LR(0) DFA

Motivation

- Suppose we have a left associative operator:
  exp $\rightarrow$ exp ADDOP NUM | NUM
- But our LL parser can't handle this
- We instead had to do:
  exp $\rightarrow$ NUM exp'
  exp' $\rightarrow$ ADDOP NUM exp' | λ
- Parse tree is very different
  - Harder to work with

Grammar

- Suppose we have:
  - stmt → assign-stmt | func-call | if-expr
  - assign-stmt → id = expr
  - func-call → id (param-list)
- This is not LL(1)
  - If we are expanding stmt and we see: “id”
    - Is it stmt → assign-stmt
    - Or stmt → func-call
- Must look ahead another token and write LL(2) parser
  - Parse table has *many* more entries

Grammar

- Alternate solution: Change grammar:
  - stmt $\rightarrow$ assignorfunc | if-expr
  - assignorfunc $\rightarrow$ id assignorfunc’
  - assignorfunc’ $\rightarrow$ (param-list) | = expr
- Consider parse tree for “x=42”
  - More confusing

Grammar

- Ambiguous grammars cause difficulty
- Grammar:
  - if-stmt $\rightarrow$ if id stmt if-stmt'
  - if-stmt' $\rightarrow$ else stmt | λ
  - stmt $\rightarrow$ if-stmt | assign-stmt
  - assign-stmt $\rightarrow$ id = num
- The parse table will have a conflict

Bottom-Up

- More powerful than top down
  - Recognizes LR languages
  - LL $\subseteq$ LR
- In particular: Left recursion is OK; left factoring is not needed
  - Ambiguous languages still a problem
- Drawback: Parser is more complex
- Drawback: Error reporting is not as nice

Top vs Bottom

- Top down parsers grow parse tree from the top down
  - Begin with S, keep deriving until we get to leaves
- Bottom up parsing goes the other way
  - Start with leaves (tokens), grow tree toward the root

Operation

- Parser maintains a stack
- Exactly two possibilities at any point in time:

Shift Move token from input to top of stack

- Corresponds to starting a new sub-tree at its leaf

Reduce Pop things representing a complete RHS from the stack and push corresponding LHS

- Corresponds to growing a subtree toward the root

Example

- Grammar:
  - $S \rightarrow S + \text{num} \mid \text{num}$
- Input: 1 + 2 + 3
- Actions:
  Shift 1, Reduce $S \rightarrow \text{num}$, Shift +, Shift 2, Reduce $S \rightarrow S+\text{num}$, Shift +, Shift 3, Reduce $S \rightarrow S+\text{num}$, Done.

Preparation

- We always add a new rule + new start symbol to grammar
  - S’ $\rightarrow$ S
- Assume input has pseudotoken $ at end
- Accepting configuration:
  - Reduce via rule S’ $\rightarrow$ S
  - No more input left

Question

- How does parser know when to shift and when to reduce?
- Concept: LR(0) Item (or just "item")
- Item = A production with a *distinguished position*
- Example: exp $\rightarrow$ exp $\bullet$ op num
  - The $\bullet$ marks the distinguished position

Example

- exp $\rightarrow$ exp op num
- This one production yields *four* LR0 items:
  - exp $\rightarrow \bullet$ exp op num
  - exp $\rightarrow$ exp $\bullet$ op num
  - exp $\rightarrow$ exp op $\bullet$ num
  - exp $\rightarrow$ exp op num $\bullet$
- Note: Rule: X $\rightarrow \lambda$ gives only:
  - X $\rightarrow \bullet$

Meaning

- If we have $X \to Y \bullet Z$
  - We are working on eventually reducing to X
  - We have successfully seen Y
    - Instead of just "Y", we might have any number of things here (even nothing)
  - We have yet to see Z
    - Which could also be zero or more symbols

Meaning

- Remember, bottom-up parsing works from right to left
- Recognize a RHS, then convert it to the LHS
- Keep doing until you get all the way to the root of the parse tree

Example

- Given grammar production: $\text{exp} \rightarrow \text{exp} + \text{num} \mid \text{num}$
- We have six LR(0) items
  - What are they?

Example

- If we are at exp $\rightarrow$ exp $\bullet$ + num
  - We have reduced something to exp already
  - If we shift a +: We will be in state exp $\rightarrow$ exp + $\bullet$ num
  - And then if we shift num: We are in state exp $\rightarrow$ exp + num $\bullet$
  - Since $\bullet$ is at end: We may reduce to exp
    - Pop three things from stack (which will represent exp, +, and num), push one (which will represent exp)

Parsing

- To keep track of this, we can construct an automaton
  - Labels = LR(0) items
  - Read input tokens
  - These determine which transitions we take in FA
  - If we reach state with label of form:
    W $\rightarrow$ X Y Z $\bullet$
    Do a reduction from X Y Z to W
- What's not to like?

