RICE

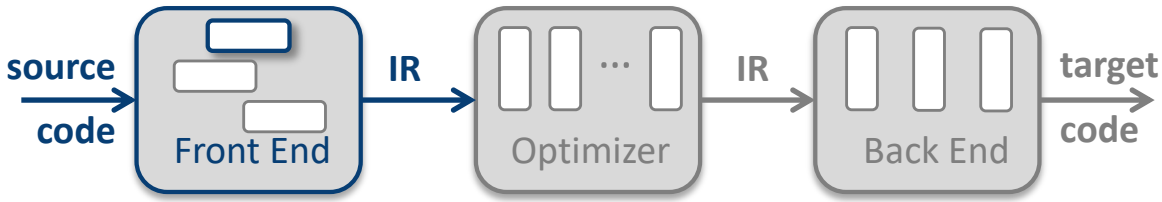
COMP 412
FALL 2018

Midterm Exam: Thursday October 18, 7PM Herzstein Amphitheater

Updated slides posted for lectures 10 & 11

Syntax Analysis, IV

Comp 412



Chapter 3 in EaC2e

Lab 2 Schedule

N.B.: Code checks 1 & 2 are milestones, not exhaustive tests. Many students still find bugs after code check 2, to say nothing of tuning for effectiveness & efficiency.

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
9	10	11	12	13	14 **Lab 2 Specs available**	15
16	17	18 **Tutorial 5 PM McMurtrt**	19	20	21 **Deadline: Code Check 1**	22
23	24 **Tutorial 5 PM McMurtry**	25 **Dan Grove Talk (Dart Group)**	26	27	28	29
30	1 **Deadline: Code Check 2**	2	3	4	5	6
7	8 **Deadline: Lab 2 Code**	9	10	12 **Deadline: Lab Report**	12	13

Advice: Pay attention to the timing / scaling tests

Focus on connecting to Lab 1 IR, and Rename

Allocate correctly

Improve performance & allocation

Review

Last Parsing Lecture (*Last Wednesday*)

- Introduced **FIRST, FOLLOW,** and **FIRST⁺** sets
- Introduced the **LL(1)** condition

  A grammar G can be parsed predictively with one symbol of lookahead if for all pairs of productions A →β and A →γ that have the same lhs A:

  $$\textbf{FIRST}^{+}(A \rightarrow \beta) \cap \textbf{FIRST}^{+}(A \rightarrow \gamma) = \emptyset$$

- Observed that predictively parsable, or **LL(1)** grammars
- Showed how to construct a recursive-descent parser for an **LL(1)** grammar

We did not discuss

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

FIRST and **FOLLOW** Sets

FIRST(α)

For some $\alpha \in (T \cup NT \cup \textbf{EOF} \cup \varepsilon)^*$, define **FIRST**($\alpha$) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in \textbf{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FIRST is defined over strings of grammar symbols: $(T \cup NT \cup \textbf{EOF} \cup \varepsilon)^*$

FOLLOW(*A*)

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

FOLLOW(*S*) = {**EOF**}, where *S* is the start symbol

FOLLOW is defined over the set of nonterminal symbols, **NT**

To build **FOLLOW** sets, we need **FIRST** sets …

EOF ≅ end of file

FIRST and **FOLLOW** Sets

FIRST(α)

For some $\alpha \in (T \cup NT \cup \textbf{EOF} \cup \varepsilon)^*$, define **FIRST**($\alpha$) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in \textbf{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FIRST is defined over strings of grammar symbols: $(T \cup NT \cup \textbf{EOF} \cup \varepsilon)^*$

FOLLOW(*A*)

