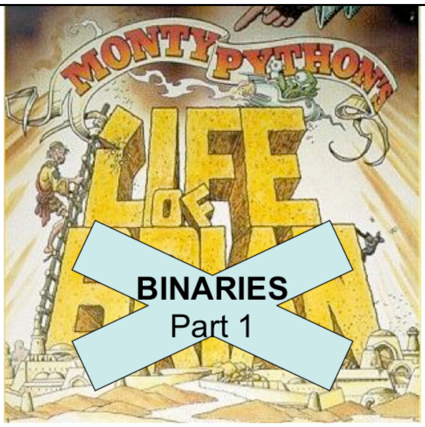


BINARIES
Part 1

Xeno Kovah – 2012
xkovah at gmail

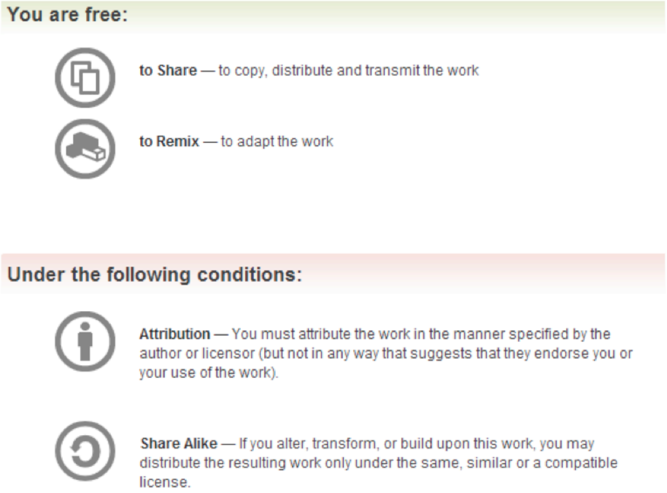
See notes for citation

Image from: http://unoperdia.blogspot.com/2010/04/life-of-brian.html

http://1.bp.blogspot.com/-9-8y-Ht4J9A/T53LDVrMwDI/AAAAAAAAAZQ/wyovxlb_ZCY/s1600/NewandImproved.jpg

All materials are licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

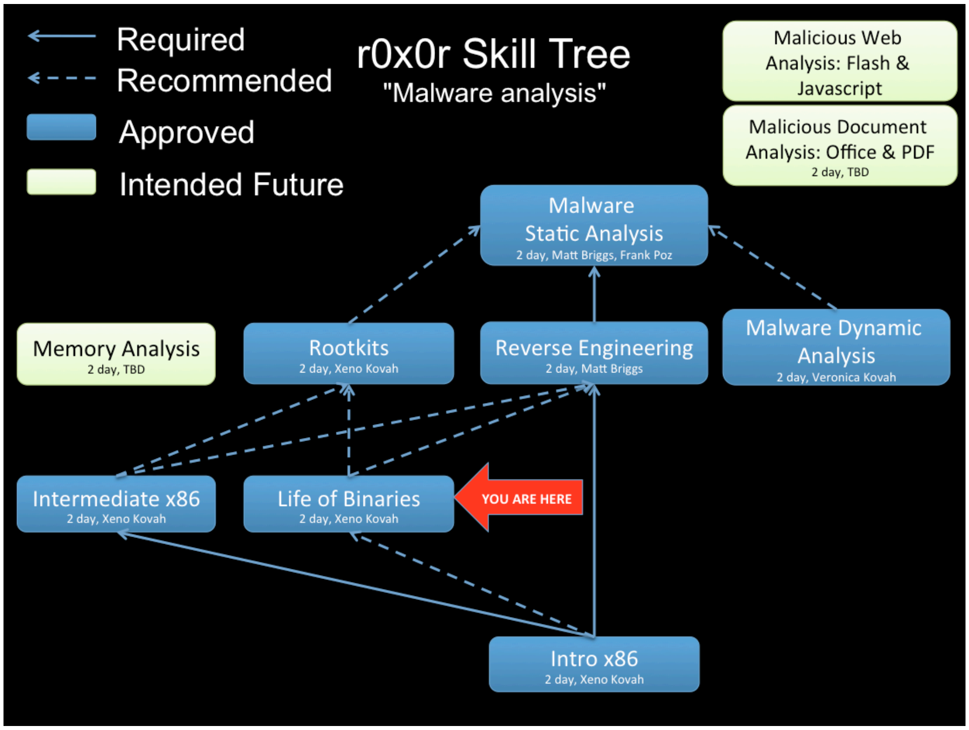
to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution conditions: Just state author name and where the slides were obtained from



r0x0r Skill Tree

"Malware analysis"

