The Basics of Programming Language **Design**

Jace J. Parks | Grade 12
Computer Science

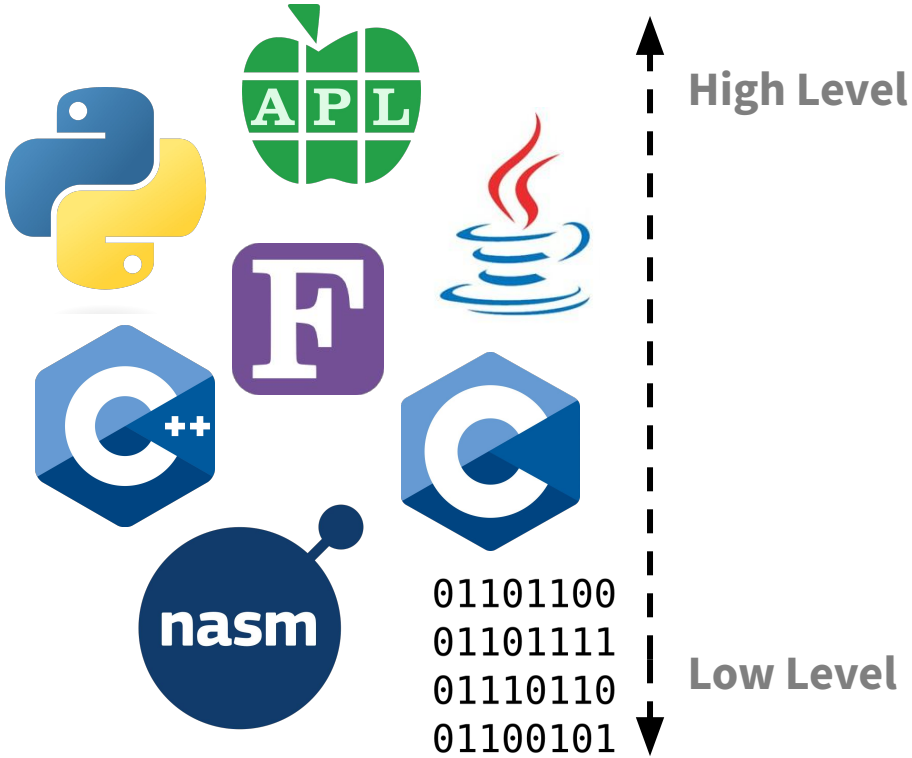
Levels of Programming Languages

High Level Languages:

- Python
- C++
- Java
- Ect.

Low Level Languages:

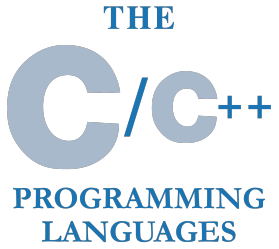
- x86 ASM (NASM)
- Machine Language



Interests

The reason I wanted to do this project was:

- 5 years of **PJAS**, wanted to *understand* and *gain appreciation* for the compiler
- I wanted to get better at **programming in ASM**
- I wanted to learn how different languages are designed



Goal

- The **main goal** of this project was to create a compiled programming language
  - **Features to include:** variables, if-statements, else-statements, while loops, functions, boolean evaluation
- Also to analyze how the programming language runs and generates.

Compiled vs *Interpreted* Languages

Compiled:

