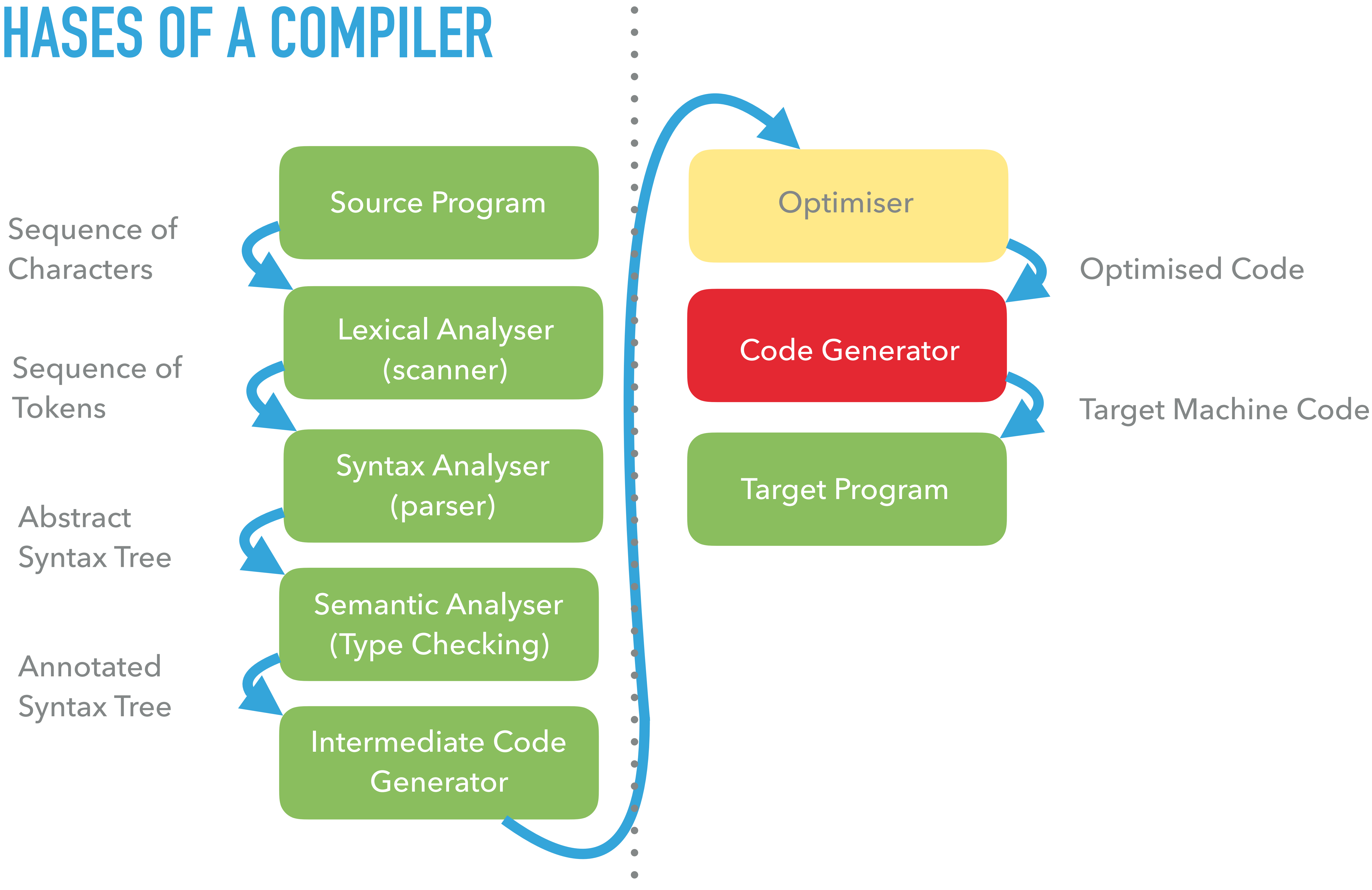
CM4106 - LANGUAGES & COMPILERS

MEMORY AND ADDRESSING

THIS WEEK

- Addressing & Jumps
- Known and unknown addressing
- Storage allocation

PHASES OF A COMPILER



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**

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 an 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

