COVER PAGE

CS323 Programming Assignments

Fill out all entries 1 - 6. If not, there will be deductions!

Peer Review (Check one)

1. Name [1.Aidin Tavassoli], (ThumbUP [X] or ThumbDown [ ])

   [2. Balwinder Singh Hayer], (ThumbUP [X] or ThumbDown [ ])

   [3. Haowen Yong], (ThumbUP [X] or ThumbDown [ ])

2. Assignment Number [2]

3. Turn-In Dates: **Final Iteration with Documentation** [11/19/2019]

4. Executable FileName [Final.exe]
   (**A file that can be executed without compilation by the instructor**)

5. Lab Room: [CS 101]
(Execute your program in a lab in the CS building before submission)

6. Operating System/Language: [Windows 10 / C++]

To be filled out by the Instructor:

GRADE:

COMMENTS:

Syntax Analyzer Documentation

1. Problem Statement

The second step of compilation requires the source code to be checked grammatically against the rules of the programming language. Tokens from the lexical analyzer are parsed using a set of grammar rules aka productions and a DFA paired with a stack. This parsing method is one of the many different parsing method which we have chosen to construct our syntax analyzer and it is categorized as left derivative push down automaton (PDA).In order for us to be able to use such method we need to eliminate left recursion which happened when a production is such that a non-terminal such as an expression the left hand side is immediately followed by the same non-terminal in this example another expression:

E -> E + T | T

In which case we use left factoring to eliminate the left recursion. For the above rule, we would introduce a new non-terminal E' and separate the production into 2 different rules:

E -> TE'

E' -> +T|ε

As for the backtracking if we have a rule that would reparse the same of previous tokens the parser cannot distinguish which rule would lead to the correct parse tree in which case need to left factor the same symbols from the right-hand side.

E -> abc-T |abd-T | T

T -> id

The productions above can be factorized to:

E -> E'c-T | E'd-T | T

E' -> ab

T -> id

Because we have eliminated the repeating term in the production rules we not have removed back-tracking and we can proceed with constructing our TDP table.

2. How to use the program

When the program is executed first it will ask the user for the name of the file including the extension .txt followed by the return key. For example, for our purposes the input would be: "SampleInputFile2.txt". Once executed the program displays the tokens and the rules applied to the token step by step. The program will also generate a file named: "output.txt" containing the parsing steps tokens, lexemes and the production rules used in each step.

Requirement: the program executable and the source input file MUST be in the same directory for the program to run independently from an IDE.

3. Program Design

The production rules we have are:

1. <mainStatement> -> <Expression-statement> | <Assignment> | <If> | <Declare> | <While>
2. <Expression-statement> -> <Expression>
3. <Assignment> -> identifier = <Expression>
4. <If> -> if <Conditional> then <StatementBlock> <ElseBlock> endif
5. <DecalIre> -> <Type> identifier <MoreIDs>
6. <While> -> while <Conditional> do <StatementBlock> whileend
7. <Type> -> int | float | bool
8. <MoreIDs> -> , id <MoreIDs> | epsilon
9. <Expression> -> <Term> <Expression Prime>
10. <Conditional> -> <Expression> <ConditionalPrime>
11. <ConditionalPrime> -> <Relop> <Expression> | epsilon
12. <Relop> -> < | <= | == | <> | >= | >
13. <Expression Prime> -> + <Term> <Expression Prime> | - <Term> <Expression Prime> | epsilon
14. <Term> -> <Factor> <Term Prime>
15. <Term Prime> ->* <Factor> <Term Prime> | / <Factor> <Term Prime> | epsilon
16. <Factor> -> identifier| - <Primary> | <Primary>
17. <Primary> -> identifier | int | (<Expression>) | real | true | false

As explained before the left recursion and back-tracking have already removed using the method shown in the example and we ended up with the set of production rules above. We will use these rules to create our table.

To create the table, we first need to get the first and follow of each non-terminal in the productions. We have done that below in 3 steps:

Step 1: get the first function outputs for all non-terminals:

First(mainStatement) =	id	if	int	float	bool	While	
First(Expression-statement) =	id						
First(Assignment) =	id						
First(If) =	if						
First(Declare) =	int	float	bool				
First(While) =	while						
First(Type) =	int	float	bool				

First(MoreIDs) =	,	ε					
First(Expression) =	id						
First(Conditional) =	id						
First(Conditional-Prime) =	<	<=	==	<>	>=	>	ε
First(Relop) =	<	<=	==	<>	>=	>	
First(Expression-Prime)	+	-	ε				
First(Term) =	id						
First(Term-Prime) =	*	/	ε				
First(Factor) =	id	-	int	real	true	false	(
First(Primary) =	id	int	(	real	true	false	

Step 2: Get the follow using the follow function rules as below:

Follow(mainStatement) =	$						
Follow(Expression-statement) =	$						
Follow(Assignment) =	$						
Follow(If) =	$						
Follow(Declare) =	$						
Follow(While) =	$						
Follow(Type) =	Id						
Follow(MoreIDs) =	$						
Follow(Expression) =	<	<=	==	<>	>=	>	$
Follow(Conditional) =	do	then					
Follow(Conditional-Prime) =	do	then					
Follow(Relop) =	id						
Follow(Expression-Prime)	<	<=	==	<>	>=	>	$
Follow(Term) =	<	<=	==	<>	>=	>	$
Follow(Term-Prime) =	<	<=	==	<>	>=	>	$
Follow(Factor) =	<	<=	==	<>	>=	>	$
Follow(Primary) =	<	<=	==	<>	>=	>	$

