Bottom-up Parsing

Bottom-up parsing is

- more general than (deterministic) top-down parsing (but just as efficient)
- builds on ideas in top-down parsing
- hence, is the preferred method in parser generation tools

Revert to the (unambiguous) natural grammar for our example

```
E --> T + E | T
T --> int * T | int | (E)
```

Consider the string `int * int + int`

Bottom-up parsing *reduces* a string to the start symbol by inverting productions

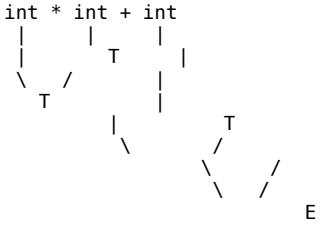
```
int * int + int               int * T + int              T --> int
int * T + int                 T + int                    T --> int * T
T + int                       T + T                      T --> int
T + T                         T + E                      E --> T
T + E                         E                          E --> T + E
E
```

Reading the productions bottom-to-top, these are the productions.
Reading them top-to-bottom, these are reductions

What's the point? The productions read backwards, trace a rightmost derivation, i.e., we are producing the right-most non-terminal at each step

Important fact #1 about bottom-up parsing
Bottom-up parsing traces a right-most derivation in reverse

Picture



Shift-reduce parsing : strategy used by bottom-up parsers

Important fact #1 has an interesting consequence:

- Let `abw` be a step of the bottom-up parse
- Assume the next reduction is by `X--> b`
- Then `w` is a string of terminals
  - because `aXw --> abw` is a step in a right-most derivation

Idea: split string into two substrings

- Right substring is as yet unexamined by parsing
- Left substring has terminals and non-terminals
- The dividing point is marked by |
  - `aX|w`
- Need two kind of moves: shift-move (discussed next) and reduce-move (already discussed)
- The shift-move moves | to the right by one character (signifying that the parser has seen this character)
- The reduce-move applies an inverse reduction to the right-hand of the left-hand string.
  - If `A --> xy` is a production, then

```
Cbxy|ijk => CbA|ijk
```

Example : `int * int + int`
Assume an oracle tells us whether to shift of reduce

```
|int * int  + int                  shift
int |* int + int                   shift
int * |int + int                   shift
int * int| + int                   shift
int *   T| + int                   reduce
T | + int                          reduce
T +| int                           shift
T + int|                           shift
T +   T|                           reduce
T +   E|                           reduce
E|                                 reduce
```

Left string can be implemented by a stack, because we only reduce the suffix of the left string. Top of the stack is |

`shift` pushes a terminal on the stack

`reduce` pops off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

In a given state, more than one action (shift or reduce) may lead to a valid parse.

If it is legal to shift or reduce, there is a *shift-reduce* conflict. Need some techniques to remove them

If it is legal to reduce using two productions, there is a *reduce-reduce* conflict. Are always bad and usually indicate a serious problem with the grammar.

In either case, the parser does not know what to do, and we either need to rewrite the grammar, or need to give the parser a hint on what to do in this situation.

Deciding when to shift and when to reduce

Example : `int * int + int`
Assume an oracle tells us whether to shift of reduce

```
|int * int  + int                    shift
int |* int + int                     shift

At this point, we could reduce.

T |* int + int                       reduce

But this would be a fatal mistake because there is no way we can reduce to E (there is no production that begins with T *).

Intuition: want to reduce only if the result can still be reduced to the start symbol (E)

Assume a right-most derivation:

S -->* \alpha X \omega --> \alpha \beta \omega

Then \alpha \beta is a handle of \alpha \beta \omega

Handles formalize the intuition: A handle is a reduction that allows further reductions back to the start symbol.

We only want to reduce at handles.

Note: we have said what a handle is, not how to find handles.

Important fact #2: In shift-reduce parsing, handles appear only at the top of the stack, never inside. Proof by induction


  1. True initially, stack is empty
  2. Immediately after reducing a handle


        ◦ rightmost non-terminal on top of stack
        ◦ next handle must be to right of rightmost non-terminal, because this is a right-most derivation.
        ◦ i.e., sequence of shift moves reaches the next handle
```

Handles are never to the left of the rightmost non-terminal, and are always at the top of the stack.