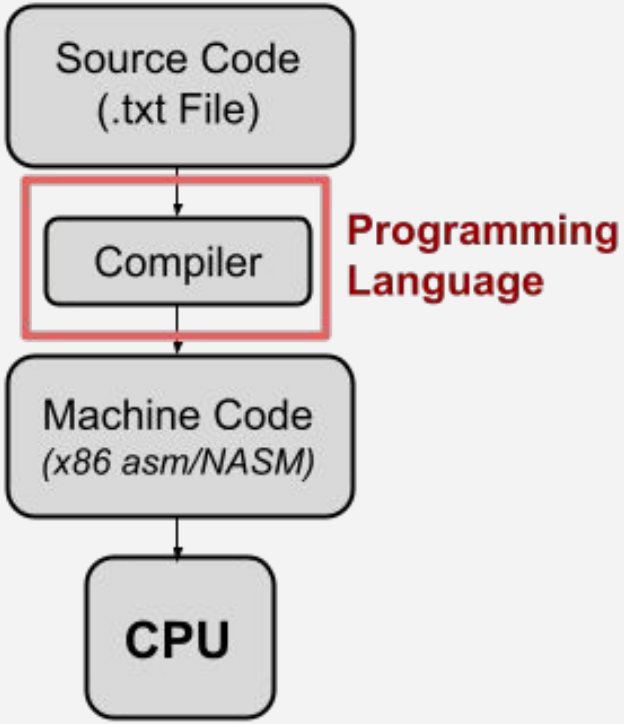
- Uses compiler on text file
- Returns machine code (as **.exe** file)
- Ex: C, C++, Go, Fortran

Interpreted:

- Text file interpreted by the language (usually a VM)
- **Line-by-Line** execution
- Ex: Python, JavaScript, C#

Flow of Compiled Languages

- The Compiler's job is to make **efficient**, **safe**, and **scalable** *Machine Code*
  - Out of Source Programming Language Syntax



Versatile (.vst)

Language's name is *"Versatile"*

- *Compiled Language*
- Compiler written in **C++**
  - Outputs **NASM assembly in .asm file**
- Designed to be lower level

```
func slope(x1, y1, x2, y2) {
    -- Input: (Point 1: x1, y1) (Point 2: x2, y2) --
    -- Output: Slope between two points (0 if negative) --
    var top = 0;
    var bottom = 0;

    -- No negative slopes allowed --
    if ((y2 >= y1) == (x2 >= x1)) {
        top = y2 - y1;
        bottom = x2 - x1;

        give(top/bottom);
    } wif ((y1 > y2) == (x1 > x2)) {
        top = y1 - y2;
        bottom = x1 - x2;

        give(top/bottom);
    } else {
        give(0); --Negative Slope
    }
}

-- Point 1 | (5,2) --
var Px1 = 5;
var Py1 = 2;

-- Point 2 | (10,12) --
var Px2 = 10;
var Py2 = 12;

ret(slope(Px1, Py1, Px2, Py2));
```

Demo of Versatile's Syntax

Why C++ and NASM?

- Compiler programmed in *C++*

Pros:

- Huge flexibility in data structures
- Fast and Low Level
- Supported across <u>most</u> operating systems

