Grammar

Billy Saint-Fort & Nathan Festoso

040-831-048, 040825359

<program> ->

PLATYPUS {<opt\_statements>}

First sets: you do not need to write that

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

<opt\_statements> ->

statements | E

Where is the first set for opt\_

FIRST (<opt\_statements>) ={AVID,SVID,KW\_T(IF),KW\_T(WHILE),KW\_T(READ),KW\_T(WRITE), E};

<statements> ->

<statement> | <statements> <statement>

Eliminating Left recursion:

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

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

<statements’> -> <statement><statements’> | E

FIRST(<statements’>) =

{AVID,SVID, KW\_T(IF),KW\_T(WHILE),KW\_T(READ),KW\_T(WRITE), E}

**3.2 Statements**

<statement> ->

<assignment statement>

| <selection statement>

| <iteration statement>

| <input statement>

| <output statement>

First sets:

FIRST(<statement>) =

{AVID,SVID,KW\_T(IF),KW\_T(WHILE),KW\_T(READ),KW\_T(WRITE)}

* + 1. **Assignment Statement**

<assignment statement> ->

<assignment expression>;

First sets:

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

< assignment expression> ->

AVID = <arithmetic expression>

| SVID = <string expression>

First sets:

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

**3.2.2 Selection Statement( the if statement)**

<selection statement> ->

IF TRUE (<conditional expression>) THEN { <opt\_statements> }

ELSE { <opt\_statements> } ;

First sets:

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

**3.2.3 Iteration Statement (the loop statement)**

<iteration statement> ->

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

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

First sets:

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

**<**pre-condition> ->

TRUE | FALSE

First sets:

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

**3.2.4 Input Statement**

<input statement> ->

READ (<variable list>);

First sets:

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

<variable list> ->

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

Eliminating Left recursion:

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

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

First sets :

FIRST(<variable list>)= {AVID, SVID};

FIRST(<variable list’>)= {, ,E};

**3.2.5 Output Statement**

<output statement> ->

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

| WRITE(STR\_T);

left factoring:

WRITE (<full\_list>);

<full\_list> -> <variable list> | SRT\_T| E

First sets:

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

FIRST(<full\_list>) = {AVID, SVID, SRT\_T, E};

**3.3 Expressions**

**3.3.1 Arithmetic Expression**

<arithmetic expression> - >

<unary arithmetic expression>

| <additive arithmetic expression>

First sets:

FIRST(<arithmetic expression>) = { ART\_OP\_T(-), ART\_OP\_T(+), LPR\_T, AVID, FPL\_T, INL\_T};

Use LPR\_T – not LBR\_(() LBR is left brace, not left parenthesis. See token.h.

<unary arithmetic expression> ->

- <primary arithmetic expression>

| + <primary arithmetic expression>

First sets:

FIRST(<unary arithmetic expression>) = { ART\_OP\_T(-),ART\_OP\_T(+) };

<additive arithmetic expression> ->

<additive arithmetic expression> + <multiplicative arithmetic expression>

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

| <multiplicative arithmetic expression>

Eliminating left recursion:

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

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

First sets:

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

FIRST(<additive arithmetic expression’>) = { ART\_OP\_T(+), ART\_OP\_T(-), E};

<multiplicative arithmetic expression> ->

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

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

| <primary arithmetic expression>

Eliminating left recursion:

<multiplicative arithmetic expression> -> <primary arithmetic expression><multiplicative arithmetic expression’>

<multiplicative arithmetic expression’>-> \*<primary arithmetic expression><multiplicative arithmetic expression’> | / <primary arithmetic expression><multiplicative arithmetic expression’> | E

First sets:

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

FIRST(<multiplicative arithmetic expression’>) = {ART\_OP\_T(\*), ART\_OP\_T(/), E}

<primary arithmetic expression> ->

AVID\_T

| FPL\_T

| INL\_T

| (<arithmetic expression>)

First sets:

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

**3.3.2 String Expression**

<string expression> ->

<primary string expression>

| <string expression> # <primary string expression>

Eliminating left recursion:

<string expression> -> <primary string expression><string expression’>

<string expression’> ->#<primary string expression><string expression’> | E

First sets:

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

FIRST(<string expression’>) = {SCC\_OP\_T, E};

<primary string expression> ->

SVID\_T

| STR\_T

First sets:

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

**3.3.3 Conditional Expression**

<conditional expression> ->

<logical OR expression>

First sets:

FIRST(<conditional expression>) = {AVID\_T, FPL\_T, INL\_T};

<logical OR expression> ->

<logical AND expression>

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

Eliminating left recursion:

<logical OR expression> ->

<logical AND expression><logical OR expression’>

<logical OR expression’>->.OR.<logical AND expression><logical OR expression’> | E

First sets:

FIRST(<logical OR expression>) = {AVID\_T, FPL\_T, INL\_T};

FIRST(<logical OR expression’>) = {LOG\_OP\_T(OR), E}

<logical AND expression> ->

<relational expression>

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

Eliminating left recursion:

<logical AND expression> -><relational expression><logical AND expression’>

<logical AND expression’>->.AND.<relational expression><logical AND expression’> | E

First sets:

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

FIRST(<logical AND expression’>) = {LOG\_OP\_T (AND), E};

**3.3.4 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>

left factoring:

<relational expression>->

<primary a\_relational expression><relational operator><primary a\_relational expression> |<primary s\_relational expression><relational operator><primary s\_relational expression>

<relational operator> -> == | <> | > | <

First sets:

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

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

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

FIRST(<relational operator>) = { REL\_OP\_T (==) , REL\_OP\_T (<>),REL\_OP\_T(>),REL\_OP\_T(<) }; one more

<primary a\_relational expression> ->

AVID\_T

| FPL\_T

| INL\_T

First sets:

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

<primary s\_relational expression> ->

<primary string expression>

First sets:

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