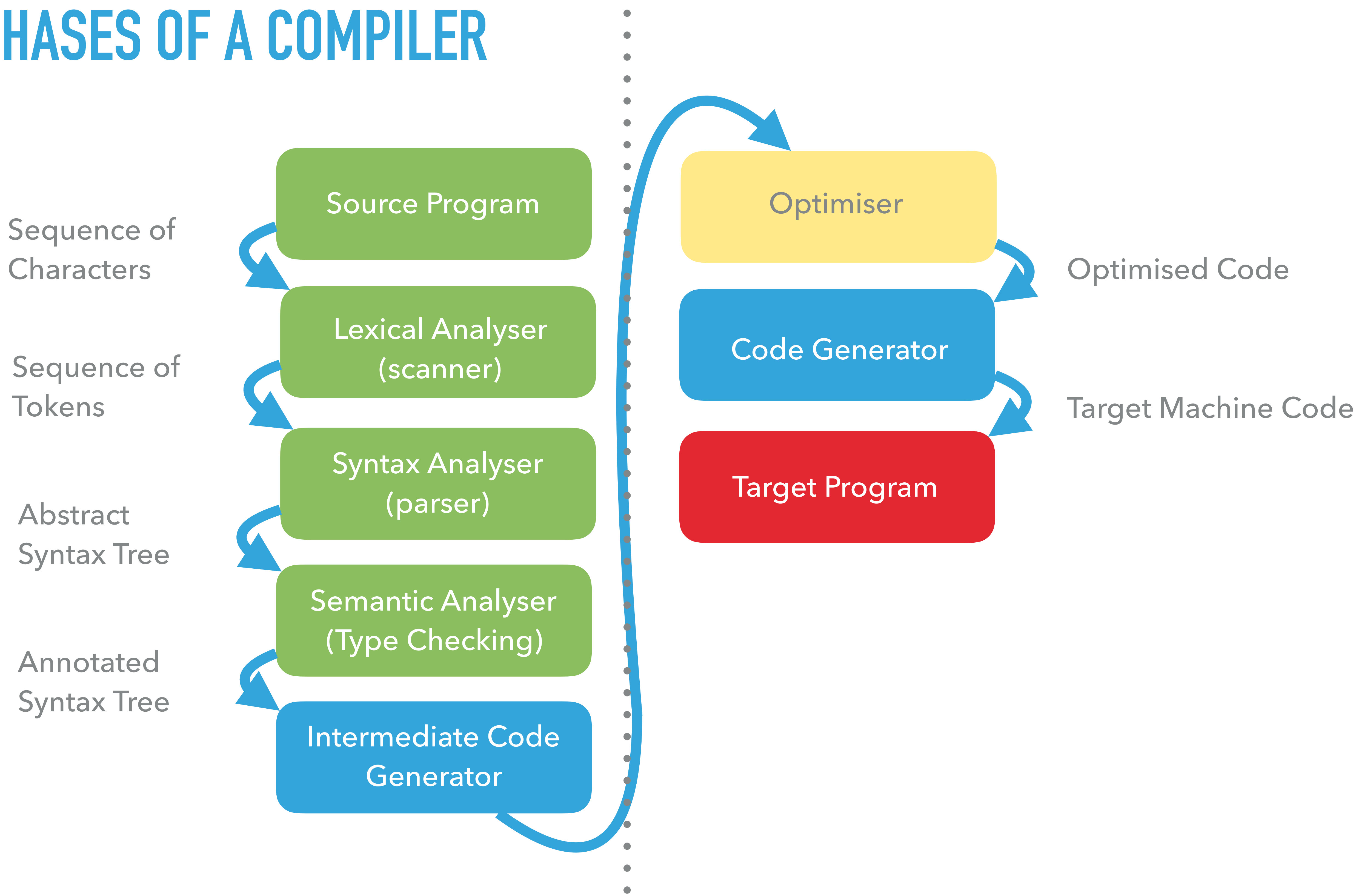
CM4106 - LANGUAGES & COMPILERS

TRIANGLE ABSTRACT MACHINE AND CODE TEMPLATING

THIS WEEK

- Run Time Organisation
- The Triangle Abstract Machine
- Code Templates

PHASES OF A COMPILER



RUN TIME ORGANISATION

- A compiler translates a high-level language program into an equivalent low-level language program.