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

FOLLOW(*S*) = {**EOF**}, where *S* is the start symbol

FOLLOW is defined over the set of nonterminal symbols, **NT**

To build **FOLLOW** sets, we need **FIRST** sets …

EOF ≅ end of file

Conceptual Sketch: Computing **FIRST** Sets

for each $x \in (T \cup \textbf{EOF} \cup \varepsilon)$
    $FIRST(x) \leftarrow \{x\}$

for each $A \in NT$, $FIRST(A) \leftarrow \emptyset$

while (*FIRST sets are still changing*) *do*
  for each $p \in P$, *of the form* $A \rightarrow B_1B_2\ldots B_k$ *do*
    $rhs \leftarrow FIRST(B_1) - \{\varepsilon\}$
    Some details go here to handle ε *productions*
    $FIRST(A) \leftarrow FIRST(A) \cup rhs$
    end // for loop
  end // while loop

for loop** iterates over all the **NTs

To begin, we will ignore ε productions

- Initialize **FIRST** set for each terminal **EOF**, ε, and nonterminal
- Then, loop through the productions and add the **FIRST** set for the leading symbol on the *rhs* to the **FIRST** set of the *lhs* (a nonterminal)
- Because *rhs* can start with a nonterminal, iterate to a fixed point

Conceptual Sketch: Computing **FIRST** Sets

for each $x \in (T \cup \textbf{EOF} \cup \varepsilon)$
    $FIRST(x) \leftarrow \{x\}$

for each $A \in NT$, $FIRST(A) \leftarrow \emptyset$

while (*FIRST sets are still changing*) *do*
  for each $p \in P$, *of the form* $A \rightarrow B_1B_2\ldots B_k$ *do*
    $rhs \leftarrow FIRST(B_1) - \{\varepsilon\}$
    Some details go here to handle ε *productions*
    FIRST(A) $\leftarrow$ ***FIRST(A)*** $\cup$ ***rhs***
    end // for loop
  end // while loop

for loop *iterates over all the* ***NTs***

Loop nest is *monotone increasing* for FIRST sets

- Outer loop is bounded by the number of grammar symbols:

  $| T \cup NT \cup \textbf{EOF} \cup \varepsilon |$

  which is finite
- A single iteration of the while loop is bounded by the size of the grammar
- *For loop* is bounded by the $|productions|$

Filling in the Details: Computing **FIRST** Sets

```
for each x ∈ (T ∪ EOF ∪ ε)
      FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← Ø
while (FIRST sets are still changing) do
   for each p ∈ P, of the form A→ B1B2…Bk do
      rhs ← FIRST(B1) – {ε}
      for i ← 1 to k–1 by 1 while ε ∈ FIRST(Bi) do
            rhs ← rhs ∪ ( FIRST(Bi+1) – { ε } )
            end      // for loop
      if  i = k and ε ∈ FIRST(Bk )
         then rhs ← rhs ∪ { ε}
      FIRST(A) ← FIRST(A) ∪ rhs
      end   // for loop
   end      // while loop
```

See also, Fig. 3.7, EaC2e, p. 104

ε complicates matters

If **FIRST**(B_1) contains ε, then we need to add **FIRST**(B_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) ← Ø

