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)
$First(\gamma) = \{b | \gamma \implies *b\beta for some \beta\}$
$Follow(A) = \{c | S' \implies \alpha Ac\beta$ for some α, β $\}$
$Predict(A, a) = \{A \implies \gamma | a \in First(\gamma)$ or (γ 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. $Z \to \epsilon$

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

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

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

	$\vdash$	$\dashv$	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'	$\vdash dfeaccb \dashv$
expand 0	$\vdash S \dashv$	$\vdash dfeaccb \dashv$
match $\vdash$	S $\dashv$	dfeaccb$\dashv$
expand 2	XS $\dashv$	dfeaccb $\dashv$
expand 5	dZeS $\dashv$	dfeaccb $\dashv$
match d	ZeS $\dashv$	feaccb $\dashv$

Bottom Up Parsing

(rules are correct for this one)
⊢ababaab⊣

1. $S' \to \vdash X \dashv$
2. $X \to XbAb$
3. $X \to XaBa$
4. $X \to \epsilon$
5. $A \to An$
6. $A \to \epsilon$
7. $B \to Bb$
8. $B \to \epsilon$

0 ⊢ 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 ⊣ R 3	6 a R 6	9 a R 4
1 ⊣ 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 ⊣ R 2	4 A S 5	7 B S 8	10 ⊣ S 11

Action	STACK	Unread Input
init	0	⊢ ababaab ⊣
shift ⊢	0 ⊢ 3	ababaab ⊣
reduce 3,	0 ⊢ 3 X 10	ababaab ⊣
shift a	0 ⊢ 3 X 10 a 7	babaab ⊣
reduce 7	0 ⊢ 3 X 10 a 7 B 8	babaab ⊣
shift b	0 ⊢ 3 X 10 a 7 B 8 b 6	abaab ⊣
reduce 6	0 ⊢ 3 X 10 a 7 B 8	abaab ⊣
shift a	0 ⊢ 3 X 10 a 7 B 8 a 2	baab ⊣
reduce 2	0 ⊢ 3 X 10	baab ⊣

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"

Gramar:
statement − > for (assign ; test ; assign) { statements }

assign − > lvalue becomes expr

Typing:
WT(assign1) and WT(test) and WT(assign2) and WT(statements)
----
for(assign1;test;assign2){statements}

code(stmt → sasn1; test; assign2){statements}=
code(assign1)
forX:
code(test)
for1beq $3, $0, skipX
code(stmts)
code(assgn2)
beq $0, $0, forX
skipX:

Adding Bitshift Operators

Tokens:
>> RShift
<< LShift

Grammar
bitexpr → expr
bitexpr → bitexpr << expr
bitexpr → bitexpr >> expr
statement → lvalue = bitexpr

CODE
code(bitexpr → 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 elmination
- common subexpression elimination
- register allocation
- strength reduction
- procedure inlining
- tail recursion