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 ***P****rogramming* ***L****anguage* – ***A****forethought* ***T****iny* ***Y****et* ***P****rocedural,* ***U****nascetic and* ***S****ophisticated* (**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 **I**nformal **L**anguage **S**pecification.

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. Comments begin with **!<** (exclamation mark followed by less-than sign) and end at the end of the current line. They 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 size of an integer number must be 2 bytes. Rational numbers must be represented internally as floating-point numbers. Their 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 a number sing (**%**). 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 (**AVID**) and string (**SVID**). *Variable identifiers* are case sensitive.

An *integer literal* (***IL***) or integer constant is a non-empty string of ASCII digits. Both *decimal integer literals* (integer numbers written in base 10) and *octal integer literals* (integer numbers written in base 8*)* must be supported. A *decimal* constant representation is any finite sequence (string) of ASCII digits that does not start with a zero (0). Examples of legal decimal constant representation are 109, 17100, and 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 00, 0097. An *octal* constant representation is any finite sequence (string) of ASCII digits that, starts with a zero (0) followed by one ASCII digit from the

set 1 to 7, which may be followed by any combination of ASCII digits from the set 0 to 7, including no digits. Octal literal strings may not be a zero strings (00 and 000 are illegal octal constants). Examples of legal octal constant representation are 01 (decimal value 1), 07 (decimal value 7), 010 (decimal value 8), 012 (decimal value 10), 077777 (decimal value 32767), 0177777 (decimal value 65535; note: this is legal as an octal number representation but it is an illegal value for integer literals – it is out of range). Examples of illegal octal constants are 00, 001, 087, and 0797. Note that the integer value of 0 has only one representation: 0, which is not part the decimal or octal representations.

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. Leading zeros are not allowed in the first string, that is, 01.0, 002.0, 00.0, .0, .01, .8 are examples of illegal constants; 0.0, 0.00, 0.010, 0., 1., 880., 7.7 are legal floating-point constants.

A *string literal* (**SL**) is a finite sequence of ASCII characters enclosed in quotation marks (“). 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 character of the variable identifier. If a variable name begins with one of the letters ***i***, ***o***, ***d***, or ***w*** the variable is of type integer and it is automatically initialized to 0. If the variable name ends with a number 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 default data type of the arithmetic variables can be changed by assigning an arithmetic constant of a different data type to the variable (initializing assignment). The type can be changed only once in a program. The type of the initializing value determines the type of the variable. Numeric values cannot be mixed with or assigned to string variables.

Example:

```
iThink              !< iThink is by default  integer
iThink = 3.5;       !< changed to floating point

iDream%             !< variable of type string
iDream% = 7;        !< illegal, string type cannot be changed
```

Once the default data type is changed by initializing assignment, the variable data type cannot be changed throughout the program. Assigning another constant or variable to the variable in consideration will cause an implicit data type conversion to take place.

Example:

```
iFloat = 7.7;     !< the default type is changed to floating-point
iFloat = 7;       !< still floating-point; 7.0 will be assigned to iFloat
iFloat = i;       !< still floating-point (i is an integer variable initialized to zero)
iFloat = “abc“;   !< illegal
```

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 (+).

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:

- # append (string concatenation or catenation)

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

Examples:

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

***Selection Statement*:**

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

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.

Iteration Statement:

USING (*Assignment Expression 1* **,** *Conditional Expression* **,** *Assignment Expression 2* **)**
REPEAT {
statements (optional)
};

None of the expressions in the USUNG clause can be omitted.
The **USING-REPEAT** statement executes the following steps:

UR1. *Assignment Expression 1* is evaluated.
UR2. *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 *Assignment Expression 2* is evaluated and step UR2 is repeated.

Conditional Expression is an 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).

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.

Example:

```
USING(mult = 2.,balance>0.0 .AND. debit<100.0 .AND. T%=="Y",
      debit=debit*mult)
REPEAT{balance = balance – debit;};
```

Input/Output Statements:

INPUT (*variable list***);**
OUTPUT (*optional variable list***);**
OUTPUT (*string literal***);**

variable list is a list of one or more *VID*s separated with a comma. ***OUTPUT()*** prints an empty line.

Example:

```
OUTPUT("Platypus has a big smile"); OUTPUT();
```

.
The words (lexemes) **PLATYPUS, IF, THEN, ELSE, USING, REPEAT, INPUT,** and **OUTPUT** 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.

CST8152 – Compilers, 29 September 2015, SiR