

# Top-down Parsing Recursive Descent & LL(1)

## Comp 412

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.  
 Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.  
 Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Question that baffled me last class:

### Left Recursion Creates Left Associative Trees

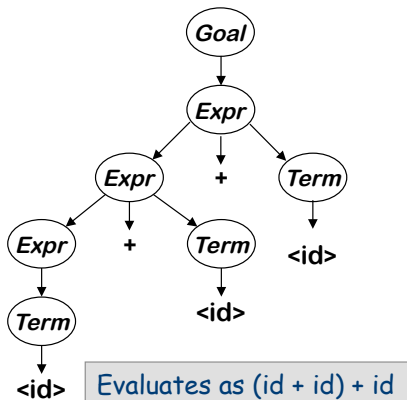


A Trivial Expression Grammar

- 0 *Goal* → *Expr*
- 1 *Expr* → *Expr* + *Term*
- 2       | *Term*
- 3 *Term* → *id*

 Derivation of *id + id + id*

Rule	Sentential Form
—	<i>Goal</i>
0	<i>Expr</i>
1	<i>Expr</i> + <i>Term</i>
3	<i>Expr</i> + <id>
1	<i>Expr</i> + <i>Term</i> + <id>
3	<i>Expr</i> + <id> + <id>
2	<i>Term</i> + <id> + <id>
3	<id> + <id> + <id>





## Roadmap (Where are we?)

*We set out to study parsing*

- Specifying syntax
  - Context-free grammars ✓
- Top-down parsers
  - Algorithm & its problem with left recursion ✓
  - Ambiguity ✓
  - Left-recursion removal ✓
- Predictive top-down parsing
  - The LL(1) condition ✓
  - Simple recursive descent parsers **today**
  - First and Follow sets **today**
  - Table-driven LL(1) parsers **today**



## Predictive Parsing

### Basic idea

*Given  $A \rightarrow \alpha \mid \beta$ , the parser should be able to choose between  $\alpha$  &  $\beta$*

### FIRST sets

For some *rhs*  $\alpha \in \mathcal{G}$ , define **FIRST( $\alpha$ )** as the set of tokens that appear as the first symbol in some string that derives from  $\alpha$

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

### The LL(1) Property

If  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

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

This is almost correct  
See the next slide



## Predictive Parsing

What about  $\epsilon$ -productions?

⇒ They complicate the definition of LL(1)

If  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  and  $\epsilon \in \text{FIRST}(\alpha)$ , then we need to ensure that  $\text{FIRST}(\beta)$  is disjoint from  $\text{FOLLOW}(A)$ , too, where

**FOLLOW(A)** = the set of terminal symbols that can immediately follow A in a sentential form

Define **FIRST<sup>+</sup>(A→α)** as

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$ , if  $\epsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$ , otherwise

Then, a grammar is LL(1) iff  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  implies

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$$



## 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

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset \text{ if } i \neq j$$

```

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

```

Grammars with the LL(1) property are called predictive grammars because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the LL(1) property are called predictive parsers.

One kind of predictive parser is the recursive descent parser.

Of course, there is more detail to "find a β<sub>i</sub>" (p. 103 in EaC, 1e)



# Recursive Descent Parsing

Recall the expression grammar, after transformation

0	<i>Goal</i>	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Term Expr'</i>
2	<i>Expr'</i>	→	+ <i>Term Expr'</i>
3			- <i>Term Expr'</i>
4			ε
5	<i>Term</i>	→	<i>Factor Term'</i>
6	<i>Term'</i>	→	* <i>Factor Term'</i>
7			/ <i>Factor Term'</i>
8			ε
9	<i>Factor</i>	→	<u>number</u>
10			<u>id</u>
11			( <i>Expr</i> )

This produces a parser with six mutually recursive routines:

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

Each recognizes one *NT* or *T*

The term descent refers to the direction in which the parse tree is built.



# 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 3.7, EaC 1e)



## Recursive Descent (Summary)

---

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

Can we automate this process?



## FIRST and FOLLOW Sets

---

### FIRST( $\alpha$ )

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

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

### 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

$\text{FOLLOW}(S) = \{\text{EOF}\}$ , where S is the start symbol

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



## Computing FIRST Sets

```

for each  $x \in T$ ,  $FIRST(x) \leftarrow \{x\}$ 
for each  $A \in NT$ ,  $FIRST(A) \leftarrow \emptyset$ 
while (FIRST sets are still changing) do
  for each  $p \in P$ , of the form  $A \rightarrow \beta$  do
    if  $\beta$  is  $B_1 B_2 \dots B_k$  then begin:
       $rhs \leftarrow FIRST(B_1) - \{\epsilon\}$ 
      for  $i \leftarrow 1$  to  $k-1$  by 1 while  $\epsilon \in FIRST(B_i)$  do
         $rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\epsilon\})$ 
      end // for loop
    end // if-then
    if  $i = k$  and  $\epsilon \in FIRST(B_k)$ 
      then  $rhs \leftarrow rhs \cup \{\epsilon\}$ 
     $FIRST(A) \leftarrow FIRST(A) \cup rhs$ 
  end // for loop
end // while loop

