Compilers (CS31003)

Syntax Analysis or Parsing

Lecture 6

A grammar

$G =< T, N, S, P >$ is a (context-free) grammar where:

T	:	Set of terminal symbols
N	:	Set of non-terminal symbols
S	:	$S \in N$ is the start symbol
P	:	Set of production rules

Every production rule is of the form: $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.

Symbol convention:

$a, b, c, \cdots$	Lower case letters at the beginning of alphabet	$\in T$
$x, y, z, \cdots$	Lower case letters at the end of alphabet	$\in T^+$
$A, B, C, \cdots$	Upper case letters at the beginning of alphabet	$\in N$
$X, Y, Z, \cdots$	Upper case letters at the end of alphabet	$\in (N \cup T)$
$\alpha, \beta, \gamma, \cdots$	Greek letters	$\in (N \cup T)^*$

A grammar

* $G_1=(\{E\}, \{+,*,(,),Id\}, P_1, E)$
* P_1 $\quad E \rightarrow E + E \mid E * E \mid (E) \mid Id$

Id+Id

Left most derivation:

$E \rightarrow E + E \rightarrow Id + E \rightarrow Id + Id$

Right most derivation

$E \rightarrow E + E \rightarrow E + Id \rightarrow Id + Id$

unambiguous grammar

A grammar

* G_1=({E}, {+,*,(,),Id}, P_1, E)
* P_1 E → E + E | E * E | (E) | Id

Id*Id+Id

ambiguous grammar

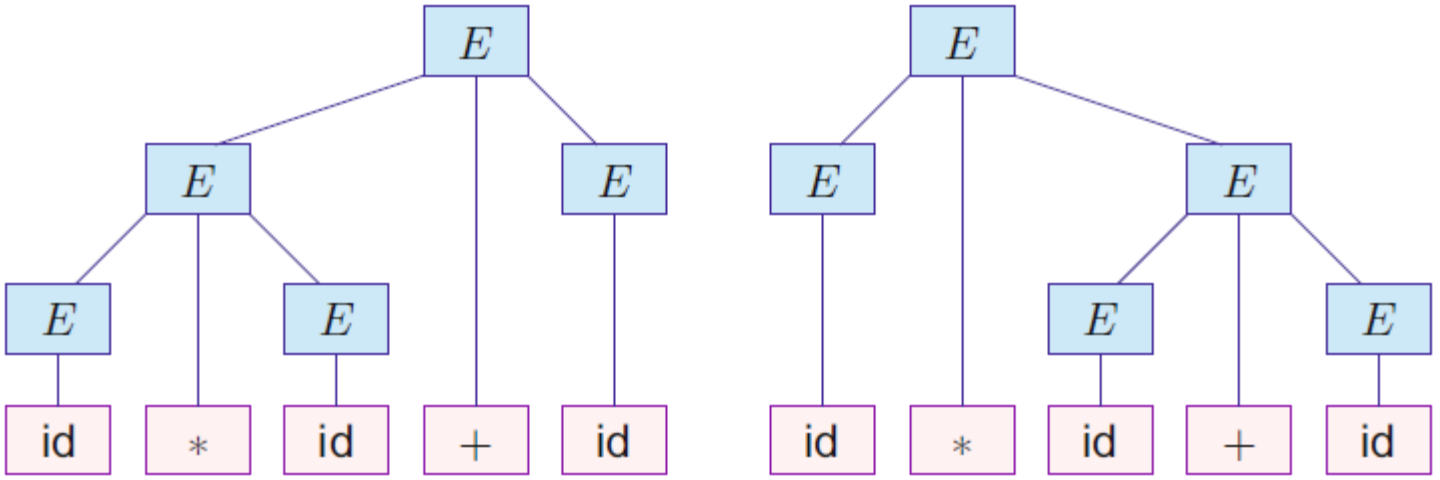
Left most derivation:
E ➔ E + E ➔ E * E + E ➔ Id * E + E ➔ Id * Id + E ➔ Id * Id + Id
E ➔ E * E ➔ Id * E ➔ Id * E + E ➔ Id * Id + E ➔ Id * Id + Id

Right most derivation
E ➔ E * E ➔ E * E + E ➔ E * E + Id ➔ E * Id + Id ➔ Id * Id + Id
E ➔ E + E ➔ E + Id ➔ E * E + Id ➔ E * Id + Id ➔ Id * Id + Id

