Language INFIX

In all of our languages so far, the following primitive operations – addition (+), subtraction (−), multiplication (⋆), and division (/) – have grammar rules that apply these primitives in *prefix* form, where the operator occurs before the operands. However, most programming languages use *infix* mathematical notation for these operations, so that instead of writing (as we would in `V6`, for example)

```
    +(x, *(4,y))
```

one would write

```
    x+4*y
```

It turns out that grammar rules that support infix notation are slightly more complicated than the prefix notation we have been using, but not enormously more so. We proceed to illustrate this in our language `INFIX`.

Language INFIX

A naive attempt to define grammar rules that support infix operations might be to replace our `PrimappExp` grammar rule with something like this:

```
<exp>:PrimappExp  ::= <exp>arg1 <prim> <exp>arg2
```

Unfortunately, this won't even pass the PLCC grammar rules checker, since the grammar rule is left recursive: the rule has the nonterminal `<exp>` on its LHS, and the same nonterminal `<exp>` appears on the left of its RHS. Left recursive grammar rules are not allowed in LL1 grammars, and PLCC expects only LL1 grammars.

Even if we were to ignore the left recursive issue, there's a basic semantic problem with infix expressions called *associativity*. Specifically, how do we deal with an expression like this?

```
1-2+3
```

Should this be interpreted with `arg1` being `1` and `arg2` being the expression `2+3` (with the `<prim>` being `SUBOP`), or should `arg1` be the expression `1-2` with `arg2` being `3` (with the `<prim>` being `ADDOP`)? In other words, if we were to fully parenthesize this expression, should it be '`1-(2+3)`' or '`(1-2)+3`'? Mathematically, these two interpretations are not the same, but both interpretations would satisfy our grammar rule, and PLCC would not know which interpretation to choose.

Language INFIX (continued)

A related problem is called *precedence*, illustrated by the expression

```
1+2*3
```

If the `plcc` tool chose left associativity (which is what it might have done, correctly, in the previous example), this would be interpreted as `arg1` being `1+2` with `arg2` being `3`, but then the result would be interpreted as 9, whereas the correct (mathematical) interpretation would be 7. The problem is that in infix notation, multiplication has a higher precedence than addition.

We correct this by introducing grammar rules that make it easy to implement semantics for associativity and precedence.

Language INFIX (continued)

Here is a set of grammar rules for arithmetic expressions that does the trick. The `INFIX` language is based on the V4 language with procedures. For the sake of simplicity, we only give grammar rules for the primitive operations, avoiding `let`-like expressions, `proc` definitions, and `proc` applications for now. A full grammar and skeleton semantics are provided in the `INFIX` directory. Note that the `LitExp` and `VarExp` rules have been replaced by `LitFactor` and `VarFactor`.

```
<exp>                    ::= <term> <terms>
<terms>                  **= <prim0> <term>
<term>                   ::= <factor> <factors>
<factors>                **= <prim1> <factor>
<factor>:LitFactor       ::= <LIT>
<factor>:VarFactor       ::= <VAR>
<factor>:ParenFactor     ::= LPAREN <exp> RPAREN
<factor>:Prim2Factor     ::= <prim2> <factor>
<prim0>:AddPrim          ::= ADDOP
<prim0>:SubPrim          ::= SUBOP
<prim1>:MulPrim          ::= MULOP
<prim1>:DivPrim          ::= DIVOP
<prim2>:UminusPrim       ::= SUBOP
```

Language INFIX (continued)

Here is a parse trace of the arithmetic expression ’1-2+3’:

```
<exp>
| <term>
| | <factor>:LitFactor
| | | LIT "1"
| | <factors>
| <terms>
| | <prim0>:SubPrim0
| | | SUBOP "-"
| | <term>
| | | <factor>:LitFactor
| | | | LIT "2"
| | | <factors>
| | <prim0>:AddPrim0
| | | ADDOP "+"
| | <term>
| | | <factor>:LitFactor
| | | | LIT "3"
| | | <factors>
```

Language INFIX (continued)

One final problem with using infix notation for arithmetic expressions is that there is nothing specific to mark the end of the expression. With prefix notation, the end of a primitive application is always a right parenthesis, but with infix notation, there is nothing similar. In most cases, it's easy to identify the end of an expression. Consider, for example, the following:

```
   if sub1(x) then x+3 else x+4
```

The end of the expression `sub1(x)` is marked by the token `then`, and the end of the expression `x+3` is marked by the token `else`, since `then` and `else` cannot appear after a `term` or a `factor`. But the final `x+4` might have additional terms or factors that do not appear on the same line. To fix this, we add an `endif` at the end of `if` expressions.

