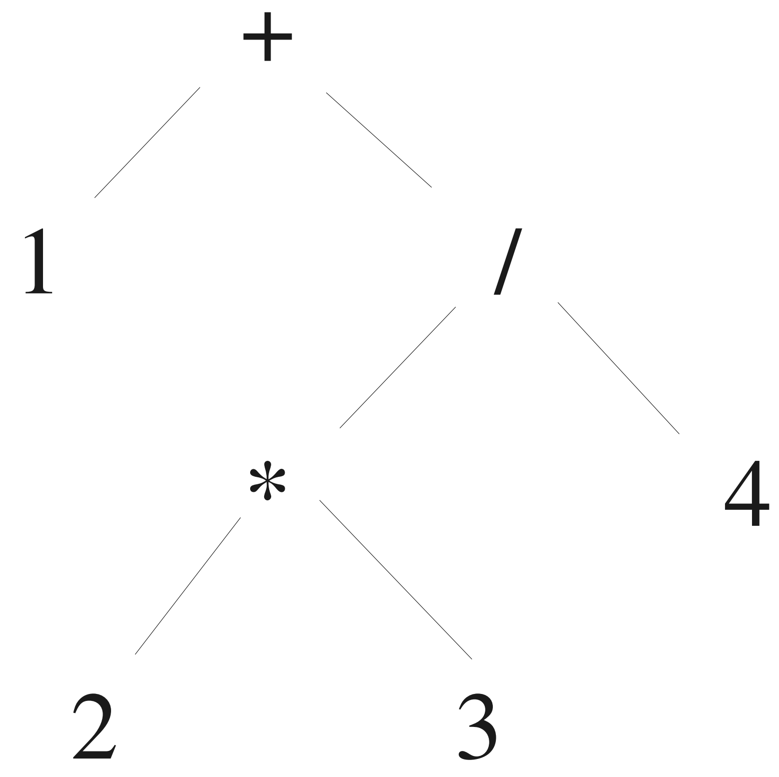
Reversible
University of
Teesside
High-level language

