RICE

COMP 412
FALL 2010

Top-down Parsing
Recursive Descent & LL(1)

Comp 412

The lecture is self consistent, but the order of the three productions for Factor in the expression grammar is different than in 2e

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 α & β

FIRST sets

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

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

The LL(1) Property

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

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

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 ε-productions?

$\Rightarrow$ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), too, where

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

Define FIRST$^+$($A \rightarrow \alpha$) as

- FIRST(α) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

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

FIRST$^+$($A \rightarrow \alpha$) $\cap$ FIRST$^+$($A \rightarrow \beta$) = $\varnothing$

The intuition is straightforward. If A expands with 2 right hand sides, those rhs' must have distinct first symbols.

We only need FIRST$^+$ sets because of ε-productions.

What If My Grammar Is Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) gramar?

- In general, the answer is no
- In some cases, however, the answer is yes

Assume a grammar G with productions $A \rightarrow \alpha\,\beta_1$ and $A \rightarrow \alpha\,\beta_2$

- If α derives anything other than ε, then

$$\text{FIRST}^+(A \rightarrow \alpha\,\beta_1) \cap \text{FIRST}^+(A \rightarrow \alpha\,\beta_2) \neq \varnothing$$

- And the grammar is not LL(1)

If we pull the common prefix, α, into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha\; A',\; A' \rightarrow \beta_1 \text{ and } A' \rightarrow \beta_2$$

Now, if $\text{FIRST}^+(A' \rightarrow \beta_1) \cap \text{FIRST}^+(A' \rightarrow \beta_2) = \varnothing$, G may be LL(1)

What If My Grammar Is Not LL(1) ?

Left Factoring

For each nonterminal A
  find the longest prefix α common to 2 or more alternatives for A
  if $\alpha \neq \varepsilon$ then
    replace all of the productions
      $A \rightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \alpha\, \beta_3 \mid \ldots \mid \alpha\, \beta_n \mid \gamma$
      with
      $A \rightarrow \alpha\, A' \mid \gamma$
      $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$
Repeat until no nonterminal has alternative rhs' with a common prefix

This transformation makes some grammars into LL(1) grammars

There are languages for which no LL(1) grammar exists

Left Factoring Example

Consider a simple right-recursive expression grammar

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term + Expr*
2		|	*Term - Expr*
3		|	*Term*
4	*Term*	$\rightarrow$	*Factor * Term*
5		|	*Factor / Term*
6		|	*Factor*
7	*Factor*	$\rightarrow$	<u>number</u>
8		|	<u>id</u>

To choose between 1, 2, & 3, an LL(1) parser must look past the <u>number</u> or <u>id</u> to see the operator.

FIRST+(1) = FIRST+(2) = FIRST+(3)

and

FIRST+(4) = FIRST+(5) = FIRST+(6)

Let's left factor this grammar.

Left Factoring Example

After Left Factoring, we have

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

Clearly,

FIRST$^+$(2), FIRST$^+$(3), & FIRST$^+$(4)

are disjoint, as are

FIRST$^+$(6), FIRST$^+$(7), & FIRST$^+$(8)

The grammar now has the LL(1) property

This transformation makes some grammars into LL(1) grammars.

There are languages for which no LL(1) grammar exists.

FIRST and FOLLOW Sets

FIRST(α)

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

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

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

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 ∈ P, of the form A→β do
        if β is B1B2…Bk then begin;
            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
        end         // if-then
        if  i = k and ε ∈ FIRST(Bk )
           then rhs ← rhs ∪ { ε }
        FIRST(A) ← FIRST(A) ∪ rhs
        end  // for loop
    end       // while loop
```

Outer loop is monotone increasing for FIRST sets

$\rightarrow$ | $T \cup NT \cup \varepsilon$ | is bounded, so it terminates

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 FIRST Sets

```
for each x ∈ T, 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→β do
        if β is B_1B_2…B_k then begin;
            rhs ← FIRST(B_1) – { ε }
            for i ← 1 to k–1 by 1 while ε ∈ FIRST(B_i) do
                rhs ← rhs ∪ ( FIRST(B_{i+1}) – { ε } )
                end  // for loop
        end          // if-then
        if  i = k and ε ∈ FIRST(B_k )
            then rhs ← rhs ∪ { ε }
        FIRST(A) ← FIRST(A) ∪ rhs
        end  // for loop
    end      // while loop
```

Outer loop is monotone increasing for FIRST sets

→ $|\, T \cup NT \cup \varepsilon \,|$ is … nates

For $SN \rightarrow$ baa

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 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 β *is* $B_1B_2 \ldots B_k$ *then begin;*
            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
        end   // if-then
        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

Outer loop is monotone increasing for FIRST sets

$\rightarrow$ | T $\cup$ NT $\cup$ ε | is

For *Goal* $\rightarrow$ *SN*   es

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

For *SheepNoise:*

FIRST(*Goal*) = { <u>baa</u> }
FIRST(*SN*) = { <u>baa</u> }
FIRST(<u>baa</u>) = { <u>baa</u> }

Computing FOLLOW Sets

$for\ each\ A \in NT,\ FOLLOW(A) \leftarrow Ø$

$FOLLOW(S) \leftarrow \{EOF\}$

```
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 => no right context
            else TRAILER ← {Bi}                     // Bi ∈ T => only 1 symbol
```

Classic Expression Grammar

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

FIRST$^+$($A\rightarrow\beta$) is identical to FIRST(β) except for productiond 4 and 8

FIRST$^+$(Expr'$\rightarrow$ ε) is {ε,), eof}

FIRST$^+$(Term'$\rightarrow$ 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

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

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

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	(	)	EOF
Factor	—	—	—	—	10	9	11	—	—

Cannot expand *Factor* into an operator ⇒ error

Expand *Factor* by rule 9 with input "number"

Building Top-down Parsers

Building the complete table

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

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	—	—

Row we built earlier

Table differs from 2e because order of productions in Factor differs.

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 for the LL(1) Construction

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*	→	number
10		|	id
11		|	(*Expr*)

Right-recursive variant of the classic expression grammar

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

Augmented FIRST sets for the grammar