

LL(*) Parsing

Terence Parr

University of San Francisco

parrt@cs.usfca.edu

parrt@ANTLR.org

Introduction

- Building a parser generator is easy...’cept for the lookahead analysis:
 - rule ref → “rule()”
 - token ref → “match(t)”
 - rule def → <<

```
void rule() {  
    if ( lookahead-expr-alt 1 ) { alt 1; }  
    else if ( lookahead-expr-alt 2 ) { alt 2; }  
    else error;  
}
```

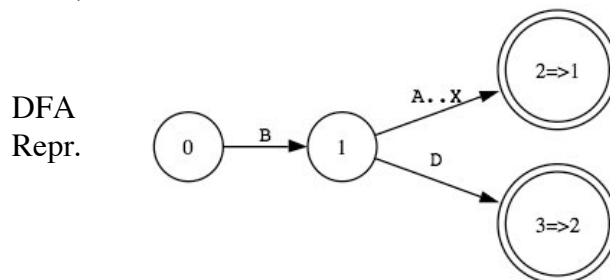
>>
- The nature of the lookahead expressions dictates the strength of your parser generator

LL(k) Lookahead

- Set of k-tuples that predict which alternative will succeed (weaker than LR(k) because LR can decide after seeing entire production)
- k-tuples represent all k-prefixes of sentences that could be matched from left edge
- Number of k-sequences is $O(|T|^k)$ where $|T|$ is size of token space
- Can view k-tuples as a DFA

LL(2) Example

- Grammar requires $k=2$ lookahead to distinguish alts of rule 'a':
 - $a : B (A | X)$ // lookahead is $\{BA, BX\}$
| $B D$ // lookahead is $\{BD\}$



Linear approximate LL(k)

- Reduces $O(|T|^k)$ to $O(|T| \times k)$
 - at the cost of less precise lookahead information
 - weakens the parser (accepts fewer grammars)
 - usually bites me once or twice in a big grammar (frustrating and often mysterious)
- Idea: the sequence of tokens is usually less important than which tokens are some depth
- Mechanism: instead of many k-sequences, use k sets to tokens: collapse all tokens at a particular lookahead depth, yielding one sequence of sets

Approximate LL(2) Example

- Grammar requires $k=2$ lookahead as with LL(2), but notice that $k=1$ set actually confuses the issue! Token at $k=2$ depth uniquely distinguishes
 - `a : B (A | X) // lookahead is {B} {A,X}`
`| B D // lookahead is {B} {D}`
`;`
 - Use: `if (LA(2) in {A,X}) alt 1; else alt 2`
- Reduces a few 2-tuple comparisons to single set test
- Lookahead analysis **AND** lookahead runtime tests are dramatically reduced in complexity
- ANTLR 2.x uses linear approximate lookahead

Part I

Why We Need LL(*)

Problems with LL(k)

- Natural grammars sometimes not LL(k):
func : type ID "(" (arg)* ")" ";"
 | type ID "(" (arg)* ")" "{" body "}"
 ;
- Could left-factor, but it's less natural
- From the left edge, lookahead is not fixed to see the ";" vs "{". We need arbitrary lookahead
- Must specify k *a priori*

Solution -- LL(*)

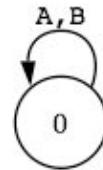
- Idea: Just “spin” ahead with a tight loop looking for the distinguishing single token: “;” or “{”. Then rewind, proceed with predicted alternative
- LL(*) encodes regular lookahead language in, possibly cyclic, DFA
 - DFA does not make method calls simulating LL parse
- Lookahead language is regular when finite; can approximate full non-regular, LL language when infinite with DFA
- LL(*) lookahead from start rule is regular approximation of whole language

LL(*) Benefits

- LL(*) is approximation to actual infinite lookahead language, but far stronger than LL(k)
- You do not need to specify k *a priori*
- Easier learning curve, many fewer ambiguity warnings
- More natural grammars (automatic syn preds)
- Automatically computes minimum lookahead per decision
- DFAs can be fast, much faster than syn preds
- Easily incorporates semantic predicates

LL(k) vs LL(*) Complexity

- Why is LL(k) lookahead $O(|T|^k)$, but LL(*) doesn't exhibit this behavior?
- LL(k) must compute all possible sequences with fixed k length, acyclic DFA
- LL(*) can use cyclic DFA
- LL(2) lookahead of $(A|B)^*$ is $\{AA, AB, BA, BB\}$
- LL(*) lookahead of $(A|B)^*$ is simply:



Relationship to Syntactic Predicates

- Strictly less powerful than syn preds, which backtrack and match exact lookahead language; LL(*) is a covering approximation
- LL(*) should be much faster and doesn't require IF-guessing gates around actions as there are no actions in DFA

