Context-Free Grammars, Pushdown Automata, and Parsing Algorithms

Andrew Clifton

Department of Computer Science

California State University, Fresno

Fresno, CA 93740

December 8, 2015

`andyclifton@csufresno.edu`

Preliminaries: Sets in Haskell

Much of our work on context-free grammars and languages will require more sophisticated set-handling facilities than previously. In this section, we detail a few areas which may need more exposition.

Set normalization

As before, in order for many of our algorithms to work correctly, sets, when represented as lists, must be both duplicate-free and in some *normal form*, so that sets can be compared for (in)equality using the normal `==` and `/=` operators. We contiune to use the function `distinct`, given in figure 1, to both sort the elements of a set (imposing a normal form) and to eliminate any duplicates.

Note that while `Data.List.nub` runs in $O(n^2)$ due to the fact that it only requires its input to be `Eq`, `distinct` requires `Ord` and thus is able to run in $O(n \log n)$.

```
import Data.List (sort)

distinct :: Ord a => [a] -> [a]
distinct = dedup . sort
    where
        dedup [] = []
        dedup [x] = [x]
        dedup (x1:x2:xs) | x1 == x2   = dedup (x2:xs)
                         | otherwise   = x1 : dedup (x2:xs)
```

Figure 1: The `distinct` function

Note, of course, that when dealing with nested sets, all levels must be normalized from the inside out.

Building sets inductively in Haskell

Many of the algorithms we will describe build sets inductively. For example, the algorithm for determining the set of nullable variables in a grammar is specified as

$$\text{Base}\,\frac{(A \to \varepsilon) \in R}{A \in \mathsf{nullable}} \qquad \text{Induction}\,\frac{(A \to w) \in R \qquad w \in \mathsf{nullable}^*}{A \in \mathsf{nullable}}$$

(That is, a variable is nullable if either it is defined as just ε, or if its definition is a string of nullable variables.)

Although we could translate this directly into a recursive predicate testing membership in nullable:

```
is_nullable :: Grammar -> Var -> Bool
is_nullable (vs,s,rs) v = (v,[Nothing]) `elem` rs ||
                          or [ is_null_str d | (v',d) <- rs, v' == v]
  where
    is_null_str s = and [is_nullable c | Just c <- s]
```

this will be horribly inefficient as we are repeatedly testing the same variables for membership, without retaining any information. Instead, we want to actually *build* and return the set nullable, so that membership becomes a simple `elem`.

The ordinary method for building an inductive set algorithmically is to add elements to the set, each time testing to see whether the "new" set is different from the old. If it is, then the elements we just added have not been processed and we have more work to do. If it is not, then the new elements were already in the set, and thus at least this branch of the computation is finished. Of course, we only really need to process the "new" elements; there is no need to re-process existing elements. In fact, what we want is some kind of duplicate-free stack or queue.

(details to follow)

Context-Free Languages and Grammars

Normal Forms for Context-Free Grammars

Often, in order to use a grammar in some particular way, we will want the rules to have particular forms. We can transform grammars in various ways, without affecting the languages that they generate, if we are careful. In this section we will examine both methods for "safely" rewriting grammars into new forms, as well as develop the forms themselves that we are interested in. (Note that every transformation we will cover will be described in terms of transforming a grammar into a new one, rather than by destructive modification of an existing grammar.)

Non-recursive start symbol

The simplest transformation we can make is to ensure that the start variable is non-recursive, either directly or indirectly. Assuming that the existing start variable is S, we do this by simply adding a new start variable S' with a single rule

$$S' \to S$$

Since S is no longer the start variable, any recursion on it is acceptable, and because the only reference to S' is in the rule we just added, it is not recursive.

As an example, consider the grammar

$$\begin{aligned} S &\to \mathtt{a}S \mid \mathtt{b}A \\ A &\to b\mathtt{A} \mid \mathtt{b} \end{aligned}$$

To make this not have a recursive start variable, we simply add

$$\begin{aligned} S' &\to S \\ S &\to \mathtt{a}S \mid \mathtt{b}A \\ A &\to b\mathtt{A} \mid \mathtt{b} \end{aligned}$$

and we are done.

Although it should be obvious, we will also show that the language generated by the grammar has not changed. Any derivation $S \overset{*}{\Longrightarrow} u$ in the original grammar can be made into a valid derivation in the new grammar by simply doing $S' \Rightarrow S \overset{*}{\Longrightarrow} u$.

