

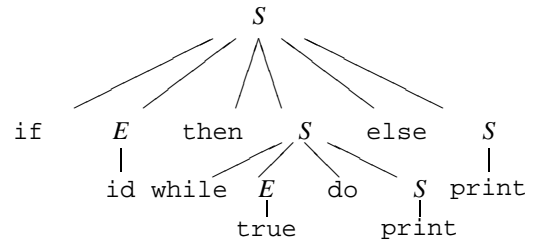
CS321 F'04 Lecture Notes
Lecture 9

Bottom-up Parse Example

$S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{print}$
 $E \rightarrow \text{true} \mid \text{false} \mid \text{id}$

if id then while true do print else print

Parse Tree:



S
 \Rightarrow_{lm} if E then S else S
 \Rightarrow_{lm} if id then S else S
 \Rightarrow_{lm} if id then while E do S else S
 \Rightarrow_{lm} if id then while true do S else S
 \Rightarrow_{lm} if id then while true do print else S
 \Rightarrow_{lm} if id then while true do print else print
 \Leftarrow_{rm} if E then while true do print else print
 \Leftarrow_{rm} if E then while E do print else print
 \Leftarrow_{rm} if E then while E do S else print
 \Leftarrow_{rm} if E then S else print
 \Leftarrow_{rm} if E then S else S
 \Leftarrow_{rm} S

Top-down vs. Bottom-up Parsing

Top-down:

- Construct tree from root to leaves.
- “Guess” which RHS to substitute for non-terminal.
- Produces left-most derivation.
- Recursive-descent, LL parsers.
- “Easy” for humans.

Bottom-up:

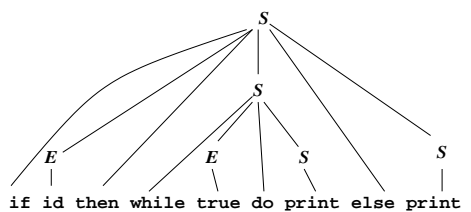
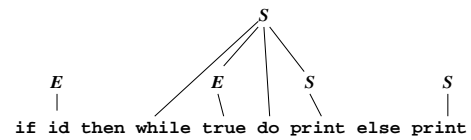
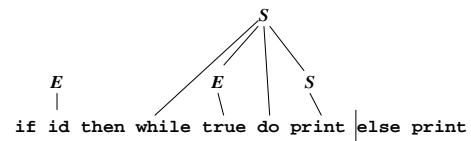
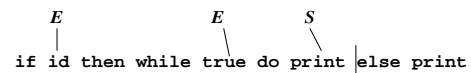
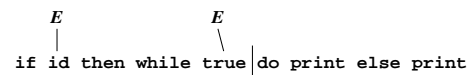
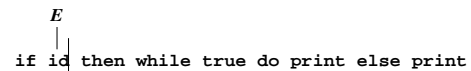
- Construct tree from leaves to root.
- “Guess” which rule to “reduce” terminals.
- Produces reverse right-most derivation.
- Shift-reduce, LR, LALR, etc.
- yacc or CUP parser generator.
- “Harder” for humans.

Bottom-up can parse a larger set of languages than top-down.

Both work for most (but not all) features of most computer languages.

Bottom-up Parse

| if id then while true do print else print



Bottom-up Parsing

There are many bottom-up parsing algorithms, suitable for different subsets of CFG's.

Basic idea: Given input string w , “**reduce**” it to the goal (start) symbol, by looking for substrings that match production r.h.s.'s.

Example:

$$\begin{aligned} S &\rightarrow aAcBe \\ A &\rightarrow Ab \mid b \\ B &\rightarrow d \end{aligned}$$

| “Right sentential form” | Reduction |
|-------------------------|-----------------------|
| $abcde$ | |
| $aAbcde$ | $A \rightarrow b$ |
| $aAcde$ | $A \rightarrow Ab$ |
| $aAcBe$ | $B \rightarrow d$ |
| S | $S \rightarrow aAcBe$ |

Steps correspond to a right-most derivation in reverse.

Note: must choose r.h.s. wisely!

“Handle Pruning”

Idea: Keep removing handles, replacing them with corresponding l.h.s. of production, until we reach S .

Another example:

$$E \rightarrow E+E \mid E^*E \mid (E) \mid id$$

| Right-sentential form | Handle | Reducing production |
|-----------------------|--------|----------------------|
| $\underline{a}+b^*c$ | a | $E \rightarrow id$ |
| $E+\underline{b}^*c$ | b | $E \rightarrow id$ |
| $E+E^*\underline{c}$ | c | $E \rightarrow id$ |
| $E+E^*E$ | E^*E | $E \rightarrow E^*E$ |
| $\underline{E+E}$ | $E+E$ | $E \rightarrow E+E$ |
| E | | |

Note that grammar is ambiguous, so there are actually **two** handles at next-to-last step.

Big question: How do we identify handles?

- We will not answer in this course (see Cooper and Torczon section 3.5).

Fortunately, we can use parser-generators that compute the handles for us.

Will concentrate on framework used for bottom-up parsing, so that we can understand generator behavior.

Handles

Don't always make progress by replacing a substring with the l.h.s. of a matching production.

Example:

$$\begin{aligned} abcde \\ aAbcde \quad A \rightarrow b \\ aAAcde \quad A \rightarrow b \end{aligned}$$

Stuck!

