Lexical and Syntax Analysis

Part II

The LL Grammar Class

- Recursive-descent and other LL parsers can be used **only** with grammars that **meet certain restrictions**.
- Left recursion causes a catastrophic problem for LL parsers. Calling the recursive-descent parsing subprogram for the following rule would cause infinite recursion:

```
A : A + B
```

- The left recursion in the rule `A : A + B` is called **direct left recursion**, because it occurs in one rule.
- **Indirect left recursion** poses the same problem as direct left recursion:

```
A : B a A
B : A b
```

Functions for Left Recursive Rules (infinite recursion!)

```
# A : A + B
def A():
  global next_token
  global l
  A()
  if next_token.get_token().value
     == TokenTypes.PLUS.value:
    next_token = l.lex()
    B()
```

Direct Left Recursion

```
# A : B a A
def A():
  global next_token
  global l
  B()
  if next_token.get_token().value
     == TokenTypes.ATOKEN.value:
    next_token = l.lex()
    A()
  else:
    error(“Expecting ATOKEN”)
```

```
# B : A b
def B():
  global next_token
  global l
  A()
  if next_token.get_token().value
     == TokenTypes.BTOKEN.value:
    next_token = l.lex()
  else:
    error(“Expecting BTOKEN”)
```

Indirect Left Recursion

Algorithm to Eliminate Direct Left Recursion

For each non-terminal A,

(1) Group the A-rules as

A : Aα_1 | … | Aα_m| β_1| … | β_n

where none of the β's begins with A.

(2) Replace the original A-rules with

A : β_1 A′ | … | β_n A′

A′ : α_1A′ | … | α_mA′ | ε

The symbol ε represents the empty string. A rule that has ε as its RHS is called an **erasure rule**, because using it in a derivation effectively erases its LHS from the sentential form.

Eliminate Direct Left Recursion: Example

Consider the following left-recursive grammar:

```
E : E + T | T
T : T * F | F
F : ( E ) | id
```

For the E-rules, $\alpha_1 = + T$ and $\beta_1 = T$

So, the new E-rules will be:

```
E  : T E′
E′ : + T E′| ε
```

For the T-rules, $\alpha_1 = * F$ and $\beta_1 = F$

So, the new T-rules will be:

```
T  : F T′
T′ : * F T′| ε
```

The F-rules do not change since there is not left recursion.

The final transformed grammar with no left-recursion:

```
E  : T E′
E′ : + T E′| ε
T  : F T′
T′ : * F T′| ε
F  : ( E ) | id
```

There exists algorithms to eliminate indirect left-recursion, but we don't cover it in this class.

Pairwise Disjointness Test

- Left-recursion is not the only trait that makes top-down parsing problematic.
- A top-down parser also needs to **choose the correct RHS** based on the next token, especially when the non-terminal being expanded has multiple rules.
- The **pairwise disjointness test** is used to test a non-left-recursive grammar to determine whether it can be parsed in a top-down fashion. This test requires computing **FIRST** sets, where

  ```
  FIRST(α) = {a | α =>* aβ}
  ```

  The symbol =>* indicates a derivation of zero or more steps. If α =>* ε, then ε is also in FIRST(α), where ε is the empty string.

- There are algorithms to compute the FIRST sets for any mixed string. In simple cases, FIRST can be computed by simply examining the grammar.

Pairwise Disjointness Test

The pairwise disjoint test: For each nonterminal `A` that has more than one RHS, and for each pair of rules, `A :` α_i and `A :` α_j, it must be true that FIRST(α_i) ∩ FIRST(α_j) = ∅.

Example 1: Consider the following grammar

A : a B
A : b A b
A : B b
B : c B
B : d

The FIRST sets for the RHS of A-rules are: FIRST(aB) = { a }, FIRST(bAb) = { b }, and FIRST(Bb) = { c, d }. These are disjoint and hence PASS the pairwise disjoint test.

The FIRST sets for the RHS of B-rules are: FIRST(cB) = { c } and FIRST(d) = { d }. These are disjoint and hence PASS the pairwise disjoint test.

So, the grammar as a whole passes the pairwise disjoint test and hence can be parsed using top-down parsers!