Hence shift-reduce parsers can only shift or reduce at the top of the stack. However, how to recognize handles, i.e., when to shift and when to reduce?

Recognizing handles

For an arbitrary grammar, no efficient algorithms known to recognize handles.

Heuristics to identify handles. On certain types of CFGs, heuristics are always correct.

Venn diagram: All CFGs \superset Unambiguous CFGs \superset LR(k) CFGs \superset LALR(k) CFGs \superset SLR(k) \superset LR(0)

LR(k) are quite general, but most practical grammers are LALR(k). SLR(k) are simplifications over LALR(k) [simple LR grammars]. There are grammars that are SLR(k) but not LALR(k) and so on.

\alpha is a *viable prefix* if there is a valid right-most derivation of the string: S' --> ... --> \alpha\omega --> ... --> string. e.g., \epsilon is a viable prefix

In other words, \alpha is a *viable prefix* if there is an \omega such that \alpha|\omega is a state of a shift-reduce parser. Here \alpha is the stack and \omega is the rest of the input. It can lookahead at \omega, but the parser does not know the whole think. The parser knows the whole stack.

What does this mean? A viable prefix does not extend past the right end of the handle. It's a viable prefix because it is a prefix of the handle. As long as a parser has viable prefixes on the stack, no parsing error has been detected.

An item is a production with a "." somewhere on the RHS in the production: e.g., the items for `T --> (E)` are: `T --> .(E)`, `T --> (.E)`, `T --> (E.)`, `T --> (E).`.

The only item for an \epsilon production, `X --> \epsilon` is `X --> .`. Items are often called "LR(0) items".

The problem in recognizing viable prefixes is that the stack has only bits and pieces of the RHS of productions

- If it had a complete RHS, we could reduce

These bits and pieces are always prefixes of RHS of some production(s).

Consider the input `(int)` for the grammar:

```
E --> T + E | T
T --> int * T | int | (E)
```

- Then `(E|)` is a state of a shift-reduce parse
- `(E` is a prefix of the RHS of `T --> (E)`
  - Will be reduced after the next shift
- Item `T --> (E.)` says that so far we have seen `(E` of this production and hope to see `)`

The stack may have many prefixes on RHS's:

```
Prefix1 Prefix2 Prefix3 ... Prefix(n-1) Prefix(n)
```

Let `Prefix(i)` be a prefix of RHS of `Xi --> \alpha_i`

- `Prefix(i)` will eventually reduce to X(i)
- The missing part of \alpha_(i-1) starts with X(i)
- i.e., there is a `X(i-1) --> Prefix(i-1)Xi\beta` production
- and so on.. (e.g., there is a `X(i-2) --> Prefix(i-2)X(i-1)\gamma` production.

Recursively, `Prefix(k+1) ... Prefix(n-1) Prefix(n)` eventually reduces to the missing part of \alpha(k)

Important fact #3 about bottom-up parsing: *For any grammar, the set of viable prefixes is a regular language*. The regular language represents the language formed by concatenating 0 or more prefixes of the productions (items).

For example, the language of viable prefixes for the example grammar:

```
S --> \epsilon | [S]
```

is

```
\epsilon | "["* | "["*S | "["*S"]"
```

The language of viable prefixes for the example grammar:

```
S --> \epsilon | [S] | S.S
```

is

```
X = \epsilon | "[" | "["* | "["*S | "["*S"]"
Y = "["*S.(X | Y)*
Z = X + Y
```

Conversely, In a string is parse-able through the bottom-up parser, then every potential state of the stack should always be a viable prefix.

Consider the string `(int * int)`:

- `int *|int)` is a state of a shift-reduce parse
- "(" is a prefix of the RHS of `T --> (E)`
- "\epsilon" is a prefix of the RHS of `E --> T`. This is an interesting case as we are considering "\epsilon" as a prefix too!
- "int *" is a prefix of the RHS of `T --> int * T`

Alternatively, we can represent this as a "stack of items"

```
T --> (.E)
E --> .T
T --> int * .T
```

says that

- We have seen "(" of `T --> (E)`
- We have seen "\epsilon" of `E --> T`
- We have seen "int *" of `T --> int * T`

Reading backwards, the LHS of every item becomes the RHS of the predecessor production.

