

CST 8152 – Compilers – The PLATYPUS Informal Language Specification

“Everything should be made as simple as possible, but not simpler” Albert Einstein

A company named **SiR** (that’s me) offers you a contract for writing a compiler for a *Programming Language – Aforethought Tiny Yet Procedural, Unascetic and Sophisticated (PLATYPUS)*. Here is an opportunity to prove is your love for Compiling and programming in C! You cannot resist. You are looking around for something to tweet about. You ask for a language specification. You receive the following Informal Language Specification.

The PLATYPUS Informal Language Specification – Platypus White Paper

The **PLATYPUS** is really small and lacks some features like functions and complex data types, but in spite of that it is a **Turing complete** language. A **Turing complete** language must be able to describe all the computations a **Turing machine** can perform, because it is proven that a **Turing machine** can perform (may be not very efficiently) any known computation described by an algorithm on a computer.

A programming language is **Turing complete** if it provides integer variables and standard arithmetic and sequentially executes statements, which include assignment, simple selection and iteration statements.

A **PLATYPUS** program is a single file program. The format of a **PLATYPUS** program is

PLATYPUS {*optional statements*}

The *optional statements* in braces form the program body. Any number of statements (maybe no statements at all) can be included in the body of a program (that is, the body of a program could be empty).

The language must allow for comments to be included in the program text. The language supports single-line type comments only. A comment begins with **!!** (two exclamation marks) and ends with a line terminator. Comments may appear outside the program or in the body of the program.

The language must be able to handle (program) computational tasks involving integer and rational numbers. That is, the language can only handle whole (integer) numbers and numbers with a fractional part, both in constants and in variables. The internal memory size of an integer number must be 2 bytes. Rational numbers must be represented internally as floating-point numbers. The internal memory size for the floating-point numbers is 4 bytes. Number type variables can hold positive, zero, or negative value numbers, that is they are always signed. Number type literals (constants) can represent non-negative values only, that is, zero or positive values.

The language must be able process strings both as variables and as constants. String concatenation operation must be allowed for both string variables and string constants.

The names of the variables (*variable identifiers*) must begin with an ASCII letter and are composed of ASCII letters and digits. The last symbol may be an *at sign* (**@**). A *variable identifier* (**VID**) can be of any length but only the first 8 characters (including the **@** sign if present) are significant. There are two types of variable identifiers; arithmetic (**AVID**) and string (**SVID**). *Variable identifiers* are case sensitive.

An *integer literal* (**IL**) or integer constant is a non-empty string of ASCII digits. Only *decimal integer literals* (integer numbers written in base 10) are supported.

A *decimal* constant representation is any finite sequence (string) of ASCII digits that, if it has a length more than one digit, should contain only zeros or it should not have leading zeros.

Examples of legal decimal constant representations are 0, 00, 000, 109, 17100, 32768 (note: this is a legal decimal number representation but it is illegal as a value for integer literals – it is out of range). Examples of illegal decimal constant representation are 01, 001, 0097.

A *floating-point literal* (**FPL**) consists of two non-empty strings of ASCII digits separated with a period (.). The first string and the dot must be always present. The second string is optional. The first string should always be a legal decimal literal. Examples of legal floating-point constant representations are 0.0, 00.0, 00., 2.01, 2.. Examples of illegal floating-point constant are; 01.0, 001.00, .0, .01, .8,

A *string literal* (**SL**) is a finite sequence of ASCII characters enclosed in quotation marks (“”). The string literal cannot contain “ as part of the string literal. Empty string is allowed, that is, “” is a legal string literal.

There is no explicit data type declaration in the language. By default, the type of a variable is defined by the first and the last symbol of the variable identifier. If a variable name begins with one of the letters **i**, **d**, **n** or **m** the variable is of type integer and it is automatically initialized to 0. If the variable name ends with an *at sign* (@), the variable is considered of type string and the type of the variable is not determined by the first letter. The string variables are initialized with an empty string. In all other cases the variables are considered floating-point and they are automatically initialized to 0.0.

The language should allow mixed type arithmetic expressions and arithmetic assignments. Mixed arithmetic expressions are always evaluated as floating point. Automatic conversions (promotions and demotions) must take place in such cases.

The language must provide the following statement types:

Assignment Statement:

Assignment Statement is an *Assignment Expression* terminated with an *End-Of-Statement* (EOS) symbol (semicolon ;).

The *Assignment Expression* has one of the following two forms:

AVID = *Arithmetic Expression*

SVID = *String Expression*

where *AVID* is Arithmetic *VID* and *SVID* is String *VID*.

