COM SCI 132 Week 7

Aidan Jan

May 15, 2024

LR Parsing

Review

Recall that

- 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

The 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 -> aABe
2 | A -> Abc
3 |    |  b
4 | B -> d
```

and the input string `abbcde`.

Prod'n	Sentential Form
3	a $\boxed{\texttt{b}}$ bcde
2	a $\boxed{\texttt{Abc}}$ de
4	aA $\boxed{\texttt{d}}$ e
1	$\boxed{\texttt{aABe}}$
-	S

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

Handles

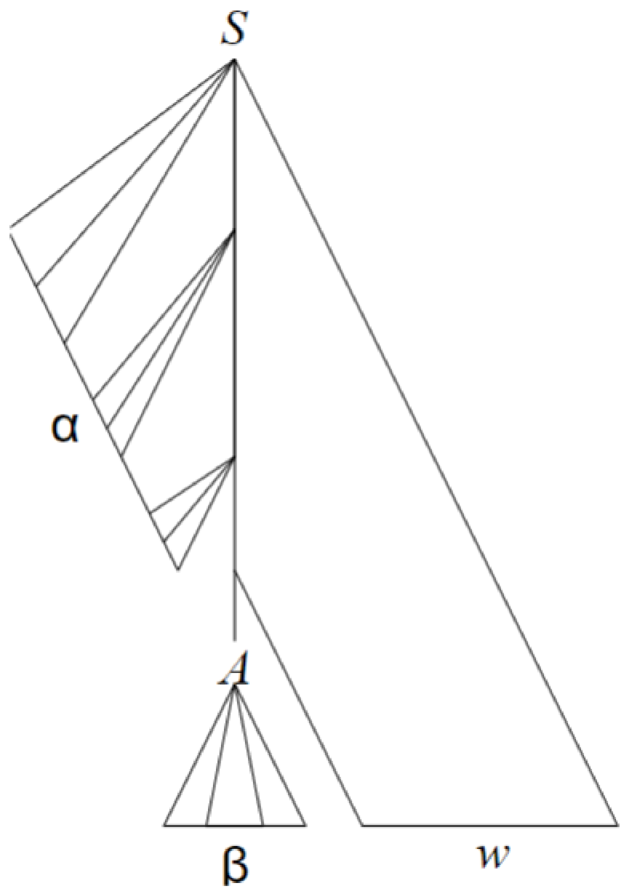
What are we trying to find?

- 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 $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$, 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.



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

Theorem:

- 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 $A \to \beta$ applied to take γ_{i-1} to γ_1
3. $\Rightarrow$ a unique position k at which $A \to \beta$ is applied
4. $\Rightarrow$ a unique handle $A \to \beta$

Example

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

    `find the handle` $A_i \to \beta_i$ `in` γ_i

    `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) *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 $A \to \beta$ on the stack, *reduce*.
      i. pop $|\beta|$ symbols off the stack
      ii. push A onto the stack

Example

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

Stack	Input	Action
$\$$	id − num ∗ id	shift
$\$\underline{\texttt{id}}$	− num ∗ id	reduce 9
$\$\underline{\langle\text{factor}\rangle}$	− num ∗ id	reduce 7
$\$\underline{\langle\text{term}\rangle}$	− num ∗ id	reduce 4
$\$\langle\text{expr}\rangle$	− num ∗ id	shift
$\$\langle\text{expr}\rangle -$	num ∗ id	shift
$\$\langle\text{expr}\rangle - \underline{\texttt{num}}$	∗ id	reduce 8
$\$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle}$	∗ id	reduce 7
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle$	∗ id	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle *$	id	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$		reduce 9
$\$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$		reduce 5
$\$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$		reduce 3
$\$\underline{\langle\text{expr}\rangle}$		reduce 1
$\$\langle\text{goal}\rangle$		accept

1. Shift until top of stack is the right end of a handle
2. Find the left end of the handle and reduce

For this example: 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 the 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-sequential form in the derivation,

1. isolate the handle of each right-sequential 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.
Formally, a grammar G is LR(k) if and only if:

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

i.e., Assume sentential forms $\alpha\beta w$ and $\alpha\beta y$ with common prefix $\alpha\beta$ and common k-symbol lookahead $\texttt{FIRST}_k(\mathrm{y}) = \texttt{FIRST}_k(\mathrm{w})$, 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 operatives

Rule of thumb:

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

Parsing Review

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

Type Checking Generices

A *generic* is a type that has to be parameterized. For example, the compiler needs to be told which object is stored in a given List.

