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 The PLATYPUS Lexical Grammar

A ***lexical grammar*** for **PLATYPUS** is given below. This grammar has as its terminal symbols the characters of the ASCII character set. It defines a set of productions, starting from the start symbol ***input***, that describe how sequences of ASCII characters are translated into a sequence of input elements. These input elements, with white space and comments discarded, form the terminal symbols for the syntactic grammar for PLATYPUS and are called PLATYPUS ***tokens***. These tokens are the variable identifier, keyword, integer literal, floating-point literal, string literal, separator, and operator of the PLATYPUS language.

1.3 The PLATYPUS Syntactic Grammar

The incomplete ***syntactic grammar*** for PLATYPUS is given below. This grammar has **PLATYPUS** tokens defined by the lexical grammar as its terminal symbols. It 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

1.4 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 ∈ will represent the empty or null string. Thus a production **A ->** ∈ 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*
a b c

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

The prefix **opt**_, which may appear before a terminal or nonterminal, indicates an ***optional symbol or element***. The alternative containing the optional 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 not a standard BNF operation but is merely a convenient abbreviation for:

<program> ->
PLATYPUS { <statements> }
| PLATYPUS { }

which in turn is abbreviation for:

<program> ->
PLATYPUS { <opt_statements> }

<opt_statements> ->
<statements> | ∈

2. The PLATYPUS Lexical Specification

This section specifies the lexical grammar (structure) of PLATYPUS.

PLATYPUS programs are single file programs written in ASCII. Lines are terminated by the ASCII characters CR, or LF, or by the combination CR LF. Source files are terminated by the SEOF character. This character is Control-Z in DOS or Control-D in UNIX. The ASCII characters are reduced to a sequence of input elements, which are white space, comments, and tokens. The tokens are the variable identifier, keyword, integer literal, floating-point literal, string literal, separator and operator of the PLATYPUS syntactic grammar.

2.1 Input Elements and Tokens

The input characters and line terminators that result from input line recognition are reduced to a sequence of <input elements>. Those input elements that are not white space or comments are ***tokens***. The tokens are the terminal symbols of the PLATYPUS syntactic grammar.

This process is specified by the following productions:

< input character > -> *one of*
ASCII characters *but not* SEOF

<input> ->
<input elements> SEOF

<input elements> ->
<input element> | <input elements> <input element>

<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 tab, and form feed characters, as well as line terminators.

<white space> ->
the ASCII SP character, also known as “space”
| the ASCII HT character, also known as “horizontal tab”
| the ASCII VT character, also known as “vertical tab”
| the ASCII FF character, also known as “form feed”
| <line terminator>

<line terminator> ->
CR | LF | CR LF

2.3 Comments

PLATYPUS supports only single-line comments: a comment is a text beginning with the comment prefix characters **!!** and ending with a line terminator. A comment is formally specified by the following grammar productions:

<comment> ->
!! <opt_characters in line> <line terminator>

<characters in line> ->
<comment character> | <characters in line> <comment character>

which actually means (see 1.4):

<comment> ->
!!<opt_characters in line> <line terminator>

<opt_characters in line>
<characters in line> | ∈

<characters in line> ->
<comment character> | <characters in line> <comment character>

<comment character> ->
<input character> but not <line terminator>

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 dollar sign ($).A *variable identifier* (***VID***) can be of any length but only the first 8 characters (not 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>

<arithmetic variable identifier> ->
<letter> <opt_letters or digits>

<letters or digits> ->
    <letter or digit> | <letters or digits> <letter or digit>

<letter> -> *one of*
    a … z A ... Z (uppercase and lowercase ASCII Latin letters A–Z)

<letter or digit> -> *one of*
    a … z A...Z 0…9 (ASCII digits 0-9)

<string variable identifier> ->
     <arithmetic variable identifier>$

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 | WHILE | REPEAT | READ | WRITE | TRUE | FALSE

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 deferent representations of integer literal: zero decimal integer literal and non-zero decimal 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>

<decimal integer literal> -> <zeros> | <non zero digit> <opt_digits>

<zeros> -> 0 | <zeros>0

<digits> ->   <digit> | <digits> <digit>

<digit> -> 0 | <non zero digit>

<non zero digit> one of
    1 2 3 4 5 6 7 8 9

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 run-time by applying unary sign arithmetic operation.

