL(1) & LL(2)

d) Neither LL(1) nor LL(2)

LL(2)

S → aA | b

First(aA) ∩ first(b)

a{a} ∩ {b}

aa ∩ b = ϕ ✓

not LL(1)

A → aB | a

First(aB) ∩ first(a)

a{first(B)} ∩ {a}

a{a, b} ∩ {a}

(aa, ab) ∩ a = ϕ ✓

B → bB | a

first(bB) ∩ first(a)

b{first(B)} ∩ {a}

b{a, b} ∩ {a}

{ba, bb} ∩ {a} = ϕ ✓

Compiler Design

Question: $G = \{ S \to aA/ab,\ A \to aA/b \}$ $\Rightarrow$ not LL(1)

LL(2)

First(aA) $\cap$ First(ab)

ϕ a $\{a, b\} \cap \{ab\}$

$\{aa, ab\} \cap \{ab\} = ab \neq \phi$ $\Rightarrow$ not LL(2)

Compiler Design

Compiler Design

Bottom-up parsing: From leaves to root, uses right most derivation in reverse order, uses reduction process

$G = \{E \to E+E \mid E*E \mid id\}$ LR(0)

Right Sentencial form	Handle	Production
id + id * id	id	E → id
E + id * id	id	E → id
E + E * id	id	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

$E \to E+E$

$E \to E*E$

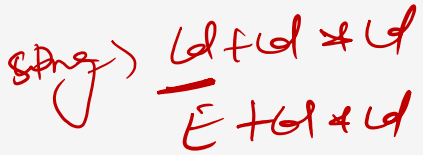
$E \to id$

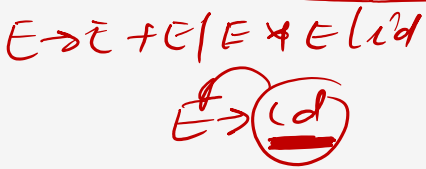
Compiler Design

Handle Pruning: Reducing the string to start symbol using the handles

Handle: A Handle is a substring of a string that matched with any of the right side of the productions then That handle will be reduced with left side of the production.

Bottom-up parsing is a process of finding the handles And using them in the reductions to get the start symbol, This entire process of reducing the string to the start symbol is called as handle pruning





Compiler Design

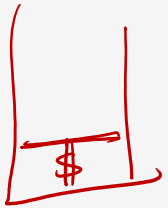
(Bottom up parsing)

Shift-Reduce Parsing: 4 Actions

- Shift
- reduce
- Accept
- Error

Two-data structures

1-STACK: $



2-Input Buffer : ω $

Compiler Design

Question:- Consider the following grammar and Parse the input string (aab) using Shift Reduce Parser

$G = \{ S \to AB, A \to aA \mid a, B \to bB \mid b \}$

Handle pruning

Viable prefixes

STACK	Input Buffer	Actions
$	aab$	Shift
$a	ab$	Shift
$aa	b$	Reduce by A → a
$aA	b$	Reduce by A → aA
$A	b$	Shift
$Ab	$	Reduce B → b
$AB	$	Reduce S → AB
$S	$	Accept

Viable prefixes.

Viable prefixes are the set of prefix of the Right Sentencial form that can appear on the STACK of a shift Reduce Parser.

Compiler Design

Method-2

Question: Consider the following grammar, which of the following are Viable Prefixes?

$G = \{ S \to AB, A \to aA \mid a, B \to bB \mid b \}$

1) aa ✓ 2) Aa ✗ 3) bb ✗ 4) bB ✗ 5) aB ✗ 6) AbB ✓ 7) ABb ✗ 8) AbbB ✓

a
A
Ab
AB

a
aa ✓
aA
A
Ab
Abb
Abbb
AbbB
AbB
AB

① aaaaA ✓

② Abbbbbbb B ✓

③ aaaaab ✗

④ aaaaabbbbb ✗

Compiler Design

$G = \{ S \to AB,\ A \to aA \mid a,\ B \to bB \mid b \}$

AB B

aaaaabbbb. ⇒ aabb ⇒ X

aaaaA ✓ Viable prefix

STACK	I/P Buff	Action
$	aabb$	
$a	abb$	Shift
$aa	bb$	Shift
$aab	b$	B → b
$aaB	b$	Shift
$aaBb	$	B → b
$aaBB	$	A → aA
$aAB	$	A → aA
$AB	$	A → a
$S	$	Accept