Step 3: Now we can construct the table using the above functions:

	mainStatement	Exp-Statem	Assignmen	If	Declare	While	Type	MoreIDs	Exp	Condl	Cond-Prm	Relop
id	1	2	3	4					9	10		
=												
+												
-												
*												
/												
<										11	11	12
<=										11	11	12
==										11	11	12
<>										11	11	12
>=										11	11	12
>										11	11	12
if	1										<C-Prm>->ε	
then												
else												
endif												
(												
)												
,								8				
while	1					6						
whileend												
int	1				5		7					
float	1				5		7					
bool	1				5		7					
real												
true												
false											<C-Prm>->ε	
do								8				
$												

Exp-Prm			13	13			<E-Prm><>ε	<E-Prm><>ε	<E-Prm><>ε	<E-Prm><>ε	<E-Prm><>ε	<E-Prm><>ε																		<E-Prm><>ε
Term	14																													
Term-Prm						15	15	<T-Prm><>ε	<T-Prm><>ε	<T-Prm><>ε	<T-Prm><>ε	<T-Prm><>ε	<T-Prm><>ε																			<T-Prm><>ε
Factor	16			16														16					16				16	16	16		
Primary	17																	17						17			17	17	17		

Using this table, we constructed functions that would take the tokens and the lexeme from the lexical analyzer function and apply the rules from the table above. In short, these functions call each other until the reach a terminal accepting state. All of them have an else which is reached when the parameters and inputs correspond with the empty blocks of the table.

1. Limitations

None

2. Shortcomings

None

Test Cases:

1)

Enter Input File Name (with extensions): **test.txt**

x=a + b;

z=c+ d;

a=e-f;

b=g*n;

-Token: identifier -Lexeme: x

<Statement> -> <Expression-statement> | <Assign> | <If> | <Decalre> | <While>

<Assign> -> identifier = <Expression>;

-Token: operator -Lexeme: =

-Token: identifier -Lexeme: a

<Expression> -> <Term> <ExpressionPrime>

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: operator -Lexeme: +

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

-Token: identifier -Lexeme: b

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: separator -Lexeme: ;

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

<Empty> -> epsilon

-Token: identifier -Lexeme: z

<Statement> -> <Expression-statement> | <Assign> | <If> | <Decalre> | <While>

<Assign> -> identifier = <Expression>;

-Token: operator -Lexeme: =

-Token: identifier -Lexeme: c

<Expression> -> <Term> <ExpressionPrime>

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: operator -Lexeme: +

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

-Token: identifier -Lexeme: d

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: separator -Lexeme: ;

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

<Empty> -> epsilon

-Token: identifier -Lexeme: a

<Statement> -> <Expression-statement> | <Assign> | <If> | <Decalre> | <While>

<Assign> -> identifier = <Expression>;

-Token: operator -Lexeme: =

-Token: identifier -Lexeme: e

<Expression> -> <Term> <ExpressionPrime>

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: operator -Lexeme: -

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

-Token: identifier -Lexeme: f

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: separator -Lexeme: ;

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

<Empty> -> epsilon

-Token: identifier -Lexeme: b

<Statement> -> <Expression-statement> | <Assign> | <If> | <Decalre> | <While>

<Assign> -> identifier = <Expression>;

-Token: operator -Lexeme: =

-Token: identifier -Lexeme: g

<Expression> -> <Term> <ExpressionPrime>

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: operator -Lexeme: *

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

-Token: identifier -Lexeme: n

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: separator -Lexeme: ;

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

<Empty> -> epsilon

Finished.

Press any key to continue . . .

2)

Enter Input File Name (with extensions): **if.txt**

if a + b then a=b; endif

-Token: keyword -Lexeme: if

<Statement> -> <Expression-statement> | <Assign> | <If> | <Decalre> | <While>

<If> -> if <Conditional> then <StatementBlock> <ElseBlock> endif

-Token: identifier -Lexeme: a

<Conditional> -> <Expression> <ConditionalPrime>

<Expression> -> <Term> <ExpressionPrime>

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: operator -Lexeme: +

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

-Token: identifier -Lexeme: b

<Term> -> <Factor> <TermPrime>

<Factor> -> - <Primary> | <Primary>

<Primary> -> identifier | int | (<Expression>) | real | true | false

-Token: keyword -Lexeme: then

<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>

<Empty> -> epsilon

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>

<Empty> -> epsilon

<ConditionalPrime> -> <Relop> <Expression> | <Empty>

<Relop> -> < | <= | == | <> | >= | >

Error: invalid relational operator - line 1

<Empty> -> epsilon

-Token: identifier -Lexeme: a

<StatementBlock> -> { <Statement> <moreStatement>

Error: invalid { statement block } - line 1

Error: invalid If block - line 1

Press any key to continue . . .