Pairwise Disjointness Test Second Example

Example 2: Consider the following grammar

```
A : a B
A : B A b
B : a B
B : b
```

The FIRST sets for the RHS of A-rules are: FIRST(aB) = { a } and FIRST(BAb) = { a, b }. These are not disjoint and hence FAIL the pairwise disjoint test.

The FIRST sets for the RHS of B-rules are: FIRST(aB) = { a } and FIRST(b) = { b }. These are disjoint and hence PASS the pairwise disjoint test.

So, the grammar as a whole fails the pairwise disjoint test and hence cannot be parsed using top-down parsers!

Left Factoring

- In many cases, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test
- The following rules do not pass the pairwise disjointness test:

```
<var> : ID | ID [ <expr> ]
```

Using the technique of "**left-factoring**", we can rewrite the rules as follows:

```
<var> : ID <new>
<new> : ε
<new> : [ <expr> ]
```

The modified grammar passes the pairwise disjointness test!

- Algorithms do exists for left-factoring, but we do not cover them in this class.
- Left-factoring cannot solve all pairwise-disjointness problems.

Bottom-Up Parsers

- The following grammar for arithmetic expressions will be used to illustrate bottom-up parsing:

```
E : E + T | T
T : T * F | F
F : ( E ) | id
```

  This grammar is left-recursive, which is acceptable to bottom-up parsers.

- A rightmost derivation for `id + id * id` is shown below (red part highlights the RHS of rules applied at each step):

```
E => E + T
  => E + T * F
  => E + T * id
  => E + F * id
  => E + id * id
  => T + id * id
  => F + id * id
  => id + id * id
```

A bottom-up parser produces the **reverse** of a **rightmost derivation** by starting with the **last** sentential form (the input sentence) and **working back** to the **start** symbol.

At each step, the parser's task is to **find the RHS** in the current sentential form that must be rewritten to get the previous sentential form.

Bottom-Up Parsers (continued)

- A right sentential form may include more than one RHS. For example, `E + T * id` contains three RHSs: E + T, T, and id.
- The task of the bottom-up parser is to find the **unique handle** of a given right sentential form.
- **Definition**: β is the **handle** of the right sentential form $\gamma = \alpha\beta w$ **if and only if**

  $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$.

  where $\Rightarrow_{rm}$ specifies a rightmost derivation step, and $\Rightarrow^*_{rm}$ specifies zero or more rightmost derivation steps.

```
E : E + T | T
T : T * F | F
F : ( E ) | id
```

Phrases

Two other concepts are related to the idea of a handle.

- **Definition:** β is a **phrase** of the right sentential form $\gamma = \alpha_1\beta\alpha_2$ <u>if and only if</u>

  $S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$

  where $\Rightarrow^+$ means one or more derivation steps.

- **Definition**: β is a **simple phrase** of the right sentential form $\gamma = \alpha_1\beta\alpha_2$ <u>if and only if</u>

  $S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$

Phrases (continued)

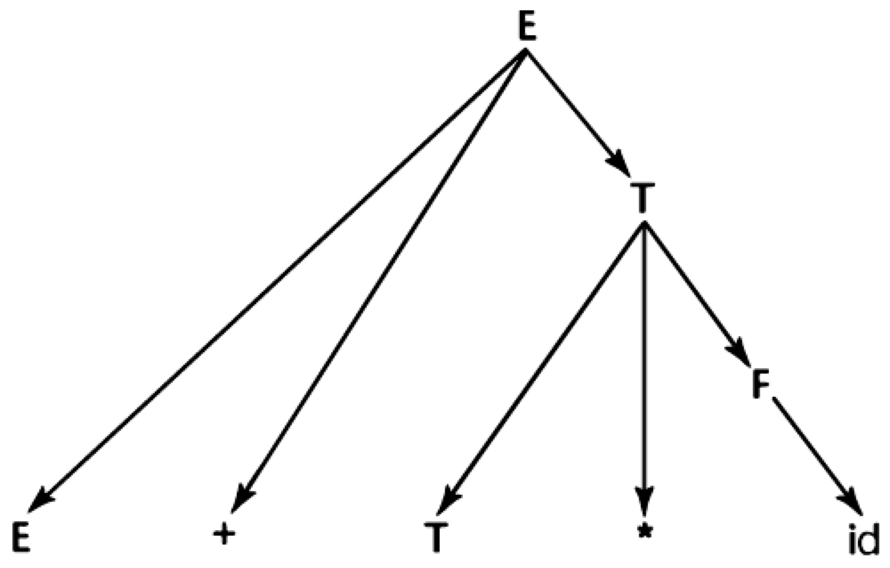
```
E : E + T | T
T : T * F | F
F : ( E ) | id
```