while (FIRST sets are still changing) do
  for each p ∈ P, of the form A→ B1B2…Bk do
    rhs ← FIRST(B1) – {ε}
    for i ← 1 to k–1 by 1 while ε ∈ FIRST(Bi) do
      rhs ← rhs ∪ ( FIRST(Bi+1) – { ε } )
      end     // for loop
    if i = k and ε ∈ FIRST(Bk )
      then rhs ← rhs ∪ { ε }
    FIRST(A) ← FIRST(A) ∪ rhs
    end  // for loop
  end    // while loop
```

See also, Fig. 3.7, EaC2e, p. 104

Loop nest is *monotone increasing* for FIRST sets

- Handling ε-productions adds a loop that iterates over the productions
- Adds a multiplier of the sum of the length of the productions
- Slower, but still finite
- Still terminates

Remember, we pay this cost at **build time**

Example

Consider the *SheepNoise* grammar & its FIRST sets

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

Left-recursive SheepNoise Grammar

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

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

Computing **FIRST** Sets

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

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

while (*FIRST sets are still changing*) *do*
  for each $p \in P$, *of the form* $A \rightarrow B_1B_2 \ldots B_k$ *do*
    $rhs \leftarrow FIRST(B_1) - \{\varepsilon\}$
    for $i \leftarrow 1$ *to* $k-1$ *by 1 while* $\varepsilon \in FIRST(B_i)$ *do*
      $rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\varepsilon\})$
      end // *for loop*
    if $i = k$ *and* $\varepsilon \in FIRST(B_k)$
      then $rhs \leftarrow rhs \cup \{\varepsilon\}$
    $FIRST(A) \leftarrow FIRST(A) \cup rhs$
    end // *for loop*
  end // *while loop*

See also, Fig. 3.7, EaC2e, p. 104

Initialization assigns each **FIRST** set a value

Symbol	FIRST Set
Goal	∅
SheepNoise	∅
baa	{ baa }

Computing **FIRST** Sets

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

for each $x \in (T \cup \textbf{EOF} \cup \varepsilon)$
    $FIRST(x) \leftarrow \{x\}$

for each $A \in NT$, $FIRST(A) \leftarrow \emptyset$

while (*FIRST sets are still changing*) *do*
  for each $p \in P$, *of the form* $A \rightarrow B_1B_2\ldots B_k$ *do*
    $rhs \leftarrow FIRST(B_1) - \{\varepsilon\}$
    for $i \leftarrow 1$ *to* $k-1$ *by 1 while* $\varepsilon \in FIRST(B_i)$ *do*
      $rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\varepsilon\})$
      end // *for loop*
    if $i = k$ *and* $\varepsilon \in FIRST(B_k)$
      then $rhs \leftarrow rhs \cup \{\varepsilon\}$
    $FIRST(A) \leftarrow FIRST(A) \cup rhs$
    end // *for loop*
  end // *while loop*

See also, Fig. 3.7, EaC2e, p. 104

Production 2

(1) sets *rhs* to **FIRST**{ baa } &

(2) copies *rhs* into **FIRST**(*SheepNoise*)

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

Computing **FIRST** Sets

0	*Goal*	→	*SheepNoise*
1	*SheepNoise*	→	*SheepNoise* baa
2		|	baa

for each $x \in (T \cup \textbf{EOF} \cup \varepsilon)$
    $FIRST(x) \leftarrow \{x\}$

for each $A \in NT$, $FIRST(A) \leftarrow \emptyset$

while (*FIRST sets are still changing*) *do*
  for each $p \in P$, *of the form* $A \rightarrow B_1B_2\ldots B_k$ *do*
    $rhs \leftarrow FIRST(B_1) - \{\varepsilon\}$
    for $i \leftarrow 1$ *to* $k-1$ *by 1 while* $\varepsilon \in FIRST(B_i)$ *do*
      $rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\varepsilon\})$
      end // *for loop*
    if $i = k$ *and* $\varepsilon \in FIRST(B_k)$
      then $rhs \leftarrow rhs \cup \{\varepsilon\}$
    $FIRST(A) \leftarrow FIRST(A) \cup rhs$
    end // *for loop*
  end // *while loop*

See also, Fig. 3.7, EaC2e, p. 104

Production 0

(1) sets *rhs* to **FIRST**(*Sheepnoise*) &

(2) copies *rhs* into **FIRST**(*Goal*)

… and one more iteration to recognize that the **FIRST** *sets have stopped changing*

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

An Example

Consider the simple parentheses grammar

0	*Goal*	→	*List*
1	*List*	→	*Pair List*
2		|	ε
3	*Pair*	→	LP *List* RP

where LP *is* (*and* RP *is*)

Symbol	FIRST Initial
Goal	∅
List	∅
Pair	∅
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simple parentheses grammar

0	*Goal*	→	*List*
1	*List*	→	*Pair List*
2		|	ε
3	*Pair*	→	LP *List* RP

where LP *is* (*and* RP *is*)

- Iteration 1 adds LP to **FIRST**(Pair) and LP, ε to **FIRST**(*List*) & **FIRST**(*Goal*)
  → *If we take them in order 3, 2, 1, 0*
- Algorithm reaches fixed point†

Symbol	FIRST Sets		
	Initial	1st	2nd
Goal	∅	LP, ε	LP, ε
List	∅	LP, ε	LP, ε
Pair	∅	LP	LP
LP	LP	LP	LP
RP	RP	RP	RP
EOF	EOF	EOF	EOF

† In the adversarial order (0, 1, 2, 3), propagating {LP, ε} through would require an iteration to reach *List* and an iteration to reach *Goal*, plus an iteration to realize that they stopped changing.

FIRST and **FOLLOW** Sets

FIRST(α)

For some $\alpha \in (T \cup NT \cup \textbf{EOF} \cup \varepsilon)^*$, define **FIRST**($\alpha$) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in \textbf{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FIRST is defined over strings of grammar symbols: $(T \cup NT \cup \textbf{EOF} \cup \varepsilon)^*$

FOLLOW(*A*)

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

FOLLOW(*S*) = {**EOF**}, where *S* is the start symbol

FOLLOW is defined over the set of nonterminal symbols, **NT**

To build **FOLLOW** sets, we need **FIRST** sets …

EOF ≅ end of file

Computing **FOLLOW** Sets

