Chap 3. Parsing

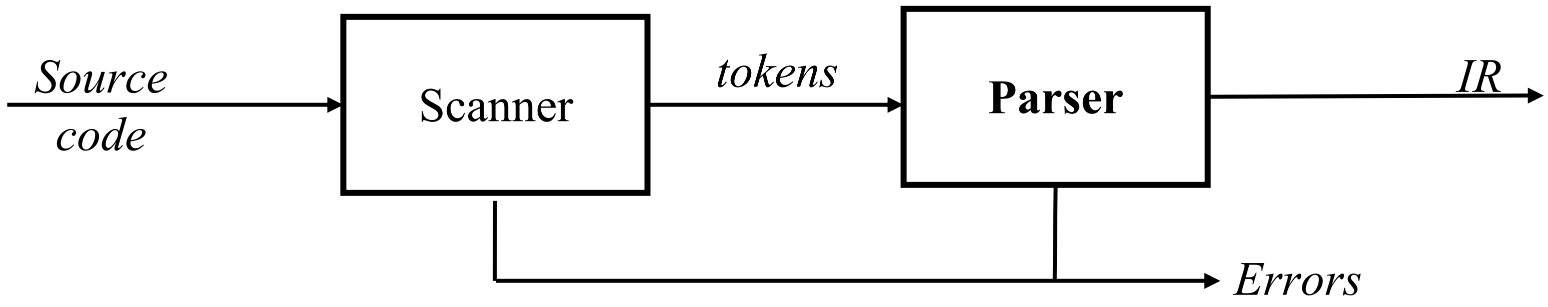
COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

3.1 Introduction

The Front End



Parser

- Checks the stream of **words** and their parts of speech for grammatical correctness
- Determines if the input is **syntactically well formed**
- Guides checking at deeper levels than syntax
- Builds an **IR representation of the code**

The Study of Parsing

- We need:
  - a grammar : **CFG** (context-free grammar)
    - $\rightarrow$ a language $L(G)$
    - keep in mind that **our goal is building parsers**, not studying the mathematics of arbitrary languages
  - parsing methods
    - **top-down parsing**
      - hand-coded recursive descent parsers
    - **bottom-up parsing**
      - generated LR(1) parsers

3.2 Expressing Syntax

Specifying Syntax with a Grammar

- **parser**
  - **an engine** that determines whether or not the input program is **a syntactically valid sentence** in the programming language.
- **context-free grammar**
  - how to syntactically form the program
- example CFG for Sheep Noise in **Backus-Naur Form**

  $$\begin{aligned} \textit{SheepNoise} &\rightarrow \underline{\text{baa}}\ \textit{SheepNoise} \\ &\mid\ \underline{\text{baa}} \end{aligned}$$

  - derivations:
    - $\textit{SheepNoise} \rightarrow \underline{\text{baa}}\ \textit{SheepNoise} \rightarrow \underline{\text{baa}}\ \underline{\text{baa}}$

Context-Free Grammar

a grammar is a four tuple, ***G* = (*T*, *NT*, *s*, *P*)**

- *T* is a set of *terminal symbols* *(words)*
  - Lexical Analyzer (scanner) returns a terminal.
- *NT* is a set of *non-terminal symbols* *(syntactic variables)*
- *s* is the *start symbol* *(set of strings in L(G))*
- *P* is a set of *productions* or *rewrite rules*
  - $P: NT \rightarrow (NT \cup T)^{+}$

BNF (Backus-Naur Form)

- the traditional notation used to represent CFG.
- original form: <SheepNoise> ::= baa <SheepNoise>
  | baa

Deriving Syntax

We can use the *SheepNoise* grammar to create sentences

- use the productions as *rewriting rules*
- 1: *SheepNoise* → baa *SheepNoise*
  2: | baa

Rule	Sentential Form
—	*SheepNoise*
2	baa

Rule	Sentential Form
—	*SheepNoise*
1	baa *SheepNoise*
1	baa baa *SheepNoise*
2	baa baa baa

Rule	Sentential Form
—	*SheepNoise*
1	baa *SheepNoise*
2	baa baa

And so on ...

A More Useful Grammar

- To explore the uses of CFGs, we need a more complex grammar

1	*Expr*	$\rightarrow$	*Expr Op Expr*
2		|	<u>num</u>
3		|	<u>id</u>
4	*Op*	$\rightarrow$	$\underline{+}$
5		|	$\underline{-}$
6		|	$\underline{*}$
7		|	$\underline{/}$

- $G = (T, NT, s, P)$
  - $T = \{\ \underline{\text{num}}, \underline{\text{id}}, \underline{+}, \underline{-}, \underline{*}, \underline{/}\ \}$, $NT = \{\ Expr, Op\ \}$, $s = Expr$

A More Useful Grammar

To explore the uses of CFGs, we need a more complex grammar

1	*Expr*	→	*Expr Op Expr*
2		|	num
3		|	id
4	*Op*	→	+
5		|	–
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
2	<id,x> *Op Expr*
5	<id,x> – *Expr*
1	<id,x> – *Expr Op Expr*
2	<id,x> – <num,2> *Op Expr*
6	<id,x> – <num,2> * *Expr*
3	<id,x> – <num,2> * <id,y>

We denote this derivation: *Expr* $\rightarrow^{*}$ id – num * id

- Such a sequence of rewrites is called a ***derivation***
- Process of discovering a derivation is called ***parsing***

Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- **Leftmost derivation** — replace leftmost NT at each step
- **Rightmost derivation** — replace rightmost NT at each step

These are the two *systematic* derivations

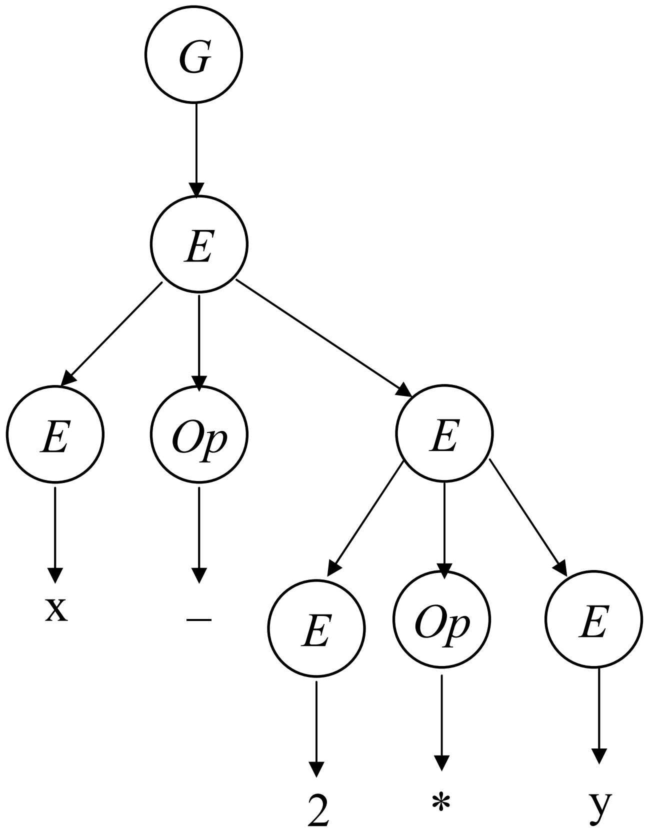
- We don’t care about randomly-ordered derivations!

Leftmost Derivation

1	*Expr*	→	*Expr Op Expr*
2		|	num
3		|	id
4	*Op*	→	+
5		|	–
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	<id,x> *Op Expr*
5	<id,x> – *Expr*
1	<id,x> – *Expr Op Expr*
2	<id,x> – <num,2> *Op Expr*
6	<id,x> – <num,2> * *Expr*
3	<id,x> – <num,2> * <id,y>

This evaluates as x – (2 * y)

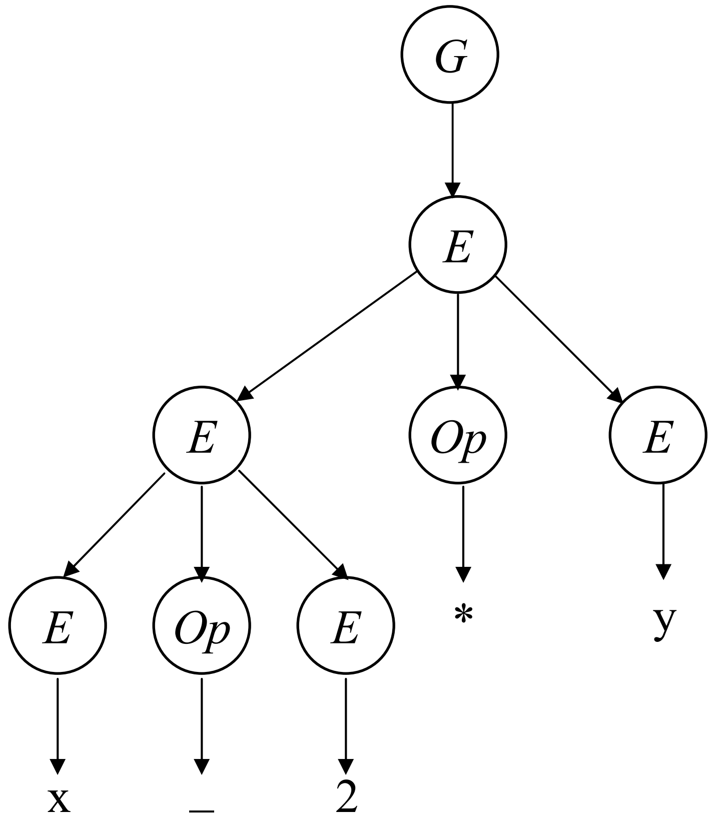


Rightmost Derivation

1	*Expr*	→	*Expr Op Expr*
2		|	num
3		|	id
4	*Op*	→	+
5		|	–
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* <id,y>
6	*Expr* * <id,y>
1	*Expr Op Expr* * <id,y>
2	*Expr Op* <num,2> * <id,y>
5	*Expr* – <num,2> * <id,y>
3	<id,x> – <num,2> * <id,y>