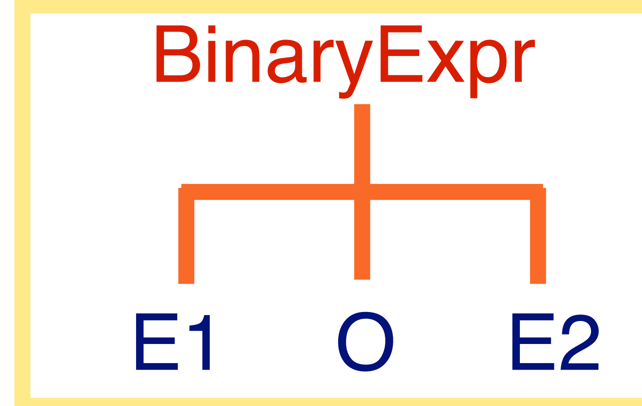
PHRASES TO VISITORS

Phrase	Visitor	Behaviour
Program	visitProgram	Generate code specified by **run P**
Command	visit…Command	Generate code specified by **execute C**
Expression	visit… Expression	Generate code specified by **evaluate E**
V-Name	visit… Vname	Return an **entity description** of the given value or variable name
Declaration	visit… Declaration	Generate the code specified by **elaborate D**
Type-Denoter	visit… TypeDenoter	Return the size of the given type

ENCODER RECAP

- Last week you started to develop the Encoder
- The encoder implements the code templates that help us translate between the Source and Target languages.

BINARY EXPRESSION



- evaluate [E1 op E2] = evaluate [E1]
  evaluate [E2]
  CALL op

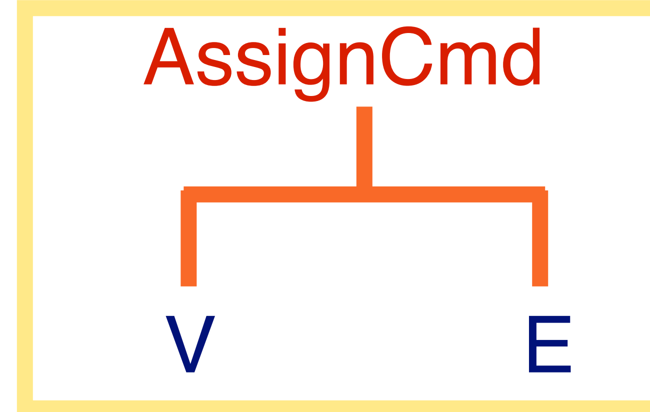
Memory Management

Size of the type

Expand the frame to fit the values

```
public int VisitBinaryExpression(BinaryExpression ast, Frame frame)
{
    var valSize = ast.Type.Visit(this, null);
    var valSize1 = ast.LeftExpression.Visit(this, frame);
    var frame1 = frame.Expand(valSize1);
    var valSize2 = ast.RightExpression.Visit(this, frame1);
    var frame2 = frame1.Replace(valSize1 + valSize2);
    ast.Operator.Visit(this, frame2);
    return valSize;
}
```

ASSIGN COMMAND



- execute [V:= E] = evaluate [E]
  assign [V]

Generate the code to push the expression value to the top of the stack

```
public Void VisitAssignCommand(AssignCommand ast, Frame frame)
{
    var valSize = ast.Expression.Visit(this, frame);
    EncodeAssign(ast.Vname, frame.Expand(valSize), valSize);
    return null;
}
```

EncodeAssign will generate the code to assign the value from the top of the stack,

(whatever the expression evaluates to) to the V-name

JUMPS & ADDRESSING

- execute [while E do C] =

Loopwhile:	evaluate [E]		Backwards Jump
	JUMPIF(0)	Loopend	
	execute [C]		
	JUMP	Loopwhile	
Loopend:			

Backwards jumps are easy: the "address" of the target has already been generated and **is known**.

FORWARD JUMPS

- **Forward jumps** are harder
- When the JUMP is called the target does not yet exist, so it has no address yet.
- execute [while E do C] =

```
Loopwhile:    evaluate [E]
              JUMPIF(0)        Loopend
              execute [C]
              JUMP             Loopwhile
Loopend:
```

Forwards Jump

BACKPATCHING

- The solution is backpatching - you saw this in the lab last week

- **Emit jump** with "**dummy**" address (e.g. simply 0).
- **Remember** the address where the jump instruction occurred.
- When the **target label is reached**, go back and **patch the jump** instruction.

