Essentials and Concepts

- **Statements**
- **Braces and Blocks**
- **Procedures**
- **Types**
- **Expressions**
- **Local and Global Variables**
- **Relational Operators**
- **Constraints**
- **Tests**
- **Assignment**
- **Witness Point**
- **Type Conversion**

Statements

A *statement* is a **constraint**, a **test** an **assignment**, a **procedure** call, a **block** built up from others statements using braces, one of the control statements **continue**, **break**, **return**, or a compound statement constructed with one of the other programming constructs of 2LP. A statement in 2LP either succeeds or fails.

Braces and Blocks

Statements in 2LP can be grouped into a single statement block by enclosing them in braces. The statements within a block are invoked as a unit.

Procedures

A procedure has the form

```
proc_name(<parameters>)
{
             <local declarations>
             <statements>
}
```

The main procedure must be called **2lp_main()**

Types

The 2LP language supports four data types: **int**, **double**, **string**, and **continuous**. The types `double` and `int` are the same types as in C and C++, i.e. `int` is the integer type and `double` is the floating point type. The type `string` is analogous to `char*` in C. Arrays of each of the four basic types can be declared.

Expressions.

An *affine expression* and its **type** are defined by induction. The most basic affine expressions are identifiers and constants, which have their given types. This is the base case of the induction. An affine expression is then defined as an identifier, a constant, or a *compound expression* built up from simpler affine expressions. A compound expression inherits its type from its constituents. Let's look at some examples:

Expressions and Types

Expression	**Type**
a	int or double
b	int or double
X	continuous
X + a	continuous
(X + a)	continuous
b*(X + a)	continuous

floor(b*(X+a))	double
Y - floor(b*(X + a))	continuous

More formally, compound expressions are built by means of the following constructions:

function (<affine expression>)
(<affine expression>)
<unary plus/minus> <affine expression>
<affine expression> <binary operator> <affine expression>

For compound terms formed by applying a function, the resulting affine expression has the type which is returned by the function. Putting parentheses around a term produces a new term of the same type; similarly, prefixing an expression with a unary plus or minus yields an expression of the same type.

In combining expressions by means of binary operators the type **continuous** dominates **double** and the types continuous and double both dominate **int**. There are two restrictions. In forming a compound affine expression, if the binary operator is *, then the left hand argument cannot be of type continuous; if the operator is /, neither argument can be of type continuous.

Hence, the basic rule is that an affine expression of type continuous can not be used as the left hand argument of a multiplication or as either argument to a division. The 2LP parser checks that complex expressions yield linear **constraints** by verifying that this rule is observed. So, for example, the line of code

```
b <= 1.5*(X + 1)*(Y - 12*X);
```

cannot be parsed successfully because the expression `(X + 1)` is being used as a coefficient.

Local and Global Variables.

Local and global variables of all **types** can be declared. Local variables of type **continuous** are of static storage class. This means that each time a procedure is called its local **continuous** variables denote the same variables as in previous calls. The constraints constructed with these variables remain in force after the procedure is exited successfully.

Relational Operators.

The relataona| operators are >=, =, >=, !=, <, and >. The first three are called closed relational operators; the second three are called the strict relational operators. The strict relational operators !=, >, and < can only be used in **tests** and not in **constraints**.

Constraints.

A constraint is a declarative **statement** which enforces relations on **continuous** variables. Formally stated, a constraint is an **expression** of the form

*<affine expression> <**closed relational operator**> <affine expression>*

where at least one of the affine expressions is of **type** continuous. The closed **relational operator** can be ==, <=, or >=. When a constraint succeeds, it is consistent with the feasible region defined so far and is added to the defining constraints of the region. However, if a new constraint is added and no points lying in the current feasible region satisfy the new constraint, then the new constraint fails.

Tests

A *test* compares two **expressions** which are of **type** **double** or **int**. Formally, a test is an expression of the form

*<**ûôôñëe expression**   **relational operator**> <affine expression>*

where theflaffine expressions are not of typö**continuous**. As with **constraints**, tests either succeed or fail. However, in contrast to constraints, tests will not affect the current feasible region or **witness point**. The strict operators !=, >, and < can only be used in tests and not in constraints. Expressions of **type** **string** can not be used in tests.

Assignment

The fundamental operation on variables of **types** **int** and **double** is the *assignment operation*. This takes the form

<variable identifier> = *<****affine expression****>;*

where the variable and the affine expression are both of type int or double.

The fundamental operation on variables of type **string** is also the assignment operation. This takes the form

<variable identifier> = *<variable identifier or constant>;*

where the variable identifiers and the constant are both of type string.

Unlike **constraints** and **tests**, an assignment **statement** always succeeds.

Witness Point

A corner point on a polyhedron is called a *vertex*. The simplex based **constraint** solver maintains the current *feasible region* together with a distinguished vertex; this vertex is called the *witness point*.

Type Conversion.

When a variable of **type** **continuous** is used in a call to a built-in function and the parameter type is **double** then the **witness point** value is used. If a variable or **expression** of type double is used in a built-in function or procedure or user-defined procedure where an integer value is expected, the value is truncated; if a variable or expression of type int is used where a double is expected, the value is coerced to type double.