A grammar

* G_1=({E}, {+,*,(,),Id}, P_1, E)
* P_1 E → E + E | E * E | (E) | Id

ambiguous grammar



Two grammars

- $G_0=(\{E,T,F\}, \{+,*,(,),Id\}, P_0, E)$
- P_0 $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow (E) \mid Id$

* $G_1=(\{E\}, \{+,*,(,),Id\}, P_1, E)$
* P_1 $E \rightarrow E + E \mid E * E \mid (E) \mid Id$

Parsing Fundamentals

Derivation	Parsing	Parser	Remarks
Left-most	Top-Down	Predictive: Recursive Descent, LL(1)	No Ambiguity No Left-recursion Tool: Antlr
Right-most	Bottom-Up	Shift-Reduce: SLR, LALR(1), LR(1)	Ambiguity okay Left-recursion okay Tool: YACC, Bison

Recursive Descent Parser

$S \rightarrow c\ A\ d$

$A \rightarrow ab \mid a$

```
int main() {
    l = getchar();
    S(); // S is a start symbol

    // Here l is lookahead. If l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
S() { // Definition of S, as per the given production
    match('c');
    A();
    match('d');
}
A() { // Definition of A as per the given production
    match('a');
    if (l == 'b') { // Look-ahead for decision
        match('b');
    }
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: cad$ ($S \Rightarrow cAd \Rightarrow cad$), cabd$ ($S \Rightarrow cAd \Rightarrow cabd$), caad$

Recursive Descent Parser

$S \rightarrow c\ A\ d$
$A \rightarrow aAb \mid a$

```
int main() {
    l = getchar();
    S(); // S is a start symbol.

    // Here l is lookahead. if l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
S() { // Definition of S, as per the given production
    match('c');
    A();
    match('d');
}
A() { // Definition of A as per the given production
    match('a');
    if (l == 'a') { // Look-ahead for decision
        A();
        match('b');
    }
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: cad$ ($S \Rightarrow cAd \Rightarrow cad$), cabd$, caabd$ ($S \Rightarrow cAd \Rightarrow caAbd \Rightarrow caabd$)

Recursive Descent Parser

$$E \rightarrow a\,E'$$
$$E' \rightarrow +\,a\,E' \mid \epsilon$$

```
int main() {
    l = getchar();
    E(); // E is a start symbol.
    // Here l is lookahead. If l = $, it represents the end of the string
    if (l == '$') printf("Parsing Successful");
    else printf("Error");
}
E() { // Definition of E, as per the given production
    match('a');
    E'();
}
E'() { // Definition of E' as per the given production
    if (l == '+') { // Look-ahead for decision
        match('+');
        match('a');
        E'();
    }
    else return (); // epsilon production
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: a$ ($E \Rightarrow aE' \Rightarrow a$), a+a$ ($E \Rightarrow aE' \Rightarrow a + aE' \Rightarrow a + a$), a+a+a$ ($E \Rightarrow aE' \Rightarrow a + aE' \Rightarrow a + a + aE' \Rightarrow a + a + a$)

Recursive Descent Parser

$E \rightarrow E + E \mid a$

```
int main() {
    l = getchar();
    E(); // E is a start symbol.

    // Here l is lookahead. if l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
E() { // Definition of E as per the given production
    if (l == 'a') { // Terminate ? -- Look-ahead does not work
        match('a');
    }

    E();            // Call ?
    match('+');
    E();
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: a+a$, a+a+a$

Top-Down parsing

- Action A: Selection of an alternative for the actual leftmost nonterminal and attachment of the right side of the production to the actual tree fragment.
- Action B: Comparison of terminal symbols to the left of the leftmost nonterminal with the remaining input.

$S \rightarrow E$ $\quad E' \rightarrow + E \mid \varepsilon$ $\quad T' \rightarrow * T \mid \varepsilon$

$E \rightarrow T\ E'$ $\quad T \rightarrow F\ T'$ $\quad F \rightarrow (E) \mid \mathrm{Id}$

Id+Id*Id

Top-Down parsing

$S \rightarrow E$ $\quad E' \rightarrow +\,E \mid \varepsilon$ $\quad T' \rightarrow *\,T \mid \varepsilon$

$E \rightarrow T\,E'$ $\quad T \rightarrow F\,T'$ $\quad F \rightarrow (E) \mid \mathrm{Id}$

Id+Id*Id

Curse or Boon 1: Left-Recursion

A grammar is left-recursive iff there exists a non-terminal A that can derive to a sentential form with itself as the leftmost symbol. Symbolically,

