Formal Languages and Compilers

Compiler and stack based machines

The compiler

- A compiler take the program source file and transforms it into an equivalent program written in another language (destination language)

- We will build a compiler that give us an intermediate language…

- …for a stack based machine..

Stack based machines

- Stack machine is a computer model that uses a pushdown stack rather than the classical registers.
- Use reverse polish notation
- Easier for us, we can obtain a general valuable compiled source rather then mere raw assembly code (whatever other destination language)

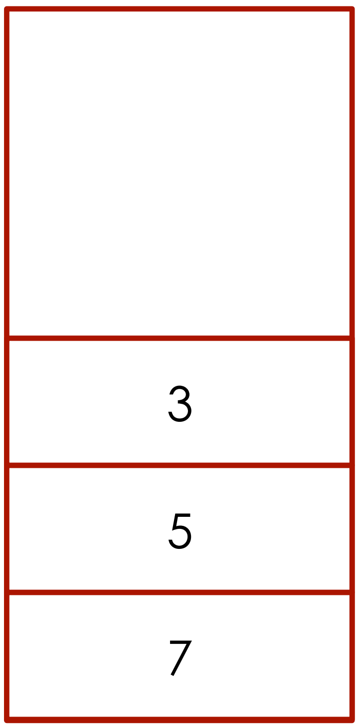
Reverse polish notation

- Aka postfix notation or RPN
  eg: 5 + 6 → 5 6 +
- Works in synergy with the stack based machine.
- Read tokens from the input, if they are number push them on the stack otherwise perform the operations by consuming items on the stack. (*sounds familiar?*)
- Errors are managed by valuating the length of the remaining items on the stack
  EG: "+" is a binary function, if you have 1 item in the stack → something wrong :S

Postfix notation

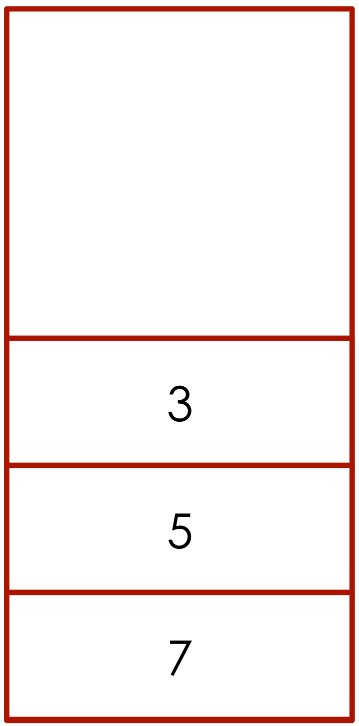
- infix expression "((6+4)/2)+3"
- Postfix 6 4 + 2 / 3 +
  due to precedence introduced by parenthesis
- Suppose we don't have parenthesis
  infix "7 + 5 * 3 – 2"
  postfix 7 5 3 * + 2 -
- EXERCISE for home: think how to build a tool with the use of lex + yacc to move from infix to postfix notation
- Postfix notation and the previous exercise, combined together are useful to understand how does the stack based parser work, as well as the stack based compiler work..

7 5 3 * + 2 - evaluation algorithm




```
While ((token = read next token) != null)
|     if typeof(token) = value then push it onto the stack
|     else if typeof(token) =  (operator) OP n-ary then
|     |           if stack.length < n then error
|     |           else
|     |                 pop n values = <args>
|     |                 res = evaluate OP(<args>)
|     |                 push res on stack
end while
if stack.length == 1 then return that value
else error
```