A *phrase* is a string consisting of **all of the leaves** of the **partial parse tree** that is **rooted** at one particular internal node of the whole parse tree.

A *simple phrase* is a phrase that is derived from a nonterminal in **a single step**.

Consider the partial parse tree shown to the right.
There are 3 internal nodes: `E`, `T`, and `F`
The three phrases corresponding these 3 internal nodes are:
`E + T * id`, `T * id`, and `id`
The only simple phrase is `id`

The handle of a right sentential form is the leftmost simple phrase.



Phrases (another example)

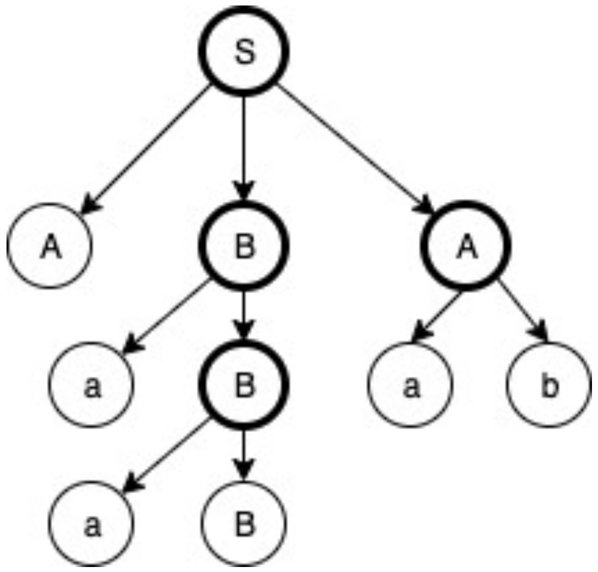
Grammar

```
S : aAb
S : ABA
A : ab
A : BAB
B : aB
B : b
```

Partial rightmost derivation for `AaaBab`

```
S => ABA
  => ABab
  => AaBab
  => AaaBab
```

Partial Parse Tree for `AaaBab`



The internal nodes of the partial parse tree give the following phrases:

```
AaaBab
aaB
```
`aB : simple phrase; handle!`
```
ab : simple phrase
```

