Syntax Analyzer Documentation

Dennis A. Canar

10 – 4 – 20

CST – 405 – TOS101

Nasser Tadayon

Main Purpose

The main purpose of this project is to build a part of a much longer project that will be done throughout the semester. The syntax analyzer is going to be used in the compiler that is being built this semester. The job of the syntax analyzer is to take the output from the lexer and create a parse tree from the input given. The syntax analyzer verifies that the input is correct by using the grammar that has been given to it along with a ll1 table assisting its building. Thus, the syntax analyzer will create output for the next part of the compiler

Implementation

­The parser will work by being given a grammar then creating an LL1 table from that grammar. The grammar will be stored in a large array of productions. And each production struct will contain one LHS, up to 12 RHS (possible replacements/ rules), firsts array, follows array, and fistsRules array (to store what rule belongs to what first).

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

These structs are then used to hold the grammar and rules, the grammar is initialized in the initialize() function below. There is more to the function but this should give an idea as to how it works.

Text

Description automatically generated

After the grammar has been set we can then create the first and follow sets. This is also done in the initialize function but at the bottom after all the grammar has been set. The first sets are created for all the grammar and the follow sets are only created for the non-terminals because that is the only case in which we would use the follow set.

Text

Description automatically generated

The first and follow sets are then created by their getFirst() and getFollow() functions.

The first function is below:

Text

Description automatically generated

The first function starts by taking the production rule who’s firsts set we are getting and iterating through its RHS’s. But before that if the LHS of the production is a terminal then we append it to the first set, this call will eventually happen to all the productions whose firsts we are finding. During iteration of the RHS’s we check if the function call is the original or the root call and if it is and the RHS is an epsilon transition then we can add it to the firsts set. When appending to the firsts set we also append the number of the rule that got us to that terminal. We accomplish this by only changing that number at the root call when the RHS’s changes. If there is no epsilon then we continue by iterating through the strings of the RHS to get their first() values. Before getting the firsts we check to see how many of the non-terminals have an epsilon transition to know how many firsts() will need to be called. We do this because if the first() of a production has an epsilon transition then technically the next character in the rule can be the first or whaterver they produce. And in the special case there is an epsilon transition in last char of the RHS we allow the getFirst() to add its epsilon transition.

Graphical user interface, text, application

Description automatically generated

The function to the right is used to check how many consecutive non terminals have epsilon transitions. The function uses a boolean that will set itself to false at the beginning of iteating through a productions RHS’s then if it finds an epsilon transition it will set itself true. If true we continue to the next char in the RHS and increase a counter. At the end we will have a number for how many firsts must be calculated.

The follow function is below:

Text

Description automatically generated

What this does is start by iterating through all of the rules then all of their RHS and search for the symbol so we can see what would be in its follow set. Once we find the symbol we are looking we see if there are any chars to the right of it to follow. If there is none then we get the follow of the LHS in which our symbol was found. If there are chars to the right of the symbol we are searching for then we calculate how many to the right of the symbol have an epsilon transition. We do this to know how many first sets will be used by our original follows set call. Also in the rule where we calculate the amount of firsts() we will be using if the last non terminal has an epsilon transition then we get the follow of the LHS.

This is similar to the chain check used in the getFirst function. Except here we are given an exact rule and position to start iterating from.

Text

Description automatically generated

Now after having all the sets we can create the LL1 table using the load table function.

Text

Description automatically generated

Here we iterate through all the rules then once were in each rule we iterate through their firsts. For each symbol we find we add it to the table. The table is organized how the grammar is, the row number is the same as the rule number. So, P will be our first row, V our second and so on. Then the columns are their rule number – the number of non-terminals. When adding all the symbols if our marker tells us the non-terminal had an epsilon transition in its firsts set then we add its follow set to the table as well.

After this step would then be to get the input from the lexer. I will not show this section because it is not very relevant to the main project and is just string manipulation while reading from a .txt file.

Once the input is ready and stored into its container we can begin to parse. Most of the main function is below.

Text

Description automatically generated

The parse function works with a stack that holds what production rule has been used. When the terminal from the inut container and top of the stack match they pop and we can move to the next terminal that needs to be popped off. The goal is to do this until the bottom of the stack ($) is on the top and the last char of the input ($) is also on top. The use table function is what drives this parse function. What it does is see what the top of the stack is and then the top of the input and find which rule we should use to get to that terminal. It knows where we should go because it uses the ll1 table we built.

A picture containing graphical user interface

Description automatically generatedOutput:

My program does not output a tree but it does output the steps used to validate the grammar, with some adjusting this could be used to create a parse tree.