A **handle** is a substring that

- is the r.h.s. of some production; **and**
- whose replacement by the production's l.h.s. is a (reverse) step in a rightmost derivation.

If grammar is unambiguous, handle is **unique**.

More formally, a handle is a **production** $A \rightarrow \beta$ and a **position** in the current right-sentential form $\alpha\beta w$ such that:

$$S \xrightarrow{*}_{rm} \alpha A w \Rightarrow_{rm} \alpha \beta w$$

For example grammar, if current right-sentential form is

$$aAbcde$$

then the handle is $A \rightarrow Ab$ at the marked position.

Note that w never contains non-terminals.

Shift-reduce Parsing

Machine framework common to bottom-up parsers.

Have **stack** to hold grammar symbols and **input buffer** to hold string to be parsed.

Machine actions:

- **Shift** input symbols from buffer to stack until a handle is formed.
- **Reduce** handle by replacing grammar symbols at top of stack by l.h.s. of production.
- **Accept** on successful completion of parse.
- **Fail** on syntax error.

Why a **stack**?

Because handles always appear at the top of a stack, i.e., there's no need to look deeper into the “state.” This is just a fact about rightmost derivations.

Shift-Reduce Parsing Example

$$E \rightarrow E+E \mid E^*E \mid (E) \mid id$$

| Stack | Input Buffer | Action |
|---------|--------------|------------------------------|
| \$ | a+b*c\$ | Shift |
| \$a | +b*c\$ | Reduce: $E \rightarrow id$ |
| \$E | +b*c\$ | Shift |
| \$E+ | b*c\$ | Shift |
| \$E+b | *c\$ | Reduce: $E \rightarrow id$ |
| \$E+E | *c\$ | Shift (*) |
| \$E+E* | c\$ | Shift |
| \$E+E*c | \$ | Reduce: $E \rightarrow id$ |
| \$E+E*E | \$ | Reduce: $E \rightarrow E^*E$ |
| \$E+E | \$ | Reduce: $E \rightarrow E+E$ |
| \$E | \$ | Accept |

We can perform “semantic actions” (e.g., build parse tree nodes) when reduce actions are performed.

Again – we haven’t said **how** actions are chosen. (In general, based on stack and input.)

Machine execution shown corresponds to this derivation:

$$E \Rightarrow_{rm} E+E \Rightarrow_{rm} E+E^*E \Rightarrow_{rm} E+E^*c \Rightarrow_{rm} E+b^*c \Rightarrow_{rm} a+b^*c$$

What about $E \Rightarrow_{rm} E^*E \Rightarrow_{rm} E^*c \Rightarrow_{rm} E+E^*c \Rightarrow_{rm} E+b^*c \Rightarrow_{rm} a+b^*c$?? See *ed Shift action.

LR Parsing

LR parsers are most general non-backtracking shift-reduce parsers known.

- **L** stands for “Left-to-right scan of input.”
- **R** stands for “Rightmost derivation (in reverse).”

Efficient implementations are possible.

Any LL grammar is also LR (and so are many others).

Suffices for almost all programming language CFG’s.

Disadvantage: Extremely tedious to build by hand, so need a generator.

Idea: Implement shift-reduce parser using a **DFA** to choose actions based on contents of stack plus zero or more symbols of lookahead.

Components of machine:

- Input buffer.
- Stack of **states** (and grammar symbols). States “summarize” stack contents.
- Parsing tables, which encode DFA.
- Driver routine (fixed for all grammars)

Machine is efficient because actions are determined by input and state at top of stack and.

Conflicts

Ambiguous grammars lead to **parsing conflicts**.

Can fix by **rewriting** grammar or by making appropriate **choice of action** during parsing.

Shift/Reduce conflicts: should we shift or reduce?

- (See previous example)
- Dangling `else` is another example.

Reduce/Reduce conflicts: which production should we reduce with?

Example:

$$\begin{aligned} stmt &\rightarrow id(param) \quad (a(i) \text{ is procedure call}) \\ param &\rightarrow id \\ expr &\rightarrow id(expr) \mid id \quad (a(i) \text{ is array subscript}) \end{aligned}$$

| Stack | Input Buffer | Action |
|----------|--------------|--------------|
| \$...a(i |)...\$ | Reduce by ?? |

Should we reduce to *param* or to *expr*? Need to know the type of *a*: is it an array or a function?

This info. must flow from declaration of *a* to this use, typically via a symbol table.

LR Grammars

If each entry in *LR* parsing table is uniquely defined, grammar is an **LR grammar**.

In an $LR(k)$ grammar, parsing moves are determined by state on top of stack and next k symbols of input. ($k = 0, 1$ usually enough.)

$LR(k)$ grammars don’t suffice for, e.g., dangling `else` construct, but it (and others) can be handled by making a choice of table entry (e.g., Shift or Reduce).

LR comes in different varieties, based on table construction method, each able to parse a somewhat different set of languages:

| | |
|----------------|---|
| <i>SLR</i> | small tables, simple languages |
| <i>LR(1)</i> | large tables, more languages |
| <i>LALR(1)</i> | same size tables as <i>SLR</i> , but more languages (CUP uses these) |

LR parsers have more information available than *LL* parsers when choosing a production:

- *LR* knows everything derived from r.h.s. plus k lookahead symbols.
- *LL* just knows k lookahead symbols into what’s derived from r.h.s.