How to create a programming language

About compilers and runtime environments

How to create a programming language

- Why I did this?
- How to start?
- Compiler implementation
- Runtime implementation
- Optimizations

How to create a programming language

- Why I did this?
- How to start?
- Compiler implementation
- Runtime implementation
- Optimizations

Why I did this?

- First motivation:
  - Multiplatform programming language
  - Remains stable over time
- Second motivation:
  - Big challenge
  - Never accomplished a project like this
  - Love programming

How to create a programming language

- Why I did this?
- How to start?
- Compiler implementation
- Runtime implementation
- Optimizations

How to start?

Strategies

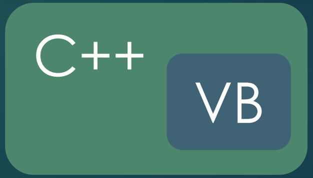
- Interpreter - Code is parsed, analysed and executed at same step.
  Examples: R / Perl / Bash / QBasic
- Transpiler – Code is transformed into another language for which there is a compiler.
  Examples: TypeScript(JavaScript) / Dart(JavaScript)
- VM Based – Code is compiled into bytecode, then executed by a software CPU (VM).
  Examples: Python / Ruby / JavaScript / Php / Lua / Erlang
- VM Based with JIT - Same as VM Based, but bytecode is compiled into CPU native code, then executed.
  Examples: .Net (C#,VB…) / Java / Scala / Kotling / LuaJIT / PyPy
- Llvm framework – Code is compiled into Llvm assembler, then Llvm tools produce CPU native code.
  Examples: Clang (C C++ Objective-C Objective-C++) / Rust / Haskell
- From scratch – Code is compiled into CPU native code directly
  Examples: gcc (C C++ Objective-C Objective-C++ Fortran Go) / TurboC / TurboPascal / FreeBasic / Haskell

How to start?

Bootstrapping vs. New Language

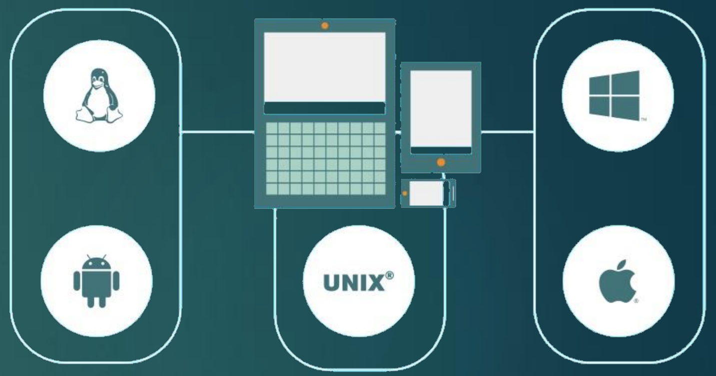
- Bootstrapping
  - Scenario on which we develop a compiler in the same language than the one being created → Compiler can compile itself.
  - Pros: When we finish a substantial part of testing is already done.
  - Cons: Only possible if a compiler for the language already exists.
    Can only create a subset/superset of an existing language.

- New language
  - Scenario on which we develop a compiler on a different language than the one being created.
  - Pros: Total freedom to design new language as desired.
  - Cons: Harder testing. We have to test all language features one by one.



How to start?

Target OS and development tools



- Multiplatform
  If the language is to be multiplatform choose a development IDE that is also multiplatform (will be much easier to port).
- Performance
  Critical if language is executed on a VM.
  Use a development IDE that produces fast code (specially for the VM)
- I/O Libraries
  IDE needs to have available I/O libraries you plan to use (graphics, audio, etc.)
  Multiplatform: Libraries also multiplatform, or alternative must exist on targets.
- Bootstrapping
  Choose an existing IDE that uses the same language.

How to start?