Denniss-MacBook-Pro:Compiler denniscanar$ ./a.out

Action: P-->VF

Stack: VF$

Input: #i;#i(){#i;i=n+i;}$

Action: V-->WV

Stack: WVF$

Input: #i;#i(){#i;i=n+i;}$

Action: W-->Ti1

Stack: Ti1VF$

Input: #i;#i(){#i;i=n+i;}$

Action: T-->#

Stack: #i1VF$

Input: #i;#i(){#i;i=n+i;}$

Action: Popping-->#

Stack: i1VF$

Input: i;#i(){#i;i=n+i;}$

Action: Popping-->i

Stack: 1VF$

Input: ;#i(){#i;i=n+i;}$

Action: 1-->;

Stack: ;VF$

Input: ;#i(){#i;i=n+i;}$

Action: Popping-->;

Stack: VF$

Input: #i(){#i;i=n+i;}$

Action: V-->e

Stack: F$

Input: #i(){#i;i=n+i;}$

Action: F-->G2

Stack: G2$

Input: #i(){#i;i=n+i;}$

Action: G-->Ti(A)B

Stack: Ti(A)B2$

Input: #i(){#i;i=n+i;}$

Action: T-->#

Stack: #i(A)B2$

Input: #i(){#i;i=n+i;}$

Action: Popping-->#

Stack: i(A)B2$

Input: i(){#i;i=n+i;}$

Action: Popping-->i

Stack: (A)B2$

Input: (){#i;i=n+i;}$

Action: Popping-->(

Stack: A)B2$

Input: ){#i;i=n+i;}$

Action: A-->e

Stack: )B2$

Input: ){#i;i=n+i;}$

Action: Popping-->)

Stack: B2$

Input: {#i;i=n+i;}$

Action: B-->{VL}

Stack: {VL}2$

Input: {#i;i=n+i;}$

Action: Popping-->{

Stack: VL}2$

Input: #i;i=n+i;}$

Action: V-->WV

Stack: WVL}2$

Input: #i;i=n+i;}$

Action: W-->Ti1

Stack: Ti1VL}2$

Input: #i;i=n+i;}$

Action: T-->#

Stack: #i1VL}2$

Input: #i;i=n+i;}$

Action: Popping-->#

Stack: i1VL}2$

Input: i;i=n+i;}$

Action: Popping-->i

Stack: 1VL}2$

Input: ;i=n+i;}$

Action: 1-->;

Stack: ;VL}2$

Input: ;i=n+i;}$

Action: Popping-->;

Stack: VL}2$

Input: i=n+i;}$

Action: V-->e

Stack: L}2$

Input: i=n+i;}$

Action: L-->S5

Stack: S5}2$

Input: i=n+i;}$

Action: S-->Z;

Stack: Z;5}2$

Input: i=n+i;}$

Action: Z-->i6

Stack: i6;5}2$

Input: i=n+i;}$

Action: Popping-->i

Stack: 6;5}2$

Input: =n+i;}$

Action: 6-->=EN

Stack: =EN;5}2$

Input: =n+i;}$

Action: Popping-->=

Stack: EN;5}2$

Input: n+i;}$

Action: E-->IN

Stack: INN;5}2$

Input: n+i;}$

Action: I-->n

Stack: nNN;5}2$

Input: n+i;}$

Action: Popping-->n

Stack: NN;5}2$

Input: +i;}$

Action: N-->OEN

Stack: OENN;5}2$

Input: +i;}$

Action: O-->+

Stack: +ENN;5}2$

Input: +i;}$

Action: Popping-->+

Stack: ENN;5}2$

Input: i;}$

Action: E-->IN

Stack: INNN;5}2$

Input: i;}$

Action: I-->i7

Stack: i7NNN;5}2$

Input: i;}$

Action: Popping-->i

Stack: 7NNN;5}2$

Input: ;}$

Action: 7-->e

Stack: NNN;5}2$

Input: ;}$

Action: N-->e

Stack: NN;5}2$

Input: ;}$

Action: N-->e

Stack: N;5}2$

Input: ;}$

Action: N-->e

Stack: ;5}2$

Input: ;}$

Action: Popping-->;

Stack: 5}2$

Input: }$

Action: 5-->e

Stack: }2$

Input: }$

Action: Popping-->}

Stack: 2$

Input: $

Action: 2-->e

Stack: $

Input: $

CONGRATS INPUT CONTAINED 0 ERRORS

Grammar and symbols table used in program:

