**Parsing**

* **Building regular expressions w/ CFG**
  + Given reg. expressions E1, E2
  + Concatenation (E1E2)
    - S → AB; A → E1; B → E2
  + Union (E1 | E2)
    - S → A; S → B; A → E1; B → E2
  + Repetition (E1\*)
    - S → SA; A → ε
* Parsing – given grammar G and word w, find a derivation for w
* **Top-down parsing**
  + Start at S, expand according to rules, produce w
  + i.e. find non-terminal, replace it with RHS of production rule
  + Stack-based parsing – can cause problems with empty words & empty stacks
  + Need to augment the grammar by adding:
    - |− start marker
    - −| end marker
    - S’ start symbol that only appears once
    - S’ → |− S −| new production rule
  + A grammar is LL(1) if each cell in the predictor table has at most one entry
    - Left-to-right scan of input
    - Left-most derivations produced
    - 1-symbol look-ahead
    - Allows unambiguous prediction
  + Ex: consider productions
    - 1. S’ → |− S −| 4. A → cd
    - 2. S → AyB 5. B → z
    - 3. A → ab 6. B → wz
  + Predictor table – first()
    - For each non-terminal, determine the possible terminals they can derive on the leftmost side

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **y** | **w** | **z** | **|−** | **−|** |
| **S’** |  |  |  |  |  |  |  | 1 |  |
| **S** | 2 |  | 2 |  |  |  |  |  |  |
| **A** | 3 |  | 4 |  |  |  |  |  |  |
| **B** |  |  |  |  |  | 6 | 5 |  |  |

* + - Start with non-terminal and try all applicable rules, keep track of the terminal in the first character on the left
    - Empty cells are error states
  + Predictor table – follow()
    - Add rule 7. B → ε

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **y** | **w** | **z** | **|−** | **−|** |
| **S’** |  |  |  |  |  |  |  | 1 |  |
| **S** | 2 |  | 2 |  |  |  |  |  |  |
| **A** | 3 |  | 4 |  |  |  |  |  |  |
| **B** |  |  |  |  |  | 6 | 5 |  | 7 |

* + - Note there’s no rule B → −| but comes from S’ → |− S −|
  + Consider α, β ∈ V\*, a, b, ∈ ∑, A ∈ N
    - Empty(α) = true if α ⇒\* ε
    - First(α) = {a | α ⇒\* aβ}
    - Follow(α) = {b | S’ ⇒\* αAbβ}
    - Predict(A, a) = {A → α | a ∈ first(a) } ∪ {A → β | a ∈ follow(A) ∧ empty(β) = true}
  + **Top-down parsing w/ stack:**
    - Start with S’ on the stack
    - When top of stack is a non-terminal → expand using rule
      * Replace non-terminal with RHS of rule
    - When top of stack is a terminal → match with input
      * Pop terminal off stack and read the next input
    - Derivation at each step = input already read + contents of stack (read top to bottom)
    - Complete = when both unread input & stack contain −|
  + Ex: LL(1) parsing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Derivation** | **Read** | **Unread** | **Stack** | **Action** |
| S’ |  | |− abywz −| | > S’ (top on left) | Predict(1) |
| |− S −| |  | |− abywz −| | |− S −| | Match |
| |− S −| | |− | abywz −| | S −| | Predict(2) |
| |− AyB −| | |− | abywz −| | A y B −| | Predict(3) |
| |− abyB −| | |− | abywz −| | a b y B −| | Match |
| |− abyB −| | |− a | bywz −| | b y B −| | Match |
| |− abyB −| | |− ab | ywz −| | y B −| | Match |
| |− abyB −| | |− aby | wz −| | B −| | Predict(6) |
| |− abywz −| | |− aby | wz −| | w z −| | Match |
| |− abywz −| | |− abyw | z −| | z −| | Match |
| |− abywz −| | |− abywz | −| | −| | Accept |

* + Pseudocode:
    - Input = w

Push |− S −|

For each a ∈ w {

While (top of stack is a non-terminal A) {

Pop A

If predict(A, a) = {A → α} // try to find prediction

Push α on stack

Else

Reject

}

Pop c // try to find match

If a ≠ c reject

}

Accept w

* **Bottom-up parsing**
  + Start at w, apply rules in reverse, recover S
  + i.e. find and replace RHS of production rule with non-terminal
  + Derivation at each step = stack (read bottom to top) + input to be read
  + LR parsing:
    - Left-to-right scan
    - Rightmost derivations
  + At each step, either:
    - Shift a char from input to the stack, or
    - Reduce – if the top of stack is the RHS of a rule, replace w/ LHS
  + Accept when:
    - Stack contains S’ and input = ε; or
    - Stack contains |− S −| and input = ε; or
    - −| is shifted
  + Ex: LR parsing
    - Recall rules:
    - 1. S’ → |− S −| 4. A → cd
    - 2. S → AyB 5. B → z
    - 3. A → ab 6. B → wz