- *NASM* used instead of native x86 asm

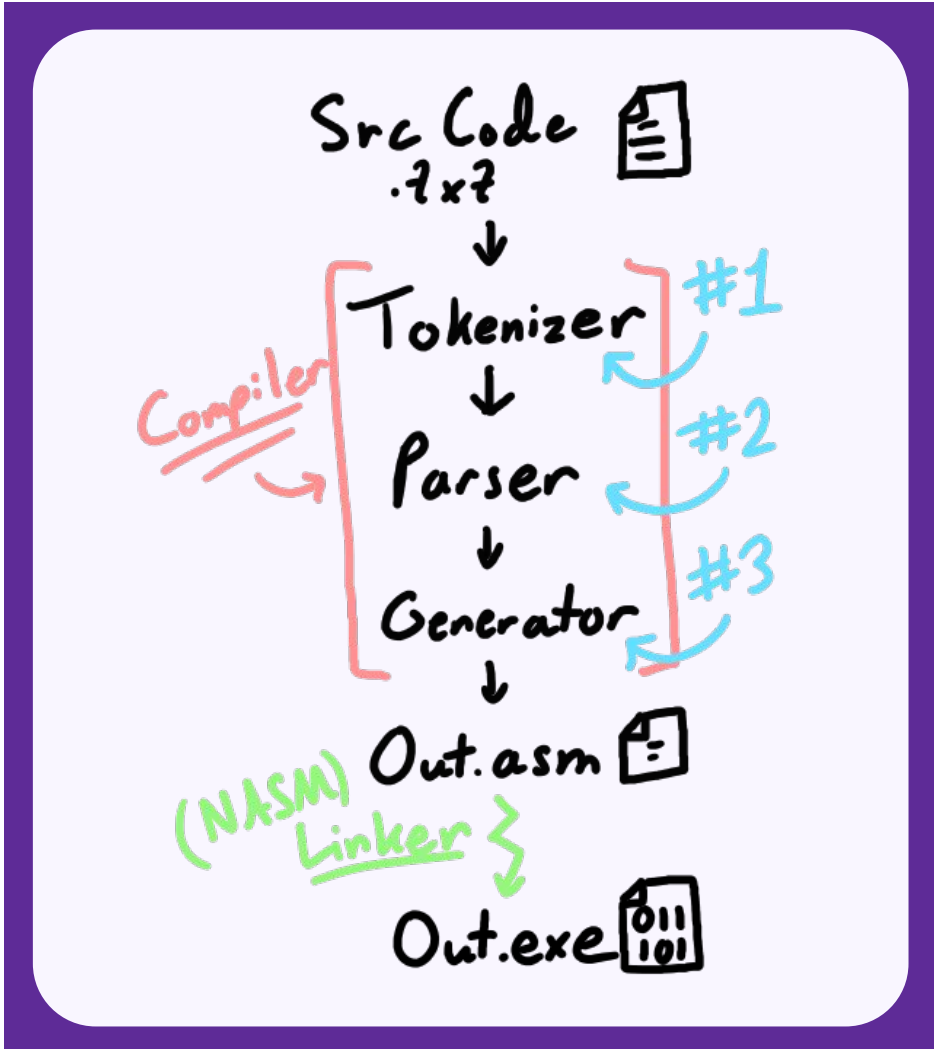
Pros:

- NASM can be used on <u>most</u> OS (that install it)
- Minor syntax changes
- VERY PORTABLE
  - Only **one** version of compiler

Compiler Flow Chart

Compiler will be a C++ file and will have **3 main functions** to run

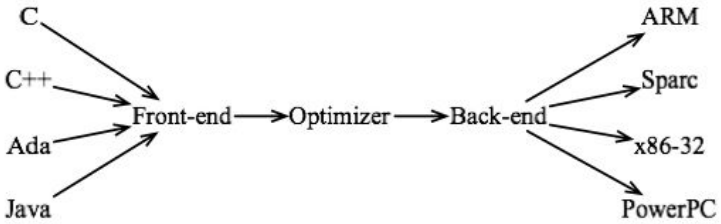
- Tokenizer (1st)
- Parser (2nd)
- Generator (3rd)
  - *Linking .asm with NASM (last)*



Compiler Flow Chart (Cont.)

Example showing the **final NASM output**

- After compiling .txt file (.vst file for *Versatile*)




```
5   var exampleVariable = 10;
6
7   ret(exampleVariable);
```