My choice of development tools

- Development language → C++
  - Pros: C & C++ produce fastest code possible
    C & C++ are the most portable languages (C even more)
    C & C++ allow low level memory manipulations (necessary for VM)
  - Cons: C & C++ lack automatic memory management
    (however on C++ it can be achieved using constructors/destructors)
- Development IDE → g++ / Sublime Text
  Why?: Very lightweight, g++ and Sublime Text are multiplatform, lots of libraries available.
- Tools
  - Linux: g++ / Sublime Text / kgdb / gprof / perf / bash / python / SDL / git
  - Windows: MinGW64(g++ gdb gprof) / Sublime Text / .bat / python / SDL / git

How to start?

Design the language

- Define language concepts
  - Comments
  - Sentence delimitation (join, split)
  - Built-in data types (char, int, float, string, …)
  - Available data structures (classes, arrays, lists, …)
  - Literal values (integers:4567, float:4.5E3, chars:'A', strings:"hello", …)
  - Operators (mathematical, comparison, logical, bitwise, assignment,…)
  - Functions (parameter passing by reference / by value)
  - Control flow statements (if, else, for, do, while, switch, …)
  - Modules and libraries

- Define syntax
  - Document all language sentences

How to start?

Syntax definition examples

- Examples:
  - import <string> as <id> → Import library (python style)
  - include <string> as <id> → Include file (C style)
  - const <typespec> <id> = <expr> → Define constant
  - <typespec> <id> [= <expr>] → Declare variable
  - systemcall<n> <typespec> <id>([ref] <id>,[ref] <id>, ...) → Declare system call (interface to runtime)
  - <typespec> <id>([ref] <id>,[ref] <id>, ...) → Declare function
  - func <typespec> <id>([ref] <id>,[ref] <id>, ...): (...) :func → Define function
  - main: (...) :main → Define program entry point
  - return [<expr>] → Return from function call
  - if(<expr>): (...) elif(<expr>): (...) else: (...) :if → If / elseif / else statement
  - while(<expr>): (...) :while → While loop (evaluation at beginning)
  - do: (...) :loop(<expr>) → Do loop (evaluation at end)
  - for(<expr> if <expr> do <expr>): (...) :for → For loop
  - switch(<expr>): when(<expr>): (...) ... default: (...) :switch → Switch sentence
  - break → Exit from loop or switch case
  - continue → Continue to next iteration
- Legend:
  - <string> Double quotes delimited string (i.e.: “mystring”)
  - <id> Identifier (i.e.: myvariable, myfunction)
  - <typespec> Type specification (int, float, string, int[], string[10])
  - <expr> Expression
  - … Repetition
  - (…) One or more sentences
  - [ ] Optional

How to start?

Think about VM runtime design (in asm!)

