MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work n.6

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

1. Theory:

The process of gathering syntactical meaning or doing a syntactical analysis over some text can also be called parsing. It usually results in a parse tree which can also contain semantic information that could be used in subsequent stages of compilation, for example. Similarly to a parse tree, in order to represent the structure of an input text one could create an Abstract Syntax Tree (AST). This is a data structure that is organized hierarchically in abstraction layers that represent the constructs or entities that form up the initial text. These can come in handy also in the analysis of programs or some processes involved in compilation.

2. Purpose of the task work:

Get familiar with parsing, what it is and how it can be programmed [1].

Get familiar with the concept of AST [2].

In addition to what has been done in the 3rd lab work do the following:

1. In case you didn't have a type that denotes the possible types of tokens you need to:
    a. Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
    b. Please use regular expressions to identify the type of the token.
2. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
3. Implement a simple parser program that could extract the syntactic information from the input text.

3. Implementation description:

Creating Grammar Rules:

The *rule* function is essential for defining grammar rules in a structured way. It takes a string representation of a grammar rule, along with parameters for tree creation and processing. The rule string is split into its left-hand side (LHS) and right-hand side (RHS). The LHS is the non-terminal being defined, while the RHS consists of the symbols that form the rule. Additional parameters include the tree name, a processing function (often a lambda function), and an optional anti-lookahead list to prevent certain tokens from following this rule. The function returns a list representing the rule, which is used later in the parsing process.

```
def rule(rstring,tname,tprocess, antilookahead=[]):

    rule = []

     #Split string into terminal/nonterminals. First one is leftside of
rewrite rule, the rest are right side of rewrite rule

    rstring = rstring.split()

    rule.append(rstring.pop(0))
```

```
    rule.append(rstring)
    #Add the tree name and process for future tree-making
    rule.append([tname])
    rule.append(tprocess)
    rule.append(antilookahead)
    return rule
```

Constructing the Parse Tree:

The *make_tree* function is responsible for transforming a list of tokens into a segment of the parse tree using the provided processing function. This function is straightforward: it applies the given lambda function to the tree, which processes and structures the tokens into a meaningful tree segment.

```
def make_tree(tree, process):
        return process(tree)
```

Adding to the Chart:

The *addto* function ensures that values are added to the chart without duplication. The chart is a critical data structure in this parser, representing the states of the parsing process at various positions. Each state in the chart is a potential parsing path that the algorithm explores. By checking for duplicates before adding new states, the parser avoids redundant computations and keeps the chart efficient.

```
def addto(curpos, val):
    ref = val[:4]
    if ref not in reference[curpos]:
        chart[curpos].append(val)
        reference[curpos].append(ref)
```

Handling Grammar Rules with Closure:

The closure function populates the chart with initial states derived from the grammar rules. When a non-terminal is encountered in a state, this function adds all rules that define the non-terminal to the current chart position. It initializes these rules as new states, ready for further parsing. Each new state is a potential path the parser can take, extending the parsing possibilities.

```
def closure(grammar,chart,token,curpos):
    for rule in grammar[token]:
        #If any grammar rule's leftside equals to first unseen
non-terminal
        #Create initialized parsing state. Mutable compenents are
copied. Turn last two components of rule into Tree instance
```

```
                                                                    state     =
[rule[0],deque([]),deque(rule[1]),curpos,list(rule[2]),rule[3],rule[4]]

            addto(curpos,state)
```

Generating the Next State:

The *nextstate* function generates the next parsing state from the current state and the next element to be added to the tree. This function is used in both shifting (advancing the token) and reduction (completing a rule). It copies the current state's components, moves the next token from the RHS to the LHS, and adds the element to the tree. This new state represents the parser's progress after considering the current token.

```
def nextstate(state,element):

                                                          nextstate       =
[state[0],deque(state[1]),deque(state[2]),state[3],list(state[4]),state[5
], state[6]]

        #Cut the beginning of the unseen to the end of the seen tokens

        shifted = nextstate[2].popleft()

        nextstate[1].append(shifted)

        nextstate[4].append(element)

        return nextstate
```

Shifting Tokens:

The *shift* function handles terminal tokens by matching the current token with the expected token in the state. If there is a match, it generates the next state and advances the parsing position. This function effectively moves the parser forward in the input token stream, exploring valid parsing paths.

```
def shift(tokens,chart,state,curpos):

        #If current token matches the next token of the parsing state

        if tokens[curpos] == state[2][0]:

              #Generate the next state by modifying the current state and
adding the current token to the tree

            addto(curpos+1,nextstate(state,tokens[curpos].value))
```

