GLR

Example

- Consider this very innocent looking language: (x=a terminal):
  $S \rightarrow x\ S\ x \mid x$

LL

- Can we parse this with an LL parser?
  - No!
  - Why not?

LL

- Needs left factoring

Factored

- Suppose we factor:
  S $\rightarrow$ x S'
  S' $\rightarrow$ S x | λ
- OK, we're good, right?

Consider

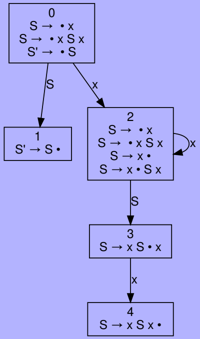
- First:
  - first[S] = {x}
  - first[S’] = {x}
- Follow:
  - follow[S] = {$,x}
  - follow[S’] = {$,x}
- Problem!
  - When we set table row S’, column x: We have a conflict

LR(1)

- Can we parse this with an SLR(1) parser?
- No. Why not?

SLR(1)

- Consider the DFA:



SLR(1)

- The SLR(1) table. Notice the conflict.

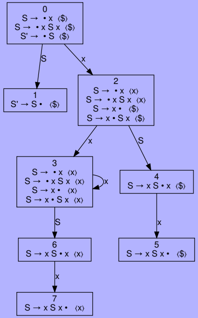
	State	\$	S	x
0	$S \to \bullet\, x$ $S \to \bullet\, x\, S\, x$ $S' \to \bullet\, S$		T,1	S,2
1	$S' \to S\, \bullet$	R,1,S’		
2	$S \to \bullet\, x$ $S \to \bullet\, x\, S\, x$ $S \to x\, \bullet$ $S \to x\, \bullet\, S\, x$	R,1,S	T,3	S,2 R,1,S
3	$S \to x\, S\, \bullet\, x$			S,4
4	$S \to x\, S\, x\, \bullet$	R,3,S		R,3,S

LR(1)

- What if we use an LR(1) parser?
- Won't work. Why not?

Example

- Consider the DFA:



Example

- Consider the parse table:

	State	\$	S	x
0	$S \rightarrow \bullet\ x \prec\$\succ$ $S \rightarrow \bullet\ x\ S\ x \prec\$\succ$ $S' \rightarrow \bullet\ S \prec\$\succ$		T,1	S,2
1	$S' \rightarrow S\ \bullet \prec\$\succ$	R,1,S'		
2	$S \rightarrow \bullet\ x \prec x\succ$ $S \rightarrow \bullet\ x\ S\ x \prec x\succ$ $S \rightarrow x\ \bullet \prec\$\succ$ $S \rightarrow x\ \bullet\ S\ x \prec\$\succ$	R,1,S	T,4	S,3
3	$S \rightarrow \bullet\ x \prec x\succ$ $S \rightarrow \bullet\ x\ S\ x \prec x\succ$ $S \rightarrow x\ \bullet \prec x\succ$ $S \rightarrow x\ \bullet\ S\ x \prec x\succ$		T,6	S,3 R,1,S
4	$S \rightarrow x\ S\ \bullet\ x \prec\$\succ$			S,5
5	$S \rightarrow x\ S\ x\ \bullet \prec\$\succ$	R,3,S		
6	$S \rightarrow x\ S\ \bullet\ x \prec x\succ$			S,7
7	$S \rightarrow x\ S\ x\ \bullet \prec x\succ$			R,3,S

Example

- Suppose we have this input:
  x x x
- What does the parser do?
  - Trace out in class: In case of conflicts, we prefer shifts vs. we prefer reductions
  - We can get successful parse if we prefer reduce to shift

Problem

- A static rule won’t work in all cases
- Consider input:
  x x x x x
- How do we get a successful parse here?

Problem

- We need to determine whether to prefer shift or reduce by considering what's next in the input stream
- If we're about to read from the exact middle of the input:
  - Shift x, then immediately reduce $S \rightarrow x$
  - Then repeatedly:
    - Shift x, reduce $S \rightarrow x\ S\ x$
- If we're before the exact middle of the input, we want to shift x but not reduce it

Problem

- This grammar is not LR(*anything*) for a finite value!
- Need to look at whole input to decide what to do

GLR

- We can parse this with a more powerful kind of parser: A *GLR* parser
  - GLR = General LR
- Idea: We run several parse operations in parallel
- We can adapt this idea to LR(0), SLR(1), LR(1), or LALR(1) tables [we haven't discussed LALR(1)]
  - In practice, SLR(1) or LR(1) seem the most useful
- Can handle *anything* (as long as it's context free)

Scheme

- Input: A parse table
  - LR(0), SLR(1), LR(1), ...
- Each table cell holds a *list* of tuples: One or more of:
  - (S, newState)
  - (R, numPop, transSymbol)
- Parser functions like an ordinary parser...Most of the time

But

- When we see a table cell with multiple possibilities, the parser *forks*
  - Make copies of the parser's internal state
    - I.e., the stack
  - Each copy reflects a different choice from the table cell

But

- If we are at a point in the parse where one of the stacks cannot proceed further, drop it
- If all stacks gone, report syntax error and halt
- If any stack gets to success (reduces to S'): Report success and stop.