Teesside
University

A Compiler which Creates Tagged Parse Trees and Executes them as FORTH Programs

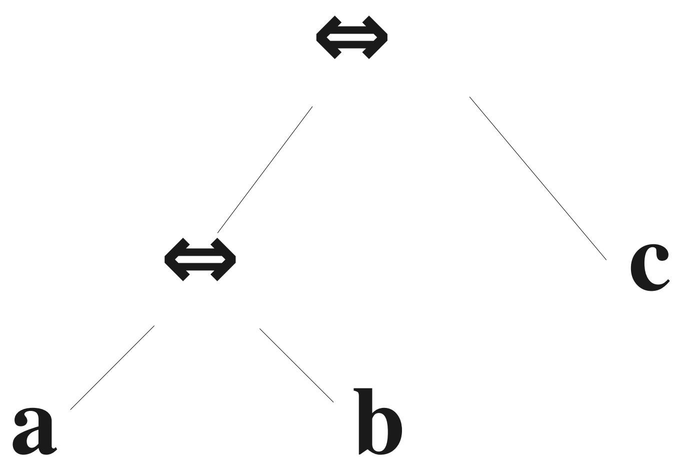
Campbell Ritchie
and
Bill Stoddart

1 + 2 * 3 / 4



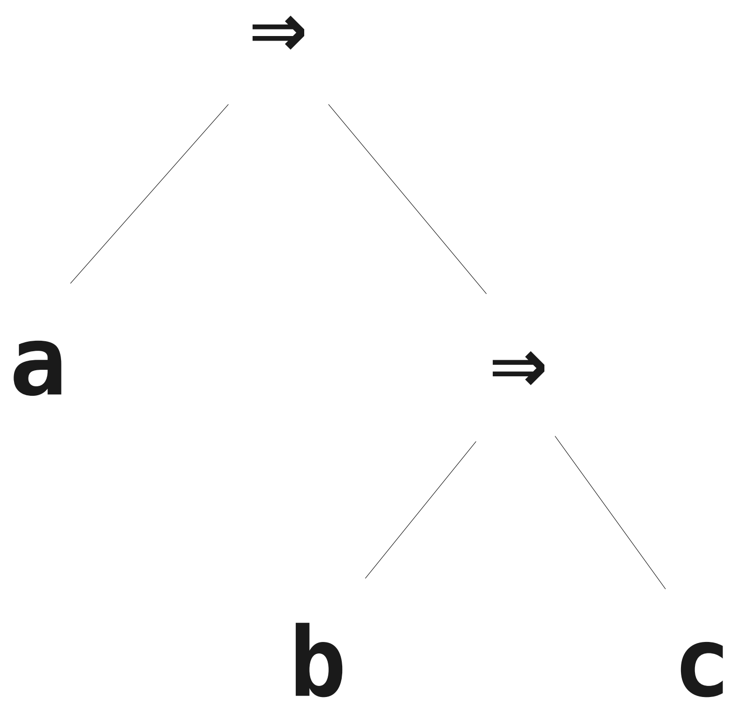
1 2 3 * 4 / +

$a \Leftrightarrow b \Leftrightarrow c$



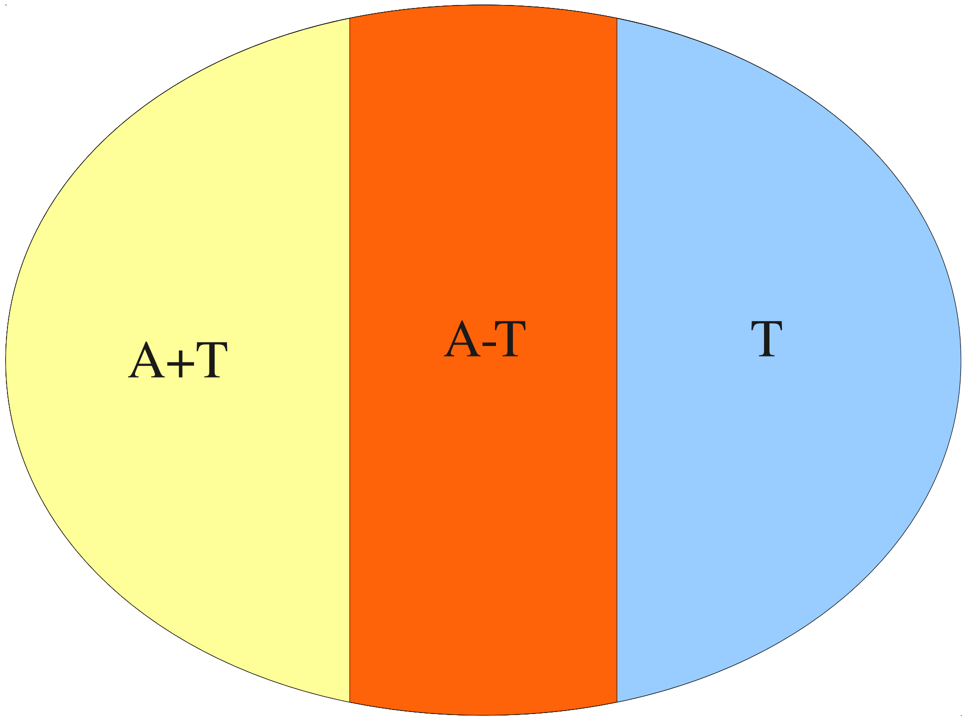
$a\ b \Leftrightarrow c \Leftrightarrow$