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Derivation** | **Read** | **Unread** | **Stack** | **Action** |
| |− abywz −| | |− | abywz −| | |− < (top on right) | Shift |− |
| |− abywz −| | |− a | bywz −| | |− a | Shift a |
| |− abywz −| | |− ab | ywz −| | |− a b | Shift b |
| |− Aywz −| | |− ab | ywz −| | |− A | Reduce(3) |
| |− Aywz −| | |− aby | wz −| | |− A y | Shift y |
| |− Aywz −| | |− abyw | z −| | |– A y w | Shift w |
| |− Aywz −| | |− abywz | −| | |− A y w z | Shift z |
| |− AyB −| | |− abywz | −| | |− A y B | Reduce(6) |
| |− S −| | |− abywz | −| | |− S | Reduce(2) |
| |− S −| | |− abywz −| |  | |− S −| | Shift −| |
| S’ | |− abywz −| |  | S’ | Reduce(1)  Accept |

* + Instead of predictor tables in LL(1), use a transducer to decide when to shift/reduce
  + Donald Knuth’s theorem:
    - { wa | ∃x . S’ ⇒\* wax} is a regular language
    - i.e. can be described by a DFA
  + The LR(0) machine
    - Item – a production with a dot in the RHS – indicates partially matched rule
    - Example: **see written note**
  + Pseudocode for LR(0) automaton:
    - For reach input token

Read stack (bottom → up) + current input

If there is a transition out of current state on current input

Shift & push input onto stack

If current state only has one item & dot is rightmost symbol

Reduce i.e. replace RHS of rule on stack w/ LHS

Pop current state off stack, reread stack

If S’ is on stack & all input is read

Accept

* + Ex: using the automaton

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbol stack** | **States stack** | **Read** | **Unread** | **Action** |
|  | 1 |  | |− id + id + id −| | Shift |− |
| |− | 1 2 | |− | id + id + id −| | Shift id |
| * At S2; transition exists for ‘id’; so shift and go to S6 | | | | |
| |− id | 1 2 6 | |− id | + id + id −| | Reduce id |
| * S6: T → id⋅ only has one item & dot is rightmost symbol; so reduce * Pop RHS (‘id’) off stack; stack = ‘|−’ * Pop top (S6) off states stack; reduce using rule (T → id); push ‘T’; go to S5 | | | | |
| |− T | 1 2 5 | |− id | + id + id −| | Reduce T |
| |− E | 1 2 3 | |− id | + id + id −| | Shift + |
| |− E + | 1 2 3 7 | |− id + | id + id −| | Shift id |
| |− E + id | 1 2 3 7 6 | |− id + id | + id −| | Reduce id |
| |− E + T | 1 2 3 7 8 | |− id + id | + id −| | Reduce E + T |
| * Pop “E + T” off symbol stack; push “E” * Pop matching # of states “3 7 8” off states stack; now at S2; go to S3 via “E” | | | | |
| |− E | 1 2 3 | |− id + id | + id −| | Shift + |
| |− E + | 1 2 3 7 | |− id + id + | id −| | Shift id |
| |− E + id | 1 2 3 7 6 | |− id + id + id | −| | Reduce id |
| |− E + T | 1 2 3 7 8 | |− id + id + id | −| | Reduce E + T |
| |− E | 1 2 3 | |− id + id + id | −| | Shift −| |
| |− E −| | 1 2 3 4 | |− id + id + id −| |  | Reduce |− E −| |
| S’ | 1 | |− id + id + id |  | Accept |

* **LR parsing conflicts**
  + Shift-reduce conflict
    - What if a state looks like:
      * A → α ⋅ c β
      * B → γ ⋅
    - Do we shift c as suggested by rule 1, or
    - Reduce γ to B as suggested by rule 2?
  + Reduce-reduce conflict
    - What if a state looks like:
      * A → α ⋅
      * B → β ⋅
    - Do we reduce α to A as suggested by rule 1, or
    - Reduce β to B as suggested by rule 2?
  + If any complete item (A → α ⋅) occurs in a state & it is not alone then there will be a conflict and the grammar is not LR(0)
  + Example: **see written note**
  + To resolve conflicts, add a look-ahead token
    - For each A → α ⋅, attach Follow(A)
    - Action only applies if next character is in Follow(A)
    - E.g. E → T ⋅ −| only applies when next char is −|
    - E → T ⋅ + E only applies when next char is +
    - This creates a SLR(1) parser
      * i.e. simple LR with 1-char look-ahead