Eliminating ε-rules

An ε rule is one that can derive the empty string ε. We wish to eliminate ε rules so that derivations like $uVw \overset{*}{\Longrightarrow} uw$ (where $V \overset{*}{\Longrightarrow} \varepsilon$) are not present. Such derivations have the effect of *reducing* the length of the string being generated; ideally, we'd like every rule application to either keep the length of the string the same, or increase it.

The trick to eliminating ε rules will be used later to eliminate other kinds of undesirable derivations, and relies on a simple general principle: if $A \overset{*}{\Longrightarrow} u$ is a derivation in G, then we can *add* a rule $A \to u$ to G without affecting the language. The new rule serves as a "shortcut" through the grammar, but does not add any new strings (because it only gives us what we could already get, just through a shorter derivation) nor remove any.

A *nullable* variable is one that can derive the empty string. A variable can be nullable in one of two ways:

- V is nullable if $(V \to \varepsilon) \in R$
- V is nullable if $(V \to u) \in R$ and every symbol in u is nullable.

Following this definition, we can inductively construct the set $\mathsf{nullable}$, the set of all variables in a grammar that can derive the empty string:

$$\text{Base}\,\frac{(V \to \varepsilon) \in R}{V \in \mathsf{nullable}} \qquad \text{Ind}\,\frac{(V \to u) \in R \qquad u \in \mathsf{nullable}^*}{V \in \mathsf{nullable}}$$

An alternate definition would be to have

$$\text{Base}'\,\frac{}{\varepsilon \in \mathsf{nullable}}$$

as the base case.

If a nullable variable occurs in the right-hand-side of a rule, the question remains as to how to eliminate it (or rather, what rules to replace it with). We must bear in mind that just because a variable is *nullable*, it does not follow that it is *null*; i.e., that the *only* string it derives is the empty string. Since a nullable variable may still derive other, non-empty terminal strings, we must capture the behavior in this case. Our transformation relies on the observation that that if V is nullable then the string uVw can derive either uVw or uw. The second alternative results when $V \overset{*}{\Longrightarrow} \varepsilon$. Thus, our process is, for each rule whose definition includes one or more nullable variables, to generate *all* the alternatives that would result from either keeping or removing any or all of the nullable variables. Note that if there are n nullable variables in a given definition, the expansion will have 2^n alternatives. Also note that distinct occurrences of the *same* nullable variable are treated as separate nullable variables.

Algorithm 1 ε-rule removal

```
1: Construct $N = \mathsf{nullable}$, the set of nullable variables in $G$
2: for all $(V \to u) \in R, u \neq \varepsilon, V \neq S$ do
3:     Let $X$ be the set of nullable variable occurrences in $u$.
4:     for all $x \in \mathscr{P}(X)$ do
5:         Build the string $u'$ such that if $A$ is a nullable variable occurrence in $u$ and $A \in x$ then $A$ remains in $u'$, otherwise it is removed.
6:         Add $V \to u'$ to $R'$
7:     end for
8: end for
9: (Any variables which have no definitions in $R'$ can be removed from $V$.)
```

As a special case, we allow $S \to \varepsilon$. If this were not allowed, it would be impossible to build a grammar for the null language. (But note that under this definition, all null languages now have the identical grammar, consisting of just the rule $S \to \varepsilon$.)

Eliminate chain rules

A *chain rule* is a rule of the form $A \to B$. The application of such a rule does not bring the string being derived any closer to being terminal; it simply renames a variable. We wish to remove such rules from the grammar.

We can inductively define the set of *chain variables of a variable* V as follows:

$$\text{Base}\,\frac{}{V \in \mathsf{chain}(V)} \qquad \text{Ind}\,\frac{A \in \mathsf{chain}(V) \qquad (A \to B) \in R}{B \in \mathsf{chain}(V)}$$

To eliminate a chain rule $A \to B$, we simply replace the alternative B with the complete *definition* (all alternatives) of B. Assuming we have "unchained" B previously, this definition of A will now be chain-free as well.

Algorithm 2 Elimination of chain rules

```
1: for all $A \in V$ do
2:     for all $A' \in \mathsf{chain}(A)$ do
3:         If $A' \to u$ is the definition of $A'$, add $A \to u$ to $R'$ (provided $u$ is not a single variable).
4:     end for
5: end for
```

Note that the added non-recursive start rule S' is *not* exempt from this processing: the definition of S' will be replaced with the complete definition of S.

Removing useless variables

A variable can be *useless* in two ways:

- If it cannot be reached from the start symbol. That is, if there is no derivation $S \overset{*}{\Longrightarrow} uVw$ then V is *unreachable*.
- If it cannot derive any terminal strings. That is, if there is no derivation $V \overset{*}{\Longrightarrow} u$ with $u \in \Sigma^*$ then V is *non-terminating*.