Compiler

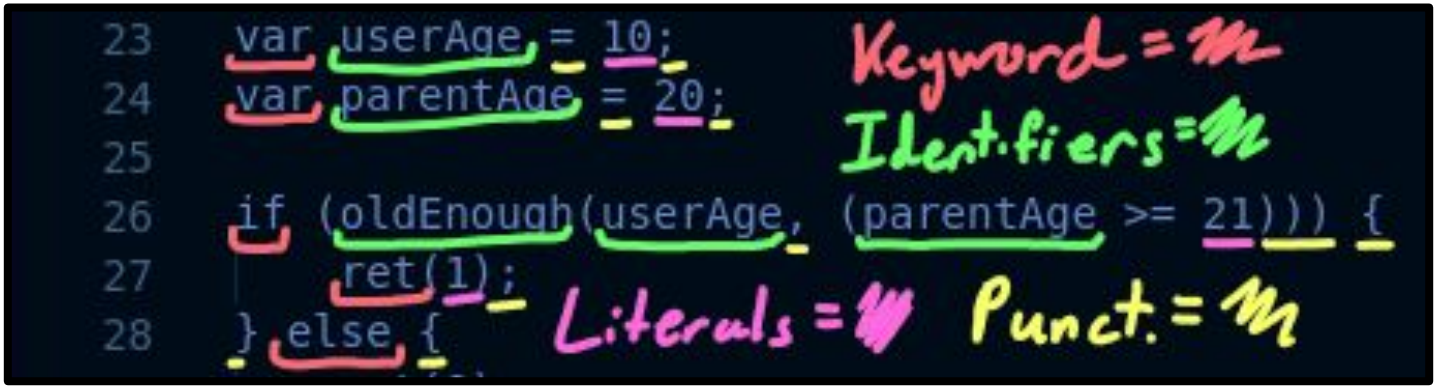
```
1   global _start:
2   _start:
3       mov rax, 10
4       push rax
5       push QWORD [rsp + 0]
6       mov rax, 60
7       pop rdi
8       syscall
9       mov rax, 60
10      mov rdi, 0
11      syscall
```

(#1) The Tokenizer & Tokenization

Lexical Analysis

- Takes away white space and makes “tokens”
  - Identifiers, Keywords, Operators, Literals, etc.
  - NO LOGIC YET (Preparing for Parser)

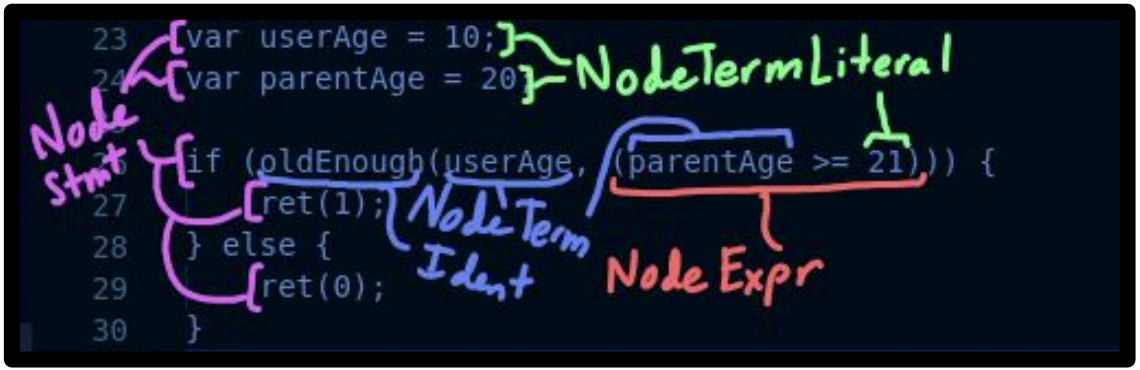
Example of how the Tokenizer Categorizes the Code