We use the special token `;` to mark the the end of a program. Here's an example of a complete program in the language `INFIX`:

```
   if sub1(x) then x+3 else x+4 endif ;
```

Language INFIX (continued)

We borrow the syntax of `let` expressions in language `V4` to create a new syntactic category called a `block`, with the following grammar rules:

```
<factor>:BlockFactor ::= <block>
<block>              ::= LBRACE <blockDecls> <exp> RBRACE
<blockDecls>         **= DEF <VAR> EQUALS <exp> SEMI
```

As shown in the `INFIX` code file, the evaluation semantics of a `<blockDecls>` rule is to create bindings of the LHS variables (`<VAR>`) to the values of their RHS values (`<exp>`), adding the bindings one-by-one as in top-level defines. Here is an example that evaluates to 8, once full evaluation semantics for the language has been implemented.

```
{
  def x=3;
  def y=5;
  x+y
}
```

We use semicolons to terminate each of the block's variable definitions (using `def`). As shown on the next slide, `block` is also used to define the body of a procedure.

The evaluation semantics of a `block` is to add the `def` bindings to a new environment and to evaluate the body of the block in this extended environment. This is similar to what a `let` expression does in language `V4`.

Language INFIX (continued)

Unlike `let` expressions, the environments created by `def` variable definitions in a block are implemented as in top-level defines. This makes it possible for the RHS of a `def` to refer to variables defined in the same block.

A `proc` definition in `INIFX` is similar to `V4`, except that the body of the procedure is a block instead of an `exp`, meaning that it can have its own "local" variables using `def`s. Here's an example:

```
proc(x) {def y=3; def z=add1(y); x+y+z}
```

A procedure application in `INIFX` is expressed in exactly the same way as in `V4`. The following program evaluates to 120:

```
{
  def f =
    proc(x) {if x then x*.f(sub1(x)) else one endif};
  def one = 1;
  .f(5)
} ;
```

Observe that `f` can refer to itself recursively, and even refer to the variable `one` before it is defined, since the `def` semantics of a `block` behave as in top-level `defines`.

Language ARRAY

The ARRAY language extends the OBJ language by adding support for arrays. This language also defines the while primitive. An array of a given size is created using the array operator followed by the size of the array in square brackets. Here's an example:

```
   define a = array[10]
```

When an array is created, its elements are initialized to nil. Array elements are references (in the sense of a ValRef) so they can appear on the LHS of set expressions, and they can refer to any OBJ value, including other arrays. In this way, a two-dimensional array can be constructed as an array of (one-dimensional) arrays.

Array indices are integers that range from zero to the array size minus one. For an array a and index i, the expression \a[i] refers to the value of the array at the given index. If \a[i] itself refers to an array, the value at its index position j is \\a[i][j].

Language ARRAY

It is possible to turn an array into a list, and vice versa. For example, here is the code of a procedure `array2list` that takes an array parameter and returns a corresponding list. The length of an array `a` is written as `len(a)`.

```
% turn an array into a list
define array2list = proc(a)
  let
    i = len(a)
    lst = []
  in
    while i do
      { set i = sub1(i)
      ; set lst = addFirst(\a[i], lst)
      }
    else
      lst
```

Language ARRAY

Here's a recursive version of `array2list`:

```
% turn an array into a list
define array2list = proc(a)
  let
    alen = len(a)
  in
    letrec
      loop = proc(i)
        if <?(i, alen)
        then addFirst(\a[i], .loop(add1(i)))
        else []
    in
      .loop(0)
```