Semantic Predicate Hoisting

- Semantic predicates describe semantic validity of alternative and hence can be used as part of lookahead to distinguish alternatives
- “Hoisting” refers to pulling predicates out of a rule for use in disambiguating another
- In general, predicates form expression of AND, OR conditions
- Predicates may only be evaluated in proper syntactic context
- Hoisting unique to PCCTS (ANTLR 1.x); algorithm is/was nasty
- LL(*) hoists, incorporates predicates naturally :)

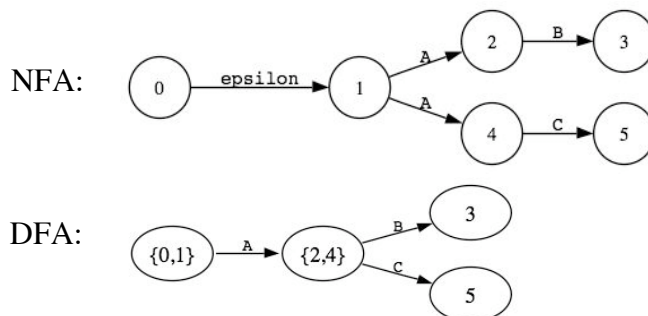
Part II How LL(*) Works

Initial Thoughts

- Build an NFA from a rule and then do a DFA conversion for each alternative starting at the NFA state on left edge of each alt
- Problems!
 - How do you know when to stop conversion? Don't want to convert whole grammar
 - How to detect nondeterminisms/ambiguities?
 - What do you do when rule references and when you hit the end of a rule?
 - How do semantic predicates fit in?
- Correct direction, but more complicated
 - I have phrased LL(*) analysis as an augmented NFA to DFA conversion

Background: NFA to DFA Conversion

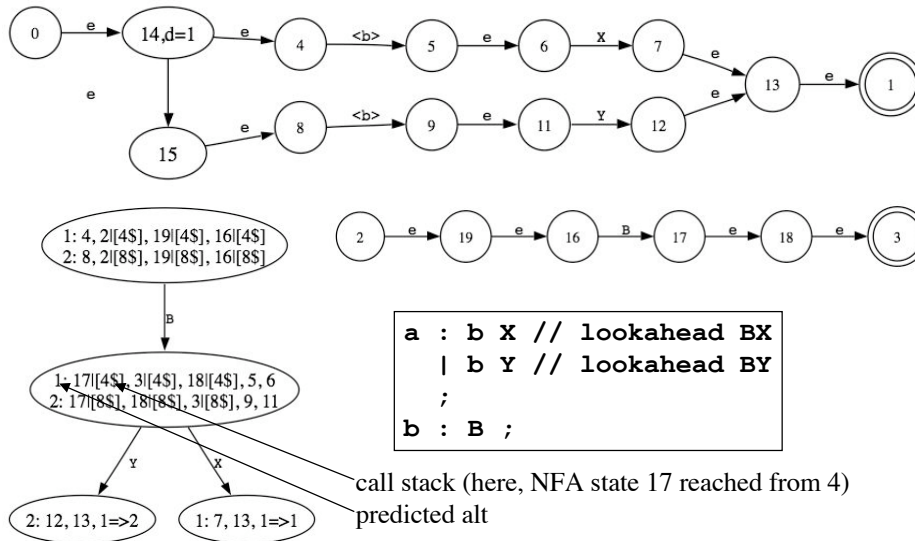
- Idea: “subset construction” since each DFA state is a set of NFA states. DFA state indicates which states the NFA could be in after reading input symbols



LL(*) Analysis

- Augmented subset construction algorithm
- Build single DFA starting at decision left edge that approximates (prefix of) valid upcoming phrases
- Track predicted alt associated with NFA states
- An NFA configuration during grammar NFA to LL(*) lookahead DFA is an NFA state, predicted alt, syntactic context (call stack), and semantic context; notation:
 - state | predicted alt | context | semantic context
- Also, NFA labels can be single int char/token, set of ints, or semantic predicate

