PLATYPUS LANGUAGE SPECIFICATION

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.

**1.1 Context-Free Grammars**

A context-free grammar (CFG), (often called Backus Normal Form or Backus-Naur

Form (BNF) grammar, consists of four finite sets: a finite set of terminals; a finite set of

nonterminals; a finite set of productions; and a start or a goal symbol.

One of the sets consists of a finite number of productions (called also replacement rules,

substitution rules, or derivation rules). Each production has an abstract symbol called a

nonterminal as its left-hand side, and a sequence of one or more nonterminal and

terminal symbols as its right-hand side. For each grammar, the terminal symbols are

drawn from a specified alphabet. Starting from a sentence consisting of a single

distinguished nonterminal, called the start symbol, a given context-free grammar specifies

a language, namely, the infinite set of possible sequences of terminal symbols that can

result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a

production for which the nonterminal is the left-hand side.

**1.2 Grammar Notation**

Terminal symbols are shown in normal font in the productions of the lexical and syntactic

grammars, and throughout this specification whenever the text is directly referring to such a

terminal symbol. These are to appear in a program exactly as written. Nonterminal symbols

are shown in triangular brackets <nonterminal> for ease of recognition. However,

nonterminals can also be recognized by the fact that they appear on the left-hand sides of

productions. The definition of a nonterminal is introduced by the name of the nonterminal

being defined followed by a -> sign. One or more alternative right-hand sides for the

nonterminal then follow on succeeding line(s) preceded by a |. The symbol .

ÎÎÎÎ will represent

the empty or null string. Thus, a production A -> .

ÎÎÎÎ states that A can be replaced by the

empty string, effectively erasing it.

When the words “one of ” follow the -> in a grammar definition, they signify that each of the

terminal symbols on the following line or lines is an alternative definition. For example, the

production:

<small letters from a to c> -> one of

abc

is not a standard BNF operation but is merely a convenient abbreviation for:

<small letters from a to c> ->

a|b|c

The right-hand side of a lexical production may specify that certain expansions are not

permitted by using the phrase “but not” and then indicating the expansions to be excluded,

as in the productions for <input character>

< input character > -> one of

ASCII characters but not SEOF

Another non-standard BNF notation is the prefix opt\_, which may appear before a terminal

or nonterminal. It indicates an optional symbol or element. The alternative containing the

opt\_ symbol actually specifies two right-hand sides, one that omits the optional element

and one that includes it. This means that:

<program> ->

PLATYPUS { <opt\_statements> }

is merely a convenient abbreviation for:

<program> ->

PLATYPUS { <statements> }

| PLATYPUS { }

In this case the implied production for the optional element is:

<opt\_statements> ->

<statements> | Î

**2. The PLATYPUS Lexical Specification**

The purpose of the lexical grammar is to describe how sequences of ASCII characters are

translated into a sequence of input elements. These input elements, called lexemes, are

recognized by the lexical analyzer (scanner) and converted into tokens, which serve as

terminal symbols for the syntactic grammar for PLATYPUS. Some of the input elements

(lexemes) like white space and comments are discarded by the scanner. The tokens of the

PLATYPUS language are variable identifier, keyword, integer literal, floating-point literal,

string literal, separator, and operator. Tokens (except for string literals), that is, variable

identifiers, integer literals, floating-point literals, keywords and two-character operators may

not extend across line boundaries.

**2.1 Input Elements and Tokens**

< input character > -> one of

ASCII characters but not SEOF

<input element > ->

<white space > | <comment> | <token>

<token> ->

<variable identifier> | <keyword> | <floating-point literal >

| <integer literal > | <string literal> | <separator> | <operator>

**2.2 White Space**

White space is 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.

**2.3 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.

**2.4 Variable Identifiers**

A variable identifier is a sequence of ASCII letters and ASCII digits, the first of which must

be a letter and the last of which may be a number sign (#).A variable identifier (VID) can be

of any length but only the first 8 characters (including the number sign if present) are

significant. There are two types of variable identifiers: arithmetic and string. They represent

the language arithmetic data types and the textual data type correspondingly. Identifiers

cannot have the same spelling (lexeme) as a keyword.

A variable is a storage location and has an associated data type. The PLATYPUS language

supports only three data type: integer, floating-point and string data type. Variable identifiers

are used to represent floating-point, integer or string variables. Determining the type of the

arithmetic variable (integer or the floating-point) is not built in the grammar but left to the

implementation.

<variable identifier> ->

<arithmetic variable identifier> | <string variable identifier>

<string variable identifier> ->

<arithmetic variable identifier>#

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

SVID\_T.

<variable identifier> -> AVID\_T | SVID\_T

2.5 Keywords

The following character sequences, formed from ASCII letters, are reserved for

use as keywords and cannot be used as identifiers:

<keyword> ->

PLATYPUS | IF | THEN | ELSE | USING | REPEAT | INPUT | OUTPUT

The scanner produces a single token: KW\_T. The type of the keyword is defined by the

attribute of the token.