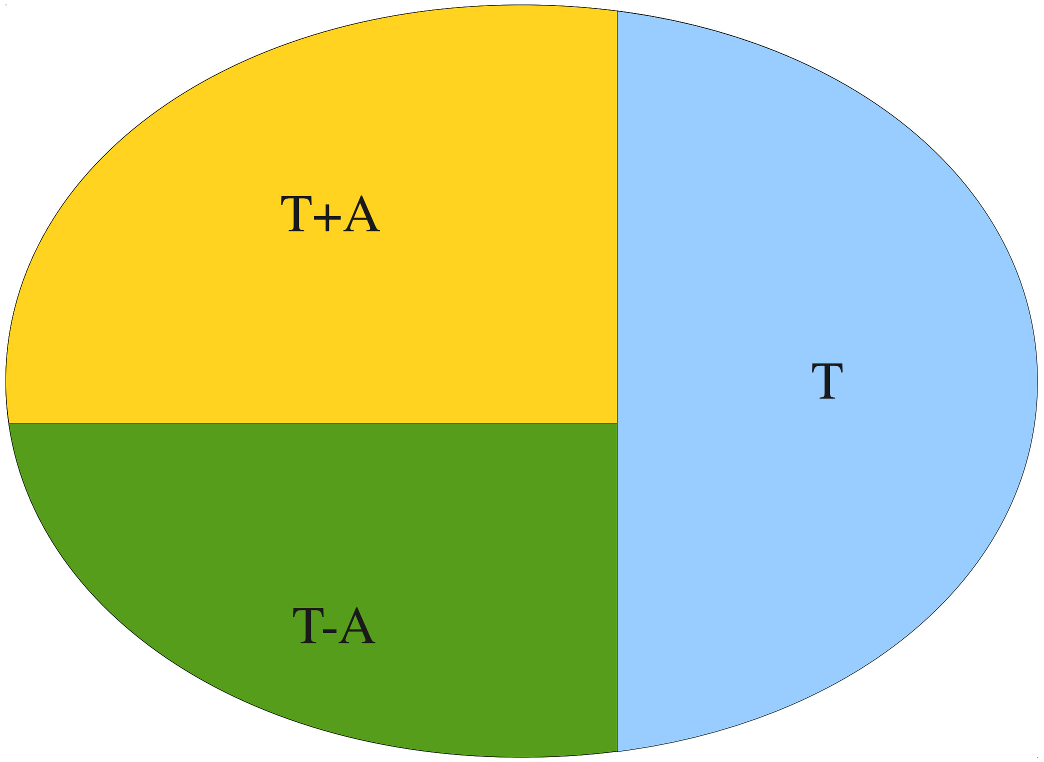
$$a \Rightarrow b \Rightarrow c$$



$$a\ b\ c \Rightarrow \Rightarrow$$

A left-associative grammar will include A = A+T, A-T, T
And a right-associative grammar would include
A = T+A, T-A, T.





```
: ⇔_ ( s1 s2 s3 s4 -- ss1 boolean )
(
    Executes the tagged tree and type testing for the equivalence operator. For
    example, "x" "boolean" "y" "boolean" --> "x y ⇔" "boolean".
)
(: l-value l-type r-value r-type :)
" ⇔" l-type r-type check-types-for-booleans
l-value sspace AZ^ r-value AZ^ "  ⇔" AZ^ l-type ;
:
```

```
: +_ ( s1 s2 s3 s4 -- ss1 ss2 )
   (: l-value l-type r-value r-type :)
   ( This can incorporate floating-point numbers as well as INTs )
   "  +" VALUE op ( note additional space, also in F+ and S>F )
   op l-type r-type check-types-for-arithmetic
   l-type float string-eq r-type float string-eq OR
   ( Either or both is float )
   IF
      l-type int string-eq
      IF  ( Add S>F as appopriate )
         l-value StoF AZ^ to l-value
      ELSE
         r-type int string-eq
         IF
            r-value StoF AZ^ to r-value
         THEN
      THEN
      "  F+" to op
      float to l-type
   THEN
   l-value sspace AZ^ r-value AZ^ op AZ^ l-type
;
```

‘1 + 2 + 3 * (4 + 5)’

