Game Architecture

Scripting

Today’s Agenda

- What is scripting?
- Wait… why are we doing this?
- Some things to consider
- Parser & generator concepts
    - Big topic: just enough to whet your appetite
- Virtual machines
- Integrating scripting with an engine
- Visual script
- Case study: Lua

What is scripting?

- Way of expressing (a portion of) game logic:
  - AI
  - Level mechanics
  - UI behavior
  - Weapons
  - Etc.
- Often:
  - Higher level than engine language
  - Allows faster iteration than core engine code
  - Not performance critical
  - More than half of ‘game code’ is script
- Implementation:
  - ‘Type-y’ language (e.g. Lua)
  - Visual programming (e.g. Unreal Blueprint)

Wait… why are we doing this?

Efficiency?

- Drops barriers for less technical developers
- Updates on-the-fly without reloading the game
- High level / domain specific script can do more with less
- Memory; script can be easily loaded and unloaded

Flexibility?

- Player generated content
- Post-ship functional updates without 1st party approval

Robustness?

- Constrained script environment avoids OOM errors, NULL pointers, etc.

So a good engine must have a scripting system?

Absolutely not.

A scripting system is complexity. Avoid it unless you really need it.

Small teams and small games often don’t need it.

Some things to consider

- Coroutines & cooperative multithreading
- Events vs polling
- Safety
- Role of script in the engine

Some things to consider

- **Coroutines & cooperative multithreading**
- Events vs polling
- Safety
- Role of script in the engine

For example, some AI script:

```
// Initially idle
self.behavior = idle

// Wait until hurt
yield_until(self.health < 1)

// Go on the attack
self.behavior = attack
```

Some things to consider

- Coroutines & cooperative multithreading
- **Events vs polling**
- Safety
- Role of script in the engine

With events:

```
on(health_message msg):
     if (self.health > 0.5 && msg.health <= 0.5)
          self.behavior = attack2
on(player_near msg):
     if (self.behavior == idle)
          self.behavior = attack1
```

With polling:

```
update():
     if (self.behavior == idle && player_near())
          self.behavior = attack1
     if (self.health <= 0.5 && last_health > 0.5)
          self.behavior = attack2
     last_health = self.health
```

Some things to consider

- Coroutines & cooperative multithreading
- Events vs polling
- **Safety**
- Role of script in the engine

- Permit memory allocation?
- Permit loops?
- Invalidating references to objects?

Some things to consider

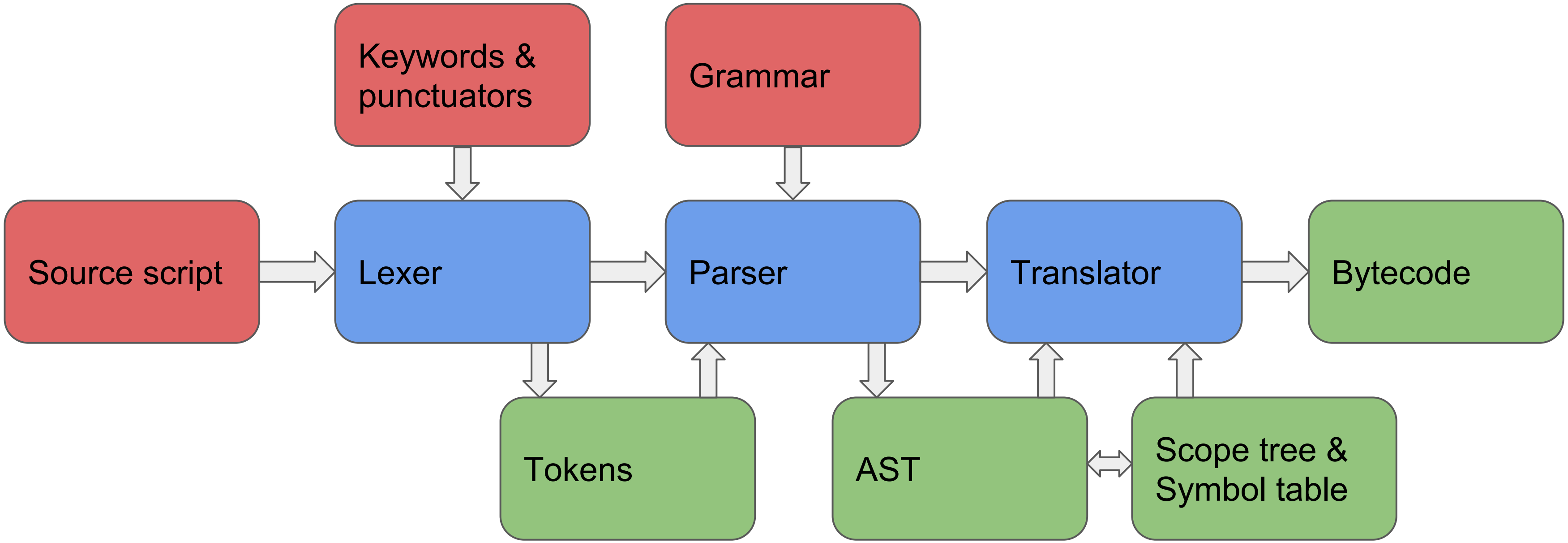
- Coroutines & cooperative multithreading
- Events vs polling
- Safety
- **Role of script in the engine**

Range of responsibility:

- Scripts are components on entities -- extend interfaces
- ...
- Scripts implement game state machine -- own main loop

Parsers and generators

A compiler for text based language...



Lexer

Our mission:

- Convert stream of characters into stream of tokens.
    - One token per language keyword. E.g.; `if`, `else`, `while`, `goto`, `int`, etc.
    - One token per language punctuator. E.g.; `++`, `==`, `(`, `<`, `>>`, etc.
    - One token per constant type. E.g.; strings, floats, integers
    - One token for some ‘identifier’

How we do it:

- LL(1) Recursive-Descent Lexer
    - LL(1) = read from Left to right, visit children Left to right, 1 character lookahead
    - Recursive = functions can call themselves
    - Descent = top-down

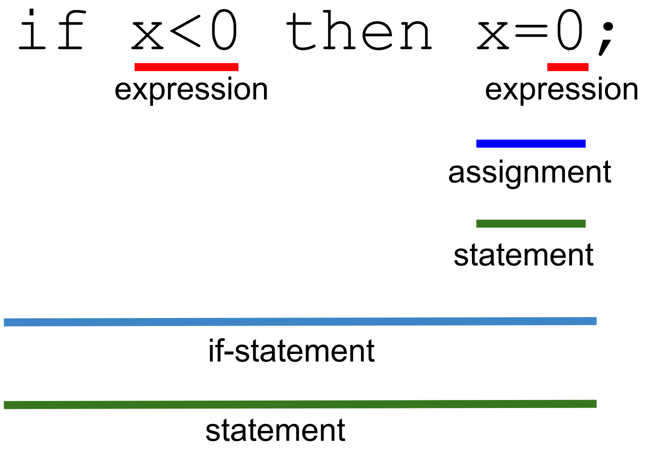
LL(1) Recursive-Descent Lexer

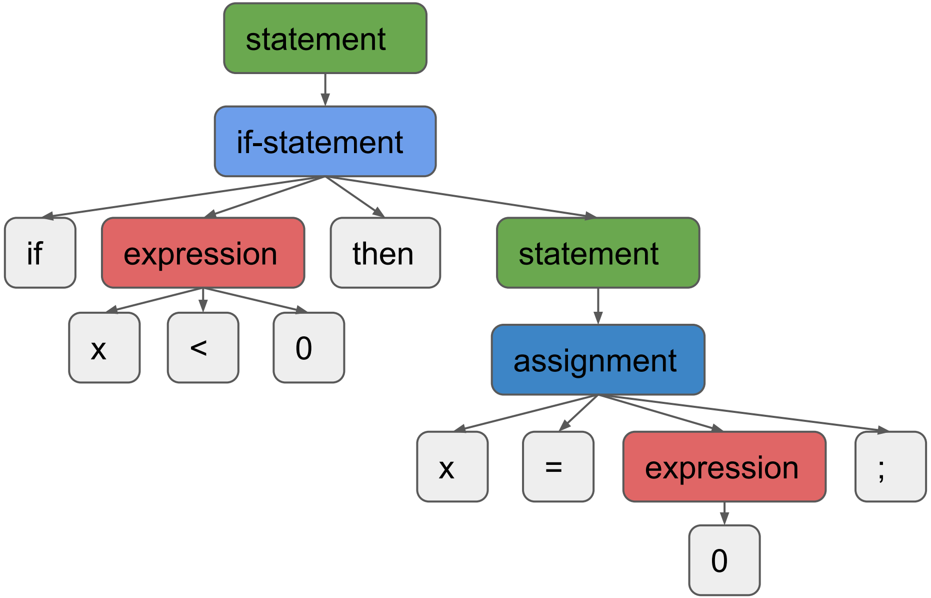
```
char* lex_get_next(
     const char* stream,
     token_t* out_token,
     const char** out_text,
     int* out_len)
{
     char* p = skip_whitespace(stream);

     if (!p) { *out_text = p; *out_token = token_eof; *out_len = 0; }
     else if (strcmp(p, “if”) == 0) { *out_text = p; *out_token = token_if; *out_len = 2; }
     else if (strcmp(p, “do”) == 0) { *out_text = p; *out_token = token_do; *out_len = 2; }
     else if (is_digit(p)) { *out_text = p; *out_token = token_integer; *out_len = xxx; }
     else if (is_nondigit(p)) { *out_text = p; *out_token = token_ident; *out_len = xxx; }
     else if (*p == ‘+’) { *out_text = p; *out_token = token_plus; *out_len = 1; }

     return p + *out_len;
}
```

Parser

Our mission:

- Recognize language phrases
- Convert a list of tokens into an abstract syntax tree (AST)





LL(k) Recursive-Descent Parser

```
void statement()
{
    ast_emit(node_statement);
    if parse_match(token_if) if_statement();
    else assignment();
}

void if_statement()
{
    ast_emit(node_if_statement);
    expression();
    parse_ensure(token_then);
    statement();
}
```

```
void expression() { /* stuff */ }

void assignment()
{
    ast_emit(node_assignment);
    parse_ensure(token_identifier);
    parse_ensure(token_equal);
    expression();
}
```

Types of ASTs

- Abstract syntax trees should not rigidly follow language structure
- Abstract syntax trees should be:
  - **Dense**; no unnecessary nodes
  - **Convenient**; easy to walk
  - **Meaningful**; emphasize operators and operations
- **Homogenous AST**; one node type with child list
  - Easy to walk, hard to work with
- **Normalized heterogeneous AST**; N node types with child list
  - Easy to walk, easier to work with
- **Irregular Heterogenous AST**; N node types
  - Hard to walk, easier to work with

Translation

Our mission:

- Convert AST into bytecode for virtual machine

Implementation:

- Repeated AST walks
- Build scope tree
- Gather symbols and resolve
- Emit bytecode

Walking the AST

Usually depth first traversal

- Sometimes looking for node of a particular type (e.g. all assignment nodes)
- Sometimes looking at structure of subtrees

As each node is visited, can perform action:

- On the way down
- On the way up

Symbol tables

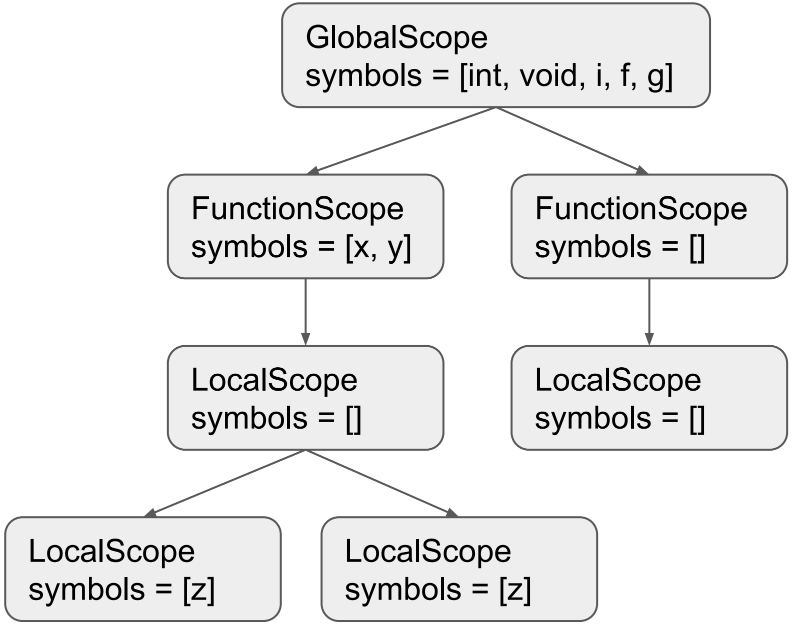
- Walk AST looking for identifier declarations
- Create entry in a table for each unique declaration
    - Variable symbols
    - Function symbols
    - Struct/class symbols
    - Enum symbols
    - Built-in symbols

Scope trees

- Walk AST looking for constructs that introduce scope
  - Global scope
  - Namespace scope
  - Class scope
  - Function scope
  - Local scope
- Create a tree of scopes
- Resolve symbol by searching up in the scope tree

Scope Tree Example

```
int i = 42;

int f(int x, int y)
{
    int i;
    { int z = x+y; i = z; }
    { int z = i+1; i = z; }
    return i;
}

void g()
{
    f(i, 2);
}
```



Generating bytecode

- Walk the AST and generate a stream of virtual machine instructions
- Use scope tree and symbol table to reserve virtual stack/registers
- Heavily dependent on virtual machine architecture...

Virtual Machines

A virtual CPU with a made up instruction set that runs our script.

- Easy to target (vs real CPU).
- Fast enough (pack instructions tight).

Two flavors:

- Stack based
- Register based

Virtual Machines

A virtual CPU with a made up instruction set that runs our script.

- Easy to target (vs real CPU).
- Fast enough (pack instructions tight).

Two flavors:

- **Stack based**
- Register based

```
iconst 1 ; Push integer 1 on stack
iconst 2 ; Push integer 2 on stack
iadd     ; Pop & add from stack
         ; Push result on stack
print    ; Pop & print top of stack
```

Virtual Machines

A virtual CPU with a made up instruction set that runs our script.

- Easy to target (vs real CPU).
- Fast enough (pack instructions tight).

Two flavors:

- Stack based
- **Register based**

```
iload r1, 1        ; Put int 1 in r1
iload r2, 2        ; Put int 2 in r2
iadd r3, r1, r2    ; r3 = r1 + r2
print r3           ; Print r3
```

The Big Switch of Death

```
switch (*bytecode++)
{
case opcode_iload:
     stack.push_int_from_mem(bytecode);
     bytecode += sizeof(int);
     break;
case opcode_iadd:
     stack.push_int(stack.pop_int() + stack.pop_int());
     break;
case opcode_print:
     switch (stack.top().type())
     {
     case type_int:
          vm_printf(“%d\n”, stack.pop_int());
          break;
     }
}
```

Integrating scripting with an engine

Our engine/script binding must allow:

- Engine to invoke script functions
- Script to invoke engine functions
- Script to use engine defined structures

And nice to have:

- Engine able to work with script structures
- Script is able to be updated on-the-fly

Integrating scripting with an engine

Our engine/script binding must allow:

- **Engine to invoke script functions**
- Script to invoke engine functions
- Script to use engine defined structures

And nice to have:

- Engine able to work with script structures
- Script is able to be updated on-the-fly

In script:

```
script_add(x, y):
      return x + y;
```

In engine:

```
int engine_add(int x, int y)
{
      vm_func_t func = vm_getfunc("script_add");
      vm_call_t call = vm_new_call(func);
      call.push_arg_int(x);
      call.push_arg_int(y);
      vm_run(call);
      return call.pop_return_int();
}
```

Integrating scripting with an engine

Our engine/script binding must allow:

- Engine to invoke script functions
- **Script to invoke engine functions**
- Script to use engine defined structures

And nice to have:

- Engine able to work with script structures
- Script is able to be updated on-the-fly

```
int engine_add(int x, int y)
{
    return x + y;
}

void engine_add_wrapper(vm_call_t* call)
{
    int result = engine_add(
        call->pop_arg_int(),
        call->pop_arg_int());
    call->push_return_int(result);
}

void vm_init()
{
    vm_register_func(engine_add_wrapper);
}
```

Integrating scripting with an engine

Our engine/script binding must allow:

- Engine to invoke script functions
- Script to invoke engine functions
- **Script to use engine defined structures**

And nice to have:

- Engine able to work with script structures
- Script is able to be updated on-the-fly

```
struct foo
{
	int x;
	float y;
};

void vm_init()
{
	vm_struct_t s = vm_new_struct(“foo”);
	s.add_field(“x”, type_int);
	s.add_field(“y”, type_float);
}
```

Integrating scripting with an engine

Our engine/script binding must allow:

- Engine to invoke script functions
- Script to invoke engine functions
- Script to use engine defined structures

And nice to have:

- **Engine able to work with script structures**
- Script is able to be updated on-the-fly

A dynamic type system, engine-side? Ouch.

Integrating scripting with an engine

Our engine/script binding must allow:

- Engine to invoke script functions
- Script to invoke engine functions
- Script to use engine defined structures

And nice to have:

- Engine able to work with script structures
- **Script is able to be updated on-the-fly**

Left as an exercise to the listener.

Final thoughts on script/engine integration

- Binding script to engine is lots of boilerplate code
- Some engines have custom header file preprocessors
  - Generate script binding boilerplate automagically
- Some engines use a mix of C preprocessor macros and typing
- Some engines just do it all by hand
- Depends on size of engine, extent of script usage, etc.