```

Outer loop is monotone increasing for FIRST sets

→  $|T \cup NT \cup \epsilon|$  is bounded, so it terminates

Inner loop is bounded by the length of the productions in the grammar

For *SheepNoise*:

$FIRST(Goal) = \{ \underline{b}aa \}$

$FIRST(SN) = \{ \underline{b}aa \}$

$FIRST(\underline{b}aa) = \{ \underline{b}aa \}$



## Computing FIRST Sets

```

for each  $x \in T$ ,  $FIRST(x) \leftarrow \{x\}$ 
for each  $A \in NT$ ,  $FIRST(A) \leftarrow \emptyset$ 
while (FIRST sets are still changing) do
  for each  $p \in P$ , of the form  $A \rightarrow \beta$  do
    if  $\beta$  is  $B_1 B_2 \dots B_k$  then begin:
       $rhs \leftarrow FIRST(B_1) - \{\epsilon\}$ 
      for  $i \leftarrow 1$  to  $k-1$  by 1 while  $\epsilon \in FIRST(B_i)$  do
         $rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\epsilon\})$ 
      end // for loop
    end // if-then
    if  $i = k$  and  $\epsilon \in FIRST(B_k)$ 
      then  $rhs \leftarrow rhs \cup \{\epsilon\}$ 
     $FIRST(A) \leftarrow FIRST(A) \cup rhs$ 
  end // for loop
end // while loop

```

Outer loop is monotone increasing for FIRST sets

→  $|T \cup NT \cup \epsilon|$  is bounded, so it terminates

For *SN* → baa

Inner loop is bounded by the length of the productions in the grammar

For *SheepNoise*:

$FIRST(Goal) = \{ \underline{b}aa \}$

$FIRST(SN) = \{ \underline{b}aa \}$

$FIRST(\underline{b}aa) = \{ \underline{b}aa \}$

## Computing FIRST Sets



```

for each  $x \in T$ ,  $FIRST(x) \leftarrow \{x\}$ 
for each  $A \in NT$ ,  $FIRST(A) \leftarrow \emptyset$ 
while (FIRST sets are still changing) do
  for each  $p \in P$ , of the form  $A \rightarrow \beta$  do
    if  $\beta$  is  $B_1 B_2 \dots B_k$  then begin:
       $rhs \leftarrow FIRST(B_1) - \{\epsilon\}$ 
      for  $i \leftarrow 1$  to  $k-1$  by 1 while  $\epsilon \in FIRST(B_i)$  do
         $rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\epsilon\})$ 
      end // for loop
    end // if-then
    if  $i = k$  and  $\epsilon \in FIRST(B_k)$ 
      then  $rhs \leftarrow rhs \cup \{\epsilon\}$ 
     $FIRST(A) \leftarrow FIRST(A) \cup rhs$ 
  end // for loop
end // while loop
  
```

Outer loop is monotone increasing for FIRST sets

$\rightarrow |T \cup NT \cup \epsilon|$  is  
For Goal  $\rightarrow SN$   $\leq S$

Inner loop is bounded by the length of the productions in the grammar

For *SheepNoise*:

```

FIRST(Goal) = { baa }
FIRST(SN) = { baa }
FIRST(baa) = { baa }
  
```

## Computing FOLLOW Sets



```

for each  $A \in NT$ ,  $FOLLOW(A) \leftarrow \emptyset$ 
 $FOLLOW(S) \leftarrow \{EOF\}$ 
while (FOLLOW sets are still changing)
  for each  $p \in P$ , of the form  $A \rightarrow B_1 B_2 \dots B_k$ 
     $TRAILER \leftarrow FOLLOW(A)$ 
    for  $i \leftarrow k$  down to 1
      if  $B_i \in NT$  then // domain check
         $FOLLOW(B_i) \leftarrow FOLLOW(B_i) \cup TRAILER$ 
        if  $\epsilon \in FIRST(B_i)$  // add right context
          then  $TRAILER \leftarrow TRAILER \cup (FIRST(B_i) - \{\epsilon\})$ 
        else  $TRAILER \leftarrow FIRST(B_i)$  // no  $\epsilon \Rightarrow$  no right context
      else  $TRAILER \leftarrow \{B_i\}$  //  $B_i \in T \Rightarrow$  only 1 symbol
  
```



# Classic Expression Grammar

0	<i>Goal</i>	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Term Expr'</i>
2	<i>Expr'</i>	→	+ <i>Term Expr'</i>
3			- <i>Term Expr'</i>
4			ε
5	<i>Term</i>	→	<i>Factor Term'</i>
6	<i>Term'</i>	→	* <i>Factor Term'</i>
7			/ <i>Factor Term'</i>
8			ε
9	<i>Factor</i>	→	<u>number</u>
10			<u>id</u>
11			( <i>Expr</i> )