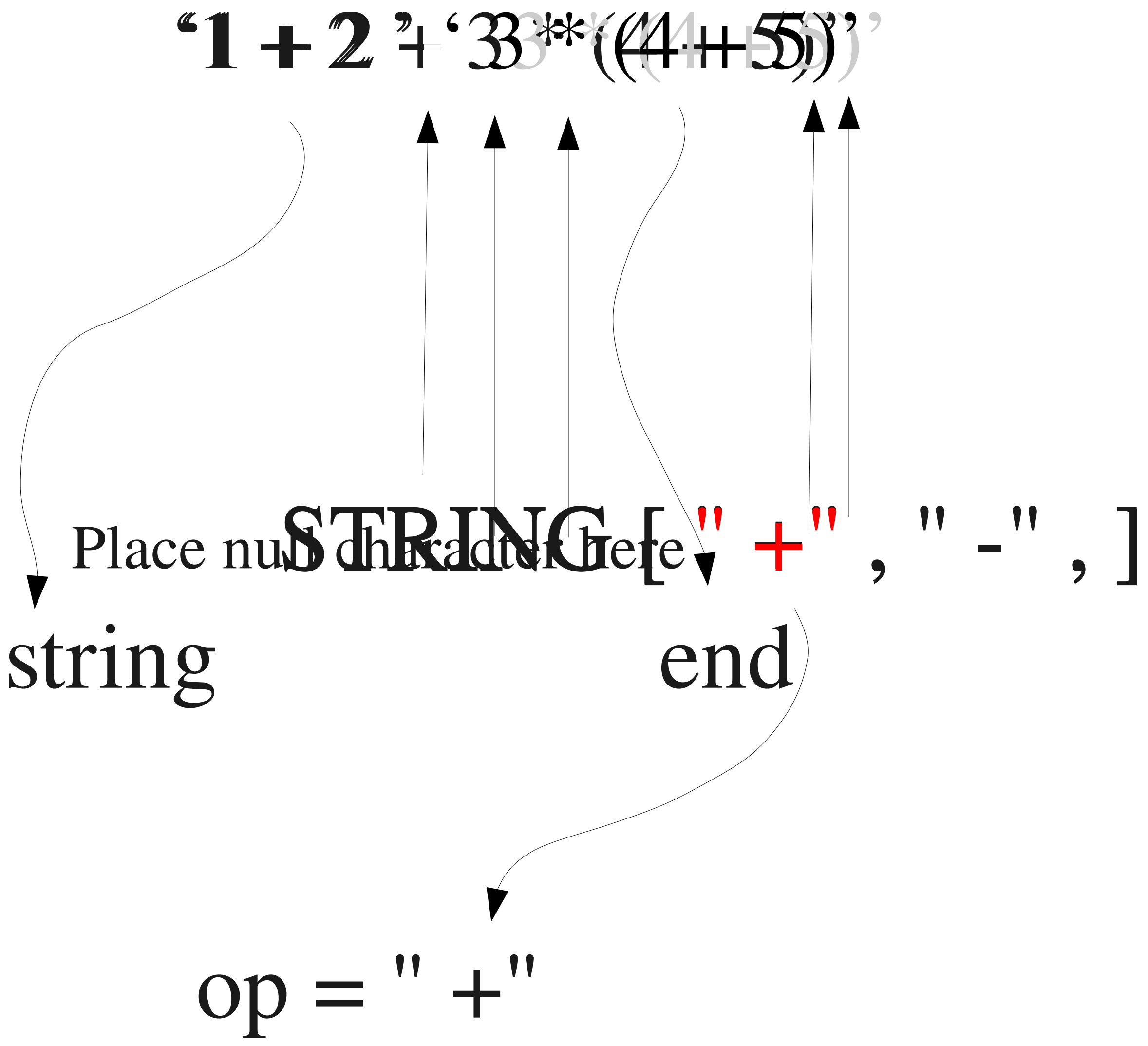
```
: Parith2 ( s -- s1 )
plusminus lsplit
DUP 0=
IF
   2DROP Pterm2 ( No operator found )
ELSE
   PUSH PUSH RECURSE ( recurse on left subexpression )
   POP Pterm2 AZ^            ( right substring to Pterm2 and catenate )
   POP AZ^ bar-line AZ^      ( add +_ or -_ )
THEN
;
```

‘ 1 + 2 + 3 * (4 + 5)’

STRING [" +" , " -" ,]

```
: lsplit ( s seq -- s1 s2 op )
( string seq already on stack ) 0 0 0 0 ( 6 values now on stack )
(: string seq end op count size :)
string endaz to end ( One of the 0s gone )
seq CARD to size    ( Second 0 gone      )

BEGIN
   end string >= op 0= AND
WHILE
   0 to count
   end bracket-avoider-for-lsplit to end ( Skip text in brackets etc )
   BEGIN
      size count > op 0= AND ( Not reached start of string, nor found op )
   WHILE
      count 1+ to count      ( Go through potential operators )
      seq count APPLY end prefix?
      IF
         seq count APPLY to op
      THEN
   REPEAT
   end 1- to end    ( Count backwards to start of string )
REPEAT
op
IF
   end 1+ to end    ( Terminate string at op )
   0 end C!
   end op myazlength + to end ( Move forward length of op )
THEN
string end op
;
```


