Lecture 15

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/26/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- LL(1) Parsing
- The FIRST Set
- Nullable Symbols
- The FOLLOW Set
- Predictive Parsing Tables

The FIRST Set

A recursive descent parser chooses the correct production by looking ahead in the stream by a fixed number of symbols.

- Typically, we want this number to be one.

We can systematize this a little more using a technique called *First-and-Follow* parsing.

- *There are pure algorithmic approaches to finding the FIRST and FOLLOW sets, and they're easy to find, but they're more trouble than they're worth for most of what you'll do in this class.*

Let's start with the FIRST set.

- Let X be a string of terminals and non-terminals.
- FIRST(X) is the set of *terminals* that can *begin* sequences derivable from X.
- More formally, with **t** a terminal, ε the empty string, and "::=*" meaning "can produce through some sequence of rewriting rules":

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

Computing the FIRST Set

FIRST(X) = { **t** | X ::=* **t** s for any s} ∪ { ε | X ::=* ε }

Consider FIRST(A B C).

- If A is a terminal, FIRST(A B C) = {A}
- If ε ∉ FIRST(A), FIRST(A B C) = FIRST(A)
- If ε ∈ FIRST(A), FIRST(A B C) = FIRST(A) – {ε} ∪ FIRST(B C)

In turn, for FIRST(B C):

- If B is a terminal, FIRST(B C) = {B}
- If ε ∉ FIRST(B), FIRST(B C) = FIRST(B)
- If ε ∈ FIRST(B), FIRST(B C) = FIRST(B) – {ε} ∪ FIRST(C)

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

First Set Example 1 and Exercise

S ::= A B C | C b B | B a
A ::= d a | B C
B ::= g | ε
C ::= h | ε

Then…

FIRST(S) = FIRST(A B C) ∪ FIRST(C b B)
∪ FIRST(B a)
FIRST(A) = FIRST(d a) ∪ First(B C) = { d }
∪ FIRST(B C)
FIRST(B) = FIRST(g) ∪ FIRST { ε } = { g, ε }
FIRST(C) = FIRST(h) ∪ FIRST { ε } = { h, ε }

Now we compute:
FIRST(BC) = FIRST(B) – { ε } ∪ { h, ε }
= { g, ε } – { ε } ∪ { h, ε }
= { g, h, ε }
and
FIRST(A) = { d } ∪ { g, h, ε }
= { d, g, h, ε }

Exercise: Compute FIRST(C b B), FIRST(B a), then FIRST(S)

One More Example

First(E + T) = { id, (}

Why? $E + T \rightarrow T + T \rightarrow F + T \rightarrow id + T$

$E + T \rightarrow T + T \rightarrow F + T \rightarrow (E) + T$

E ::= E "+" T | T

T ::= T "*" F | F

F ::= "(" E ")" | id

First(E) = { id, (}

Why? $E \rightarrow T \rightarrow F \rightarrow id$

$E \rightarrow T \rightarrow F \rightarrow (E)$

Nullable Symbols

…are symbols that can produce the empty string. Consider a grammar:

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

Now observe:

X → Y → ε	Y → ε
Z → d	Z → X Y Z

Then…

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

(Note that something is nullable if and only if it can derive ε .)

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 Example

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

Observations:

- FOLLOW(S), where S is the start symbol, will always contain $. (Think about it.)
- If there's a derivation containing A**t**, then **t** is in FOLLOW(A).
- If there's a derivation containing AB, everything in FIRST(B) is in FOLLOW(A), except {ε}.
- If there's a derivation containing AB**t**, *and* B *is nullable*, then **t** is also in FOLLOW(A).
- If there's a derivation containing ABC, *and* B *is nullable*, then everything in FIRST(C) is in FOLLOW(A), except {ε}.

Consider this grammar:

S ::= Z

Z ::= d	Y ::= ε	X ::= Y
Z ::= X Y Z	Y ::= c	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 }	{ $ }

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 Part 1

Now let's look at a more interesting grammar.

S ::= E

E ::= E "+" T	T ::= T "*" F	F ::= id
E ::= T	T ::= F	F ::= "(" E ")"

First let's get rid of left recursion.

S ::= E

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

E' ::= ε   T' ::= ε

Now we can figure out FIRST, FOLLOW and the nullables…

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

…and from there, a parse table.

Predictive Parsing: Example 2 Part 2

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
E	No	{ id , “(“ }	{ “)”, $ }
E’	Yes	{ “+”, ε }	{ “)”, $ }
T	No	{ id , “(“ }	{ “)”, “+”, $ }
T’	Yes	{ “*”, ε }	{ “)”, “+”, $ }
F	No	{ id , “(“ }	{ “)”, “*”, “+”, $ }

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**]

	+	*	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: 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: Writing a Parser

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();
    }
}
```

Recursion, Factoring and Lookahead

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:
Code Generation