Vitor Cortes de Souza & Zaid Versey

PLATYPUS LANGUAGE SYNTACTIC GRAMMAR SPECIFICATION

BLACK – Original Grammar
RED – Description of transformation
BLUE – The Transformed Grammar
PURPLE – FIRST set

PLATYPUS Syntactic Specification:

3.1 PLATYPUS Program

<program> -> PLATYPUS {<opt_statements>} SEOF
FIRST(<program>) = {KW_T(PLATYPUS)}
<opt_statements> -> <statments> | ε
FIRST(<opt_statements>) = { AVID_T, SVID_T, KW_T(IF), KW_T(WHILE), KW_T(READ), KW_T(WRITE), ε }
<statements> -> <statement> | <statements> <statement>
Rearrange:
<satements> -> <statements><statement> | <statement>
Eliminate Left Recursion:
<statements> -> <statement><statements'>
FIRST(<statements>) = {AVID_T, SVID_T, KW_T(IF), KW_T(WHILE), KW_T(READ), KW_T(WRITE)}
<statements'> -> <statement><statements'> | ε
FIRST(<statements'>) = {AVID_T, SVID_T, KW_T(IF), KW_T(WHILE), KW_T(READ), KW_T(WRITE), ε}

3.2 Statements

<statement> -> <assignment statement> | <selection statement> |<iteration statement> |<input statement> |<output statement>
FIRST(<statement>) = {AVID, SVID, KW_T(IF), KW_T(WHILE), KW_T(READ), KW_T(WRITE)}

3.2.1 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 }

3.2.2 Selection Statement

<selection statement> ->
IF TRUE (<conditional expression>) THEN { <opt_statements> }
ELSE { <opt_statements> } ;
FIRST(<selection statement>) = { KW_T(IF) }

3.2.3 Iteration 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) }

3.2.4 Input Statement

<input statement> -> READ (<variable list>);
FIRST(<input statement>) = { KW_T(READ) }
<variable list> -> <variable identifier> | <variable list> , <variable identifier>
Rearrange:
<variable list> -> <variable list> , <variable identifier> | <variable identifier>
Eliminate Left Recursion:
<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}
<variable indentifier> -> AVID_T | SVID_T
FIRST(<variable identifier>) = { AVID_T, SVID_T }

3.2.5 Output Statement

<output statement> ->WRITE(<opt_variable list>); | WRITE (STR_T);
Left Factoring to Remove NFA
<output statement> -> WRITE (<output statement'>);
FIRST(<output statement>) = { KW_T(WRITE) }
<output statement'> -> <opt_variable list> | STR_T
FIRST(<output statement'>) = { AVID_T, SVID_T, ε, STR_T }
<opt_variable list> -> <variable list> | ε
FIRST(<opt_variable list>) = { AVID_T, SVID_T, ε}

3.3 Expressions:

3.3.1 Arithmetic Expression

<arithmetic expression> - >
<unary arithmetic expression>
| <additive arithmetic expression>
FIRST(<arithmetic expression>) = { ART_OP_T(-), ART_OP_T(+), AVID_T, FPL_T, INL_T, LPR_T }

<unary arithmetic expression> ->
- <primary arithmetic expression>
| + <primary arithmetic expression>
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>
Eliminate Left Recursion for <additive arithmetic expression>:
<additive arithmetic expression> ->
<multiplicative arithmetic expression> <additive arithmetic expression'>

FIRST(<additive arithmetic expression>) = { AVID_T, FPL_T, INL_T, LPR_T }

<additive arithmetic expression'> ->
+ <multiplicative arithmetic expression> <additive arithmetic expression'>
| - <multiplicative arithmetic expression> <additive arithmetic expression'>
| ε
FIRST(<additive arithmetic expression'>) = { ART_OP_T(+),ART_OP_T(-), ε }

<multiplicative arithmetic expression> ->
<multiplicative arithmetic expression> * <primary arithmetic expression>
| <multiplicative arithmetic expression> / <primary arithmetic expression>
| <primary arithmetic expression>
Eliminate Left Recursion for <multiplicative arithmetic expression>:
<multiplicative arithmetic expression> ->
<primary arithmetic expression> <multiplicative arithmetic expression'>
FIRST(<multiplicative arithmetic expression>) = { AVID_T, FPL_T, INL_T, LPR_T }
<multiplicative arithmetic expression'> ->
* <primary arithmetic expression> <multiplicative arithmetic expression'>
| / <primary arithmetic expression> <multiplicative arithmetic expression'>
| ε
FIRST(<multiplicative arithmetic expression'>) = { ART_OP_T(*), ART_OP_T(/), ε }

<primary arithmetic expression> -> AVID_T | FPL_T | INL_T | (<arithmetic expression>)
FIRST (<primary arithmetic expression>) = { AVID_T, FPL_T, INL_T, LPR_T }

3.3.2 String Expression

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

Rearrange:
<string expression> -> <string expression> # <primary string expression>
| <primary string expression>
Eliminate Left Recursion:
<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'>) = { SCC_OP_T, ε}
<primary string expression> -> <SVID_T> | <STR_T>
FIRST (<primary string expression>) = { SVID_T, STR_T }

3.3.3 Conditional Expression

<conditional expression> -> <logical OR expression>
FIRST (<conditional expression>) = { AVID_T, FPL_T, INL_T, SVID_T, STR_T }

<logical OR expression> -> <logical AND expression>
| <logical OR expression> .OR. <logical AND expression>
Rearrange logical OR expression:
<logical OR expression> -> <logical OR expression> .OR. <logical AND expression>
| <logical AND expression>
Eliminate Left Recursion for logical OR expression:
<logical OR expression> -> <logical AND expression> <logical OR expression'>
FIRST (<logical OR 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'>) = { LOG_OP_T(.OR.) , ε}

<logical AND expression> -> <relational expression>
| <logical AND expression> .AND. <relational expression>
Rearrange logical AND expression:
<logical AND expression> -> <logical AND expression> .AND. <relational expression>
| <relational expression>
Eliminate Left Recursion for logical AND 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'>) = {LOG_OP_T(.AND.) , ε}

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 to Eliminate NFA for relational expression:

<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>

FIRST (<primary a_relational expression'>) = { REL_OP_T(EQ), REL_OP_T(NE), REL_OP_T(LT), REL_OP_T(GT) }

<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'>) = { REL_OP_T(EQ), REL_OP_T(NE), REL_OP_T(LT), REL_OP_T(GT) }

<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 }