```
for each A ∈ NT
    FOLLOW(A) ← Ø

FOLLOW(S) ← { EOF }

while (FOLLOW sets are still changing)
    for each p ∈ P, of the form A → B1B2 … Bk
        TRAILER ← FOLLOW(A)
        for i ← k down to 1
            if Bi ∈ NT then                          // domain check
                FOLLOW(Bi) ← FOLLOW(Bi) ∪ TRAILER
                if ε ∈ FIRST(Bi)                     // add right context
                  then TRAILER ← TRAILER ∪ ( FIRST(Bi ) – { ε } )
                  else TRAILER ← FIRST(Bi)           // no ε => truncate the right context
            else TRAILER ← {Bi }                     // Bi ∈ T  =>  only 1 symbol
```

Don't add ε

Figure 3.8, page 106, EaC2e

Computing **FOLLOW** Sets

This algorithm has a completely different feel than computing FIRST sets

For a production $A \rightarrow B_1 B_2 \ldots B_k$:

- It works its way backward through the production: B_k, B_{k-1}, … B_1
- It builds the **FOLLOW** sets for the *rhs* symbols, B_1, B_2, … B_k, not A
- In the absence of ε, **FOLLOW**(B_i) is just **FIRST**(B_{i+1})
  - *As always, ε makes the algorithm more complex*

To handle ε, the algorithm keeps track of the *first word* in the trailing right context as it works its way back through the *rhs*: B_k, B_{k-1}, … B_1

- It uses **FOLLOW**(A) to initialize the *Trailer* for B_k
  - That use is the only mention of **FOLLOW**(A) in the algorithm
- *Trailer* approximates the **FIRST**$^+$ set for the trailing right context

An Example

Consider, again, the simple parentheses grammar

0	*Goal*	→	*List*
1	*List*	→	*Pair List*
2		|	ε
3	*Pair*	→	*LP List RP*

Symbol	FOLLOW Sets	
	Initial	1st
Goal	**EOF**	**EOF**
List	∅	**EOF**, RP
Pair	∅	**EOF**, LP

Initial Values:

- *Goal, List,* and *Pair* are set to ∅
- Goal is then set to { **EOF** }

An Example

Consider, again, the simple parentheses grammar

0	*Goal*	→	*List*
1	*List*	→	*Pair List*
2		|	ε
3	*Pair*	→	*LP List RP*

Symbol	FOLLOW Sets	
	Initial	1st
Goal	**EOF**	**EOF**
List	∅	**EOF**, RP
Pair	∅	**EOF**, LP

Iteration 1:

- Production 0 adds **EOF** to FOLLOW(*List*)
- Production 1 adds LP to FOLLOW(*Pair*)
  - → from **FIRST**(*List*)
- Production 2 does nothing
- Production 3 adds RP to FOLLOW(*List*)
  - → from **FIRST**(*RP*)

Symbol	FIRST
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	**EOF**

An Example

Consider, again, the simple parentheses grammar

0	*Goal*	→	*List*
1	*List*	→	*Pair List*
2		|	ε
3	*Pair*	→	*LP List RP*

Symbol	FOLLOW Sets		
	Initial	1st	2nd
Goal	EOF	EOF	EOF
List	∅	EOF, RP	EOF, RP
Pair	∅	EOF, LP	EOF, LP, RP

Iteration 2:

- Production 0 adds nothing new
- Production 1 adds RP to FOLLOW(*Pair*)
  - → from FOLLOW(*List*), ε ∈ FIRST(*List*)
- Production 2 does nothing
- Production 3 adds nothing new

Iteration 3 produces the same result ⇒ reached a fixed point

Classic Expression Grammar

Right-recursive version

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	identifier

FIRST$^+$(A→β) is identical to FIRST(β) except for productions 4 and 8

FIRST$^+$(Expr'→ ε) is {ε,), eof}

FIRST$^+$(Term'→ ε) is {ε,+,-,), eof}

Symbol	FIRST	FOLLOW
num	num	Ø
id	id	Ø
+	+	Ø
-	-	Ø
*	*	Ø
/	/	Ø
(	(	Ø
)	)	Ø
eof	eof	Ø
ε	ε	Ø
Goal	(,id,num	eof
Expr	(,id,num	), eof
Expr'	+, -, ε	), eof
Term	(,id,num	+, -,), eof
Term'	*, /, ε	+, -,), eof
Factor	(,id,num	+, -,* ,/,), eof

*

Classic Expression Grammar

Right-recursive version

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	identifier

Prod'n	FIRST$^+$
0	(,id,num
1	(,id,num
2	+
3	-
4	ε,), eof
5	(,id,num
6	*
7	/
8	ε,+,-,), eof
9	(
10	number
11	identifier

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

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

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

looking for number, identifier, or (, found token instead, or failed to find *Expr* or) after (

