COMP 321: Design of programming languages
Parsing

---

1. Parsing in-line arithmetic expressions

We begin with parsing arithmetic expressions such as $5 + 7 * 3$. We assume the following tokens:

- $I(x)$ representing the identifier named by the string x.
- $N(x)$ representing the numeral named by the natural numeral x.
- *Neg*, *Plus*, *Minus*, *Times*, *Div* representing the corresponding operations.

We don't have parentheses yet, and we assume the usual precedence (unary negation binds tighter than multiplication and division, which bind tighter than additional and subtraction) and associativity (right for unary operations, left for binary operations).

As discussed in class, the basic parsing algorithm maintains an *expression stack* of abstract syntax trees and an *operator stack* of pending operations. Identifiers and numerals are pushed onto the expression stack. When we read an operator token, we force any pending operations of higher precedence, then push the operator token onto the operator stack.

The basic algorithm is given in figure 1. In all algorithms, *es* is a stack **of** *AST*, *op* is an *operation*, and *ops* is a stack **of** *operation*. We will generally take the type *operation* to be the type of tokens for convenience, although formally the two types should be different.

2. Adding parentheses and controlling associativity

Now we assume that we have tokens for left- and right-parentheses that are used to alter the pre-defined precedence of operations. Let's consider the possible situations in which we encouter parentheses:

- When we encounter an *LParen* token in the token stream, we push it onto the operations stack without forcing any pending operations.
- When we are forcing pending operations and encounter an *LParen* on the top of the operator stack, we must stop forcing any more pending operations.
- When we encounter a *RParen* token in the token stream, we must force all pending operations up to the top-most (i.e., most recent) *LParen* operation.

These modifications can be handled by treating parentheses as special operators, but I find it easiest just to include special cases in the basic algorithm. Another intriguing possibility is to write a recursive parser, since once you encounter an *LParen* in the token stream, you know that you need to parse an expression up to the matching *RParen*.

Right now when we are forcing operations and see an operator of equal precedence on the pending operations stack, we force that pending operation. This has the effect of saying that all (binary) operations are left-associative—i.e., when we write 3+4+5+6, we really get the AST corresponding to $((3 + 4) + 5) + 6$. If addition were right-associative, we would want the AST corresponding to $3 + (4 + (5 + 6))$. We might want right-associativity in the following cases:

- Unary operations are typically right-associative.
- The arrow constructor for types is right associative; $\rho \to \sigma \to \tau$ really means $\rho \to (\sigma \to \tau)$.

Controlling associativity is straightforward. We add an additional function or lookup table *assoc* that gives the associativity (*LEFT* or *RIGHT*) of each operator. We then modify the conditional

```
function parse (toks : seq of token) : AST
begin
    es : stack of AST
    ops : stack of operations
    foreach t in toks do
        case t of
            I(x)
            N(x):
                push t es
                break
            <operation>:
                force_ops t es ops
                push t ops
        endcase
    endf

    force_all_ops es ops
    return (peek es)
end

function force_ops op es ops : void
begin
    while ops non−empty and prec op ≤ prec
        (peek ops) do
        force_op es ops
    endw
end

function force_op es ops : void
begin
    if (hd ops) is unary then
        arg ← pop es
        push −arg es
        pop ops
    else
        right ← pop es
        left ← pop es
        C ← AST constructor corresponding to
            top of ops
        push C(left, right) es
        pop ops
    endif
end

function force_all_ops es ops : void
begin
    force_ops Finish es ops
end
```

FIGURE 1. The basic parsing algorithm. *prec* : *op* → int is a function that returns the precedence of its argument (higher means more tightly binding than lower). *Finish* is a special operator with precedence lower than all other operators.

in *force_ops* to force a pending operation of precedence equal to the current operation only if the pending operation is left-associative; i.e., we replace the conditional with

```
while ops non−empty and
      ((prec op < prec (peek ops)) or (prec op = prec (peek ops) and assoc op = LEFT)) do
    ...
endw
```

3. HANDLING END-OF-STREAM

Now that we have parentheses, we can use them to handle the end of the token stream. In the basic algorithm, when we get to the end of the token stream, we force all remaining operations. But this is essentially what happens when we encounter an *RParen* token. So we can rewrite *parse* by removing the invocation of *force_all_ops* and instead doing the following:

(1) Push an *LParen* on the operator stack.
(2) Parse tokens with the main loop of *parse*.
(3) Act as though we have encountered a *RParen* token in the token stream.

4. Anonymous functions

Anonymous functions (i.e., abstraction) can be handled as a unary operator. On input

`fn fred => (3 + fred)`

a good tokenizer will return the token stream *Lambda*("*fred*"), *LParen*, *N*(3), *Plus*, *I*("*fred*"), *RParen*, and thus we get a single token for this unary operator. The abstraction operator should be right-associative and of low precedence (so, e.g., `fn x => x + 3` parses as an application node with body an addition node, rather than as an addition node with left-hand side an abstraction node).

