Compilers Assignment #3:

The Parser

|  |  |
| --- | --- |
| **Student Name** | Seongyeop Jeong (040 885 882) |
| **Course** | CST8152 – Compilers |
| **Lab Section** | 012 |
| **Assignment** | 3 |
| **Professor** | Sv. Ranev |
| **Due date** | December 6, 2018 |
| **Date** | December 6, 2018 |

**Contents**

* Documentation
  + Left factoring
  + Left recursion
  + First set
* Marking sheet

**PLATYPUS Program**

<program> -> PLATYPUS { <opt\_statements> } SEOF

FIRST (<program>) = { KW\_T(PLATUPUS) }

<opt\_statements> -> <statements> | 

FIRST (<opt\_statements>) = { AVID\_T, SVID\_T,

KW\_T(WHILE),

KW\_T(IF),

KW\_T(READ), KW\_T(WRITE), ε }

<statements> -> <statement> | <statements> <statement>

***Transforming grammar***

<statements> -> <statements> <statement> | <statement>

<statements> -> <statement><statements’>

FIRST (<statements>) = { AVID\_T, SVID\_T,

KW\_T(WHILE),

KW\_T(IF),

KW\_T(READ), KW\_T(WRITE) }

***Left recursion***

<statements’> -> <statement><statements’> | ε

FIRST (<opt\_statements>) = { AVID\_T, SVID\_T,

KW\_T(WHILE),

KW\_T(IF),

KW\_T(READ), KW\_T(WRITE),

KW\_T(TRUE), KW\_T(FLASE), ε }

<statement> ->

<assignment statement>

| <selection statement>

| <iteration statement>

| <input statement>

| <output statement>

FIRST (<statement>) = { AVID\_T, SVID\_T, KW\_T(IF), KW\_T(WHILE), KW\_T(WRITE), KW\_T(READ) }

**Assignment Statement**

<assignment statement> ->

<assignment expression>;

FIRST(<assignment statement>) = { AVID\_T, SVID\_T }

< assignment expression> ->

AVID = <arithmetic expression>

| SVID = <string expression>

FIRST(<assignment expression>) = { AVID\_T, SVID\_T }

**Selection Statement( the if statement)**

<selection statement> ->

IF <pre-condition> (<conditional expression>) THEN { <opt\_statements> }

ELSE { <opt\_statements> } ;

FIRST(<selection statement >) = { KW\_T(IF) }

**Iteration Statement (the loop statement)**

<iteration statement> ->

WHILE **<**pre-condition> **(<**conditional expression>**)**

REPEAT **{**<statements*>***};**

FIRST(<iteration statement>) = { KW\_T(WHILE) }

**<**pre-condition> ->

TRUE | FALSE

FIRST(<pre-condition>) = { KW\_T(TRUE), KW\_T(FALSE) }

**Input Statement**

<input statement> ->

READ (<variable list>);

FIRST(<input statement >) = { KW\_T(READ) }

<variable list> ->

<variable identifier> | <variable list>,<variable identifier>

***Left recursion***

<variable list> ->

<variable list>,<variable identifier> | <variable identifier>

<variable list> -> <variable identifier><variable list’>

FIRST(<variable list> = { AVID\_T, SVID\_T }

<variable list’> -> , <variable identifier> <variable list’> | ε

FIRST(<variable list’> = { COM\_T, ε }

**Output Statement**

<output statement> ->

WRITE (<*opt\_variable list>*);

| WRITE(STR\_T);

***Applying left factoring***

<output statement> ->

WRITE (<output list>);

FIRST(<ouput statement>) = { KW\_T(WRITE) }

<output list> ->

<*opt\_variable list> | STR\_T*;

FIRST(<output list >) = { KW\_T(WRITE) }

FIRST(<opt\_variable list > = { AVID\_T, SVID\_T, ε }

**Expressions**

**Arithmetic Expression**

<arithmetic expression> - >

<unary arithmetic expression>

| <additive arithmetic expression>

FIRST (<arithmetic expression>) = { -, + ,FIRST(<additive arithmetic expression>) }

