COMPUTER SYSTEMS AND ORGANIZATION

C compilation Part 2

Daniel G. Graham Ph.D

University of Virginia | ENGINEERING

Contents

SIMPLE PROGRAM

```
int main() {
	return 7;
}
```

~ — -zsh   ...dgg6b@kytos02: ~/escaperoom/rooms/room9 — ssh dgg6b@portal.cs.virginia.edu   ~ — -zsh

GNU nano 6.3   main.c

```c
int main(){
        return 2;
}
```

GNU nano 6.3   main.s

```
        .text
        .file   "main.c"
        .globl  main                                    >
        .p2align        4, 0x90
        .type   main,@function
main:                                                  #>
        .cfi_startproc
# %bb.0:
        movl    $2, %eax
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                                       #>

        .ident  "clang version 14.0.6 (ht>
        .section        ".note.GNU-stack">
        .addrsig
```

[Read 17 lines]

^G Help ^O Write ^W Where ^K Cut   ^G Help ^O Write ^W Where ^K Cut

SIMPLE PROGRAM

```c
int main() {
    return 7;
}
```

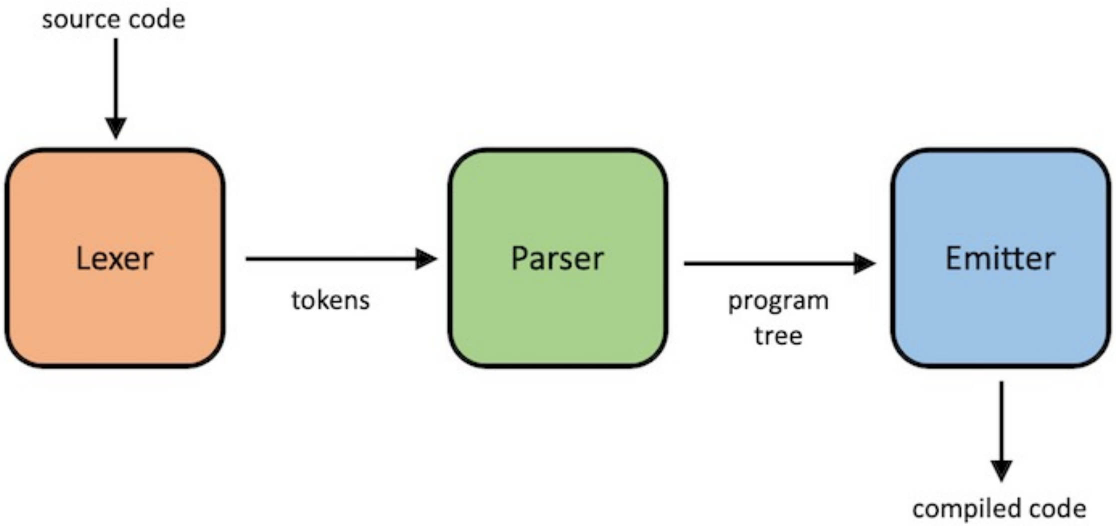
```
        .globl  main
main:
        movl    $2, %eax
        retq
```

CAN WE BUILD SOMETHING REALLY SIMPLY THAT COMPILES THIS?

```python
import sys, os

source_file = sys.argv[1]
assembly_file = os.path.splitext(source_file)[0] + ".s"

with open(source_file, 'r') as infile, open(assembly_file, 'w') as outfile:
    source = infile.read().strip()

    # Find the index of "int main()" and "return" in the source code
    main_start = source.find("int main()")
    return_start = source.find("return", main_start)

    if main_start != -1 and return_start != -1:
        # Extract the return value
        return_value = source[return_start + 6:].strip().rstrip(";\n}")
        # Write the assembly code to the output file
        assembly_code = f"""
        .globl main
        main:
        movl ${return_value}, %eax
        ret
        """
        outfile.write(assembly_code)
    else:
        print("Error: Couldn't find 'int main()' or 'return' in the source code.")
```

