Lecture 15a

COP3402 FALL 2015 – DR. MATTHEW GERBER – 11/9/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- More about FOLLOW
- Predictive Parsing Tables

Review: The FIRST Set

$FIRST(X) = \{ \mathbf{t} \mid X ::=^* \mathbf{t}\ s \text{ for any } s\} \cup \{ \varepsilon \mid X ::=^* \varepsilon \}$

Consider FIRST(A B C).

- If A is a terminal, FIRST(A B C) = {A}
- If $\varepsilon \notin$ FIRST(A), FIRST(A B C) = FIRST(A)
- If $\varepsilon \in$ FIRST(A), FIRST(A B C) = FIRST(A) – $\{\varepsilon\} \cup$ FIRST(B C)

In turn, for FIRST(B C):

- If B is a terminal, FIRST(B C) = {B}
- If $\varepsilon \notin$ FIRST(B), FIRST(B C) = FIRST(B)
- If $\varepsilon \in$ FIRST(B), FIRST(B C) = FIRST(B) – $\{\varepsilon\} \cup$ FIRST(C)

We only include ε if ε is in FIRST(A), FIRST(B) and FIRST(C).

Review: The FOLLOW Set

We need one more set before we can really get moving.

- Let A be a nonterminal.
- FOLLOW(A) is the set of terminals that can *immediately follow* A.
- More formally, with **t** a terminal, S the start symbol, α and β arbitrary sequences, and "::=*" meaning "can produce through some sequence of rewriting rules":

$$\text{FOLLOW}(A) = \{\ t \mid S ::=^* \alpha\ A\ \mathbf{t}\ \beta \text{ for some } \alpha, \beta\ \}$$

FOLLOW Set Rules

Let \$ represent the end of the sequence. (In program terms, the end of the file.)

- FOLLOW(S), where S is the start symbol, will always contain \$. (Think about it.)

Now let α and β be any strings of terminals or nonterminals.

If there is a production A ::= α B, *then*

- FOLLOW(A) $\subset$ FOLLOW(B)

If there is a production A ::= α B β, *then*

- (FIRST(β) – {ε}) $\subset$ FOLLOW(B)
- If β is nullable, FOLLOW(A) $\subset$ FOLLOW(B)

FOLLOW Set: Example 1

- $\$ \in$ FOLLOW(S).
- Let α and β be any strings, then:
- If there is a production A ::= α B, then
  - FOLLOW(A) $\subset$ FOLLOW(B)
- If there is a production A ::= α B β, then
  - (FIRST(β) – {ε}) $\subset$ FOLLOW(B)
  - If β is nullable, FOLLOW(A) $\subset$ FOLLOW(B)

S ::= Z
Z ::= d
Z ::= X Y Z
Y ::= ε
Y ::= c
X ::= Y
X ::= a

	Nullable	FIRST	FOLLOW
X	Yes	{ a, c, ε }	{ a, c, d }
Y	Yes	{ c, ε }	{ a, c, d }
Z	No	{ a, c, d }	{ $ }

FOLLOW Set: Example 2

- \$ $\in$ FOLLOW(S). 0
- Let α and β be any strings, then:
- If there is a production A ::= α B, then
  - FOLLOW(A) $\subset$ FOLLOW(B) 1
- If there is a production A ::= α B β, then
  - (FIRST(β) – {ε}) $\subset$ FOLLOW(B) 2
  - If β is nullable, FOLLOW(A) $\subset$ FOLLOW(B) 3

S ::= E

E ::= T E’	T ::= F T’	F ::= id
E’ ::= “+” T E’	T’ ::= “*” F T’	F ::= “(“ E “)”
E’ ::= ε	T’ ::= ε	

	Nullable	FIRST	FOLLOW	Why?
E	No	{ id , “(“ }	{ “)”, \$ }	<0>, F ::= “(“ E “)” <2>
E’	Yes	{ “+”, ε }	{ “)”, \$ }	E ::= T E’ <1>
T	No	{ id , “(“ }	{ “)”, “+”, \$ }	E ::= T E’ <2, 3>
T’	Yes	{ “*”, ε }	{ “)”, “+”, \$ }	T ::= F T’ <1>
F	No	{ id , “(“ }	{ “)”, “*”, “+”, \$ }	T ::= F T’ <2, 3>

Predictive Parsing

So we know how to find the FIRST and FOLLOW sets, as well as how to find nullable symbols. What we can do with this information is construct a *predictive parsing table*.

- In a parse table, rows are labeled for nonterminals, and columns are labeled for terminals.

Consider a grammar, and a parsing table **m**.

For every production A ::= α in the grammar:

- ∀ **t** ∈ FIRST(α), add A ::= α to **m**[A, **t**]
- *If* α *is nullable*, then ∀ **t** ∈ FOLLOW(A), add A ::= α to **m**[A, **t**]

Predictive Parsing: Example 1

Consider the grammar:

S ::= Z		
Z ::= d	Y ::= ε	X ::= Y
Z ::= X Y Z	Y ::= c	X ::= a

Recall these results:

	Nullable	FIRST	FOLLOW
X	Yes	{ a, c, **ε** }	{ a, c, d }
Y	Yes	{ c, **ε** }	{ a, c, d }
Z	No	{ a, c, d }	{ $ }

For every production A ::= α in the grammar:

- ∀ **t** ∈ FIRST(α), add A ::= α to **m**[A, **t**]
- *If α is nullable*, then
  ∀ **t** ∈ FOLLOW(A), add A ::= α to **m**[A, **t**]

	a	**c**	**d**
X	X ::= a X ::= Y	X ::= Y	X ::= Y
Y	Y ::= ε	Y ::= c Y ::= ε	Y ::= ε
Z	Z ::= XYZ	Z ::= XYZ	Z ::= d Z ::= XYZ

