

# PLATYPUS LANGUAGE SPECIFICATION[[1]](#footnote-1)

### General View

***Grammar, which knows how to control even kings . . .***

*—*Molière*, Les Femmes Savantes* (1672), Act II, scene vi

A context-free grammar is used to define the lexical and syntactical parts of the **PLATYPUS** language and the lexical and syntactic structure of a **PLATYPUS** program.

1. **The PLATYPUS 3.0 Lexical Specification**
   1. **White Space**

White spaceis defined as the ASCII space, horizontal and vertical tabs, and form feed characters, as well as line terminators. White space is discarded by the scanner.

**<white space>** → *one of* { SPACE TAB FF NL CR NLCR }

* 1. **Comments**

PLATYPUS supports only single-line comments: all the text from the ASCII characters **%%** to the end of the line is ignored by the scanner.

**<comments>** → **%%** { sequence of ASCII chars }

* 1. **Variable Identifiers**

The following variable identifier (VID) tokens are produced by the scanner: AVID\_T and SVID\_T.

**<variable identifier>** → AVID\_T | SVID\_T

* 1. **Keywords**

The scanner produces a single token: **KW\_T**. The type of the keyword is defined by the attribute of the token (the index of the keywordTable []).

* 1. **Integer Literals**

The scanner produces a single token: **INL\_T** with an integer value as an attribute.

* 1. **Floating-point Literals**

**FPL\_T** token with a real decimal value as an attribute is produced by the scanner.

* 1. **String Literals**

**STR\_T** token is produced by the scanner. The attribute is the string literal offset (currentToken.attribute.str\_offset) from the beginning of the string literal buffer (stringLiteralTable->string).

* 1. **Separators**

**<separator>** → *one of* {( ){ } , ; }

Seven different tokens are produced by the scanner - **LPR\_T**, **RPR\_T**, **LBR\_T**, **RBR\_T**, **COM\_T**, **EOS\_T**.

* 1. **Operators**

**<arithmetic operator>** → *one of* { +, -, \*, / }

A single token is produced by the scanner: **ART\_OP\_T**. The type of the operator is defined by the attribute of the token.

**<string concatenation operator>** → ++

A single token is produced by the scanner: **SCC\_OP\_T**.

**<relational operator>** → *one of* { >, <, ==, != }

A single token is produced by the scanner: **REL\_OP\_T**. The type of the operator is defined by the attribute of the token.

**<logical operator>** → *one of* { .AND., .OR., .NOT. }

A single token is produced by the scanner: **LOG\_OP\_T**. The type of the operator is defined by the attribute of the token.

**<assignment operator>** → =

A single token is produced by the scanner: **ASS\_OP\_T**.

1. **The PLATYPUS Syntactic Specification**
   1. **PLATYPUS Program**
      1. **Program**

**<program>** → **MAIN** {<opt\_statements>}

**First Set**

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

**Optional Statements:**

**<opt\_statements>** → <statements> | ϵ

**First Set**

TODO\_01 First(<opt\_statements>) = {AVID\_T,SVID\_T,KW\_T(IF),KW\_T(WHILE),KW\_T(READ)

KW\_T(WRITE), Ε}

* + 1. **Statements**

**<statements>** → <statement> | <statements> <statement>

* **PROBLEM DETECTED: Left recursion (SOLVED for you here):**

**New Grammar**

**<statements>** → <statement><statements Prime>

**<statementsPrime>** → <statement><statements Prime> | ϵ

**First Set**

TODO\_02 First(<statements>)= {AVID\_T,SVID\_T,KW\_T(IF),KW\_T(WHILE),KW\_T(READ)

KW\_T(WRITE)}

First(<statementsPrime>) = {{AVID\_T, SVID\_T, KW\_T(IF),KW\_T(WHILE),KW\_T(READ)

KW\_T(WRITE), Ε}

* 1. **Statements**

**<statement>** → <assignment statement> | <selection statement> | <iteration statement>

| <input statement> | <output statement>

02 First(<statement>)={AVID\_T,SVID\_T,KW\_T(IF),KW\_T(WHILE),KW\_T(READ)

KW\_T(WRITE)}

**First Set**

TODO\_03

* + 1. **Assignment Statement**

**<assignment statement>** → <assignment expression>;

**First Set**

TODO\_04 First (<assignment statement>) = {AVID\_T,SVID\_T}

* + 1. **Assignment Expression**

**<assignment expression>** → AVID = <arithmetic expression> | SVID = <string expression>

**First Set**

TODO\_05 First(<assignment expression>) = {AVID\_T,SVID\_T}

* + 1. **Selection Statement (if statement)**

**<selection statement>** → IF <pre-condition> (<conditional expression>)

THEN { <opt\_statements> }

ELSE { <opt\_statements> } ;