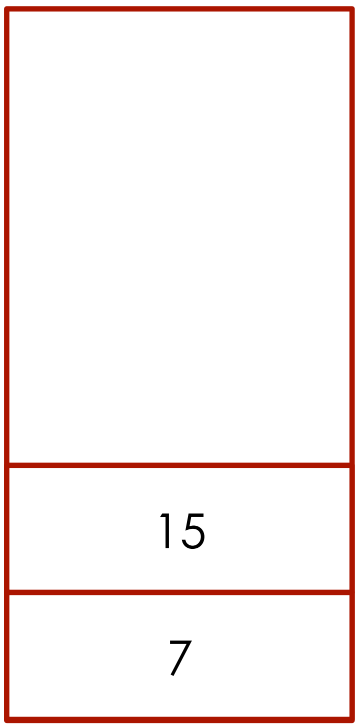
7 5 3 * + 2 - calculating result



Remaining input:
* + 2 -

Push values on the stack

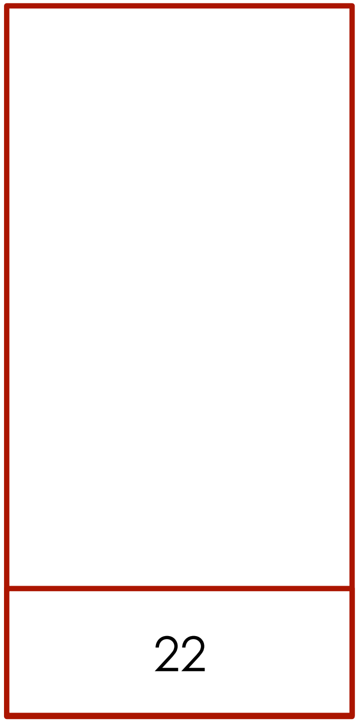
7 5 3 * + 2 - calculating result



Remaining input:
+ 2 -

* Is binary → pop 2 values
evaluate operator → 5 * 3 = 15
push 15 onto the stack

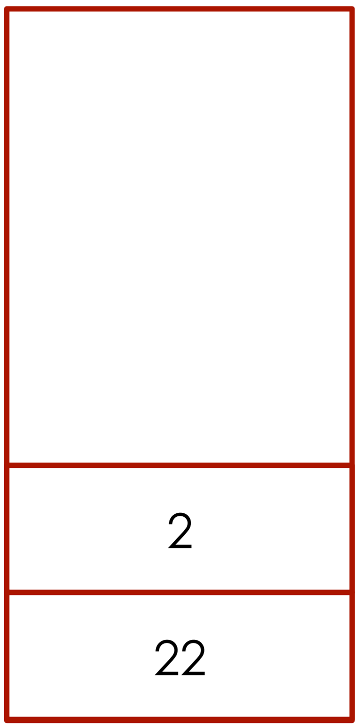
7 5 3 * + 2 - calculating result



Remaining input:
2 -

+ Is binary → pop 2 values
evaluate operator → 15 + 7 = 22
push 2onto the stack

7 5 3 * + 2 - calculating result



Remaining input:
-

2 is a value – push it onto the stack

7 5 3 * + 2 - calculating result

20

Remaining input:

- Is binary → pop 2 values
evaluate operator → 22 – 2 = 20
push 20 onto the stack

No more input, stack length equals 1 so such lonely item is the result

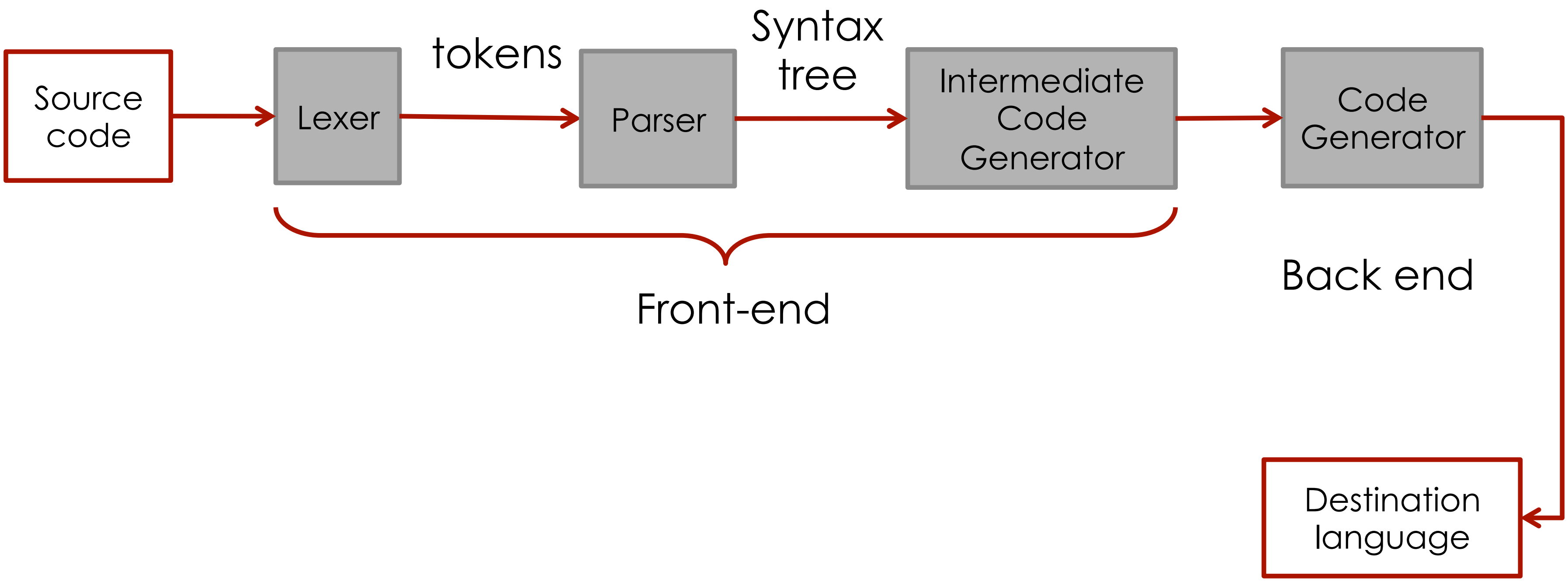
Stack based machine (cont)

