Programming Languages: Lecture 11 Introduction to Parsing

Rishabh Dhiman

27 January 2022

1 Syntax Diagrams

Can read on this in the slides later.

2 Parsing

Since

- parsing requires checking whether a given token stream *conforms* to the rules of the grammar and
- sicne a CFG may generate an infinite number of different strings.

Any parsing method should be guided by the given input (token) string, so that a deterministic strategy may be evolved.

Aim of parsing is to transform a linear sequence of tokens into a derivation tree (parse tree).

Design of the PL should enable a linear-time parsing algorithm.

Most programming languages are designed to ensure a linear-time parsing algorithm.

3 Parsing Methods

Two kinds of parsing methods

1. Top-down parsing. Try to *generate* the given input sentence form the start symbol of the grammar by applying the production rules.
2. Bottum-up parsing. Try to *reduce* the given input sentence to the start symbol by applying the rules in *reverse*.

- Top-down predictive parsing using a LL(1) grammar. Recursive-descent parsing posisble when the implementation language supports recursion.

- Botton-up parsing using a LR(1) or LALR(1) parser.
- LL(1) = *L*eft-to-Right scanning of tokens with *L*eftmost derivation with *1* token look-ahead
- LL(1) = *L*eft-to-Right scanning of tokens with (reverse of) *R*ightmost derivation with *1* token look-ahead
- LALR(1) = *L*ook *A*head LR(1) parser with *1* token lookeahead. Restricted LR(1) with 'LR(1) items'.

4 Top-Down Parsing

- Try to generate the given input sentence from the start symbol of the grammar by applying the production rules.
- Not the most general.
- But most modern high-level PLs are designed to be efficiently parsed by this method.
- *Recursive-descent* is the most frequently employed technique when language $\mathcal{C}$ in which the complier is written, supports recursion.

5 Recursive-Descent Parsing

- Suitable for grammars that are LL(1) parseable.
- A set of (mutually) recursive procedures.
- Has a single procedure/function for each non-terminal symbol.
- Allows for syntax errors to be pinpointed more accurately than most other parsing methods.

5.1 Caveats with RDP

They cannot handle situations of such forms,

- Any left-recursion in the grammar can lead to infinite recursive calls during which no input token is consumed and there is no return from the recursion, ie, they shouldn't be of the form

  $$A \to A\alpha.$$

  This would result in an infinite recursion with no input token consumed, such a recursion is called a *direct left recursion*.

- You can generalize this to a sequence of production rules which leads to a production of the form $A \stackrel{*}{\Rightarrow} A\alpha$. This is called an *indirect left recursion.*
- For RDP to succeed without backtracking, for each input token and each non-terminal symbol there should be only one rule applicable.

  Example. *A set of productions of the form*

  $$A \to aB\beta \mid aC\gamma$$

  where B and C stand for different phrases would lead to non-determinism. The normal practice then would be to left-factor *the two productions by introducing a new non-terminal symbol A' and rewrite the rule as*

  $$\begin{aligned} A &\to aA', \\ A' &\to B\beta \mid C\gamma, \end{aligned}$$

  provided B and C generate terminal strings with different first symbols (otherwise more left-factoring needs to be performed).

Example (Left-Recursive Grammar)**.** *The following grammar is unambiguous and implements both left-associativity and precedence of operators,*

$$G = \langle \{E, T, D\}, \{a, b, -, /, (,)\}, P, E \rangle$$

whose productions are

$$\begin{aligned} E &\to E - T \mid T, \\ T &\to T/D \mid D, \\ D &\to a \mid b \mid (E). \end{aligned}$$

$T \to T/D \to T/a$ gives an indirect left recursion, and $E \to E - T$ is a direct left recursion.

We can get rid of indirect left-recursion as follows

$$\begin{aligned} E &\to TE' \\ E' &\to -TE' \mid \varepsilon \\ T &\to DT' \\ T' &\to /DT' \mid \varepsilon \\ D &\to a \mid b \mid (E). \end{aligned}$$

Now it is no lonegr left-recursive and may then be parsed by a recursive-descent parser.

5.2 Determinism in RDP

RDP can be deterministic only if,

- the input token lookahead uniquely determins the production to be applied,
- we need to define the *first* symbols that will be generated by each production.
- in the presense of ε productions, symbols that can *follow* a given non-terminal symbol also need to be specieifed.

Definition 1 (Nullability)**.**

A nonterminal symbol X is *nullable* if it can derive the empty string, ie, $X \stackrel{*}{\Rightarrow} \varepsilon$. The following algorithm computes $nullable(X)$ for each non-terminal symbol. For convenience, *nullable* is set to false for each terminal symbol.

Algorithm 1 (Nullable)**.** The weird-ass C++ code is

```
vector<bool> nullable(CFG G) {
        auto [N, T, P, S] = G;

        vector<bool> res(N.size() + T.size());
        for (auto a : T) res[a] = false;

        for (auto X : N) res[X] = ∃X → ε ∈ P;

        for (bool nullable_changed = true; nullable_changed; ) {
                nullable_changed = false;
                for (X → α1 ... αk ∈ P) { // this is a single production rule
                        if (res[α1] && res[α2] && ... && res[αk]) {
                        // all symbols are nullable
                                res[X] = true;
                                nullable_changed = true;
                        }
                }
        }
        return res;
}
```

We are NOT just checking if ε is reachable from X to ε if the CFG is viewed as a "graph".

Actually, how do you view a CFG as a graph? Need a hypergraph I think. // think about this dum-dum

Definition 2 (First)**.** $first(\alpha)$ is the set of terminal symbols that can be the first symbol of any string that α can derive, ie, $a \in first(\alpha)$ if and only if there exists a derivation $\alpha \stackrel{*}{\Rightarrow} ax$ for any string of terminals x.

Algorithm 2 (First). The weird-ass C++ code is

```
vector<vector<symbol>> FIRST(CFG G) {
        auto [N, T, P, S] = G;

        vector<vector<symbol>> first;
        for (auto a : T) first[a] = {a};

        for (auto X : N) first[X] = {};

        for (bool changed = true; changed; ) {
                changed = false;
                for (X → α_1 ... α_k ∈ P) { // this is a single production rule
                        for (int i = 1; i <= k; ++i) {
                                first[X].unite(first[α_i]);
                                if (first[X] changed) changed = true;
                                if (!nullable[α_i]) break;
                        }
                }
        }

        return first;
}
```

Definition 3 (Follows). $follows(X)$ is the set of terminal symbols a such that there exists a rightmost derivation of the form

$$S \stackrel{*}{\Rightarrow} \alpha X a \beta$$

for $\alpha, \beta \in (N \cup T)^*$, ie, $follow(X)$ is the set of all terminal symbols that can occur immediately to the right of X in a *rightmost* derivation.

// Come up with an algorithm for this yourself! (though hypernotes do have it)

5.3 Pragmatics

// There's a PASCAL code here, add it later.