EXPRESSIONS
OBJECTS
VARIABLES
PROCEDURES
METHODS

- Run-time organisation represents high-level structures in terms of low-level machines memory architecture

REGISTERS
MEMORY
MACHINE INSTRUCTIONS
STACK

TARGET MACHINE

- TAM is a simplified virtual machine that can execute programs compiled from block based languages (Triangle, Pascal, Algol)
- Everything his executed on a single stack
- Primitive arithmetic, logic and other operations are all handled by programmed functions

DATA-REPRESENTATION

- Many languages have lots of possible types (float, double, array, list)
- In Triangle-Reduced we really only need deal with primitives Integer, Char and Boolean
  - each of these types will require a different amount of space in memory
  - but each value of each type should use the same amount of space
  - In TAM each type is actually represented by a single 16-bit word

EXPRESSION EVALUATION

- The point in our compiler is to turn high level code in to low level instructions.
- Our **low level** machine has a number of **instructions** for basic arithmetic (add, multiply, divide etc)
  - all of these instructions work on **2 operands** (values) as you saw way back in lecture 2!
  - Where do we store the **intermediate results**?
  - on a **stack machine** it becomes the next value on the stack (easy)
  - On a **register machine** (pointers etc, much harder!)

STACK MACHINE INSTRUCTIONS

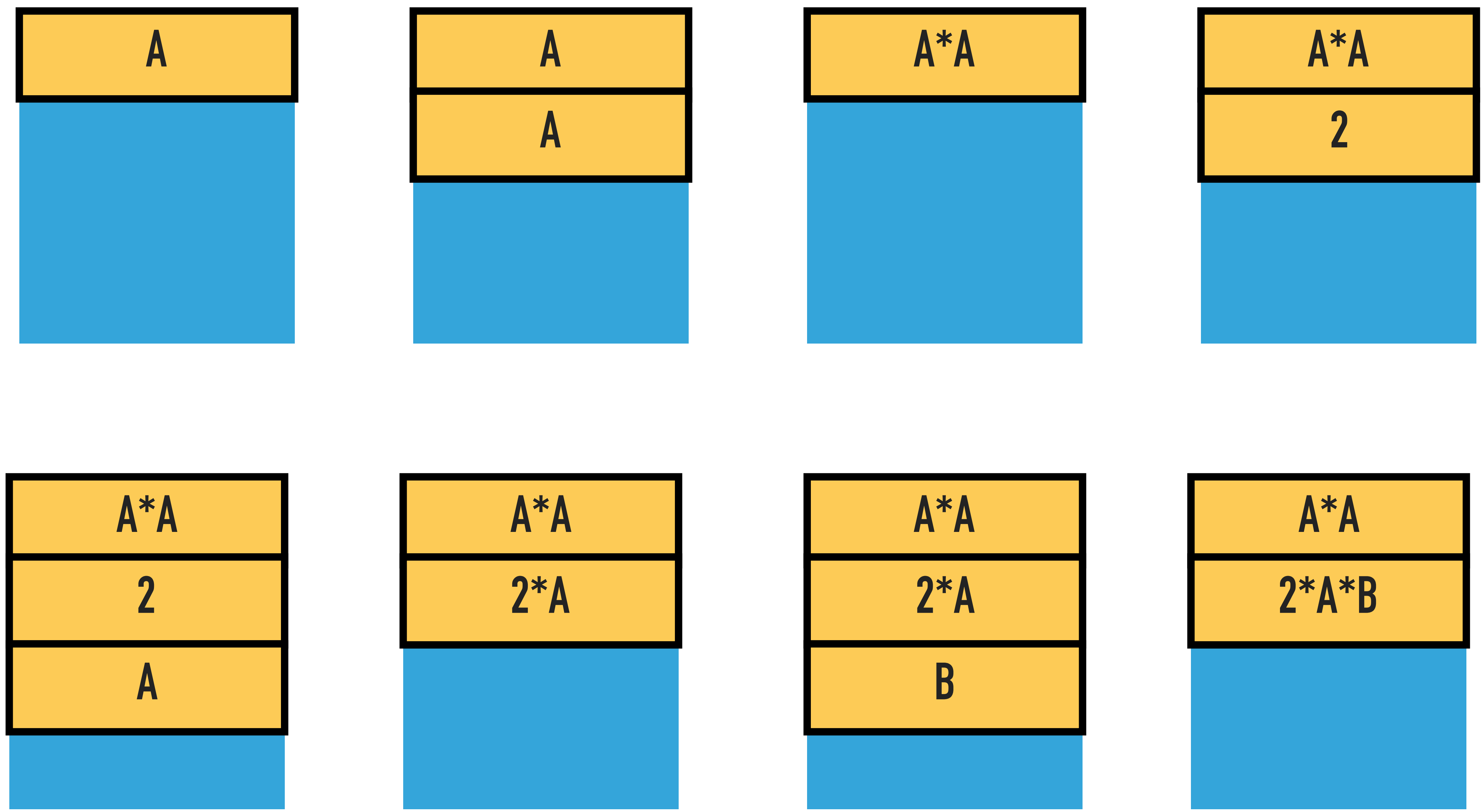
Instruction	Definition
STORE a	pop the **top** value off the stack and store it in address **a**
LOAD a	get a value from address **a** and **push** it back on the stack
LOADL n	**push** the **literal value n** onto the stack
ADD	**replace** the top to values on the stack with their **sum**
SUB	**replace** the top to values on the stack with their **difference**
MUL	**replace** the top to values on the stack with their **product**

