Language INFIX

In all of our languages so far, the following primitive operations
subtraction (−), multiplication (⋆), and division (/) – have gramma
ply these primitives in *prefix* form, where the operator occurs befor
However, most programming languages use *infix* mathematical not
operations, so that instead of writing (as we would in V6, for examp

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

one would write

```
x+4*y
```

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

Language INFIX

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

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

Unfortunately, this won't pass the PLCC grammar rules checker, sinc rule is left recursive: the rule has the nonterminal `<exp>` on its LH nonterminal `<exp>` appears on the left of its RHS. Left recursive are not allowed in LL(1) grammars, and PLCC expects only LL(1) g this up!).

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

```
1-2+3
```

Should this be interpreted with `arg1` being `1` and `arg2` being the e (with the `<prim>` being `SUBOP`), or should `arg1` be the expres `arg2` being `3` (with the `<prim>` being `ADDOP`)? In other words, fully parenthesize this expression, should it be '`1-(2+3)`' or '`(1-2`' matically, these two interpretations are not the same, but both interpr our grammar rule. Which interpretation should we choose?

Language INFIX (continued)

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

```
1+2*3
```

If we were to chose left associativity (which is what it might have c in the previous example), this would be interpreted as `arg1` being `1` being `3`, but then the result would be interpreted as 9, whereas the ematical) interpretation is 7. The problem is that in mathematifcal multiplication has a higher precedence than addition. (Note that vi gramming languages that use infix notation use the mathematical in these kinds of expressions.)

We solve these problems in PLCC by introducing grammar rules tha dle the mathematical interpretation of expressions – including as precedence – and that make it easy to implement semantics. Our also permit using parentheses to treat a group of terms as a single se that

```
(1+2)*3
```

evaluates to 9.

Language INFIX (continued)

Here is a set of grammar rules for arithmetic expressions that does
INFIX language is based on the SET language, with call-by-value
variable assignment. We start with grammar rules for expressions
primitive arithmetic operations of addition, subtraction, multiplica
sion. We will discuss additional INFIX grammar rules later. (A ful
skeleton semantics appear in the INFIX directory.) In this simpli
INFIX grammar rules for an expression <exp>, we show the <f
terminal as representing a numeric literal. Later we will introduce
the <factor> nonterminal represents variables, if expressions, p
nitions, and such.

```
<exp>                ::= <term> <terms>
<terms>              **= <prim0> <term>
<term>               ::= <factor> <factors>
<factors>            **= <prim1> <factor>
<factor>:LitFactor   ::= LIT
<prim0>:AddPrim      ::= ADDOP
<prim0>:SubPrim      ::= SUBOP
<prim1>:MulPrim      ::= MULOP
<prim1>:DivPrim      ::= DIVOP
```

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)

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>
| | | | | <prim1>:MulPrim1
| | | | | | MULOP "*"
| | | | | <factor>:LitFactor
| | | | | | LIT "3"
```

Notice how the second occurrence of `<factors>` appears more d
the parse trace, suggesting that the multiplication operation `MULOP`
out before performing the subtraction operation `SUBOP`. This is con
mathematical convention that multiplication and division have hig
than additional and subtraction.

Language INFIX (continued)

An additional problem with parsing infix arithmetic expressions is th
ing specific to mark the end of the expression. With prefix notatio
primitive application is always a right parenthesis, but with infix
is nothing similar. In most cases, it's easy to identify the end of
Consider, for example, the following sort of infix expression that i
expression in the `SET` language, except that it uses infix arithmetic

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

The end of the expression `sub1(x)` is marked by the token `then`
the expression `x+3` is marked by the token `else`, since `then` an
appear after a `term` or a `factor` in an arithmetic expression. Bu
might have additional terms or factors that do not appear on the sa
this, we change the syntax for `if` expressions by adding an `endif`
resulting `INFIX` expression would look like this:

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

Similarly, we use the special token '`;`' to mark the the end of a prog
example of a complete program in the language `INFIX`

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

that takes into account these observations.

Language INFIX (continued)

Slide 6.4 is a simplified version of the actual grammar for Langua example, we want to include a "unary minus" primitive that acts like itive in Assignment A3 but that uses the traditional unary minus pref that has precedence higher than addition/subtraction and multiplic Here are grammar rules that support this, where SUBOP is the token

```
<factor>:Prim2Factor  ::= <prim2> <factor>
<prim2>:NegPrim2      ::= SUBOP
```

An expression like

```
-3+5;
```

evaluates to 2.

In the full INFIX language, a <factor> BNF rule also has a l which defines rules for if expressions, procedure definitions, pr cations, blocks, variables, and literals (which we showed in simp a <factor> on Slide 6.4). We call these *atoms* because they ar the precedence hierarchy. The <atom> nonterminal in our INFIX serves to define their syntax.

Language INFIX (continued)

In keeping with using mathematical notation for arithmetic expressio expressing function calls using traditional notation *not* using a DO that instead of writing

