CS 314 Principles of Programming Languages

Lecture 7: LL(1) Parsing

Zheng (Eddy) Zhang

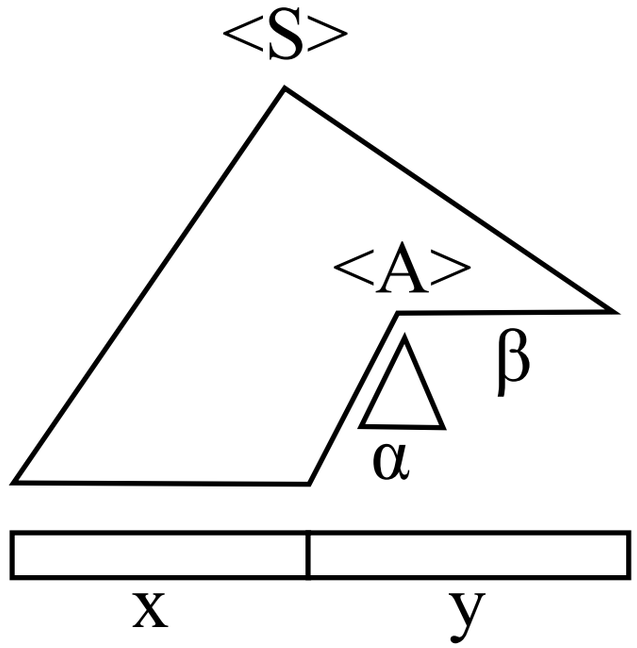
Rutgers University

February 7, 2018

Class Information

- Homework 3 will be posted this coming Monday.

Review: Top-Down Parsing - LL(1)



Basic Idea:

- The parse tree is constructed from the root, expanding non-terminal nodes on the tree's frontier following a **leftmost** derivation.
- The input program is read from **left** to right, and input tokens are read (consumed) as the program is parsed.
- The next non-terminal symbol is replaced using one of its rules. The particular choice has to be unique and uses parts of the input (partially parsed program), for instance the first token of the remaining input.

Review: Predictive Parsing

Basic idea:

For any two productions A ::= α and A ::= β, we would like *a distinct way of choosing the correct production to expand.*

First Set

For some string α, define **FIRST**(α) as the set of tokens that appear as the first symbol in some string derived from α.

That is

$x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$ for some string γ

Follow Set

For a non-terminal A, define **FOLLOW**(A) as the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's **FOLLOW** set specifies the tokens that can legally appear after it. A terminal symbol has no **FOLLOW** set.

FIRST and FOLLOW sets can be constructed automatically

Predictive Parsing

Key Property:

Whenever two productions A ::= α and A ::= β both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$, and
- if $\alpha \Rightarrow^* \varepsilon$, then $FIRST(\beta) \cap FOLLOW(A) = \varnothing$

Analogue case for $\beta \Rightarrow^* \varepsilon$.

Note: due to first condition, at most one of α and β can derive ε.

> This would allow the parser to make a correct choice with a lookahead of only one symbol!

LL(1) Grammar

Define $FIRST^{+}(A ::= \delta)$ for rule $A ::= \delta$

- $FIRST(\delta)$ - { ε } U Follow (A), if $\varepsilon \in FIRST(\delta)$
- $FIRST(\delta)$ otherwise

A Grammar is LL(1) iff

(A ::= α and A ::= β) implies

$$FIRST^{+}(A ::= \alpha) \cap FIRST^{+}(A ::= \beta) = \varnothing$$

Back to Our Example

Start ::= S **eof**
S ::= **a** S **b** | ε

FIRST(aSb) = {a}

FIRST(ε) = {ε}

FOLLOW(S) = {eof, b}

Is the grammar LL(1)?

$FIRST^{+}$(S ::= aSb) = {a}

$FIRST^{+}$(S ::= ε) = (*FIRST*(ε) - {ε}) ∪ *FOLLOW*(S)

Define $FIRST^{+}$(A ::= δ) for rule A ::= δ

- *FIRST* (δ) - { ε } U Follow (A), if ε ∈ *FIRST*(δ)
- *FIRST* (δ) otherwise

Table Driven LL(1) Parsing

Example:

S ::= **a** S **b** | ε

$FIRST^{+}$(S ::= aSb) = {a}

$FIRST^{+}$(S ::= ε) ={eof, b}

LL(1) parse table

Terminal Symbols

Non-Terminal Symbols

	a	b	eof	other
S	S ::= aSb	S ::= ε	S ::= ε	error

A row represents one non-terminal (NT)
A column represents one terminal (T)