STORE A
LOAD A
LOADL N
ADD
SUB
MUL

WHAT DOES THAT LOOK LIKE

- d := a*a + 2*a*b - 4*a*c;

```
LOAD a
LOAD a
MUL
LOADL 2
LOAD a
MUL
LOAD b
MUL
ADD
LOADL 4
LOAD a
MUL
LOAD c
MUL
SUB
STORE d
```

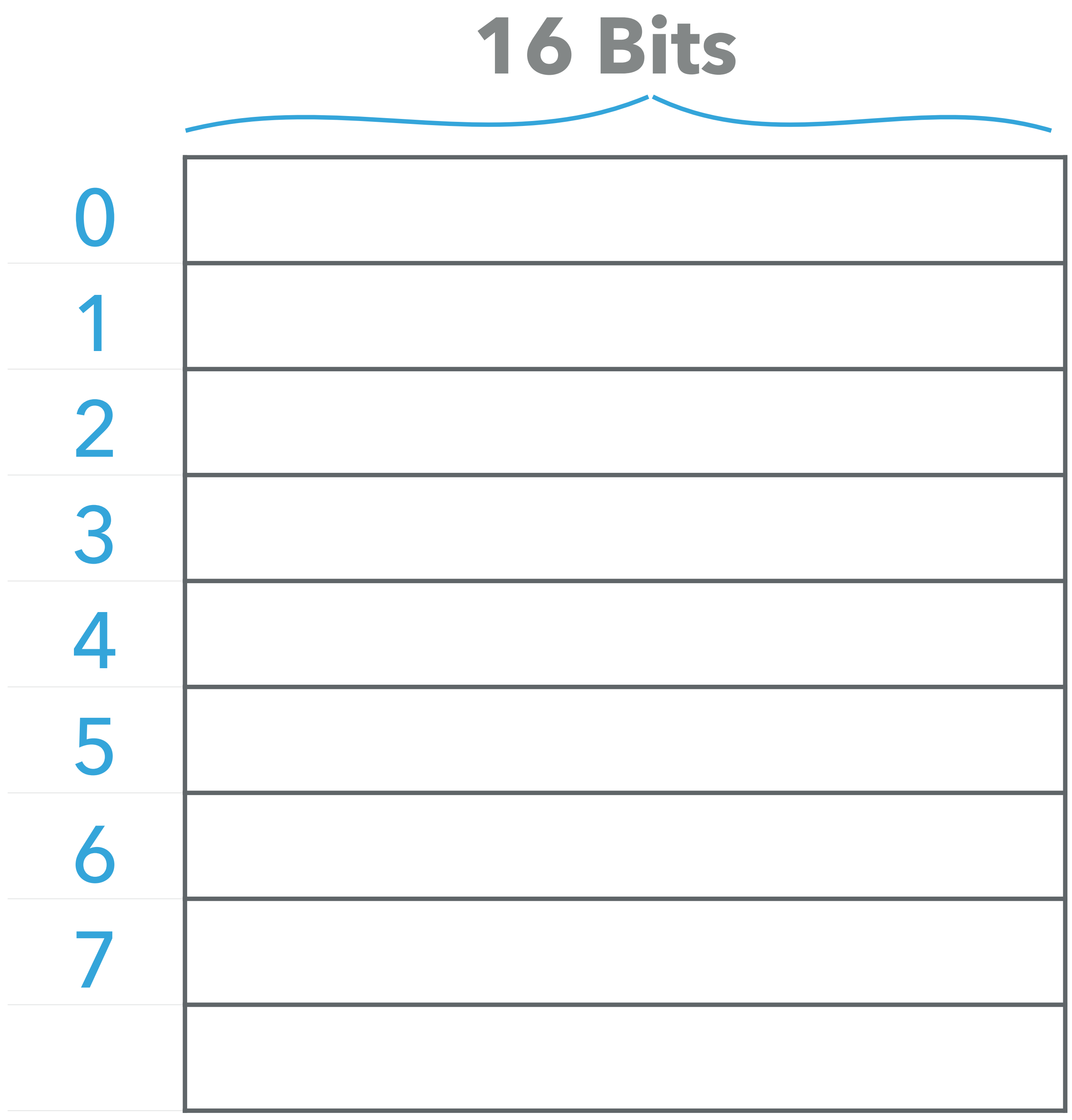


STORAGE ALLOCATION

- Our target machine will have a system for storing values
- Usually just our variable values set when we declare them
  - we need a way of getting these values
  - and adhering to any scope in our source code.
- The target machine has Global, local and a heap for storing our information