← Required

⇠ Recommended

🟦 Approved

⬜ Intended Future

Malicious Web Analysis: Flash & Javascript

Malicious Document Analysis: Office & PDF
2 day, TBD

Malware Static Analysis
2 day, Matt Briggs, Frank Poz

Memory Analysis
2 day, TBD

Rootkits
2 day, Xeno Kovah

Reverse Engineering
2 day, Matt Briggs

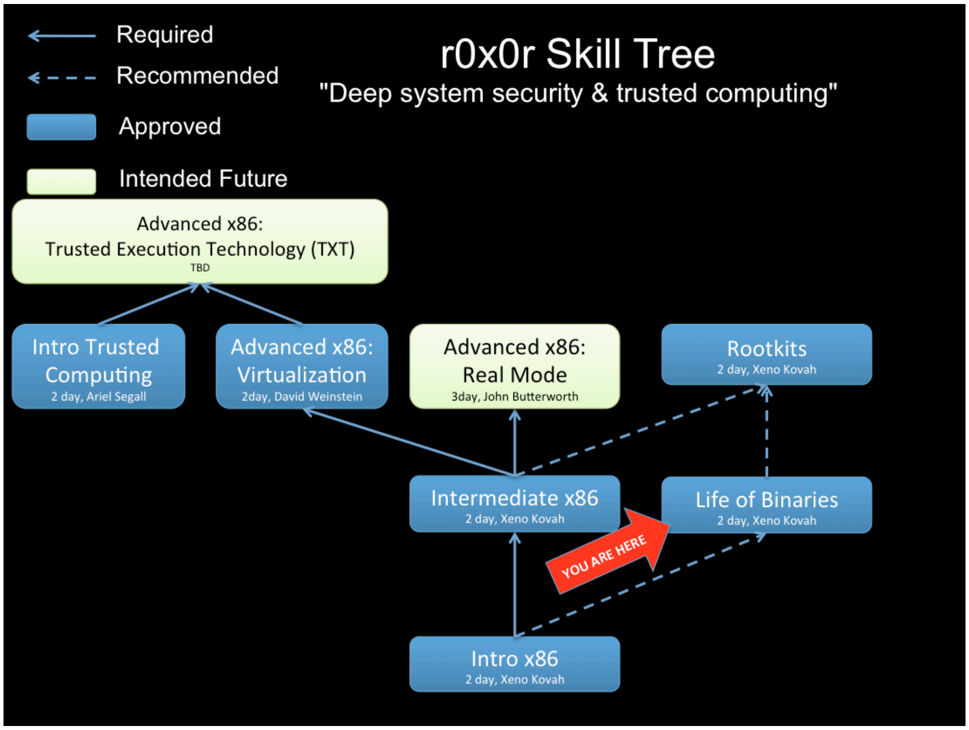
Malware Dynamic Analysis
2 day, Veronica Kovah

Intermediate x86
2 day, Xeno Kovah

Life of Binaries
2 day, Xeno Kovah

YOU ARE HERE

Intro x86
2 day, Xeno Kovah



r0x0r Skill Tree

"Deep system security & trusted computing"

