Parsing review

Top-down parsers

- start at the root of derivation tree and fill in
- choosing production for nonterminal is the key problem
- LL (predictive) parsers use lookahead to choose production

Bottom-up parsers

- start at leaves and fill in
- choosing right-hand side (*rhs*) of production is the key problem
- LR parsers use current stack and lookahead to choose production

LR(k) parsers are more powerful than LL(k) parsers because they can see the entire *rhs* before choosing a production.

Some definitions

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

- If α contains only *terminal symbols*, α is a sentence in $L(G)$.
- If α contains one or more non-terminals, 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 the right-hand side *rhs* of a production against a substring in the current right-sentential form.

At each match, it applies a *reduction* to build the parse tree.

- each reduction replaces the matched substring with the nonterminal on the left-hand side *lhs* of the production
- each reduction adds an internal node to the current parse tree
- the result is another right-sentential form

The final result is a rightmost derivation, in reverse.

Example

Consider the grammar

1	⟨goal⟩	::=	`a` ⟨A⟩ ⟨B⟩ `e`
2	⟨A⟩	::=	⟨A⟩ `b` `c`
3		|	`b`
4	⟨B⟩	::=	`d`

and the input string `abbcde`.

Prod'n.	Sentential Form	Handle
—	`abbcde`	3,2
3	`a`⟨A⟩`bcde`	2,4
2	`a`⟨A⟩`de`	4,3
4	`a`⟨A⟩⟨B⟩`e`	1,4
1	⟨goal⟩	—

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

Handles

We trying to find a substring α of the current right-sentential form where:

- α matches some production $A ::= \alpha$
- reducing α to A is one step in the reverse of a rightmost derivation.

We will call such a string a *handle*.

Formally,

- a *handle* of a right-sentential form γ is a production $A ::= \beta$ and a position in γ where β may be found.
- If $(A ::= \beta, k)$ is a handle, then replacing the β in γ at position k with A produces the previous right-sentential form in a rightmost derivation of γ.

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

Handles

Provable fact:

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

Proof: *(by definition)*

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

Example

The left-recursive expression grammar

(*original form*, before left factoring)

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`

Prod'n.	Sentential Form	Handle
—	⟨goal⟩	—
1	⟨expr⟩	1,1
3	⟨expr⟩ `-` ⟨term⟩	3,3
5	⟨expr⟩ `-` ⟨term⟩ `*` ⟨factor⟩	5,5
9	⟨expr⟩ `-` ⟨term⟩ `* id`	9,5
7	⟨expr⟩ `-` ⟨factor⟩ `* id`	7,3
8	⟨expr⟩ `- num * id`	8,3
4	⟨term⟩ `- num * id`	4,1
7	⟨factor⟩ `- num * id`	7,1
9	`id - num * id`	9,1

