1 Introduction

The Setup language defines a syntax for operating on finite sets. Setup provides a level of abstraction to the user which makes set manipulation more intuitive. We anticipate users will solve simple set-oriented problems like schedule, rudimentary databases, and probability problems.

2 Lexical Conventions

There are 5 kinds of tokens: identifiers, keywords, literals, operators and punctuation. Whitespace characters (blanks, tabs and newlines) are ignored and used only to separate tokens. At least one whitespace character is required to separate adjacent tokens.

2.1 Comments

Block comments are introduced with /* and terminated with */. Nesting of comments is not permitted. Comments are in general ignored by the compiler.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. Names are case-sensitive.
2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

- set
- int
- bool
- float
- string
- if
- tuple
- in
- then
- union
- else
- intersect
- minus
- while
- cross
- yup
- function
- returns
- nope
- return

2.4 Primitives

There are four types of primitives used in Setup:

2.4.1 Integers

An integer is a sequence of digits. All integers are lexed as a sequence of digits with an optional leading minus sign for negative integers. They are represented internally using architecture native integer representation.

2.4.2 Strings

A string is a sequence of characters enclosed in double quotes as in "string". Two adjacent strings are concatenated using the "+" sign. As in

"string" + "concat" -> "stringconcat".

2.4.3 Floats

We adopt the C Reference Manual definition of a floating point number:

A floating constant consists of an integer part, a decimal point, a fraction part, an e and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

All floating point numbers will be 64-bit double precision.
2.4.4 Boolean

A Boolean value can take either yup or nope.

3 Syntax Notation

In this manual, elements of language syntax are indicated by italic type. Literal words and characters are written in verbatim. Alternatives are listed using the ”|” character: item | item.

4 Objects

4.1 Variables

A variable token is an identifier to a stored primitive, tuple or set. A variable must begin with an alphabetic character followed by zero or more letters and digits. Variables are declared by referencing their type followed by the associated token and an optional initializer, as in int a; or int a = 3;

A variable, once declared, may be reassigned but cannot be declared again in the same scope.

int a = 3;
a=4;      //ok
int a = 5; //error

4.1.1 Initialization

Variables for primitives not explicitly initialized when declared will be initialized as follows:

\[
\begin{align*}
int & \rightarrow 0 \\
string & \rightarrow "" \\
float & \rightarrow 0.0 \\
bool & \rightarrow nope \\
set & \rightarrow {}
\end{align*}
\]

Uninitialized tuples are not permitted.
4.2 Tuples

A \textit{n-tuple} is an ordered collection of \textit{n} comma-delimited \textit{elements} enclosed in parentheses. An \textit{element} is either a primitive, set or tuple. An \textit{n-tuple} and \textit{m-tuple} are considered of the same type if the following two conditions are satisfied:

\begin{itemize}
  \item \textit{n} = \textit{m}.
  \item The type of each coordinate \textit{element} is of the same type.
\end{itemize}

For example, the following tuple elements are not the same type because the first coordinate \textit{tuples} are not of the same \textit{type}:

\begin{verbatim}
((1,"a"),2) //type: ((int,str),int)
((1,2),3) //type: ((int,int),int)
\end{verbatim}

4.3 Sets

A set is a (potentially empty) collection of comma-delimited elements. Every element of a set is unique (duplicate elements are discarded). A set must be \textit{homogeneous in type}, meaning that every element within the set has matching type. All sets are typed as \textit{set}, regardless of their contents. A set containing sets of varying types is allowed. Sets can be initialized in various ways:

4.3.1 Literal Initialization

A set may be initialized with a comma-delimited list of elements or identifiers of matching type within curly braces:

\begin{verbatim}
set A = {1, 2, 3, 4, 5, 6}; //ok
string b = "name";
set B = {"this", "works", b}; //ok
set C = {1, 2, 3, b}; //error: elements have different type
\end{verbatim}

4.3.2 Range Initialization

For sets of integer type, we allow the following range initialization using "\ldots":

\begin{verbatim}
set A = {1 ... 6}; //ok: returns {1, 2, 3, 4, 5, 6}
set B = {1 ... 6 10...12}; //ok: returns {1, 2, 3, 4, 5, 6, 10, 11, 12}
set C = {3 ... -1}; //ok: {3, 2, 1, 0, -1}
\end{verbatim}
4.3.3 Set-Builder Initialization

The following syntax is used for initializing a set through set-builder notation:

\[
\{ \text{expr} \mid \text{sourcelist} \}\]

The source list is a sequence of comma-delimited expressions of the form

\[\text{id in set}\]

Each \text{id} represents a local variable which may be used to construct new elements for a set via the expression appearing on the left hand side of the \mid symbol. The sequenced sourcelist expressions are evaluated left-to-right. The \text{in} operator is right associative, and the \text{id} is not assigned a value until the right operand has been resolved. The resulting set will include the resulting left side expression evaluated for all potential \text{id} values, with duplicates removed.

\[
\text{set } A = \{ (x,y) \mid x \text{ in } \{1,2,3\}, y \text{ in } \{"a","b","c"\} \};
\]
\[
B = \{ (1,"a"),(1,"b"),(1,"c"),
(2,"a"),(2,"b"),(2,"c"),
(3,"a"),(3,"b"),(3,"c" \} ;
\]
\[
A == B; //returns yup
\]

