CS321 Languages and Compiler Design I

Winter 2012

Lecture 9

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.
- Can parse a larger set of languages than top-down.

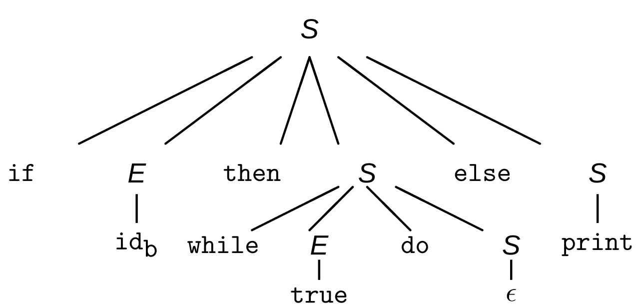
Bottom-up Parse Example

$S \rightarrow$ `if` E `then` S `else` S | `while` E `do` S | `print` | ϵ

$E \rightarrow$ `true` | `false` | `id`

`if` $\mathtt{id_b}$ `then while true do else print`

Parse Tree:



Left-most vs. Right-most derivations

S

$\Rightarrow_{lm}$ `if` E `then` S `else` S

$\Rightarrow_{lm}$ `if` $\mathtt{id_b}$ `then` S `else` S

$\Rightarrow_{lm}$ `if` $\mathtt{id_b}$ `then while` E `do` S `else` S

$\Rightarrow_{lm}$ `if` $\mathtt{id_b}$ `then while true do` S `else` S

$\Rightarrow_{lm}$ `if` $\mathtt{id_b}$ `then while true do else` S

$\Rightarrow_{lm}$ `if` $\mathtt{id_b}$ `then while true do else print`

$\Leftarrow_{rm}$ `if` E `then while true do else print`

$\Leftarrow_{rm}$ `if` E `then while` E `do else print`

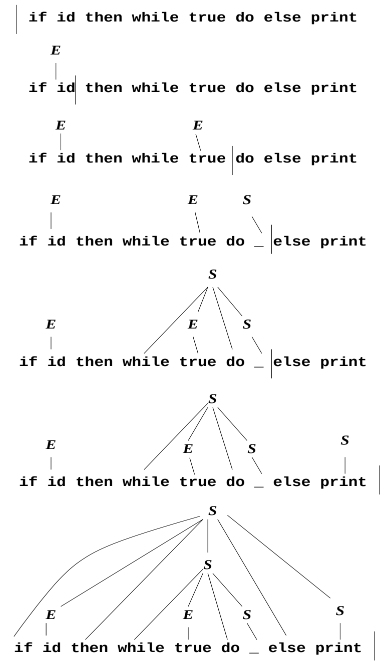
$\Leftarrow_{rm}$ `if` E `then while` E `do` S `else print`

$\Leftarrow_{rm}$ `if` E `then` S `else print`

$\Leftarrow_{rm}$ `if` E `then` S `else` S

$\Leftarrow_{rm}$ S

BOTTOM-UP PARSE



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 right-hand sides.

Example:

$$S \rightarrow aAcBe$$
$$A \rightarrow Ab \mid b$$
$$B \rightarrow d$$

"Right sentential form"	Reduction
$a\underline{b}bcde$	
$a\underline{Ab}cde$	$A \rightarrow b$
$aAc\underline{d}e$	$A \rightarrow Ab$
$\underline{aAcBe}$	$B \rightarrow d$
S	$S \rightarrow aAcBe$

Steps correspond to a right-most derivation in reverse.

HANDLES

We must choose the production to use wisely!

We don't always making progress by reducing with a production even when its right-hand sides match the input.

Example:

$a\underline{b}bcde$	
$aA\underline{b}cde$	$A \rightarrow b$
$aAAcde$	$A \rightarrow b$

Stuck!

A **handle** is a substring that

- is the right-hand side of some production; **and**
- whose replacement by the production's left-hand side is a (reverse) step in a rightmost derivation.

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

Handles, Formally

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

$$S \stackrel{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta \mid w$$

For example grammar, if current right-sentential form is

$$a\underline{Ab}\mid cde$$

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

Note that w never contains non-terminals.

Handle Pruning

Idea: Keep removing handles, replacing them with corresponding left-hand side of production, until we reach S.

Another example:

$E \rightarrow E\texttt{+}E \mid E\texttt{*}E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$

Right-sentential form	Handle	Reducing production
$\underline{\texttt{a}}\texttt{+b*c}$	a	$E \rightarrow \texttt{id}$
$E\texttt{+}\underline{\texttt{b}}\texttt{*c}$	b	$E \rightarrow \texttt{id}$
$E\texttt{+}E\texttt{*}\underline{\texttt{c}}$	c	$E \rightarrow \texttt{id}$
$E\texttt{+}\underline{E\texttt{*}E}$	$E\texttt{*}E$	$E \rightarrow E\texttt{*}E$
$\underline{E\texttt{+}E}$	$E\texttt{+}E$	$E \rightarrow E\texttt{+}E$
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 textbook on "Building LR(1)

tables").

Shift-reduce Parsing

Happily, we can use parser **generators** that compute the handles for us.

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

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 | E∗E | (E) | 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

Gives $E \Rightarrow_{rm} E$+$E \Rightarrow_{rm} E$+E∗$E \Rightarrow_{rm} E$+E∗c $\Rightarrow_{rm} E$+b∗c $\Rightarrow_{rm}$ a+b∗c.

Why not $E \Rightarrow_{rm} E$∗$E \Rightarrow_{rm} E$∗c $\Rightarrow_{rm} E$+E∗c $\Rightarrow_{rm} E$+b∗c $\Rightarrow_{rm}$ a+b∗c ??

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:

stmt	→	id(*expr*)	(a(i) is procedure call)
expr	→	id(*expr*) | id	(a(i) is array subscript)

Stack	Action
$...a(i)	Reduce by ??

Should we reduce to *stmt* or to *expr* ? Need to know the type of a: is it an array or a function? This information must flow from declaration of a to this use, typically via a symbol table.

LR PARSING

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

- **L** stands for "**L**eft-to-right scan of input."
- **R** stands for "**R**ightmost 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.

LR PARSER ENGINE

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.

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

LR GRAMMARS

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:

SLR	small tables, simple languages
$LR(1)$	large tables, more languages
$LALR(1)$	same size tables as SLR, but more languages (`CUP` uses these)

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

- $LR(k)$ knows everything derived from r.h.s. plus k lookahead symbols.
- $LL(k)$ just knows k lookahead symbols into what's derived from r.h.s.