We will develop definitions for both of these concepts separately, and then combine them to form a single notion of *useless variables* and to arrange their removal.

Unreachable symbols: A symbol is unreachable if it does not occur in any string derived from the start symbol. We will find it easier to define the complementary set of all *reachable* variables:

$$\text{Base}\,\frac{}{S \in \mathsf{reachable}} \qquad \text{Ind}\,\frac{V \in \mathsf{reachable} \qquad (V \to uV'w) \in R}{V' \in \mathsf{reachable}}$$

Then the set of unreachable variables is simply $V - \mathsf{reachable}$.

Non-terminating variables: A variable is *non-terminating* if it cannot derive any terminal strings. As above, we will find it easier to define the set of *terminating* variables which *can* derive terminal strings, and then take its complement.

$$\text{Base}\,\frac{(V \to u) \in R \qquad u \in \Sigma^*}{V \in \mathsf{terminating}} \qquad \text{Ind}\,\frac{(V \to u) \in R \qquad u \in (\Sigma \cup \mathsf{terminating})^*}{V \in \mathsf{terminating}}$$

Then the set of non-terminating variables is simply $V - \mathsf{terminating}$.

An alternate base case would be

$$\text{Base}'\,\frac{a \in \Sigma}{a \in \mathsf{terminating}}$$

Then, the set terminating is the set of all symbols—terminal and non-terminal—that can derive terminal strings.

Useless variables: A variable is useless iff it is either non-terminating or unreachable. We thus define the set useless as

$$\mathsf{useless} = V - (\mathsf{terminating} \cap \mathsf{reachable})$$

To remove useless variables from the grammar, we simply delete any rules that refer to them, either on the left (i.e., as a definition of them) or on the right. If V is a useless variable, then an alternative $A \rightarrow uVw$ will either a) not be able to derive any terminal strings (if V is non-terminating), b) not be derivable from S (if V is unreachable) or c) both. And if $V \rightarrow u$ *defines* a useless variable V then removing the definition will have no effect on the language of the grammar.

Chomsky normal form

In Chomsky normal form, every rule must have one of the following forms:

$$S \rightarrow \varepsilon$$
$$A \rightarrow \mathtt{a}$$
$$A \rightarrow BC$$

The third form essentially structures derivations as binary trees, with terminal derivations (via the second form) as the leaves.

To translate a grammar into CNF we begin by adding rules of the second form for all terminal symbols which are present in the grammar:

$$\frac{V \rightarrow s \qquad s = u\mathtt{a}v}{(\mathtt{A} \rightarrow \mathtt{a}) \in R'}$$

(details to follow)

Greibach normal form

In Greibach normal form, every rule must have one of the following forms:

$$S \rightarrow \varepsilon$$
$$A \rightarrow \mathtt{a}A_1A_2 \dots A_n \qquad (n \geq 0)$$

Note that a grammar in Greibach normal form is guaranteed to be left-recursion-free.

Transforming a grammar into Greibach normal form usually results in an explosion in the number of rules: in the worst case, the normalized grammar will have $O(|R|^4)$ rules.

The advantage to Greibach normal form is that it has an easy translation into an extended pushdown automaton[1]: The automaton will have two states, q_0 initial, and q_1 final. and a rule of the form

[1] An extended PDA is one which can push any number of symbols onto the stack in a single transition.

$$A \rightarrow \mathtt{a}A_1A_2 \dots A_n \qquad A \neq S$$

is translated into a transition from q_1 to q_1 that reads an $\mathtt{a}$, pops an A, and then pushes $A_1A_2 \dots A_n$. A rule of the form

$$S \rightarrow \mathtt{a}A_1A_2 \dots A_n$$

is translated into a transition from q_0 to q_1. If $(S \to \varepsilon) \in R$ then there is an ε-transition (reads nothing, pushes and pops nothing) from q_0 to q_1.

This translation serves to prove that every context-free grammar can be accepted by a PDA (since any CFG can be translated into Greibach normal form, and any extended PDA can be transformed into a traditional PDA).

Pushdown Automata

Parsing Algorithms

Parsing Expression Grammars

Parsing expression grammars are a *recognition*-based framework for parsing. While our previous definition(s) for grammars defined them in terms of the strings that the *generate* (via derivations), PEGs' functionality is defined in terms of the strings that it *recognizes*.