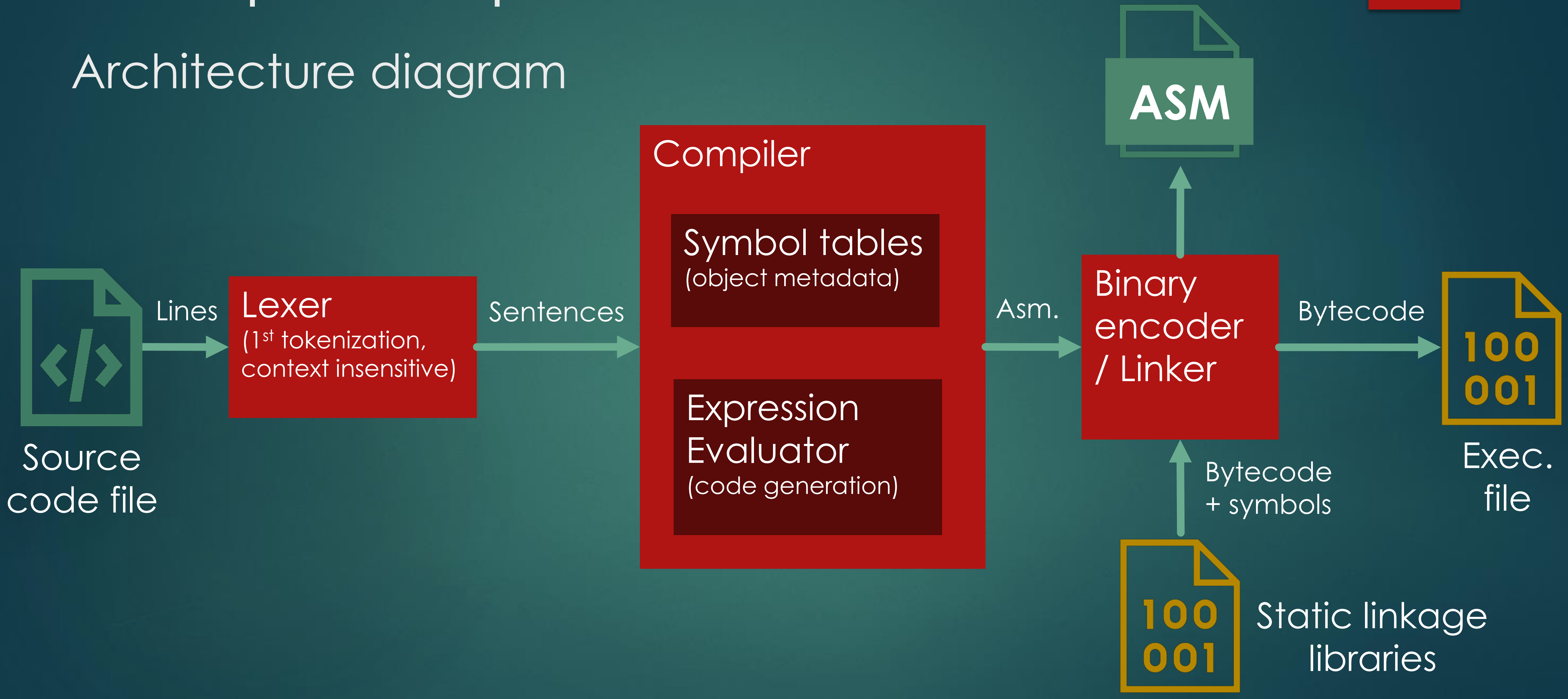

```
(compare_datetime)          FUNCTION

.section DECL

<$result>                   PARM REFERENCE                      ;Address=<0000000000000000h>
<dt1>                       PARM CONST REFERENCE                ;Address=<0000000000000010h>
<dt2>                       PARM CONST REFERENCE                ;Address=<0000000000000020h>
<$Ref000t>                  VAR REFERENCE                       ;Address=<0000000000000030h>
<$Ref001t>                  VAR REFERENCE                       ;Address=<0000000000000040h>
<$Bol000t>                  VAR BOOLEAN                         ;Address=<0000000000000050h>
<$Bol001t>                  VAR BOOLEAN                         ;Address=<0000000000000051h>

.section CODE
                            ;Reserve function stack size
                            STACK (L)82                         ;[000000000000B1AAh] 00BB 80 0000000000000052

                            ;return (dt1._dt==dt2._dt && dt1._tm==dt2._tm?true:false)
                            REFOF <$Ref000t>,*<dt1>,(L)0        ;[000000000000B1B5h] 00AE 18 0000000000000030 0000000000000010 0000000000000000
                            REFOF <$Ref001t>,*<dt2>,(L)0        ;[000000000000B1D0h] 00AE 18 0000000000000040 0000000000000020 0000000000000000
                            EQUi <$Bol000t>,*<$Ref000t>,*<$Ref001t>   ;[000000000000B1EBh] 006B 14 0000000000000050 0000000000000030 0000000000000040
                            REFOF <$Ref000t>,*<dt1>,(L)4        ;[000000000000B206h] 00AE 18 0000000000000030 0000000000000010 0000000000000004
                            REFOF <$Ref001t>,*<dt2>,(L)4        ;[000000000000B221h] 00AE 18 0000000000000040 0000000000000020 0000000000000004
                            EQUl <$Bol001t>,*<$Ref000t>,*<$Ref001t>   ;[000000000000B23Ch] 006C 14 0000000000000051 0000000000000030 0000000000000040
                            LAND <$Bol000t>,<$Bol000t>,<$Bol001t>     ;[000000000000B257h] 0032 00 0000000000000050 0000000000000050 0000000000000051
                            JMPFL <$Bol000t>,{CN0001510000F}    ;[000000000000B272h] 0135 20 0000000000000050 {CN0001510000F}
                            MVb <$Bol000t>,(B)true              ;[000000000000B285h] 0076 20 0000000000000050 01
                            JMP {CN0001510000E}                 ;[000000000000B291h] 0136 80 {CN0001510000E}
{CN0001510000F}:            MVb <$Bol000t>,(B)false             ;[000000000000B29Ch] 0076 20 0000000000000050 00
{CN0001510000E}:            MVb *<$result>,<$Bol000t>           ;[000000000000B2A8h] 0076 40 0000000000000000 0000000000000050
                            RET                                 ;[000000000000B2BBh] 00E6

                            END
```