TAM STORAGE

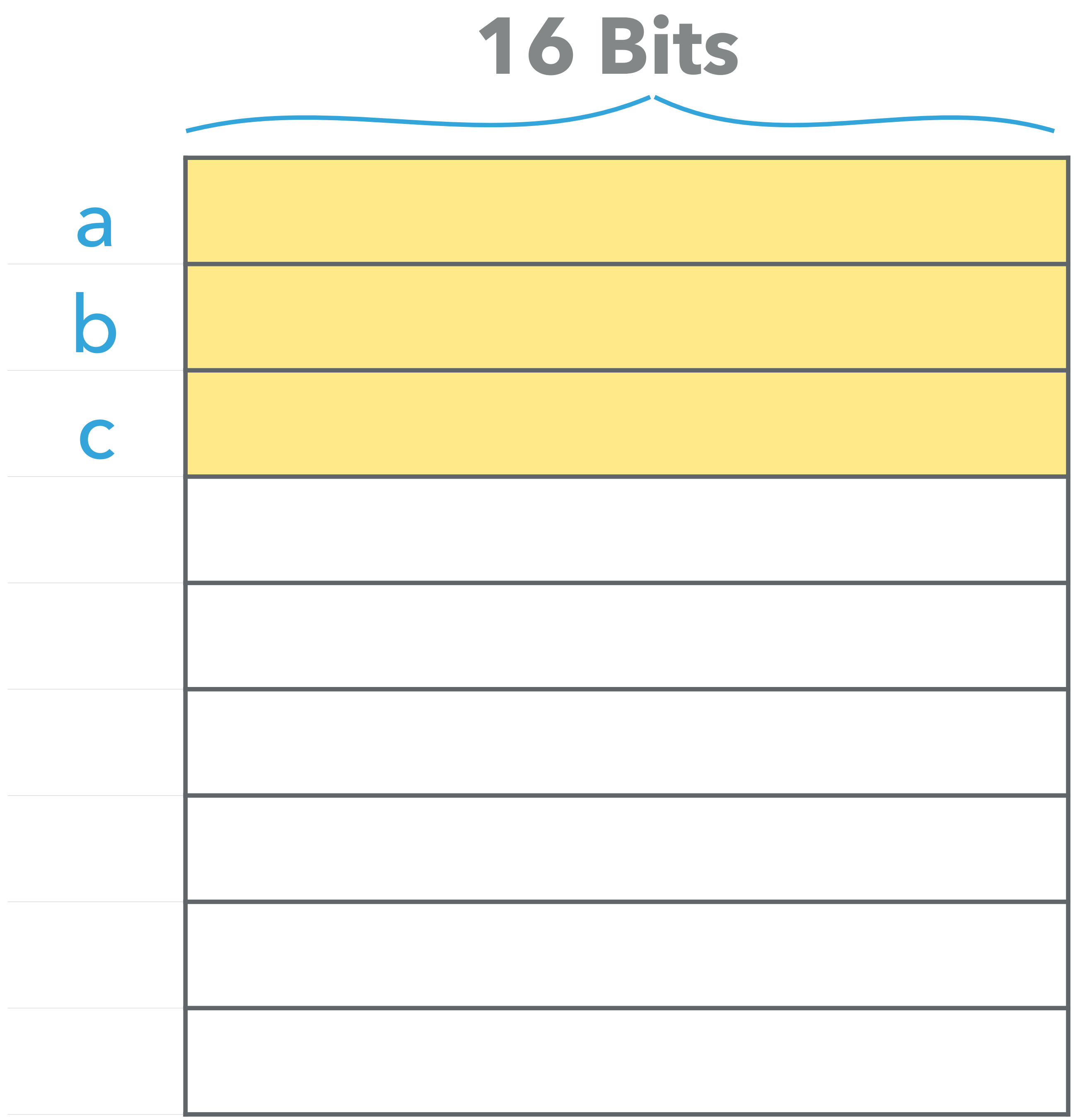
- Everything in TAM is stored in single 16 bit words



GLOBAL VARIABLES

- The compiler can compute how much space is needed
- and allocate it

```
let
 var a : Integer;
 var b : Char;
 var c : Boolean;
in
```



LOCAL VARIABLES

- defined inside Let-In blocks in triangle

```
let //global
var b: Boolean;
var c: Char
in
 let //block1
  var d: Integer
 in ...
 let //block2
  var f: Integer
 in
 begin ...
```

global variables; lifetime: throughout the program

Local to this level (block 1)