```
.f(x,y,z)
```

we write

```
f(x,y,z)
```

This change will also affect the syntax of expressions.

Here are the changes to the grammar rules relating to the <atom> n how it connects to the <factor> nonterminal:

```
<factor>:AtomFactor ::= <atom> <fncalls>
<atom>:PrimappAtom  ::= <prim> LPAREN <rands> RPAREN
<atom>:IfAtom       ::= IF <exp>testExp THEN <exp>trueExp ELSE
<atom>:LitAtom      ::= <LIT>
<atom>:BlockAtom    ::= <block>
<atom>:VarAtom      ::= <VAR> <assign>
<atom>:ParenAtom    ::= LPAREN <exp> RPAREN
<atom>:ProcAtom     ::= PROC LPAREN <formals> RPAREN <block>
<assign>:Assign0    ::=
<assign>:Assign1    ::= EQUALS <atom>
<fncalls>           **= LPAREN <rands> RPAREN
```

Language INFIX (continued)

The INFIX language also supports the assignment of values to var
that is similar to the set expression in Language SET but without
set. The right-hand-side of a such an assignment must be an ato
arbitrary expression. The <assign> grammar rule defines its sy
on Slide 6.9). Because the RHS in an Assign1 rule is an atom,
higher precedence than any infix arithmetic operations.

```
<atom>:VarAtom          ::= <VAR> <assign>
<assign>:Assign0        ::=
<assign>:Assign1        ::= EQUALS <atom>
```

For example, in the following code, the expression x*y evaluates to

```
define x=5;
define y=11;
x=(y-3);      % => 8
y=x+5;        % => assigns 8 to y, evaluates to 13
x*y;          % => 64
```

We borrow the syntax of letrec expressions in Language V5 to cre
called a <block>, with the following grammar rules:

```
<atom>:BlockAtom  ::= <block>
<block>           ::= LBRACE <blockDecls> <exp> <exp
<blockDecls>      **= DEF <VAR> EQUALS <exp> SEMI
<exps>            **= SEMI <exp>
```

Language INFIX (continued)

We use semicolons in the syntax of a block to terminate each of the b
definitions (using def). As shown in Slide 6.9, a block is also use
body of a procedure. This means that the body of a procedure can
local variables using def. Since the variable bindings in a block crea
environment using the letrec strategy, the RHS expressions in a
to variables defined in the same block, and procedures can be self-re

As noted above, procedure applications in INIFX are expressed us
mathematical notation. For example, the following program evaluat

```
{ def f = proc(x) {if x then x*f(x-1) else one endi
  def one = 1;
  f(5)
} ;
```

Observe that body of the the procedure f can refer to itself recurs
even refer to the free variable one before it is defined in the same b
semantics of a block behave as in letrec.

Also observe that function calls can be nested, as shown in this e
evaluates to 19):

```
proc(x) {proc(y) {3*x+y}}(4)(7);
```

Language INFIX (continued)

Consider the expression

```
x=y+5+x
```

Since INFIX is still expression-based – every expression has a val
expression must have a value, just as we have required in language
the expression also has an assignment operation with a side-effect th
value of x. Since the RHS of an assignment must be an <atom>,
the above expression evaluates like this:

```
(x=y)+5+x % assign y to x, then return this value p
```

If we wanted the entire RHS of the above expression to be assigned
need to re-write it as follows:

```
x=(y+5+x)
```

Language INFIX (continued)

Language INFIX defines a unary minus primitive (defined in cla
that extends the Prim2 class) that acts as a prefix operator. Its
(arithmetically) negate its Factor operand.

Language INFIX defines four non-infix primitives, all of which are i
Prim class. Three of them are taken directly from Language SET: t
semantics are unchanged from Language SET. The PospPrim i
(pos?) is drawn from Assignment A8.

All of the <atom> grammar rules and related semantics in Langua
similar to those in Language SET. As shown in the ProcAtom BN
6.9, the body of a proc definition must be a <block> (in Lang
an exp). The ProcVal class is mostly unchanged from that of I
the only exception being that its apply method does not have a
parameter. The apply method body is unchanged.

Language INFIX, as given in the Code directory, has skeleton de
eval semantics for expressions. In an assignment, you are aske
these definitions for a full implementation of Language INFIX.

Language ARRAY

The ARRAY language extends the OBJ language by adding support f
language also defines the while primitive. An array of a given size i
the array operator followed by the size of the array in square brac
example:

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

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

Array indices are integers that range from zero to the array size mir
array a and index i, the expression \a[i] refers to the value in t
given index. If \a[i] itself refers to an array, the value at its inde
\\a[i][j].

Language ARRAY

It is possible to turn an array into a list, and vice versa. For exam
code of a procedure array2list that takes an array parameter
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)
```