THIS DOESN'T SCALE TO MORE COMPLEX PROGRAMS

THE PROCESS OVERVIEW



STEP 1: LEXING/SCANNING TOKENING

Raw text: WHILE nums > 0 REPEAT

Tokens: WHILE nums > 0 REPEAT

TOKENS HAVE MEANINGS

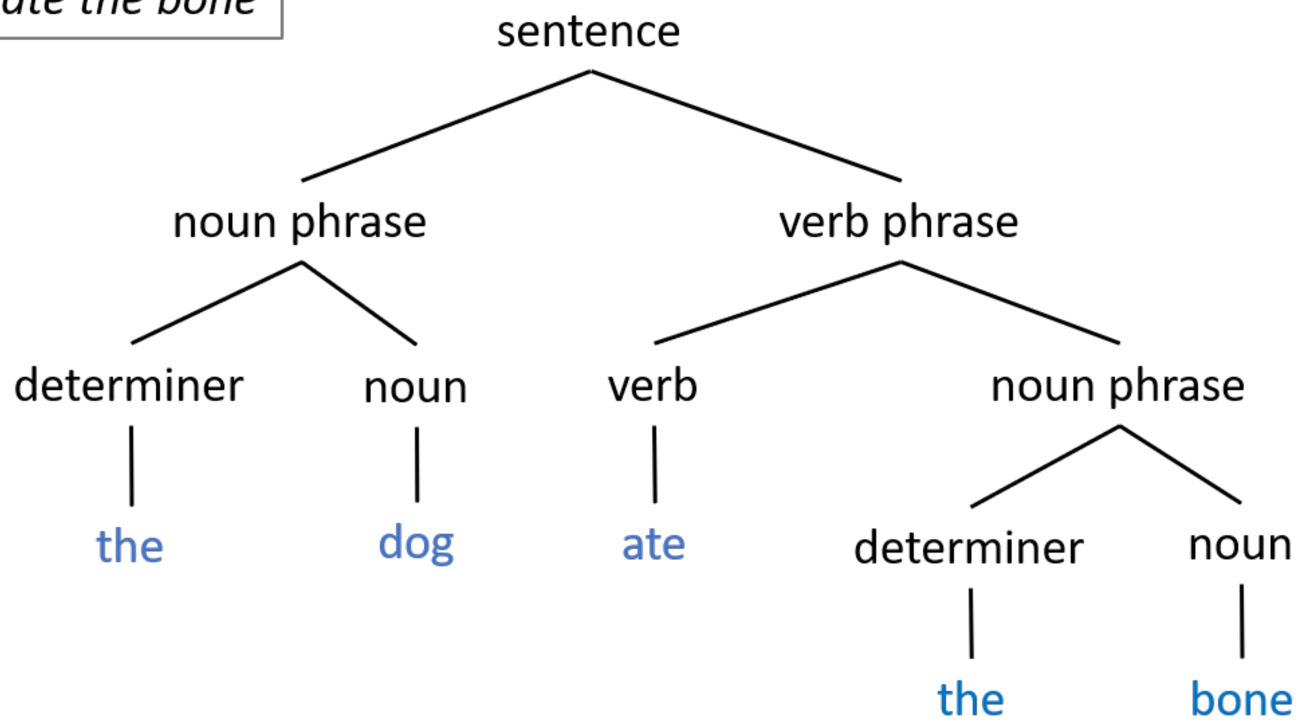
Raw text: WHILE nums > 0 REPEAT

Tokens: WHILE nums > 0 REPEAT

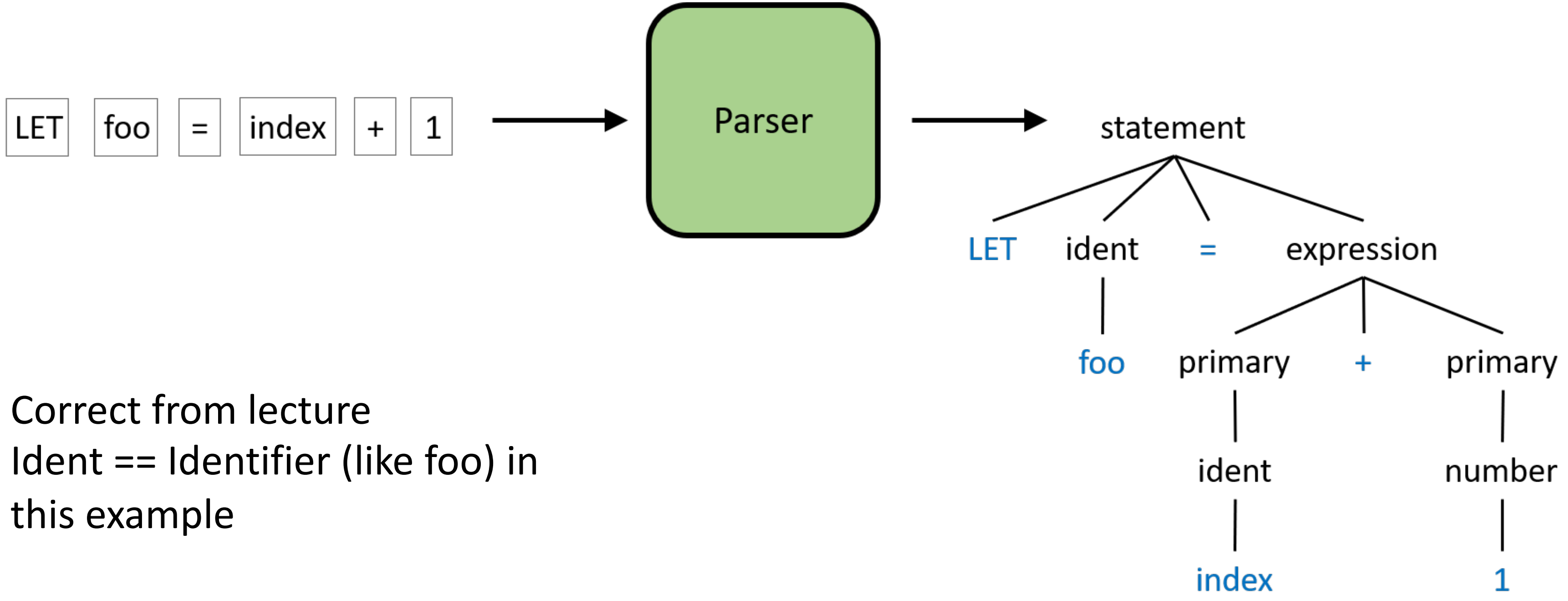
keyword identifier operator number keyword

STEP 2: PARSING

the dog ate the bone



STEP 2: PARSING



Correct from lecture
Ident == Identifier (like foo) in this example

Not indent** Sorry

BACKUS-NAUR FORM

<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}”
<statement> ::= "return" <exp> ";"
<exp> ::= <int>

```
int main() {
    return 7;
}
```

<Non-terminal> ::= Terminal

```
program ::= {statement}
statement ::= "PRINT" (expression | string) nl |
        "IF" comparison "THEN" nl {statement} "ENDIF" nl |
        "WHILE" comparison "REPEAT" nl {statement} "ENDWHILE" nl |
        "LABEL" ident nl |
        "GOTO" ident nl |
        "LET" ident "=" expression nl |
        "INPUT" ident nl
comparison ::= expression (("==" | "!=" | ">" | ">=" | "<" | "<=") expression)
expression ::= primary {operator primary}
primary ::= number | ident
```

Simplified to support single operators. So that we don’t have to deal with operator precedence