Item

- We can represent an Item like so (C# style syntax):

```
public class LR0Item
{
    public readonly string Lhs;
    public readonly List<string> Rhs;
    public readonly int Dpos;   //index of thing after dist. pos.
    public LR0Item(string lhs, List<string> rhs, int dpos)
    {
        this.Lhs = lhs;
        this.Rhs = rhs;
        this.Dpos = dpos;
    }
    public bool DposAtEnd(){
        return Dpos == Rhs.Count;
    }
}
```

Pitfall

- Note: Need to override some builtin functions
  - Ex: Python: __hash__ and __eq__ and __ne__
  - Ex: C#: GetHashCode and Equals and operator== and operator!= for LR0Item class

C# Syntax

```
public class LR0Item {
    public override int GetHashCode(){
        ...
    }
    public override bool Equals(object oo){
        if(oo == null)
            return false;
        LR0Item o = oo as LR0Item;
        if(o == null)
            return false;
        ...
    }
    public static bool operator ==(LR0Item o1, LR0Item o2){
        return Object.Equals(o1,o2);
    }
    public static bool operator!=(LR0Item o1, LR0Item o2){
        return !(o1 == o2);
    }
}
```

DFA State

- We have an DFA state represented like so:

```
class State:
    def __init__(self, itemSet):
        self.items = itemSet
        self.transitions = {}
```

- itemSet = set of Item's
- Transitions: Key = string, value = a single State

Bootstrap

- To begin the process:
  - Q = State()
  - Q.items.add((“S’”, [“S”], 0))
- Q is our start state

Function

- We need a function (call it “computeClosure”) which will take a set of LR(0) items and compute its closure:
  - If any • appears before a nonterminal N:
    - Construct all possible Items that have lhs of N and distinguished position at beginning
    - Add those items to the set
    - Repeat until everything stabilizes

Example

- Suppose we have these grammar rules:

$S \rightarrow A\ A \mid \lambda$

$A \rightarrow B\ c\ d \mid z$

$B \rightarrow C\ y \mid D\ x$

$C \rightarrow q \mid w\ w\ C$

$D \rightarrow z$

Example

- Suppose we call computeClosure({ $S \rightarrow \bullet$ A A })
- Since $\bullet$ precedes an A, we are potentially able to recognize an A
- So we add two more items to the set:
  - $A \rightarrow \bullet$ B c d
  - $A \rightarrow \bullet$ z

But...

- But: look again: $A \rightarrow \bullet B\ c\ d$
- This says "we are about to recognize a B"
- So we add two more items:
  - $B \rightarrow \bullet C\ y$
  - $B \rightarrow \bullet D\ x$

Result

- Thus, the set now has these items:
  - $S \rightarrow \bullet\ A\ A$
  - $A \rightarrow \bullet\ B\ c\ d$
  - $A \rightarrow \bullet\ z$
  - $B \rightarrow \bullet\ C\ y$
  - $B \rightarrow \bullet\ D\ x$

But...

- But we aren't done yet!
- B $\rightarrow$ • C y
  - We might be about to recognize a C
- Add two more items:
  - C $\rightarrow$ • q
  - C $\rightarrow$ • w w C
- B $\rightarrow$ • D x
  - Might be about to recognize a D
- So add:
  - D $\rightarrow$ • z

Result

- All of these end up in the set:
  - $S \rightarrow \bullet\ A\ A$
  - $A \rightarrow \bullet\ B\ c\ d$
  - $A \rightarrow \bullet\ z$
  - $B \rightarrow \bullet\ C\ y$
  - $B \rightarrow \bullet\ D\ x$
  - $C \rightarrow \bullet\ q$
  - $C \rightarrow \bullet\ w\ w\ C$
  - $D \rightarrow \bullet\ z$

Meaning

- What this means: We originally had "S $\rightarrow\bullet$ A A"
- That said "We are about to recognize two A's"
- The first step of that is that we might be about to recognize "B c d" or "z" (which reduces to A) or "C y" or "D x" or "q" or "w w C" or "z" (which reduces to D)

Function

- How do we code our computeClosure()?

```
def computeClosure(S):  #S is a set of Item's
    S2 = copy of S
    toConsider = list(S)
    i=0
    while i < len(toConsider):
        item = toConsider[i]
        i += 1
        lhs,rhs,dpos = item
        if dpos not at end of rhs:
            sym = symbol after dpos
            if sym is nonterminal:
                for all productions P with lhs of sym:
                    item2 = Item(sym, P, 0 )
                    if item2 not in S2:
                        S2.add(item2)
                        toConsider.append(item2)
    return S2
```

Example

- Suppose we have this grammar:
  S' $\rightarrow$ S
  S $\rightarrow$ z
- Show: computeClosure({ S' $\rightarrow$ •S })

Example

- Suppose we have this grammar:
  $S' \rightarrow S$
  $S \rightarrow y \mid z$
- Show: computeClosure({ $S' \rightarrow \bullet S$ })

Example

- Suppose we have this grammar:
  S' $\rightarrow$ S
  S $\rightarrow$ A x | B y | z
  A $\rightarrow$ q r
  B $\rightarrow$ S B | λ
- Show: computeClosure({ S' $\rightarrow$ •S })