<unary arithmetic expression> ->

- <primary arithmetic expression>

| + <primary arithmetic expression>

FIRST (<unary arithmetic expression>) = { -, + }

<additive arithmetic expression> ->

<additive arithmetic expression> + <multiplicative arithmetic expression>

| <additive arithmetic expression> - <multiplicative arithmetic expression>

| <multiplicative arithmetic expression>

***Left Recursion***

<additive arithmetic expression> ->

<multiplicative arithmetic expression> < additive arithmetic expression’>

FIRST (<additive arithmetic expression> ) = {AVID\_T, FPL\_T, INL\_T }

<additive arithmetic expression’> ->

+ <multiplicative arithmetic expression> <additive arithmetic expression’>

| - <multiplicative arithmetic expression> <additive arithmetic expression’>

| ε

FIRST (<additive arithmetic expression’> ) = { +, -, ε }

<multiplicative arithmetic expression> ->

<multiplicative arithmetic expression> \* <primary arithmetic expression>

| <multiplicative arithmetic expression> / <primary arithmetic expression>

| <primary arithmetic expression>

***Left Recursion***

<multiplicative arithmetic expression> ->

<primary arithmetic expression><multiplicative arithmetic expression’>

FIRST(<multiplicative arithmetic expression>) = { AVID\_T, FPL\_T, INL\_T }

<multiplicative arithmetic expression’> ->

\* <primary arithmetic expression><multiplicative arithmetic expression’>

| / <primary arithmetic expression><multiplicative arithmetic expression’>

| ε

FIRST(<multiplicative arithmetic expression>) = { \*, /, ε }

<primary arithmetic expression> ->

AVID\_T

| FPL\_T

| INL\_T

| (<arithmetic expression>)

FIRST(<primary arithmetic expression >) = { AVID\_T, FPL\_T, INL\_T, ( }

**String Expression**

<string expression> ->

<primary string expression>

| <string expression> # <primary string expression>

***Left Recursion***

<string expression> ->

<string expression> # <primary string expression>

| <primary string expression>

<string expression> ->

<primary string expression><string expression’>

FIRST(<string expression>) = { SVID\_T, STR\_T }

<string expression’> ->

# <primary string expression><string expression’> | ε

FIRST(<string expression’>) = { #, ε }

<primary string expression> ->

SVID\_T

| STR\_T

FIRST(<primary string expression >) = { SVID\_T, STR\_T }

**Conditional Expression**

<conditional expression> ->

<logical OR expression>

FIRST(<logical AND expression>) = { AVID\_T, FPL\_T, INL\_T, SVID\_T, STR\_T }

s<logical OR expression> ->

<logical AND expression>

| <logical OR expression> .OR. <logical AND expression>

***Left recursion***

<logical OR expression> ->

<logical OR expression> .OR. <logical AND expression>

| <logical AND expression>

<logical OR expression> ->

<logical AND expression> <logical OR expression’>

FIRST(<logical AND expression>) = { AVID\_T, FPL\_T, INL\_T, SVID\_T, STR\_T }

<logical OR expression’> ->

.OR. <logical AND expression><logical OR expression’> | ε

FIRST(<logical OR expression>) = { .OR., ε }

<logical AND expression> ->

<relational expression>

| <logical AND expression> .AND. <relational expression>

***Left recursion***

<logical AND expression> ->

<logical AND expression> .AND. <relational expression>

| <relational expression>

<logical AND expression> ->

<relational expression><logical AND expression’>

FIRST(<logical AND expression>) = { AVID\_T, FPL\_T, INL\_T, SVID\_T, STR\_T }

<logical AND expression’> ->

.AND. <relational expression> <logical AND expression’> | ε

FIRST(<logical AND expression>) = { .AND., ε }

**Relational Expression**

<relational expression> ->

<primary a\_relational expression> == <primary a\_relational expression>

| <primary a\_relational expression> <> <primary a\_relational expression>

| <primary a\_relational expression> > <primary a\_relational expression>

| <primary a\_relational expression> < <primary a\_relational expression>

| <primary s\_relational expression> == <primary s\_relational expression>

| <primary s\_relational expression> <> <primary s\_relational expression>

| <primary s\_relational expression> > <primary s\_relational expression>

| <primary s\_relational expression> < <primary s\_relational expression>

***Apply left factoring***

<relational expression> →

<primary a\_relational expression><primary a\_relational expression’>

| <primary s\_relational expression><primary s\_relational expression’>

FIRST(<relational expression>) = { AVID\_T, FPL\_T, INL\_T, SVID\_T, STR\_T }

<primary a\_relational expression’> →

== <primary a\_relational expression>

| <> <primary a\_relational expression>

| > <primary a\_relational expression>

| < <primary a\_relational expression>

| ε

FIRST(<primary a\_relational expression’>) = { ==. <>, >, <, ε }

<primary s\_relational expression’> →

== <primary s\_relational expression>

| <> <primary s\_relational expression>

| > <primary s\_relational expression>

| < <primary s\_relational expression>

FIRST(<primary s\_relational expression’>) = { ==. <>, >, < }

<primary a\_relational expression> ->

AVID\_T

| FPL\_T

| INL\_T

FIRST(<primary a\_relational expression>) = { AVID\_T, FPL\_T, INL\_T }

<primary s\_relational expression> ->

<primary string expression>

FIRST(<primary s\_relational expression>) = { SVID\_T, STR\_T }