

Copyright (C) 2021 Timothy Fossum

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation. A copy of the license is included in the file "COPYING", entitled "GNU Free Documentation License".

Language INFIX

In all of our languages so far, the following primitive operations – addition (+), subtraction (−), multiplication (*), and division (/) – have grammar rules that allow us to 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 more complex than the prefix notation we have been using, but not enormously so. We will now proceed to illustrate this in our language INFIX.

Language INFIX

A naive attempt to define grammar rules that support infix operators would be to 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, since the rule is left recursive: the rule has the nonterminal `<exp>` on its LHS and the nonterminal `<exp>` appears on the left of its RHS. Left recursive rules are not allowed in LL(1) grammars, and PLCC expects only LL(1) grammars (so fix this up!).

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 interpret 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` and `arg2` being 3 (with the `<prim>` being `ADDOP`)? In other words, if we 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 are valid for our grammar rule. Which interpretation should we choose?

Language INFIX (continued)

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

$$1+2*3$$

If we were to choose left associativity (which is what it might have been in the previous example), this would be interpreted as `arg1` being `1+2` and `arg2` being 3, but then the result would be interpreted as 9, whereas the mathematical interpretation is 7. The problem is that in mathematical notation, multiplication has a higher precedence than addition. (Note that virtually all programming languages that use infix notation use the mathematical interpretation of these kinds of expressions.)

We solve these problems in PLCC by introducing grammar rules that handle the mathematical interpretation of expressions – including associativity and precedence – and that make it easy to implement semantics. Our grammar also permit using parentheses to treat a group of terms as a single semantic unit, 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 expression: primitive arithmetic operations of addition, subtraction, multiplication. We will discuss additional INFIX grammar rules later. (A full skeleton semantics appear in the INFIX directory.) In this simplified INFIX grammar rules for an expression `<exp>`, we show the `<factor>` nonterminal represents variables, if expressions, 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 deeply nested in the parse trace, suggesting that the multiplication operation `MULOP` is performed out before performing the subtraction operation `SUBOP`. This is consistent with the mathematical convention that multiplication and division have higher precedence than addition and subtraction.

Language INFIX (continued)

An additional problem with parsing infix arithmetic expressions is the lack of a token specific to mark the end of the expression. With prefix notation, the 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 sort of infix expression that is legal in the SET language, except that it uses infix arithmetic operators:

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

The end of the expression `sub1(x)` is marked by the token `then`, and the expression `x+3` is marked by the token `else`, since `then` and `else` can only appear after a term or a factor in an arithmetic expression. But the expression might have additional terms or factors that do not appear on the same line. To fix this, we change the syntax for `if` expressions by adding an `endif` token. The 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 program. Here is an 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 Language A3. For example, we want to include a “unary minus” primitive that acts like the primitive in Assignment A3 but that uses the traditional unary minus prefix that has precedence higher than addition/subtraction and multiplication. 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 <atom> nonterminal which defines rules for if expressions, procedure definitions, procedure calls, blocks, variables, and literals (which we showed in simplified grammar on Slide 6.4). We call these *atoms* because they are at the top of the precedence hierarchy. The <atom> nonterminal in our INFIX language serves to define their syntax.

Language INFIX (continued)

In keeping with using mathematical notation for arithmetic expressions, we express function calls using traditional notation *not* using a DCL-style prefix 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> nonterminal and 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 <exp>falseExp
<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 variables that is similar to the `set` expression in Language SET but without the `set`. The right-hand-side of a such an assignment must be an atom or an arbitrary expression. The `<assign>` grammar rule defines its syntax (see Slide 6.9). Because the RHS in an `Assign1` rule is an atom, it has a 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 64.

```
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 create 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 block definitions (using `def`). As shown in Slide 6.9, a `block` is also used as the body of a procedure. This means that the body of a procedure can create local variables using `def`. Since the variable bindings in a block create a new environment using the `letrec` strategy, the RHS expressions in a block refer to variables defined in the same block, and procedures can be self-referential.

As noted above, procedure applications in INFIX are expressed using mathematical notation. For example, the following program evaluates to 19.

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

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

Also observe that function calls can be nested, as shown in this example (which 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 value, every expression must have a value, just as we have required in languages like SET. In this expression the expression also has an assignment operation with a side-effect that changes the 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 to `x`, we would need to re-write it as follows:

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

Language INFIX (continued)

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

Language INFIX defines four non-infix primitives, all of which are in the `Prim` class. Three of them are taken directly from Language SET: the `PospPrim` (positively) is drawn from Assignment A8.

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

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

Language ARRAY

The ARRAY language extends the OBJ language by adding support for arrays. The ARRAY 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. For 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 right-hand side of expressions, and they can refer to any OBJ value, including other arrays. For example, 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 in `a` at index `i`. If `\a[i]` itself refers to an array, the value at its index `j` is written as `\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 the 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)
```