This evaluates as (x – 2) * y



Derivations and Precedence

- derivation에 따라, operator precedence가 다르게 나옴
  - compiler 입장에서는 심각한 문제
  - 해결책? grammar 자체의 수정
    - 새로운 non-terminal 도입

	1	*Goal*	→	*Expr*
level two	2	*Expr*	→	*Expr* + *Term*
	3		|	*Expr* – *Term*
	4		|	*Term*
level one	5	*Term*	→	*Term* * *Factor*
	6		|	*Term* / *Factor*
	7		|	*Factor*
	8	*Factor*	→	number
	9		|	id

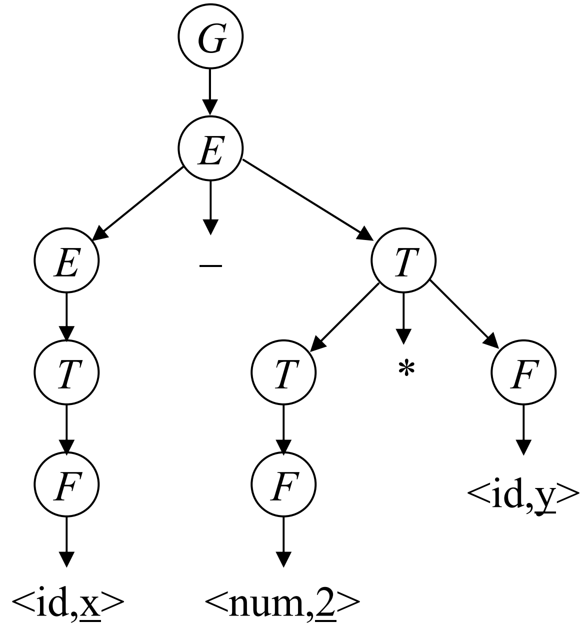
1	*Expr*	→	*Expr Op Expr*
2		|	num
3		|	id
4	*Op*	→	+
5		|	–
6		|	*
7		|	/

Derivations and Precedence

- 수정된 grammar에서는 leftmost derivation, rightmost derivation 모두 같은 parse tree를 생성
  - 모두 x – (2 * y) 생성
  - 왜? **grammar 자체에서 다른 derivation을 금지**

Rule	*Sentential Form*
—	*Goal*
1	*Expr*
3	*Expr* – *Term*
5	*Expr* – *Term* * *Factor*
9	*Expr* – *Term* * <id,y>
7	*Expr* – *Factor* * <id,y>
8	*Expr* – <num,2> * <id,y>
4	*Term* – <num,2> * <id,y>
7	*Factor* – <num,2> * <id,y>
9	<id,x> – <num,2> * <id,y>

The rightmost derivation



Its parse tree

Ambiguous Grammar

1	*Expr*	→	*Expr Op Expr*
2		|	num
3		|	id
4	*Op*	→	+
5		|	–
6		|	*
7		|	/

- original grammar에는 또다른 문제점 !
  - leftmost derivation만 적용해도 다른 parse tree 가능

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	<id,x> *Op Expr*
5	<id,x> – *Expr*
1	<id,x> – *Expr Op Expr*
2	<id,x> – <num,2> *Op Expr*
6	<id,x> – <num,2> * *Expr*
3	<id,x> – <num,2> * <id,y>

Original choice

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
1	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> – *Expr Op Expr*
2	<id,x> – <num,2> *Op Expr*
6	<id,x> – <num,2> * *Expr*
3	<id,x> – <num,2> * <id,y>

New choice

Ambiguous Grammars

Definitions

- If a grammar has **more than one leftmost derivation** for a single *sentential form*, the grammar is ***ambiguous***
- If a grammar has **more than one rightmost derivation** for a single sentential form, the grammar is ***ambiguous***

- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Ambiguous grammar는 **automatic parsing** 불가능

Ambiguous Grammar: If-Then-Else

- $Stmt \rightarrow$ <u>if</u> $Expr$ <u>then</u> $Stmt$
  | <u>if</u> $Expr$ <u>then</u> $Stmt$ <u>else</u> $Stmt$
  | $AssignStmt$

- input: <u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$
- output1: <u>if</u> $Expr_1$ <u>then</u> **{** <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$ **}**
- output2: <u>if</u> $Expr_1$ <u>then</u> **{** <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ **}** <u>else</u> $Stmt_2$

- 해결책
  - in Shell, Perl, etc.: introduce **elif**, **endif**
  - in most programming languages:
    - use **inner-most unmatched-if rule**

Ambiguous Grammar: If-Then-Else

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- **Match each <u>else</u> to innermost unmatched <u>if</u>**
- With this grammar, the example has only one derivation

- *Stmt* → <u>if</u> *Expr* <u>then</u> *Stmt*
  | <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt*
  | *AssignStmt*

- *WithElse* → <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *WithElse*
  | *AssignStmt*

An Example: If-Then-Else

- input: <u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$

1	*Stmt*	→	<u>if</u> *Expr* <u>then</u> *Stmt*
2		|	<u>if</u> *Expr* <u>then</u> *WithElse* else *Stmt*
3		|	<u>*AssignStmt*</u>
4	*WithElse*	→	<u>if</u> *Expr* <u>then</u> *WithElse* else *WithElse*
5		|	*AssignStmt*

Rule	*Sentential Form*
—	*Stmt*
1	<u>if</u> *Expr* <u>then</u> *Stmt*
2	<u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt*
3	<u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *AssignStmt*
5	<u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *AssignStmt* <u>else</u> *AssignStmt*

CFG and RE

- Regular Expression : Regular Grammar 와 equivalent
- **RG** (regular grammar) : **CFG with only left-linear rules**.
  - All rules are: $\mathbf{A \rightarrow a}$ or $\mathbf{A \rightarrow a\ B}$
    - A, B ∈ NT, a ∈ T
- 즉, CFG 만으로 RE 까지 표현 가능
  - 그러나, 여전히 **RE**를 쓴다
    - DFA-based scanner가 more efficient
    - comment 처리 ? CFG로는 처리 곤란
- CFG의 계층 구조
  - $\mathbf{RG \subset LL(1) \subset LR(1) \subset CFG}$

3.3 Top-Down Parsing

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

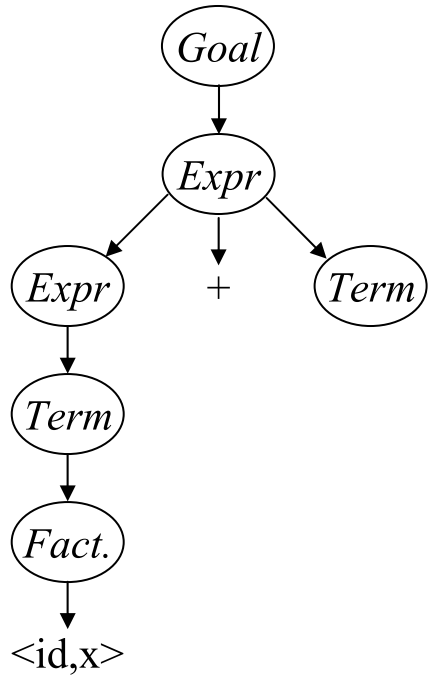
Top-down Parsing Algorithm

- Construct the **root** node of the parse tree
- Repeat until the leaf nodes of the parse tree matches the input string
  - At a node labeled *A*,
    select a production with *A* on its LHS and,
    for each symbol on its RHS,
    construct the appropriate child
  - When a terminal symbol is added to the leaf node and it doesn't match the leaf node, **backtrack**
  - Find the next node to be expanded
- 어떤 **production rule**을 고르느냐 **?**

An Example

- input: x – 2 * y
  - ↑ : current scanning pointer

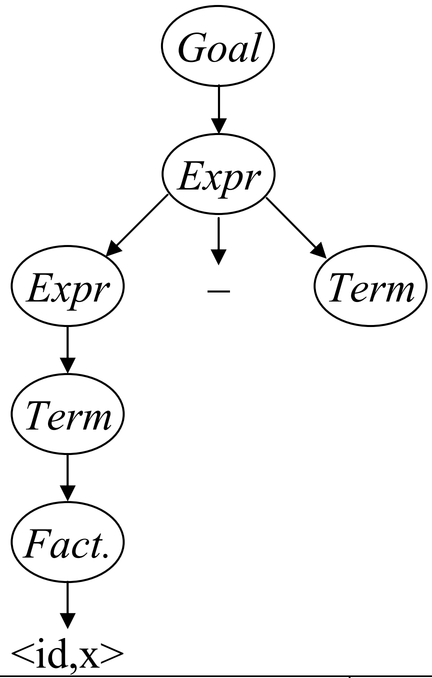
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	<u>number</u>
9		|	<u>id</u>



Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
4	*Term* + *Term*	↑x – 2 * y
7	*Factor* + *Term*	↑x – 2 * y
9	<id,x> + *Term*	↑x – 2 * y
9	<id,x> + *Term*	x ↑– 2 * y

An Example

- input: x – 2 * y
- "–" doesn't match "+"
- We need **backtracking** !



Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
4	*Term* + *Term*	↑x – 2 * y
7	*Factor* + *Term*	↑x – 2 * y
9	<id,x> + *Term*	↑x – 2 * y
9	<id,x> + *Term*	x ↑– 2 * y

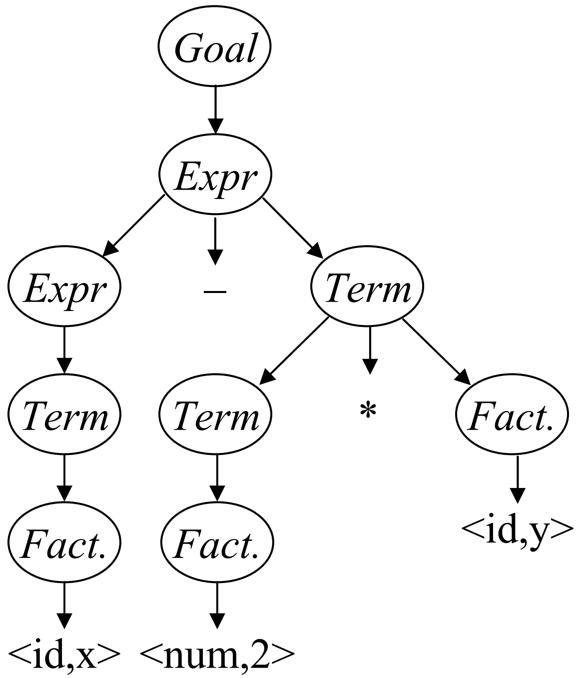
Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr* – *Term*	↑x – 2 * y
4	*Term* – *Term*	↑x – 2 * y
7	*Factor* – *Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y

An Example

- continue from x – 2 * y : one more **backtracking**

Rule	*Sentential Form*	*Input*
—	<id,x> – *Term*	x – ↑2 * y
7	<id,x> – *Factor*	x – ↑2 * y
9	<id,x> – <num,2>	x – ↑2 * y
—	<id,x> – <num,2>	x – 2 ↑* y

Rule	*Sentential Form*	*Input*
—	<id,x> – *Term*	x – ↑2 * y
5	<id,x> – *Term* * *Factor*	x – ↑2 * y
7	<id,x> – *Factor* * *Factor*	x – ↑2 * y
8	<id,x> – <num,2> * *Factor*	x – ↑2 * y
—	<id,x> – <num,2> * *Factor*	x – 2 ↑* y
—	<id,x> – <num,2> * *Factor*	x – 2 * ↑y
9	<id,x> – <num,2> * <id,y>	x – 2 * ↑y
—	<id,x> – <num,2> * <id,y>	x – 2 * y↑



Another Possible Parse

Other choices for expansion are possible : no input consume !

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr + Term*	↑x – 2 * y
2	*Expr + Term +Term*	↑x – 2 * y
2	*Expr + Term + Term +Term*	↑x – 2 * y
2	*Expr +Term + Term + …+Term*	↑x – 2 * y

This doesn't terminate

- Wrong choice of expansion leads to **non-termination**
- Non-termination is a **bad property** for a parser to have
- Parser must make the right choice

Left Recursion

- Top-down parsers **cannot handle left-recursive grammars**

Formally,

A grammar is ***left recursive*** if $\exists\ A \in NT$ such that

$\exists$ a derivation $\boldsymbol{A} \Rightarrow^{+} \boldsymbol{A\alpha}$, for some string $\alpha \in (NT \cup T)^{+}$

Our expression grammar is left recursive

- This can lead to **non-termination in a top-down parser**
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Eliminating Left Recursion

To remove left recursion, we can **transform the grammar**

Consider a grammar fragment of the form

$$\begin{aligned} Fee &\rightarrow Fee\ \alpha \\ &\mid\ \beta \end{aligned}$$

where neither α nor β start with ***Fee***

We can rewrite this as

$$\begin{aligned} Fee &\rightarrow \beta\ \boldsymbol{Fie} \\ \boldsymbol{Fie} &\rightarrow \alpha\ \boldsymbol{Fie} \\ &\mid\ \varepsilon \end{aligned}$$

where *Fie* is a new non-terminal

This accepts the same language, but uses only right recursion

Example

- Original Grammar

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	<u>number</u>
9		|	<u>id</u>

- New Grammar

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	– *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	<u>number</u>
11		|	<u>id</u>
12		|	<u>(</u> *Expr* <u>)</u>

Eliminating Left Recursion

- general algorithm:

arrange the NTs into some order $A_1, A_2, ..., A_n$

for $i \leftarrow 1$ **to** n

  for $s \leftarrow 1$ **to** $i - 1$

   replace each production $\boldsymbol{A_i \rightarrow A_s \gamma}$
    with $\boldsymbol{A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma}$,
    where $\boldsymbol{A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k}$ are
    all the current productions for A_s

   eliminate any immediate left recursion on A_i
    using the direct transformation

- 이제, termination은 보장… backtracking 해결 필요

Backtrack-Free Grammar

- parser가 backtrack을 안 하려면,
  항상 correct choice 를 하면 된다.
- key idea
  - $A \to \alpha \mid \beta$ 이고,
  - $\alpha \to^* \underline{\mathbf{a}}\,\gamma$ and $\beta \to^* \underline{\mathbf{b}}\,\lambda$
  - scanner에서 look-ahead 한 terminal 이
    a, b 중 어느 것이냐에 따라, correct choice 가능 !
  - a 이면, $A \to \alpha \to^* \underline{a}\,\gamma$ 로 진행
  - b 이면, $A \to \beta \to^* \underline{b}\,\lambda$ 로 진행
- **LL(1) grammar property** !

FIRST( )

- **FIRST(α)** is the set of words that can appear as the first symbol in some string derived from α.
  - $\alpha \in T \cup NT \cup \{ \varepsilon \}$
  - **$\underline{\underline{x}} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{\underline{x}}\,\gamma$, for some γ**
  - **$\varepsilon \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \varepsilon$**
- example
  - $Expr' \rightarrow +\ Term\ Expr'$
    $\quad\quad\quad | \ -\ Term\ Expr'$
    $\quad\quad\quad | \ \varepsilon$
  - FIRST(*Expr'*) = { +, –, ε }
- see page 99 for the detailed algorithm.

FOLLOW( )

- **FOLLOW(*A*)** is the set of symbols that can occur immediately after some non-terminal *A* in a valid sentence.
  - $B \rightarrow A\ \underline{a}$
    $A \rightarrow \underline{c}\ |\ \varepsilon$
  - FIRST(A) = { $\underline{c}$, ε }
  - FOLLOW(A) = { $\underline{a}$ }
  - $\varepsilon \in$ FIRST(A) 일 때 의미를 가짐
  - see page 99 for the detailed algorithm

- $\text{FIRST}^{+}(\alpha) = \text{FIRST}(\alpha)$ if $\varepsilon \notin \text{FIRST}(\alpha)$
- $\text{FIRST}^{+}(\alpha) = \text{FIRST}(\alpha) \cup \text{FOLLOW}(\alpha)$

LL(1) Property

- LL(1) property: **backtrack-free grammar**의 조건
  - for any non-terminal A with the rule $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$,
  - it must be true that $\text{FIRST}^+(\beta_i) \cap \text{FIRST}^+(\beta_j) = \emptyset, \quad \forall\ 1 \le i < j \le n$
- examples

Factor	→	<u>Identifier</u>
	|	<u>Identifier</u> <u>[</u> *ExprList* <u>]</u>
	|	<u>Identifier</u> <u>(</u> *ExprList* <u>)</u>

no backtrack-free

Factor	→	<u>Identifier</u> *Arguments*
Arguments	→	<u>[</u> *ExprList* <u>]</u>
	|	<u>(</u> *ExprList* <u>)</u>
	|	ε

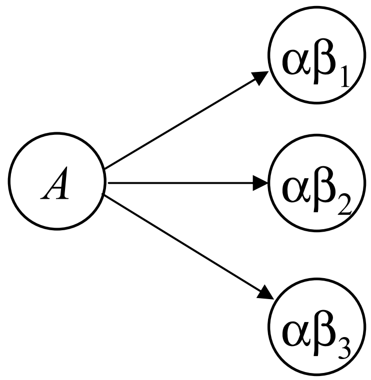
backtrack-free

Left Factoring

What if my grammar does not have the LL(1) property?

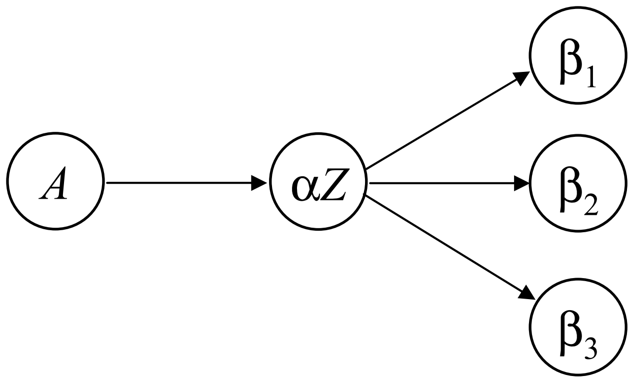
$\Rightarrow$ Sometimes, we can **transform the grammar**

$$
\begin{aligned}
A \rightarrow\ & \alpha\beta_1 \\
\mid\ & \alpha\beta_2 \\
\mid\ & \alpha\beta_3
\end{aligned}
$$



becomes

$$
\begin{aligned}
A &\rightarrow \alpha Z \\
Z &\rightarrow \beta_1 \\
&\mid\ \beta_2 \\
&\mid\ \beta_3
\end{aligned}
$$



Left Factoring

The Algorithm:

$\forall\ A \in NT$,

find **the longest prefix** α that occurs in two or more right-hand sides of A

if $\alpha \neq \varepsilon$ then replace all of the A productions,

$$\mathbf{A} \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n \mid \gamma\,,$$

with

$$\mathbf{A} \rightarrow \alpha\,\mathbf{Z} \mid \gamma$$

$$\mathbf{Z} \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where Z is a new element of NT

Repeat until no common prefixes remain

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Can **write a simple routine** to recognize each *lhs*
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$\textbf{FIRST}^+(\beta_1) \cap \textbf{FIRST}^+(\beta_2) \cap \textbf{FIRST}^+(\beta_3) = \varnothing$$

```
/* find an A */
if (current_word ∈ FIRST(β1))
   find a β1 and return true
else if (current_word ∈ FIRST(β2))
   find a β2 and return true
else if (current_word ∈ FIRST(β3))
   find a β3 and return true
else
   report an error and return false
```

직접 C code 작성 !