IN ACTION

```
public Void VisitWhileCommand(WhileCommand ast, Frame frame)
{
    var jumpAddr = _emitter.Emit(OpCode.JUMP, Register.CB); 1
    var loopAddr = _emitter.NextInstrAddr; 2
      ast.Command.Visit(this, frame); 3
      _emitter.Patch(jumpAddr);4
      ast.Expression.Visit(this, frame);5
      _emitter.Emit(OpCode.JUMPIF, Machine.TrueValue, Register.CB, loopAddr)
    return null;                                                          6
}
```

- emit a jump with no (dummy) address
- store the loop address - to loop back to
- visit the command - we have no idea how long this is
- then patch the jump address back in - once we have sorted the command
- evaluate the expression for the while condition
- then emit the JUMPIF to test the expression value

CONSTANTS AND VARIABLES

- The **LetCmd** is where declarations appear.
- Variables and Constants are given a memory address relative to the Stack Base (see last week)

fetch [V] = LOAD(1) d[SB]

assign [V] = STORE(1) d[SB]

Where d is the address of the variable

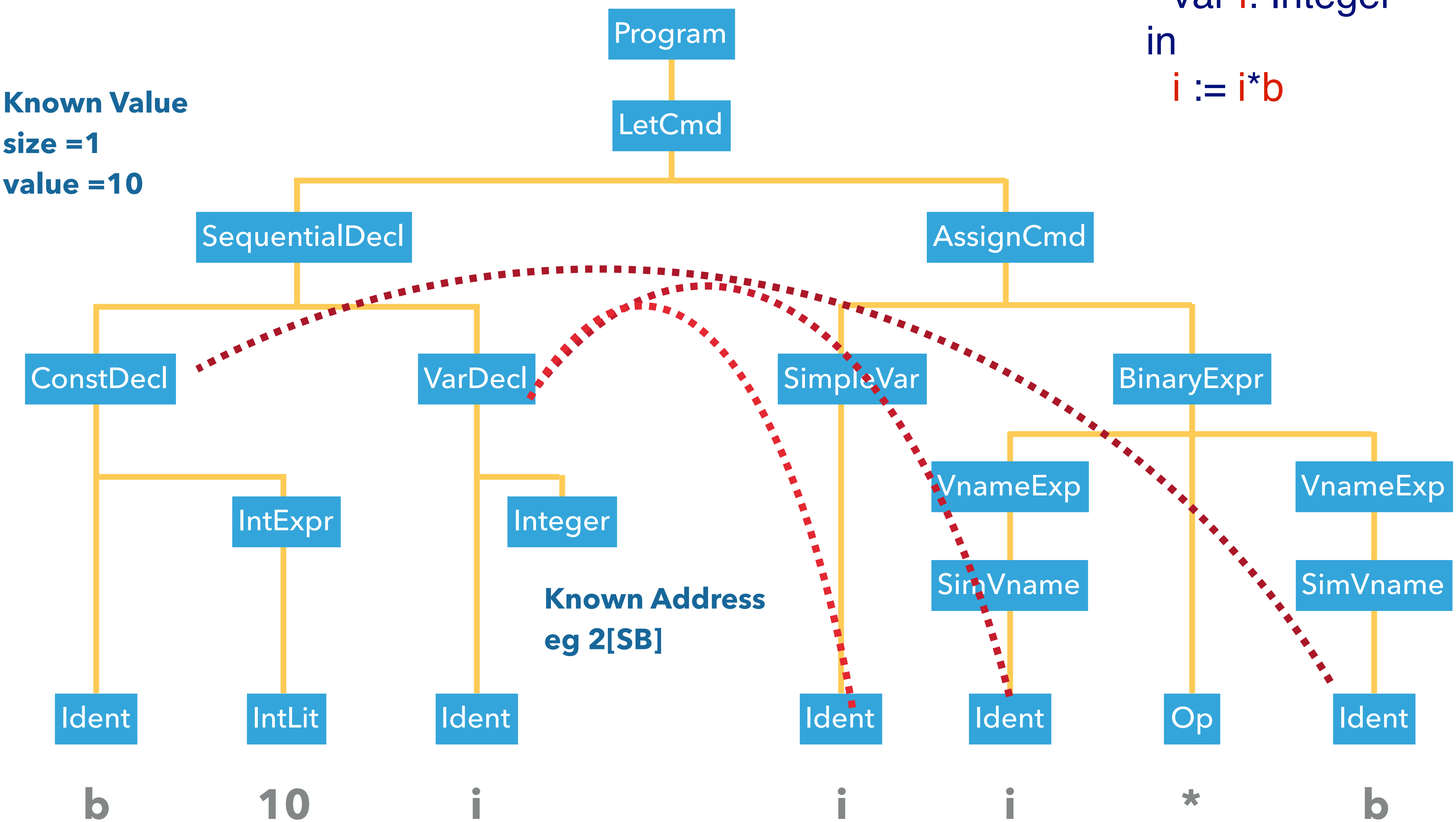
KNOWN VALUE & KNOWN ADDRESS

```
let
  const b ~ 10;
  var i: Integer
in
  i := i*b
```

```
PUSH    1
LOAD    2[SB]
LOADL   10
CALL    mult
STORE   2[SB]
POP     1
```