In other words, every viable prefix can be represented as a stack of items, where the (n+1)th item is a production for a non-terminal that follows the "." in the nth item.

To recognize viable prefixes, we must

- Recognize a sequence of partial RHS's of productions, where
- Each partial RHS can eventually reduce to part of the missing suffix of its predecessor

Recognizing Viable Prefixes

1. Add a dummy production `S' --> S` to `G`
   This ensures that the start symbol (`S'`) is used only in one place which is the LHS of this production
2. We will construct an NFA that will behave as follows:

```
NFA(stack) = yes if stack is a viable prefix
           = no otherwise
```

3. The NFA will read the input (stack) bottom-to-top
4. The NFA states are the items of G
   - Including the extra production `S' --> S`
5. For item `E --> \alpha.X\beta`, add transition from (E --> \alpha.X\beta) --X--> (E --> \alphaX.\beta)
   - i.e., if we see X in the left state, then we go to the right state.
6. For item `E --> \alpha.X\beta` and production `X --> \gamma`, add (E --> \alpha.X\beta) --\epsilon--> (X --> .\gamma)
7. Every state is an accepting state (i.e., if the entire stack is consumed, the stack is a viable prefix)
8. Start state is (S' --> .S)

Valid Items

- Can construct a DFA from NFA using the standard subset-of-states construction
- The states of the DFA are "canonical collections of items" or "canonical collections of LR(0) items"
  - The dragon book gives another way of constructing the LR(0) items

Item `X --> \beta.\gamma` is *valid* for a viable prefix `\alpha\beta` if

```
S' -->* \alpha X \omega --> \alpha \beta \gamma \omega
```

by a right-most derivation.

After parsing `\alpha \beta`, the valid items are the possible tops of the stack of items

An item `I` is valid for a viable prefix `\alpha` if the DFA recognizing viable prefixes terminates on input `\alpha` in a state `s` containing `I`.

The items in `s` describe what the top of the item stack might be after reading input \alpha

An item is often valid for many prefixes. e.g., The item `T --> (.E)` is valid for prefixes

```
(
((
(((
((((
...
```

We can see this by looking at the DFA, which will keep looping into the same DFA state for each open paren. `Need to show the NFA and DFA construction for our example grammar, and the valid items for these string prefixes. Will need a laptop and a projector!`

Simple LR (SLR) parsing

- LR(0) Parsing: Assume
  - stack contains \alpha
  - next input is `t`
  - DFA on input \alpha terminates in state `s`
- Reduce by `X --> \beta` if
  - `s` contains item `X --> \beta`
- Shift if
  - `s` contains item `X --> \beta.t\omega`
  - equivalent to saying that `s` has a transition labeled `t`
- LR(0) has a reduce/reudce conflict if:
  - Any state has two reduce items: `X --> \beta.` and `Y --> \omega.`
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item: `X --> \beta.` and `Y --> \omega.t\delta`

SLR = "Simple LR": improves on LR(0) shift/reduce heuristics so fewer state have conflicts

Idea: Assume

- stack contains \alpha
- next input is `t`
- DFA on input \alpha terminates in state `s`

- Reduce by `X --> \beta` if
  - `s` contains item `X --> \beta`
  - `t \in Follow(X) [only change to LR(0)]`
- `Shift if`
  - `s contains item X --> \beta.t\omega`

`If there are conflicts under these rules, the grammar is not SLR`

- In other words, SLR grammars are those where the heuristics detect exactly the handles
- SLR(1) grammars work with the SLR algorithm and one lookahead

- Lots of grammars are not SLR
  - including all ambiguous grammars
- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts

- Consider the ambiguous grammar:
  - E --> E+E|E*E|(E)|int
- The DFA for this grammar contains a state with the following items:
  - E-->E*E. and E-->E.+E
  - shift/reduce conflict!
- Declaring "* has higher precedence than +" resolves this conflict in favor of reducing

- The term "precedence declaration" is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!
  - Tools allow you to print out the parsing automaton, and you will be able to see the conflict resolution in the automaton (recommended)