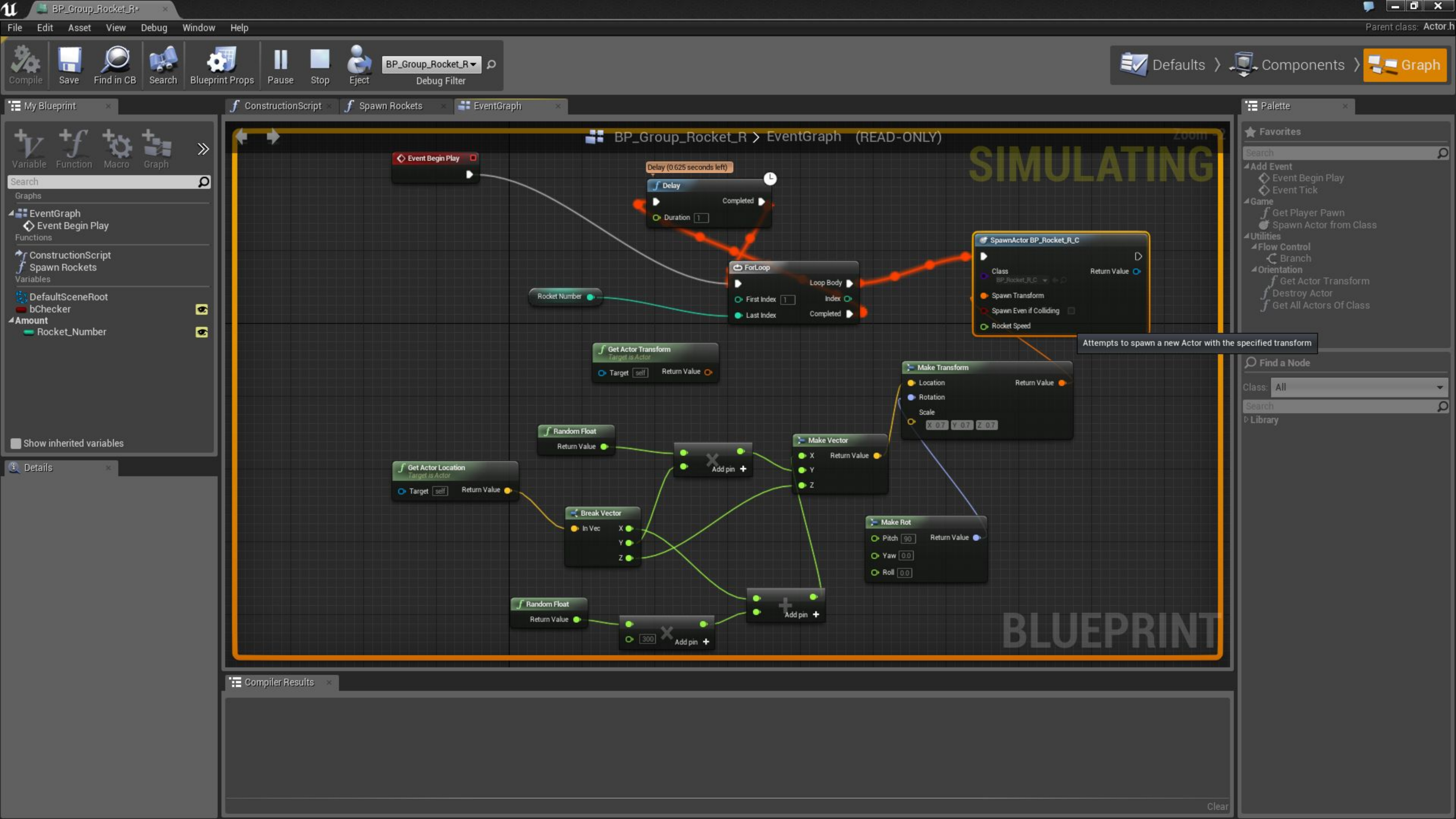
Visual Script

For some:

- Typing can be intimidating
- Syntax is a big challenge
- Knowing all possible valid actions at any point is helpful
- Exploration is preferred (over understanding)

For those folks, we have visual languages. Visual languages take many forms:

- Graphs (nodes and edges)
- Fill-the-blank statement composition
- Tiles (e.g. Scratch)


BP_Group_Rocket_R*
File
Edit
Asset
View
Debug
Window
Help
Parent class: Actor
Compile
Save
Find in CB
Search
Blueprint Props
Pause
Stop
Eject
BP_Group_Rocket_R
Debug Filter
Defaults
Components
Graph
My Blueprint
Variable
Function
Macro
Graph
Search
Graphs
EventGraph
Event Begin Play
Functions
ConstructionScript
Spawn Rockets
Variables
DefaultSceneRoot
bChecker
Amount
Rocket_Number
Show inherited variables
Details
ConstructionScript
Spawn Rockets
EventGraph
BP_Group_Rocket_R > EventGraph (READ-ONLY)
SIMULATING
Event Begin Play
Delay (0.625 seconds left)
Delay
Completed
Duration
ForLoop
Loop Body
First Index
Index
Last Index
Completed
Rocket Number
SpawnActor BP_Rocket_R_C
Class
Return Value
Spawn Transform
Spawn Even if Colliding
Rocket Speed
Attempts to spawn a new Actor with the specified transform
Get Actor Transform
Target is Actor
Target
Return Value
Make Transform
Location
Rotation
Scale
Random Float
Return Value
Add pin
Make Vector
Get Actor Location
Target is Actor
Break Vector
In Vec
Make Rot
Pitch
Yaw
Roll
Random Float
Add pin
Add pin
BLUEPRINT
Compiler Results
Clear
Palette
Favorites
Search
Add Event
Event Begin Play
Event Tick
Game
Get Player Pawn
Spawn Actor from Class
Utilities
Flow Control
Branch
Orientation
Get Actor Transform
Destroy Actor
Get All Actors Of Class
Find a Node
Class:
All
Search
Library

