CS 241 Lecture 14

Graham Cooper

June 22nd, 2015

Top-Down Parsing

$$S \implies \alpha_1 \implies ... \implies \alpha_n \implies w$$

Invariant: Consumed input + reverse(stack contents) = α_i

Example:
S → AyB
A → ab
A → cd
B → z
B → wx

For simplicity, we use augumented grammars for parsing, ie. invent two new symbols $\vdash, \dashv$, new start symbol S'

1. S' $\rightarrow \vdash S \dashv$
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wx

Stack	Read Input	Unread Input	Action
S'	ϵ	$\vdash abywx \dashv$	Pop S'; push $\dashv, S, \vdash$
$\dashv S \vdash$	ϵ	$\vdash abywx \dashv$	Match $\vdash$
$\dashv S$	$\vdash$	abywx$\dashv$	Pop S; Push B,y,A
$\dashv ByA$	$\vdash$	abywx$\dashv$	PopA; push b,a
$\dashv Byba$	$\vdash$	abywx$\dashv$	Match a
$\dashv$ Byb	$\vdash a$	bywx$\dashv$	Match b
$\dashv$ By	$\vdash ab$	ywx$\dashv$	Match y
$\dashv$ B	$\vdash aby$	wx$\dashv$	Pop B; push x,w
$\dashv$ xw	$\vdash aby$	wx$\dashv$	Match w
$\dashv$ x	$\vdash$abyw	x$\dashv$	Match x
$\dashv$	$\vdash$abywx	$\dashv$	Match $\dashv$
	$\vdash$abywx$\dashv$	ϵ	Accept

When top of stack (TOS) is a terminal pop and match against input.
When TOS is a non-terminal A, popA and push α^R (α reversed), where A $\rightarrow \alpha$ is a grammar rule.
Accept when stack and input are both empty

BUT what if there is >1 production with A on the LHS? How can we know which one to pick?

- Brute force (try all compinations until one works)
- Our solution: Use the next symbol of input (look ahead) to help decide (predictor table)

Construct a predictor table!
- Given a non-terminal on the stack and input symbol, tell us which production to use.

1. S' $\rightarrow \vdash S \dashv$
2. S $\rightarrow$ AyB
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd

5. B $\rightarrow$ z

6. B $\rightarrow$ wx

	$\vdash$	a	b	c	d	w	x	y	z	$\dashv$
S'	1									
S		2		2						
A		3		4						
B						6		5		

Empty cell = parse error

Descriptive: "Parse error at row,col: expecting one of "chars
for which curren top of stack has entries

What if $A \rightarrow$ ad, then you would have multiple states in a single table column

What if a cell contains more than one entry?
THIS DOESN'T WORK...

A grammar is called LL(1) if each cell of the predictor table has at most one entry. IF we have this situation then this table will work.

LL(1) = left-to-right scan of input leftmost derivations produced 1 symbol of look ahead.
That was way to long to be just LL...

Automaticall computing the predictor table

Predict(A, a) gives you the rules that apply when A is on the stack and a is the next input character

Predict(A, a) = {A $\rightarrow \beta$ | $a \in$ First(β) }
First(β) - $\beta \in V^*$ - set of characters that can be the first letter of a derivation starting from β
First(β) = {a | $\beta \implies *a\gamma$ }

So Predict(A, a) = {A $\rightarrow \beta$ | $\beta \implies$ *aγ }

So really:

Predict(A,a) = {A $\rightarrow \beta$ | a $\in$ First(β) } $\cup$ {A $\rightarrow \beta$ | Nullable(β), a $\in$ Follow(A) }
Nullable(β) = true if and only if $\beta \implies *\epsilon$
Follow(A) = {b | S' $\implies$ *$\alpha A b \beta$ }
- Terminal symbols that can come immediately after A in a derivation starting from S'

Computing Nullable

```
initialize Nullable[A] = false for all A
repeat:
-- For each rule B -> B1 ... Bk
-- -- if k = 0 or Nullable[Bi] for all i
-- -- -- Nullable[B] <- True
until nothing changes
```

Computing First

```
initialize First[A] = {} for all A
repeat
-- for each rule B -> B1...Bk
-- -- for i = 1 ... k
-- -- -- if Bi is a terminal a
-- -- -- -- First[B] += {a}
-- -- -- -- break
-- -- -- else
-- -- -- -- First[B] += First[Bi]
-- -- -- -- if not Nullable[Bi] break;
until nothing changes
```

Computing First*(β)

- first set for a string of symbols

```
First*(Y1...,Yn)
-- result <- empty
-- for i = 1 ... n
-- -- if Yi not in Sigma (ie non-terminal)
-- -- -- result += First[Yi]
-- -- -- if not Nullable[Yi] break;
-- -- else (terminal)
-- -- -- result += {Yi}
-- -- -- break
-- return result
```

Computing Follow

```
initialize Follow[A] = {} for all A != S'
repeat
-- for each rule B -> B1 ... Bn
-- -- for i = 1....n-1
-- -- -- if Bi is in N
-- -- -- -- Follow[Bi] += First*(Bi+1 ... Bn)
-- -- -- -- if all Bi+1 ... Bn are nullable (includes i = n)
-- -- -- -- -- Follow[Bi] += Follow[B]
until nothing changes
```