(#2) The Parser

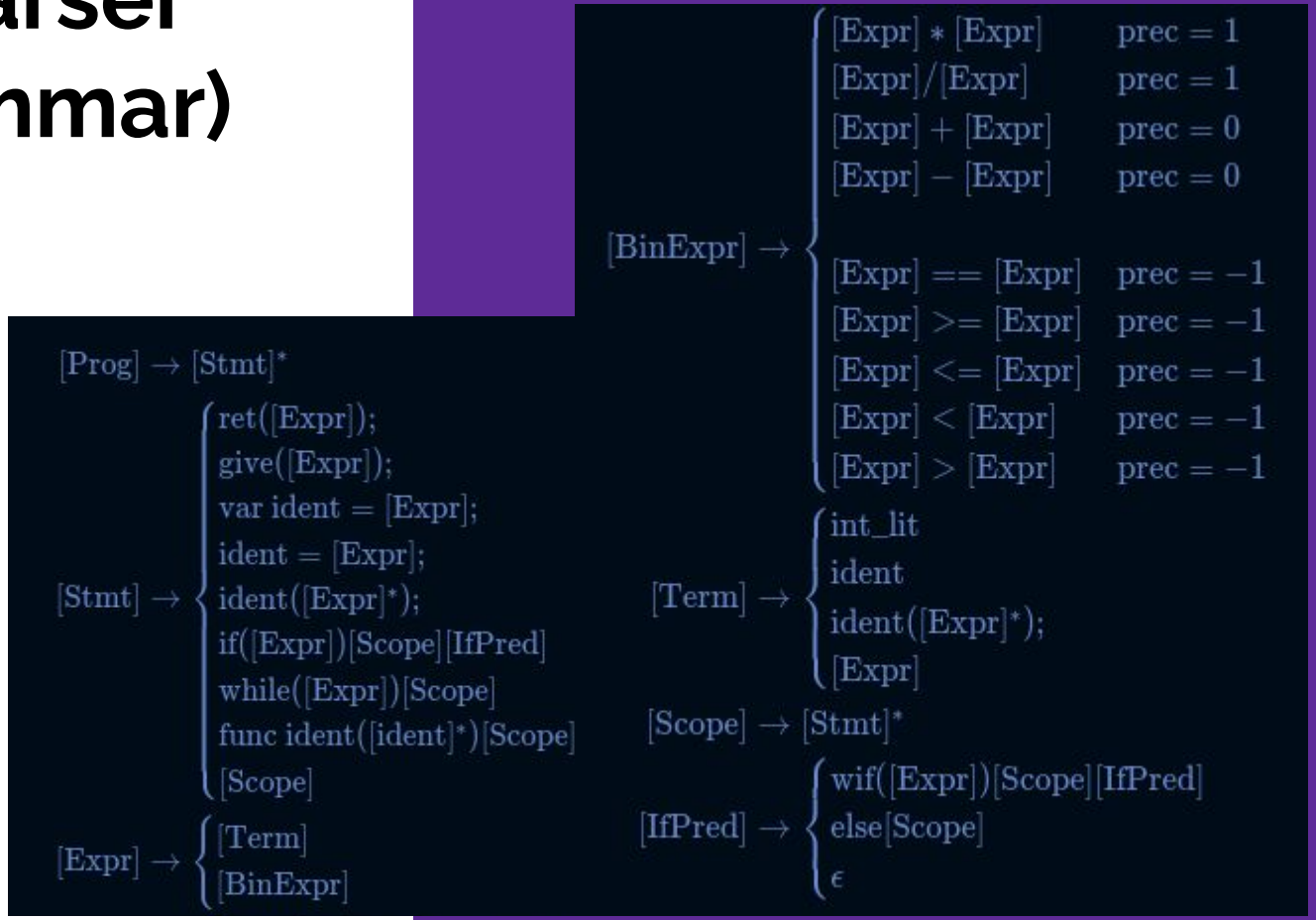
Takes in Tokens and makes Nodes

- Nodes are structs that contain other nodes (Bin Tree)
  - **Needs Grammar!!!**
- Also does syntax checking and some logic
- Need to allocate Nodes as well into memory…