5. Application

Application poses difficulties, because it is an operator with no corresponding token. Here we describe one approach to handling this case.

As a first pass, assume that we require that application must always be delimited by parentheses; e.g., (`f x 3`) is legal, but `f x 3` is not. When we encounter the *RParen* token, we will have three identifiers on the expression stack and an *LParen* on the operator stack. In general, when we encounter the *Rparen* token, we will force all pending operations up to the top-most (i.e., most recent) *LParen*. We now need to know how many of those expressions form the (nested) application expression. In the example just given, it is all the expressions; but this is not the case for something like 3 + (`f x`).

The solution is as follows:

(1) When we push *LParen* onto the pending operations stack, also push a special expression *BGroup* onto the expression stack (this includes the first *LParen* that is not part of the token stream).
(2) When we encounter an *RParen* token:
    (a) Force operations to the top-most *LParen*. The resulting expression stack will have the form *eK* :: ... :: *e1* :: *BGroup* :: *es* and operation stack will have the form *LParen* :: *ops*.
    (b) Collect all expressions in the expression stack to the top-most *BGroup* into a nested application node; remove the *BGroup* and *LParen* and push the application node onto the expression stack. So the two stacks will end up as (...((*e1 e2*)*e3*)...*eK*):: *es* and *ops*.
(3) When we are forcing pending operations, we follow the same procedure as before until we get to the *LParen*. We collect the expressions just as for an *RParen* token, but we leave the *BGroup* and *LParen* on their respective stacks to end up with (...((*e1 e2*)*e3*)...*eK*):: *BGroup* :: *es* and *LParen* :: *ops*.

Notice that we are OK so far provided that our application expressions occur before any other operators; e.g., $f\,x + 3$ will parse as an addition node with left-hand side the application of f to x. But we are still in trouble if it occurs after other operators. Right now, $3 + f\,x$ will parse as the (non-sensical) AST representing $(3)(f+x)$. The problem is that when we reach the end of the token stream (really, any right-parenthesis), we will start collecting expressions into nested applications. But we need to know when to stop (e.g., when we hit any other operator). We can accomplish this by following every operator with an "implicit" left-parenthesis token and corresponding *BGroup* expression. Now when we are forcing operations, if we encounter an implicit left-parenthesis, we collect expressions into a nested application, and then continue forcing.

Putting this all together, we have the following:

(1) When we encounter an operator token:
   (a) Force pending operators.
   (b) Push the operator on the operator stack.
   (c) Push *BGroup* and *ILParen* (for "implicit" left-parenthesis) onto the expression and operator stacks.
(2) When we are forcing operations because of a non-*RParen* operator token and encounter an *LParen*:
   (a) Collect expressions to transform the stacks *eK* :: ... :: *e1* :: *BGroup* :: *es* and *LParen* :: *ops* to (...((*e1 e2*)*e3*)...*eK*):: *BGroup* :: *es* and *LParen* :: *ops*.

   If we are forcing because of an *RParen* token, delete the *BGroup* and *LParen* tokens.
(3) When we are forcing operations because of a non-*RParen* operator token and encounter an *ILParen*:
   (a) Collect expressions to transform the stacks *eK* :: ... :: *e1* :: *BGroup* :: *es* and *ILParen* :: *ops* to (...((*e1 e2*)*e3*)...*eK*):: *es* and *ops*.
   (b) Continue forcing pending operations.
(4) When we are forcing operations because of a non-*RParen* operator token and encounter any other operator, use precedence and associativity. If we are forcing because of an *RParen* token, force unconditionally.

5.1. **An alternative approach.** Michael suggested an alternative approach to the one here (if I understand his suggestion correctly). It is actually complementary to this approach: instead of identifying (via *ILParen* operators) positions just before tokens that start an expression that is the head of a nested application expression, instead identify positions just after tokens that end expressions which might be the head of a binary application expression. The idea is that we push an application operator onto the operator stack every time we read a token that ends a legal (sub)expression, such as identifiers, literals, *and RParen tokens*. We treat such an operator as any other operator, with high precedence and left-associativity. However, there are still some issues to iron out:

(1) When forcing an application operation, it might not really represent an application. For example, consider the expression $x + 3$; when we see the $+$, we will force the application operation; but instead of resulting in an actual application node being placed on the expression stack, instead we will just leave $I(x)$.
(2) We still have to indicate somehow (with something like *BGroup*) that the application does not travel any further down the expression stack. If we don't, then consider what happens when we handle the "fake" *RParen* after finishing the token sequence for the expression $x+3$. The expression stack will be x :: 3 :: *BGroup* :: [ ] and the operator stack will be *App* :: + :: *LParen*. But now when we force *App*, we'll end up with (3)(x):: *BGroup* :: [ ] on the expression stack, which is definitely not what we want.

So while I am pretty sure Michael's idea will work, I am less sure that it is really any less effort than the one presented here with implicit left parentheses.