← Required

⇠ Recommended

🟦 Approved

▭ Intended Future

Advanced x86:
Trusted Execution Technology (TXT)
TBD

Intro Trusted Computing
2 day, Ariel Segall

Advanced x86: Virtualization
2day, David Weinstein

Advanced x86: Real Mode
3day, John Butterworth

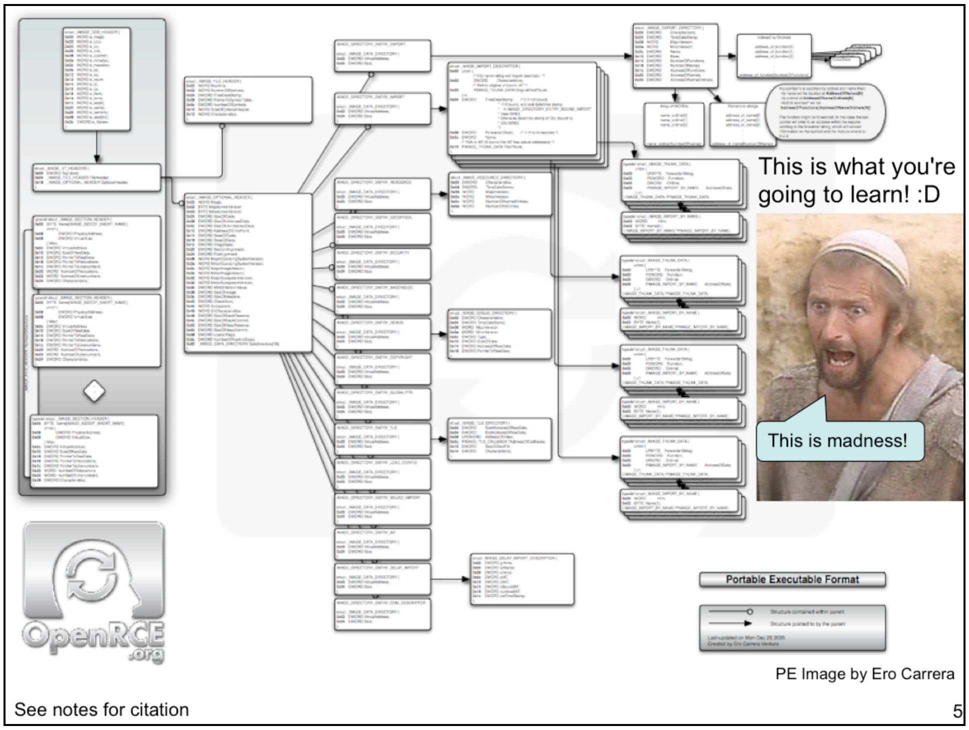
Rootkits
2 day, Xeno Kovah

Intermediate x86
2 day, Xeno Kovah

Life of Binaries
2 day, Xeno Kovah

YOU ARE HERE

Intro x86
2 day, Xeno Kovah



This is what you're going to learn! :D

This is madness!

PE Image by Ero Carrera

See notes for citation

Image from: http://unoperdia.blogspot.com/2010/04/life-of-brian.html

&

PE Image by Ero Carrera

http://www.openrce.org/reference_library/files/reference/PE%20Format.pdf

Whatcha gonna do, cry about it?

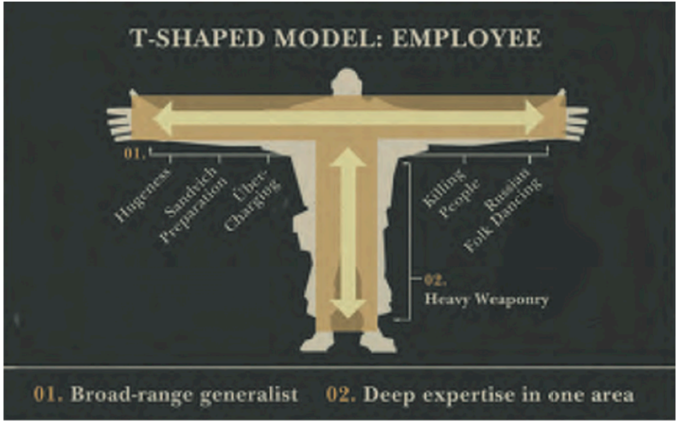
(It's funnier if you already took Intermediate x86)

See notes for citation

image from: http://www.wallyconger.com/wp-content/uploads/2009/12/Brian.jpg

About Me

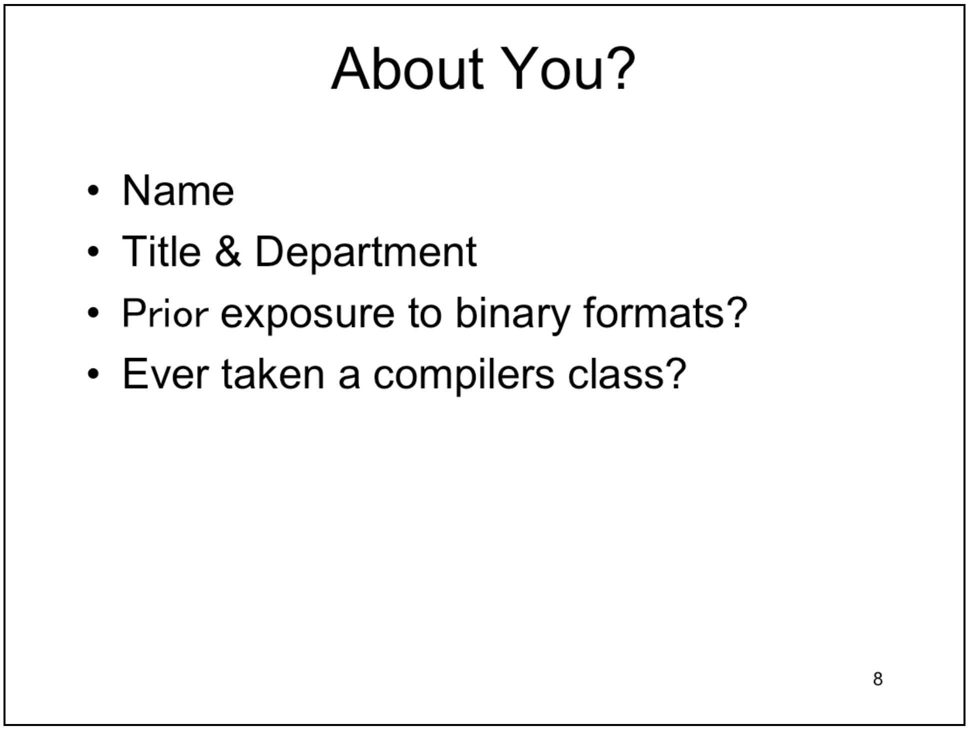
- Security nerd – T-Shaped!