Table Driven LL(1) Parsing

Input: *a string w and a parsing table M for G*

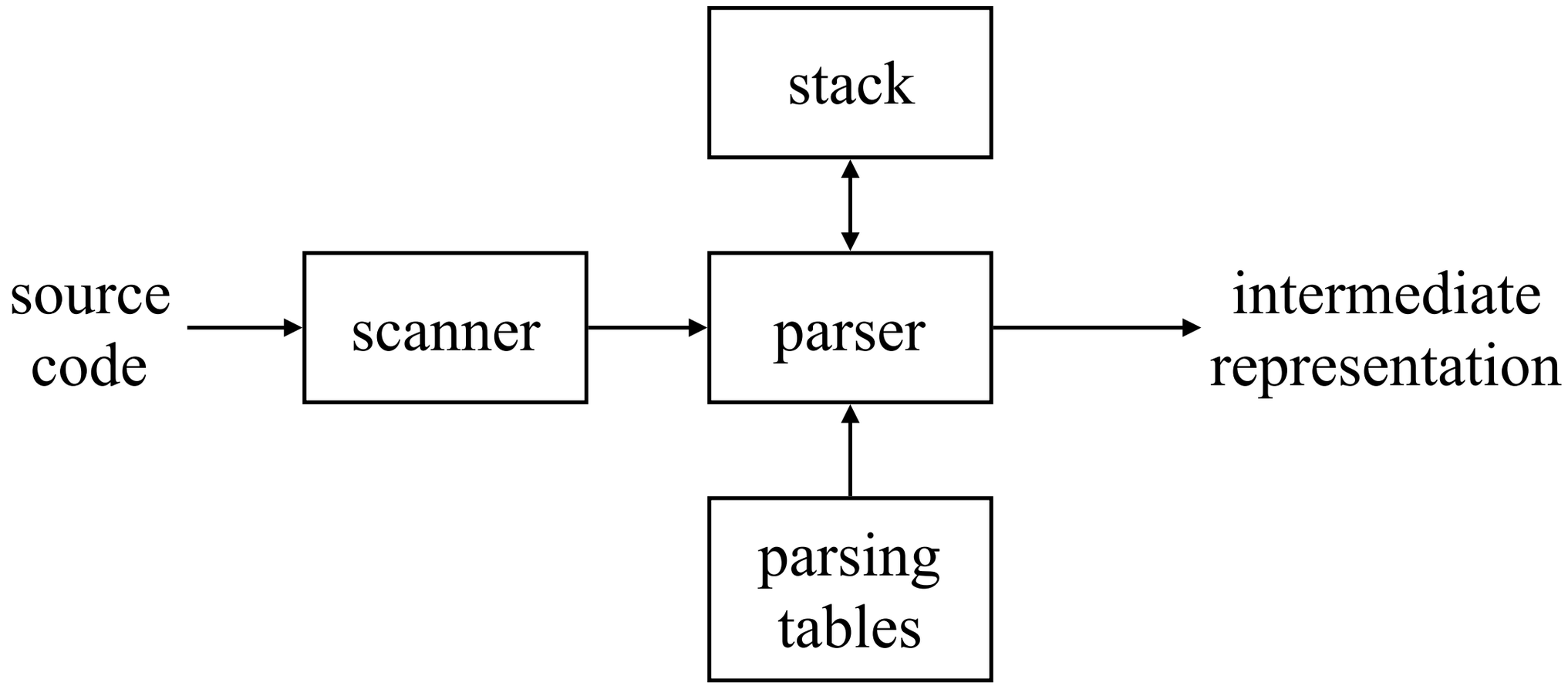
```
push eof
push Start Symbol
token ← next_token()
X ← top-of-stack
repeat
    if X is a terminal then
      if X == token then
          pop X
          token ← next_token()
      else error()
    else /* X is a non-terminal */
          if M[X, token] == X → Y1Y2 . . . Yk then
              pop X
              push Yk, Yk–1, . . . , Y1
           else error()
    X ← top-of-stack
until X = EOF
if token != EOF then error()
```

