CSE 401 – Compilers

Lecture 9: SLR Parsers, FIRST/FOLLOW Sets

Michael Ringenburg

Winter 2013

Reminders/ Announcements

- Project part 1 is due today!
  - I hope this isn't a surprise for any of you. ☺
- Homework 2 will be assigned today or tomorrow. Due in one week.
- Project part 2 will be assigned this Wednesday. Due on Wednesday, February 13.
- Midterm in class on Friday, February 15.

LR(0) Parser for

0. *S* ::= *E* $
1. *E* ::= *T* + *E*
2. *E* ::= *T*
3. *T* ::= x

What Do We Do?

- How do we solve conflicts like this?
  - Lookahead: look at the next symbol after the handle before deciding whether to reduce.
  - Simplest: SLR (Simplified LR) Parsing uses knowledge of which terminals can follow the LHS nonterminal of a reduction.
  - More complicated LALR and LR parsers actually store a lookahead symbol in items, corresponding to what can follow a *paritcular instance* of a reduction.
    - E.g., If B::= ab | a, then closure of [X::=a.Be] could contain [B::=.a, e] and [B::=.ab, e] (character after ',' is lookahead)

SLR Parsers

- Idea: Reduce conflicts by using information about what can follow a non-terminal to decide if we should perform a reduction: don't reduce if the next input symbol can't follow the resulting non-terminal
- We need to be able to compute FOLLOW(*A*) – the set of symbols that can follow *A* in any possible derivation
  - i.e., t is in FOLLOW(A) if any derivation contains At
  - To compute this, we need to compute FIRST(γ) for strings γ that can follow *A*

Calculating FIRST(γ)

- Sounds easy… If γ = *X Y Z* , then FIRST(γ) is FIRST(*X*), right?
  - But what if we have the rule *X* ::= ε?
  - In that case, FIRST(γ) includes FIRST(Y) … and FIRST(Z) if Y can derive ε.
  - So, computing FIRST and FOLLOW requires knowledge of other symbols FIRST and FOLLOW, as well as which symbols can derive ε.

So How Do We Calculate FIRST/FOLLOW?

- Actually calculate three equations: FIRST, FOLLOW, and nullable
- nullable(*X*) is true if *X* can derive the empty string
- Given a string γ of terminals and non-terminals, FIRST(γ) is the set of terminals that can begin strings derived from γ.
  - Actually, for SLR construction, just need to calculate FIRST(X), where X is a single symbol (terminal or nonterminal)
- FOLLOW(*X*) is the set of terminals that can immediately follow *X* in some derivation
  - We only really need this for nonterminals, but we'll compute it for everything for illustration.
- All three of these are computed together

Computing FIRST, FOLLOW, and nullable

- We use another fixed point algorithm
  - Start with a simple initial state
    - Basically, FIRST(a) = {a} for all terminals **a**
  - Repeatedly apply four simple observations to modify the state
  - Stop when the state no longer changes

Observation 1

- Given a production X ::= Y_1 Y_2 … Y_k
  - If every symbol on the right is nullable (or if there are 0 symbols on the right), then X is nullable.

```
if Y1 … Yk are all nullable (or k == 0)
  set nullable[X] = true
```


Observation 2

- Given a production X ::= Y_1 Y_2 … Y_k and 1 ≤ i ≤ k
  - If the first i-1 symbols on the right are all nullable, then a string derived from X could begin with any terminal that could begin a string derived from Y_i.

```
if Y1 … Yi-1 are all nullable (or i == 1)
  add FIRST[Yi] to FIRST[X]
```

Observation 3

- Given a production X ::= Y_1 Y_2 … Y_k and 1 ≤ i ≤ k
  - If every symbol after Y_i is nullable, then anything that could follow X could also follow Y_i.

```
if Y_{i+1} … Y_k are all nullable (or i == k )
  add FOLLOW[X] to FOLLOW[Y_i]
```


Observation 4

- Given a production X ::= Y_1 Y_2 … Y_k and 1 ≤ i ≤ j ≤ k
  - If every symbol between Y_i and Y_j is nullable, then anything that could start Y_j could follow Y_i.

```
if Y_{i+1} … Y_{j-1} are all nullable (or i+1==j)
  add FIRST[Y_j] to FOLLOW[Y_i]
```