The ______ are eating bread.

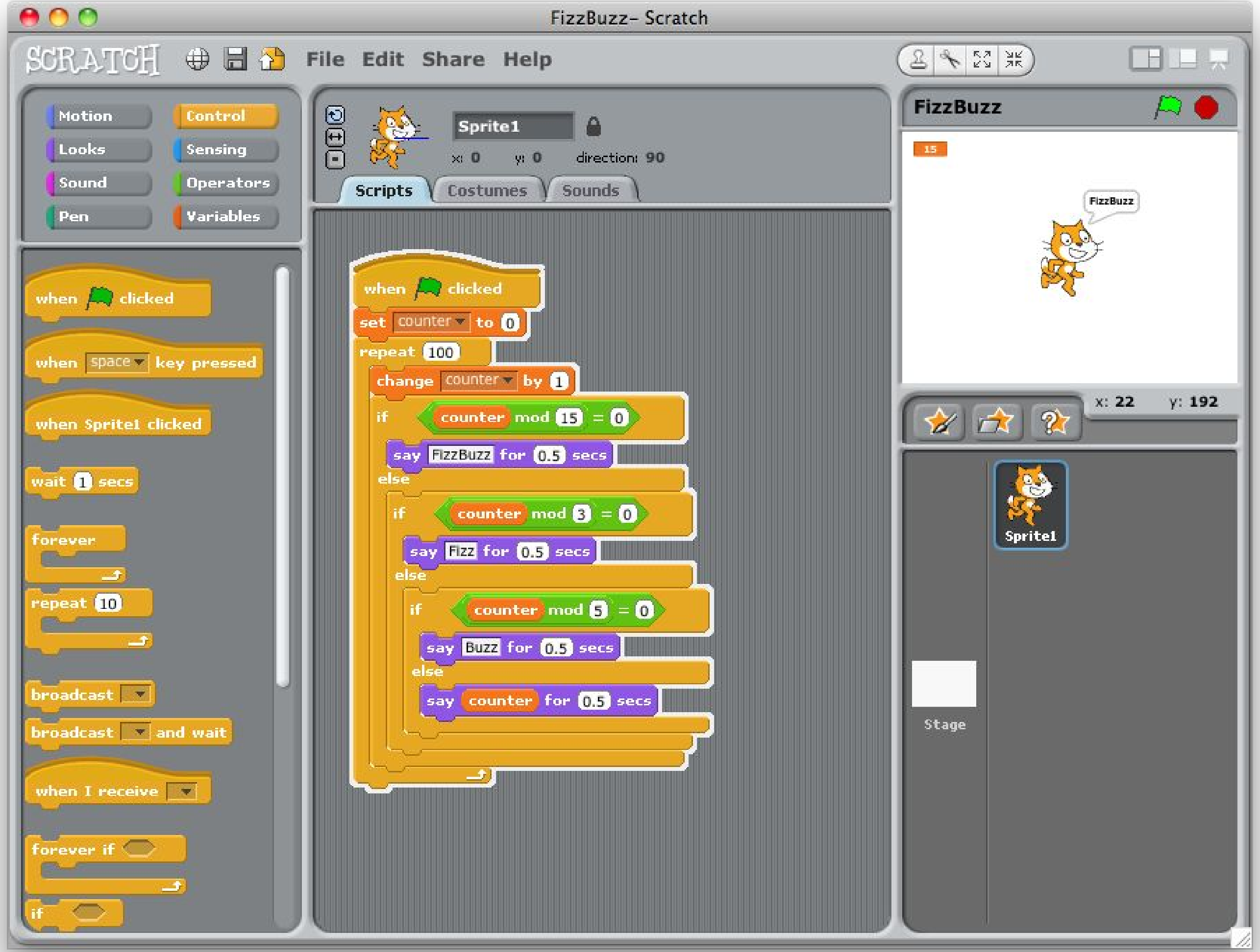
Let's run and jump ______ !