Recursive-Descent Parser

- one kind of predicted parser
  - 대표적인 hand-written LL(1) parser

1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Term Expr′*
3	*Expr′*	$\rightarrow$	+ *Term Expr′*
4		|	– *Term Expr′*
5		|	ε
6	*Term*	$\rightarrow$	*Factor Term′*
7	*Term′*	$\rightarrow$	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	$\rightarrow$	<u>number</u>
11		|	<u>id</u>

This produces a parser with six *<u>mutually recursive</u>* routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term *<u>descent</u>* refers to the direction in which the parse tree is built.

Recursive-Descent Parser

```
Goal( ) : Goal → Expr
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
        then next compilation step;
        else
            report syntax error;
            return false;

Expr( ) : Expr → Term Expr'
  if (Term( ) = false)
    then return false;
    else return Eprime( );
```

```
Factor( ) : Factor → number | id
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
     token ← next_token( );
     return true;
  else
     report syntax error;
     return false;
```

- see page 105 for all routines

Recursive Descent (Summary)

1. Build FIRST (and FOLLOW) sets
2. Massage grammar to have *LL(1)* condition
    a. Remove left recursion
    b. Left factor it
3. Define a procedure for each non-terminal
    a. Implement a case for each right-hand side
    b. Call procedures as needed for non-terminals
4. Add extra code, as needed
    a. Perform context-sensitive checking
    b. Build an IR to record the code

Can we automate this process?

Building Top-down Parsers

Given an *LL(1)* grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate RHS's
  - Each returns true on success and throws an error on false
  - Simple, working **(, *perhaps ugly*,)** code

Improving matters

- **What about a table to encode the options?**
  - Interpret the table with a skeleton, as we did in scanning

Building Top-down Parsers

Strategy

- **Encode knowledge in a table**
- Use a standard "skeleton" parser to interpret the table

Example

- 10: Factor → id
- 11: Factor → number
- Table might look like:

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	EOF
Factor	—	—	—	—	10	11	—

Error on `+ '

Reduce by rule 10 on 'Id'

Building Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an algorithm to build the table

Filling in TABLE[X, y], $X \in NT$, $y \in T$

1. entry is the rule $X \rightarrow \beta$**, if y** $\in$ **FIRST(**β**)**
2. entry is the rule $X \rightarrow \varepsilon$
   if $y \in$ **FOLLOW(**X**) and** $X \rightarrow \varepsilon \in$ **G**
3. entry is **error** if **neither 1 nor 2 define it**

If any entry is defined multiple times, G is not *LL(1)*

This is the ***LL(1)*** **table construction algorithm**

3.4 Bottom-Up Parsing

Parsing Techniques

Top-down parsers LL(1)

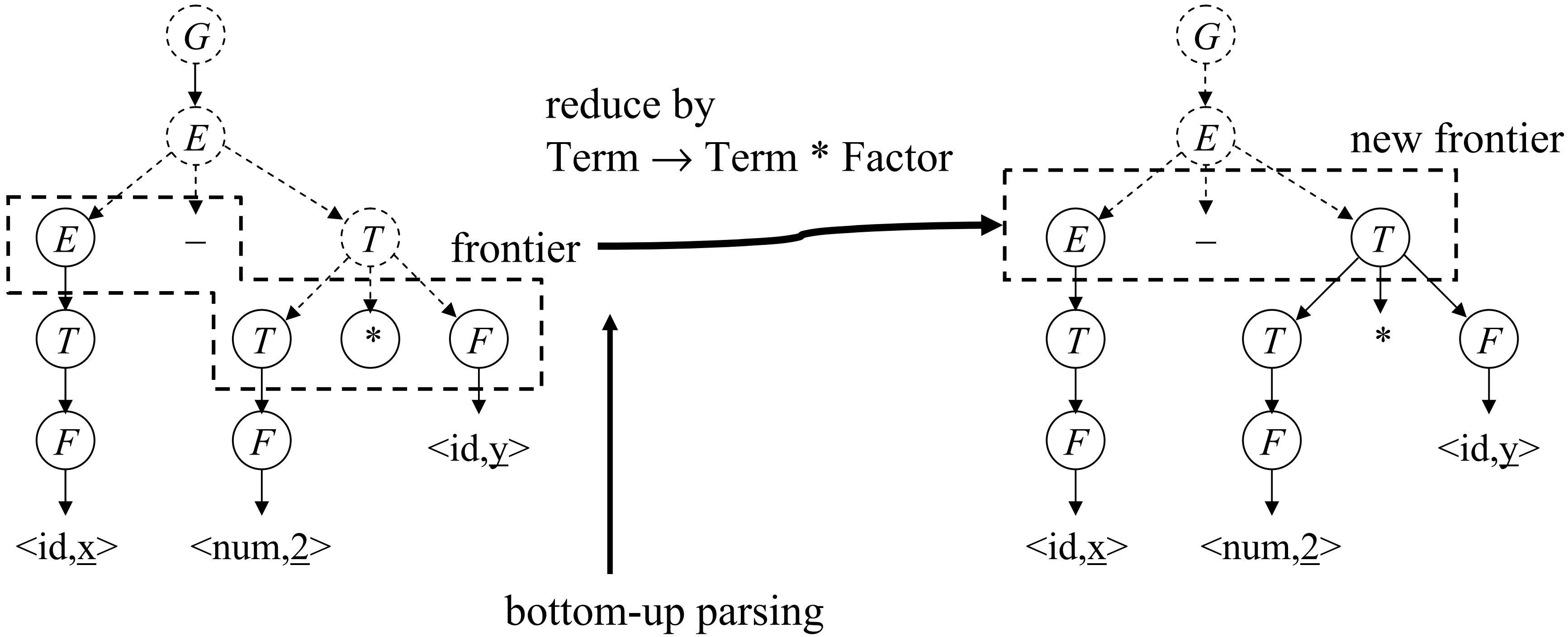
- Start at the root of the parse tree and grow toward leaves
- **Pick a production & try to match the input**
- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free

Bottom-up parsers LR(1)

- Start at the leaves and grow toward root
- As input is consumed, **encode possibilities in an internal state**
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Frontier & Reduce

- **frontier** (= upper frontier)
  – bottom-up parsing 중의, 현재 가장 upper node들



Bottom-up Parsing

- parsing의 핵심은 derivation

  $$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow sentence$$

  – **sentence** : a set of terminals (no non-terminal)
- bottom-up parsing
  – input string : 완전히 derive된 sentence 라고 가정
  – parsing 과정 : **sentence**에서 **start symbol S**까지 형성

  $$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow sentence$$

  $\longleftarrow$ bottom-up

- To reduce γ_i to γ_{i-1},
  find β in γ_i, then replace it using $A \to \beta$ (handle)
  – $\gamma_{i-1} = \alpha\, A\, \delta \Rightarrow \gamma_i = \alpha\, \beta\, \delta$

  $\longleftarrow$ bottom-up

Handle

- rightmost derivation을 가정
  - 실제로는 backward로 처리 $\rightarrow$ leftmost *NT*부터 처리
- A **handle** of a right-sentential form γ is **a pair** $\boldsymbol{<A \rightarrow \beta, k>}$
  - $\boldsymbol{A \rightarrow \beta \in P}$
  - k is **the position** in γ of β's rightmost symbol.
- handle을 선택하면,
  - **replacing** $\boldsymbol{\beta}$ **at** $\boldsymbol{k}$ **with** A produces
    the right sentential form
    from which γ is derived in the rightmost derivation.
  - right-sentential form이므로,
    right of a handle은 only terminal symbols

Example: Grammar

- original grammar

2	*Expr*	$\rightarrow$	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	$\rightarrow$	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	$\rightarrow$	<u>number</u>
9		|	<u>id</u>
10		|	<u>(</u> *Expr* <u>)</u>

- augmented grammar
  - $S \rightarrow S'$ 형태를 추가
  - 성공(accept) 판정을 쉽게

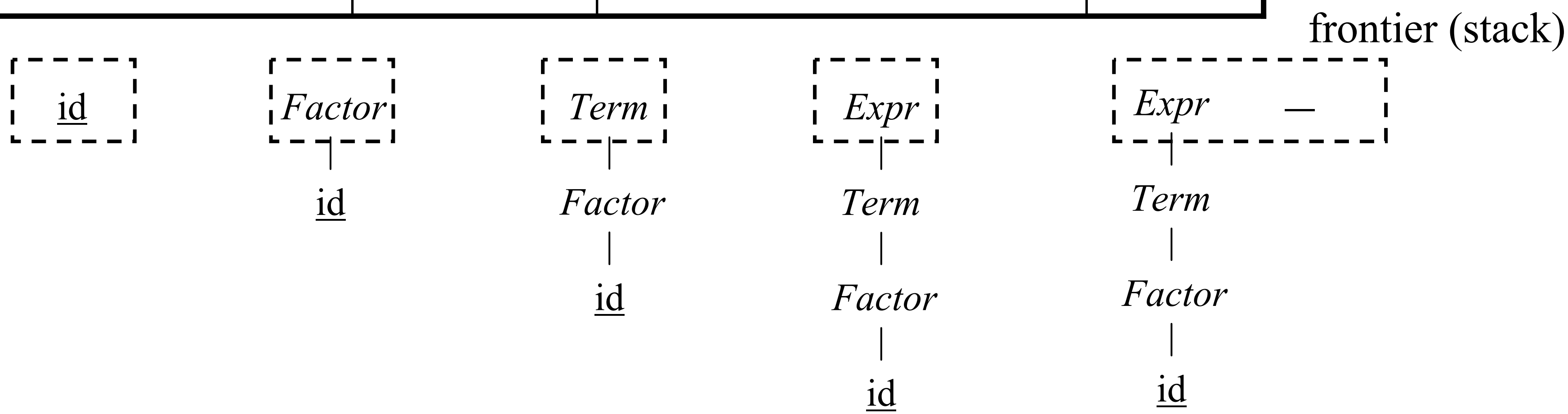
1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	$\rightarrow$	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	$\rightarrow$	<u>number</u>
9		|	<u>id</u>
10		|	<u>(</u> *Expr* <u>)</u>