Example DFA Construction



Termination, Nondeterminisms

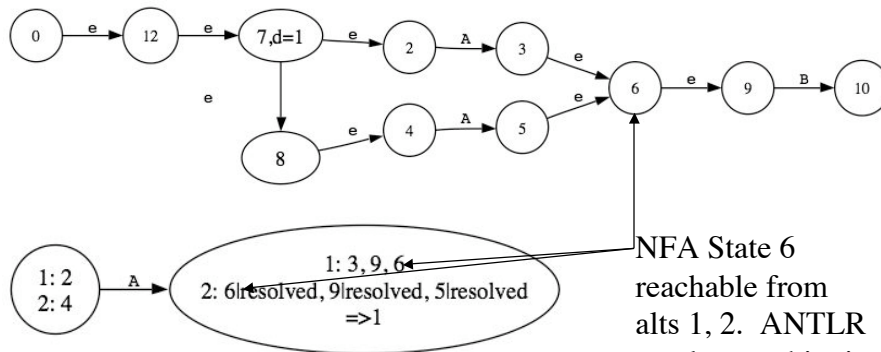
- Terminate DFA construction when a DFA state's configuration set uniquely predicts single alternative or nondeterminism detected
- Nondeterminism: If a DFA state has configs $(s | i | ctx)$ and $(s | j | ctx)$, then some NFA state s predicts alts both i and j with same ctx (call stack); report nondeterminism between i and j
- Final step: verify DFA is reduced (all states have path to accept/alt-predicting state) and that all alternatives have a predict state; if not, nondet.
- This work beyond usual NFA- \rightarrow DFA expensive
 - java.g processing: code optimized from 20min to 10s

Algorithm Terminates

- Algorithm guaranteed to terminate due to fixed amount of work
 - Two DFA states equals() if NFA state | alt pairs same
 - Number of NFA states and alternatives finite
 - Implies fixed number of DFA states to process
 - This equals() definition affects accuracy of lookahead negatively
 - Accuracy/power tradeoff
 - analogous to LALR vs LR; LALR has same (linear) number of states as SLR, but power near LR
 - LR states are unique when all item set and lookahead pairs are unique; exponentially big
 - "compression" where two LALR states are same if they have same core items (doesn't consider lookahead in equals())

Example Nondeterminism

a : (A|A) B ;



NFA State 6
reachable from
alts 1, 2. ANTLR
resolves ambiguity
by choosing lesser
alt number

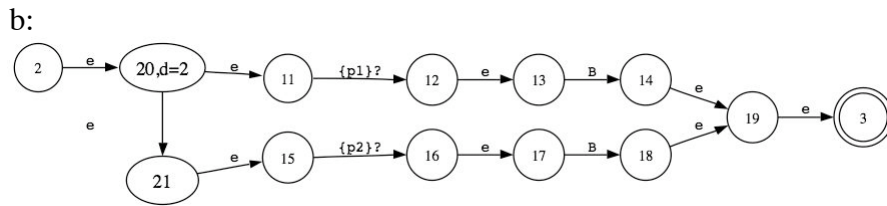
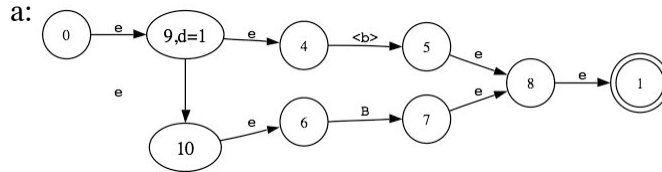
Predicate Hoisting

- Upon syntactic nondeterminism, use semantic predicates to resolve if n-1 predicates available for n alternatives
- Predicates carried along as context just call call stack and predicted-alternative

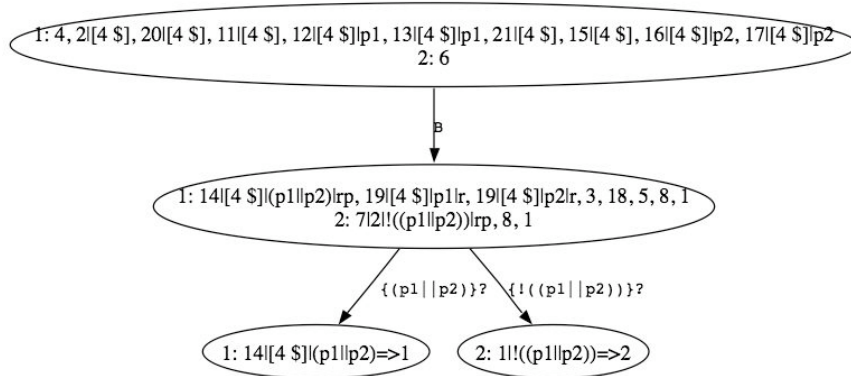
```

a : b // lookahead LA(1)==B && (p1||p2)
   | B // lookahead LA(1)==B && !(p1||p2)
   ;
b : {p1}? B
   | {p2}? B
   ;
    
```

NFA With Predicates



DFA With Predicates



Note: checks syntactic context first, then evaluates predicates to predict alternative (r=resolved, rp=resolved with pred)

Analysis Source Code

- 3542 lines of well-structured, well-documented java code
- 4th complete rewrite
- Primary algorithm extremely clean, but tracking nondeterminisms makes it less clear
- Classes
 - DFA, DFAState, Label, NFA, NFAConfiguration, NFAContext, NFAFactory, NFAState, RuleClosureTransition, SemanticContext, State, StateCluster, Transition