Putting it all together

- Initialization

  set all FIRSTs and FOLLOWs to be empty sets

  set nullable to false for all symbols

  set FIRST[a] to a for all terminal symbols a

Putting it all together

repeat
  for each production $X := Y_1\ Y_2\ \ldots\ Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$, i.e., empty string)
      set nullable[X] = true
    for each i from 1 to k and each j from $i + 1$ to k
      if $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = 1$)
        add FIRST[Y_i] to FIRST[X]
      if $Y_{i+1} \ldots Y_k$ are all nullable (or if $i = k$)
        add FOLLOW[X] to FOLLOW[Y_i]
      if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or if i+1=j)
        add FIRST[Y_j] to FOLLOW[Y_i]
Until FIRST, FOLLOW, and nullable do not change

Example

Z ::= d
Z ::= $X\ Y\ Z$
Y ::= ε
Y ::= c
X ::= Y
X ::= a

repeat
  for each production $X := Y_1\ Y_2 \ldots Y_k$
   if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$, i.e., empty string)
    set nullable[X] = true
   for each i from 1 to k and each j from i +1 to k
    if $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = 1$)
      add FIRST[Y_i] to FIRST[X]
    if $Y_{i+1} \ldots Y_k$ are all nullable (or if $i = k$)
      add FOLLOW[X] to FOLLOW[Y_i]
    if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or if i+1=j)
      add FIRST[Y_j] to FOLLOW[Y_i]
Until FIRST, FOLLOW, and nullable do not change

LR(0) Reduce Actions (review)

- In a LR(0) parser, if a state contains a reduction, it is unconditional regardless of the next input symbol
- Algorithm, where *R* is the set of reduction actions:

```
Initialize R to empty
for each state I in T
   for each item [A ::= α .] in I
      add (I, A ::= α) to R
```

SLR Construction

- This is identical to LR(0) – states, etc., except for the calculation of reduce actions.
- Algorithm, where (I, a, A ::= α) means reduce α to *A* in state *I* if the lookahead is 'a':

```
Initialize R to empty
for each state I in T
   for each item [A ::= α .] in I
      for each terminal a in FOLLOW(A)
          add (I, a, A ::= α) to R
```

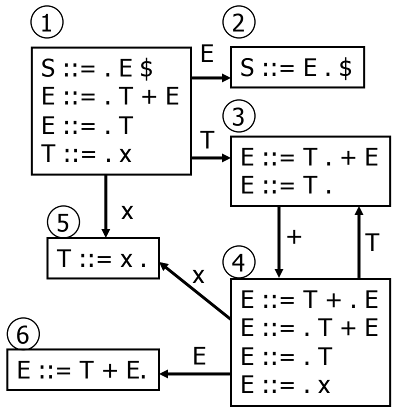

FIRST/FOLLOW for

0. *S* ::= *E* $
1. *E* ::= *T* + *E*
2. *E* ::= *T*
3. *T* ::= x

repeat
  for each production $X := Y_1\ Y_2 \ldots Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$, i.e., empty string)
      set nullable[X] = true
    for each i from 1 to k and each j from $i+1$ to k
      if $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = 1$)
        add FIRST[Y_i] to FIRST[X]
      if $Y_{i+1} \ldots Y_k$ are all nullable (or if $i = k$)
        add FOLLOW[X] to FOLLOW[Y_i]
      if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or if i+1=j)
        add FIRST[Y_j] to FOLLOW[Y_i]
Until FIRST, FOLLOW, and nullable do not change

LR(0) Parser for

0. *S* ::= *E* $
1. *E* ::= *T* + *E*
2. *E* ::= *T*
3. *T* ::= x

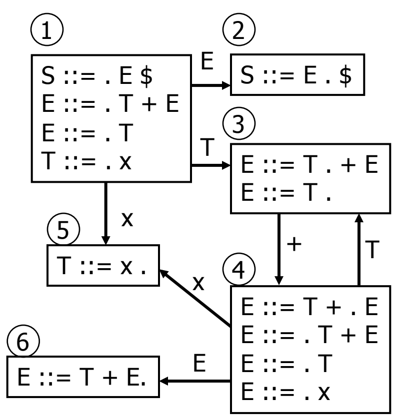


	x	+	$	E	T
1	s5			g2	g3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

FOLLOW(T) = { +, $ }
FOLLOW(E) = { $ }

SLR Parser for

0. *S* ::= *E* $
1. *E* ::= *T* + *E*
2. *E* ::= *T*
3. *T* ::= x



	x	+	$	E	T
1	s5			g2	g3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

FOLLOW(T) = { +, $ }
FOLLOW(E) = { $ }

LR(1) grammars

- Many practical grammars are SLR
- LR(1) is more powerful yet
- Similar construction, but notion of an item is more complex, incorporating lookahead information