A PEG can be thought of as a CFG where every decision is *greedy*; in a set of alternatives, the *first* alternative to match is the one chosen, and repetition always captures as much of the input as possible.

A PEG has the following general syntax:

$P \leftarrow P_1 P_2 \ldots P_n$	(Sequence)
$P \leftarrow P_1 \,/\, P_2 \,/\, \ldots \,/\, P_n$	("Ordered choice")
$P \leftarrow P^*$	(Zero-or-more)
$P \leftarrow P^+$	(One-or-more)
$P \leftarrow P^?$	(Zero-or-one)
$P \leftarrow \&P$	(Positive lookahead)
$P \leftarrow\, !P$	(Negative lookahead)
$P \leftarrow (P)$	
$P \leftarrow \mathtt{c}$	(Terminal symbol)
$P \leftarrow .$	(Any terminal symbol)

PEG parsing can best be thought of as a process of matching and consuming *prefixes* of the input string:

- To match a sequence $P_1 P_2 \ldots P_n$, first match P_1. If that is successful, match P_2 at the point where P_1 left off. Continue until P_n is matched and consumed. If any of the P_i fails to match, the entire sequence fails, and nothing is consumed.

- To match an ordered choice $P_1 \,/\, P_2 \,/\, \ldots \,/\, P_n$ first try to match P_1. If that is successful, then the entire match succeeds, consuming whatever P_1 consumed. If P_1 fails then try to match P_2. If, finally, P_n fails, then the entire choice fails and nothing is consumed. Note that while in a sequence each P_i

is started at the point where the previous left off, in an ordered choice the original starting position is restored before matching each P_i.

- To match P^*, just match
$$T \leftarrow PT \,/\, \varepsilon$$
- To match P^+, just match PP^*.
- To match $P^?$, just match
$$T \leftarrow P \,/\, \varepsilon$$
- To match &P, try to match P. If this is successful, then the lookahead succeeds *but consumes nothing*. Otherwise the lookahead fails.
- To match !P, try to match P. If this is successful, then the lookahead fails, otherwise the lookahead succeeds, consuming nothing.

The fact that positive and negative lookahead accept arbitrary PEG expressions as their arguments means that lookahead in PEGs is much more powerful than the fixed-length lookahead of LL(k) or LR(k) languages, or even the RE-based lookahead of LL($*$).

To match each of the PEG constructs, we present pseudo-code algorithms that describe matching in terms of advancing a match pointer `at` in a global input string s. Each match construct takes the current match position as an argument, and returns the new match position. If the match fails, -1 is returned as a sentinel value. (We assume that beyond the end of the string is padded with a unique terminal symbol `#` which does not occur anywhere else.)

To match a terminal symbol `c`:

```
if $s$[at] = c then return at + 1
else return $-1$
end if
```

To match any terminal symbol `.`:

```
if $s$[at] $\neq$ # then return at + 1
else return $-1$
end if
```

To match a sequence $P_1P_2 \ldots P_n$:

```
if (at ← $P_1$(at)) $\geq 0$ then
    if (at ← $P_2$(at)) $\geq 0$ then
        ...
        if (at ← $P_n$(at)) $\geq 0$ then
            return at
        end if
    end if
end if
```

```
return $-1$
```

Note that after matching each P_i we update the current position `at`.

To match an ordered choice $P_1 \,/\, P_2 \,/\, \ldots \,/\, P_n$:

```
orig $\leftarrow$ at
if (at $\leftarrow P_1$(orig)) $\geq 0$ then
    return at
else if (at $\leftarrow P_2$(orig)) $\geq 0$ then
    return at
else if ... then
    ...
else if (at $\leftarrow P_n$(orig)) $\geq 0$ then
    return at
else
    return $-1$
end if
```

To match zero-or-more P^*:

```
last $\leftarrow$ at
while at $\leftarrow P$(at) $\geq 0$ do
    last $\leftarrow$ at
end while
return last
```

To match one-or-more P^+, just match the sequence PP^*.

To match zero-or-one $P^?$:

```
orig $\leftarrow$ at
if (at $\leftarrow P$(at)) $\geq 0$ then
    return at
else
    return orig
end if
```

To match the positive lookahead $\&P$:

```
if $P$(at) $\geq 0$ then
    return at
else
    return $-1$
end if
```

To match the negative lookahead $!P$:

```
if $P$(at) $< 0$ then
    return at
else
```

```
4:     return $-1$
5: end if
```

Note that implementing PEGs in a language which supports first-class functions is particularly easy. Each of the above constructs becomes a higher-order function taking one or more "matcher" functions P_i as inputs.