2.6 Integer Literals

An integer literal (constant) is the source code representation of an integer decimal value

or integer number. The PLATYPUS language supports two types of integer literal

representation: decimal integer literal and octal integer literal.

The internal (machine) size of an integer number must be 2 bytes. The literals by default

are non-negative, but their sign can be changed at run-time by applying unary sign

arithmetic operation.

<integer literal> ->

<decimal integer literal> | <octal integer literal>

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

2.7 Floating-point Literals

A floating-point literal is the source code representation of a fixed decimal value. The

numbers must be represented internally as floating-point numbers. The internal size must

be 4 bytes. The literals by default are non-negative, but their sign can be changed at runtime

by applying unary sign arithmetic operation.

<floating-point literal> ->

<decimal integer literal> . <opt\_digits>

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

2.8 String Literals

A string literal is a sequence of ASCII characters (including no characters at all) enclosed

in double quotation marks. The quotation mark and the source-end-of-file character SEOF

cannot be a string character. SEOF is implementation dependent.

STR\_T token is produced by the scanner.

2.9 Separators

The following seven ASCII characters are the PLATYPUS separators (punctuators):

<separator> -> one of

( ){ } , ; “

Seven different tokens are produced by the scanner.

2.10 Operators

The following tokens are the PLATYPUS operators, formed from ASCII characters:

<operator> ->

< arithmetic operator > | <string concatenation operator>

| < relational operator> | < logical operator >

|< assignment operator >

<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> ->

.AND. | .OR.

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.

**3 The PLATYPUS Syntactic Specification**

The syntactic grammar for PLATYPUS is given below. This grammar has PLATYPUS

tokens defined by the lexical grammar as its terminal symbols. For the sake of readability,

the corresponding lexemes are used in lieu of the tokens for keywords, separators, and

operators. For example, the lexeme + is used instead of ART\_OP\_T with an attribute

PLUS. The syntactic grammar is to define a set of productions -starting from the start

symbol <program> -that describe how sequences of tokens can form syntactically correct

PLATYPUS programs

**3.1 PLATYPUS Program**

A PLATYPUS program is a sequence of statements -no statements at all, one statement, or more than one statement, enclosed in braces { }. The compilation unit is a single file containing one program and terminated by the SEOF character (SEOF\_T token).

<program> ->

PLATYPUS {<opt\_statements>}

<program> => FIRST(PLATYPUS)

<statements> ->

<statement> | <statements> <statement>

<statements> => FIRST(AVID, SVID, IF, USING, INPUT, OUTPUT)

<statements’> => FIRST(AVID, SVID, IF, USING, INPUT, OUTPUT, ε)

**3.2 Statements**

The sequence of execution of a PLATYPUS program is controlled by statements. Some

statements contain other statements as part of their structure; such other statements are

substatements of the statement. PLATYPUS supports the following five types of

statements: assignment, selection, iteration, input and output statements.

<statement> -> <assignment statement> | <selection statement> | <iteration statement>

| <input statement> | <output statement>

<statement> => FIRST(AVID, SVID, IF, USING, INPUT, OUTPUT)

**3.2.1 Assignment Statement**

<assignment statement> -> <assignment expression>;

< assignment expression> ->

AVID = <arithmetic expression>

| SVID = <string expression>

<assignment statement> => FIRST(AVID, SVID)

<assignment expression> => FIRST(AVID, SVID)

The assignment statement is evaluated in the following order. First, the assignment

expression on the right side of the assignment operator is evaluated. Second, the result

from the evaluation is stored into the variable on the left side of the assignment operator.

If the assignment expression is of arithmetic type and the data types of the variable and

the result are different, the result is converted to the variable type implicitly.

String expressions operate on strings only and no conversions are allowed.

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

The selection statement is an alternative selection statement, that is, there are two

possible selections.

It the conditional expression evaluates to true, the statement(s) included in the THEN clause are

executed. It the conditional expression evaluates to false, the statement(s) included in the ELSE

clause are executed. The THEN clause may be empty – no statements at all. The ELSE clause is not optional but may be empty.

<selection statement> ->

IF (<conditional expression>) THEN <opt\_statements>

ELSE { <opt\_statements> } ;

<selection statement> => FIRST(IF)

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

The iteration statement is used to implement iteration control structures. The loop

statement executes the loop body statement(s) repeatedly until the value of the

conditional expression is false.

<iteration statement> ->

USING (<assignment expression> , <conditional expression> , <assignment expression> )

REPEAT {

< opt\_statements>

};

<iteration statement> => FIRST(USING)

The USING-REPEAT statement executes the following steps:

UR1. The first assignment expression is evaluated.

UR2. The conditional expression is evaluated. If true, the body of the loop specified by the

REPEAT clause is executed. The loop body can be empty (that is, {} is a legal loop

body). If false, the loop is terminated.