	a	b	eof	other
S	S ::= aSb	S ::= ε	S ::= ε	error

M is the parse table

Predictive Parsing

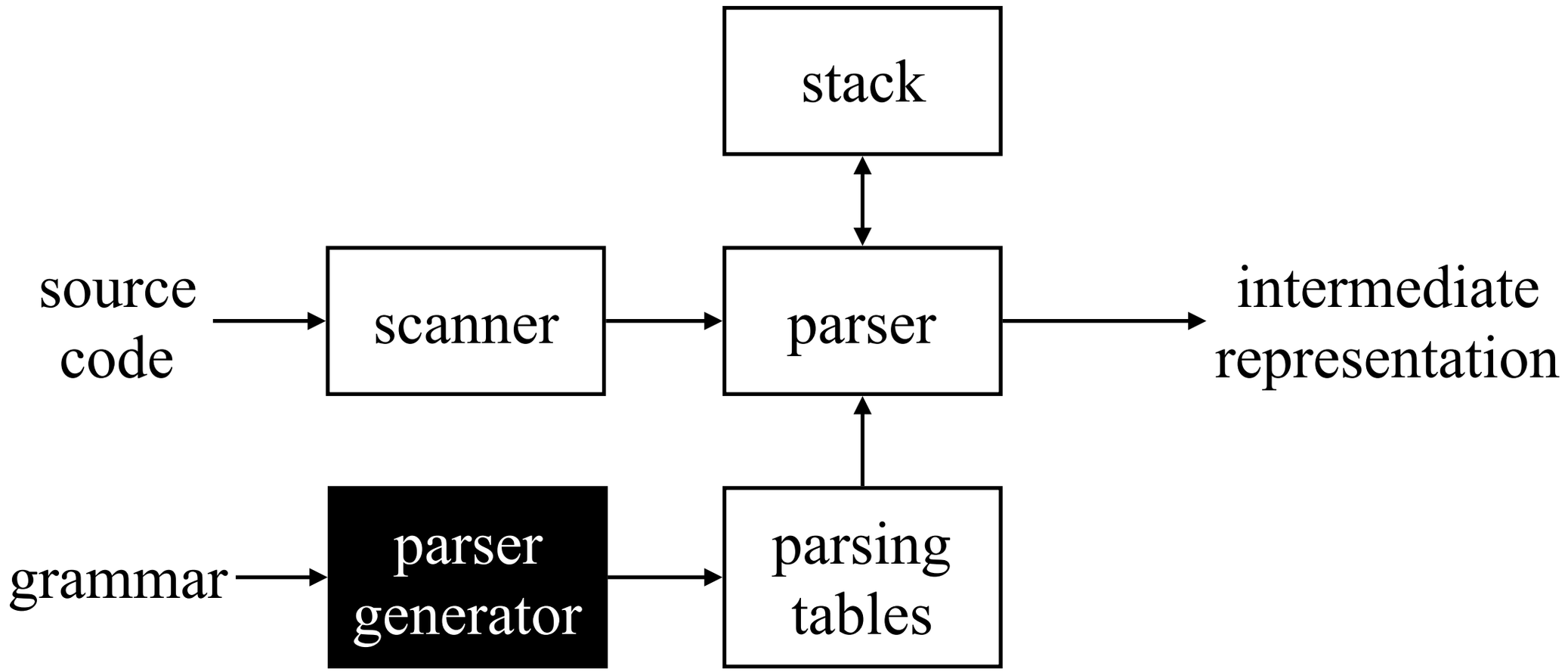
Now, a predictive parser looks like:



Rather than writing code, we build tables.

Predictive Parsing

Now, a predictive parser looks like:



Rather than writing code, we build tables.
Building tables can be automated!

Recursive Descent Parsing

Recursive descent parser for LL(1)

- Each **non-terminal** has an associated parsing procedure that can recognize any sequence of tokens generated by that **non-terminal**
- There is a main routine to initialize all globals (e.g:the *token variable* in previous code example) and call the start symbol. On return, check whether token==EOF, and whether errors occurred.
- Within a parsing procedure, both **non-terminals** and **terminals** can be matched:
  - ➡ Non-terminal A: call procedure for A
  - ➡ Token t: compare t with current input token;
    if matched, **consume input**, otherwise, ERROR
- Parsing procedure may contain code that performs some useful "computations" (*syntax directed translation)*

Recursive Descent Parsing (pseudo code)

	a	b	EOF	other
S	S ::= aSb	S ::= ε	S ::= ε	error

```
main:  {
        token := next_token( );
        if (S( ) and token == EOF) print “accept” else print “error”;
}
```