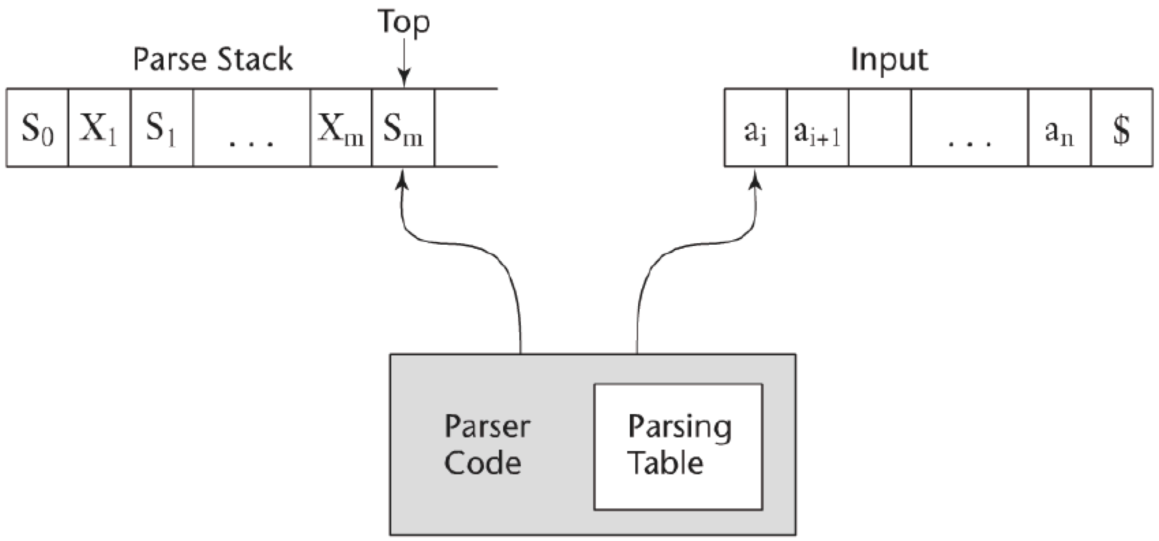
Shift-Reduce Algorithms

- Bottom-up parsers are often called **shift-reduce** algorithms, because **shift** and **reduce** are their two fundamental actions.
- The **shift** action moves the next input token onto the parser's stack. A **reduce** action replaces a RHS (the handle) on top of the parser's stack by its corresponding LHS.
- Most bottom-up parsing algorithms belong to the **LR family**. LR parsers use a relatively small amount of code and a parsing table.
- The original LR algorithm was designed by **Donald Knuth**, who published it in 1965.
- **Advantages of LR parsers**: (1) They can be built for all programming languages. (2) They can detect syntax errors as soon as possible in a left-to-right scan. (2) The LR class of grammars is a proper **superset** of the class parsable by LL parsers.
- It is difficult to produce an LR parsing table **by hand.** However, there are many programs (e.g. **PLY**) available that take a grammar as input and produce the parsing table.

LR Parsers (Structure)

LR-parsers employ a **stack** to store parsing status at any point in time and use a **parsing table** that informs the next action (shift or reduce).

In general, **each grammar symbol** on the stack will be **followed** by a **state symbol** (often written as a subscripted uppercase S).



LR Parsers (Continued)

- The contents of the parse stack for an LR parser has the following form, where the Ss are **state symbols** and the Xs are **grammar symbols**:

  $S_0X_1S_1X_2S_2\ldots X_mS_m$ `(top)`

- An LR parser **configuration** is a pair of strings representing the stack and the **remaining** input:

  $(S_0X_1S_1X_2S_2\ldots X_mS_m,\ a_ia_{i+1}\ldots a_n\$)$

  The dollar sign is an end-of-input marker.

- The LR parsing process is based on the parsing table, which has two parts, **ACTION** and **GOTO**.
- The ACTION part has **state symbols as its row labels** and **terminal symbols as its column labels**.
- The parse table specifies what the parser should do, based on the **state symbol on top of the parse stack** and the **next input symbol**.

LR Parser Actions (Shift/Reduce/Accept/Error)

The two primary actions are **shift** (shift the next input symbol onto the stack) and **reduce** (replace the handle on top of the stack by the LHS of the matching rule).

Two other actions are possible: **accept** (parsing is complete) and **error** (a syntax error has been detected).

The values in the **GOTO part of the table** indicate which state symbol should be pushed onto the parse stack after a **reduction** has been completed. The row is determined by the state symbol on top of the parse stack **after** the handle and its associated state symbols have been removed. The column is determined by the **LHS of the rule** used in the reduction.

LR Parser: Parsing Algorithm

Initial state of the LR Parser:

`(S0, a₁…aₙ$)`

Informal definitions of parser actions:

Shift: The next input symbol is pushed onto the stack, along with the state symbol specified in the ACTION table.