- Mac OS person who's had to learn Windows for work. And if I have to learn something, I may as well learn it well.

See notes for citation

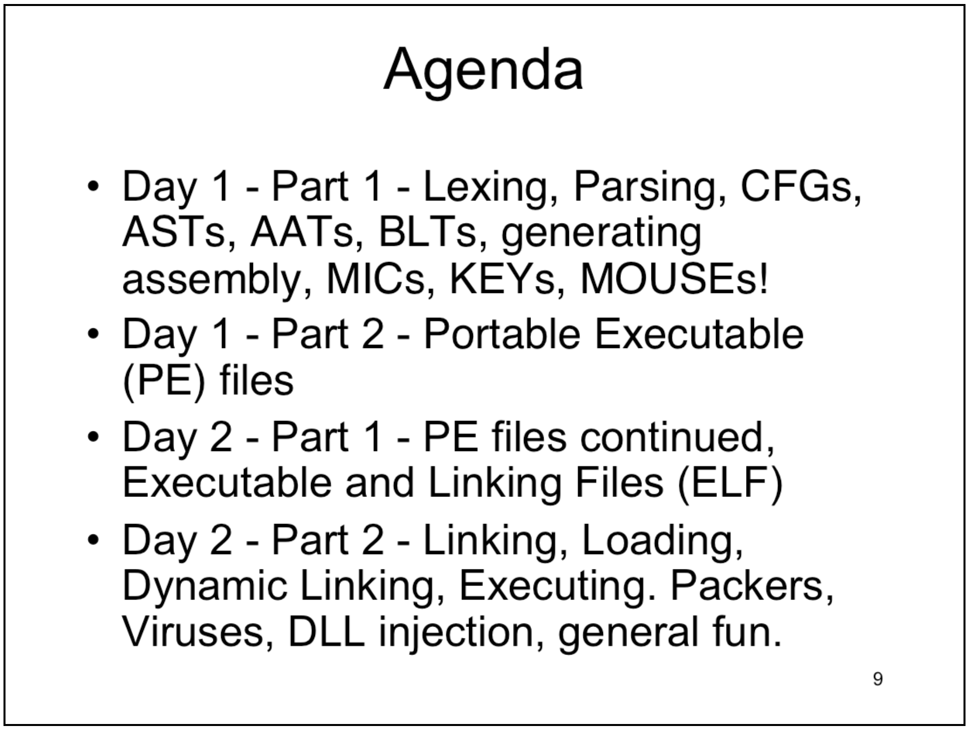
http://newcdn.flamehaus.com/Valve_Handbook_LowRes.pdf



About You?

- Name
- Title & Department
- Prior exposure to binary formats?
- Ever taken a compilers class?