______ are you drinking?

Some of ______ friends are in the park.

Let's play basketball! Are you ______ ?

outside	my
What	robot
try	practice
children	girl
sandwich	ready


FizzBuzz- Scratch
File Edit Share Help
Motion
Control
Looks
Sensing
Sound
Operators
Pen
Variables
Sprite1
x: 0 y: 0 direction: 90
Scripts
Costumes
Sounds
when clicked
set counter to 0
repeat 100
change counter by 1
if counter mod 15 = 0
say FizzBuzz for 0.5 secs
else
if counter mod 3 = 0
say Fizz for 0.5 secs
else
if counter mod 5 = 0
say Buzz for 0.5 secs
else
say counter for 0.5 secs
FizzBuzz
15
FizzBuzz
x: 22 y: 192
Sprite1
Stage

Visual script backend

Assume the frontend exists. Options for backend:

- Visuals map to script code in a text file. Just compile and run the text.
- Visuals map to objects in a high level state machine. Make those objects.

That’s all we will say about that.

Case Study: Lua

- Handwritten recursive descent parser
  - No AST, just emit bytecode during parse
- Register based VM
- Instruction width is 32 bits

Lua 5.1 Instruction Set

MOVE Copy a value between registers
LOADK Load a constant into a register
LOADBOOL Load a boolean into a register
LOADNIL Load nil values into a range of registers
GETUPVAL Read an upvalue into a register
GETGLOBAL Read a global variable into a register
GETTABLE Read a table element into a register
SETGLOBAL Write a register value into a global variable
SETUPVAL Write a register value into an upvalue
SETTABLE Write a register value into a table element
NEWTABLE Create a new table
SELF Prepare an object method for calling
ADD Addition operator
SUB Subtraction operator
MUL Multiplication operator
DIV Division operator
MOD Modulus (remainder) operator
POW Exponentiation operator
UNM Unary minus operator

NOT Logical NOT operator
LEN Length operator
CONCAT Concatenate a range of registers
JMP Unconditional jump
EQ Equality test
LT Less than test
LE Less than or equal to test
TEST Boolean test, with conditional jump
TESTSET Boolean test, with conditional jump and assignment
CALL Call a closure
TAILCALL Perform a tail call
RETURN Return from function call
FORLOOP Iterate a numeric for loop
FORPREP Initialization for a numeric for loop
TFORLOOP Iterate a generic for loop
SETLIST Set a range of array elements for a table
CLOSE Close a range of locals being used as upvalues
CLOSURE Create a closure of a function prototype
VARARG Assign vararg function arguments to registers

Lua likes Tables

In Lua, lots of things are tables (associative arrays):

- Vectors
- Maps
- Objects
- Modules…

For example:

```
Point = { x=5, y=10 }
print(Point["x"], Point["y"])
```

Let’s look at our homework assignment...

Summary

- Features of scripting languages in games
- Parser and generator concepts
- Virtual machines
- Integrating scripting into an engine
- Visual script
- Lua

End lecture

A lot of the content in this lecture was adapted from:

Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Language. Terence Parr.

Start writing your own language, compiler and VM:

- *http://llvm.org/docs/tutorial/index.html*
- *http://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinyc.c*