How to create a programming language

- Why I did this?
- How to start?
- Compiler implementation
- Runtime implementation
- Optimizations

Compiler implementation

Architecture diagram



Compiler implementation

Lexer

- It is first stage when compiling code and it does:
  - Filter comments and blank lines
  - Identify sentences
  - Split sentences into tokens
    (1st tokenization, context insensitive)

- Token = Source code atomic unit
  - Keywords (if, for, while, etc.)
  - Literal values (numbers, chars, strings)
  - Identifiers (variable names, function names, etc.)
  - Operators (+ - * / = etc.)
  - Punctuators ({ } [ ] ( ) , : ; etc.)

Compiler implementation

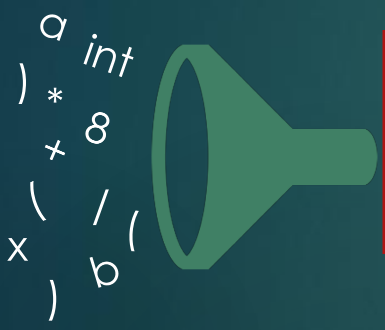
Symbol tables

- Store all metadata for objects & functions in the source code:
  - Modules (as one source can have included files or linked libraries)
  - Type definitions (basic type, scalar / array, references)
  - Fields for classes and enumerated types
  - Functions and parameters (including class methods)
  - Search indexes for fast identifier lookup (sorted insertion & binary search)

- Scope stack (It also keeps track of scopes):
  - Module public (objects visible outside of module)
  - Module private (objects not visible outside of module)
  - Local (inside a function)
  - Inner local (function inside a function)

Compiler implementation

Expression evaluator



Parser
(2nd tokenization
(context sensitive)

int x = (a + 1) / (b * 8)
Context identified

RPN conversion
(shunting yard)

x a 1 + b 8 * / =
RPN form

Compute evaluation

Operand stack

ADD r1, a, 1
MUL r2, b, 8
DIV x, r1, r2
Assembler code

Compiler implementation

Expression evaluation example: RPN(x a 1 + b 8 * / =)

Step	1	2	3	4	5	6	7	8	9
Token	x	a	1	+	b	8	*	/	=
Operand stack	x	x, a	x, a, 1	x, r1	x, r1, b	x, r1, b, 8	x, r1, r2	x, r3	
Emitted code				ADD r1,a,1			MUL r2,b,8	DIV r3,r1,r2	MOV x,r3