Recursive Descent Parsing (pseudo code)

	a	b	EOF	other
S	S ::= aSb	S ::= ε	S ::= ε	error

```
bool S( ):  {
        switch token {
                case a:  token := next_token( );
                         call S( );
                         if (token == b)  {
                              token := next_token( );
                              return true;
                         }
                         else
                              return false;
                         break;
                case b:
                case EOF:  return true;
                           break;
                default: return false;
        }
}
```

Predictive Parsing

So far:

- Introduced **FIRST**, **FOLLOW**, and **FIRST+** sets
- Introduced **LL(1)** condition:
  - A grammar G can be parsed predictively with one symbol of lookahead if for all pairs of productions A ::= α and A ::= β that satisfy:
    $FIRST^{+}(A ::= \alpha) \cap FIRST^{+}(A ::= \beta) = \varnothing$
- Introduced a recursive descent parser for an **LL(1)** grammar

We did not cover:

- An algorithm to construct ***FIRST*** sets.
- An algorithm to construct ***FOLLOW*** sets.

FIRST and FOLLOW Sets

FIRST(α):

For some α ∈ (T ∪ NT ∪ EOF ∪ ε)*, define **FIRST** (α) as the set of tokens that appear as the first symbol in some string that derives from α.

That is, **x** ∈ FIRST(α) iff α ⇒* **x**γ for some γ

FIRST set is defined over the strings of grammar symbols
(T ∪ NT ∪ EOF ∪ ε)*

T: terminals NT: non-terminals

FIRST and FOLLOW Sets

FOLLOW(A):

For A $\in$ **NT** , define **FOLLOW**(A) as the set of tokens that can occur immediately after A in a valid sentential form.

FOLLOW set is defined over the set of non-terminal symbols, **NT**.

First Set Construction

Build FIRST(X) for all grammar symbols X:

- For each X as a terminal, then FIRST(X) is {X}
- If X ::= ε, then ε ∈ FIRST(X)
- For each X as a non-terminal, initialize FIRST(X) to ∅

Iterate until no more terminals or ε can be added to any FIRST(X):

For each rule in the grammar of the form $X ::= Y_1 Y_2 \ldots Y_k$

add a to FIRST(X) if $a \in FIRST(Y_1)$

add a to FIRST(X) if $a \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$ for all $1 \leq j \leq i\text{-}1$ and $i \geq 2$

add ε to FIRST(X) if $\varepsilon \in FIRST(Y_i)$ for all $1 \leq i \leq k$

End iterate

Filling in the Details: Computing *FIRST* sets

```
for each x ∈ ( T ∪ EOF ∪ ε)
    FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← ∅

while (FIRST sets are still changing) do
    for each p ∈ P, of the form X → Y1Y2…Yk do
    temp ← FIRST(Y1) - { ε }
        i ← 1
        while ( i ≤ k-1 and ε ∈ FIRST(Yi) )
            temp ← temp ∪ (FIRST(Yi+1) - { ε })
            i ← i + 1
        end // while loop
        if i == k and ε ∈ FIRST(Yk)
        then temp ← temp ∪ { ε }
        FIRST(X) ← FIRST(X) ∪ temp
    end // if - then
  end // for loop
end // while loop
```

Initially, set ***FIRST*** for each terminal symbol, EOF and ε

Filling in the Details: Computing *FIRST* sets

for each $x \in (T \cup EOF \cup \varepsilon)$
    FIRST$(x) \leftarrow \{x\}$
for each A ∈ **NT**, ***FIRST***(A) ← ∅

while (***FIRST*** sets are still changing) do
    for each p ∈ P, of the form $X \rightarrow Y_1 Y_2 \ldots Y_k$ do
    temp ← ***FIRST***(Y_1) - { ε }
        i ← 1
    while (i ≤ k-1 and ε ∈ ***FIRST***(Y_i))
        temp ← temp ∪ (***FIRST***(Y_{i+1}) - { ε })
        i ← i + 1
    end // while loop
    if i == k and ε ∈ ***FIRST***(Y_k)
        then temp ← temp ∪ { ε }
    FIRST(X) ← ***FIRST***(X) ∪ temp
    end // if - then
    end // for loop
end // while loop

ε complicates matters

If ***FIRST***(Y_1) contains ε, then we need to add ***FIRST***(Y_2) to rhs, and …

If the entire rhs can go to ε, then we add ε to ***FIRST***(lhs)

