

# CS 241 Final Review Session

Graham Cooper

August 7th, 2015

Summary:

- Top Down Parsing
- Bottom up Parsing
- Error Checking
- Code Generation
- Optimization
- Memory management

## Top Down Parsing

LL(1)

$First(\gamma) = \{b | \gamma \implies *b\beta \text{ for some } \beta\}$

$Follow(A) = \{c | S' \implies \alpha A c \beta \text{ for some } \alpha, \beta\}$

$Predict(A, a) = \{A \implies \gamma | a \in First(\gamma) \text{ or } (\gamma \text{ is nullable and } a \in Follow(A))\}$

## Problem 1

(SUBTRACT 1 from all of the rules)

1.  $S' \rightarrow \vdash S \dashv$
2.  $S \rightarrow aYb$
3.  $S \rightarrow XS$
4.  $y \rightarrow ccZy$
5.  $y \rightarrow \epsilon$
6.  $X \rightarrow dZe$
7.  $Z \rightarrow f$

8.  $Z \rightarrow \epsilon$

$\text{first}(S') = \{\vdash\}$   
 $\text{first}(S) = \{a, d\}$   
 $\text{first}(y) = \{c\}$   
 $\text{first}(X) = \{d\}$   
 $\text{first}(Z) = \{f\}$

Epsilon does not appear in any of the above sets as it is a string containing NO TERMINALS

$\text{follow}(S') = \{\}$   
 $\text{follow}(S) = \{\neg\}$   
 $\text{follow}(y) = \{b\}$   
 $\text{follow}(X) = \{a, d\}$   
 $\text{follow}(Z) = \{c, e, b\}$

	$\vdash$	$\neg$	a	b	c	d	e	f
S'	0							
S			1			2		
y				4	3			
X						5		
Z				7	7		7	6

To go through this, go through the rules and check the first of the left hand side, and put the state in the appropriate box

Then find all nullables and include the follow of the left hand side of the rule.

Since there is no cell with more than 1 item it is LL(1)

PARSE:

Action	TOP Stack BOTTOM	Unread
init	S'	$\vdash dfeaccb \neg$
expand 0	$\vdash S \neg$	$\vdash dfeaccb \neg$
match $\vdash$	S $\neg$	dfeaccb $\neg$
expand 2	XS $\neg$	dfeaccb $\neg$
expand 5	dZeS $\neg$	dfeaccb $\neg$
match d	ZeS $\neg$	feaccb $\neg$

# Bottom Up Parsing

(rules are correct for this one)

$\vdash \text{ababaab} \vdash$

1.  $S' \rightarrow \vdash X \vdash$

2.  $X \rightarrow XbAb$

3.  $X \rightarrow XaBa$

4.  $X \rightarrow \epsilon$

5.  $A \rightarrow An$

6.  $A \rightarrow \epsilon$

7.  $B \rightarrow Bb$

8.  $B \rightarrow \epsilon$

0 $\vdash$ S 3	3 a R 3	5 a S 9	8 a S 2
1 a R 1	3 b R 3	5 b S 1	8 b S 6
1 b R 1	3 $\vdash$ R 3	6 a R 6	9 a R 4
1 $\vdash$ R 1	3 X S 10	6 b R 6	9 b R 4
2 a R 2	4 a R 5	7 a R 7	10 a S 7
2 b R 2	4 b R 5	7 b R 7	10 b S 4
2 $\vdash$ R 2	4 A S 5	7 B S 8	10 $\vdash$ S 11

Action	STACK	Unread Input
init	0	$\vdash \text{ababaab} \vdash$
shift $\vdash$	0 $\vdash$ 3	ababaab $\vdash$
reduce 3,	0 $\vdash$ 3 X 10	ababaab $\vdash$
shift a	0 $\vdash$ 3 X 10 a 7	babaab $\vdash$
reduce 7	0 $\vdash$ 3 X 10 a 7 B 8	babaab $\vdash$
shift b	0 $\vdash$ 3 X 10 a 7 B 8 b 6	abaab $\vdash$
reduce 6	0 $\vdash$ 3 X 10 a 7 B 8	abaab $\vdash$
shift a	0 $\vdash$ 3 X 10 a 7 B 8 a 2	baab $\vdash$
reduce 2	0 $\vdash$ 3 X 10	baab $\vdash$

When do we accept/reject?

- accept on empty stack and empty input - reject on bad input or if only one of stack or input is empty

## Error Checking

```
int wain(int * a, int *b){  
-- int c = 5;  
-- if(c !=== a){  
-- -- *(a + 4) = c;  
-- }  
-- return b;  
}
```

errors:

scanning, semantic or syntax

- `int * b` is a semantic error (type)
- `!=== a` is a scanning error
- Must have an else case (parsing error = syntax)
- `return` has `a` to return an `int` (semantic)

## Code Generation

### Want to add simple for loop

- exactly one assignment in initializer portion
- exactly one test
- exactly one assignment increment portion
- new variables can be made in initializer portion

What do we need to change/add in WLP4 to make this work?

Tokens:

FOR "for"

Grammar:

statement  $\rightarrow$  for ( assign ; test ; assign ) { statements }

assign  $\rightarrow$  lvalue becomes expr

Typing:

WT(assign1) and WT(test) and WT(assign2) and WT(statements)

-----  
for(assign1;test;assign2){statements}

code(stmt  $\rightarrow$  sasn1; test; assign2){statements}=

code(assign1)

forX:

code(test)

for1beq \$3, \$0, skipX

code(stmts)

code(assign2)

beq \$0, \$0, forX

skipX:

## Adding Bitshift Operators

Tokens:

>> RShift

<< LShift

Grammar

bitexpr  $\rightarrow$  expr

bitexpr  $\rightarrow$  bitexpr << expr

bitexpr  $\rightarrow$  bitexpr >> expr

statement  $\rightarrow$  lvalue = bitexpr

CODE

code(bitexpr  $\rightarrow$  bitexpr << expr)=

code(bitexpr)

push(\$3)

code(expr)

pop(\$5)

lis \$2

.word 2

```
topX:
beq $3, $0, endX
mult $5, $2
mflo $5
sub $3, $3, $11
beq $0, $0, topX
endX:
add $3, $5, $0
```

## Optimization

- constant propagation
- constant folding
- dead code elimination
- common subexpression elimination
- register allocation
- strength reduction
- procedure inlining
- tail recursion

## Memory Management

Fragmentation

- External ()
- internal ()

**Explicit**

Binary Buddy System

**Implicit (garbage collection)**

Mark and Sweep

Reference Counting

Copying GC