FIRST\*( $A \rightarrow \beta$ ) is identical to FIRST( $\beta$ )  
except for production 4 and 8

FIRST\*( $Expr' \rightarrow \epsilon$ ) is {ε, eof}

FIRST\*( $Term' \rightarrow \epsilon$ ) is {ε, +, -, }, eof}

Symbol	FIRST	FOLLOW
<u>num</u>	<u>num</u>	∅
<u>id</u>	<u>id</u>	∅
+	+	∅
-	-	∅
*	*	∅
/	/	∅
(	(	∅
)	)	∅
<u>eof</u>	<u>eof</u>	∅
ε	ε	∅
<i>Goal</i>	(, <u>id</u> , <u>num</u>	eof
<i>Expr</i>	(, <u>id</u> , <u>num</u>	), eof
<i>Expr'</i>	+, -, ε	), eof
<i>Term</i>	(, <u>id</u> , <u>num</u>	+, -, ), eof
<i>Term'</i>	*, /, ε	+, -, ), eof
<i>Factor</i>	(, <u>id</u> , <u>num</u>	+, -, *, /, ), eof



# Classic Expression Grammar

0	<i>Goal</i>	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Term Expr'</i>
2	<i>Expr'</i>	→	+ <i>Term Expr'</i>
3			- <i>Term Expr'</i>
4			ε
5	<i>Term</i>	→	<i>Factor Term'</i>
6	<i>Term'</i>	→	* <i>Factor Term'</i>
7			/ <i>Factor Term'</i>
8			ε
9	<i>Factor</i>	→	<u>number</u>
10			<u>id</u>
11			( <i>Expr</i> )

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



# Building Top-down Parsers for LL(1) Grammars

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
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

I don't know of a system that does this ...



# Building Top-down Parsers

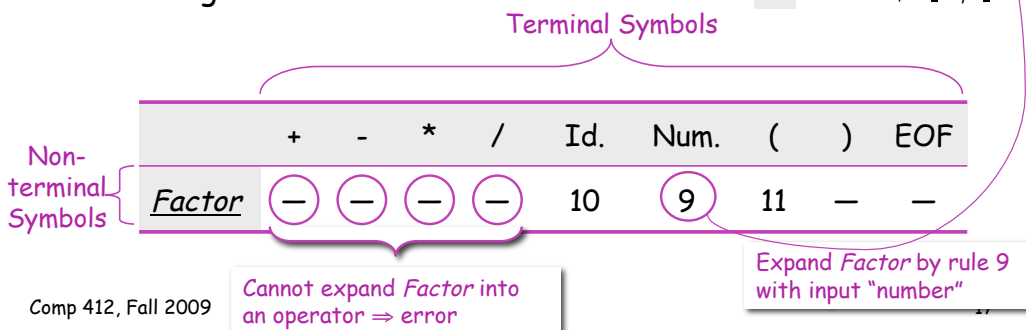
Strategy

- 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	<i>Goal</i>	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Term Expr'</i>
2	<i>Expr'</i>	→	+ <i>Term Expr'</i>
3			- <i>Term Expr'</i>
4			ε
5	<i>Term</i>	→	<i>Factor Term'</i>
6	<i>Term'</i>	→	* <i>Factor Term'</i>
7			/ <i>Factor Term'</i>
8			ε
9	<i>Factor</i>	→	<u>number</u>
10			<u>id</u>
11			[ <i>Expr</i> ]





## 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  $LL(1)$

We call this algorithm the  $LL(1)$  table construction algorithm

## Grammar and Sets $LL(1)$ Construction



0	<i>Goal</i>	$\rightarrow$	<i>Expr</i>
1	<i>Expr</i>	$\rightarrow$	<i>Term Expr'</i>
2	<i>Expr'</i>	$\rightarrow$	$+$ <i>Term Expr'</i>
3			$-$ <i>Term Expr'</i>
4			$\epsilon$
5	<i>Term</i>	$\rightarrow$	<i>Factor Term'</i>
6	<i>Term'</i>	$\rightarrow$	$*$ <i>Factor Term'</i>
7			$/$ <i>Factor Term'</i>
8			$\epsilon$
9	<i>Factor</i>	$\rightarrow$	<u>number</u>
10			<u>id</u>
11			$($ <i>Expr</i> $)$

Right-recursive variant of the classic expression grammar

Prod'n	FIRST+
0	$(, id, num$
1	$(, id, num$
2	$+$
3	$-$
4	$\epsilon, ), eof$
5	$(, id, num$
6	$*$
7	$/$
8	$\epsilon, +, -, ), eof$
9	<u>number</u>
10	<u>id</u>
11	$($

Augmented FIRST sets for the grammar

# LL(1) Expression Parsing Table



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