Computing *FIRST* sets

```
for each x ∈ ( T ∪ EOF ∪ ε)
    FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← ∅

while (FIRST sets are still changing) do
    for each p ∈ P, of the form X → Y1Y2…Yk do
    temp ← FIRST(Y1) - { ε }
        i ← 1
        while ( i ≤ k-1 and ε ∈ FIRST(Yi) )
            temp ← temp ∪ (FIRST(Yi+1) - { ε })
            i ← i + 1
        end // while loop
        if i == k and ε ∈ FIRST(Yk)
            then temp ← temp ∪ { ε }
        FIRST(X) ← FIRST(X) ∪ temp
    end // if - then
  end // for loop
end // while loop
```

Outer loop is monotone increasing for ***FIRST*** sets ⇒ | T ∪ NT ∪ EOF ∪ ε | is bounded, so it terminates

Example

Consider the SheepNoise grammar and its ***FIRST*** sets

Goal	::= SheepNoise
SheepNoise	::= SheepNoise **baa** |
	baa

Clearly, ***FIRST***(x) = {baa}, $\forall$ x $\in$ (**T** $\cup$ **NT**)

Symbol	***FIRST*** Set
Goal	**baa**
SheepNoise	**baa**
baa	**baa**

Computing *FIRST* sets

```
for each x ∈ ( T ∪ EOF ∪ ε)
    FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← ∅
```

```
while (FIRST sets are still changing) do
    for each p ∈ P, of the form X → Y1Y2…Yk do
    temp ← FIRST(Y1) - { ε }
        i ← 1
        while ( i ≤ k-1 and ε ∈ FIRST(Yi) )
            temp ← temp ∪ (FIRST(Yi+1) - { ε })
            i ← i + 1
        end // while loop
        if i == k and ε ∈ FIRST(Yk)
            then temp ← temp ∪ { ε }
        FIRST(X) ← FIRST(X) ∪ temp
    end // if - then
end // for loop
end // while loop
```

Initialization assigns each ***FIRST*** set a value

Symbol	***FIRST*** Set
Goal	∅
SheepNoise	∅
baa	{baa}

Computing *FIRST* sets

```
for each x ∈ ( T ∪ EOF ∪ ε)
    FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← ∅

while (FIRST sets are still changing) do
    for each p ∈ P, of the form X → Y1Y2…Yk do
    temp ← FIRST(Y1) - { ε }
        i ← 1
        while ( i ≤ k-1 and ε ∈ FIRST(Yi) )
            temp ← temp ∪ (FIRST(Yi+1) - { ε })
            i ← i + 1
        end // while loop
        if i == k and ε ∈ FIRST(Yk)
            then temp ← temp ∪ { ε }
        FIRST(X) ← FIRST(X) ∪ temp
    end // if - then
  end // for loop
end // while loop
```

Production 3

SheepNoise ::= **baa**

(1) sets temp to ***FIRST***{baa}
(2) copies temp into
    FIRST(SheepNoise)

```
Goal ::= SheepNoise
SheepNoise ::= SheepNoise baa
             | baa
```

Symbol	*FIRST* Set
Goal	∅
SheepNoise	{baa}
baa	{baa}

Computing *FIRST* sets

for each $x \in (T \cup EOF \cup \varepsilon)$
    FIRST$(x) \leftarrow \{x\}$
for each A ∈ **NT**, ***FIRST***(A) ← ∅

while (***FIRST*** sets are still changing) do
    for each p ∈ P, of the form $X \rightarrow Y_1 Y_2 \ldots Y_k$ do
    temp ← ***FIRST***(Y_1) - { ε }
        i ← 1
        while (i ≤ k-1 and ε ∈ ***FIRST***(Y_i))
            temp ← temp ∪ (***FIRST***(Y_{i+1}) - { ε })
            i ← i + 1
        end // while loop
        if i == k and ε ∈ ***FIRST***(Y_k)
            then temp ← temp ∪ { ε }
        FIRST(X) ← ***FIRST***(X) ∪ temp
    end // if - then
  end // for loop
end // while loop

Production 1

(1) sets temp to ***FIRST***
    {SheepNoise}
(2) copies temp into
    FIRST(Goal)

Goal ::= SheepNoise
SheepNoise ::= SheepNoise **baa**
| **baa**

Symbol	***FIRST*** Set
Goal	{baa}
SheepNoise	{baa}
baa	{baa}

Next Lecture

Things to do:

- Read Scott, Chapter 2.1 - 2.3.3; ALSU 2.4