# *CS 440 Exam 2 review 2020-04-14*

**Lectures 13 & 14 - LL table-driven parsing**

- LL(1) parsing is predictive
- Be able to define / calculate First, Follow sets given a grammar
  - With and without ε-rules
- Define / calculate LL(1) Parsing Prediction table given a grammar
  - (Requires First/Follow sets)
- Recognize / simulate some step(s) of a table-driven LL(1) parse

**Lecture 15 Typechecking**

- What is a type? type variable? monomorphic, polymorphic types?
  - Where can type variables appear given two types we want to check for compatibility
  - What is compatibility?
- Instantiation of a type variable means? (type var replaced by type). Show result of an instantiation,
- Are these two types compatible? With what instantiations? What is the common result type?

**Lecture 15 Substitution**

- What is substitution? Textual operation, replace a variable by a piece of text.
  - Textual equality ≡ vs semantic equality (everyday =).
- Notation [Var ↦ Expr], [V1 ↦ E1, V2 ↦ E2], vs [V1 ↦ E1][V2 ↦ E2] (does V2 appear in E1?)
  - Our uses were pretty much all sequential
- Notation σ for substitution (or sequence of substitutions)
  - term σ
  - (X * Y) [X ↦ X+2] ≡ (X+2) * Y — precedences important
  - term1 * term2 σ -— apply just to term2 (substitution treated as high precedence)

**Lectures 15 & 16: Unification**

- Unification equation t ≡ u (?) (t and u are terms) find substitution σ that makes t and u textually the same
  - ie t σ ≡ u σ (exact ≡ here)
- Unification problem — set of unification equations {t1 ≡ u1, t2 ≡ u2}
  - Solve all the equations simultaneously with one σ.
  - t1 σ ≡ u1 σ, t2 σ ≡ u2 σ.
  - {X +Y*z ≡ W*a + (y+z) * z, X+a ≡ Z+a] solved by [X ↦ W*a][Y ↦ y+z][Z ↦ W*a]
    - {W*a + (y+z)*z ≡ W*a + (y+z)*, W*a ≡ W*a}

 

- Don't want a var to appear in substitutions to its right so that we don't reintroduce the variable when we do the sequence of substitutions.

- Unifier - solves a unification problem
  - Generally want most general unifier (mgu)
  - Any other unifier can be derived from an mgu by substitutions ("more general" than derived one)
  - [X ↦ W*a][Y ↦ y+z][Z ↦ W*a] is more general than [X ↦ x*a][Y ↦ y+z][Z ↦ x*a] because we can use [W ↦ x] to go from first to second.
  - mgu's unique up to renaming. Solutions [X ↦ Y][Z ↦ Y] and [Y ↦ X][Z ↦ X] are just renamings of each other (variable to variable), so they're both mgu's
    - X*Y+Z goes to Y*Y+Y or X*X+X, which are just renamings of each other
  - [X ↦ Y] and [Y ↦ X] are renamings, e.g. take X*X+Y to Y*Y+Y or to X*X+X
  - Be able to solve unification problem or say why you can't or verify a solution

**Lecture 17: Bottom-Up and Shift-Reduce Parsing**

- How does bottom-up differ from top-down?
  - Direction of tree building, generates reverse rightmost derivation
  - Non-predictive (don't always know exactly what rule we're trying to parse).
  - Go left-to-right through input, keep track of what we've been able to parse so far.
- Shift-Reduce parsing
  - Use stack to hold incompletely processed terminals & nonterminals so far.
  - Shift - terminal onto stack.
  - Reduce: take top of stack matching rhs of a rule, pop it off, push nonterminal on lhs
    - R → α so stack that had β α turns into β R
    - One reduction can lead to another (if we had a rule T → β R, we could apply it)

**Lecture 17: LR(0) items**

- LR(0) items derived from rules, add dot to indicate where we are in parsing the rule
  - S → • a b, S → a • b, S→ a b •
  - In n.d. finite state machine for parsing, each item is its own state
  - {S → • a b} on a → {S → a • b} on b → {S→ a b •}
    - Action table takes us through this sequence (shift a, shift b, reduce}
  - Have to go back to where we were when we decided to parse the S
    - E.g. {T → α • S β} on ε → {S → • a b} etc
    - have to know to get from {T → α • S β} on S to {T → α S • β}
  - This is what the Go-To table is for.
- To get deterministic version of finite state machine, we use set-of-states transformation
  - In any given deterministic state, we may be in one of some number of n.d. states but we don't know which.

 

**Lecture 18 : LR(0) and SLR(1) parsing**

- All LR family parsers shift exactly same way in same situation (you see symbol c, you push it).
- Differences come in when they decide to reduce.
- LR(0) — reduce whenever possible [0 symbols lookahead]
- SLR(1) — reduce if next symbol in Follow(lhs nonterminal)
  - For $R \rightarrow \alpha \bullet$ figure out what can follow an R. No point in reducing now if that next symbol can't follow an R (it'll just produce an error when we try to shift).
  - LR(0) parser will reduce $R \rightarrow \alpha \bullet$ and generate the shift error
- LR(0) parser action table row will all have the same reduce operation (if it has any)
- SLR(1) row could have a mix of this reduce, another reduce, or blanks. (Say r1 on x, r2 on y)
  - LR(0) parser will have both reductions in every entry in the row (r1 and r2 on x, r1 and r2 on y)
    - Reduce-reduce conflict tells you the language is not LR(0).
- SLR(1) can simulate LR(0) [just ignore the lookahead]

**Lecture 19: Canonical / Full LR(1) Parsing**

- LR(1) parser decides to reduce using an even fine-grained decision.
  - $R \rightarrow \alpha \bullet$ gets reduced if next symbol $\in$ subset of Follow(R) that can follow this particular use of R.
  - E.g. $S \rightarrow R\ x\ R\ y$
    - No point in doing first reduction if next symbol $\neq$ x (e.g. y)
    - No point doing second reduction if next symbol $\neq$ y (e.g. x)
    - SLR(1) parser will do both reductions because next symbol x or y $\in$ Follow(R) = {x, y}
  - LR(1) parser generates error when it can't reduce; SLR(1) reduces and gets error on shift.
- For language that's LR(0), SLR(1) and LR(1) parsers exist also
  - All behave the same except that LR(1) might flag error before SLR(1) before LR(0).
- That's why LR(1) parser has LR(1) items, $R \rightarrow \beta \bullet \gamma$, L where $L \subseteq$ Follow(R) is the follow set for this particular use of R.

**Kinds of LR questions**

- Given a grammar
  - What are the LR(0), LR(1) items?
  - What is the (ε-transition) closure for an item? What transitions does … state have?
  - Given bad input, when will LR(0), SLR(1), LR(1) parser generate error?

 