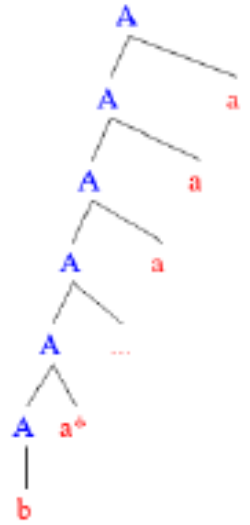
$$A \Rightarrow^{+} A\alpha$$

We cannot have a recursive descent or predictive parser (with left-recursion in the grammar) because we do not know how long should we recur without consuming an input

Curse or Boon 1: Left-Recursion

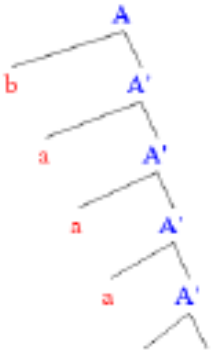
Note that, $A \rightarrow A\alpha$, $A \rightarrow \beta$ leads to:

$$A\$ \Rightarrow A\alpha\$ \Rightarrow A\alpha\alpha\$ \Rightarrow A\alpha\alpha\alpha\$ \cdots$$
$$\Rightarrow A\alpha^*\$ \Rightarrow \beta\alpha^*\$$$



Removing left-recursion $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$ leads to:

$$A\$ \Rightarrow \beta A'\$ \Rightarrow \beta\alpha A'\$ \Rightarrow \beta\alpha\alpha A'\$ \cdots$$
$$\Rightarrow \beta\alpha^* A'\$ \Rightarrow \beta\alpha^*\$$$



Left-Recursion: Example

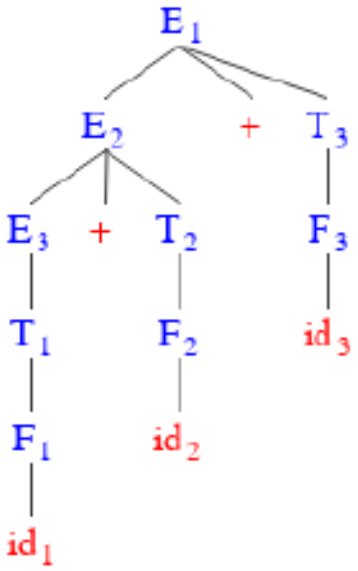
Grammar G_1 before Left-Recursion Removal

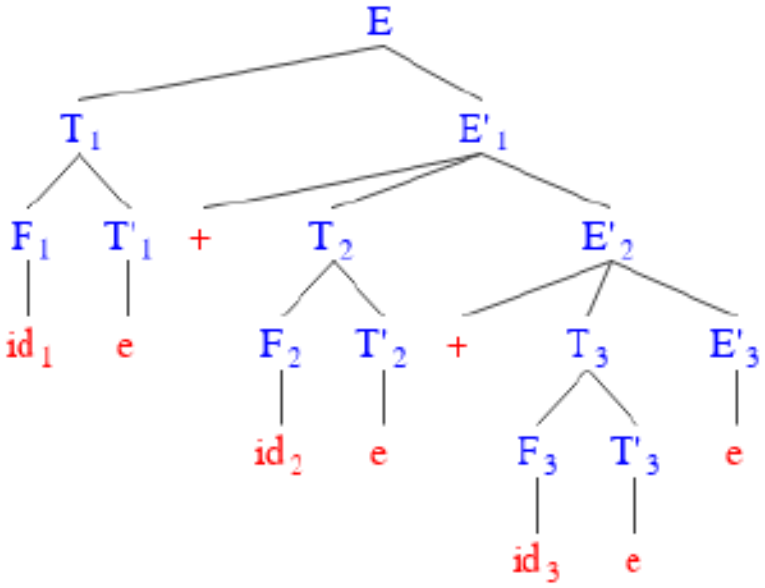
1:	E	$\rightarrow$	$E + T$
2:	E	$\rightarrow$	T
3:	T	$\rightarrow$	$T * F$
4:	T	$\rightarrow$	F
5:	F	$\rightarrow$	(E)
6:	F	$\rightarrow$	**id**

Grammar G_2 after Left-Recursion Removal

1:	E	$\rightarrow$	$T\ E'$		
2:	E'	$\rightarrow$	$+\ T\ E'$	|	ϵ
3:	T	$\rightarrow$	$F\ T'$		
4:	T'	$\rightarrow$	$*\ F\ T'$	|	ϵ
5:	F	$\rightarrow$	(E)		
6:	F	$\rightarrow$	**id**		

- These are syntactically equivalent. But what happens semantically?
- Can left recursion be effectively removed?
- What happens to Associativity?