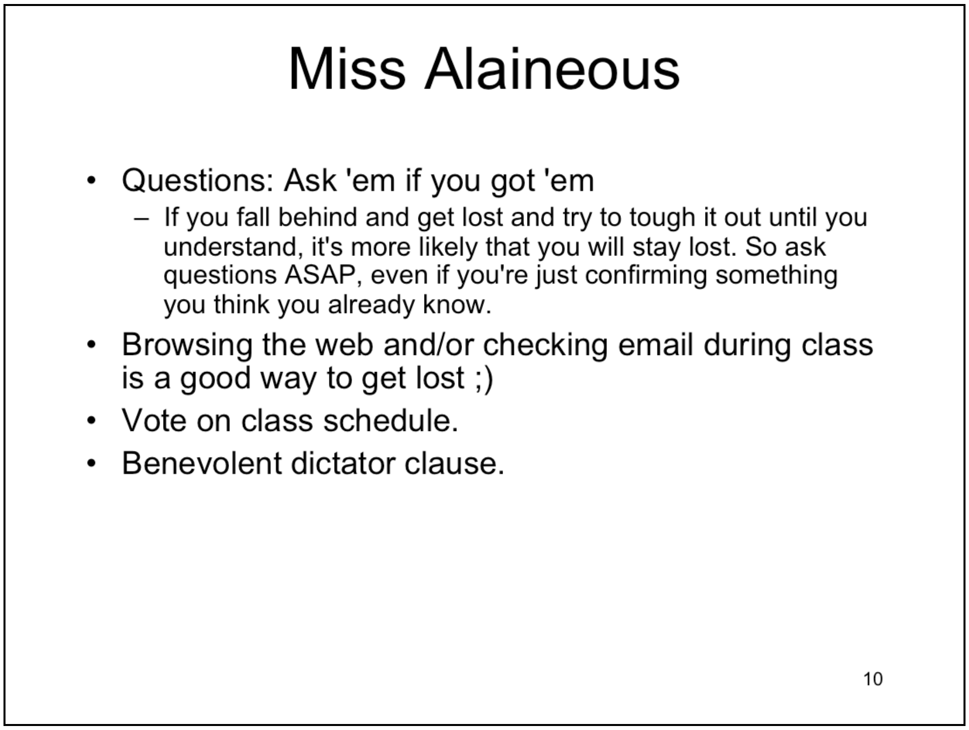
10 min



Agenda

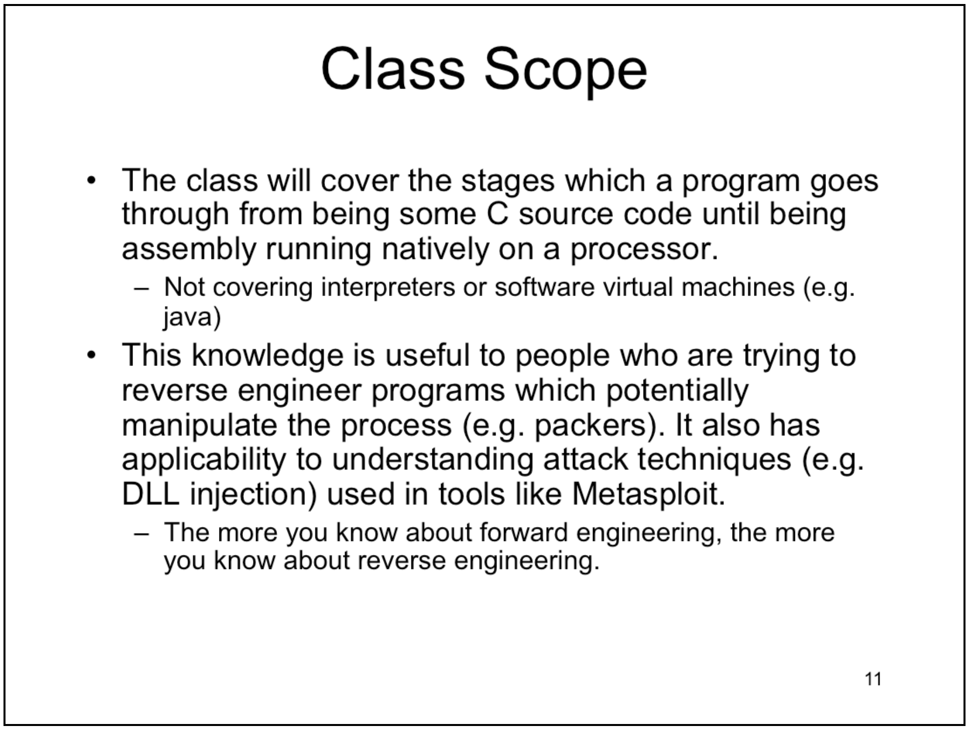
- Day 1 - Part 1 - Lexing, Parsing, CFGs, ASTs, AATs, BLTs, generating assembly, MICs, KEYs, MOUSEs!
- Day 1 - Part 2 - Portable Executable (PE) files
- Day 2 - Part 1 - PE files continued, Executable and Linking Files (ELF)
- Day 2 - Part 2 - Linking, Loading, Dynamic Linking, Executing. Packers, Viruses, DLL injection, general fun.