```
class List<A> extends Object {
    public <R> R accept(ListVisitor<R, A> f) {
        // ... do stuff
        return this.accept(f);
    }
}

class ListNull<A> extends List<A> {
    public<R> R accept(ListVisitor<R, A> f) {
        return f.visit(this)
    }
}
```

The above program above will in fact, type check.

Example

Grammar:

(*Class*) $\texttt{L} ::= \texttt{class C<}\bar{x}\texttt{> extends N \{}\bar{s}\bar{f}\texttt{; K }\bar{M}\texttt{\}}$

(*Constructor*) $\texttt{K} ::= \texttt{C(}\bar{T}\bar{f}\texttt{) \{super(f); this.}\bar{f}\texttt{ = }\bar{f}\texttt{\}}$

(*Method*) $\texttt{M} ::= \texttt{<}\bar{Y}\texttt{> U m(}\bar{U}\bar{x}\texttt{) \{ return e; \}}$

(*Expression*) $\texttt{e} ::= \texttt{x | e.f | e.<}\bar{v}\texttt{> m(}\bar{e}\texttt{) | new N(}\bar{e}\texttt{)}$

(*Type*) $\texttt{S, T, U, V} ::= \texttt{X | N}$

$\texttt{N} ::= \texttt{C<}\bar{T}\texttt{>}$

Type Rules:

$$\Gamma + e \, : \, T$$

where

- Γ represents the type environment

- e represents an expression
- T represents a type

$$\frac{\Gamma \vdash e \,:\, T \qquad \texttt{fields}(T) = \bar{T}\bar{f}}{\Gamma \vdash e.f_i \,:\, T_i} \tag{1}$$

$$\frac{\texttt{fields}(N) = \bar{T}\bar{f} \qquad \Gamma + \bar{e} \,:\, \bar{S} \qquad \bar{S} \leq \bar{T}}{\Gamma \vdash \texttt{new } N(\bar{e}) \,:\, N} \tag{2}$$

$$\frac{\Gamma + e_0 \,:\, T_0 \qquad \texttt{mtype}(m, T_0) = < \bar{Y} > \bar{U} \to U \qquad \bar{S} \leq [\bar{V}/\bar{Y}]\bar{U}}{\Gamma \vdash e_0. < \bar{V} > m(\bar{e}) \,:\, [\bar{V}/\bar{Y}]U} \tag{3}$$

- `mtype` refers to method type
- $< \bar{Y} > \bar{U}$ are the types of formal parameters and formal parameters, respectively.
- Type expression (3) is created via substitution with (1) and (2).
- $\bar{S} \leq \bar{T}$ is to check class extends; S and T are in a class-superclass relationship.
- Substitution Notation: $[\bar{V}/\bar{X}]$ means the same thing as $[\bar{X} := \bar{V}]$
- `mtype` is a placeholder function for method type checking

At this point, we have type checked methods and classes. However, we still have to do the following:

- Override methods
- Fields
- Subtyping
- `mtype`

Subtyping

Recall that our types:

(*Type*) `S, T, U, V` ::= `X | N`
`N` ::= `C<`$\bar{T}$`>`

Subtyping rules:

$$T \leq T$$

(one type is a subtype of another type)

$$\frac{S \leq T \qquad T \leq U}{S \leq U} \tag{4}$$

$$\frac{\texttt{class } C < \bar{x} > \texttt{extends } N\{\ldots\}}{C < \bar{T} > \leq [\bar{T}/\bar{X}]N} \tag{5}$$

- (5) states that if C is a use of N, as in, N is a parent class to C, then a use of T in C (as a parameter of C) is a subtype of the use of T in N.
  - In simpler terms, C parameterized with T would be a subtype of N parameterized with T, because C is a subtype of N.

Fields

$$\frac{\texttt{class}\ C < \bar{x} > \texttt{extends}\ N\{\bar{S}\bar{f};\dots\} \qquad \texttt{fields}\ ([\bar{T}/\bar{x}]N) = \bar{V}\bar{g}}{\texttt{fields}(C < \bar{T} >) = \bar{U}\bar{g}, [\bar{T}/\bar{X}]\bar{f}} \tag{6}$$

- $\bar{S}\bar{f}$ is one parameter, where S is the type, f is the field.

`mtype` (Method Typing)

$$\frac{\texttt{class}\ C < \bar{X} >\ \texttt{extends}\ N\{\dots\bar{M}\} \qquad (< \bar{Y} >\ U\, m(\bar{U}\ \bar{x})\{\texttt{return}\ e\}) \in \bar{M}}{\texttt{mtype}(m, C < \bar{T} >) = [\bar{T}/\bar{X}](< \bar{Y} > \bar{U} \to U)} \tag{7}$$

$$\frac{\texttt{class}\ C < \bar{x} >\ \texttt{extends}\ N\{\dots\bar{M}\} \qquad m \notin \bar{M}}{\texttt{mtype}\ (m, C < \bar{T} >) = \texttt{mtype}\ (m, [\bar{T}/\bar{X}]\bar{N})} \tag{8}$$

- $\bar{M}$ are the methods of the class C.

Method Overriding

$$\frac{\texttt{If}\ \{\texttt{mtype}(m, N) = < \bar{z} > \bar{U} \to U\}\ \texttt{Then}\ \{\bar{T} = [Y/\bar{Z}]\bar{U} \qquad T = [\bar{Y}/\bar{Z}]U\}}{\texttt{override}(m, N, < \bar{Y} > \bar{T} \to T)} \tag{9}$$

In the override statement...

- m = method name
- N = superclass of the current class
- $< \bar{Y} > \bar{T} \to \bar{T}$ = method header