Reducing Non-Terminals:

The *reduction* function completes a non-terminal when a state is finished. It goes back to the origin position to find the originating state and attempts to extend it by adding the completed non-terminal. This function plays a crucial role in building the parse tree by combining smaller tree segments into larger ones.

```
    def reduction(origin,chart,equal,curpos,tree):

          #Go back to the origin chart position to look for the origin
```

```
state

        for state in chart[origin]:

                #If the state isn't finished and its pending token is the
desired non-terminal

            if state[2] and state[2][0] == equal:

                    #Generate the next state by modifying the origin state
and add to chart

                addto(curpos,nextstate(state,tree))
```

The Parsing Process:

The parse function orchestrates the entire parsing process. It initializes the chart and reference structures, adds the end marker to prevent out-of-bounds errors, and populates the starting rule at position zero. The main parsing loop iterates over each position in the token stream, processing all states at each position. If a position in the chart is empty, it indicates a syntax error in the input, and an exception is raised.

For each state at the current position, the parser checks whether the state is finished. If it is, and the next token is not in the anti-lookahead list, the state is reduced by completing its tree and adding it to the chart. If the state's next symbol is a non-terminal, the closure function is invoked to expand it. If the next symbol is a terminal, the shift function is used to match and advance the token.

Finally, if the end of the token stream is reached and the initial rule is completed, the parse tree is returned. This tree represents the fully parsed structure of the input tokens according to the grammar.

```
#Create alternate version of the chart as reference to addto()

    reference = {}

    #End marker to prevent shifting outside of the token list at the end

    endline, endpos = tokens[-1].line, tokens[-1].col

    tokens.append(Token("endmarker",'eof',-1,-1,endline,endpos))

      #Initialize chart positions as lists, add the starting rule to
chart[0]

    for n in range(len(tokens)+1):

        chart[n] = []

        reference[n] = []


chart[0].append([startrule[0],[],deque(startrule[1]),0,startrule[2],start
rule[3],startrule[4]])
```

```
    for curpos in range(len(tokens)+1):

          #If current postion is empty, no state has shifted successfully
and the string is invalid

        if chart[curpos] == []:

            curtoken = tokens[curpos-1]

              raise Exception('Unexpected '+str(curtoken.value)+' at line
'+str(curtoken.line)+' position '+str(curtoken.col)+'.')



          #For each state in the current chart position. Loop will include
new states added by closure.

        for state in chart[curpos]:

            #Variables for components of current parsing state

            equal = state[0]

            seen = state[1]

            unseen = state[2]

            origin = state[3]

            tree = state[4]

            process = state[5]

            antilookahead = state[6]



              #If we are at the end of the tokens and we have the state we
started with finished, string is valid. Then the tree is returned

                if curpos == len(tokens)-1 and equal == startrule[0] and
unseen == deque([]) and origin == 0:

                return make_tree(tree,process)



                   #If state is finished and the next token isn't an
anti-lookahead, finish its tree and run reduction to it, passing in the
finished tree

            if not unseen:

                if tokens[curpos] not in antilookahead:
```

```
                        tree = make_tree(tree,process)
                        reduction(origin,chart,equal,curpos,tree)
                    else:
                        continue
                #If state's pending token is non-terminal(first letter cap) then run closure
                elif unseen[0][0] >= 'A' and unseen[0][0] <= 'Z':
                    closure(grammar,chart,unseen[0],curpos)
                #If state's pending token is terminal run shifting to it
                else:
                    shift(tokens,chart,state,curpos)
```

4. Program execution:

For this input string:

input:

```
string = '''proc (1+2-3,4); proc (1+2-3,4);'''
```

the AST will be:
output:

```
[('statement', ('call', 'proc', [('minus', ('add', ('int', 1), ('int', 2)), ('int', 3)), ('int', 4)])), ('statement', ('call', 'proc', [('minus', ('add', ('int', 1), ('int', 2)), ('int', 3)), ('int', 4)]))]
```

5. Conclusion:

This parser is a sophisticated tool that leverages Earley parsing principles to handle a wide range of context-free grammars. Its modular design, with functions dedicated to specific parsing tasks, makes it flexible and powerful. By carefully managing states and transitions, it constructs a parse tree that represents the syntactic structure of the input tokens, providing a foundation for further processing or interpretation.