2 min



Miss Alaineous

- Questions: Ask 'em if you got 'em
  - If you fall behind and get lost and try to tough it out until you understand, it's more likely that you will stay lost. So ask questions ASAP, even if you're just confirming something you think you already know.
- Browsing the web and/or checking email during class is a good way to get lost ;)
- Vote on class schedule.
- Benevolent dictator clause.

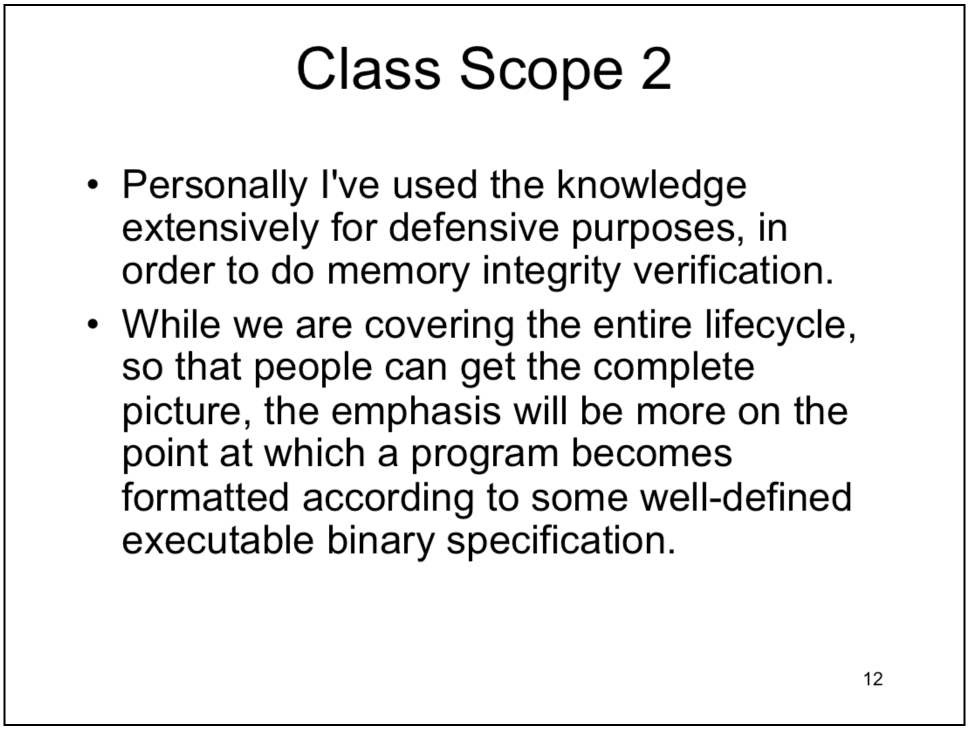


Class Scope

- The class will cover the stages which a program goes through from being some C source code until being assembly running natively on a processor.
  - Not covering interpreters or software virtual machines (e.g. java)
- This knowledge is useful to people who are trying to reverse engineer programs which potentially manipulate the process (e.g. packers). It also has applicability to understanding attack techniques (e.g. DLL injection) used in tools like Metasploit.
  - The more you know about forward engineering, the more you know about reverse engineering.