Example: parsing states

frontier (stack)	next token	handle	action
	id – num * id		shift
id	– num * id	<*Factor* → id, 1>	reduce
Factor	– num * id	<*Term* → *Factor*, 1>	reduce
Term	– num * id	<*Expr* → *Term*, 1>	reduce
Expr	– num * id		shift
Expr –	num * id		shift
Expr – num	* id	<*Factor* → num, 3>	reduce
Expr – *Factor*	* id	<*Term* → *Factor*, 3>	reduce
Expr – *Term*	* id		shift
Expr – *Term* *	id		shift
Expr – *Term* * id	<eof>	<*Factor* → id, 5>	reduce
Expr – *Term* * *Factor*	<eof>	<*Term* → *Term* * *Factor*, 5>	reduce
Expr – *Term*	<eof>	<*Expr* → *Expr* – *Term*, 3>	reduce
Epxr	<eof>	<*Goal* → *Expr*, 1>	reduce
Goal	<eof>		accept

Example: parse tree

frontier (stack)	next token	handle	action
	id – num * id		shift
id	– num * id	<*Factor* → id, 1>	reduce
Factor	– num * id	<*Term* → *Factor*, 1>	reduce
Term	– num * id	<*Expr* → *Term*, 1>	reduce
Expr	– num * id		shift
Expr –	num * id		shift
…	…	…	…



Bottom-up Parser의 action

- data structure : a stack + scanner output
- **shift** : scanner → push on the stack
  - how many shifts in a parser?   # of tokens
- **reduce** : pop from the stack → push a NT on the stack
  - how many reduces in a parser?
    - O(rules) = O(# of tokens)
- so, finite !

- reduce에서의 특징 : stack top 에서 pop
  - → more simple notation
  - **place holder : •** → current stack top을 의미

Example: new handle notation

frontier (stack)	next token	handle (new notation)	action
	id – num * id		shift
id	– num * id	<*Factor* → id • >	reduce
Factor	– num * id	<*Term* → *Factor* • >	reduce
Term	– num * id	<*Expr* → *Term* • >	reduce
Expr	– num * id		shift
Expr –	num * id		shift
Expr – num	* id	<*Factor* → num • >	reduce
Expr – *Factor*	* id	<*Term* → *Factor* • >	reduce
Expr – *Term*	* id		shift
Expr – *Term* *	id		shift
Expr – *Term* * id	<eof>	<*Factor* → id • >	reduce
Expr – *Term* * *Factor*	<eof>	<*Term* → *Term* * *Factor* • >	reduce
Expr – *Term*	<eof>	<*Expr* → *Expr* – *Term* • >	reduce
Epxr	<eof>	<*Goal* → *Expr* • >	reduce
Goal	<eof>		accept

Potential Handle

frontier (stack)	next token	handle (new notation)	action
	id – num * id		shift
id	– num * id	<*Factor* → id • >	reduce
Factor	– num * id	<*Term* → *Factor* • >	reduce
Term	– num * id	<*Expr* → *Term* • >	reduce
Expr	– num * id	why empty ?	shift
…	…	…	…

rules with *Expr* on RHS:

Goal → *Expr*

Expr → *Expr* + *Term*

Expr → *Expr* – *Term*

potential handles:

Goal → *Expr* •

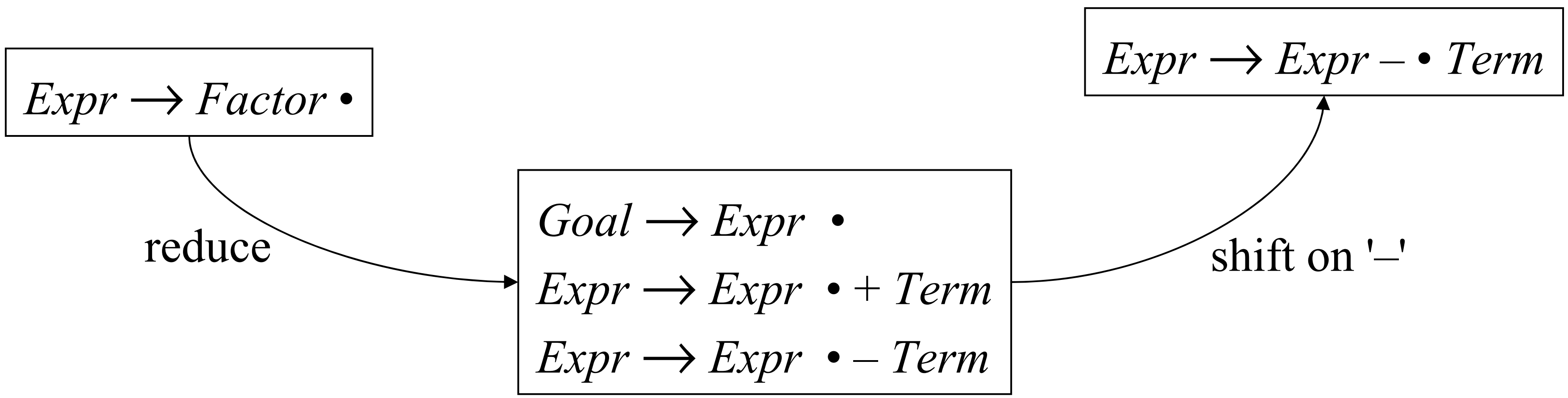
Expr → *Expr* • + *Term*

Expr → *Expr* • – *Term*

이 중의 하나이다 !

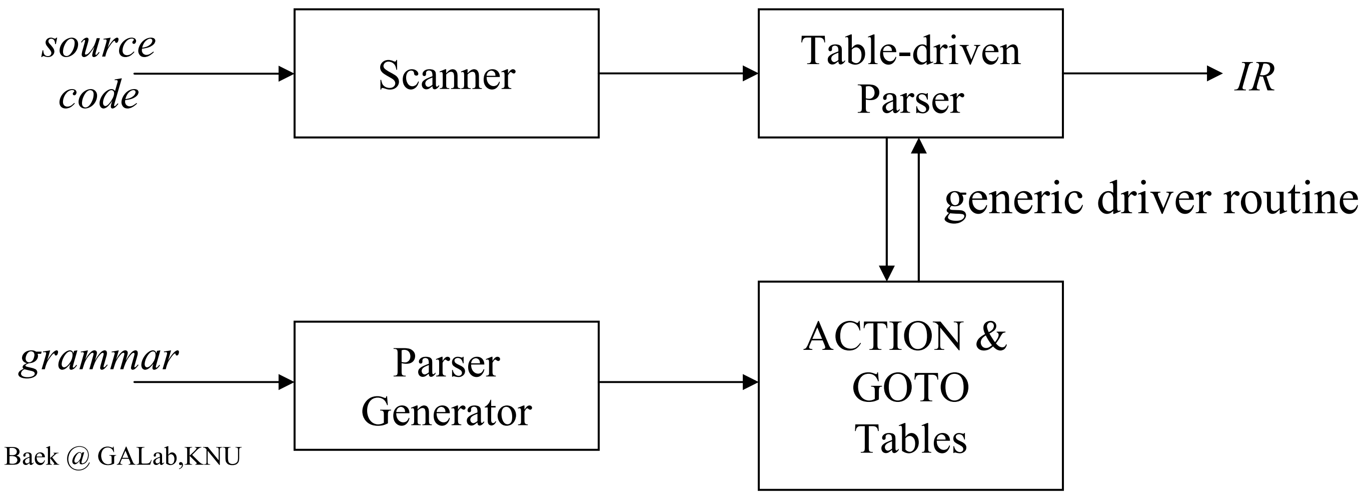
LR(1) Parser Strategy

- rules → all possible handles
  - rule with k symbols → $(k + 1)$ possible handles
- **handle recognizing DFA**
  - **states** : power set of all possible handles
  - **shift** : jump to another state with a token shift
  - **reduce** : jump to another state with a stack reduction



LR(1) Parser

- LR(1)
  - **left-to-right scan** : scanner는 오직 한번 left-to-right scan
  - **rightmost derivation** : parsing tree는 rightmost derivation
  - **1 symbol of look-ahead**
- handle recognizing DFA + frontier stack 사용
  - DFA state + look-ahead symbol → new DFA state
  - DFA transition function → 2차원 table



