CSE431 – Translation of Computer Languages

- Top Down Parsers

Doug Shook

Top Down Parsers

- Two forms:
  - Recursive Descent
  - Table

- Also known as LL(k) parsers:
  - Read tokens from **L**eft to right
  - Produces a **L**eftmost derivation
  - Uses **k** symbols of lookahead
    - k is typically = 1

Recursive Descent Parsers

- One method for each non-terminal
- Use lookahead to examine the next symbol and decide which production to take
  - What do we use to make this decision?
- Use mutual recursion to descend the tree

Recursive Descent Parsers

```
function P() {
  if (LookaheadIs(“(“)) {
    Match(“(“);
    P();
    Match(“)”);
  } elseif(LookaheadIs(“a”)) {
    Match(“a”);
  } else {
    Error();
  }
}
```

```
P -> ( P )
   | a
```

LL(1) Grammars

- Lookahead is used to predict which production to take next
  - Why would we restrict this to 1?
- Any grammar that can be successfully parsed using this method with lookahead = 1 is known as an LL(1) grammar
  - Not all grammars are LL(1)

LL(1) Grammars

- Consider the following productions:

```
A -> Aa
   |  b
```

- What happens if the predict set tells us to take the first production?

Left Recursion

- The solution is to transform the production to use right recursion instead:

A -> XY
X -> b
Y -> aY
| λ

LL(1) Grammars

- Now consider the following production:

```
A -> bC
   |  bD
```

- What is the problem?

Common Prefix

- The production can be factored so that the common element is in a single production:

```
A -> bX
X -> C
   | D
```

- Does this rule apply to common non-terminal prefixes too?

Non-LL(1) Grammars

- Grammars are not LL(1) when they:
  - Have overlapping predict sets for the same nonterminal
  - Can never advance the input
- There are three cases when the above will occur
  - We just saw two of them
  - What is the third?

Non-LL(1) Grammars

- Consider the Dangling Bracket Language:

$$\{[^i\,]^j \mid i >= j >= 0\}$$

- What's wrong with this grammar?

S -> [S C
  | λ
C ->]
  | λ

Non-LL(1) Grammars

- That one was ambiguous, so let's try this one:

```
S -> [ S
   | T
T -> [ T ]
   | λ
```

- What's wrong with this grammar?

- Unfortunately there is just no way to make it LL(1)
  - What can we do?

Practice

- Write a pseudocode recursive descent parser for the following grammar:

S -> A $
A -> B c
B -> d B
| λ

Practice

- The following grammar is not LL(1). Fix it so that it is LL(1). Make sure to list every operation you perform.

```
S -> A $
A -> a b c D E
   | a b E
D -> D d
   | e
E ->  f
```

Table Driven Parsers

- What are the possible drawbacks of recursive descent parsers?

- Table driven parsers rely on a stack
  - Special TOS() function can read the top of the stack without popping
  - Two cases:
    - Terminal – match and pop
    - Nonterminal – use lookahead and parse table to predict the next production

- What can we use to construct the table?

Table Driven Parsers

1	S	→	A	C	$	
2	C	→	c			
3		|	λ			
4	A	→	a	B	C	d
5		|	B	Q		
6	B	→	b	B		
7		|	λ			
8	Q	→	q			
9		|	λ			

Nonterminal	Lookahead					
	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

LL(1) Properties

- LL(1) grammars:
  - Create leftmost derivations
  - Are unambiguous
- How efficient are these parsers?
  - Run time of a table driven parser?
  - Recursive descent?

Error handling

- How do we know when an error occurs?

- Two approaches
  - Error recovery
  - Error repair

Error Recovery

- Error recovery methods attempt to reset the parser to a valid state
- Common implementation: panic mode
  - What symbols in languages typically denote the end of a statement?
  - Ignore input until one of those symbols is found, then restart
- Drawbacks?

Error Repair

- Several possibilities here:
  - Insert or delete symbols from the already parsed portion
  - Remove the symbol that caused the error
  - Insert values into the unparsed portion
- Which of the above seem most reasonable to you?
  - Which seem problematic?

More Practice

- Revisit the pseudocode you wrote for slide 14, and add code to implement panic mode error handling

- Create a parse table for the following grammar:

```
S -> V $
V -> n
   | ( E )
E -> + V V
   | * W
W -> V W
   | λ
```

- Use your table to parse “(*(+nn))$”

More Practice

- Create a grammar that recognizes valid regular expressions. You only need to handle the primary operators (*, |) and parentheses. For simplicty, use a single terminal as a placeholder for all characters.

- Examples:

  x|x
  x|xx
  (x|x)x

- Show the derivation of x|xx* using your grammar

More Practice

- Is your regex grammar suitable for LL(1) parsing? If not, modify it.
- Create the parse table for your grammar.