1) Push a space onto the stack
2) Load the value at address 2[SB] (where i is stored)
3) Load the constant
4) Multiply them calling built in must function
5) Store the result back at address 2[SB]

```
let
  const b ~ 10;
  var i: Integer
in
  i := i*b
```



UNKNOWN VALUE & KNOWN ADDRESS

Known Address = 5

```
let
  var x: Integer
in
let
  const y ~ 365 + x
in
  putint(y)
```

Unknown Value
size = 1
address =6

```
PUSH     1         ; room for x
PUSH     1         ; room for y
LOADL    365
LOAD     5[SB]     ; load x
CALL     add       ; 365+x
STORE    6[SB]     ; y ~ 365+x
LOAD     6[SB]
CALL     putint
POP      1
POP      1
```

Y is not known at compile time

DEALING WITH VARIABLES & CONSTANTS

- When a declaration is encountered the code generator binds the ID into an entity description
  - - known value: record the **value** and its **size**
  - - known address: record the **address** and **reserved space**

IDENTIFIER OCCURRENCE

- When an Identifier is encountered the code generator consults the entity description bound to it.
- then translates the entity

known value	const declaration using a literal
unknown value	const declaration using an expression
known address	variable declaration
unknown address	argument address bound to a var-parameter

IMPLEMENTATION OF ENTITIES

```
public abstract class RuntimeEntity
{

  readonly int _size;

  protected RuntimeEntity(int size)
  {
     _size = size;
  }

  public int Size
  {
     get { return _size; }
  }
}
```

abstract class that handles the entity size

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 with an 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

KNOWN VALUE – ENCODE FETCH

```
public class KnownValue : RuntimeEntity, IFetchableEntity
{

      readonly int _value;

      public KnownValue(int size, int value)
         : base(size)
      {
         _value = value;
      }

      public void EncodeFetch(Emitter emitter, Frame frame, int size, Vname vname)
      {
         // offset = 0 and indexed = false

         emitter.Emit(OpCode.LOADL, 0, 0, _value);
      }
}
```

For a known value all we need to do is LOAD the literal value onto the stack

UNKNOWN VALUE – ENCODE FETCH

```
 public class UnknownValue : RuntimeEntity, IFetchableEntity
{

    readonly ObjectAddress _address;

    public UnknownValue(int size, int level, int displacement)
        : base(size)
    {
        _address = new ObjectAddress(level, displacement);
    }

    public UnknownValue(int size, Frame frame)
        :
        this(size, frame.Level, frame.Size)
    {
    }

    public void EncodeFetch(Emitter emitter, Frame frame, int size, Vname vname)
    {
            emitter.Emit(OpCode.LOAD, size, frame.DisplayRegister(_address),
                _address.Displacement);

    }
}
```

Get the current frame

Get where the current vale sits in that frame

For a unknown value all we need to LOAD the value at a specific address relative to the stack base

KNOWN ADDRESS - ENCODE ASSIGN

```
public override void EncodeAssign(Emitter emitter, Frame frame, int size, Vname vname)
{
   emitter.Emit(OpCode.STORE, size, frame.DisplayRegister(Address),
      Address.Displacement);

}
```

- Here we use the STORE operator to assign what ever is at the top of the stack at that point to the address defined by the variable name entity.
- this will be the v-names place in the current stack eg - the current frame

UNKNOWN ADDRESS

```
public override void EncodeAssign(Emitter emitter, Frame frame, int size, Vname vname)
{
     emitter.Emit(OpCode.LOAD, Machine.AddressSize, frame.DisplayRegister(_address),
                 _address.Displacement);

     emitter.Emit(OpCode.STOREI, size, 0, 0);
}
```

- If we have an Unknown Address we will still know the displacement within frame because we will know the frame size and size of the variable
- So we work out from this what address to load onto the top of the stack (not a value yet)
- We then store the next value on the stack into this address.

ENCODE FETCH AND ASSIGN

- There are a few more methods in each of the Known and Unknown entities that you are going to develop in the lab
- This will complete the compiler and you will then be able to compiler your own code into object files
- Remember I gave you an interpreter for TAM that you can run your compiled files against

SUMMARY

- We saw a number of techniques here that allow us to successfully generate code
- Backpatching lets us determine where to go when we need to jump forward to a future address
- The runtime entities let us represent things that will exist when the program is executed and to work out their value and memory size when they are used.

NEXT WEEK

- Revision Lecture for the Exam
- No new lab material