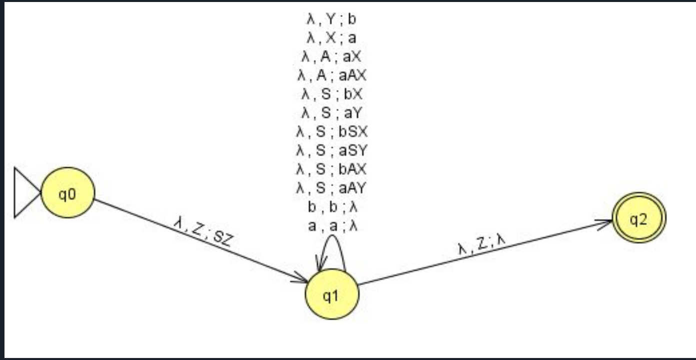
Graphical Conversion of CFG to PDA

Patrick Cook
Ryan Barrington
Nathaniel Aldino
Matias Cinera

Context Free Grammars, and Pushdown Automata

- CFG formal definition developed in 1950's by Noam Chomsky
  - Defined by a 4-tuple with non-terminal symbols, terminal symbols, start symbol, and productions.
  - G = {V, T, S, P}, V = non-terminals, T = terminals, S = start symbol, and P = productions.
- Pushdown Automata also first formalized by Noam Chomsky
  - More powerful than a finite state machine since a PDA also has a memory in the form of a stack.
  - Can be used to describe CFG's, and prove a language is a CFG. Can also describe regular languages.



Process Overview of CFG to PDA.

- Simplification of CFG first.
  - Lambda production removals first.
  - Unit production removals.
  - Useless production removals.
- Then conversion to Greibach Normal Form.
- Convert GNF grammar to a PDA.
- Output both transition functions and PDA in the console.
- Prove PDA describes simplified grammar by parsing a string using transition functions.

Lambda Removal

- Lambda Productions are denoted with ‘?’
- Once a production with ‘?’ is found, erase and insert each instance that occurs in other productions.
- Once this is done, each production rule with ‘?’ is removed.

```
char key;
string temp;
vector<pair<char, string>> temp_vec;

for(unsigned int i = 0; i < grammar.size(); i++)
{
    if(grammar[i].second[0] == '?'){
        key = grammar[i].first;
        for(unsigned int j = 0; j < grammar.size(); j++){
            temp = grammar[j].second;
            for(string::size_type u = 0; u < grammar[j].second.size(); u++){
                if(grammar[j].second[u] == key){ //u = production
                    temp.erase(u, 1);
                    temp_vec.push_back(make_pair(grammar[j].first, temp)); // Push new productions into temp container.
                }
            }
        }
    }
}
```

Unit Removal

- Vector of productions with only terminals used to find and replace unit productions.
- Searches and finds matching states, and replaces those production rules.

```
string temp;
vector<pair<char, string>> temp_vec;

for (auto it = grammar.begin(); it != grammar.end(); ++it)
{
    // First find productions of length = 1 and non-terminal.
    if (it->second.length() == 1 && (isupper(it->second[0])))
    {
        // Now iterate through the single_terminals vector to find anything that matches.
        for (auto jt = single_terminals.begin(); jt != single_terminals.end(); ++jt)
        {
            if (jt->first == it->second[0])
            {
                temp_vec.push_back(make_pair(it->first, jt->second));
            }
        }
    }
}
```

Useless Removal

- Iterates through each string in a productions rule, checking whether it is reachable in another production rule.
- Pushes reachable productions, which is later referenced in removing unreachable production rules.

```
vector<char> keys;        // hold the reachable variables
vector<pair<char, string>> ret, temp; // will override the grammar at the end
vector<char>::iterator it;

keys.push_back(grammar[0].first);
for(unsigned int i = 0; i < grammar.size(); i++){
    for(string::size_type k = 0; k < grammar[i].second.size(); k++)
    {
        if(isupper(grammar[i].second[k]))
        {
            for(unsigned int j = 0; j < grammar.size(); j++)
            {
                if(grammar[i].second[k] == grammar[j].first)
                {
                    // Check if the key is already present in the container.
                    it = find(keys.begin(), keys.end(), grammar[i].second[k]);
                    if (it == keys.end())
                    {
                        keys.push_back(grammar[i].second[k]);   // push the reachable keys
                    }
                }
            }
        }
    }
}
```