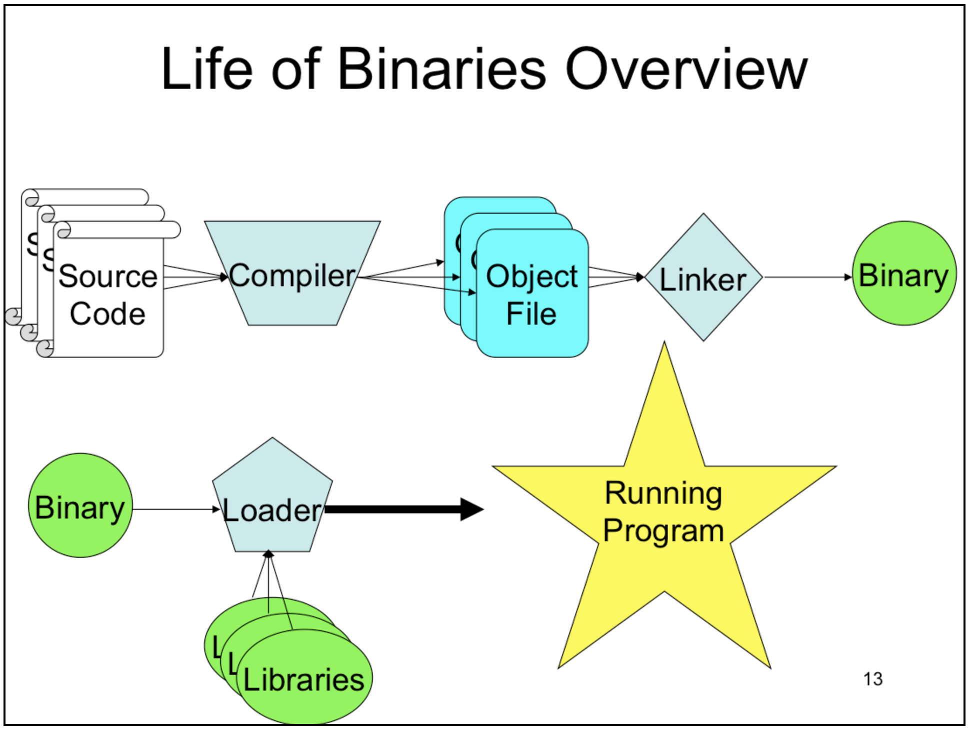
11

2 min

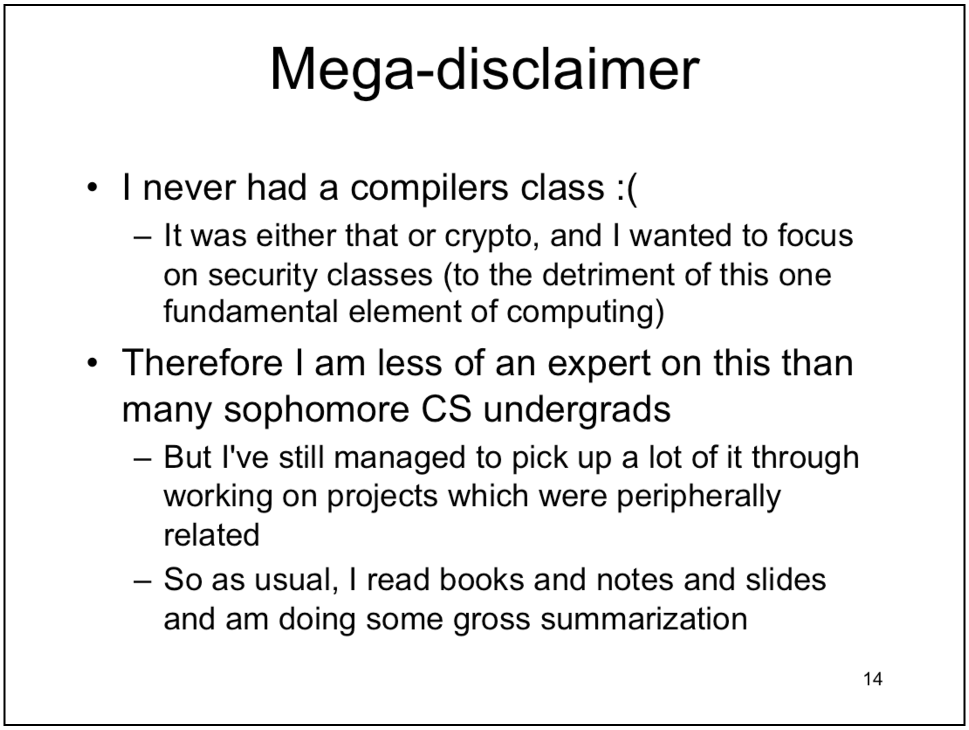


Class Scope 2

- Personally I've used the knowledge extensively for defensive purposes, in order to do memory integrity verification.
- While we are covering the entire lifecycle, so that people can get the complete picture, the emphasis will be more on the point at which a program becomes formatted according to some well-defined executable binary specification.

