Bottom Up Parsing

Bottom Up Parsing

Also known as Shift-Reduce parsing

More powerful than top down
- Don't need left factored grammars
- Can handle left recursion

Attempt to construct parse tree from an input string
- beginning at leaves and working to top
- Process of reducing strings to a non terminal – shift-reduce
- Uses parse stack
  - Contains symbols already parsed
  - **Shift** until match RHS of production
  - **Reduce** to non-terminal on LHS
  - Eventually reduce to start symbol

Shift and Reduce

Shift:
- Move the first input token to the top of the stack.

Reduce:
- Choose a grammar rule $X \rightarrow \alpha\ \beta\ \gamma$
- pop γ β α from the top of the stack
- push X onto the stack.

Stack is initially empty and the parser is at the beginning of the input.

Shifting $ is **accepts**.

Sentential Form

A **sentential form** is a member of $(T \cup N)^*$ that can be derived in a finite number of steps from the start symbol S.

A sentential form that contains no nonterminal symbols (i.e., is a member of T*) is called a **sentence**.

Handle

Intuition: reduce only if it leads to the start symbol

Handle has to
- match RHS of production and
- lead to rightmost derivation, if reduced to LHS of some rule

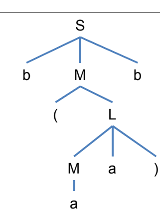
Definition:
- Let $\alpha\beta w$ be a sentential form where:

  α is an arbitrary string of symbols
  $X \rightarrow \beta$ is a production
  w is a string of terminals

  Then β at $\alpha\beta$ is a handle of $\alpha\beta w$ if
  $S \Rightarrow \alpha X w \Rightarrow \alpha\beta w$ by a rightmost derivation
- Handles formalize the intuition (reduce β to X), but doesn't say how to find the handle

Parse Tree

```
S → b M b
M → ( L
M → a
L → M a )
L → )
```



Considering string:
`b ( a a ) b`

$S \Rightarrow b\ M\ b \Rightarrow b\ (\ L\ b \Rightarrow b\ (\ M\ a\ )\ b \Rightarrow b\ (\ a\ a\ )\ b$

Try to find handles and then reduce from sentential form via rightmost derivation

$b\ (\ a\ a\ )\ b \Rightarrow b\ (\ M\ a\ )\ b \Rightarrow b\ (\ L\ b \Rightarrow b\ M\ b \Rightarrow S$

Bottom Up Parsing

Grammar

E → E + E
E → E * E
E → (E)
E → id

Sentential form	Handle	Products
id_1 + id_2 * id_3	id_1	E→ id
E + id_2 * id_3	id_2	E→ id
E + E * id_3	id_3	E→ id
E + E * E	E*E	E→ E*E
E + E	E+E	E→ E+E
E		

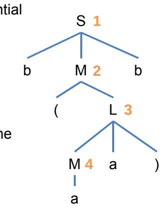
Use • to indicate where we are in string:

id_1• + id_2 * id_3 ⇒ E• + id_2 * id_3 ⇒ E +• id_2* id_3 ⇒
E + E• * id_3 ⇒ E + E * id_3• ⇒ E + E * E• ⇒
E + E• ⇒ E

Issues

We need to locate the handle in the right sentential form and then decide what production to reduce it to – which of the RHS of our grammar.

Notice in right-most derivation, where right sentential form is:



Parsing never has to guess about the middle of the string. The right side always contains terminals.

Thus, we can discover the rightmost derivation in reverse: 4 3 2 1

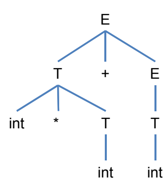
Bottom Up Parsing

Consider our usual grammar and the problem of when to reduce:

E → T + E | T
T → int * T | int | (E)

For the string: `int * int + int`

Sentential form	Production
int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	



Viable Prefix

Definition: α is a **viable prefix** if

- There is a w where αw is a right sentential form
- α•w is a configuration of a shift-reduced parser

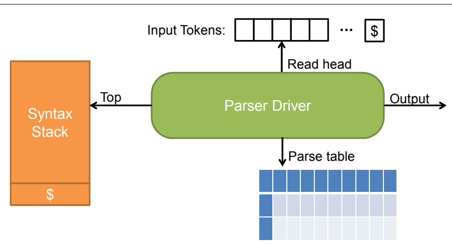
b (a• a) b ⇒ b (M• a) b ⇒ b (L• b ⇒ b M• b ⇒ S•

Alternatively, a prefix of a rightmost derived sentential form is viable if it does not extend the right end of the handle.

A prefix is viable because it can be extended by adding terminals to form a valid (rightmost derived) sentential form

As long as the parser has viable prefixes on the stack, no parsing error has been detected.

