Reverse Polish Notation (RPN)

- **Well-known from HP calculators**
  - HP 48 used to be ubiquitous in engineering schools
  - HP 12c everywhere in the financial services industry
- **Opertor follows its operands**
  - 3 4 +
  - 5 3 4 * +

Stack-oriented programming languages

- **Forth**
- **Postscript**
  - Page description language
  - PDF is a subset of postscript
    - Removed if and loop, etc.
    - PDF is already interpreted: File format, not a programming language
- **Inter**
  - Simple stack-based language used as target for your compiler project

Inter

- **Two address spaces**
  - Separate code space and data space
  - Only the data space is available to the programmer
- **Data space**
  - Addresses from 0 to 5119
  - Addresses from 1024 are part of the stack
  - Stack pointer
- **Stack operations**
  - push c: push a constant value c onto the stack
  - pop: remove the top value from the stack
  - write: remove the top value from the stack and print it
  - read: read a value and push it onto the stack
  - swap: swaps the topmost value with the one below it
- **Arithmetics on the stack**
  - not, odd, +, uminus, -, *, /
  - Examples:
    - a + b * c
      - a b c * +
    - (a-b)/(a+b) (a b -- result)
      - Need to use a and b twice. How?
- **The stack pointer**
  - pushsp: Pushes the current value of the stack pointer onto the stack
  - rvaltop: Pop a value off the stack, consider it an address, fetch the value at that address and push it onto the stack.

- (a-b)/(a+b) (a b -- result)

```
pushsp
push 2
-
rvaltop  -- stack now contains a b a
pushsp
push 2
-
rvaltop   -- stack now contains a b a b
+         -- stack now contains a b (a-b)
pushsp
push 3
-
rvaltop   -- stack now contains a b (a-b) a
pushsp
push 3
-
rvaltop   -- stack now contains a b (a-b) a b
+         -- stack now contains a b (a-b) (a+b)
/         -- stack now contains a b ((a-b)/(a+b))
swap      -- stack now contains a ((a-b)/(a+b)) b
pop       -- stack now contains a ((a-b)/(a+b))
swap      -- stack now contains ((a-b)/(a+b)) a
pop
```

  - Very inconvenient
  - Inter is clearly not a very nice programming language
    - Doesn't matter for our purpose.
    - The compiler will write the code.
    - Names like a and b in epsilon will refer to variables (local or global).
      - How to get to the variable values?

- **Global variables**
  - Inter does not have a concept of variables
  - It does, however, allow us to access memory.
  - The portion of memory below the stack is suitable for global variables.
    - Suppose global variable "a" is stored in memory location 4 and "b" is stored in location 5.
    - (a-b)/(a+b)

```
rvalue 4
rvalue 5
-
rvalue 4
rvalue 5
+
/
```

    - Can we always find the memory location of a global variable at compile time? Yes!

- **Local variales**
  - Local variables stored in the activation record, somewhere on the stack.
  - Activation record
    - Return value
    - Parameters
    - Return address
    - Old base address
    - Local variables
  - Idea: If we have a pointer to the activation record, we can compute the value of any parameter or local variable by adding an offset to that pointer.
    - Called a base address, base pointer, or frame pointer
    - The base address is known only at run time
    - Store the current base address at some static memory location (e.g., 0)
  - One possible calling convention:
    - The caller pushes a (bogus) return value (just to reserve the space)
    - The caller pushes the parameters
    - The caller does a "call". This pushes the return address.
    - The callee pushes the old base address
    - The callee sets the current base address to the current stack pointer.
    - The callee pushes local variables
    - The callee's code executes.
    - The callee pops its local variables.
    - The callee restores the old base address and pops it.
    - The callee sets the return value.
    - The callee does a "ret". This pops the return address.
    - The caller pops the parameters (and does copy-out).
    - The caller is left with the return value on the stack.
- **Routine calls in Inter**
  - "call" and "ret" push and pop the program counter, respectively.
  - "call" takes an operand (a label)
    - The label is a unique string in the Inter program.
    - Has no significance; it only refers to an address in the code space.
  - Other branch operations: goto, gofalse
    - Use for loops

Forth

- **Lack of syntax**
  - Program is a list of words
  - Lack of syntax appears foreign to most programmers
    - Possible to get used to; don't get xenofobia
    - Flexible: Structure of the program given by words you define yourself
      - Object system defined in Forth itself
    - "Forth can have any syntax"

- Untyped
- **Widely used**
  - Small microcomputers in the 1980s
  - Embedded systems
  - OpenBoot in Sun, Apple, and IBM computers
- **Parameter stack**
  - Like the stack in Inter
  - More complete set of words for stack manipulation
    - swap, dup, over, rot, drop
      - : over swap dup rot swap ;
    - 2swap, 2dup, 2over, 2drop
    - Example: a2 + ab + c (c a b -- result)
      over dup * rot rot * + +
    - Example: (a+b)/(a-b)
      2dup + rot rot - swap /
- **Dictionary**
  - Maps words to definitions
  - Words, variables, and constants
- **Defining new words**
  - : star 42 emit ;
  - : over swap duip rot swap ;
  - Building the language bottom-up
- **Loops**
  - 10 0 do i . loop
    - prints "0 1 2 3 4 5 6 7 8 9"
  - 3 10 0 do dup i + . loop
    - prints "3 4 5 6 7 8 9 10 11 12"
  - How does it work?
    - Need the concept of the return stack
- **Return stack**
  - Second stack, normally not visible.
  - Used by the system to keep activation records.
    - Can only be used within definitions (procedures)
    - Must pop anything you push
  - "do" pushes 10 and 0 onto the return stack
  - "i" copies the top of the return stack onto the parameter stack
  - loop increments the top of the return stack, and repeats if less than the second element on the return stack
  - When the loop terminates, "loop" pops the 10 and the index.
  - Could loop have been implemented using only the parameter stack?
    - Yes, but it would interfere with the user's code
    - Couldn't, for instance, push 10 number on the stack:

        - 10 0 do i loop
- **The Forth interpreter**
    - Read a word
    - Look it up in the dictionary