Key

- We run the parsers *in parallel*
- Idea: For most practical programming languages, ambiguities will resolve fairly quickly
- So we don't get an exponentially increasing number of stacks
  - Note that with a crafted grammar, that might be a problem
  - We would quickly run out of memory

Data Structure

- Suppose we have a stack data structure with these operations:
  - push(x): Adds x to stack
  - pop(): Returns item
  - top(): Returns item
  - clone(): Makes a copy

Operation

- We need to try all the reductions first, then the shifts
- Why?
  - Remember, we're doing operations in parallel
  - So we need to make sure that all the stacks are ready for shifting simultaneously
  - Shifting affects the input (consumes a token), so once we do it, parser state is permanently changed

Initialization

- Trace out operation of GLR parser with SLR table for the grammar example given previously and input 'x x x'
- Do this in class

Complications

- What happens if we have a grammar with rules like this:
  $S \rightarrow A$
  $A \rightarrow B \mid x$
  $B \rightarrow A \mid y$
- Consider the input "x"
- What's the correct sequence of actions?

Problem

- We could do
  $S \rightarrow A \rightarrow x$
  - Shift x, reduce $A \rightarrow x$, reduce $S \rightarrow A$, reduce $S' \rightarrow S$
- Or:
  $S \rightarrow A \rightarrow B \rightarrow A \rightarrow x$
  - Shift x, reduce $A \rightarrow x$, reduce $B \rightarrow A$, reduce $A \rightarrow B$, reduce $S \rightarrow A$, reduce $S' \rightarrow S$
- Or:
  $S \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow A \rightarrow x$
  - Shift x, reduce $A \rightarrow x$, reduce $B \rightarrow A$, reduce $A \rightarrow B$, reduce $B \rightarrow A$, reduce $A \rightarrow B$, reduce $S \rightarrow A$, reduce $S' \rightarrow S$
- Infinite number of possible parses

Reductions

- We can't just forbid doing multiple reductions in a row
  - In some cases, we might need to reduce repeatedly before we can shift a token
- Ex: Suppose we have a grammar like so:

  $S \to A\ y$

  $A \to B\ y \mid B$

  $B \to C\ y \mid C$

  $C \to x$
- Suppose we've just shifted an x and the next token is a y
- Before we shift the y, we must reduce using rule $B \to C$ and then reduce again using rule $A \to B$ so when we read the y, we are then ready to reduce using rule $S \to A\ y$

Problem

- What if we have a grammar like this:
  $S \rightarrow x\ Y\ x \mid x$
  $X \rightarrow Y$
  $Y \rightarrow Y \mid S \mid X$
- Consider the input “x x x”
  - And this input: x x x
- What happens?

Problem

- No reductions are possible initially
- The parser shifts x
- Then it reduces $S \rightarrow x$
- Then it reduces $Y \rightarrow S$
- And now it can reduce $X \rightarrow Y$
- And then reduce $Y \rightarrow X$
  - This can be repeated indefinitely!

Loop

- We need to tweak our algorithm
- When performing a reduction, if that would result in creating a duplicate of an existing live stack, don't do that reduction
- We'll assume our stack has a key() function
  - Returns some unique identifier for that particular stack's contents
- And we'll keep a set of keys that corresponds to our active stacks
- We drop any stack that has a duplicate key

Problem

- We can still run into a problem: What if we have a grammar like so:
  $S \rightarrow A\ x$
  $A \rightarrow A\ A \mid \lambda$
  - What happens here?
    - Can do infinitely many reductions!
    - Since stack keeps getting items added, we *never* see duplicate stacks!

Solution

- We'll take the easy way out: Forbid λ productions in the grammar
  - Every reduction either leaves the stack the same size or makes it smaller
    - So we cannot continue indefinitely without seeing duplicates
- We could make the reduction code more complex and allow λ productions, but we won't consider that here

Problem

- The algorithm as we've coded it is wasteful of memory
- Many stacks will share substantial part of their contents
  - No reason to duplicate them
  - Slower to copy the data when we fork

Solution

- Use linked lists for stacks
- Assume we have a list node structure:

```
class Node:
    def __init__(self,val,nxt):
        self.value=val  #integer: State number
        self.next=nxt   #next Node
```

List

- Define a stack-as-linked-list:

```
class StackAsList:
    def __init__(self):
        self.top_ = None
    def push(self,stateNumber):
        self.top_ = Node(stateNumber,self.top_)
    def pop(self,num=1):
        for i in range(num):
            self.top_ = self.top_.next
    def top(self):
        return self.top_.value
    def topNode(self):
        return self.top_
    def clone(self):
        copy = StackAsList()
        copy.top_ = self.top_
        return copy
    def key(self):
        return (self.top(),self.topNode().next)
```