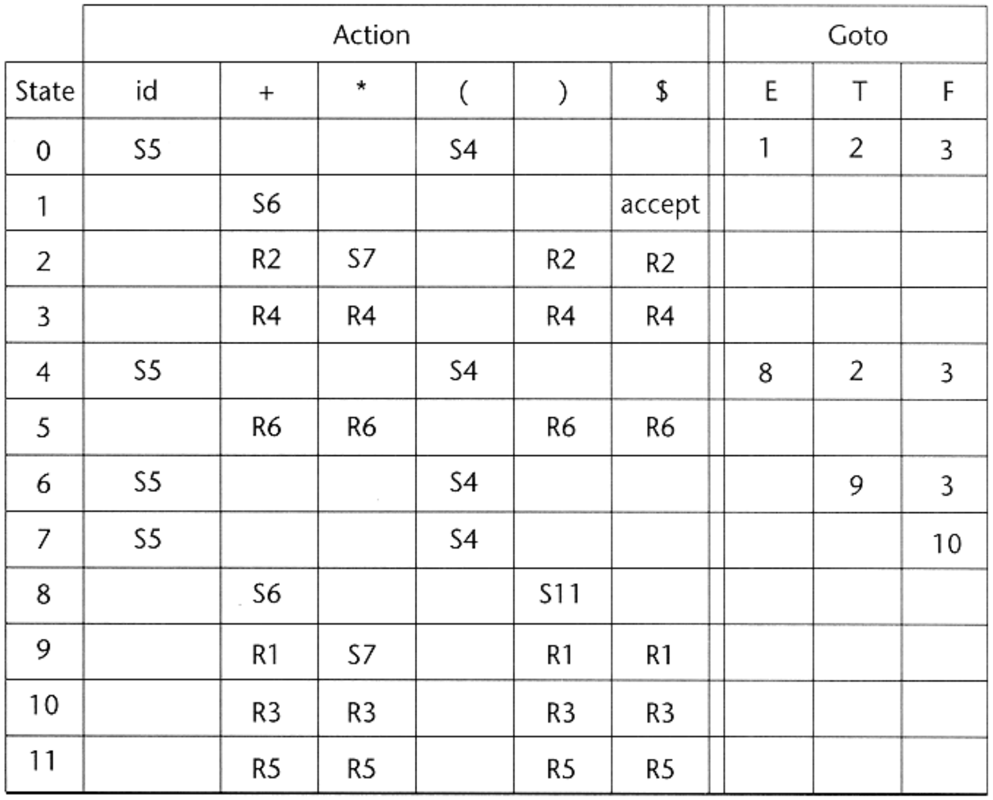
Reduce: First, the handle is removed from the stack. For every grammar symbol on the stack there is a state symbol, so the number of symbols removed is twice the number of symbols in the handle. Next, the LHS of the rule is pushed onto the stack. Finally, the GOTO table is used to determine which state must be pushed onto the stack.

Accept: The parse is complete and no errors were found.

Error: The parser calls an error-handling routine.

LR Parser: Parsing Algorithm in Action

```
1. E : E + T
2. E : T
3. T : T * F
4. T : F
5. F : ( E )
6. F : id
```



	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

R4 means reduce using rule 4; S6 means shift the next input symbol onto the stack and push state 6. Empty positions in the ACTION table indicate syntax errors.

Trace for input: `id + id * id`

```
Stack         Input          Action
0             id + id * id $ Shift 5
0id5          + id * id $    Reduce 6 (use GOTO[0, F])
0F3           + id * id $    Reduce 4 (use GOTO[0, T])
0T2           + id * id $    Reduce 2 (use GOTO[0, E])
0E1           + id * id $    Shift 6
0E1+6         id * id$       Shift 5
0E1+6id5      * id $         Reduce 6 (use GOTO[6, F])
0E1+6F3       * id $         Reduce 4 (use GOTO[6, T])
0E1+6T9       * id $         Shift 7
0E1+6T9*7     id $           Shift 5
0E1+6T9*7id5  $              Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10  $              Reduce 3 (use GOTO[6, T])
0E1+6T9       $              Reduce 1 (use GOTO[0, E])
0E1           $              Accept
```

Obtaining the Rightmost Derivation

```
1. E : E + T
2. E : T
3. T : T * F
4. T : F
5. F : ( E )
6. F : id
```

Reduce actions

```
Reduce 6 (use GOTO[0, F])
Reduce 4 (use GOTO[0, T])
Reduce 2 (use GOTO[0, E])
Reduce 6 (use GOTO[6, F])
Reduce 4 (use GOTO[6, T])
Reduce 6 (use GOTO[7, F])
Reduce 3 (use GOTO[6, T])
Reduce 1 (use GOTO[0, E])
```