Parser Structure



Operations

1. **Shift** – shift input symbol onto the stack
2. **Reduce** – RHS of a non-terminal handle is at the top of the stack. Decide which non-terminal to reduce it to
3. **Accept** – success
4. **Error**

Parse

S → b M b
M → (L
M → a
L → M a)
L →)

String:
b (a a) b $

Stack	Input	Action
$	b (a a) b $	shift
$ b	(a a) b $	shift
$ b (	a a) b $	shift
$ b (a	a) b $	reduce
$ b (M	a) b $	shift
$ b (M a	) b $	shift
$ b (M a)	b $	reduce
$ b (L	b $	reduce
$ b M	b $	shift
$ b M b	$	reduce
$ Z	$	accept

Ambiguous Grammars

Conflicts arise with ambiguous grammars
- Ambiguous grammars generate conflicts but so do other types of grammars

Example:
- Consider the ambiguous grammar

  E → E * E | E + E | (E) | int

Sentential form	Actions
int * int + int	shift
…	…
E * E• + int	reduce E → E * E
E• + int	shift
E +• int	shift
E + int•	reduce E → int
E + E•	reduce E → E + E
E•	

Sentential form	Actions
int * int + int	shift
…	…
E * E• + int	shift
E * E +• int	shift
E * E + int•	reduce E → int
E * E + E•	reduce E → E + E
E * E•	reduce E → E * E
E•	

Ambiguity

In the first step shown, we can either shift or reduce by E → E * E
- Choice because of precedence of + and *
- Same problem with association of * and +

We can always rewrite ambiguous grammars of this sort to encode precedence and association in the grammar
- Sometimes this results in convoluted grammars.
- The tools we will use have other means to encode precedence and association

We must get rid of conflicts !
- Know what a handle is but not clear how to detect it

Properties about Bottom Up Parsing

Handles always appear at the top of the stack
- Never in middle of stack
- Justifies use of stack in shift–reduce parsing

General shift–reduce strategy
- If there is no handle on the stack, shift
- If there is a handle, reduce to the non-terminal

Conflicts
- If it is legal to either shift or reduce then there is a **shift-reduce conflict**.
- If it is legal to reduce by two or more productions, then there is a **reduce-reduce conflict.**

LR Parsers

LR family of parsers
- LR(k)
  - L – left to right
  - R – rightmost derivation in reverse
  - k elements of look ahead

Attractive
- LR(k) is powerful – virtually all language constructs
- Efficient
- LL(k) $\subset$ LR(k)
- LR parsers can detect an error as soon as it is possible to do so
- Automatic technique to generate – YACC, Bison, Java CUP

LR and LL Parsers

LR parser, each reduction needed for parse is detected on the basic of
- Left context
- Reducible phrase
- k terminals of look ahead

LL parser
- Left context
- First k symbols of what right hand side derive (combined phrase and what is to right of phrase)

Types of LR Parsers

SLR – simple LR
- Easiest to implement
- Not as powerful

Canonical LR
- Most powerful
- Expensive to implement

LALR
- Look ahead LR
- In between the 2 previous ones in power and overhead

Overall parsing algorithm is the same – table is different

LR Parser Actions

How does the LR parser know when to shift and when to reduce?

By using a DFA!

The edges of the DFA are labeled by the symbols (terminals and non-terminals) that can appear on the stack.

Five kinds of actions:
1. **s***n* Shift into state *n*;
2. **g***n* Goto state *n*;
3. **r***k* Reduce by rule *k*;
4. **a** Accept;
5. **Error**

LR Parser Actions

Shift(*n*):
- Advance input one token; push *n* on stack.

Reduce(*k*):
- Pop stack as many times as the number of symbols on the right-hand side of rule *k*
- Let *X* be the left-hand-side symbol of rule *k*
- In the state now on top of stack, look up *X* to get "goto *n*"
- Push *n* on top of stack.

Accept:
- Stop parsing, report **success**.

Error:
- Stop parsing, report **failure**.

LR Parsers

Can tell handle by looking at stack top:
- (grammar symbol, state) and k input symbols index our FSA table
- In practice, k<=1

How to construct LR parse table from grammar:
1. First construct SLR parser
2. LR and LALR are augmented basic SLR techniques
3. 2 phases to construct table:
   I. Build deterministic finite state automation to go from state to state
   II. Build table from DFA

Each state – how do we know from grammar where we are in the parse. Production already seen.

Notion of an LR(0) item

An **item** is a production with a distinguished position on the right hand side. This position indicates how much of the production already seen.