- We use stack machines because they are useful for us and keeps our work simple..
  Any how they have..
- Some advantages like: simple interpreter and compilers, easy to implement
- As well some disadvantages:
  local variables management, register management is completely missing..

Compiler

- By the code provided for a stack based machine we can obtain code for different architectures..
- The compiler of our calculator is just a function that takes in input the syntax tree built by the front-end (lex+yacc) and produces as output an equivalent program in stack based code.

Front-end structure



Example

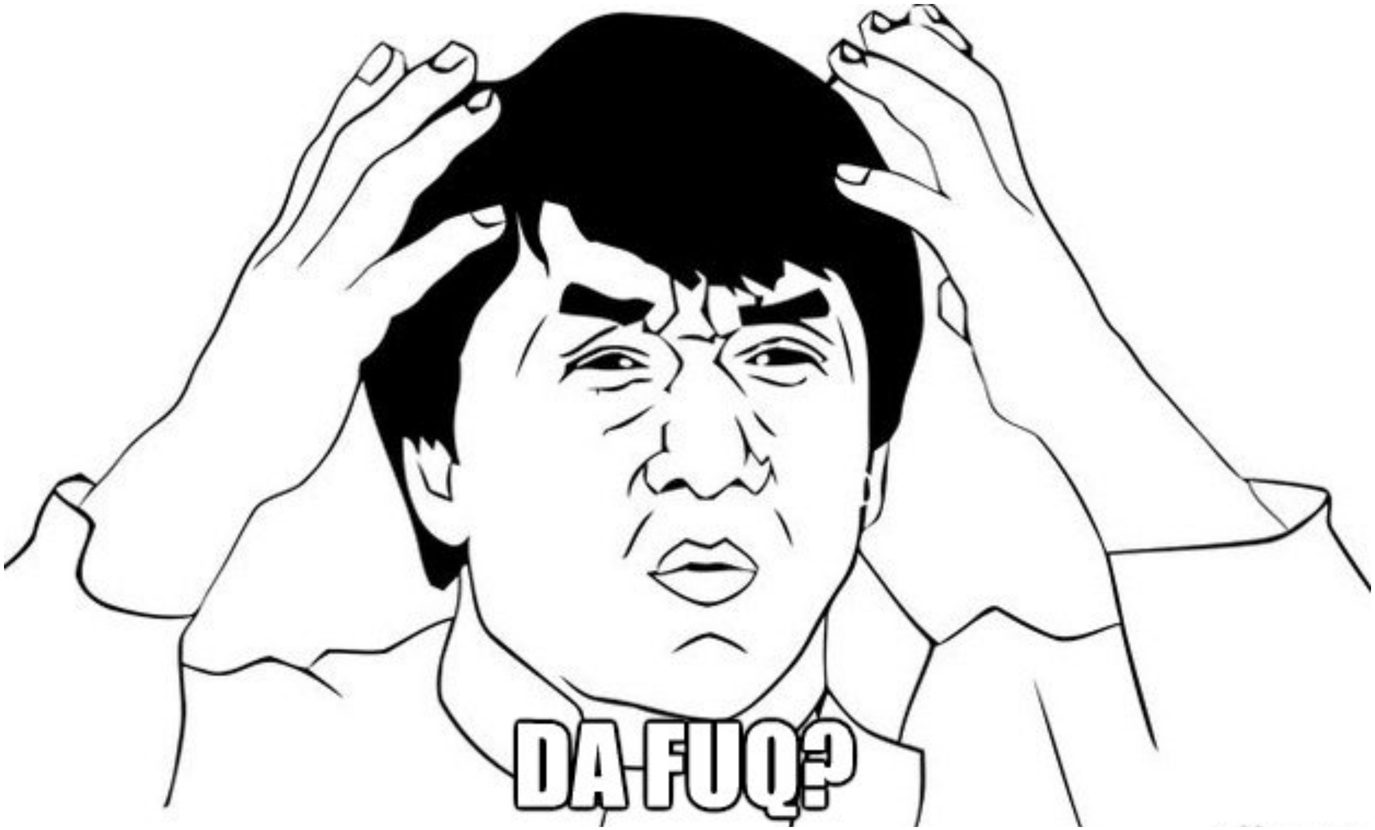
- The following is a legit program, given our specification

```
i = 0;
While(i<3){
        Print I;
        i = i +1;
}
```