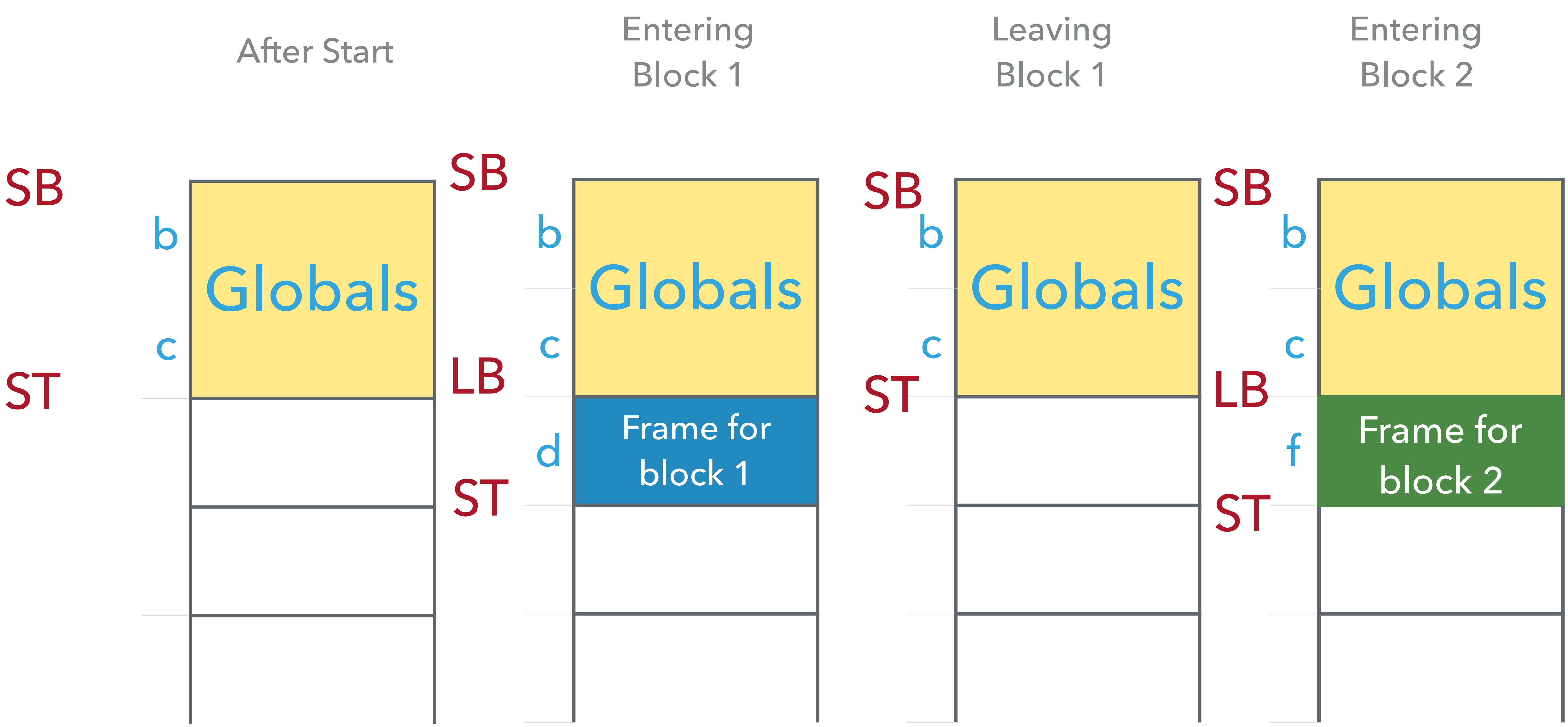
Local to this level (block2)

STACK FRAME

- Each block or procedure has a stack frame
  - local variables
  - administration data (return address, dynamic/static link)
  - actual parameters
- When the procedure is called its stack frame is allocated on the stack. When the procedure has ended, the stack frame is popped from the stack.

STACK ALLOCATION

Registers	
SB	Stack BAse
LB	Local Base
ST	Stack Top



STACK STORAGE (MEMORY) INSTRUCTIONS

- **LOAD d[reg]** – push the value at address d relative to the contents of reg (e.g., SB or LB) onto the stack.
- **STORE d[reg]** – pop the value on top of the stack to the address d relative to the contents of reg (e.g., SB or LB).
- Accessing Variables
  - **global variables** are in the **SB** frame
    - LOAD d[SB] and STORE d[SB]
  - **local variables** are in the **LB** frame
    - LOAD d[LB] and STORE d[LB]

NESTING

- This gets more complicated with nested procedures but from our symbol table we already know the scope of the variable so we can define levels

```
let ! level 1 var a: Integer;
 let ! level 2 var b: Integer;
  let ! level 3 var c: Integer;
   let ! level 4 var d: Integer;
   in ...
  in ...
 in ...
in ...
```

	level	scope	address
a	1	global	0[SB]
b	2	local -2	3[L2]
c	3	local -1	3[L1]
d	4	local	3[LB]