**First Set**

TODO\_06 First(<selection statement>) = {KW\_T(IF)}

* + 1. **Iteration Statement (the loop statement)**

**<iteration statement>** → WHILE <pre-condition> (<conditional expression>)

DO { <statements>};

**First Set**

TODO\_07 First(<iteration statement>) = {KW\_T(WHILE)}

**<pre-condition>** → TRUE | FALSE

**First Set**

TODO\_08 First(<pre-condition>) = {KW\_T(TRUE), KW\_T(FALSE)}

* + 1. **Input Statement**

**<input statement>** → READ (<variable list>);

**First Set**

TODO\_09 KW\_T(<input statement>) = {KW\_T(READ)}

**Variable List:**

**<variable list>** → <variable identifier> | <variable list>,<variable identifier>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

TODO\_10 <variable list> 🡪 <variable identifier> <variable list Prime>

<variable list Prime> 🡪 <variable identifier> < variable lis tPrime>|Ε

**First Set**

TODO\_11 First(<variable list>) = { AVID\_T, SVID\_T}

First(<variable list Prime>) = { AVID\_T, SVID\_T,Ε}

**Variable Identifier:**

**<variable identifier>** →AVID\_T | SVID\_T

**First Set**

TODO\_12 First(<variable identifier>) = {AVID\_T, SVID\_T}

* + 1. **Output Statement**

**<output statement>** → WRITE (<opt\_variable list>); | WRITE (STR\_T);

* **PROBLEM DETECTED: Left factoring (SOLVED for you here):**

**New Grammar**

<output statement>→WRITE (<output statement Prime>);

<output statement Prime>→ <opt\_variable list> | STR\_T

**First Set**

TODO\_13 First (<output statement>) = {KW\_T(WRITE)}

**Optional Variable List:**

**<opt\_variable list>** →<variable list> | ϵ

**First Set**

TODO\_14 First(<opt variable list>)= {AVID\_T,SVID\_T}

* **Note:** In some cases, the grammar may be transformed to predictive grammar without applying the general rule. For example, the grammar above can be rewritten as follows.
* **Rewriting the grammar:**

**New Grammar**

<output statement>→OUTPUT (<output list>);

<output\_list>→ <opt\_variable list> | STR\_T

**First Set**

TODO\_15 First(<output statement>) = { KW\_T(OUTPUT)}

First(<output list>)= {AVID\_T,SVID\_T,STR\_T,Ε}

* 1. **Expressions**
     1. **Arithmetic Expression**

**<arithmetic expression>** → <unary arithmetic expression> | <additive arithmetic expression>

**First Set**

TODO\_16 First(<arithmetic expression>) = { -,+,AVID\_T, FLP\_T, INL\_T, (}

**Unary Arithmetic Expression:**

**<unary arithmetic expression>** → - <primary arithmetic expression>

| + <primary arithmetic expression>

**First Set**

TODO\_17 First(<unary arithmetic expression>) = { -, +}

**Additive Arithmetic Expression:**

**<additive arithmetic expression>** →

<additive arithmetic expression> + <multiplicative arithmetic expression>

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

| <multiplicative arithmetic expression>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

TODO\_18 <additive arithmetic expression> 🡪 <multiplicative arithmetic expression>

<additive arithemetic expression Prime>

<additive arithmetic expression Prime> 🡪 + <multiplicative arithmetic expression> <additive Arithmetic expression> < additive arithmetic expression Prime> | - <multiplicative arithmetic expression>

<additive arithmetic expression Prime>| Ε

**First Set**

TODO\_19 First(<additive arithmetic expression>) = {AVID\_T, FLP\_T,INL\_T,(}

First(<additive arithmetic expression Prime>) = {+,-,Ε}

**Multiplicative Arithmetic Expression:**

**<multiplicative arithmetic expression>** →

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

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

| <primary arithmetic expression>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

TODO\_20 <multiplicative arithmetic expression> 🡪 <primary arithmetic expression>

<multiplicative Arithmetic expression Prime>

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

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

< multiplicative arithmetic expression Prime>| Ε

**First Set**

TODO\_21 First (<multiplicative arithmetic expression>) = {AVID\_T, FLP\_T,INL\_T,(}

First (<multiplicative arithmetic expression Prime>) = {\*,/, Ε}

**Primary Arithmetic Expression:**

**<primary arithmetic expression>** → AVID\_T | FPL\_T | INL\_T

| (<arithmetic expression>)

**First Set**

TODO\_22 First(<primary arithmetic expression>) = {AVID\_T, FPL\_T, INL\_T, (}

* + 1. **String Expression**

**<string expression>** →

<primary string expression> | <string expression> ++ <primary string expression>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