Handle-pruning

The process we use 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

```
do i = n to 1 by -1
  (1) find the handle (A_i ::= β_i, k_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

Shift-reduce parsing

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 "*end of file*"

   a) *find 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 $(A ::= \beta, k)$ on the stack, *reduce*

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

      ii) push A onto the stack

Back to "`x - 2 * y`"

Stack	Input	Handle	Action
$	`id - num * id`	*none*	shift
$`id`	`- num * id`	9,1	reduce 9
$⟨factor⟩	`- num * id`	7,1	reduce 7
$⟨term⟩	`- num * id`	4,1	reduce 4
$⟨expr⟩	`- num * id`	*none*	shift
$⟨expr⟩ `-`	`num * id`	*none*	shift
$⟨expr⟩ `-` `num`	`* id`	8,3	reduce 8
$⟨expr⟩ `-` ⟨factor⟩	`* id`	7,3	reduce 7
$⟨expr⟩ `-` ⟨term⟩	`* id`	*none*	shift
$⟨expr⟩ `-` ⟨term⟩ `*`	`id`	*none*	shift
$⟨expr⟩ `-` ⟨term⟩ `*` `id`		9,5	reduce 9
$⟨expr⟩ `-` ⟨term⟩ `*` ⟨factor⟩		5,5	reduce 5
$⟨expr⟩ `-` ⟨term⟩		3,3	reduce 3
$⟨expr⟩		1,1	reduce 1
$⟨goal⟩		*none*	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
- have a simple, table-driven, *shift-reduce* skeleton
- encode grammatical knowledge in tables

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

LR(1) grammars

Informally, we say that a grammar G is LR(1) 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 1 symbol beyond the right end of the handle of γ_i.

Formality will come later.

Why study LR(1) grammars?

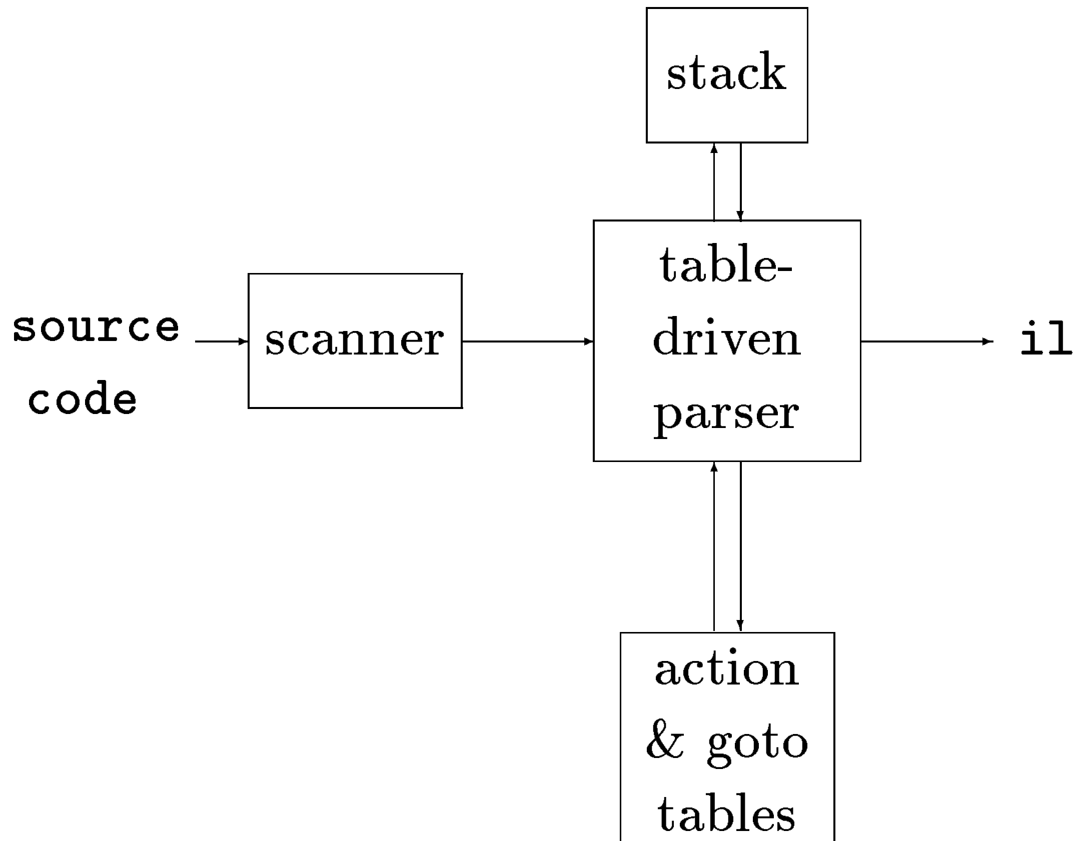
LR(1) grammars are often used to construct shift-reduce parsers.

We call these parsers LR(1) parsers.

- everyone's favorite parser (*EFP*)
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce 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 LL (predictive) parsers

Table-driven *LR*(1) parsing

A table-driven LR(1) parser looks like



Stack two items per state: *state* and *symbol*

Table building tools are readily available (`yacc`)

We'll learn how to build these tables by hand!

LR(1) parsing

The skeleton parser:

```
token = next_token()
repeat forever
  s = top of stack

  if action[s,token] = "shift s_i"' then
    push token
    push s_i
    token = next_token()

  else if action[s,token] =
       "reduce A ::= β" then
    pop 2 * | β | symbols
    s = top of stack
    push A
    push goto[s,A]

  else if action[s, token] = "accept" then
    return

  else error()
```

This takes k shifts, l reduces, and 1 accept, where k is the length of the input string and l is the length of the reverse rightmost derivation.

Note: Equivalent to Figure 4.30, Aho, Sethi, and Ullman

Example tables

	ACTION				GOTO		
	id	+	*	$	<expr>	<term>	<factor>
S_0	s4	—	—	—	1	2	3
S_1	—	—	—	acc	—	—	—
S_2	—	s5	—	r3	—	—	—
S_3	—	r5	s6	r5	—	—	—
S_4	—	r6	r6	r6	—	—	—
S_5	s4	—	—	—	7	2	3
S_6	s4	—	—	—	—	8	3
S_7	—	—	—	r2	—	—	—
S_8	—	r4	—	r4	—	—	—

The Grammar

1	<goal>	::=	<expr>
2	<expr>	::=	<term> + <expr>
3		|	<term>
4	<term>	::=	<factor> * <term>
5		|	<factor>
6	<factor>	::=	`id`