COMS W4115
Programming Languages and Translators
Lecture 10: Bottom-Up Parsing
February 25, 2013

Lecture Outline

1. Bottom-up parsing
2. LR(1) parsing
3. Constructing a simple LR(1) parser
4. DFA for viable prefixes

1. Bottom-up Parsing

- Bottom-up parsing can be viewed as trying to find a rightmost derivation in reverse for an input string.
- A *handle* is a rightmost substring in right-sentential form that matches the body of a production and whose reduction by that production represents one step in the reverse of the rightmost derivation.
  - Consider the grammar *G*:

```
(1) S â†’ S ( S )
(2) S â†’ Îµ
```

- The handles in a rightmost derivation for the input string `( ) ( )`:

```
S â‡’ S ( S )      // handle is S ( S )
  â‡’ S ( )        // handle is the empty string between ( )
  â‡’ S ( S ) ( )  // handle is S ( S )
  â‡’ S ( ) ( )    // handle is the empty string between first ( )
  â‡’ ( ) ( )      // handle is the empty string prefix
```

- *Shift-reduce parsing* is a form of bottom-up parsing in which we shift terminal symbols of the string to be parsed onto a stack until a handle appears on top of the stack. We then replace the handle by the nonterminal symbol on the left-hand side of the associated production (this is a "reduce" action). We keep repeating this process until we have reduced the input string to the start symbol of the grammar. This process simulates the reverse of a rightmost derivation for the input string. Thus, we can think of shift-reduce parsing as "handle pruning."

2. LR(1) Parsing

- Model of an LR(1) parser (Fig. 4.35).
- "L" means left-to-right scanning of the input, the "R" means constructing a rightmost derivation in reverse, and the "1" means one symbol of lookahead in making parsing decisions.
- LR parsing table for *G*:

State	**Action**			**Goto**
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

r2 means reduce the handle on top of the stack by production (2) `S â†’ Îµ`.
s2 means shift the input symbol on the stack and then push state 2 on top of the stack.
acc means accept and stop parsing.
Goto[0,S] = 1 means push state 1 on top of the stack after reducing a handle to the nonterminal S in state 0.
A blank entry means report a syntax error.

- Moves made by an LR(1) parser on input `( ) ( )` [Alg. 4.44].

```
Stack       Input          Action
0           ()()$    reduce by (2) S â†’ Îµ; push state 1 on stack
0S1         ()()$    shift ( on stack; push state 2 on stack
0S1(2        )()$    reduce by (2) S â†’ Îµ and push state 3
0S1(2S3      )()$    shift ( and push state 2
0S1(2S3)4     ()$    reduce by (1) S â†’ S(S) and push state 1
0S1           ()$    shift ( and push state 2
0S1(2          )$    reduce by (2) S â†’ Îµ and push state 3
0S1(2S3        )$    shift ) and push state 4
0S1(2S3)4       $    reduce by (1) S â†’ S(S) and push state 1
0S1             $    accept
```

- Note that an LR parser is a shift-reduce parser that traces out a rightmost derivation in reverse.

3. Constructing a Simple LR(1) Parsing Table for a Grammar

- An *LR*(0) *item* of a grammar is a production of the grammar with a dot at some position of the right side. E.g., `S â†’ Â·S(S)`, `S â†’ SÂ·(S)`, or `S â†’ S(S)Â·`.
- We will use two functions to construct the sets of items for a grammar:
  - *closure*(*I*), where *I* is a set of items, is the set of items constructed by the following two rules:

1. Initially, put every item in *I* into *closure*(*I*).
2. If *A* â†’ Î±Â·*B*Î² is in *closure*(*I*) and *B â†’ Î³* is a production, then add the item *B* â†’ Â·Î³ to *closure*(*I*) if it is not already there. Keep repeating this step until no more new items can be added to *I*.

- *goto*(*I*, *X*), where *I* is a set of items and *X* is a grammar symbol, is the closure of the set of all items *A* â†’ Î±*X*Â·Î² where *A* â†’ Î±Â·*X*Î² is in *I*.
- An *augmented* grammar *G′* is one to which we have added a new starting production *S′* â†’ *S* where *S* is the start symbol of the given grammar *G*. Reducing by the new starting production signals acceptance of the input string being parsed. We will always augment a grammar when we construct an SLR parsing table for it.
- The sets-of-items construction
- Input: An augmented grammar *G′*.
- Output: *C*, the canonical collection of sets of LR(0) items for *G′*.
- Method:

```
I_0 = closure({[S' â†’ Â·S]});
C = {I_0};
repeat
  for each set of items I in C and grammar symbol X such that
    goto(I,X) is not empty and not in C do
      add goto(I,X) to C;
until no more sets of items can be added to C;
```

- Example: Given the augmented grammar *G′*

```
S' â†’ S
S  â†’ S(S)
S  â†’ Îµ
```

C, the canonical collection of sets of LR(0) items for *G′*, is

```
I_0: S' â†’ Â·S
    S â†’ Â·S(S)
    S â†’ Â·

I_1: S' â†’ SÂ·
    S â†’ SÂ·(S)

I_2: S â†’ S(Â·S)
    S â†’ Â·S(S)
    S â†’ Â·

I_3: S â†’ S(SÂ·)
    S â†’ SÂ·(S)
```

I_4: `S' â†' S(S)Â·`

- Algorithm to construct the SLR(1) parsing table from `C`, the canonical collection of sets of LR(0) items for an augmented grammar *G'*
- Input: `C = {`I_0`, `I_1`, ... , `I_n`}`.
- Output: The SLR parsing table functions `action` and `goto`.
- Method:
  - State `i` and its `action` and `goto` functions are constructed from I_i as follows:
    - If item [Aâ†' Î±Â·aÎ²] is in I_i and `goto(`I_i`, a) = `I_j, then add "`shift j`" to `action[i, a]`. Here a is a terminal.
    - If item [`A â†' Î±Â·`] is in I_i, then add "`reduce A â†' Î±`" to `action[i, a]` for all a in FOLLOW(A). Here A cannot be `S'`.
    - If item [`S' â†' SÂ·`] is in I_i, then add "`accept`" to `action[i, $]`.
  - If `goto(`I_i`, A) = `I_j, then in the parsing table set `goto[i, A] = j`.
  - The initial state of the parser is constructed from the set of items containing [`S' â†' Â·S`].
- Notes:
  - If each parsing table entry has at most one action, then the grammar is said to be *SLR(1)*. If any entry has more than one action, then the algorithm fails to produce a parser.
  - All undefined entries are made `error`.
- Example: the LR parsing table above is an SLR(1) parsing table for the balanced-parentheses grammar.

4. DFA for Viable Prefixes

- A *viable prefix* is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.
- The shift and goto functions of the canonical collection of sets of LR(0) items for a grammar *G* define a DFA that recognizes the viable prefixes of *G*.
- An item [`A â†' Î²Â·Î³`] is *valid* for a viable prefix `Î±Î²` if there is a rightmost derivation from `S'` to `Î±`Aw to `Î±Î²Î³w`.

5. Practice Problems

Consider the following grammar G:

```
(1) S â†' S S +
(2) S â†' S S *
(3) S â†' a
```

- Construct a rightmost derivation and parse tree for the input string `aaa*+$`.
- Show the handle in each sentential form in the derivation.
- Construct the canonical collection of sets of LR(0) items for the augmented grammar.
- Construct an SLR(1) parsing table for G.
- Show how your SLR(1) parser processes the input string `aaa*+$`.

6. Reading

- ALSU, Sects. 4.5, 4.6.

---

aho@cs.columbia.edu