Bottom-up parsing

- Bottom-up parsing works from the input token sequence “up” the parse tree towards the root
- At each step it identifies some sequence of tokens/nonterminals that are the RHS of a rule in the grammar, and applies the rule to work up to a new non-terminal
- The current working sequence of tokens/nonterminals is referred to as the 'frontier', and the rule we select to apply as a 'handle'

Example

stmt --> type VAR EQ expr
expr --> MINUS val | val
val --> NUM | VAR
type --> INT | REAL

Input sequence “int foo = - 27”
Starting frontier, just showing token types:
INT VAR EQ MINUS NUM

<INT> <VAR,foo> <EQ,=> <MINUS,-> <NUM,27>

Arbitrarily try handle type-->INT
gives new frontier: type VAR EQ MINUS NUM

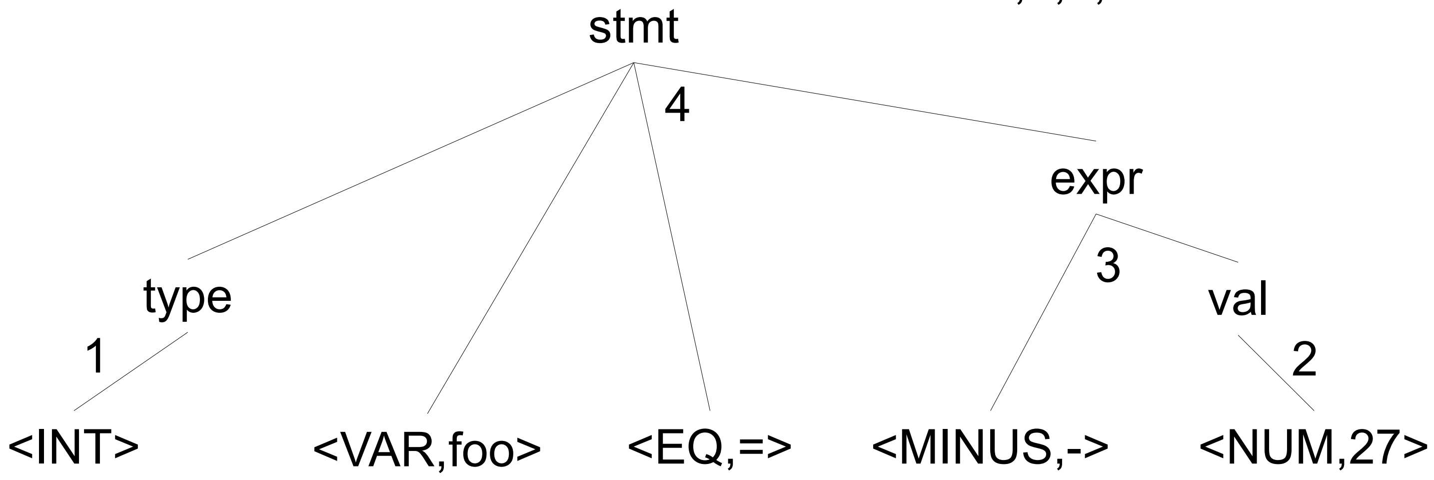
type

<INT> <VAR,foo> <EQ,=> <MINUS,-> <NUM,27>

Example: continued

stmt --> type VAR EQ expr
expr --> MINUS val | val
val --> NUM | VAR
type --> INT | REAL

Next try handle: val-->NUM
..gives frontier type VAR EQ MINUS NUM
...then expr-->MINUS val gives frontier type VAR EQ expr
....finally stmt --> type VAR EQ expr takes us to root
Note order 2,3,1,4 would have worked too...