- What do you expect it to do?
- What do you expect by the compiled version, in its stack machine code…?

Result

```
    push    0
    pop     x
L000:
    push    x
    pop     3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```



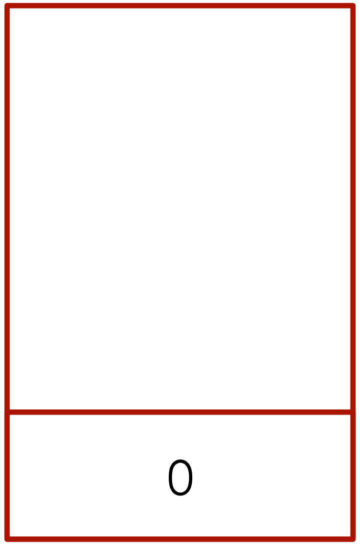
Result (cont.)

- Result is obtained by passing the whole tree to the execution function which evaluates each node
- To each node corresponds a set of action to be executed (as we already saw in the interpreter version)
- Result is more complex
- Result is not a numeric value, rather is a way to compute such value for any given architecture (in other words.. We obtain an equivalent program/ procedure written in another language…)

Result

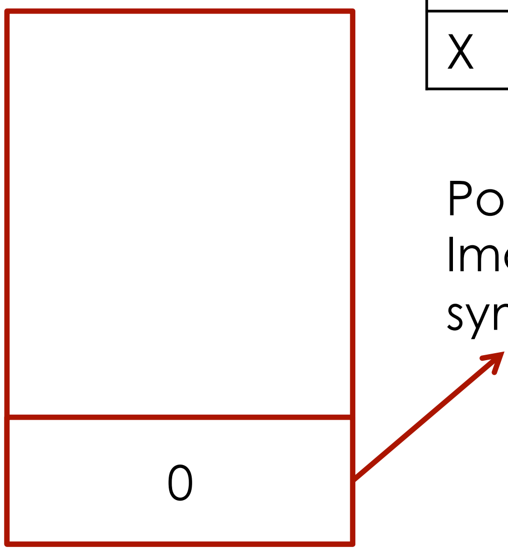
```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```

- Let's understand what is going on here.
- Remember we have only a stack for managing our program

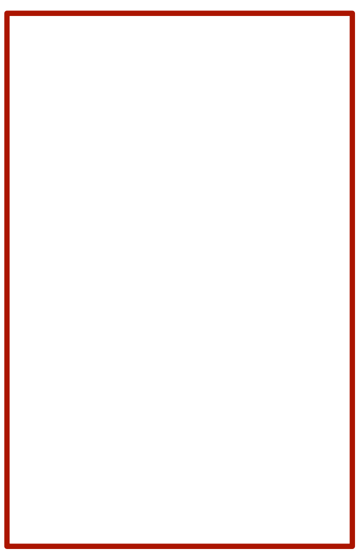


- Fist instructions: push a value and pop a variable? :S:S what does it mean? *We assign such var the popped value*