TODO\_23 <string expression> 🡪 <primary string expression> <string expression Prime>

<string expression Prime> 🡪 ++<primary string expression> <string expression Prime>|Ε

**First Set**

TODO\_24 First(<string expression>) = { SVID\_T, STR\_T}

First(<string expression Prime>) = { ++, Ε}

**Primary String Expression:**

**<primary string expression>** → SVID\_T | STR\_T

**First Set**

TODO\_25 First(<primary string expression>) = { SVID\_T, STR\_T}

* + 1. **Conditional Expression**

**<conditional expression>** → <logical OR expression>

**First Set**

TODO\_26 First(<conditional expression>) = {AVID\_T, INL\_T, FPL\_T, SVID\_T, STR\_T}

**Logical OR Expression:**

**<logical OR expression>** → <logical AND expression>

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

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

TODO\_27 <logical OR expression> 🡪 <logical AND expression> <logical OR expression Prime>

<logical OR expression Prime> 🡪 .OR. <logical AND expression> <logical OR expression Prime>

**First Set**

TODO\_28 First (<logical OR expression>) = {AVID\_T, INL\_T, FPL\_T, SVID\_T, STR\_T}

First(<logical OR expression Prime>) = {.OR., Ε}

**Logical AND Expression:**

**<logical AND expression>** → <logical NOT expression>

| <logical AND expression> **.AND.** <logical NOT expression>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

TODO\_29 <logical AND expression> 🡪 <logical NOT expression> <logical AND expression Prime>

<logical AND expression Prime> 🡪 .AND. <logical NOT expression> < logical AND expression Prime>

**First Set**

TODO\_30 First(<logical AND expression>) = { AVID\_T,FLP\_T,INL\_T,SVID\_T,STR\_T}

First(<logical AND expression Prime>) = {.AND., Ε}

**Logical NOT Expression:**

**<logical NOT expression>** → **.NOT.** <relational expression>

| <relational expression>

**First Set**

TODO\_31 First(<logical NOT expression>) = {.NOT., AVID\_T,FLP\_T,INL\_T,SVID\_T,STR\_T}

* + 1. **Relational Expression**

**<relational expression>** →

<relational a\_expression> | <relational string expression>

**First Set**

TODO\_32 First(<relational expression>) = {AVID\_T,FLP\_T,INL\_T,SVID\_T,STR\_T}

**Relational Arithmetic Expression:**

**<relational a\_expression>** →

<primary Arithmetic relational expression> == <primary arithmetic relational expression>

| <primary arithmetic relational expression> != <primary arithmetic relational expression>

| <primary arithmetic relational expression> > <primary arithmetic relational expression>

| <primary arithmetic relational expression> < <primary arithmetic relational expression>

* **PROBLEM DETECTED: Left factoring:**

**New Grammar**

TODO\_33 <relational a\_expression> 🡪 <primary arithmetic relational expression> <relation al\_ expression Prime>

< relational a\_ expression Prime> 🡪 == <primary arithmetic relational expression> | != <primary arithmetic relational expression> | > <primary arithmetic relational expression> | < <primary arithmetic relational expression>

**First Set**

TODO\_34 First(<relational a\_expression>) = {AVID\_T,FLP\_T, INL\_T}

First< relational a\_ expression Prime>) = { == , !=, > , < }

**Primary Arithmetic Relational Expression:**

**<primary arithmetic relational expression>** →AVID\_T | FPL\_T | INL\_T

**First Set**

TODO\_35 First(<primary arithmetic relational Expression>) = { AVID\_T, FPL\_T, INL\_T}

**Relational String Expression:**

**<relational string expression>** →

| <primary String relational expression> == <primary String relational expression>

| <primary String relational expression> != <primary String relational expression>

| <primary String relational expression> > <primary String relational expression>

| <primary String relational expression> < <primary String relational expression>

* **PROBLEM DETECTED: Left factoring:**

**New Grammar**

TODO\_36 <relational string expression> 🡪 <primary string expression> <relational string expression Prime>

<relational string expression Prime> 🡪 == <primary String relational expression> | != <primary String relational expression> | > <primary String relational expression>| < <primary String relational expression>

**First Set**

TODO\_37 First(relational string expression>) = {SVID\_T, STR\_T}

First(relational string expression Prime>) = { ==, != , > , < }

**Primary String Relational Expression:**

**<primary String relational expression>** → <primary string expression>

**First Set**

TODO\_38 First(<primary String relational expression>) = {SVID\_T, STR\_T}

**Good luck with Assignment 3!**

1. Adapted from resources developed by Prof. Svillen Ranev (Algonquin College, 2019) [↑](#footnote-ref-1)