ROUTINE CALLS

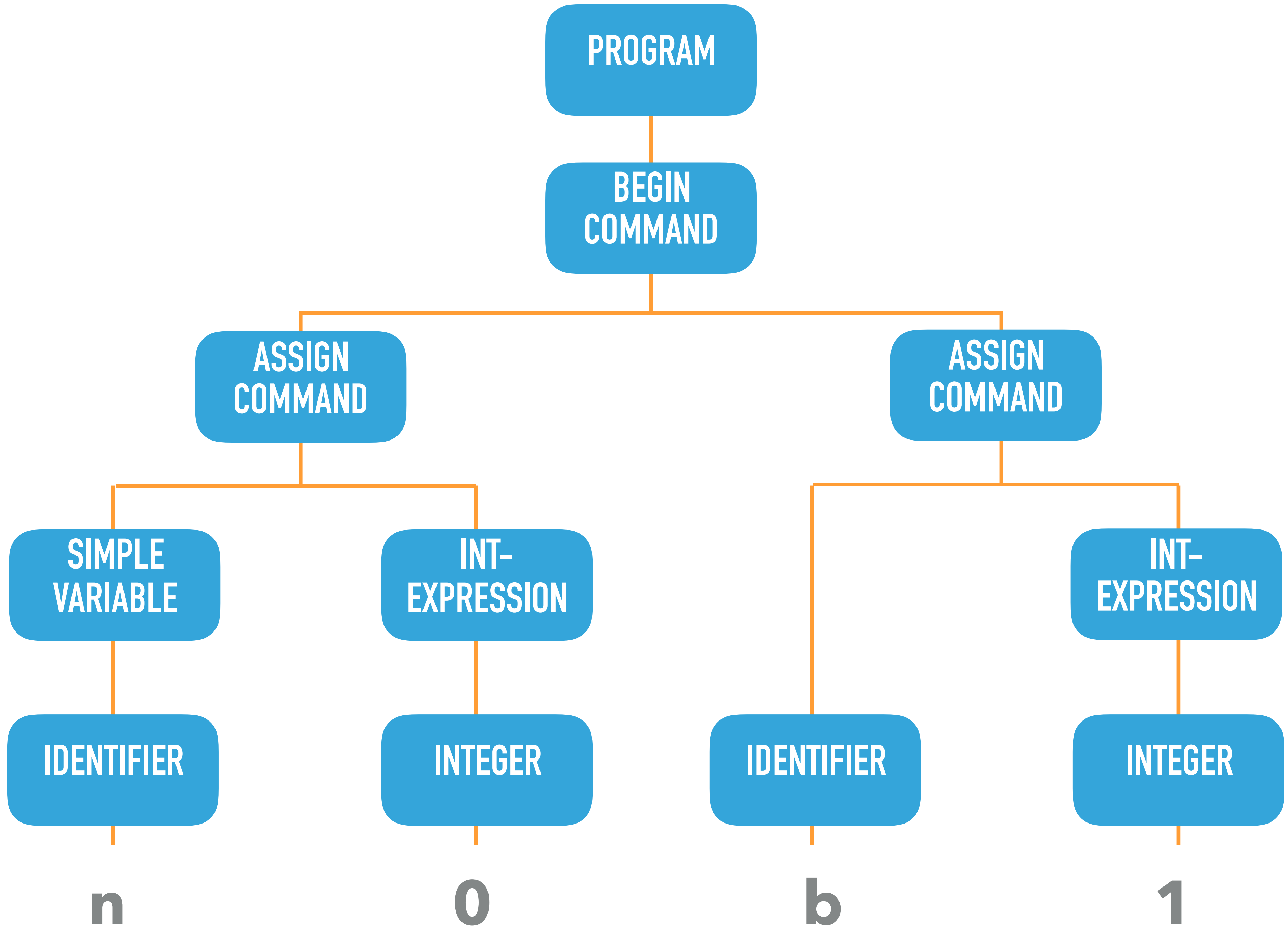
- In our version of Triangle we haven't looked at routes or procedures
- But there are some builtin ones. eg putint() and getChar etc
- The machine has instructions for these too
  - CALL r - **push** current address to the call stack and jump to instruction **r**
  - RETURN - **pop** address from the call stack and transfer control to this saved address
- procedures are stored in the stake too, all we do is jump to their position!

SO HOW DO WE COMPILE TO THIS?

- These cover most of the instructions used by TAM
- The whole set is stored in the Machine.cs file which is part of the Triangle.NetCore package in the Triangle.AbstractMachine project
- LOAD, LOADA, LOADI, LOADL, STORE, STOREI, CALL, CALLI, RETURN, PUSH, POP, JUMP, JUMPI, JUMPIF, HALT
- We need to translate from our source into these.