Greibach Normal Form Conversion

- Variables + temporary vectors that are needed are intialized
- Simplified grammar is copied into temp_grammar for GNF modification
- Single_terminals converted and pushed into <string,string> pair form to allow GNF_singles to operate with Z(n) productions (Z1, Z2, Z3 … Z(n))

```
//new_var and new_prod are used to help generate Z(n) productions, temp is used for char->string conversion
string new_var, new_prod, temp;
//counter keeps track of n in the Z(n) productions
int counter = 0;
//separate vector temp_grammar is used to edits are separate from original grammar
vector<pair<char,string>> temp_grammar = grammar;
//production will be used a temp to help compare substitutions for GNF conversion
pair<string,string> production;

//vector to store productions that start with non terminals to then remove later
vector<pair<char,string>> nonterminals;
vector<pair<char,string>> nonterminal_subs;
int static_tempsize = temp_grammar.size();

//convert single_terminals into GNF_singles since GNF_singles will contain Z(n) productions
for(auto pair: single_terminals) {
    temp = pair.first;
    GNF_singles.push_back(make_pair(temp, pair.second));
}
```

Greibach Normal Form Conversion

- Looping through simplified grammar
- First we check if it starts with terminal and is not a single production
- Second we loop through the production's derivation to replace every terminal with a matching variable production that produces the same
- If one exists we replace it
- Else if, if it is a nonterminal or a Z(n) variable we skip
- Else, we create a new Z(n) production that matches

```
//begin loop to convert
for (unsigned int i=0; i < temp_grammar.size(); i++) {

    //if production starts with a terminal and has more afterwards, search through it
    if( islower(temp_grammar[i].second[0]) && temp_grammar[i].second.size()>1 ) {
        for(unsigned int j=1; j<temp_grammar[i].second.size(); j++) {
            /*  any terminals after the first will be exchanged with non-terminals
             *  seen in the single_terminal productions, if not they will be created and pushed into GNF_singles
             */
            if( islower(temp_grammar[i].second[j]) && !isdigit(temp_grammar[i].second[j])) {
                //searches through GNF_singles each time to find a matching single production, then sets a temporary pair
                for(unsigned int k = 0; k<GNF_singles.size(); k++) {
                    if(temp_grammar[i].second[j] == GNF_singles[k].second[0]) {
                        production = GNF_singles[k];
                    }
                }

                //temporary pair is then compared to the production each symbol at a time
                if( production.second[0] == temp_grammar[i].second[j] ) {
                    temp_grammar[i].second.replace(j,1,production.first);
                }
                else if( isupper(temp_grammar[i].second[j]) || temp_grammar[i].second[j] == 'Z' || isdigit(temp_grammar[i].second[j]) ) {
                    continue;
                }
                else {
                    new_var = "Z" + to_string(counter);
                    new_prod = temp_grammar[i].second[j];
                    GNF_singles.push_back(make_pair(new_var,new_prod));
                    temp_grammar[i].second.replace(j,1,new_var);
                    counter++;
                }
            }
        }
```

Greibach Normal Form Conversion

- If the derivation does not start with a nonterminal, it must begin with a variable production
- The variable identified will be substituted by every single matching production that matches in a temporary vector and stored
- The production that begins with said variable is stored in a temporary vector
- At the end, substitutions are pushed into temp_grammar
- Then the productions that begin with variables in question will be searched through and removed

```
    //in the case of productions that begin with a nonterminal they will be replaced
    else {
        for ( int j = 0; j < static_tempsize; j++)
        {
            /*  each case identified will be pushed to nonterminals and the new productions without the beginning
             *  terminal will be replaced and pushed back into another vector, nonterminal_subs. to be handled afterwards
             */
            string s = temp_grammar[i].second;
            if(temp_grammar[i].second[0] == temp_grammar[j].first) {
                if(islower(temp_grammar[j].second[0])) {
                    nonterminals.push_back(temp_grammar[i]);
                    s.replace(0,1,temp_grammar[j].second);

                    nonterminal_subs.push_back(make_pair(temp_grammar[i].first,s));
                }
            }

        }
    }
}

//push nonterminal subs into temp_grammar
for(auto pair: nonterminal_subs) {
    temp_grammar.push_back(make_pair(pair.first, pair.second));
}
//remove leftover productions that begin with a nonterminal
for(unsigned int k = 0; k<temp_grammar.size(); k++) {
    for(auto duplicate: nonterminals) {
        if(temp_grammar[k].first == duplicate.first && temp_grammar[k].second == duplicate.second)
            temp_grammar.erase(temp_grammar.begin()+k);
    }
```