<floating-point literal> ->
<decimal integer literal> **.** <opt_digits>

2.8 String Literals

A ***string literal*** is a sequence of ASCII characters enclosed in double quotation marks. The " and source-end-of-file character SEOF cannot be a string character. SEOF is implementation dependent.

<string literal> ->
"<opt_string characters>"

<string characters> ->
<string character> | <string characters> <string character>
< string character > -> *one of*
ASCII characters *but not* SEOF "

2.9 Separators

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

<separator> -> *one of*
`( ) { } , ; " .`

They can only be used in a specific context defined by the grammar**.**

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*
`+ - * /`
<string concatenation operator> -> #

<relational operator> -> *one of*
`> < == <>`
<logical operator> -> `.AND. | .OR.`

<assignment operator> -> =

3 The PLATYPUS Syntactic Specification

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>}

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

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>

3.2.1 Assignment Statement

<assignment statement> ->
<assignment expression>;

< assignment expression> ->
AVID = <arithmetic expression>
| SVID = <string expression>

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.
If the *conditional expression* evaluates to true and the *pre-condition* is the keyword **TRUE**, the statements (if any) contained in the **THEN** clause are executed and the execution of the program continues with the statement following the selection statement. If the *conditional expression* evaluates to false, only the statement (if any) contained in the ELSE clause are executed and the execution of the program continues with the statement following the selection statement.
If the *conditional expression* evaluates to false and the *pre-condition* is the keyword **FALSE**, the statements (if any) contained in the **THEN** clause are executed and the execution of the program continues with the statement following the selection statement.
If the *conditional expression* evaluates to true, only the statement (if any) contained in the ELSE clause are executed and the execution of the program continues with the statement following the selection statement.
Both **THAN** and **ELSE** clauses must be present but may be empty – no statements at all.

<selection statement> ->
IF <pre-condition> (<conditional expression>) THEN { <opt_statements> }
ELSE { <opt_statements> } ;

3.2.3 Iteration Statement (the loop statement)

The ***iteration statement*** is used to implement iteration control structures. The ***iteration statement*** executes repeatedly the statements specified by the **REPEAT** clause of the **WHILE** loop depending on the *pre-condition* and *conditional expression*. If the *pre-condition* is the keyword **TRUE**, the statements are repeated until the evaluation of the *conditional expression* becomes false. If the *pre-condition* is the keyword **FALSE**, the statements are repeated until the evaluation of the *conditional expression* becomes true.

<iteration statement> ->
WHILE **<**pre-condition> **(<**conditional expression>**)**
REPEAT **{** <statements>**};**

<pre-condition> ->
TRUE | FALSE

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> ->
READ (<variable list>);

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

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> ->
    WRITE (<*opt_variable list*>);
  | WRITE (<string literal>);

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 performed 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> ->
<arithmetic variable identifier>
| <floating-point literal>
| <integer literal>
| (<arithmetic expression>)

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> ->
<string variable identifier>
| <string literal>

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>

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.

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> ->
<floating-point literal>
| <integer literal>
| <arithmetic variable identifier>

<primary s_relational expression> ->
<primary string expression>

Enjoy the PLATYPUS Grammar and do not forget that:

"If linguistics has any shallower goal than that of a deeper insight into the nature of mind, we would prefer to have nothing to do with it." Noam Chomsky
And remember to remember:
"There ain't no cure for love** for compilers **…" by Leonard Cohen
and also
"Language design is compiler construction" Nicolas Wirth (the creator of Pascal)

CST8152 – Compilers, 1 October 2018, S^R