Handles, building parse tree

- Selecting the correct handle is obviously the crucial point
- Will use <A-->X,k> to refer to the handle whose grammar rule is A-->X, where X represents sequence X1,X2,...Xn of tokens/nonterminals, and k is position of the *right* end of X
- Will refer to replacement action as a reduction (it reduces number of symbols in target string unless n==1)
- The parser would create a new node for A, and link it as parent to existing nodes X1..Xn

Derivations vs bottom-up parse

- A derivation sequence might look like

  start --> X Y Z --> foo Y Z --> foo and Z --> foo and blah

- The desired bottom-up handle sequence would be its reverse, i.e.

  foo and blah --> foo and Z --> foo Y Z --> X Y Z --> start

- LR(1) parsers will scan from left to right and build a rightmost derivation, using one symbol of lookahead
- (it actually builds the rightmost derivation in reverse)

Example: grammar

Similar to grammar considered in scanner section

prog --> BEGIN list END

list --> list stmt | stmt

stmt --> VAR EQ assign TERM

stmt --> PRINT VAR TERM

stmt --> INT VAR TERM

assign --> VAR EQ assign | INUM | RNUM

Sample program

- Sample program and token types (making assumptions about the regex's for the various token types)

```
begin
    int x ;
    int y ;
    x = y = 10 ;
    print x ;
end
```

TOKEN SEQUENCE

```
BEGIN
INT VAR TERM
INT VAR TERM
VAR EQ VAR EQ INUM TERM
PRINT VAR TERM
END
```

Derivation sequence

read BEGIN ***INT VAR TERM***, replace with stmt

now: BEGIN ***stmt***, replace with list

now: BEGIN list

read ***INT VAR TERM***, replace with stmt

now: BEGIN ***list stmt***, replace with list

now: BEGIN list

read VAR EQ VAR EQ ***INUM,*** replace with assign

now: BEGIN list VAR EQ ***VAR EQ ASSIGN,*** replace w/assign

Derivation example continued

now: BEGIN list ***VAR EQ assign TERM,*** replace with stmt

now: BEGIN ***list stmt***, replace with list

read ***PRINT VAR TERM***, replace with stmt

now: BEGIN ***list stmt***, replace with list

now BEGIN list

read END, replace BEGIN list END with prog

now at end of input, only item left is top-level nonterminal (prog), so accept

Stack-based algorithm

- initialize stack to empty, start at beginning of input
- while stack not empty or still input to read:
  - if sequence of items at top of stack match the RHS of a grammar rule then pop those items and push the nonterminal for that grammar rule (e.g. A-->XY and Y is top of stack and X is immediately below Y: pop X and Y, push A)
  - else if out of input and stack contains anything but the root nonterminal then break
  - else read next word of input and push on stack
- accept iff stack contains only the root nonterminal

State-based stack algorithms

- The prev algorithm assumes we look “down” through the stack at each step, to see if we have a reduce match
- Alternatively, we can also record a current state, which keeps track of what kind of matchable things we've currently got on the top of the stack
- e.g. For a rule A --> X Y Z we might have states for (i) haven't seen any of them yet, (ii) have seen a possible X, (iii) have seen possible X Y, (iv) have seen possible X Y Z
- Our shift/reduce decision would then be based on the current state and the next word of input (i.e. LR(1))

Coding approaches

- As with other scanners/parsers, can take either a direct coded approach or a lookup table approach
- Similar limitations: memory use by table, memory use by code, speed of lookup vs speed of code, need to generate either the table or the code
- Typically when we apply a reduction we also want to record (with it) what kind of reduction was applied: either explicitly building a parse tree as we go, or keeping a derivation history so the parse tree can be produced later

Limitations

- Not all grammars are LR(k) for any fixed k:
  - Sometimes you cannot tell whether to push (aka shift) or reduce
  - Sometimes you cannot tell which is the correct grammar rule to reduce
- Heuristics sometimes added for handling these decisions (e.g. prioritize grammar rules in order A B C)
- LR(k) and LR(1) have equivalent power in terms of the languages they recognize, but LR(k) may be able to do so with simpler grammars for a given language