UR3. The second assignment expression is evaluated and step UR2 is repeated.

**3.2.4 Input Statement**

The input statement reads a floating-point, an integer or a string literal from the standard

input and stores it into a floating-point, an integer variable or a string variable.

<input statement> -> INPUT (<variable list>);

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

<input statement> => FIRST(INPUT)

**3.2.5 Output Statement**

The output statement writes a variable list or a string to the standard output. Output

statement with an empty variable list prints an empty line.

<output statement> ->

OUTPUT (<opt\_variable list>);

| OUTPUT (STR\_T);

<output statement> => FIRST(OUTPUT)

**3.3 Expressions**

Most of the work in a PLATYPUS program is done by evaluating expressions, either for

their side effects, such as assignments to variables, or for their values, which can be used

as operands in larger expressions, or to affect the execution sequence in statements, or

both.

This section specifies the meanings of PLATYPUS expressions and the rules for their

evaluation.

An expression is a sequence of operators and operands that specifies a computation.

When an expression in a PLATYPUS program is evaluated (executed), the result denotes

a value. There are four of expressions in the PLATYPUS language: arithmetic expression,

string expressions, relational expressions, and conditional expression.

The expressions are always evaluated from left to right.

**3.3.1 Arithmetic Expression**

An arithmetic expression is an infix expression constructed from arithmetic variables,

arithmetic literals, and the operators plus (+), minus (-), multiplication (\*), and division (/).

The arithmetic expression always evaluates either to a floating-point value or to an integer

value. Mixed type arithmetic expressions and mixed arithmetic assignments are allowed.

The data type of the result of the evaluation is determined by the data types of the

operands. If there is at least one floating-point operand, all operands are converted to

floating-point type, the operations are preformed as floating-point, and the type of the

result is floating-point.

The type conversion (coercion) is implicit. All operators are left associative. Plus and

minus operators have the same order of precedence. Multiplication and division have the same order of precedence but they have a higher precedence than plus and minus

operators. Plus and minus can be used as unary operator to change the sign of a value.

In this case they have the highest order of precedence and they are evaluated first.

The formal syntax of the arithmetic expression is listed below.

<arithmetic expression> ->

<unary arithmetic expression>

| <additive arithmetic expression>

<unary arithmetic expression> ->

-<primary arithmetic expression>

| + <primary arithmetic expression>

<additive arithmetic expression> ->

<additive arithmetic expression> + <multiplicative arithmetic expression>

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

| <multiplicative arithmetic expression>

<multiplicative arithmetic expression> ->

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

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

| <primary arithmetic expression>

<primary arithmetic expression> ->

AVID\_T

| FPL\_T

| INL\_T

| (<arithmetic expression>)

<arithmetic expression> => FIRST( + , - )

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

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

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

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

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

<primary arithmetic expression> => FIRST(AVID\_T, FPL\_T, INL\_T, + , - )

**3.3.2 String Expression**

A string expression is an infix expression constructed from string variables, string

literals, and the operator append or concatenation (<>). The string expression always

evaluates to a string (or a pointer to string). The append operator is left associative.

<string expression> ->

<primary string expression>

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

<primary string expression> ->

SVID\_T

| STR\_T

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

<string expression’> => FIRST( <> , ε )

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

**3.3.3 Conditional Expression**

A conditional expression is an infix expression constructed from relational expressions

and the logical operators .AND. and/or .OR.. The logical operator .AND. has a higher

order of precedence than .OR.. Parentheses are not allowed in the conditional

expressions, thus the evaluation order cannot be changed. All operators are left

associative

The conditional expressions evaluate to true or false. The internal representation of the

values of true and false are left to the implementation.

The formal syntax of the conditional expression follows.

<conditional expression> ->

<logical OR expression>

<logical OR expression> ->

<logical AND expression>

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

<logical AND expression> ->

<relational expression>

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

<logical OR expression> -> <logical AND expression><logical OR expression’>

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

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

LF

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

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

<conditional expression> => FIRST(OR, AND)

<logical OR expression> => FIRST(OR)

<logical OR expression’> => FIRST(OR, ε)

<logical AND expression> => FIRST(AND)

<logical AND expression’> => FIRST(AND, ε)

**3.3.4 Relational Expression**

A relational expression is an infix expression constructed from variable identifiers (VID),

literals (constants), and comparison operators (==, !=, <, >). The comparison operators

have a higher order of precedence than the logical operators do.

The relational expressions evaluate to true or false which are represented in PLATYPUS

with the integer literals 1 (true) and 0 (false).

The formal syntax of the relational expression follows.

<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>

<primary a\_relational expression> ->

AVID\_T

| FPL\_T

| INL\_T

<primary s\_relational expression> ->

<primary string expression>

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

<relational expression op> => FIRST(==, !=, <, > )

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

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