Optimization	**DIV x,r1,r2**

x a 1 + b 8 * / =

ADD r1, a, 1
MUL r2, b, 8
DIV x, r1, r2

Compiler implementation

RPN Conversion

- Shunting Yard algorithm is used
  - Invented by mathematician, Edsger Dijkstra, 1961 (ALGOL lang.)
  - Can be used to produce RPN expression or Abstract Syntax Tree
- Operator precedence
  - Every operator has a precedence (priority)
  - Used to resolve order of evaluation when parenthesys are not given
  - x / z + y * 3 → Equivalent to: (x / z) + (y * 3)
    (operators * and / have higher precedence than +)
- Operator associativity
  - Solves order of evaluation when operators have same precedence
  - Left-associative: Groups operations on left side
  - Right-associative: Groups operations on the ride

Left associative:
7 + 1 – 4 + 5 → (((7 + 1) - 4) + 5)

Arithmetic operators are left associative

Right associative:
a = b = c = d → a = (b = (c = d))

Assignment operator is right associative

Compiler implementation

Several things that happen during expression evaluation (1)

- Automatic data type conversions:
  - Safe: int i = (char)32 + (int)12384 → int i = **(int)32** + (int)12384
  - Unsafe: char c = (char)32 + (int)12384 → char c = (char)32 + **(char)96**

  (*) 12384 = 0x00003060 → 0x60 = 96
- On-the-fly calculations:
  - Compiler can calculate intermediate results for operations with literal values and constants to emit less code
  - const int K = 4; int x = a + (32/K) + b / (2*1000*K)
  - Becomes → **int x = a + 8 + b / 8000**
- Operator overloads:
  - When evaluating an operator check if there is an operator overload function for the same arguments that can be called instead

Compiler implementation

Several things that happen during expression evaluation (2)

- L-Value checkings:
  - R-Values:
    - On the right of assignment (=)
    - Undefined memory address, not reachable from source
    - Literal values or intermediate calculations
  - L-Value
    - On the left side of assignment (=)
    - Defined memory address, reachable from source
    - Variables, class members, array elements, etc.
  - Certain operators require that one of the operand is an L-Value (assignment)

    ✗ a + 1 = c + 32 * k ✗ "hello" = str1 + "." + str2 ✓ a[i] = b[j] + 1
  - Parameters passed by reference require L-Value

    int myfunction(ref int x) →
    ✗ myfunction(abs(i))
    ✗ myfunction(i+1)
    ✓ myfunction(i)

Compiler implementation

Several things that happen during expression evaluation (3)

- Function / method calls:
  - Parameters can be passed by value or reference
  - By value → A copy of the value is moved to the stack
  - By reference → A pointer to a L-Value is moved to the stack
  - Function result is always passed by reference

- Array / string indexing:
  - a[n] →Results in pointer to n^{th} element within the array / string

- Pointers and indirections:
  - Compiler works with expressions that are naturally a pointer (by ref. parameters, indexing…) to generate indirections and pointer addresses automatically
  - No need to explicitly have indirection and address-of operators in the code!

C / C++

```
void myfunction(int *result){
   *result=1;
}
myfunction(&x);
```

C#

```
void myfunction(ref int result){
  result=1;
}
myfunction(ref x);
```

Compiler implementation

Other compiler tasks (1)

- Forward function call resolution:
  - Call to functions in the code can happen before function is actually defined → **Function call address cannot be resolved !!**
  - Compiler tracks and stores all unresolved function calls
  - Before finishing compilation all function calls are resolved (using symbol tables)

C / C++ Example:

```
int myfunction(void);          → Function declaration (compiler becomes aware of function)
int main(void){
  myfunction();                →Function is called (still address is now known)
}
int myfunction(void){          → Function is defined (code address is defined and stored on symbol tables)
  printf("Hello!";
}
```

Compiler implementation

Other compiler tasks (2)

- Jump address resolution:
  - Control flow statements produce jump instructions to go to other code block
  - When flow produces a forward jump it cannot be resolved
  - Compiler tracks and stores all unresolved jumps for later resolution