LR(1) Parser의 4가지 action

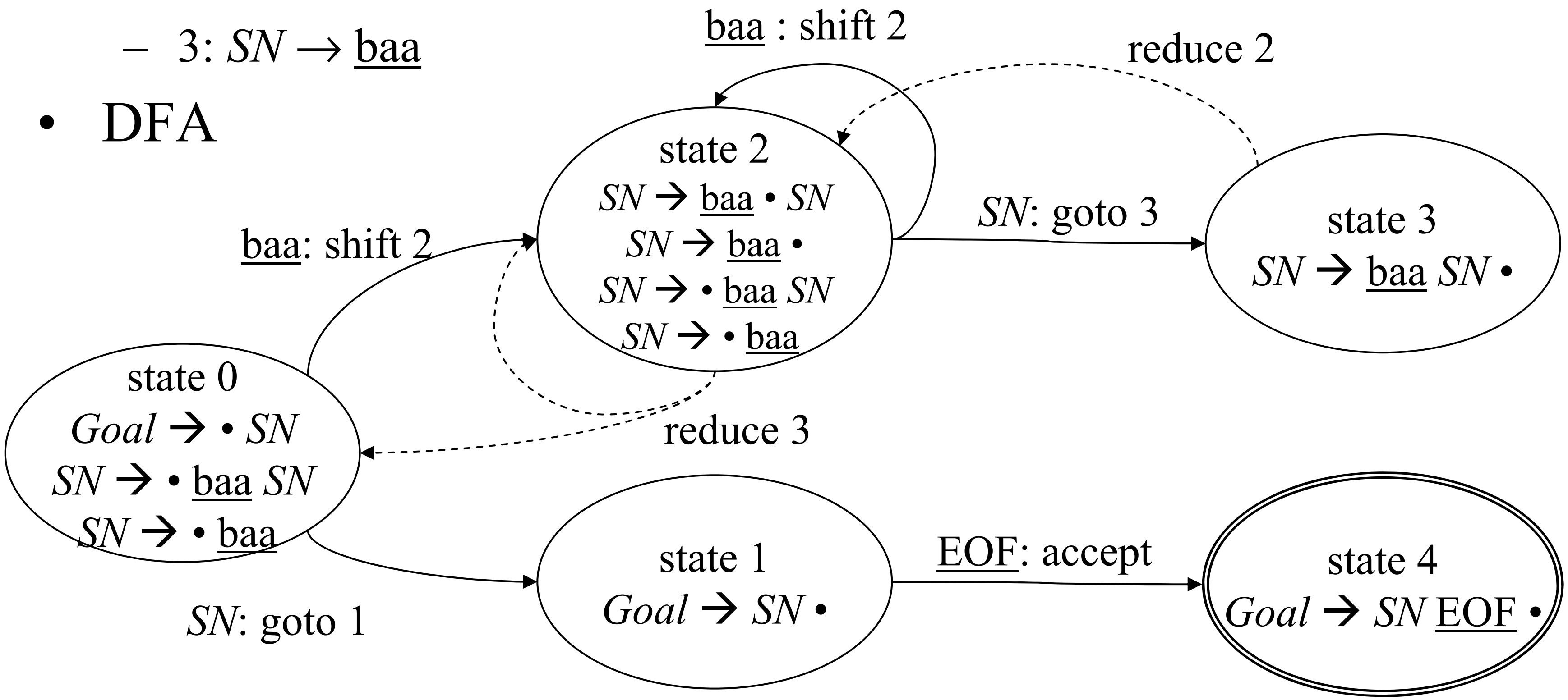
- handle finding을 좀더 쉽게 하려면?
  - handle recognition **DFA**의 **state**도 **stack**에 기억하자 **!**
  - stack에 symbol, state가 교대로 push 된다.
- **shift** : get a token x, then push x, s_{new}
  - … $(y, s_a) \rightarrow$ … $(y, s_a), (x, s_{\text{new}})$
- **reduce** : apply a rule and go to a new state
  - … $(w, s_a), (x, s_b), (y, s_c), (z, s_d) \rightarrow$ … $(w, s_a), (A, s_{\text{new}})$
- **accept** : stop parsing & report success
  - $(\$, s_0), (\mathit{Goal}, s_n)$, token = EOF $\rightarrow$ accept !
- **error** : call an error reporting / recovery routine

Two Tables

- **action table**
  - (*state*, *token*)
    $\rightarrow$ shift $state_{new}$
  - (*state*, *token*)
    $\rightarrow$ reduce by the rule *number*
    - reduce 후의 $state_{new}$ 는 ?
  - (*state*, EOF)
    $\rightarrow$ accept
  - undefined : error
- **goto table**
  - (*state*, *non-terminal*)
    $\rightarrow$ go to $state_{new}$
    - non-terminal은 reduce 후에 생긴 것

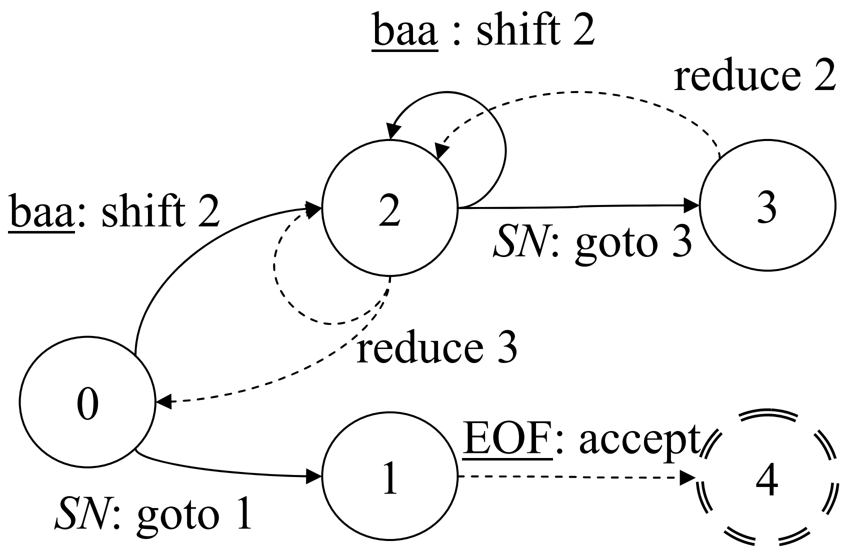
Example : SheepNoise

- augmented grammar
  - 1: *Goal* $\rightarrow$ *SN*
  - 2: *SN* $\rightarrow$ baa *SN*
  - 3: *SN* $\rightarrow$ baa
- DFA



Example : SheepNoise

- two tables



ACTION		
State	EOF	baa
0	—	shift 2
1	accept	—
2	reduce 3	shift 2
3	reduce 2	—

GOTO	
State	*SheepNoise*
0	1
1	—
2	3
3	—

Example: "baa"

- input string "baa"

Stack	Input	Action
$ 0	baa EOF	shift 2
$ 0 baa 2	EOF	reduce 3
$ 0 *SN* 1	EOF	accept

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	—
2	reduce 3	shift 2
3	reduce 2	—

GOTO	
State	*SheepNoise*
0	1
1	—
2	3
3	—

Example: "baa baa"

- input string "baa baa"

Stack	Input	Action
\$ 0	baa baa EOF	shift 2
\$ 0 baa 2	baa EOF	shift 2
\$ 0 baa 2 baa 2	EOF	reduce 3
\$ 0 baa 2 *SN* 3	EOF	reduce 2
\$ 0 *SN* 1	EOF	accept

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	—
2	reduce 3	shift 2
3	reduce 2	—

GOTO	
State	*SheepNoise*
0	1
1	—
2	3
3	—

LR(1) Parser Again

- **parser driver routine** : see page 116 for more details

  push \$, 0 to the stack
  token ← from scanner
  while (true)
  state ← top(stack)
  if Action[*state*, *token*] = "shift *i*" **then** push token, *i*;
  else if Action[*state*, *token*] = "reduce $A \rightarrow \beta$" **then**
  pop 2 * | β | symbols; *state* ← top(stack);
  push *A*, Goto[*state*, *A*]
  else if Action[*state*, *token*] = "accept" **then** return
  else error **endif**

- **remaining thing ?**
  - how to (automatically) construct the tables ?

3.5 Building LR(1) Tables

LR(1) Items

- a pair $[A \rightarrow \beta \bullet \gamma, \mathrm{a}]$
  - $A \rightarrow \beta \bullet \gamma$: a handle or a potential handle
    - Caution: β, γ는 a sequence of symbols
  - $\mathrm{a} \in T$: look-ahead token : 모든 token을 cover해야 함!
  - handle-recognition DFA의 basic item
- possibility: $[A \rightarrow \bullet \beta \gamma, \mathrm{a}]$
- partially complete: $[A \rightarrow \beta \bullet \gamma, \mathrm{a}]$
- complete: $[A \rightarrow \beta \gamma \bullet, \mathrm{a}]$

Example: Sheep Noise

- augmented grammar

  1: $Goal \rightarrow SN$

  2: $SN \rightarrow$ baa SN

  3: $SN \rightarrow$ baa

- all LR(1) items : 실제로 다 쓰이지는 않음…

  [$Goal \rightarrow \bullet\ SN$, EOF]

  [$Goal \rightarrow SN\ \bullet$, EOF]

  [$SN \rightarrow \bullet$ baa, EOF]

  [$SN \rightarrow$ baa $\bullet$, EOF]

  [$SN \rightarrow \bullet$ baa, baa]

  [$SN \rightarrow$ baa $\bullet$, baa]

  [$SN \rightarrow \bullet$ baa SN, EOF]

  [$SN \rightarrow$ baa $\bullet\ SN$, EOF]

  [$SN \rightarrow$ baa $SN\ \bullet$, EOF]

  [$SN \rightarrow \bullet$ baa SN, baa]

  [$SN \rightarrow$ baa $\bullet\ SN$, baa]

  [$SN \rightarrow$ baa $SN\ \bullet$, baa]

Closure(*s*) and Goto(*s*,*x*)

- closure(s) : DFA state 만들기
  - 같은 상황인 LR(1) item들
    → DFA state

closure(s : a set of states)
  while (s is still changing)
    $\forall$ item $[A \rightarrow \beta \bullet C\ \delta, \mathrm{a}] \in \mathrm{s}$
      $\forall$ rule $C \rightarrow \gamma$
        $\forall\ \mathrm{b} \in \mathrm{FIRST}(\delta \mathrm{a})$
          $s \leftarrow s \cup \{\ [C \rightarrow \bullet\ \gamma, \mathrm{b}]\ \}$

- Example: 아래 3개는 같은 상황
  - $[\mathit{Goal} \rightarrow \bullet\ \mathit{SN}, \mathrm{EOF}]$
  - $[\mathit{SN} \rightarrow \bullet\ \underline{\mathrm{baa}}, \mathrm{EOF}]$
  - $[\mathit{SN} \rightarrow \bullet\ \underline{\mathrm{baa}}\ \mathit{SN}, \mathrm{EOF}]$

- goto(s, x) : DFA transition
  - 어디로 가야 하나

goto(s, x : an NT or T)
  $\mathit{moved} \leftarrow \varnothing$
  $\forall$ item $i \in s$
    if i is $[A \rightarrow \beta \bullet x\ \delta, \mathrm{a}]$ then
      moved $\leftarrow$ moved
        $\cup\ \{\ [A \rightarrow \beta\, x \bullet \delta, \mathrm{a}]\ \}$
  return closure(*moved*)