LET'S PARSE A PROGRAM

```
program ::= {statement}
statement ::= "PRINT" (expression | string) nl |
        "IF" comparison "THEN" nl {statement} "ENDIF" nl |
        "WHILE" comparison "REPEAT" nl {statement} "ENDWHILE" nl |
        "LABEL" ident nl |
        "GOTO" ident nl |
        "LET" ident "=" expression nl |
        "INPUT" ident nl
comparison ::= expression (("==" | "!=" | ">" | ">=" | "<" | "<=")
expression)
expression ::= primary {operator primary}
primary ::= number | ident
```

```
PRINT "How many fib #?"
INPUT nums
PRINT ""
LET a = 0
LET b = 1
WHILE nums > 0 REPEAT
    PRINT a
    LET c = a + b
    LET a = b
    LET b = c
    LET nums = nums - 1
ENDWHILE
```

CODE GENERATION

HOW DO WE GO FROM OUR AST TO ASSEMBLY

WE'LL MAKE IT A SUBSET OF PYTHON

Python

pyast64

ToyG

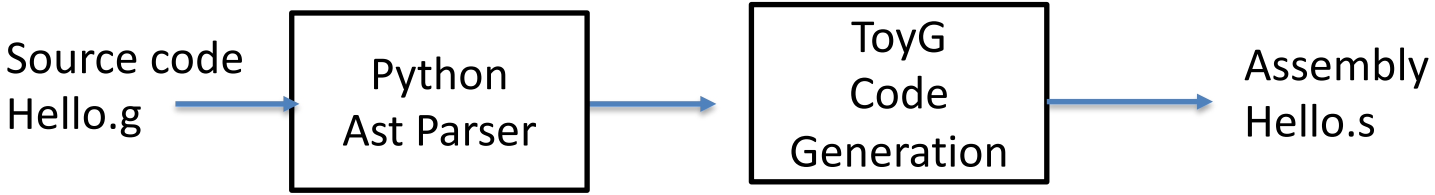
Instead of be interpreted like python.

Our language will be compiled meaning that will write a compiler that outputs assembly

THE PROCESS

Source code
Hello.g → ToyG Compiler → Assembly
Hello.s

LEVERAGE THE PYTHON AST PARSER



Common to use framework when developing your own language for example LLVM has library and support for developing your own lexer and parsers

LET'S START BUILDING OUR CODE GENERATOR

```
if __name__ =='__main__':
     gFile = sys.argv[1]
     source = open(gFile).read()
     node = ast.parse(source, filename=gFile)
     compiler = Compiler()
     compiler.compile(node)
     assembled_code = "\n".join(compiler.asm)
     output =f"""
.globl main

main:
{assembled_code}
ret"""
     open(sys.argv[1].replace(".g",".s"), 'w').write(output)
```

```
class Compiler:
def __init__(self):
     self.asm = []
     self.localVars = []

def compile(self, node):
     self.visit(node)

def visit(self, node):
     ## Get the name of AST Node
     name = node.__class__.__name__
     visit_func = getattr(self, 'visit_' + name, None)
     print("visiting "+ast.dump(node)+"\n")
     if(visit_func == None):
     print(name+" node not supported \n")
     sys.exit() # end compilation

     visit_func(node)
```

NOW LET'S IMPLEMENT THE VISITORS

EXPRESSION AND CONSTANT VISITORS

```
def visit_Module(self, node):
    for statement in node.body:
        self.visit(statement)

def visit_Expr(self, node):
    self.visit(node.value)
    self.asm.append('popq %rax')

def visit_Constant(self, node):
    self.asm.append('pushq ${}'.format(node.value))
```

NOW LET'S EXTEND IT SO THAT WE CAN DO ASSIGNMENTS

For example, we want to be able to write

X = 5

Y = 3

LET'S WRITE THE ASSIGNMENT VISITOR

BUT FIRST LET'S MAKE ROOM ON THE STACK

```
output =f"""
.globl main

main:
movq %rsp, %rbp # prolgue
subq $64, %rsp # allocate room for up to 8 local
variables
{assembled_code}
addq $64, %rsp # deallocate space on the stack
ret"""
```