Source Code:

```
int i
for(i=0 if i<5 do i++):
  if(i==0):
    con.print("Hello1")
  else:
    con.print("Hello2")
  :if
:for
```

```
                       ;for(i=0 if i<5 do i++):
                       MVi <i>,(I)0                          ;[00062h] 0079 20 00000000 00000000
{000001for-beg}:       LESi <$Bo1000t>,<i>,(I)5              ;[00071h] 004F 08 00000004 00000000 00000005
                       JMPFL <$Bo1000t>,{000001for-exit}     ;[00088h] 0137 20 00000004 {000001for-exit}       → Forward jump

                       ;if(i==0):
                       EQUi <$Bo1000t>,<i>,(I)0              ;[0009Bh] 006B 08 00000004 00000000 00000000
                       JMPFL <$Bo1000t>,{000002ifs-cond1}    ;[000B2h] 0137 20 00000004 {000002ifs-cond1}      → Forward jump

                       ;con.print("Hello1")
                       REFPU [STR(0000006Ch)]                ;[000C5h] 00D8 00 8000006C
                       SCALL (I)9                            ;[000D0h] 00EB 80 00000009

                       ;else:
                       JMP {000002ifs-exit}                  ;[000D7h] 0138 80 {000002ifs-exit}                → Forward jump

                       ;con.print("Hello2")
{000002ifs-cond1}:     REFPU [STR(00000070h)]                ;[000E2h] 00D8 00 80000070
                       SCALL (I)9                            ;[000EDh] 00EB 80 00000009

                       ;:for
{000002ifs-exit}:      PINCi <$Int000t>,<i>                  ;[000F4h] 0029 00 00000005 00000000
                       JMP {000001for-beg}                   ;[00107h] 0138 80 {000001for-beg}                 → Backward jump
{000001for-exit}:      RET                                   ;[00112h] 00E8
```

Compiler implementation

Other compiler tasks (3)

- Function return values:
  - Compiler has to check if function that returns a value has return statement
  - A failsafe check must analyse all function flow paths to check if they end in return statement
- Uninitialized variables
  - Before using a variable compiler can check if it was initialized already
  - A failsafe check must analyse if all flow paths before variable is used assign a value to the variable
- Unused variables
  - After closing a variable scope compiler can check for unused variables
  - Unused: Variable that does not happen as operand in any expression
- Error / warning message reporting:
  - Compiler must send comprehensive error / warning messages to the developer
  - All messages should be related to a file, line number and column → Linters!!
  - Compilation goes on discarding or undoing sentences with syntax errors

Compiler implementation

Binary file generation – Executable files

Header
Global buffer
Code buffer
Debug symbols

- Header
  - Compiler version, build date & time, etc.
  - Sizes of the buffers (global data, code, debug symbols)
- Global buffer
  - Initial values of all program variables with global scope (public & private)
- Code buffer
  - Compiled code, binary representation of assembler instructions
- Debug symbols
  - Compiled code only contains addresses
  - Variable names or function names do not appear in compiled code
  - Debug symbols help to identify addresses in the code and assign a meaningful name (related to the source code)

Compiler implementation

Binary file generation – Static libraries

- Header
- Global buffer
- Code buffer
- Debug symbols
- Relocation table
- Public symbols