Var	Value
X	0



Pop x;
Imagine that the table on top is our symbol table

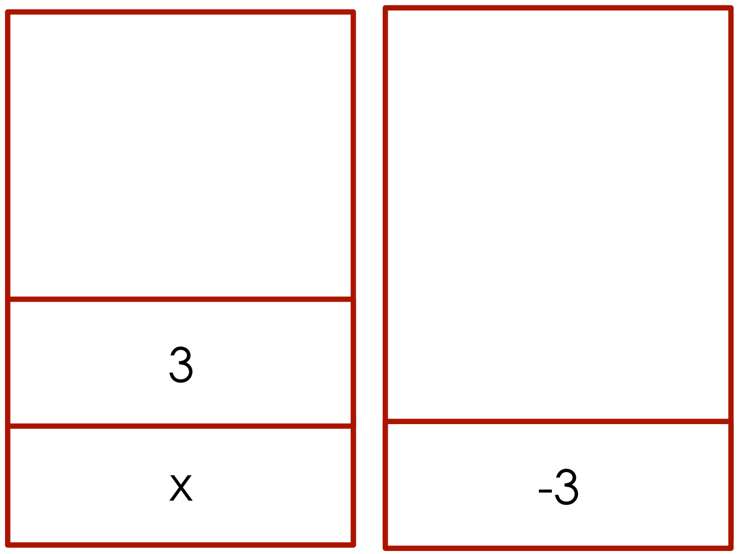


Situation after push 0 and pop x, let's move on

Var	Value
X	0

L000

```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```



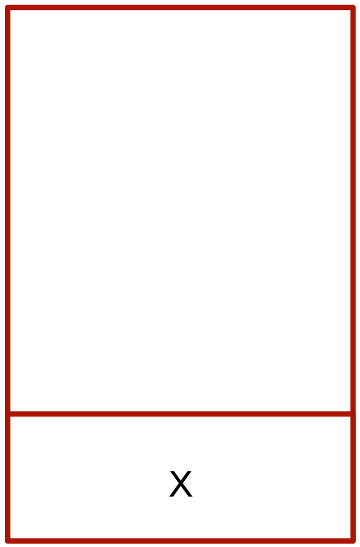
compLT = compare LT
generally cmp instruction subtracts two operands and sets a flag

jz = jump if zero and go to specified location
if this is zero go to L001

Var	Value
X	0

L000 (cont. push x)

```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```



Var	Value
X	0

L000 (cont. print)

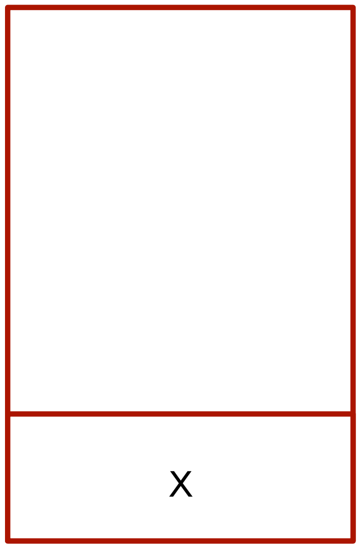
```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```

Print consumes x
SIDE EFFECT: zero is printed

Var	Value
X	0

L000 (cont. push x)

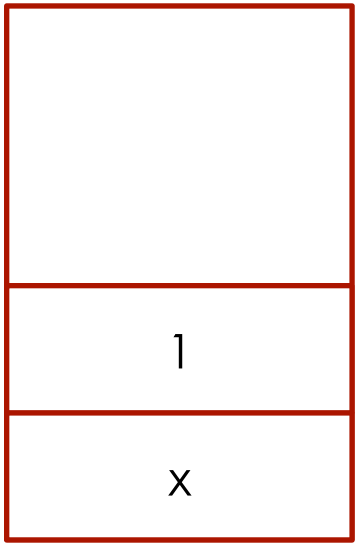
```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```



Var	Value
X	0

L000 (cont. push 1)

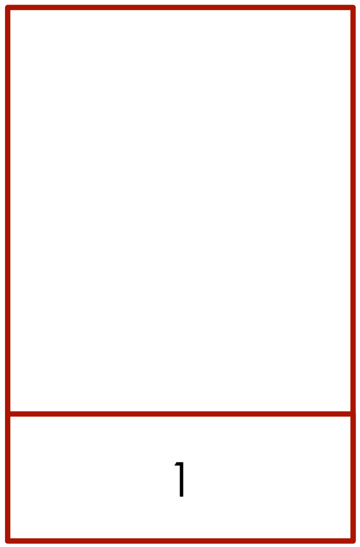
```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```



Var	Value
X	0

L000 (cont. add)