Canonical Collection Construction

- $\mathcal{CC}$: canonical collection of sets of LR(1) items
  – final parser에서 하나의 state가 되는 items
  – $\mathcal{CC} = \{ CC_0, CC_1, CC_2, \ldots \}$
- $\mathcal{CC}$ construction algorithm

  $\mathcal{CC} \leftarrow \{ CC_0 = \text{closure}(\{ [S' \rightarrow \bullet\, S, \text{EOF}] \}) \}$
  while ($\mathcal{CC}$ is still changing)
    $\forall$ set $CC_j \in \mathcal{CC}$
      $\forall$ x following a • in an item in CC_j
        $CC_k \leftarrow \text{goto}(CC_j, \text{x})$ // 내부에서 closure( ) 계산 !
        if $CC_k \notin \mathcal{CC}$ then add it to $\mathcal{CC}$
        record the transition $CC_j \rightarrow CC_k$ on x

Example: SheepNoise

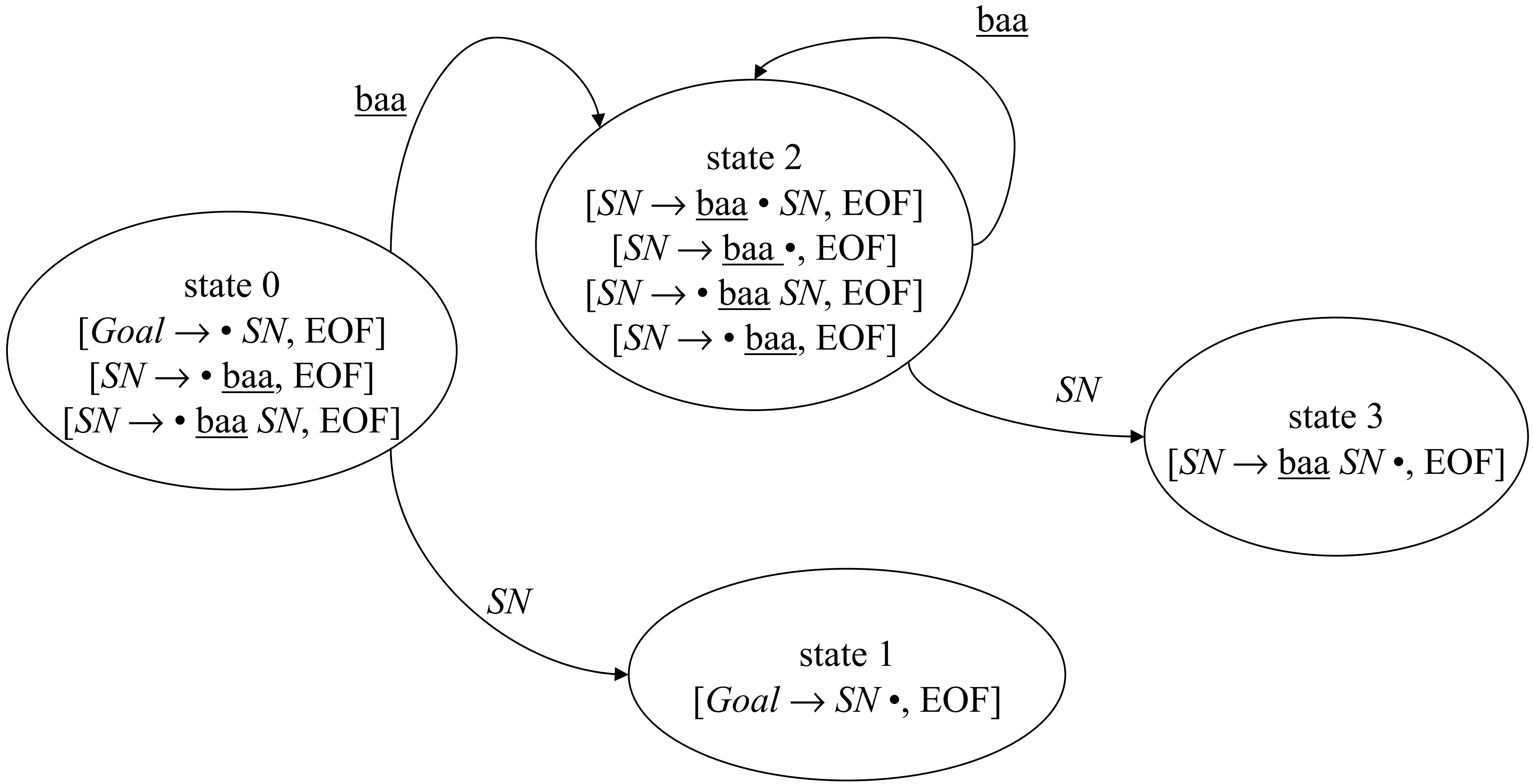
- CC_0 = closure([$Goal \rightarrow \bullet\ SN$, EOF])
  = { [$Goal \rightarrow \bullet\ SN$, EOF], [$SN \rightarrow \bullet$ <u>baa</u> SN, EOF], [$SN \rightarrow \bullet$ <u>baa</u>, EOF] }

- CC_1 = goto(CC_0, SN)
  = { [$Goal \rightarrow SN\ \bullet$, EOF] }

- CC_2 = goto(CC_0, baa)
  = { [$SN \rightarrow$ <u>baa</u> $\bullet\ SN$, EOF], [$SN \rightarrow$ <u>baa</u> $\bullet$, EOF],
  [$SN \rightarrow \bullet$ <u>baa</u> SN, EOF], [$SN \rightarrow \bullet$ <u>baa</u>, EOF] }

- CC_3 = goto(CC_2, SN)
  = { [$SN \rightarrow$ <u>baa</u> $SN\ \bullet$, EOF] }

Example: Final DFA



From DFA to the Table

- item $[A \rightarrow \beta \bullet C\ \delta, \text{a}]$: make a **shift** entry in Action Table
  - on C, shift to the state with $[A \rightarrow \beta\, C \bullet \delta, \text{a}]$
- item $[A \rightarrow \beta \bullet, \text{a}]$: make a **reduce** entry in Action Table
  - reduce by the rule $A \rightarrow \beta$
- special item $[S \rightarrow S' \bullet, \text{EOF}]$: **accept** in Action Table

- making **Goto** Table

  for each $n \in NT$,

  if goto(CC_j, n) = CC_k then Goto[j, n] $\leftarrow k$

Final Table

- two table form

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	—
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	—
2	3
3	—

- single table form

state	EOF	baa	*SN*
0		shift 2	goto 1
1	accept		
2	reduce 3	shift 2	goto 3
3	reduce 2		

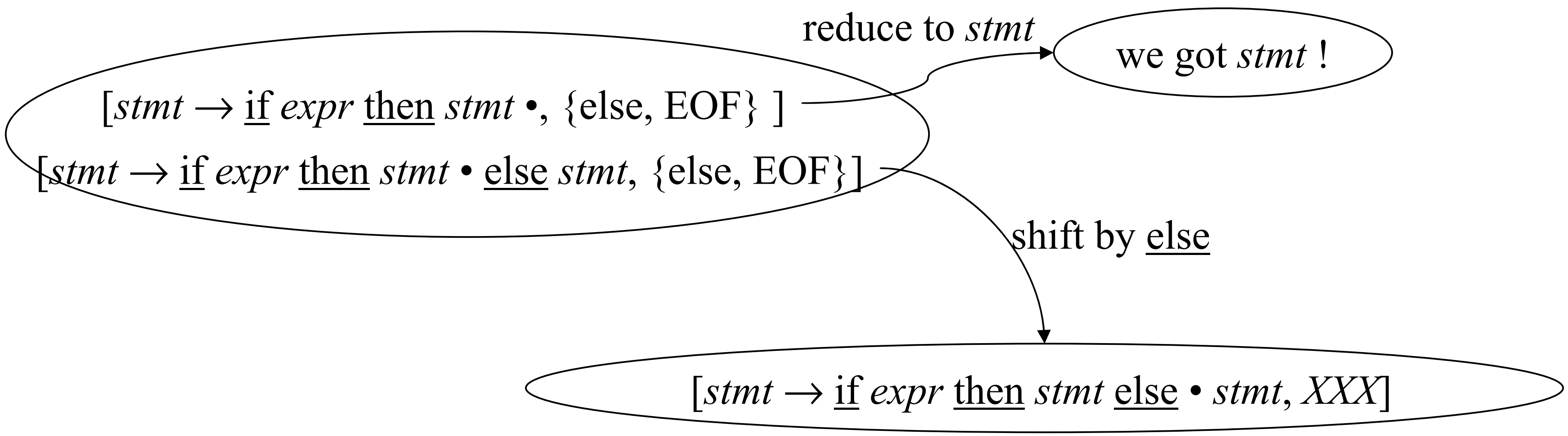
Shift-Reduce Conflict

- ambiguous if-then-else grammar

  stmt → **if** *expr* **then** *stmt*
  | **if** *expr* **then** *stmt* **else** *stmt*
  | *assign*

  - *CC* 생성 중에, 다음 상황 발생
    - Action table에 **conflict 발생**

- 해결책 ?
  - change your grammar
  - choose shift always
    - yacc의 해법



Reduce-Reduce Conflict

- in Fortran and Ada,

  Factor → *FuncCall*
  | *ArrayRef*
  | others…

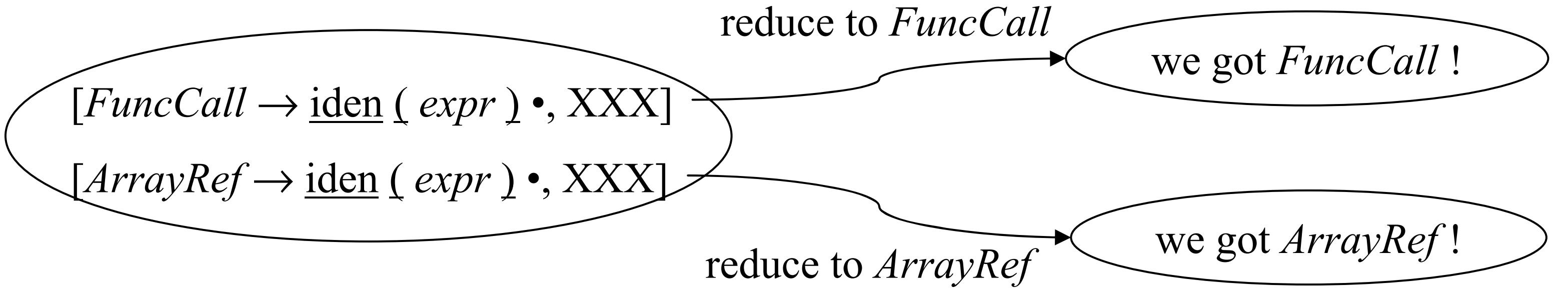
  FuncCall → iden (*expr*)

  ArrayRef → iden (*expr*)
- *CC* 생성 중에, 다음 상황 발생

- 해결책 ?
  - change your language…
  - scanner + symbol table 처리
    - symbol table은 iden이 function인지, array인지 앎.