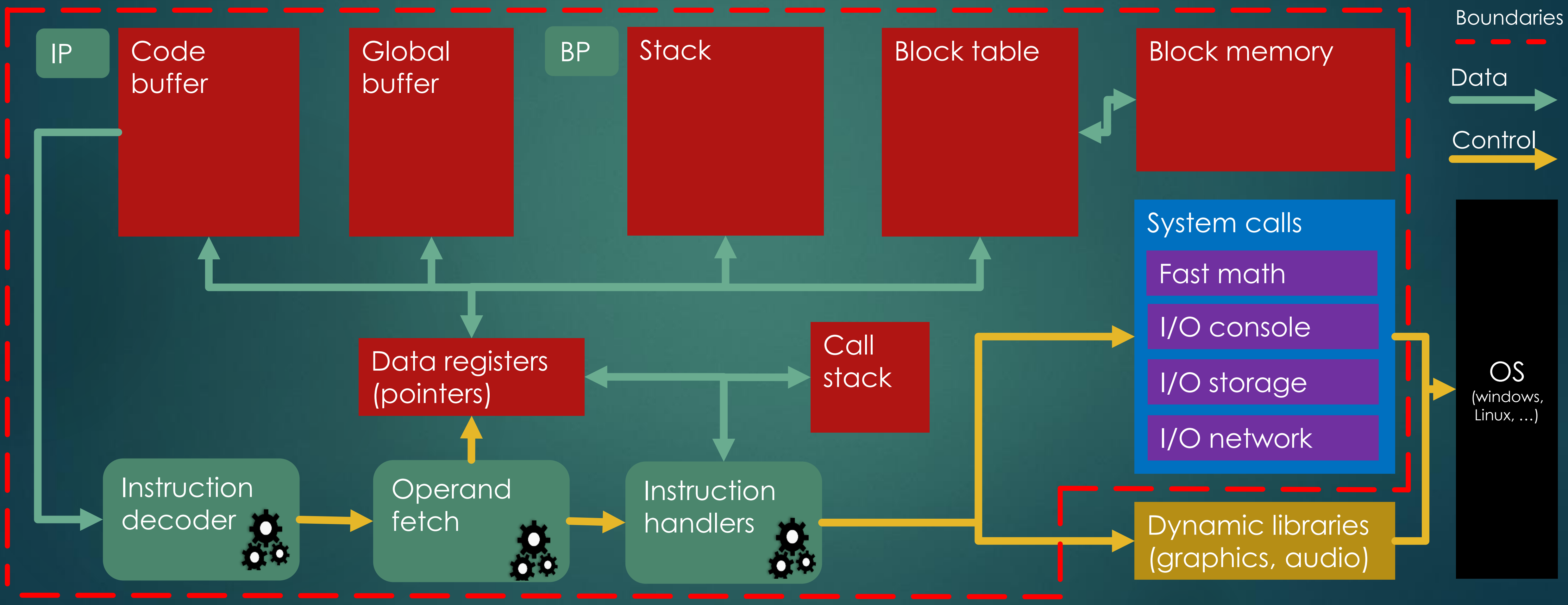
- Header, Global buffer, Code buffer, Debug symbols
  → Same as for executable files
- Relocation table
  - Addresses of global variables and functions are coded in binary code
  - When linking libraries, code buffer is relocated to a different position to when it was compiled
  - All references in the binary code to global variable addresses and functions have to be changed!!
- Public symbols
  - Information about the public objects that library exposes
  - Compiler uses this information to update his own symbol tables

How to create a programming language

- Why I did this?
- How to start?
- Compiler implementation
- Runtime implementation
- Optimizations

Runtime implementation

Virtual machine architecture diagram



Runtime implementation

Instruction set

- Typical CPU operations:
  - Arithmetic: add, sub, mul, div, mod, inc, dev
  - Logical: not, and, or
  - Bitwise: not, and, or, xor, shr, shl
  - Comparison: equ, neq, les, gre, leq, geq
  - Assignment: mov, mvad, mvsu, mvmu, mvdi, mvmo, mvsl, mvsh, mvan, mvor, mvxo
  - Control flow: jmp, jmpfl, jmptr,
  - Functions: push, pop, call, ret

- Specific for VM:
  - Memory: Pointer arithmetics, memory block copy
  - Arrays: Array operations (memory management, element addressing,…)
  - Strings: String operations (search, replace, upper, lower, split, concatenate,…)
  - Conversions: Between all numeric/strings
  - Interfaces: System calls (host OS) & .dll/.so calls

Runtime implementation

Instruction format & addressing modes

CCCC MM AAAA... BBBB... CCCC... DDDD...

