SYNTAX OF INTEGER EXPRESSIONS

Although assertions are expressed in the language of logic, they are composed of statements concerning the values of variables in the program. Since the only objects in any assertion are variables and these values, we need a set of relations or predicates to talk about them. Assuming that the set of values is the integers, we can build a grammar consisting of:

1. variable names
2. integer values
3. arithmetic expressions involving variables and values

There are many ways we could do this, but in order to be as formal as possible, we could start with the number zero, and the two functions ‘succ’, which the next number in the sequence, and ‘pred’, which gives the previous one. This fits with an axiomatization of arithmetic that started with Peano and makes up what we know as number theory. In this theory zero is the only value that is truly primitive. The other integers are just short hand for applications of succ or pred. In fact:

1 is short hand for succ(0)
2 is short hand for succ(succ(0)) and so on.

The negatives are handled by pred:
-1 is short hand for pred(0)
-2 is short hand for pred(pred(0)) and so on

This start, together with recursion is actually sufficient to describe all of arithmetic. We will not be so deep, but instead will appeal to ideas we all know from arithmetic, and use the following abstract syntax for an integer expression $E_I$:

$$
E_I ::= n \mid id \mid o_u(E_I) \mid (E_I o_b E_I)
$$

$$
o_u ::= - \mid \text{abs}
$$

$$
o_b ::= + \mid - \mid * \mid ** \mid / \mid \text{mod}
$$

$$
n ::= \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
$$

$$
id ::= x \mid y \mid z \mid \ldots
$$

Thus we can form familiar expressions like $x^*((y-1)/(z \text{ mod } 3))$. This is easy to understand except for the distinction between an identifier in an arithmetic expression, such as $x$, or $y$, and their counterparts in the program code, $x$ and $y$. Clearly we intend that for every program variable name, there is a corresponding arithmetic expression identifier. Both are ‘variables’, but in different senses. A program code variable is so called because it can be assigned to, whereas a variable in an arithmetic expression is a placeholder for any integer value. The distinction is subtle, but important, because one of our goals in formal semantics is to distinguish between syntax, as in program code variables, and semantics, as in arithmetic variables. The distinction is even more obvious when we consider the important operation of substitution.

SUBSTITUTION

Program state changes according to the assignments that change it. We will need a similar effect in the semantic side. The idea of substitution is the means to achieve this. Substitution is just about the simplest operation we can think of to achieve a change. Any expression can have its variables substituted by another expression. First, this is allowed by the recursion in the syntax, and can be done using purely textual means, i.e. the operation is purely syntactic in nature. We will use the following syntax, which is very similar to what will be used in the lambda calculus as well.

$$
E_i[E'_i \setminus id] \ means \ substitute \ expression \ E'_i \ for \ id \ in \ expression \ E_i
$$

For example $(x + y*3)[(z - 1) \setminus y]$ gives the expression $(x + (z - 1)*3)$ and $(z/5)[(x - 3) \setminus z]$ gives $((x - 3)/5)$
Note that we may have to surround the substituted expression with parentheses in order to preserve the intended meaning. For instance the first example should not be \((x + z - 1)*3\) which is at best ambiguous, and at worst, given the normal precedence of operators, just plain wrong. We may also have use for an extension of this idea:

\[ E_i [E'_i \setminus E'_i] \text{ means substitute expression } E'_i \text{ for expression } E'_i \text{ in expression } E_i \]

For example, \(((x * 2) + 1/(x * 2))[(y - 1) \setminus (x * 2)]\) gives \(((y - 1) + 1/(y - 1))\)

It is no coincidence that the syntax for substitution is very close to the operation of updating a function. In fact, we shall essentially be using expression substitution to handle state change, just as we used function update in operational semantics to do the same thing.

**ASSERTIONS**

The addition of the standard operators of logic gets us where we want to go – a language for expressing constraints on program state. The syntax for assertions is also given in by an abstract syntax:

\[
A ::= (E_i \ r \ E'_i) | \neg A | (A \ c_b A') | \forall X. A | \exists X. A
\]

\[
r ::= = | \ne | > | < | \ge | \le
\]

\[
c_b ::= \land | \lor | \Rightarrow | \Leftrightarrow
\]

Now we can express constraints like:

\[
x > (y + 1) \land x < 0 \text{ and } y / 2 + z \neq 3
\]

The quantifiers need some explanation, because they introduce another kind of variable, the logical variable. To distinguish logical variables from arithmetic variables, we will use upper case letters. So a possible quantified assertion is:

\[
\forall X. X > 1 \Rightarrow X > 0
\]

This is clearly true but how do we know for sure? The answer lies in the last but one link in the chain (the last is the set of axioms). In order to handle different assertions in a program, we will need a proof technique.

