Chapter 4: LR Parsing

Some definitions

Recall

For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a *sentential form*

- If $\alpha \in V_t^*$, then α is called a *sentence* in $L(G)$
- Otherwise it is just a sentential form (not a sentence in $L(G)$)

A *left-sentential form* is a sentential form that occurs in the leftmost derivation of some sentence.

A *right-sentential form* is a sentential form that occurs in the rightmost derivation of some sentence.

Bottom-up parsing

Goal:

> *Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.*

The parser repeatedly matches a *right-sentential* form from the language against the tree's upper frontier.

At each match, it applies a *reduction* to build on the frontier:

- each reduction matches an upper frontier of the partially built tree to the RHS of some production
- each reduction adds a node on top of the frontier

The final result is a rightmost derivation, in reverse.

Example

Consider the grammar

1	S	$\rightarrow$	$\mathtt{a}AB\mathtt{e}$
2	A	$\rightarrow$	$A\mathtt{bc}$
3		$\mid$	$\mathtt{b}$
4	B	$\rightarrow$	$\mathtt{d}$

and the input string `abbcde`

Prod'n.	Sentential Form
3	$\mathtt{a}\boxed{\mathtt{b}}\mathtt{bcde}$
2	$\mathtt{a}\boxed{A\mathtt{bc}}\mathtt{de}$
4	$\mathtt{a}A\boxed{\mathtt{d}}\mathtt{e}$
1	$\boxed{\mathrm{a}AB\mathrm{e}}$
–	S

The trick appears to be scanning the input and finding valid sentential forms.

Handles

What are we trying to find?

A substring α of the tree's upper frontier that

> matches some production $A \to \alpha$ where reducing α to A is one step in the reverse of a rightmost derivation

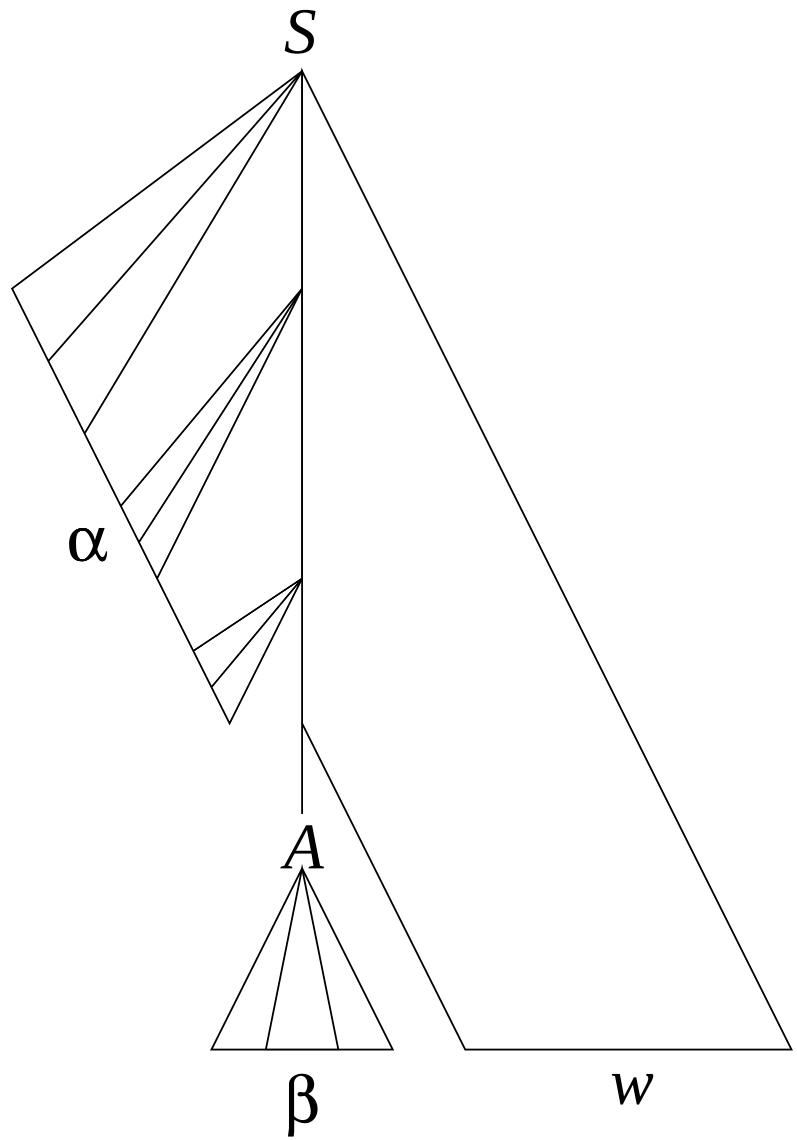
We call such a string a *handle*.

Formally:

> a *handle* of a right-sentential form γ is a production $A \to \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ
>
> i.e., if $S \Rightarrow^*_{\mathrm{rm}} \alpha A w \Rightarrow_{\mathrm{rm}} \alpha\beta w$ then $A \to \beta$ in the position following α is a handle of $\alpha\beta w$

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

Handles



The handle $A \to \beta$ in the parse tree for $\alpha\beta w$

Handles

Theorem:

> If G is unambiguous then every right-sentential form has a unique handle.