```
Factor( )
  if (token = number) then
     token ← next_token( );
     return true;
  else if (token = identifier) then
     token ← next_token( );
     return true;
  else if (token = lparen)
     token ← next_token( );
     if (Expr( ) = true & token = rparen) then
        token ← next_token( );
        return true;
  // fall out of if statement
  report syntax error;
     return false;
```

EPrime, Term, & TPrime follow the same basic lines (Figure 3.10, EaC2e)

Review

Recursive Descent

Page 111 in EaC2e sketches a recursive descent parser for the RR CEG.

- One routine per **NT**
- Check each **RHS** by checking each symbol
- Includes ε-productions

```
Main()
  /* Goal → Expr */
  word ← NextWord();
  if (Expr())
    then if (word = eof)
      then report success;
      else Fail();

Fail()
  report syntax error;
  attempt error recovery or exit;

Expr()
  /* Expr → Term Expr' */
  if (Term())
    then return EPrime();
    else Fail();

EPrime()
  /* Expr' → + Term Expr' */
  /* Expr' → - Term Expr' */
  if (word = + or word = -)
    then begin;
      word ← NextWord();
      if (Term())
        then return EPrime();
        else Fail();
    end;
  else if (word = ) or word = eof)
    /* Expr' → ε */
    then return true;
    else Fail();

Term()
  /* Term → Factor Term' */
  if (Factor())
    then return TPrime();
    else Fail();

TPrime()
  /* Term' → × Factor Term' */
  /* Term' → ÷ Factor Term' */
  if (word = × or word = ÷)
    then begin;
      word ← NextWord();
      if (Factor())
        then return TPrime();
        else Fail();
    end;
  else if (word = + or word = - or
          word = ) or word = eof)
    /* Term' → ε */
    then return true;
    else Fail();

