CFG & PDA

Alex S.[*]

1 Context Free Languages

Pal (language of palindromes) is context free.

Same as regular languages, only with recursion. Have no context.

1.1 Context Free Grammars

Grammar is a 4-tuple, $(V, \sum, S, P)$, where V is a set of variables, $\sum$ is a set of terminals, S is the starting variable ($S \in V$), and P is a set of formulas of the form $A \rightarrow$ *something*, where A is element of V, and *something* is element of $(V \cup \sum)*$.

derivation trees, ambiguity

1.2 First & Follow

There are some relationships which are convinient to use when dealing with grammars:

First is the set of all terminals that can begin a sentential form derived from α:

$$First(\alpha) = \{a \in V_t | \alpha \Rightarrow^* a\beta\} \bigcup (\text{if } \alpha \Rightarrow^* \lambda \text{ then } \{\lambda\} \text{ else } \emptyset)$$

There's also the follow set:

$$Follow(A) = \{a \in V_t | S \Rightarrow^+ \cdots Aa \cdots\} \bigcup (\text{if } S \Rightarrow^+ \alpha A \text{ then } \{\lambda\} \text{ else } \emptyset)$$

1.3 Recursive Descent, LL(1)

There is a relatively simple parser type called recursive descent parser. These can parse LL(1) grammars. These can look ahead one token to determine their next step.

The next few sections show how to take a grammar and attempt to transform it into an LL(1) grammar. I say attempt, because not all grammars can be transformed into LL(1) grammars. If you cannot transform a grammar into LL(1), usually one tries to change the grammar slightly to make it compatible.

If you look at any early programming languages, nearly all of them use LL(1), or equivalent, grammar.

[*] `alex@theparticle.com`

1.3.1 Common Prefixes

Because the parser needs to look a few (or one) tokens ahead to make a decision, grammar productions that have a common prefix present a problem. For example, lets say we have a production of:

```
<stmt> -> if <expr> then <stmt list> end if ;
<stmt> -> if <expr> then <stmt list> else <stmt list> end if ;
```

The above seems relatively sensible for any programming language to have. Unfortunately just by looking at the first few tokens (like `if`), we cannot determine which production to use.

We need to eliminate common prefixes by factoring them out of the grammar, and turn the above `if` statements into something like:

```
<stmt>      -> if <expr> then <stmt list> <if suffix>
<if suffix> -> end if ;
<if suffix> -> else <stmt list> end if ;
```

The general rule for this is: For all grammar productions with same LHS (Left Hand Side) that have a common prefix,

$$S = \{A \rightarrow \alpha\beta, \ldots, A \rightarrow \alpha\zeta\}$$

In the above case, α is the common prefix. Create a new nonterminal N, and replace S with:

$$\{A \rightarrow \alpha N, N \rightarrow \beta, \ldots, N \rightarrow \zeta\}$$

1.3.2 Left Recursion

When constructing grammars, we often end up with grammars like (for mathematical expression):

```
E -> E + T
E -> T
T -> T * P
T -> P
P -> ID
```

The problem is that we cannot use this grammar for certain types of parsers, like LL(1). In order to 'fix' it, we have to eliminate left recursion, by transforming the grammar into something like this:

```
E     -> T Etail
Etail -> + T Etail
Etail -> Lambda
```

```
T     -> P Ttail
Ttail -> * P Ttail
Ttail -> Lambda
P     -> ID
```

The general rule for this is: For all grammar production that have left recursion,

$$S = \{A \to A\alpha, A \to \beta, \ldots, A \to \zeta\}$$

Create two new nonterminals, T and N, and replace set S with:

$$\{A \to NT, N \to \beta, \ldots, N \to \zeta, T \to \alpha T, T \to \lambda\}$$

getting rid of lambdas

1.4 Push-Down Automata

PDA is a 7-tuple, $(Q, \sum, \Gamma, q_0, Z_0, A, \delta)$, where Q are states, $\sum$ is input alphabet, Γ is the stack alphabet, q_0 is the first state, Z_0 is the initial stack symbol (element of Γ), A is a set of accepting states (subset of Q), and $\delta : Q \times (\sum \cup \{\lambda\}) \times \Gamma$ (finite subsets of $Q \times \Gamma *$).

We move from "configuration" to configuration.

Configuration of PDA is current state, input string, and top of stack.