Proof: (by defi nition)

1. G is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to take γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position k at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $A \rightarrow \beta$

Example

The left-recursive expression grammar (*original form*)

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\mid$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\mid$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\texttt{num}$
9		$\mid$	$\texttt{id}$

Prod'n.	Sentential Form
–	$\langle\text{goal}\rangle$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$
7	$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle} * \texttt{id}$
8	$\langle\text{expr}\rangle - \underline{\texttt{num}} * \texttt{id}$
4	$\underline{\langle\text{term}\rangle} - \texttt{num} * \texttt{id}$
7	$\underline{\langle\text{factor}\rangle} - \texttt{num} * \texttt{id}$
9	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

Handle-pruning

The process to construct a bottom-up parse is called *handle-pruning*.

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

we set i to n and apply the following simple algorithm

`for i =` n `downto 1`

1. `find the handle` $A_i \rightarrow \beta_i$ `in` γ_i
2. `replace` β_i `with` A_i `to generate` γ_{i-1}

This takes $2n$ *steps, where* n *is the length of the derivation*

Stack implementation

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $

2. Repeat until the top of the stack is the goal symbol and the input token is $

   a) *fi nd the handle*
   if we don't have a handle on top of the stack, *shift* an input symbol onto the stack

   b) *prune the handle*
   if we have a handle $A \rightarrow \beta$ on the stack, *reduce*

      i) pop $|\beta|$ symbols off the stack

      ii) push A onto the stack

Example: back to $\mathtt{x} - 2 * \mathtt{y}$

1	⟨goal⟩	::=	⟨expr⟩
2	⟨expr⟩	::=	⟨expr⟩ + ⟨term⟩
3		|	⟨expr⟩ − ⟨term⟩
4		|	⟨term⟩
5	⟨term⟩	::=	⟨term⟩ ∗ ⟨factor⟩
6		|	⟨term⟩/⟨factor⟩
7		|	⟨factor⟩
8	⟨factor⟩	::=	num
9		|	id

Stack	Input	Action
$	id − num ∗ id	shift
$id	− num ∗ id	reduce 9
$⟨factor⟩	− num ∗ id	reduce 7
$⟨term⟩	− num ∗ id	reduce 4
$⟨expr⟩	− num ∗ id	shift
$⟨expr⟩ −	num ∗ id	shift
$⟨expr⟩ − num	∗ id	reduce 8
$⟨expr⟩ − ⟨factor⟩	∗ id	reduce 7
$⟨expr⟩ − ⟨term⟩	∗ id	shift
$⟨expr⟩ − ⟨term⟩ ∗	id	shift
$⟨expr⟩ − ⟨term⟩ ∗ id		reduce 9
$⟨expr⟩ − ⟨term⟩ ∗ ⟨factor⟩		reduce 5
$⟨expr⟩ − ⟨term⟩		reduce 3
$⟨expr⟩		reduce 1
$⟨goal⟩		accept

1. *Shift until top of stack is the right end of a handle*

2. *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 accept

Shift-reduce parsing

Shift-reduce parsers are simple to understand

A shift-reduce parser has just four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack
2. *reduce* — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal LHS
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine

The key problem: to recognize handles (not covered in this course).

LR(k) grammars

Informally, we say that a grammar G is LR(k) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation,

1. *isolate the handle of each right-sentential form*, and
2. *determine the production by which to reduce*

by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i.

LR(k) grammars

Formally, a grammar G is LR(k) iff.:

1. $S \Rightarrow^*_{\text{rm}} \alpha A w \Rightarrow_{\text{rm}} \alpha\beta w$, and
2. $S \Rightarrow^*_{\text{rm}} \gamma B x \Rightarrow_{\text{rm}} \alpha\beta y$, and
3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

$\Rightarrow \alpha A y = \gamma B x$

i.e., Assume sentential forms $\alpha\beta w$ and $\alpha\beta y$, with common prefix $\alpha\beta$ and common k-symbol lookahead $\text{FIRST}_k(y) = \text{FIRST}_k(w)$, such that $\alpha\beta w$ reduces to $\alpha A w$ and $\alpha\beta y$ reduces to $\gamma B x$.

But, the common prefix means $\alpha\beta y$ also reduces to $\alpha A y$, for the same result.

Thus $\alpha A y = \gamma B x$.

Why study LR grammars?

LR(1) grammars are often used to construct parsers.

We call these parsers LR(1) parsers.

- everyone's favorite parser
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

  LL(k)**:** recognize use of a production $A \to \beta$ seeing first k symbols of β

  LR(k)**:** recognize occurrence of β (the handle) having seen all of what is derived from β plus k symbols of lookahead

Left versus right recursion

Right Recursion:

- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left Recursion:

- works fine in bottom-up parsers
- limits required stack space
- left associative operators

Rule of thumb:

- right recursion for top-down parsers
- left recursion for bottom-up parsers

Parsing review

Recursive descent

A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k)

An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

LR(k)

An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

The dilemmas:

- LL dilemma: pick $A \to b$ or $A \to c$?
- LR dilemma: pick $A \to b$ or $B \to b$?