Processing

- Our parse operation:
- Input: Parse table + list of tokens
- Parse table: A list of dictionaries. table[i] tells transitions for DFA state i
  - Each dictionary has key of string (terminal or nonterminal) and value (list of actions)
  - Action: A tuple: One of
    - S, state number [shift]
    - T, state number [transition]
    - R, numpop, state number [reduce]

Code

```
def parse(table, tokens):
    stacks = [ StackAsList() ]
    stacks[0].push( 0 ) #start state
    tokenIndex = 0
    while True:
        nextToken = tokens[tokenIndex]
        apply reductions
        apply shifts
        if stacks is empty:
            report failure
        tokenIndex += 1
```

Reductions

- To perform reductions: First, initialize some variables:

```
nextStacks=[]
active=set()
```

  - nextStacks will be the stacks for the next step of parsing
  - active is a set of stack keys so we can avoid duplicates

Reductions

- We loop over the stacks and process each one:

```
stackIndex=0
while stackIndex < len(currentStacks):
    stk = currentStacks[stackIndex]
    stateNumber = stk.top()
    for op in table[stateNumber][sym]:
        ...process table entry...
    stackIndex+=1
```

Reductions

- How do we process table entry “op”
- First, determine if op calls for shift or reduce
- If shift and stack’s key is not in active:
  - Add the stack to nextStacks
  - Add its key to active
- If reduction:
  - If reducing to S’: Stop: Report success
  - Else, Perform reduction...

Reductions

- Performing the reduction:
  - Let numPop and tSym be the data in op
  - Let stk be the stack that we're using for the reduction
  - stk2 = stk.clone()
  - Pop numPop things from stk2
  - Let newState be table[stk2.top()][tSym]
  - Push newState to stk2
  - If stk2's key not in active:
    - Add stk2's key to active and add stk2 to nextStacks and add stk2 to currentStacks
    - Why add to currentStacks? We might need to do another reduction using stk2 before shifting

Shifts

- Once we've finished reducing, we're ready to shift
- Set stacks = nextStacks
- Let nextStacks = []
- For each stack stk in stacks:
  - Let stateNumber = stk.top()
  - If there's a shift entry in table[stateNumber][nextToken.sym]
    - stk.push(stateNumber)
    - Append stk to nextStacks
- When done, set stacks = nextStacks

Tree

- How to recreate the tree? It's not very useful to just report "parsed" or "failed"
- Add a member to each list Node saying how it was obtained
- So we end up with code like this:

```
class Node:
    #how = how we created this:
    #   (S, node, tokenIndex) or
    #   (R, node, numpop, sym)
    def __init__(self,data,how,nxt):
        self.data = data
        self.next = nxt
        self.how = how
```

Tree

- And now we tweak the push function:

```
def push(self,data,how):
    self.top_ = Node(data,how,self.top_)
```

Push

- The initial push to the initial stack looks like this:
  stacks[0].push(0,None)
- When we do a reduction, we do this:
  stk2.push(newState, ("R", stk.topNode(), numPop, tsym))
- When we do a shift, we do:
  stk.push(newState, ("S", stk.topNode(), tokenIndex))

Tree

- If we reduce to S': We reconstruct the tree by walking the "how" links in reverse
- Let successfulStack be the stack which we are going to reduce to S'

```
actions=[]
N = successfulStack.topNode()
while N.how:
    #how is either (S, node reference, tokenIndex) or
    # (R, node reference, num pop, symbol)
    actions.append( (N.how,N) )
    N = N.how[1]
```

Tree

- We can now build the tree

```
treeNodeStack=[]
for how,listNode in reversed(actions):
    if how[0] == "S":
        _,_,ti = how
        token = tokens[ti]
        treeNode = TreeNode(token.sym)
        treeNode.token = token
        treeNodeStack.append(treeNode)
    else:
        assert how[0] == "R"
        _,_,numPop,tSym = how
        node = TreeNode(tSym)
        for i in range(numPop):
            c = treeNodeStack.pop()
            node.children.prepend(c)
        treeNodeStack.append(node)
```

- When done, treeNodeStack[0] has the root of the parse tree

Analysis

- Can deal with *any* CFG (even ambiguous ones)
  - Albeit we need to remove λ productions
  - That's a straightforward transformation
- But: Could be slow
  - Depends on the grammar
  - If we have only one choice at a particular point, just act like normal LR parser
    - Faster
    - No need to clone stacks; just apply the (unambiguous) operation to the stack in question
- Crafted grammars/inputs might force large number of stacks

Ambiguity

- Suppose we have ambiguous construct (so we have ≥ two live stacks)
  - Once we're past ambiguous part, both stacks "collapse" into one
    - Amounts to resolving ambiguity by arbitrarily choosing one option
  - Could add some rules saying which one to eliminate

Example

- Suppose grammar was $E \longrightarrow E + E \mid T$
- Could examine stack actions and prefer one that was left-recursive

Assignment

- Bonus lab: Implement a GLR parser. Use the previous lab's test harness to generate a parse tree.

Sources

- Wikipedia. Parsing Expression Grammar.
- http://bford.info/pub/lang/thesis/
- GLL parsing: http://ldta.info/2009/ldta2009proceedings.pdf

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org