"1 + 2 + 3 * (4 + 5)"
Place null character here
STRING [" +" , " -" ,]
string
end
op = " +"

‘ 1 + 2 * -3’

‘ 1 + 2 * 3.45e-67’

```
: Pterm2 ( s -- s1 )
(
    Where s is in the form 123 * 456 or similar, and the String is split with
    timesdivide and the lsplit operation. Assuming an operator is found, the
    left subexpression recurses, the right subexpression is passed to the next
    parser (Puminus2) and the whole lot is catenated with the operator to form a
    String in this format: " 123" " INT" " 456" " INT" *_ which is later passed
    to the *_ operation; any errors should become obvious then.
    If no operator, the right subexpression and operator are dropped as nonsense
    and the left subexpression passed to Puminus2.
)
timesdivide lsplit
DUP 0=
IF
( left subexpression to next parser: returns two strings catenated with quotes )
    2DROP Puminus2
ELSE
    PUSH PUSH RECURSE    ( recurse on left subexpression )
    POP Puminus2 AZ^     ( right subexpression to Puminus2 )
    POP AZ^ bar-line AZ^ ( add *_ or /_ )
THEN
;
```

‘4’4"5" INT" " 5" " INT" *_’

‘5’5" " INT" ’

‘*’

‘-123’

User stack

‘2-123’

‘*’

null

5

```
: Pterm2 ( s -- s1 )
timesdivide lsplit
DUP 0=
IF
   2DROP Puminus2
ELSE
   PUSH PUSH RECURSE
   POP Puminus2 AZ^
   POP AZ^ bar-line AZ^
THEN
;
```

```
" 1 + 2 + 3 * (4 + 5)" Pexpression2 .AZ
" 1" " INT" " 2" " INT" +_
" 3" " INT" " 4" " INT" " 5" " INT" +_
*_
+_
ok
" 1" " INT" " 2" " INT" +_ ok..
" 3" " INT" " 4" " INT" " 5" " INT" +_ ok......
*_ ok....
+_ ok..
.AZ INTok.
.AZ 1 2 + 3 4 5 + * +ok
1 2 + 3 4 5 + * + . 30 ok
```

RL(0)

$$\begin{tikzcd} & \text{‘}a \Rightarrow b\text{’} \arrow[dl] \arrow[dr] & \\ \text{Next Parser} & & \text{RECURSE} \end{tikzcd}$$

LL(0)

Drawbacks

Each scan along a string has a duration depending on the string’s length.

The number of scans depends on the string’s length

This technique runs in $O(n^2)$ time.

Drawbacks

This technique is difficult to use if there are any overloaded operators, and lookahead is awkward.

Drawbacks

If a low-precedence operator is “part of” a high-precedence operator, it is difficult to distinguish the two

‘1 + 2 - ++*i*’

Drawbacks

Writing such a parser is rather repetitive; it may however be easy to automate the process

Advantages

This is a simple procedure, allowing one to see the parsing process.

It is particularly useful as a teaching tool.

Advantages

This is a simple procedure, allowing one to see the parsing process.
One can us the two-stage process to parse a program in stages.

Advantages

The process is modular. It is quite easy to “insert” a level of precedences into the grammar.

FORTH interest

The entire compiler is written in FORTH.

Since FORTH accepts postfix syntax naturally, one can parse a program into FORTH and not need to provide a compiler back-end. Only a front-end is necessary

Future Work

Enhance some parsers to take specific types, eg

- Pair ($\mapsto$) may need to take types of operand.
- Some equality and inequality operators need to accept FLOAT operands

Refactoring, eg

- Multiple if blocks

Future Work

Some parts of the grammar not yet implemented, e.g.

- String literals
- Lambda expressions

Adding reversible features, especially guards

Future Work

Add control structures, eg

- Selection (“if-then-else-end”)
- Iteration (“while-do-end”)

Implement arbitrary types

Implement user-defined functions

Advantages

This parsing techniques easily handles descent both by right and by left recursion.

so . . .

If the name “RUTH” has already been used, we must resort to . . .

Reversible
University of
Teesside
High-level language, with

Descent by

Recursion