Arguments, 0 to 4 (variable length)

Addressing modes of arguments:

- Bits 0-1: Argument 1
- Bits 2-3: Argument 2
- Bits 4-5: Argument 3
- Bits 6-7: Argument 4

- 00: Literal value (on instruction)
- 01: Global / Stack address
- 10: Reference indirection

Instruction code (16-bit)

Runtime implementation

Stack vs. Registers vs. Addresses in VM operations

	Stack based operations	Register based operations	Address based operations
Source code	`Gross = Net + Tax`	`Gross = Net + Tax`	`Gross = Net + Tax`

Assembler	`PUSH [Net]`
	`PUSH [Tax]`
	`ADD`
	`POP  [Gross]`
		`MOV R1,[Net]`
		`MOV R2,[Tax]`
		`ADD R3,R1,R2`
		`MOV [Gross],R3`
			`ADD [Gross],[Net],[Tax]`

Used in	Java .Net runtime Python Ruby	Lua Dalvik (Android)	Erlang Elixir

How to create a programming language

- Why I did this?
- How to start?
- Compiler implementation
- Runtime implementation
- Optimizations

Optimizations

Why important?

- Compiler speed:
  - Impact on development time (when compiling code & linting code)
  - Slow and heavy development environment is frustrating!
- Runtime speed:
  - Not important but **absolutely critical**!!
  - Entire language usability is impacted by poor runtime performance

Optimizations

Code optimizations

- Enable optimization options in compiler:
  - gcc / g++ → Command line options –O1, -O2, -O3
  - c# → Command line option -optimize
    Build configuration <Optimize>true</Optimize>
  - msvc → Command line options /Ox (several)

- Find parts to optimize:
  - Use code profiling tools to find code parts that can be optimized (g++: gprof / perf)
  - Optimize the parts in the code with highest impact on performance:
    → Code that runs slow but that is also called high amount of times!

Optimizations

Code optimizations

- Avoid doing things twice:
  - If possible reuse result of a calculation instead of repeating calculation.
  - For languages that can control how parameters are passed to functions
    → It is more efficient to pass a constant reference to a big object than a copy
- Loops:
  - Analyse start point and direction (forward / reverse) to complete as less steps as possible.
  - If goal is to find a specific record → Exit loop immediately, never continue till end
- If we have to search element within array:
  - Use sorted insertion to add elements to the array
  - Use binary search to find elements
- In case we have to copy memory:
  - Use wide data type to copy data
  - Copy one byte a time vs Copy int64 at a time → Much less iterations!
- Function calls vs macros (or inline functions):
  - Function calls imply certain overhead (create / destroy stack frame)
  - Macros and inline functions will increase code size but do not have overhead

Optimizations

The evil goto statement

<table>
<tr>
<td>If & else
Time to fetch not constant</td>
<td>Switch
Time to fetch not constant</td>
<td>Function pointers
Time to fetch constant!</td>
<td>Code address pointer
Time to fetch constant!
No function call overhead!</td>
</tr>
<tr>
<td><pre>(...)
if(InstCode==0){
  //Do add instruction
}
else if(InstCode==1){
  //Do sub instruction
}
else if(InstCode==2){
  //Do mul instruction
}
else if(InstCode==3){
  //Do div instruction
}
(...)</pre></td>
<td><pre>(...)
Switch(InstCode){
 case 0:
    //Do add instruction
    break;
 case 1:
    //Do sub instruction
    break;
 case 2:
    //Do mul instruction
    break;
 case 3:
    //Do div instruction
    break;
}
(...)</pre></td>
<td><pre>void add();
void sub();
void mul();
void div();

void (*p[4])()=
{&add,&sub,&mul,&div};

(...)
(*p[InstCode])();
(...)</pre></td>
<td><pre>const void *Address[4]={
&&add,&&sub,&&mul,&&div};

add:;
//Do add instruction
goto end;
sub:;
//Do sub instruction
goto end;
mul:;
//Do mul instruction
goto end;
div:;
//Do div instruction
goto end;

(...)
goto *Address[InstCode];
end:;
(...)</pre></td>
</tr>
</table>

Thanks for your attention!