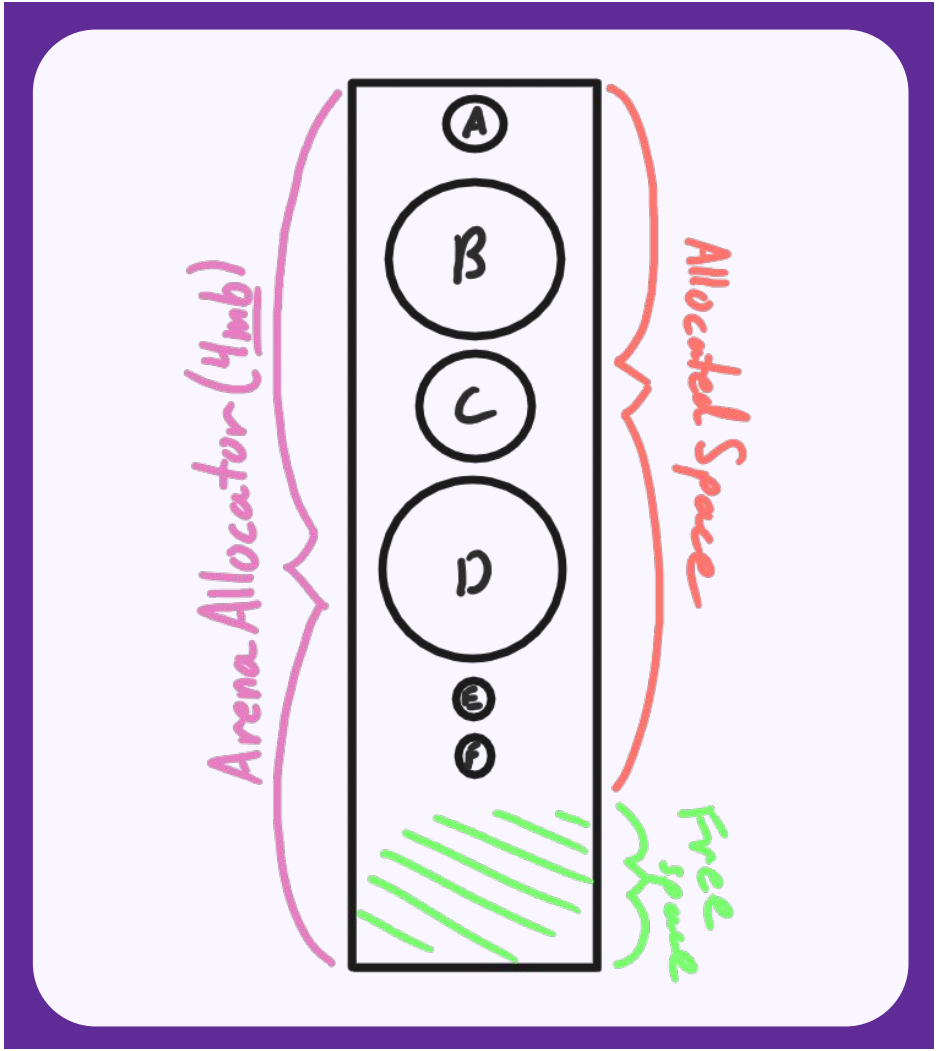
(#2) The Parser cont. (Grammar)

- *Grammar* is needed for syntax and **program validity**
- Grammar for “Versatile” *(Right)*



(#2) The Parser cont. (Allocator)

- All *Nodes* are referred to in *pointers*
  - Need to **allocate memory & access** them



When initialized with memory (4mb):

```
7   class ArenaAllocator {
8   public:
9       inline ArenaAllocator(size_t bytes) : m_size(bytes){
10          m_buffer = static_cast<std::byte*>(malloc(m_size)); //Check what this does!
11          m_offset = m_buffer;
12      }
13
14      template<typename T>
15      inline T* alloc() {
16          void* offset = m_offset;
17          m_offset += sizeof(T);
18
19          return static_cast<T*>(offset);
20      }
21
22      inline ArenaAllocator(const ArenaAllocator& other) = delete; //Constructor
23
24      inline ArenaAllocator operator=(const ArenaAllocator& other) = delete; //Equals Operator
25
26      inline ~ArenaAllocator() { //Destructor
27          free(m_buffer);
28      }
29
30  private:
31      size_t m_size;
32      std::byte* m_buffer;
33      std::byte* m_offset;
34  };
```