Factor()
  /* Factor → ( Expr ) */
  if (word = ( ) then begin;
    word ← NextWord();
    if (not Expr())
      then Fail();
    if (word ≠ ) )
      then Fail();
    word ← NextWord();
    return true;
  end;
  /* Factor → num */
  /* Factor → name */
  else if (word = num or
            word = name)
    then begin;
      word ← NextWord();
      return true;
    end;
  else Fail();
```

Implementing a Recursive Descent Parser

A nest of if-then else statements may be slow

- A good case statement would be an improvement† ***Python?***
  - See EaC2e, § 7.8.3
  - Encode with computation rather than repeated branches
- Order the cases by expected frequency, to drop average cost

† a good case statement can be hard to find

Implementing a Recursive Descent Parser

A nest of if-then else statements may be slow

- A good case statement would be an improvement† ***Python?***
  - See EaC2e, § 7.8.3
  - Encode with computation rather than repeated branches
- Order the cases by expected frequency, to drop average cost

What about encoding the decisions in a table?

- Replace if then else or case statement with an address computation
- Branches are slow and disruptive
- Interpret the table with a skeleton parser, as we did in scanning

† a good case statement can be hard to find

Building Table-Driven Top-down Parsers

Strategy

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

Example

- The non-terminal *Factor* has 3 expansions
  - (*Expr*) or Identifier or Number
- Table might look like:

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	identifier

Terminal Symbols

	EOF	+	-	*	/	(	)	id.	num.
Factor	—	—	—	—	—	9	—	11	10

Non-terminal Symbols

Cannot expand *Factor* into an operator ⇒ *error*

Expand *Factor* by rule 10 with input "number"

Building Top-down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*

LL(1) Table for the Expression Grammar

	EOF	+	-	*	/	(	)	id.	num.
Goal	—	—	—	—	—	0	—	0	0
Expr	—	—	—	—	—	1	—	1	1
Expr'	4	2	3	—	—	—	4	—	—
Term	—	—	—	—	—	5	—	5	5
Term'	8	8	8	6	7	—	8	—	—
Factor	—	—	—	—	—	9	—	11	10

Row we built earlier

Figure 3.11(b), page 112, EaC2e

Building Top-down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an interpreter for the table (*skeleton parser*)

LL(1) Skeleton Parser

```
word ← NextWord()          // Initial conditions, including
push EOF onto Stack        // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack

loop forever
  if TOS = EOF and word = EOF then
    break & report success   // exit on success

  else if TOS is a terminal then
    if TOS matches word then
      pop Stack              // recognized TOS
      word ← NextWord()
    else report error looking for TOS  // error exit

  else                       // TOS is a non-terminal
    if TABLE[TOS,word] is A → B1B2…Bk then
      pop Stack              // get rid of A
      push Bk, Bk-1, …, B1   // in that order
    else break & report error expanding TOS

  TOS ← top of Stack
```

Building Top-down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need a table-driven interpreter for the table
- 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$^+(X \rightarrow \beta)$
2. entry is *error* if rule 1 does not define

If any entry has more than one rule, G is not **LL**(1)

Incrementally tests the **LL(1)** criterion on each ***NT***.

An efficient way to determine if a grammar is **LL(1)**

This algorithm is the LL(1) table construction algorithm

In Lab 2, you would have built a recursive descent parser for a modified form of **BNF** and build **LL(1)** tables for the grammars that are **LL(1)**. (A good weekend project)

Recap of Top-down Parsing

- Top-down parsers build syntax tree from root to leaves
- Left-recursion causes non-termination in top-down parsers
  - Transformation to eliminate left recursion
  - Transformation to eliminate common prefixes in right recursion
- **FIRST**, **FIRST**$^{+}$, & **FOLLOW** sets + **LL(1)** condition
  - **LL(1)** uses left-to-right scan of the input, leftmost derivation of the sentence, and 1 word lookahead
  - **LL(1)** condition means grammar works for predictive parsing
- Given an **LL(1)** grammar, we can
  - Build a recursive descent parser
  - Build a table-driven **LL(1)** parser
- **LL(1)** parser doesn't build the parse tree
  - Keeps lower fringe of partially complete tree on the stack