SLAG
Structured Language Applied to Gaming

Language Reference Manual

COMS W4115: Programming Languages and Translators
Professor Stephen A. Edwards
Computer Science Department
Spring 2008 Columbia University

Edward Arnold-Berkovits
ea2124@columbia.edu

A. Introduction
This language reference manual outlines the Slag programming language. Regular expression notation is used to describe the language. This manual will use the following notation within the regular expressions:
( ) – the enclosed terminals and nonterminals are a group
'a' – the symbol 'a' is a terminal
a? – the symbol 'a' is optional
a* – the symbol 'a' will occur, zero, one, or more times
a+ – the symbol 'a' will occur, one or more times
a | b – a choice between the symbols 'a' and 'b'

B. Lexical Conventions
A program consists of one SLAG specification stored in a file. Programs are in the ASCII character set.

B1. Tokens
The language contains five classes of tokens: identifiers, keywords, constants, string literals, and operators. Blank (white) space, any type of tab, and comments are ignored by the parser.

B2. White Space
A semicolon denotes the end of a statement. Spaces, horizontal tabs, and carriage return characters are whitespace. Comments are ignored by the lexer. Slag occasionally requires whitespace to separate adjacent tokens that would otherwise be a single token. For example, identifiers and keywords must have separating whitespace. The newline character at the end of a file is optional.

B3. Comments
Single line comments start with // and end at a carriage return (the only line terminator available) or at the end of file. /* denotes a multi-line comment.

B4. Identifiers
An identifier is a sequence of letters or digits, the first of which much be a letter. Letters are considered any letter of the alphabet (upper and lower case) as well as '_'. Upper and lower case letters are different. It is conventional to begin with a lower-case character. Two identifiers are the same if they have the same ASCII character for every letter and digit.
Identifier : *letter (letter* | *digit)**

B4.1 Primitive Data Types
The primitive data types for identifiers are strings, numbers, and boolean values (true/false). Slag does not require type declarations.

B5. Keywords
Slag reserves the following identifiers as keywords; all keywords are lower case and use only letters.

if elseif else while
uses true false
movement attack defense damage health
mv av dv dmg
character battle attack
display exit compare
The dot operator, “.”, is reserved for use with <entity>.<value> or <structure>.<substructure>
Braces are reserved to delineate structures.

B6. String Literals

A string literal, or string constant, is a sequence of characters surrounded by double quotes, like “ ... “. The addition operator (+) concatenates strings. String literals may not contain double quotes or span multiple lines.

B7. Separators

The following characters are separators:
{ } , ( ) . ;
A period is not used at a concatenation operator, but as the separator between an entity name and a value.

B8. Constants

Constants are a sequence of digits representing an integer. If an arithmetic expression between two integers evaluates to a real, the output will be automatically cast as a real.
Number : digit+ (. digit+)?
Slag does not allow or use real or decimal numbers for programming purposes. There is internal use of decimals, but they are rounded up in all final calculations. An example would be a battlefield rule such that damage is halved at the same time that an ability declares that damage is halved. Starting with a damage value of 3 means halving to 1.5, then to .75, then rounded up to the nearest integer of 1.

B9. Type casting

Addition of a String with any primitive non-String implicitly type casts the non-String to a String. The display(*expression*) function implicitly type casts any non-String argument as a String.

B10. Scope

Static scoping rules in Slag are as follows:
1. Identifiers are valid from where they are initialized and are no longer valid after the end of the block within which they were made.
2. Identifiers accessible where a new block starts are valid inside that block.
3. A file that does not contain any {}'s is considered one block for scoping purposes.

C. Expressions

The precedence of expression operators is identical to the order in which they are documented below. Within each subsection, the operators have the same precedence. Each subsection is left associative unless otherwise specified. The order of evaluation of expressions is left to right. Expressions are the building blocks of statements and programs in the Slag Language.

C1. Primary Expressions

factor:
| (*expression*)
| *left-value*
| *number*
| *boolean*
| *string*

C2. Arithmetic Expressions

Arithmetic expressions take the primary expressions as their operands. While the productions of the expressions appear to create right associative rules, when implemented in ANTLRv2, the language becomes left-associative.

C2.1 Unary Expressions

unary-expression:
| - *unary*
| ! *unary*

A unary expression is an operation with at most one operand ('-' or '!'). The prefix notation binary operators '-' and '!' represent negative and not.

C2.2 Multiplicative Expressions

mult-expression:
| *unary* (* *unary*)*
| *unary* (/ *unary*)*

The binary operators '*' and '/' represent multiplication and division.

C2.3 Additive Expressions

add-expression:
| *mult-expression* (+ *mult-expression*)*
| *mult-expression* (- *mult-expression*)*

The binary operators '+' and '-' indicate addition and subtraction.

C2.4 Comparison Expressions

comp-expression:
| *add-expression* (> *add-expression*)*
| *add-expression* (< *add-expression*)*
| *add-expression* (>= *add-expression*)*
| *add-expression* (<= *add-expression*)*
equal-expression:
| *comp-expression* (== *comp-expression*)*
| *comp-expression* (!= *comp-expression*)*

Comparison expressions can intuitively compare primary expressions. Using the Kleene close an expression can evaluate to any combination of comparisons and mathematical expressions or to a simple terminal such as a number of string.