(#3) The Generator

Traverses the NodeTree and generates valid NASM

- *Bulk* of logic checking occurs
- Assembly *Stack* is utilized**!!!**

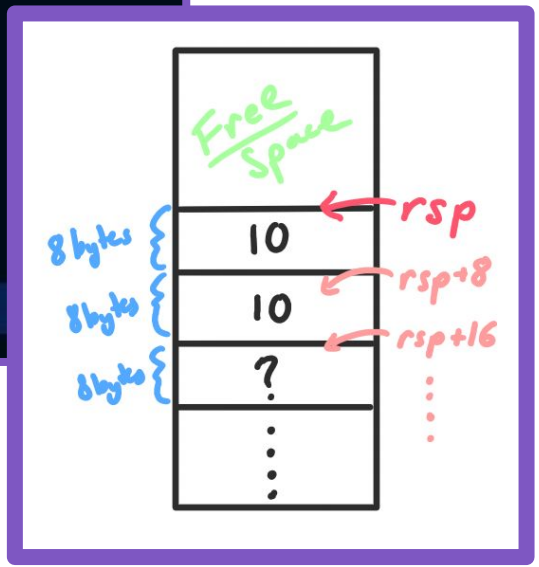
```
19  void gen_term(const NodeTerm* term) {
20      struct TermVisitor {
21          Generator* gen;
22          void operator()(const NodeTermIntLit* term_int_lit) const {
23              gen->m_out << "    mov rax, " <<term_int_lit->int_lit.value.value() << "\n";
24              gen->push("rax"); //ontop of stack
25          }
26          void operator()(const NodeTermIdent* term_ident) const {
27              //Extracting Variable Contents (get from top of stack)
28              auto it = std::find_if(gen->m_vars.cbegin(), gen->m_vars.cend(), [&](const Var& var){
29                  return var.name == term_ident->ident.value.value();
30              });
31              if (it == gen->m_vars.cend()) {
32                  std::cerr << "Undeclared identitfier: " << term_ident->ident.value.value() << std::endl;
33                  exit(EXIT_FAILURE);
34              }
35              //%rsp is the guy// //(*it) is the variable :)//
36              std::stringstream offset;
37              offset << "QWORD [rsp + " << (gen->m_stack_size - (*it).stack_location - 1) * 8 << "]";
38              gen->push(offset.str());
39          }
40          void operator()(const NodeTermFunc* term_func) const {
```

(#3) The Generator cont. (Stack)

Can't only use registers (rax, rbx, etc.) for vars & exprs

- Use *Stack* instead (essentially a large array)

```
1   global _start:
2   _start:
3       mov rax, 10
4       push rax
5       push QWORD [rsp + 0]
6       mov rax, 60
7       pop rdi
8       syscall
9       mov rax, 60
10      mov rdi, 0
11      syscall
12
```



Example program (above) with final stack (right)

Testing/Programming Environment

Using Win. VirtualBox with **64-bit Linux (CentOS 8)**

- RAM: **8GB** Memory
- CPU: 4-Core Intel i5-3570k @ **3.3GHz** *(100% Alloc)*
- Disk: Samsung 256GB SSD

VS Code used along with g++ (8.5.0) for running C++
NASM version 2.16.01

Results/Data

Compiler runs relatively quickly!

- Language has most features implemented with a **low** Token-to-NASM Line Ratio

Prog Name	Average Run Time (s)	Token Count	NASM Line Count	File Line Count
VST Compiler	0.00454556	N/A	N/A	1438
age.vst	0.00144785	81	91	30
fibonacci.vst	0.00121443	77	88	20
power.vst	0.00121991	64	71	15
slope.vst	0.00154596	139	168	31
example.vst	0.00112496	10	11	7

Analysis

Versatile is grammatically close to **C++**

- Takes inspiration from other languages (Lua comments, “var” declarations in JS, etc.)