In addition, the left side expression may contain references to variables in enclosing scopes (local sourcelist variables shadow enclosing scopes). For example:

\[
\text{int } a = 0;
\text{int } x = 5;
\text{set } A = \{ (a,x) \mid x \text{ in } \{1,2,3\} \}; //sourcelist x shadows enclosing x = 5
\]
\[
B = \{ (0,1),(0,2),(0,3) \};
A == B; //returns yup
\]
5 Operators

5.1 Arithmetic Operations

The following arithmetic operations are provided: +, -, * for types int and float. The return type is as follows:

- \( \text{int binop int} \rightarrow \text{int} \)
- \( \text{float binop float} \rightarrow \text{float} \)
- \( \text{float binop int} \rightarrow \text{float} \)
- \( \text{int binop float} \rightarrow \text{float} \)

There are two division operators:

1. \( \div \), which accepts as arguments any two numerical values and is guaranteed to return a float.
2. \( \text{\#/div} \) accepts as arguments any two numerical values and is guaranteed to return an int truncated toward zero.

The following relational operators are provided for all primitive types: <, >, <=, >=, ==, !=. Types passed to these operators must match, except in the case of <, > when comparing int and float. String comparison is done lexicographically as per strcmp in the C standard library. The only character set supported is 8-bit ASCII.

5.2 Set Operations

The following set operations are provided:

- \text{union}
- \text{intersect}
- \text{minus}
- \text{cross}
- \text{#}

\text{union}

\text{union} is a binary operator which returns a union of two sets. Both sets must contain elements of the same type.

\text{intersect}

\text{intersect} is a binary operator which returns the intersection of two sets. Both sets must contain elements of the same type.
minus

`minus` is a binary operator which returns a set of elements which are present in the first set but not in the second set. Both sets must contain elements of the same type.

cross

cross takes as left operand a set with elements of type `a` and as right operand a set of elements with type `b` and returns an exhaustive set of tuples of type `(a, b)`, duplicates removed.

#

`#` is a unary operator returning the number of elements in a set.

6 Context-Free-Grammar

```
prog → ϵ | funcdef | funcdef prog
stmt-list → stmt | stmt stmt-list
stmt → if (expr) then {stmt-list} else {ϵ | stmt-list}
stmt → while (expr) {stmt-list}
stmt → expr;
stmt → return expr;

expr → id = expr | funcall | expr binop expr | {exprlist} | {int...int} |

expr pipe sourcelist} | int | float | string | tuple | id
expr → unop expr
exprlist → expr | expr, exprlist
sourcelist → id in set | id in set sourcelist
tuple → (exprlist)

funcdef → function id [formals-list] returns typespec {stmt-list}
funcall → id(arglist)
arglist → ϵ | exprlist
formals-list → ϵ | formals-tail
formals-tail → formal | formal, formals-tail
formal → typespec id

typespec → int | set | float | string | tuple | bool
binop → + | - | * | / | // | < | > | <= | >= | == | != | && | || | union | cross
```
7 Functions

Each program is a sequence of zero or more function definitions. Execution begins by calling the main function which returns an int. Syntax for user-defined function declarations is as follows:

```
funcdef → function id [func-param-list] returns typespec {stmt-list}
```

Function declarations may not be nested. Every function must return a value. The value type must be specified in the declaration, as shown in the following example:

```
function float_to_int[float f] returns int { return f//1; }
```

8 Scope Rules

All variables are of local function scope. That is, functions may not access variables outside of their bodies.

Each set-builder expression defines a local scope. Expressions in a given set-builder expression may access variables in enclosing scopes. SourceList variables used in set-builder notation can shadow variables that live in enclosing scopes. Function nesting is not permitted.

A simple example of set-builder scope:

```
set A1 = {1,2};
set A2 = {3,4};
set A = {A1,A2};

set B = {(x,y) | a in A, x in a, y in a-{x}};
  //B = {(1,2),(2,1),(3,4),(4,3)}
```

A richer example, using nested scopes:

```
set A1 = {1,2};
set A2 = {3,4};
set A = {A1,A2};
```
set $B = \{x \times 2 \mid a \in A, x \in \{y+1 \mid y \in a\}\}; \quad \text{//result: } B = \{4, 6, 8, 10\}$

/*Pseudocode:
foreach $a$ in $A$
    foreach $y$ in $a$
        $x = y + 1$
        $B.add(x \times 2)$
    next $y$
next $a$
*/

An example of variable shadowing:

```plaintext
int x = 2;

set $A = \{x \times x \mid x \in \{y \mid y \in \{x, x \times x\}\}\}; \quad \text{//result: } A = \{4, 16\}$
x; \quad \text{//result: } x=2
```

/*Pseudocode
$x = 2$ //outer scope $x$
temp $t = \{x, x \times x\}$ //\{2,4\}
foreach $y$ in $t$
    $x = y$; //does not touch outer scope $x$
    $A.add(x \times x)$
next $y$
*/