C3. Assignment Expressions

To assign an argument to a value, a declaration must be made as detailed below:
assignment:
| *left-value = right-value*
Primitives are assigned by value. Variables cannot be re-assigned or an error will be generated.

C3.1 Left-Value Expressions

left-value:
| *identifier* | *array* | *member*
member:
| *identifier* (. *identifier*)+
A left-value expression describes all valid syntax that may appear on the left hand side of an assignment. This is used for declaring/accessing identifiers and characteristics of identifiers.

C3.2 Right-Value Expressions

right-value:
| new *identifier*
| *expression*
A right-value expression describes all valid syntax that may appear on the right hand size of an assignment. This is used for declaring and accessing identifiers and expressions.

D. Slag Program

D1. Programs

Slag programs are a sequence of one or more statements.
program:
| *statement* +
Certain types of statements may contain blocks of code made up of more statements which will be formatted as follows:
block:
| { *statement* * }

D2. Statements

Statements are executed in sequence. Statements can fall into several groups.
statement:
| *function*
| *selection-statement*
| *iteration-statement*
| *foreach-statement*
| *looping-statement*
| *character-statement*
| *battle-statement*

| *attack-statement*
| *compare-statement*
| *display-statement*
| *exit-statement*

D2.1 Functions

There are no user defined functions.

D2.2 Selection Statements

Selection statements must be formatted as follows.

selection-statement:
| if (*expression*)
block
(elseif (*expression*)
*block)**
(else
block)?

In the if statement, the *expression* is evaluated and if it evaluates to true the *block* on the following line is executed. With the optional elseif, the second *block* is executed if the *expression* evaluates to false. That continues until the else statement, which executes if there are no matches. There is no elseif or else ambiguity because of the rule that { }'s must surround a block.

D2.3 Iteration Statements

Iteration statements iterate over a range of values.

iteration-statement:
| for (*assignment* ; *expression* ; *assignment*)
block

The arguments within the *for* statement are required for the loop to compile correctly. In the for statement, the first assignment is evaluated once, and specifies the initialization for the loop.

foreach-statement:
| foreach *identifier* in *expression*
block

D2.4 Looping Statements

while-statement:
| while *expression*
block

D2.5 Character Statements

character-statement:
| character *character-name character-common-name point-value*
core-value-list
ability-list

core-value-list:
| *(core-value : integer-list)+*

ability-list:
| *(ability-area ability-name ability-common-name, core-value +/- integer)**

Core values and abilities are only declared within the context of a *character-statement*.

```
character Spiderman "Spider-man" 89 {
    health: 8;    // For error-checking declarations that follow. Error if not equal to lists' lengths.
    mv : 8, 8, 8, 7, 7, 7, 6, 6;
    av : 10, 10, 9, 9, 8, 7, 7, 6;
    dv : 17, 17, 16, 16, 15, 15, 15, 14;
    dmg : 3, 3, 2, 2, 2, 2, 2, 2, 1;
    attack webslingers "Webslingers" av+1, dmg-1;
    attack incap “Incapacitate” av-1, dmg+1;
    defense spisense “Spider Sense” dv+2;
   };
```

D2.6 Battle Statements

battle-statement:

| battle *battle-name character-name, character-name*

D2.7 Attack Statements

attack-statement:

display-statement attack-statement

Used in the context of a display statement. Attack is a reserved keyword that contains the results of the most recent attack as a String. This includes damage done, and whether a character is knocked out.

D2.8 Compare Statements

compare-statement:

| compare (*expression* to *expression)*

Compare statements utilize the attack-to-defense comparison method. Slag uses a random dice roll of 2 d6 added together, with an equal chance of [1-6] on each die. The use of compare will cause a characters health pointer to be updated.

D2.9 Display Statements

The display statement prints data to standard output. No other form of output is supported.

display-statement:

| display (*expression*)

If the statement is not passed a string, the terminal of the expression will be cast to a String.

D2.10 Exit Statement

exit-statement:

| exit

This causes the program to exit.

E. Code Sample

```
display (“You are Spider-man. You need to save the city from the rampage of Sandman.\n”);
display (“You can attack using the keyword ‘attack’ or request character status using ‘health’.\n”);
display (“Use the form ‘attack using *ability*’ where ability is an ability the character is using.\n”);
display (“Type ‘exit’ to end the battle simulation.\n”;)
display health;
```

```
prompter = “What do you do? (attack, health, exit) >”;
display (prompter);
// characters Spiderman and Sandman already set up using character statements
entity1 = spiderman;
entity2 = sandman;

while (read input) {      // the primary program loop. Accept repeated input
        if (<input>.<first word> = “attack”) {    // Input is 1,2 or 3 words
                if (attack uses webslingers) {
                        entity1.dmg = entity1.dmg-1;
                        compare (entity1.av +1 to entity2.dv);
                } elseif (attack uses incap) {
                        compare (entity1.av to entity2.dv);
                } else {
                        compare (entity1.av to entity2.dv);
                }
                display attack;
        } elseif (<first word of input> = “health”) {
                display health;
        } elseif (<first word of input> = “exit”) {
                exit;
        } else {
                display (”You cannot perform that action. Please use attack, health or exit.”);
        }
        display health;
        currDefenseValue = entity1.dv;
        while (entity1.dv > currDefenseValue – 3) {
                display (entity2 + “ is attacking you!”);
                compare (entity2.av to entity1.dv);     // a counter-attack
                display health;
        }
        display (prompter);
}
```