**PROOFS OF ASSERTIONS INVOLVING INTEGER EXPRESSIONS**

The universal assertion above is obviously true, but we don’t have much to go on in order to prove it. In fact, in addition to the axiomatization of the constant numbers, there is also an axiomatization of the arithmetic operators. Axioms like:

\[
\forall X. X*0 = 0 \text{ and } \forall X. X*1 = X
\]

really define what we mean by multiplication. There are similar axioms for the other operators, and Gumb’s book (see the references) contains a full accounting of them. We will not go this deep into arithmetic, but instead rely on our knowledge of algebraic manipulations in order to justify assertions like:

\[
x > 4 \Rightarrow x > 2 \text{ and } x > y \land y = 0 \Rightarrow x > 0
\]

The first comes from the fact \(4 > 2\), so anything greater than 4 must also be greater than 2. The second is justified by algebraic substitution: if \(x > y \) and \(y = 0\), then \(x > 0\). So without long laborious proofs, we can still achieve our goal of proving programs correct.

**NATURAL DEDUCTION: RULES OF INFERENCE**

Although we will use very little of it, the proof technique we shall use is called natural deduction, it is based on a set of inference rules that are mostly obvious since they stem from ‘natural’ thinking. Even so, these rules are purely
syntactic; and their semantics have been shown to be adequate for the task we have here. As an example, let us prove something fairly simple to illustrate the axioms of arithmetic and natural deduction.

We will prove the formula $\forall X.0 + X = X$. We will need four axioms:

1. $\forall X.X + 0 = X$, which is part of the definition of +

   (note that we cannot assume that $\forall X, Y.X + Y = Y + X$ or commutativity of +, which is not an axiom, so the formula to be proved differs from the axiom 1.1.

2. $\forall X, Y.X + (Y + 1) = (X + Y) + 1$ (associativity of +)

3. $\forall E, E', E^* \cdot E' = E^* \Rightarrow E\left[E^* \setminus E'\right] \Leftrightarrow E\left[E' \setminus E^*\right]$ (the identity axiom, i.e. we can substitute different expressions in the same expression, as long as they are equal)

4. $\forall E, E', E^*, E = E' \land E = E^* \Rightarrow E = E^*$ (transitivity of identity, or two expressions that are equal to a third are themselves equal)

The proof is by induction. Let us denote $0 + x = x$ by $B(x)$. This is the induction base expression.

We get there by removing the quantifier on the goal to be proved, i.e. if $0 + X = X$ for all $X$, then it is true for a particular variable $x$. At the end we hope to add the quantifiers back for the general result.

First, we must prove $B(0)$, i.e. $0 + 0 = 0$. This is simply an instance of the axiom 1.1 (remove the quantifiers and substitute 0 for $x$, so the base case is true. Now we must prove that $B(x) \Rightarrow B(x+1)$, the induction step:

1. assume $B(x)$ is true, i.e $0 + x = x$

2. $x + (y + 1) = (x + y) + 1$ (remove quantifiers from axiom 1.2. $x$ and $y$ could be the same

3. $0 + (x + 1) = (0 + x) + 1$ (an instance of 2, obtained using axiom 1.3 by substituting 0 for $x$ on both sides of the equality, and then substituting $x$ for $y$)

4. $0 + (x + 1) = x + 1$ (from $B(x)$ and substituting equals for equals)

5. this is $B(x + 1)$, so $B(x) \Rightarrow B(x+1)$ (conditional proof, from 1 and 6)

6. $\forall X. B(X) \Rightarrow B(X + 1)$ (since $x$ was not chosen specially)

And, by the induction principle from 8, since we already have proved $B(0)$, we can say:

7. $\forall X. B(X)$

Q.E.D.

So, give or take some quantifiers, we have proved that $0 + x = x$ based on the axiom $x + 0 = x$, and a bunch of other axioms and rules of inference. It may be hard to get the full picture here. The bottom line is that proofs of arithmetic assertions are hard. We will avoid them and appeal to easy rules of algebra to help us out. Does that mean that the proofs we will present are not really proofs? If we are very strict, the answer is yes, they are not proofs. However, if the result we are trying to incorporate is ‘obvious’ we can skip the detail, knowing that we could do it if needed. Unfortunately, sometimes the results are not quite so obvious. In these cases, we must resort to mechanical proof techniques which are incorporated into computer programs. These cannot really help us with proofs, but can certainly check proofs if we can express them, and help us in other ways as well. As we shall see next, the Hoare axioms are much easier to apply.