Example:

`S → a B S` is a production

Items for the production:

```
S→ • a B S
S→ a • B S
S→ a B • S
S→ a B S •
```

Basic idea: Construct a DFA that recognizes the viable prefixes group items into sets

Construction of LR(0) items

Create augmented grammar G'

```
G:    S → α | β
G':   S' →S
      S → α | β
```

What else is needed:
- `A → c • d E`
  - Indicate a new state by consuming symbol d: need **goto** function
- `A → c d • E`
  - What are all possible things to see – all possible derivations from E? Add strings derivable from E – **closure** function
- `A → c d E •` – reduce to A and **goto** another state

Compute functions **closure** and **goto** will be used to determine the action and goto parts of the parsing table
- **closure** – essentially defines what is expected
- **goto** – moves from one state to another by consuming symbol

LR(0) States

Start with our usual grammar:

```
1.) E → T + E
2.) T → int * T
3.) T → ( E )
```

Add a special start symbol, S, that goes to our original start symbol and $:

```
0.) S → E $
```

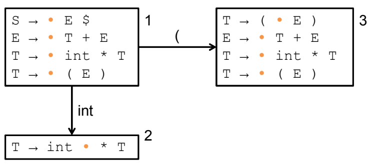
The LR(0) start state will be the set of LR(0) items:

```
S → • E $
E → • T + E
T → • int * T
T → • ( E )
```

LR(0) States

What happens if we **shift** an `int` onto the stack from the start state (1)?

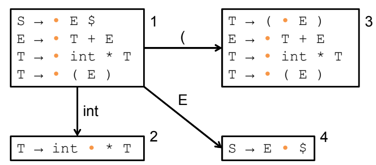
What happens if we **shift** a '(' onto the stack from this start state (1)?



LR(0) States

What happens if we parse some string derived from nonterminal E?

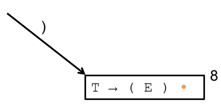
We will execute a **goto** for E in state 1, yielding state 4.



LR(0) States

In state 8, we find that the parsing position is at the end of the item. This means that the top of the stack has the complete RHS of a production on its top.

This is a **reduce** action.



LR(0) Operations

Compute **closure**(I) and **goto**(I, X), where I is a set of items and X is a grammar symbol (terminal or nonterminal).

```
Closure(I) =
    repeat
        for any item A → α•Xβ in I
            for any production X → γ
                I ← I ∪ {X →•γ }
    until I does not change.
    return I
```

Closure adds more items to a set of items when there is a dot to the left of a nonterminal

```
Goto(I, X) =
    J ← {}
    for any item A →α•Xβ in I
        add A →αX•β to J
    return Closure(J)
```

Goto moves the dot past the symbol *X* in all items.

LR(0) Parser Construction

1. Augment the grammar with an auxiliary start production $S \rightarrow S\$$.
2. Let *T* be the set of states seen so far,
3. Let *E* the set of (shift or goto) edges found so far.
4. Make an **accept** action for the symbol $ (do not compute **Goto**(I, $))

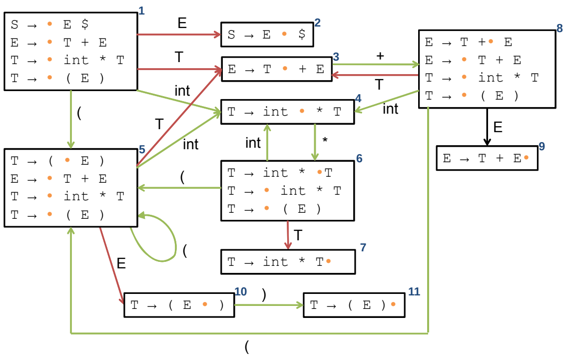
```
Initialize T to {Closure({S' → •S$})}
Initialize E to {}

repeat
    for each state I in T
        for each item A →α•Xβ in I
            let J be Goto(I, X)
            T ← T ∪ {J}
            E ← E ∪ {I ᵡ→ J}
until E and T did not change in this iteration
```

LR(0) Reduce Actions

R is the set of reduce actions

```
R ← {}
for each state I in T
    for each item A → α• in I
        R ← R ∪ {(I, A→α)}
```

DFA



Building the LR(0) Table

```
for each edge I X→ J
    if X is a terminal
        M[I,X] = shift J
    if X is a nonterminal
        M[I,X] = goto J

for each state I containing an item S → S.$
    M[I,$] = accept

for a state containing an item A → γ •
    // A production n with the dot at the end
    for every token Y
        M[I,Y] = reduce n
```

LR(0) Parse Table

	int	(	)	*	+	$	E	T
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								

LR(0) Parse Table

	int	(	)	*	+	$	E	T
1	s4	s5					g2	g3
2						a		
3					s8			
4				s6				
5	s4	s5					g10	g3
6	s4	s5						g7
7	r2	r2	r2	r2	r2	r2		
8							g9	g3
9	r1	r1	r1	r1	r1	r1		
10			s11					
11	r3	r3	r3	r3	r3	r3		