```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```



Add has arity = 2
in fact is called binary, pops 2
item from the stack, consumes them
and pushes back the result of the sum

Var	Value
X	1

L000 (cont. pop x and jump)

```
    push    0
    pop     x
L000:
    push    x
    push    3
    cmpLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```

Update value of x
Unconditional jump to location L000

we cycle over again.
Exercise:
Go to slide marked with L000
and repeat the steps
keeping the value of x updated until
you reach the end of the code.

L001

- Nothing more to do, no more instruction.
- Let's see the code to build this compiler.
- Open file 2.2 – calc / calcCompiler.c
  like in the interpreter code we find a switch at the top.. Let's look at the file together to understand what's happening.

  switch (p->type) {….}

  Recall p is a nodeType – defined in our header file

The base

- Functions: when we find a function we want to be able to apply it, thus we write the corresponding function..

- default: //match operators with 2 operands

```
ex(p->opr.op[0]);
ex(p->opr.op[1]);
 switch (p->opr.oper) {
        case'+':
                printf("\tadd\n");
                break;
              case'-':
                 printf("\tsub\n");
                break;
              ...
              }
```

Pushing items onto the stack

- When we find a variable or a constant we want to push it onto the stack

```
  case typeCon:
    printf("\tpush\t%d\n",p->con.value);
    break;
case typeId:
    printf("\tpush\t%c\n",p->id.i + 'a');
    break;
```

Operators

- How to treat more complex operators?
  Like =, IF or Print…

- IF

```
ex(p->opr.op[0]);
if(p->opr.nops > 2){
        printf("\tjz\tL%03d\n", lbl1=lbl++);
        ex(p->opr.op[1]);
        printf("\tjmp\tL%03d\n",lbl2=lbl++);
        printf("L%03d:\n",lbl1);
        ex(p->opr.op[2]);
        printf("L%03d:\n",lbl2);
}
Else{
        printf("\tjz\tL%03d\n", lbl1 = lbl++);
        ex(p->opr.op[1]);
        printf("L%03d:\n",lbl1);
}
```

Exercise

- Verify by your self that the if then else code works as expected

Solution

High level code	x = 0;
	If(x){
	  print x;
	}
		x = 1;
		If(x){
		  print x;
		}
			x = 0
			If(x){
			  print x;
			}else{
			  print 4;
			}

Compiled / intermediate code	Push 0 Pop x Push x Jz L000 Push x Print L000	Push 1 Pop x Push x Jz L000 Push x Print L000	Push 0 Pop x Push x Jz L000 Push x Print jmp L001 L000 push 4 print L001

Operators

- While

```
(1) printf("L%03d:\n",lbl1 = lbl++);
(2) ex(p->opr.op[0]);
(3) printf("\tjz\tL%03d\n", lbl2 = lbl++);
(4) ex(p->opr.op[1]);
(5) printf("\tjmp\tL%03d\n",lbl1);
(6) printf("L%03d:\n",lbl2);
```

- 1 – set the beginning of the loop: create a new label
  2 – 3 put the expression on the stack and evaluate the guard
  4 – execute statements
  5 – go back to the beginning of the loop (go to 2 namely)
  6 – label the end of the loop

Building the syntax tree

- While moving across the input source the front end “builds” a syntax tree.
- Is not always the case that such structure is built – the parser can be strong enough to avoid its construction
- Just for the sake of knowledge we will try to build our own graph representation

Syntax tree

- We will use the structure built so far, we will change only the ex function
- We define an interface that allow us to draw the tree, since is going to be output on the terminal this is a bit bare and complex.
- We need functions for
  - Drawing and defining the limit of a box
  - Drawing arrows
  - Drawing the entire graph
- Let's look at 2.2 – calc / calcGraph.c

Home exercise

- Insert grammar rules and action to manage the following statement (simplified version of for)

```
for (i = 0 to n){
        do something; // eg print i;
}
```

- Implement the ex function for both the interpreter and the compiler

Bibliography

- Tom Niemann – Lex and Yacc tutorial
  epaperpress.com/lexandyacc