Curse or Boon 2: Left-Recursion

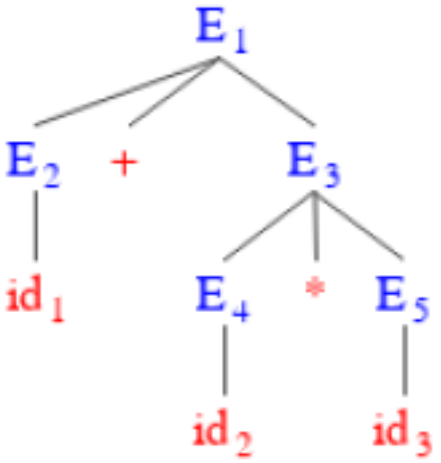
1: $E \rightarrow E + E$
2: $E \rightarrow E * E$
3: $E \rightarrow (E)$
4: $E \rightarrow \mathbf{id}$

- Ambiguity simplifies. But, ...
  - Associativity is lost
  - Precedence is lost
- Can *Operator Precedence (infix $\rightarrow$ postfix)* give us a clue?

Ambiguous Derivation of id + id * id

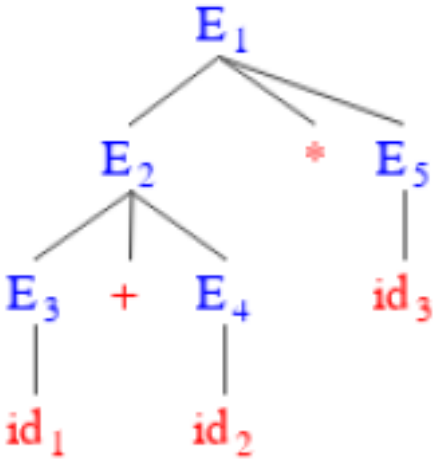
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{E * E}\ \$ \\
&\Rightarrow E + E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

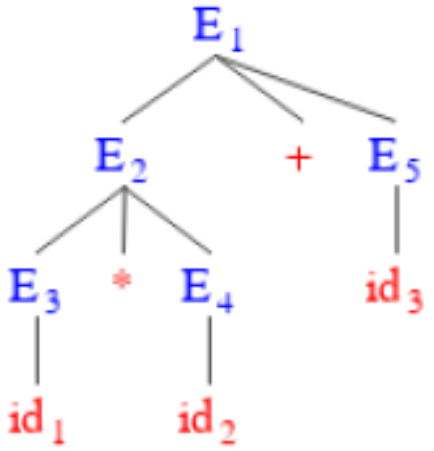
$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{E + E} * \mathbf{id}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Ambiguous Derivation of id * id + id

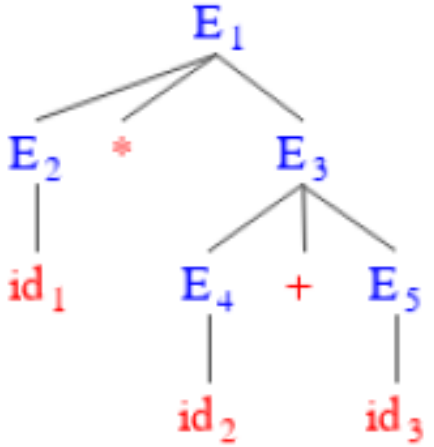
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{E * E} + \mathbf{id}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} * \mathbf{id} + \mathbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{E + E}\ \$ \\
&\Rightarrow E * E + \underline{\mathbf{id}}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} * \mathbf{id} + \mathbf{id}\ \$
\end{aligned}
$$



Remove: Ambiguity and Left-Recursion

1: $E \rightarrow E + E$
2: $E \rightarrow E * E$
3: $E \rightarrow (E)$
4: $E \rightarrow \mathbf{id}$

Removing ambiguity:

1: $E \rightarrow E + T$
2: $E \rightarrow T$
3: $T \rightarrow T * F$
4: $T \rightarrow F$
5: $F \rightarrow (E)$
6: $F \rightarrow \mathbf{id}$

Removing left-recursion:

1: $E \rightarrow T\ E'$
2|3: $E' \rightarrow +\ T\ E' \mid \epsilon$
4: $T \rightarrow F\ T'$
5|6: $T' \rightarrow *\ F\ T' \mid \epsilon$
7: $F \rightarrow (E)$
8: $F \rightarrow \mathbf{id}$