LR(1) Items

- An LR(1) item [$A ::= \alpha . \beta$, a] is
  - A grammar production ($A ::= \alpha\beta$)
  - A right hand side position (the dot)
  - A lookahead symbol (a)
- Idea: This item indicates that an A followed by an a would be consistent with the input the parser has seen up to this point.
- Item [$A ::= \alpha$. , a] means reduce to A *if* the next symbol (the lookahead) is a.
  - Note **not** only if – may be item [$A ::= \alpha$. , b] in state
- Key difference is in how you compute the closure.

LR(1) Closure

Closure(S) =
  repeat
    for any item [A ::= α . *X* β, c] in *S*
      for all productions *X* ::= γ
        for each b in FIRST(βc)
          add [*X* ::= . γ, **b**] to *S*
  until S does not change

LR(1) Tradeoffs

- LR(1)
  - Pro: extremely precise; largest set of grammars
  - Con: potentially **VERY** large parse tables with many states
    - This explosion happens during the last step of the *Transition* (aka *Goto*) computation, when you check if an equivalent state already exists. Now, you have to also check whether or not the lookaheads match, and they often don't.
    - Previously, a single state could encode many uses of a handle in the grammar, but now the states encode more contextual information.

Extra State Example (Time Permitting)

0) S′ ::= S$
1) S ::= aAa
2) S ::= bAb
3) A ::= x

LALR(1)

- Variation of LR(1), but merge any two states that differ only in lookahead (all items identical apart from lookahead).
  - Example: these two would be merged

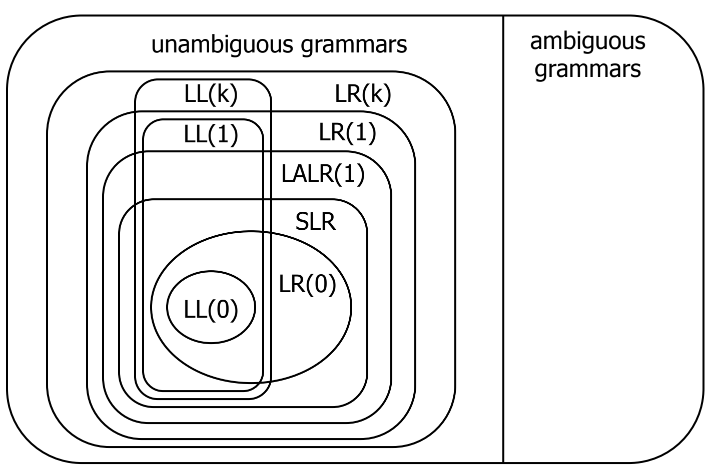
    [*A* ::= x . , a]

    [*A* ::= x . , b]

LALR(1) vs LR(1)

- LALR(1) tables can have many fewer states than LR(1)
  - Somewhat surprising result: will actually have the same number of states as SLR parsers, even though LALR(1) are more powerful.
  - After the merging, acts like SLR parser with "smarter" FOLLOW sets (may be specific to particular handles).
- LALR(1) may have conflicts where LR(1) would not (but in practice this doesn't happen often)
- Most practical bottom-up parser tools are LALR(1) (e.g., yacc, bison, CUP, …)

Language Heirarchies



Coming Attractions

- ASTs – what you do with the parsing!
  - Also, the visitor pattern (useful for traversing the AST, and doing work at each node).
  - Visitor pattern has tripped people up during prior instances of this class, so you'll get it twice – once from me and once from the TAs in section.
- LL(k) Parsing – Top-Down/Recursive Descent Parsers
  - LL Parsers: less powerful, but you can write them completely by hand.