FROM LAST WEEK

- We now have a annotated AST from the semantic analysis
- We know through our Visits that the tree is semantically correct.



SO?

- At each stage of our Annotated AST we have a representation of a correct phrase in that language.
- eg an declaration has the identifier X and the type Integer
- With only these two pieces of info how would you write a declaration in a language you already know? eg JAVA

SEMANTIC EQUIVALENCE

```
let
 var x: integer;
 var y: integer
in
begin
 y := 2;
 x := 7;
 printint(y);
 printint(x)
end
```

→

```
PUSH        2
LOADL       2
STORE(1)    1[SB]
LOADL       7
STORE(1)    0[SB]
LOAD(1)     1[SB]
CALL        putint
LOAD(1)     0[SB]
CALL        putint
POP         2
HALT
```

CODE TEMPLATES

- We know the **phrases** that we have in our language
- and we know the **instructions** that can be sent to the target machine.
- All of the structures follow a **set pattern** with **different values**.
- We can create a set of templates to turn an **annotated AST** node into a set of **instructions**

CODE FUNCTIONS

- the code templates we need fit into a small number of categories

Class	Code Function	effect of generated code
Program	*run* P	run the program P then halt, starting and finishing wit han empty stack
Command	*execute* C	execute the command C, possibly changing variables, but not expanding or contracting the stack
Expression	*evaluate* E	evaluate the expression E putting its value on the top of the stack
V-name	*fetch* V	push the value of the constant or variable named V onto the top of the stack
V-name	*assign* V	pop a value from the stack top and store it in the variable V
Declaration	*elaborate* D	elaborate the Declaration D expanding and contracting the stack to make space for new constants and variables

EXAMPLE CODE TEMPLATES

- A mini triangle program is simply a Command so the code template would be

```
execute C

HALT
```

- the execute would be passed to the correct execute method and the HALT is a instruction recognised by TAM

COMMAND TEMPLATES

- an assignment command - V :=E

```
evaluate E

assign V
```

E will be evaluated and the result put on the top of the stack, then when assign is called it will use that value

- a call command - I(E)

```
evaluate E

CALL p          (p is the address of I)
```

again E will be evaluated then the CALL instruction will use that value as the parameter

MORE COMPLEX COMMAND

if E then C1 else C2

```
evaluate E
JUMPIF(0)  g
execute C1
JUMP h

g: execute C2
h:
```

E will be evaluated and the result put on the top of the stack

JUMPIF instruction looks at the top of the stack and tests its value

if the value is 0 i.e false, control is passed to the instructions at address g: where C2 will be executed

Otherwise execute C1 (i.e the true command) and JUMP to h and continue with the rest of the instructions

ITERATING DOWN TO THE TERMINAL EXPRESSIONS

- so for a IntegerLiteral IL

```
LOADL v    (where v is the value of IL)
```

- or CharacterLiteral CL

```
LOADL v    (where v is the value of CL)
```

Both templates just load the value of the literal onto the top of the stack

EXAMPLE – TRANSLATION OF A WHILE

- while i > 0 do i := i -2 !valid statement in Triangle
- remember **while Expression do Command**

We are part of some bigger program and start at address 30 as illustration

execute the command i : i = i-2

sub represents the code within the while loop

evaluate the expression i>0

the execution jumps to 35 initially

then back to 31 if our while is true

Address	instruction	value
30	JUMP	35
31	LOAD	i
32	LOADL	2
33	CALL	sub
34	STORE	i
35	LOAD	i
36	LOADL	0
37	CALL	gt
38	JUMPIF (1)	31

THE ENCODER & EMITTER

- We pull this all together in classes called the Encoder and the Emitter
- The encoder visits the nodes of our AST just like the checker did last week
- but this time uses the Emitter to write out the machine code based on the templates.
- Building up an instruction set for the target machine.

SUMMARY

- Our target machine can use a number of instructions
- We need to create an instruction set from our source file which matches the semantics of the source program
  - i.e they have the same meaning
- A set of templates defines what our encoder, that visits our AST nodes will, write to the instruction set.