Reverse Reduce actions

```
Reduce 1 (use GOTO[0, E]) E : E + T
Reduce 3 (use GOTO[6, T]) T : T * F
Reduce 6 (use GOTO[7, F]) F : id
Reduce 4 (use GOTO[6, T]) T : F
Reduce 6 (use GOTO[6, F]) F : id
Reduce 2 (use GOTO[0, E]) E : T
Reduce 4 (use GOTO[0, T]) T : F
Reduce 6 (use GOTO[0, F]) F : Id
```

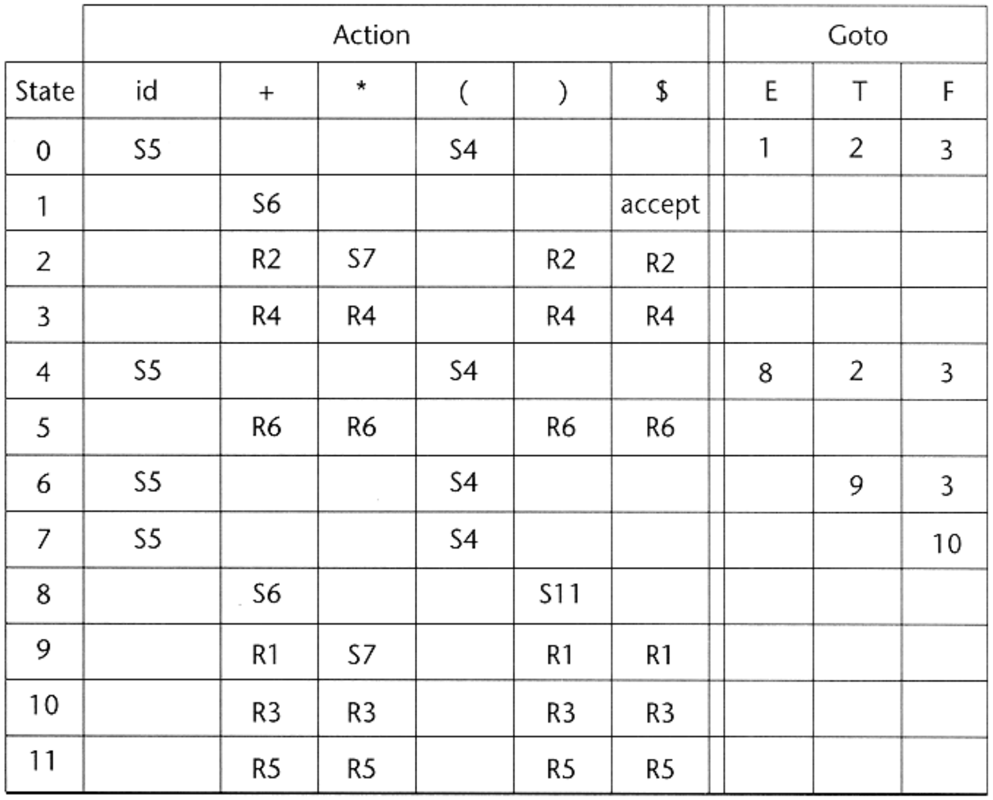
Rightmost Derivation

```
E
=> E + T
=> E + T * F
=> E + T * id
=> E + F * id
=> E + id * id
=> T + id * id
=> F + id * id
=> id + id * id
```

Parse Tree may be obtained easily from rightmost derivation

LR Parser: Parsing Algorithm in Action: Another Example

```
1. E : E + T
2. E : T
3. T : T * F
4. T : F
5. F : ( E )
6. F : id
```



	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

R4 means reduce using rule 4; S6 means shift the next input symbol onto the stack and push state 6. Empty positions in the ACTION table indicate syntax errors.

Trace for input: `id + (* id id)`

```
Stack        Input          Action
0            id + (* id id)$ Shift 5
0id5         + (* id id)$   Reduce 6 (use GOTO[0, F])
0F3          + (* id id)$   Reduce 4 (use GOTO[0, T])
0T2          + (* id id)$   Reduce 2 (use GOTO[0, E])
0E1          + (* id id)$   Shift 6
0E1+6        (* id id)$     Shift 4
0E1+6(4      * id id)$      ERROR! Action(4,*)=blank
```