Example

- It's all of these!
  - S' $\rightarrow \bullet$ S
  - S $\rightarrow \bullet$ z
  - S $\rightarrow \bullet$ A x
  - S $\rightarrow \bullet$ B y
  - A $\rightarrow \bullet$ q r
  - B $\rightarrow \bullet$ S B
  - B $\rightarrow \bullet$

DFA Generation

- Now we can make a DFA
- Remember, the DFA will guide the parse
- Bootstrap:

```
S = set()
S.add( S' → • S )
startState = DFAState( computeClosure(S) )
todo = [startState]
seen = map()          #map from closure set to DFA state
seen[startState.items] = startState
```

- Pitfall: Need to teach C# how to compare sets for 'seen'

Example

```
class EQ : IEqualityComparer<HashSet<LR0Item> > {
    public EQ(){}
    public bool Equals(HashSet<LR0Item> a, HashSet<LR0Item> b){
        return a.SetEquals(b);
    }
    public int GetHashCode(HashSet<LR0Item> x){
        int h=0;
        foreach(var i in x){
            h ^= i.GetHashCode();
        }
        return h;
    }
}
...
var seen = new Dictionary<HashSet<LR0Item>,DFAState>( new EQ() );
```

Loop

- Do until we've generated entire automaton:

```
while todo not empty:
    Q = todo.pop()
    transitions = computeTransitions(Q)
    addStates( Q, transitions, seen, todo )
```

computeTransitions

- Compute all the transitions that go out of state Q
- Key = symbol (string), value = set of Item's

```
def computeTransitions(Q):
    transitions = {}
    for I in Q.items:
        lhs,rhs,dpos = I
        if dpos not at end:
            sym = rhs[dpos]
            if sym not in transitions:
                transitions[sym]=set()
            transitions[sym].add((lhs,rhs,dpos+1))
    return transitions
```

addStates

- Add new states to automaton and record them as ones to process, if needed
- Inputs:
  - Q (the state to add transitions to)
  - transitions (dictionary; key=string, value=set of LR0Items)
  - seen (map from set of LR0Items to DFA states)
  - todo (newly created states that need to be processed

addStates

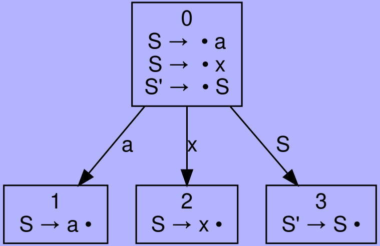
```
def addStates(Q, transitions, seen, todo ):
    for sym in transitions:
        I2 = computeClosure( transitions[sym] )
        if I2 not in seen:
            Q2 = DFAState(I2)
            seen[I2]=Q2
            todo.append(Q2)
        Q.transitions[sym]=seen[I2]
```

Example 1

- Construct DFA for this grammar:
  - $S \rightarrow a \mid x$

Example 1

- The DFA

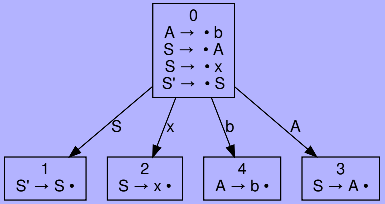


Example 2

- Construct DFA for this grammar:
  - $S \to A \mid x$
  - $A \to b$

Example 2

- The DFA

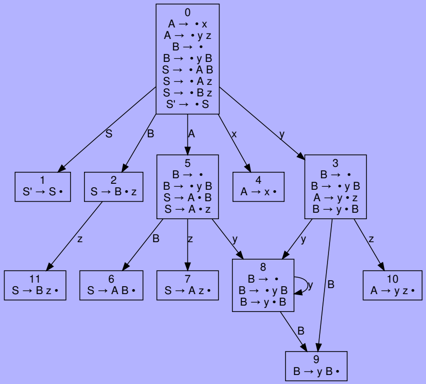


Example 3

- Construct DFA for this grammar:
  - $S \rightarrow A\ B \mid A\ z \mid B\ z$
  - $A \rightarrow y\ z \mid x$
  - $B \rightarrow y\ B \mid \lambda$

Example 3

- The DFA

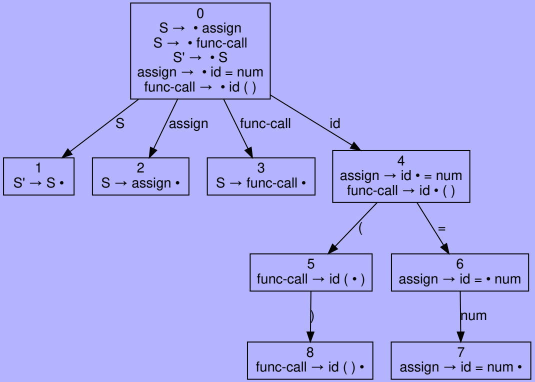


Example 4

- This was one which LL(1) parser couldn't handle
  - S $\rightarrow$ assign | func-call
  - assign $\rightarrow$ id = num
  - func-call $\rightarrow$ id ( )

Example 4

- The DFA



Assignment

- Write a program which works with the test harness

Sources

- K. Louden. *Compiler Construction: Principles and Practice*

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