Greibach Normal Form Conversion

- Temp_grammar is pushed into class data member vector, GNF, via char-> conversion.
- Lastly, new Z(n) productions in GNF_singles, are pushed into GNF and the conversion to Greibach Normal Form is complete and ready to be converted to a Pushdown Automata

```
//now that conversion is finished, temp_grammar is now converted to a <string,string> form to accept Z(n) productions in private var GNF
for(auto pair: temp_grammar) {
    temp = pair.first;
    GNF.push_back(make_pair(temp, pair.second));
}
//after conversion the Z(n) productions GNF_singles are added to the main GNF grammar
for(auto pair: GNF_singles) {
    if(pair.first[0] == 'Z')
        GNF.push_back(make_pair(pair.first,pair.second));
}
```

Conversion to PDA

- Each PDA converted will have three states.
- Struct created to hold the transition functions of the PDA.

```cpp
struct PDA {
    char state;
    std::string read;
    std::string pop;
    std::string push;
};
```

```cpp
temp.state = '1';
temp.read = "?";
temp.pop = "Z";
temp.push = "SZ";
automaton.push_back(temp);
```

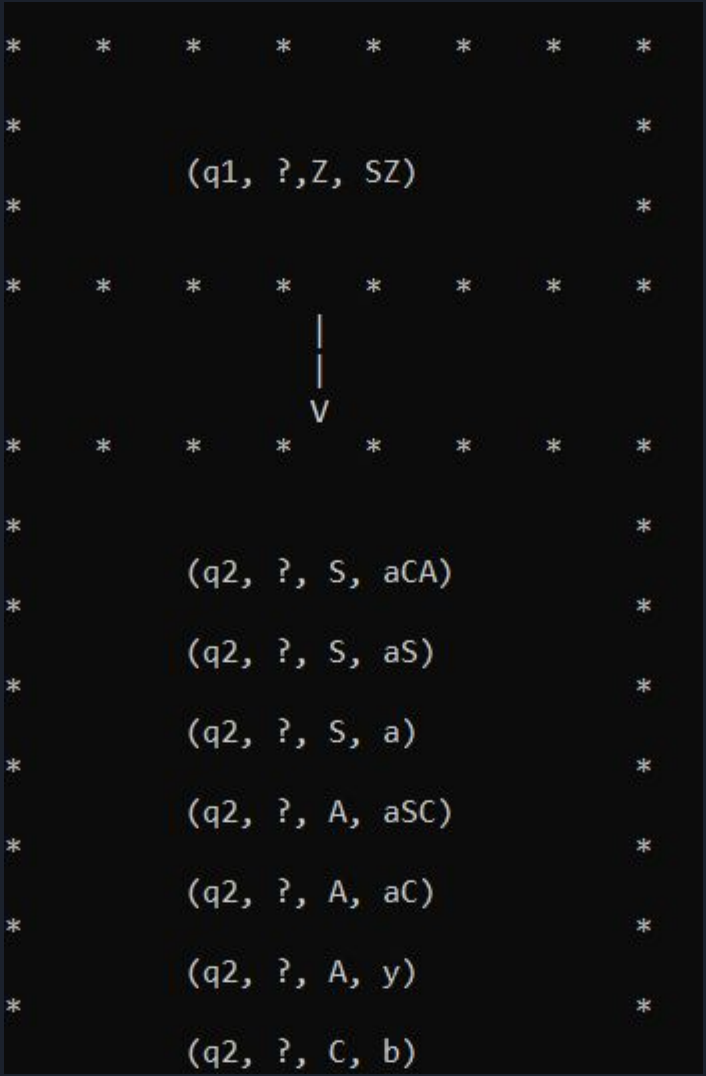
```cpp
temp.state = '3';
temp.read = "?";
temp.pop = "Z";
temp.push = "?";
automaton.push_back(temp);
```

```cpp
for (auto it = GNF.begin(); it != GNF.end(); ++it)
{
    temp.state = '2';
    temp.read = "?";
    temp.pop = it->first;
    temp.push = it->second;
    automaton.push_back(temp);
}

for (auto it = GNF_singles.begin(); it != GNF_singles.end(); ++it)
{
    temp.state = '2';
    temp.read = it->second;
    temp.pop = it->second;
    temp.push = "?";
    automaton.push_back(temp);
}
```

Displaying PDA

- To display the PDA, we used ASCII art to create boxes to hold all the transitions in each of the states.
- To do this, we generate the first and last transition with adding a ‘Z’ to the stack and removing it at the end, then in between the these state transitions is a loop that displays the rest of the transitions in the second state.
- The image shows a portion of the display PDA function.



Observations and Conclusions

- Trial and error of implementation.
  - Attempted using hash tables, but collisions with keys kept occurring.
  - Settled on vector of pairs, due to familiarity.
- Process of simplification.
  - It became necessary to sort the grammar.
  - Multiple cases tested to prevent unwanted behavior of the program.
- PDA conversion.
  - Most PDA's converted from a grammar follow a very simple structure.