SLR Parsing algorithm

1. Let M be DFA for viable prefixes of G
2. Let |x1...xn$ be initial configuration
3. Repeat until configuration is S|$
   - Let \alpha|\omega be current configuration
   - Run M on current stack \alpha
   - If Mrejects \alpha, report parsing error
     - Stack \alpha is not a viable prefix
   - If M accepts \alpha with items I, let a be next input
   - Shift if X --> \beta.a\gamma \in I
   - Reduce if X --> \beta. \in I and a \in Follow(X)
   - Report parsing error if neither applies

If there is a conflict in the last step, grammar is not SLR(k). k is the amount of lookahead (in practice, k = 1)

SLR Improvements

- Rerunning the viable prefixes automaton on the stack at each step is wasteful
- Most of the work is repeated
- Remember the state of the automaton on each prefix of the stack
- Change stack to contain pairs <Symbol, DFA state>
- For a stack: <sym1,state1>...<symn,staten>
  staten is the final state of the DFA on sym1..symn
- Detail: the bottom of the stack is <any,start> where
  - any is any dummy symbol
  - start is the start state of the DFA
- Define goto[i,A] = j if statei -->A statej
- goto is just the transition function of the DFA
- Shift x
  - Push <a,x> on the stack
  - a is current input
  - x is a DFA state
- Reduce X --> \alpha
  - As before
- Accept
- Error

Action table: For each state si and terminal a

- If si has item X --> \alpha.a\beta and goto[i,a] = j, then action[i,a] = shift j
- If si has item X --> \alpha. and a \in Follow(X) and X != S', then action[i,a] = reduce X-->\alpha
- If si has item S'-->S., then action[i,$] = accept
- Otherwise, action[i,a]=error

SLR Improvements

- Let I = w$ be initial input
- Let j = 0
- Let DFA state 1 have item S'-->.S
- Let stack = <dummy, 1>
- repeat
  case action[top_state(stack),I[j]] of
  - shift k: push < I[j++], k >
  - reduce X-->A:
    - pop |A| pairs
    - push <X,goto[top_state(stack),X]>
  - accept: halt normally
  - error: halt and report error

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!
- However, we still need the symbols for semantic actions (e.g., code generation)

- Some common constructs are not SLR(1)
- LR(1) is more powerful
  - Build lookahead into the items (fine-grained disambiguation for each item, rather than just checking the follow set)
  - An LR(1) item is a pair: LR(0) item, lookahead
  - [T --> .int *T,$] means
    - After seeing T--> int * T, reduce if lookahead is $
  - More accurate than just using follow sets
  - Take a look at the LALR(1) automaton for your parser! (uses these items, but has a slight optimization over it)

- LALR automaton is composed of states containing LR(1) items, with the added optimization that if two states differ only in lookahead then we combine those states. If the resulting states (after this minimization), have no conflict in the grammar, then that grammar is LALR also.

SLR Examples

```
S' --> S
S --> Sa
S --> b

SLR parsers do not mind left-recursive grammars

The first state in the corresponding DFA will look like (\epsilon closure of the NFA state): (state 1)

S' --> .S
S --> .Sa
S --> .b

If we see a b in this state, we get another DFA state: (state 2)

S --> b.

Alternatively, if we see a S in this state, we get another DFA state: (state 3)

S' --> S.
S --> S.a

From this state, if we see a, we get (state 4)

S --> Sa.

The only state with a shift-reduce conflict is state3. Here, if we look at follow of S', we have only "$", and hence we
can resolve this conflict by one lookahead. Hence this is an SLR grammar

Another example grammar

S' --> S
S --> SaS
S --> b

Looking at the corresponding DFA: (state 1)

S' --> .S
S' --> .SaS
S --> .b

One possibility is that we see b in this state to get: (state 2)

S --> b.

Another possibility is that we see S in this state to get: (state 3)

S' --> S.
S' --> S.aS

If we get a in this state, we get (state 4)

S' --> Sa.S
S --> .SaS
S --> .b

(notice that we formed an \epsilon-closure of the first item to add more items

From here, if we get S, we get the following state: (state 5)

S --> SaS.
S --> S.aS

From here, if we get a again, we go back to state 4! If we get b, we go to state 2

The only states that have conflicts are: state3 (resolved by follow as follow(S') = $) and state5 (has a shift/reduce
conflict because a \in follow(S))

Thus this is not an SLR(1) grammar
```