Also runs in about the same time as other languages (within *ms*)

Lets see some example programs…

Example Programs:

```
1   func power(base, pow) {
2       var return = base;
3       var it = 0;
4
5       while (pow > 1) {
6           return = return * base;
7           ret(return);
8           pow = pow - 1;
9       }
10      give(return);
11  }
12
13  var ans = 0;
14
15  ret(power(3,2));
```

Power.vst: **Returns the power input to the base parameter (ex: 3, 2 in = 3^2 = 9)**

```
1   func fibonacci(count) {
2       --[[returns val in position #(count) in fib seq]]--
3       var a = 0;
4       var b = 1;
5       var c = 0;
6
7       count = count + 2;
8
9       var counter = 1;
10      while (counter <= count - 2) {
11          c = a + b;
12          a = b;
13          b = c;
14
15          counter = counter + 1;
16      }
17      give(c);
18  }
19
20  ret(fibonacci(9));
```

Fibonacci.vst: **Returns the respective number in the fibonacci sequence**

Example Programs (cont):

```
func oldEnough(age, pCons) {

    if (age >= 18) {
        give(true);
    } wif (pCons) {
        give(true);
    } else {
        give(false);
    }

}

--[[
INPUTS:
userAge = age of user
parentAge = age of parent

OUTPUT:
1 = Allow User into Program
0 = Disallow User into Program
]]--

var userAge = 10;
var parentAge = 20;

if (oldEnough(userAge, (parentAge >= 21))) {
    ret(1);
} else {
    ret(0);
}
```

Age.vst: **Returns 1 *(true)* or 0 *(false)* if the input Age Variables are not great enough**

```
var exampleVariable = 10;

ret(exampleVariable);
```

Example.vst: **Returns the initialized variable**

```
global _start:
_start:
    mov rax, 10
    push rax
    push QWORD [rsp + 0]
    mov rax, 60
    pop rdi
    syscall
    mov rax, 60
    mov rdi, 0
    syscall
```

Errors and Bugs

- Assembly debugging is painful (stack overflows & seg faults)
- C++ data structures became complex
- Virtual machine ran CentOS slow

I *fixed* these issues by:

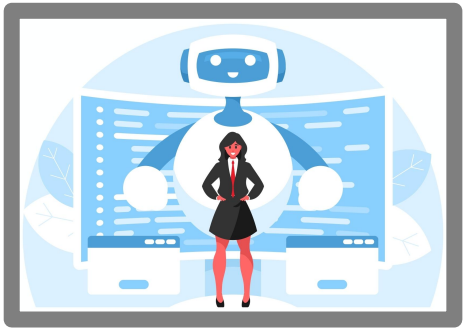
- Talking to **my teachers**
- Consulting Youtube & Stack Overflow
- Using test oriented development

Conclusion

- I met my goal of creating a compiled programming language and testing it sufficiently
  - If I would do this project again, I would implement more **data-types** and **syscall integrations** into the program
- *This project taught me that mistakes are inevitable, but having **determination** is the only way to prevail.*

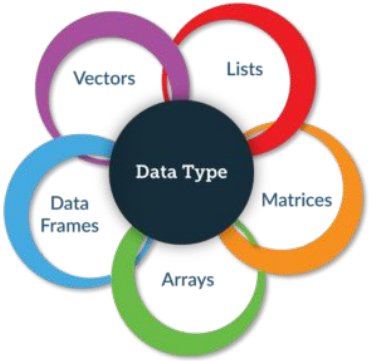
Other “Practical” Uses

- Many concepts used in **Compilers** in other areas:
  - Lexical Analysis/Tokenization used in **NLP** (Natural Language Processing in AI)
  - Memory Mapping (Arena Allocation)
  - *Precedence Climbing* (Parsing Order of operations for Calculators)



Expansion

- Using my new knowledge of **assembly** and **programming language design**, I could expand on this project by:
  - Implementing **char & class** data types into *Versatile*
  - Improving *readability* of errors and compiler code
  - Integrating different programming languages into Versatile
  - Making keywords dynamically typed so Users can change them



References

- **https://www.geeksforgeeks.org/ C++ and Nasm Debugging**
- *Programming from the Ground UP* by Jonathan Bartlett (ASM Reference)
- Precedence Climbing Algorithms https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing

My Github Repository: (Source Code + Tests)
https://github.com/Gameknight-Bit/PJAS2023-2024