NOW LET'S WRITE SOMETHING THAT CALCULATES WHERE WE SHOULD PUT AND FIND VARIABLES ON THE STACK

Draw this stack

```
def local_offset(self, name):
    if not name in self.localVars:
        self.localVars.append(name)
        index = self.localVars.index(name)
        print(self.localVars)
        return (index) * 8 + 8
```

NOW LET'S DO THE ASSIGNMENT

Draw the stack

```
def visit_Assign(self, node):
    self.visit(node.value)
    offset = self.local_offset(node.targets[0].id)
    self.asm.append('popq {}(%rbp)'.format(offset))

def visit_Constant(self, node):
    self.asm.append('pushq ${}'.format(node.value))
```

LET'S BUILD UP ENOUGH LANGUAGE SO THAT WE CAN IMPLEMENT FIBONACCI

So we need

5 + 2

And

X + Y

so let's build these

BUILD BINARY UP

```
def visit_BinOp(self, node):
    self.visit(node.left)
    self.visit(node.right)
    self.visit(node.op)
```

Let’s think about this case for

5+3

What does the stack look like?

How could we implement add

```
def visit_Constant(self, node):
    self.asm.append('pushq ${}'.format(node.value))
```

LET BUILD THE OP ADD VISIT

```
def visit_Add(self, node):
    self.asm.append('popq %rdx')
    self.asm.append('popq %rax')
    self.asm.append('addq %rdx, %rax')
    self.asm.append('pushq %rax')
```

What does the stack look like?

5 + 3

WHAT ABOUT EXPRESSION ADDS

```
def visit_Add(self, node):
    self.asm.append('popq %rdx')
    self.asm.append('popq %rax')
    self.asm.append('addq %rdx,
    %rax')
    self.asm.append('pushq %rax')
```

What about something like

X + Y

```
def visit_Name(self, node):
    offset = self.local_offset(node.id)
    self.asm.append('pushq -{}(%rbp)'.format(offset))
```

LET LOOK AT ASSIGNMENTS

```
def visit_AugAssign(self,node):
    self.visit(node.target)
    self.visit(node.value)
    self.visit(node.op)
    offset = self.local_offset(node.target.id)
    self.asm.append('popq -{}(%rbp)'.format(offset))
```

```
def visit_Name(self, node):
    offset = self.local_offset(node.id)
    self.asm.append('pushq -{}(%rbp)'.format(offset))
```

```
def visit_Constant(self, node):
    self.asm.append('pushq ${}'.format(node.value))
```

How about something like

X+= 5

or

X+= Y

WHAT ABOUT LOOPS (LET'S WRITE COMPARE)

```
def visit_Compare(self, node):
    self.visit(node.left)
    self.visit(node.comparators[0])
    self.visit(node.ops[0])
```

Let's write the compare first

5 < 3

X < 4

4 < X

X < Y

WHAT ABOUT LOOPS (LET’S WRITE COMPARE)

```
def visit_Lt(self, node):
    self.asm.append('popq %rdx')
    self.asm.append('popq %rax')
    self.asm.append('cmpq %rdx,
    %rax')
```

Let’s write the compare first

5 < 3

X < 4

4 < X

X < Y

WHAT ABOUT LOOPS

```
def visit_While(self, node):
    self.asm.append(".while:")
    self.visit(node.test)
    self.asm.append("jz .break")
    for statement in node.body:
    self.visit(statement)
    self.asm.append("jmp .while")
    self.asm.append(".break:")
```

We can now run a program like this:

```
x = 2
y = 5
z = 0
while ( z < 0 ):
    x+= y
    z = 5
y = z
```

DEMO AND TALK ABOUT HOW FAR WE COME

REFERENCES

- https://norasandler.com/2017/11/29/Write-a-Compiler.html
- Abdulaziz Ghuloum's An Incremental Approach to Compiler Construction
- https://benhoyt.com/writings/pyast64/
- https://austinhenley.com/blog/teenytinycompiler2.html