STACK	I/P Buff	Action
$	aaaaA$	
$a	aaaA$	Shift
$aa	aaA$	Shift
$aaa	aA$	Shift
$aaaa	A$	Shift
$aaaaA	$	A → a
$aaaA	$	A → aA
$aaA	$	A → aA
$aA	$	A → aA
$AB	$	B → b
$S	$	Accept

Compiler Design

Ex: $G = \{ S \to CC,\ C \to aC \mid b \}$ Viable prefixes

a) Cb

b) aC

c) aaC

d) Ca

Yes) aSb

STACK	I/P Buffer	Action
$	$	C → b
$ab	—	
$aC	$	
$	$	
$	$	

Compiler Design

Break 5 mins

Compiler Design

Compiler Design

Compiler Design

Compiler Design

Compiler Design

Compiler Design

Question:- $G = \{ S \to CC, C \to aC/b \}$ viable prefixes ?

a) Cb b) aC c) aaC d) Ca e) abb

Compiler Design

Compiler Design

Compiler Design

Compiler Design

Compiler Design

Operator Precedence Parsing:

- Applicable for operator Grammar
- It can Parse ambiguous Grammar also by using Precedence & Associativity rules
- It perfor 4 Actions
  - Shift
  - Reduce
  - Accept
  - Error
- Two data structure
  - STACK:
  - Input Buff

Operator Grammar:-

i) No ϵ-production
ii) No consecutive Non terminals

$A \to \epsilon$

$S \to AB$

$S \to ABC$

$E \to EAF \mid id$
$A \to + \mid *$

not operator Gramm

↳ Convert

$E \to E+F \mid E*F \mid id$
$A \to + \mid *$

Operator Gramma

Compiler Design

$a \doteq b$

$a \lessdot b$

$a \doteq b$

Compiler Design

Operator Precedence Parsing Table :- E <· id + $

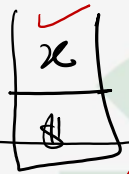
How to Parse String id+id*id

1) Let the topmost symbol in the Stack is x and the present symbol in the input buffer is y then Proceed as follows

1) if x <· y Then Shift y into the STACK

2) if x ·> y Then there will be a Handle in the STACK then Perform a reduce operation with an appropriate production

	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	



NOTE: Non terminal have least precedence but greater than $

Compiler Design

$E \to E+E$
$E \to E*E$
$E \to id$

$\omega = id + id * id$

STACK	Input Buffer	Action
$	id + id * id $	Shift
$ id	+ id * id $	Reduce E→id
$ E	+ id * id $	Shift
$ E +	id * id $	Shift
$ E + id	* id $	Reduce E→id
$ E + E	* id $	Shift
$ E + E *	id $	Shift
$ E + E * id	$	Reduce E→id
$ E + E * E	$	E→E*E
$ E + E	$	E→E+E
$ E	$	Accept

Space complexity = $O(n^2)$

	id	+	*	$
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

Size of table = $O(n^2)$

where n = no. of terminals

Q

Size = $(n+1) * (n+1)$

$\Rightarrow O(n^2)$

4

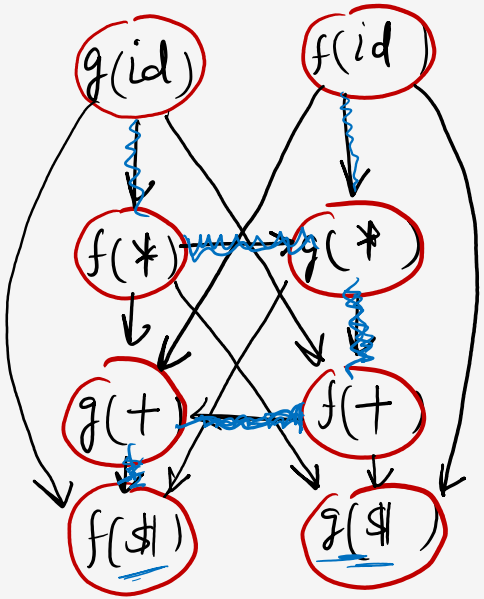
$\Rightarrow (4+1)(4+1) = 25$

Compiler Design

Operator function Table

If $f(x) > g(y)$

$f(x) \rightarrow g(y)$



Sizeof $O(n^2)$

	g: id	g: +	g: *	g: $
f: id	-	>	>	>
f: +	<	>	>	>
f: *	<	>	>	>
f: $	<	<	<	-

Operator function Table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Sizeof Table = $2 * (n)$

$= O(n)$

$f(+) > g(+)$

$f(+) \rightarrow g(+)$

$f(+) < g(id)$

$f(+) \leftarrow g(id)$

Compiler Design

Thank You !