ALGONQUIN COLLEGE

SVP Language LANGUAGE SPECIFICATION

General View

This document focusses on **SVP** LS (Language Specification) that is based on combination between PLATYPUS language, originally created by Prof. Svillen Ranev for Algonquin College.

Grammar, which knows how to control even kings . . .
—Molière, *Les Femmes Savantes* (1672), Act II, scene vi

Note 1

Please change this template, replacing any "**SVP LANGUAGE**" reference by your language name. Remember that this document is using the professor's language and you need to adapt the BNF to your own language. This time, you just need to define the grammar, using **white boxes**. It is not necessary to solve problems such as **LR** (Left Recursion) and **LF** (Left Factoring). You don't need to define the FIRST set (that will be used later in the implementation).

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

1. The SVP LANGUAGE Lexical Specification

1.1. 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.

<white space> → *one of* { SPACE, TAB }

1.2. Comments

SVP LANGUAGE supports only single-line and multi-line comments. Single-line comments are donated by " // ", while multiple-line comments are enclosed within " /* … /* ".

<comments> → // { sequence of ASCII chars } \n -> /* { sequence of ASCII chars } */ \n

1.3. Variable Identifiers

The following variable identifier (VID) tokens are produced by the scanner: two kinds of arithmetic tokens: **IVID_T** (integer) and **FVID_T** (float-point numbers) and one kind of string: **SVID_T**.

<variable identifier> → SVID_T

1.4. Keywords

SVP has the following keywords represented by the **KW_T token**. The type of the keyword is defined by the attribute of the token (the index of the keywordTable[]). Remember that the list of keywords in **SVP** language is given by:

fn, let, const, string, if, then, else, while, do

1.5. Integer Literals

Integer Literals SVP recognizes integer literals with the **INL_T** token, with an integer value as an attribute.

<integer_literal> → INL_T

1.6. Floating-point Literals

Floating-point Literals SVP recognizes floating-point literals with **FPL_T** token, carrying the real decimal value as an attribute.

<float_literal> → FPL_T

1.7. String Literals

. String Literals SVP recognizes string literals with the STR_T token.

<string_literal> → STR_T

1.8. Separators

Separators SVP recognizes various separators as tokens:

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

Some different tokens are produced by the scanner - **LPR_T**, **RPR_T**, **LBR_T**, **RBR_T**, **COM_T**, **EOS_T**.

1.9. Operators

<separator> → *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.

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

A single token is produced by the scanner: **SCC_OP_T**.

<string concatenation operator> → ++

A single token is produced by the scanner: **REL_OP_T**. The type of the operator is defined by the attribute of the token.

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

A single token is produced by the scanner: **LOG_OP_T**. The type of the operator is defined by the attribute of the token.

<logical operator> → *one of* {"&&" , "||", "!"}

A single token is produced by the scanner: **ASS_OP_T**.

<assignment operator> → =

2. The SVP LANGUAGE Syntactic Specification

2.1. SVP LANGUAGE Program

2.1.1. Program

SVP LANGUAGE program is composed by one special function: "**main**" (Method name) defined as follows.

<program> → fn **main()** {
<code_session>
}

Variable Lists

The optional variable list declarations are used to define several datatype declarations:

<opt_varlist_declarations> → <varlist_declarations> | ϵ

Variable Declarations

<varlist_declarations> → <varlist_declaration>
| <varlist_declarations><varlist_declaration>

- **PROBLEM DETECTED: Left recursion – SOLVING FOR YOU:**

New Grammar

<varlist_declarations> → <varlist_declaration> <varlist_declarationsPrime>
<varlist_declarationsPrime> → <varlist_declaration> <varlist_declarationsPrime> | ϵ

Each variable declaration can be done as follows:

<varlist_declaration> → <integer_varlist_declaration>
| <float_varlist_declaration>
| <string_varlist_declaration>

2.1.2. Declaration of Lists:

Declaration of Lists SVP supports variable declarations for integers, floating-point numbers, and strings.

<integer_varlist_declaration>	→	**let** <integer_variable_list>: **i16**;
<float_varlist_declaration>	→	**let** <float_variable_list> : f32;
<string_varlist_declaration>	→	**let** <string_variable_list> : **string**;

2.1.3. List of Variables:

. List of Variables Variables can be declared in lists for integers, floating-point numbers, and strings.

< variable_list>	→	< variable>
		| < variable_list>, < variable>
< variable>	→	VID_T

___

2.1.4. CODE session:

The second part (CODE) is the place we have statements:

<code_session> → <opt_statements>

Optional Statements:

<opt_statements> → <statements> | ϵ

2.1.5. Statements:

Statements in SVP include assignment statements, selection statements (if-else), iteration statements (while loop), input statements, and output statements.

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

2.2. Statement

<statement> → <assignment statement> | <selection statement> | <iteration statement>
| <input statement> | <output statement>

2.2.1. Assignment Statement

<assignment statement> → <assignment expression>

2.2.2. Assignment Expression

<assignment expression> → <integer_variable> = <arithmetic expression>
| <float_variable> = <arithmetic expression>
| <string_variable>= <string expression>

2.2.3. Selection Statement (if statement (optional else))

<selection statement> → **if** (<conditional expression>)
{ <opt_statements> }
<optional else>

2.2.4. Optional else

<optional else> → **else** { <opt_statements> } | ϵ;

2.2.5. Iteration Statement (the loop statement)

<iteration statement> → **while** (<conditional expression>)
{ <statements>};

2.2.6. Input Statement

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

Variable List:

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

Variable Identifier:

<variable identifier> → <integer_variable>
| <integer_variable>
| <string_variable>

2.2.7. Output Statement

<output statement> → **stdout** (<opt_variable list>); | **stdout** (STR_T);

Optional Variable List:

<opt_variable list> → <variable list> | ϵ

2.3. Expressions

2.3.1. Arithmetic Expression

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

Unary Arithmetic Expression:

<unary arithmetic expression> → - <primary arithmetic expression>
| + <primary arithmetic expression>

Additive 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> →
<multiplicative arithmetic expression> * <primary arithmetic expression>
| <multiplicative arithmetic expression> / <primary arithmetic expression>
| <primary arithmetic expression>

Primary Arithmetic Expression:

<primary arithmetic expression> → <integer_variable>
| <float_variable>
| FPL_T | INL_T
| (<arithmetic expression>)

2.3.2. String Expression

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

Primary String Expression:

<primary string expression> → <string_variable> | STR_T

2.3.3. Conditional Expression

<conditional expression> → <logical OR expression>

Logical OR Expression:

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

Logical AND Expression:

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

Logical NOT Expression:

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

2.3.4. Relational Expression

<relational expression> →
<relational a_expression> | <relational s_expression>

Relational Arithmetic Expression:

<relational a_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>

Relational String Expression:

<relational s_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 Arithmetic Relational Expression:

<primary a_relational expression> → <integer_variable> | <float_variable> | FPL_T | INL_T

<primary s_relational expression> → <primary string expression>

Good luck with Assignment 3.1!