Arithmetic Expression is an infix expression constructed from arithmetic variable identifiers, constants, and arithmetic operators. The following operators are defined on arithmetic variables and constants:

- + addition
- subtraction
- * multiplication
- / division

The operators follow the standard associativity and order of precedence of the arithmetic operations. Parentheses () are also allowed. '+' and '-' may be used as unary sign operators only in single-variable, single-constant, or single () expressions, that is, -a+5.0, --a, and a+-b are illegal expressions; -a, +5.0, -5, -07, and -(a-5.0) are legal expressions. The default sign (no sign) of the arithmetic literals and variables is positive (+). The *Arithmetic Expression* evaluates to a numeric value (integer or real number).

Examples:

```
sum = 0.0;
first = -1.0;
second = -(1.2 + 3.0);
third = 7; fourth = 04;
sum = first + second - (third +fourth);
```

String Expression is an infix expression constructed from string variable identifiers, string constants, and string operators. Only one string manipulation operator is defined on string variables and constants:

<< string concatenation or catenation (or append)

The << is a binary operator and the order of evaluation (associativity) of the concatenation operator is from left to right. Parentheses () are not allowed. The *String Expression* evaluates to a string.

Examples:

```
light@ = "sun ";
day@ = "Let the " << light@ << "shines!"; !! day@ will contain "Let the sun shines!"
```

Selection Statement:

```
IF pre-condition (Conditional Expression)  
THEN  
{  
    statements (optional - may contain no statements at all)  
}  
ELSE  
{  
    statements (optional - may contain no statements at all)  
};
```

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.

Iteration Statement:

```
WHILE pre-condition (Conditional Expression)  
REPEAT  
{  
    statements  
};
```

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** and the *Conditional Expression* evaluates to false, the statements in the **REPEAT** are not executed at all; otherwise the statements are repeated until the evaluation of the *Conditional Expression* becomes false. If the *pre-condition* is the keyword **FALSE** and the *Conditional Expression* evaluates to true, the statements in the **REPEAT** are not executed at all; otherwise the statements are repeated until the evaluation of the *Conditional Expression* becomes true.

Conditional Expression is a left-associative infix expression constructed from relational expression(s) and the logical operators **.AND.** and **.OR..** The operator **.AND.** has higher order of precedence than **.OR..** Parentheses are not allowed. The evaluation of the conditional expression stops abruptly if the result of the evaluation can be determined without further evaluation (short-circuit evaluation).

The *Conditional Expression* evaluates to true or false. The values of true and false are implementation dependent.

Relational Expression is a binary (two operands) infix expression constructed from variable identifiers and constants of compatible types and comparison operators **==**, **<>**, **<**, **>**. The comparison operators have a higher order of precedence than the logical operators. The *Relational Expression* evaluates to true or false. The values of true and false are implementation dependent.

Example:

```
WHILE TRUE(balance>0.0 .AND. T@=="Y")  
REPEAT{balance = balance – debit;};
```

Input/Output Statements:

READ (*variable list*);

variable list is a list of one or more *VIDs* separated with commas.

The **READ** statement receives values from the standard input and assigns them to the variables in the *variable list* in the same order they are listed in the *variable list*.

Example:

```
READ(a,b,c@);
```

WRITE (*optional variable list*);

WRITE (*string literal*);

The **WRITE** statement sends the string literal or the values of the variable in the *variable list* to the standard output. **WRITE()** with no parameter outputs a line terminator.

Example:

```
WRITE("Platypus has a big smile"); WRITE(haha@); WRITE();
```

The words (lexemes) **PLATYPUS**, **IF**, **THEN**, **ELSE**, **WHILE**, **REPEAT**, **READ**, **WRITE**, **TRUE**, and **FALSE** are keywords and they can not be used as variable identifiers. **.AND.** and **.OR.** are reserved words.

The **PLATYPUS** language is a free-format language. Blanks (spaces), horizontal and vertical tabs, new lines, form feeds, and comments are ignored. If they separate tokens, they are ignored after the token on the left is recognized. 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.

Enjoy the language and do not forget that:

"When I use a word," Humpty Dumpty said, "it means just what I choose it to mean – neither more or less." Lewis Carroll, Through the Looking Glass

And also do not forget that if you understand the meaning if the following

“The area of a circle is radius times radius times that cool looking wavy-roofed Greek letter that sounds like that round pastry, you know what I mean?”

you are definitely not a computer.

Platypus fun fact:

“The platypus is usually nocturnal, coming out at night to feed, not unlike the stereotypical programmer.”

Courtesy of Matt Pucci

CST8152 – Compilers, 2 October 2019, SiR