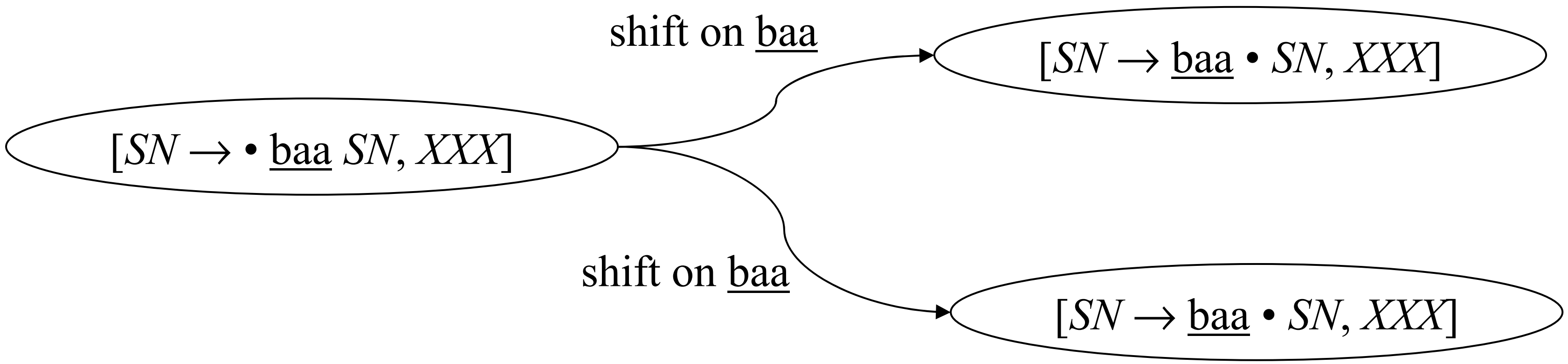
  FuncCall → funcname (*expr*)

  ArrayRef → arrayname (*expr*)



Shift-Shift Conflict

- shift-shift conflict
  - **never occurs !**
  - why?   we use DFA rather than NFA.



LL(1) vs. LR(1)

	Advantages	Disadvantages
Top-down recursive descent	Fast Good locality Simplicity **Good error detection**	Hand-coded High maintenance **Right-Recursive Grammar**
LR(1)	Fast Deterministic langs. Automatable **Left or Right-Recursive Grammar**	Large working sets Poor error messages Large table sizes

3.6 Practical Issues

Error Recovery

- LL(1) 또는 LR(1) parser의 교과서적 구현
  → single error detection and stop
- practical parser
  – **multiple error detection** 필요
  – how?
    - C/C++/Java 등: 모든 문장은 ';' 으로 끝
    - error 발생 시의 처리
      – scanner : ';' 까지 계속 eating
      – parser : error가 발생한 stmt 직전까지 pop

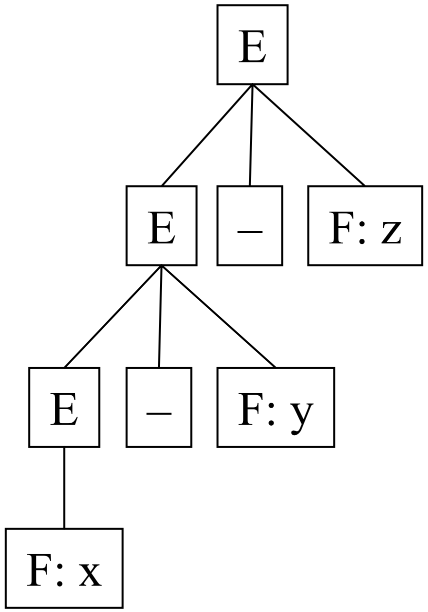
Left and Right Recursion

- left-recursive grammar
  - LL(1), LR(1)에서 가능
- *list* → *list* elem
      | elem
- example : elem elem elem
  - • elem elem elem : shift
  - elem • elem elem : reduce
  - *list* • elem elem : shift
  - *list* elem • elem : reduce
  - *list* • elem : shift
  - *list* elem • : reduce
  - *list* • : accept
- 이쪽을 선호 !

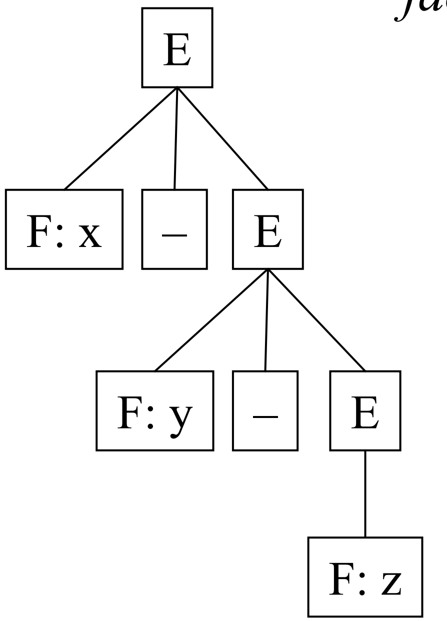
- right recursive grammar
  - LR(1)에서 가능
- *list* → elem *list*
      | elem
- example: elem elem elem
  - • elem elem elem : shift
  - elem • elem elem : shift
  - elem elem • elem : shift
  - elem elem elem • : reduce
  - elem elem *list* • : reduce
  - elem *list* • : reduce
  - *list* • : accept
- **stack depth: 훨씬 깊어 진다 !**

Associativity

- left-recursive grammar
- *expr* → *expr* '+' *fact*
  | *expr* '–' *fact*
  | *fact*

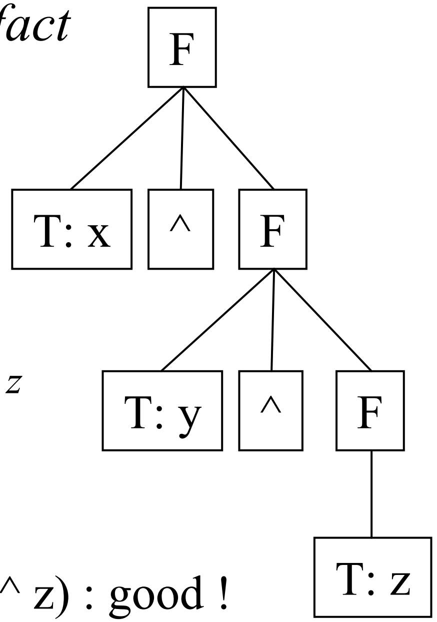


(x – y) – z : good !



x – (y – z) : bad…

- right recursive grammar
- *expr* → *fact* '+' *expr*
  | *fact* '–' *expr*
  | *fact*
- *fact* → *term* '^' *fact*
  | *term*



x^{y^z}

x ^ (y ^ z) : good !

3.7 Advanced Topics

Optimizing a Grammar

- original grammar

Expr → *Expr* + *Term*
  | *Expr* – *Term*
  | *Term*
Term → *Term* * *Fact*
  | *Term* / *Fact*
  | *Fact*
Fact → (*Expr*)
  | num
  | iden

- optimized grammar
  - parse tree depth 축소 효과

Expr → *Expr* + *Term*
  | *Expr* – *Term*
  | *Term*
Term → *Term* * (*Expr*)
  | *Term* * num
  | *Term* * iden
  | *Term* / (*Expr*)
  | *Term* / num
  | *Term* / iden
  | (*Expr*)
  | num
  | iden

Shrinking the Grammar

- original grammar
  - scanner의 return 형태:
    - <+, NULL>, …

Expr → *Expr* + *Term*
  | *Expr* – *Term*
  | *Term*
Term → *Term* * *Fact*
  | *Term* / *Fact*
  | *Fact*
Fact → (*Expr*)
  | num
  | iden

- reduced grammar
  - scanner의 return 형태:
    - <addop, +>, <mulop,*>, …
  - parser가 간단해짐

Expr → *Expr* addop *Term*
  | *Term*
Term → *Term* mulop *Fact*
  | *Fact*
Fact → (*Expr*)
  | num
  | iden

Other Construction Algorithms

- **LR(1)** : canonical LR(1) parsing
- **SLR(1)** : simple LR(1) parsing
  - LR(1)에서 lookahead를 제거한 LR(1) item 사용
  - power가 떨어짐 → 그러나, 대부분 문제 없음
- **LALR(1)** : lookahead LR(1) parsing
  - LR(1)에서, 모든 item이 아니라, 대표 item만 사용
  - 대부분의 programming language에서는 무난
  - yacc는 LALR(1) 사용

  - **LALR(1) ⊂ SLR(1) ⊂ LR(1) ⊂ CFG**