Predictive Parsing: Example 2

S ::= E
E ::= T E'
E' ::= "+" T E'
E' ::= ε
T ::= F T'
T' ::= "*" F T'
T' ::= ε
F ::= id
F ::= "(" E ")"

	Nullable	FIRST	FOLLOW
E	No	{ id , "(" }	{ ")", $ }
E'	Yes	{ "+", ε }	{ ")", $ }
T	No	{ id , "(" }	{ ")", "+", $ }
T'	Yes	{ "*", ε }	{ ")", "+", $ }
F	No	{ id , "(" }	{ ")", "*", "+", $ }

	+	*	id	(	)	$
E			E ::= T E'	E ::= T E'		
E'	E' ::= "+" TE				E' ::= ε	E' ::= ε
T			T ::= F T'	T ::= F T'		
T'	T' ::= ε	T' ::= "*" F T'			T' ::= ε	T' ::= ε
F			F ::= id	F ::="(" E ")"		

For every production A ::= α in the grammar:

- ∀ **t** ∈ FIRST(α), add A ::= α to **m**[A, **t**]
- *If α is nullable*, then
  ∀ **t** ∈ FOLLOW(A), add A ::= α to **m**[A, **t**]

Predictive Parsing: Enlarged Table

	+	*	id	(	)	$
E			E ::= T E’	E ::= T E’		
E’	E’ ::= “+” TE				E’ ::= ε	E’ ::= ε
T			T ::= F T’	T ::= F T’		
T’	T’ ::= ε	T’ ::= “*” F T’			T’ ::= ε	T’ ::= ε
F			F ::= id	F ::=“(“ E “)”		

Predictive Parsing, Method 1

	+	*	id	(	)	$
T’	T’ ::= ε	T’ ::= “*” F T’			T’ ::= ε	T’ ::= ε

```
void Tprime (void) {
    switch (token) {
        case PLUS:   break;
        case TIMES:  accept(TIMES); F(); Tprime(); break;
        case RPAREN: break;
        default:     fail();
    }
}
```

Predictive Parsing, Method 2 – LL(1)

We've discussed *left-recursion*, and getting rid of it, before.

We've also discussed *left factoring* to avoid having more than one production for a nonterminal that starts with the same symbol.

Grammars with both of these problems solved are called *LL(1) grammars*.

- The first L stands for *left-to-right parsing* – the string can be read left to right.
- The second L stands for *leftmost derivation.*
- The 1 stands for *one-symbol lookahead* – only one symbol worth of lookahead is required.

For grammars that meet these requirements, we can parse them without worrying about recursive function calls at all.

Table-Driven Parsing: Overview

	+	*	id	(	)	$
E			E ::= T E'	E ::= T E'		
E'	E' ::= "+" TE				E' ::= ε	E' ::= ε
T			T ::= F T'	T ::= F T'		
T'	T' ::= ε	T' ::= "*" F T'			T' ::= ε	T' ::= ε
F			F ::= id	F ::="(" E ")"		

STACK **INPUT**

$E **id + id * id$**

Top of stack symbol (X) **Current input symbol (`cis`)**

Let's take:

- The parse table for an LL(1) grammar
- An input buffer
  - Initialize to the string to be parsed, followed by $
- A stack
  - Initialize by pushing $ then the start symbol – in this case, E
- A couple of variables
  - `cis` will be the *current input symbol*
  - X will be the symbol at the top of the stack

Table-Driven Parsing: Algorithm

	+	*	id	(	)	$
E			E ::= T E’	E ::= T E’		
E’	E’ ::= “+” TE				E’ ::= ε	E’ ::= ε
T			T ::= F T’	T ::= F T’		
T’	T’ ::= ε	T’ ::= “*” F T’			T’ ::= ε	T’ ::= ε
F			F ::= id	F ::=“(“ E “)”		

STACK **INPUT**

$E **id + id * id$**

Top of stack symbol (X) **Current input symbol (cis)**

Push $ onto the stack
Push start symbol E onto the stack
Repeat { /* …while the stack isn’t empty */
    If (X is nonterminal) {
        pop the stack;
        if (M[X, cis] is empty) fail();
        else push the RHS of M[X, cis] in reverse order;
    } elseif (X = cis) {
        pop the stack;
        advance cis;
    } else fail();
    Let X point to the top of the stack.
}
until (X = $)
accept()

Table-Driven Parsing: Execution

Stack	Input	Production
$E	id + id * id$	
$E'T	id + id * id$	E ::= TE'
$E'T'F	id + id * id$	T ::= FT'
$E'T'id	id + id * id$	F ::= id
$E'T'	+ id * id$	match id
$E'	+ id * id$	T' ::= ε
$E'T+	+ id * id$	E' ::= +TE'
$E'T	id * id$	match +
$E'T'F	id * id$	T ::= FT'
$E'T'id	id * id$	F ::= id
$E'T'	* id$	match id
$E'T'F*	* id$	T' ::= *FT'
$E'T'F	id$	match *
$E'T'id	id$	F ::= id
$E'T'	$	match id
$E'	$	T' ::= ε
$	$	E' ::= ε

```
Push $ onto the stack
Push start symbol E onto the stack
Repeat { /* …while the stack isn't empty */
    If (X is nonterminal) {
        pop the stack;
        if (M[X, cis] is empty) fail();
        else push the RHS of M[X, cis] in reverse order;
    } elseif (X = cis) {
        pop the stack;
        advance cis;
    } else fail();
    Let X point to the top of the stack.
}
until (X = $)
accept()
```

Next Time: Assemblers