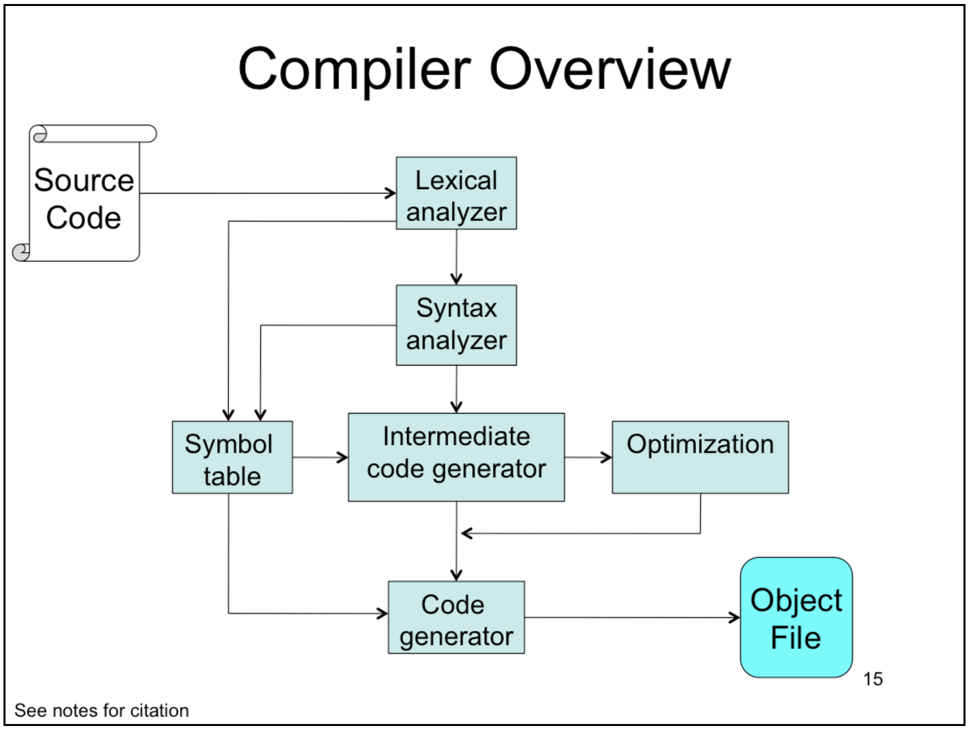


Life of Binaries Overview



Mega-disclaimer

- I never had a compilers class :(
  - It was either that or crypto, and I wanted to focus on security classes (to the detriment of this one fundamental element of computing)
- Therefore I am less of an expert on this than many sophomore CS undergrads
  - But I've still managed to pick up a lot of it through working on projects which were peripherally related
  - So as usual, I read books and notes and slides and am doing some gross summarization



Compiler Overview

Source Code

Lexical analyzer

Syntax analyzer

Symbol table

Intermediate code generator

Optimization

Code generator

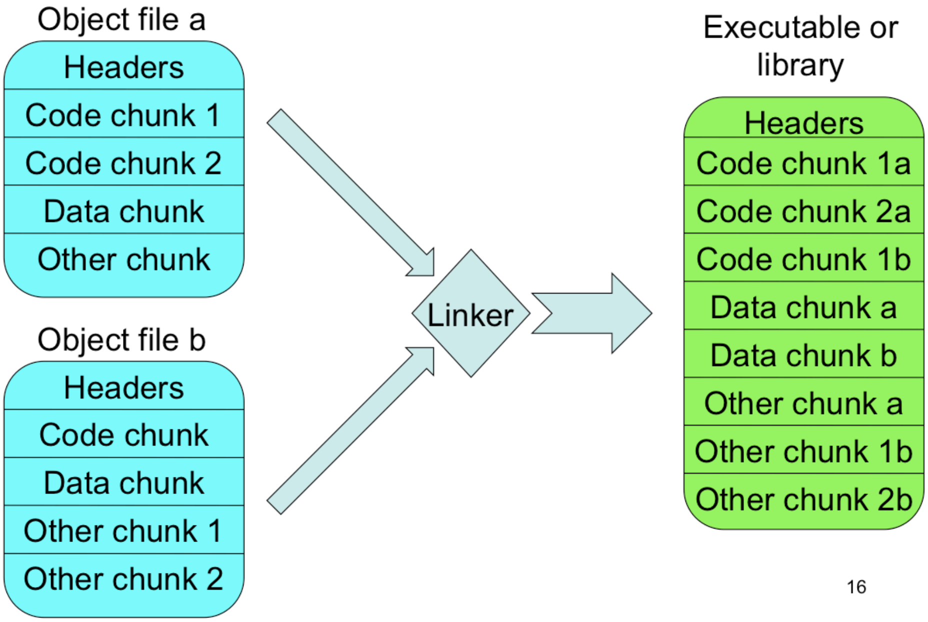
Object File

15

See notes for citation

Diagram outline adapted from "Concepts of Programming Languages 4th ed." Figure 1.3 page 28

Linker Overview



Object file a

- Headers
- Code chunk 1
- Code chunk 2
- Data chunk
- Other chunk

Object file b

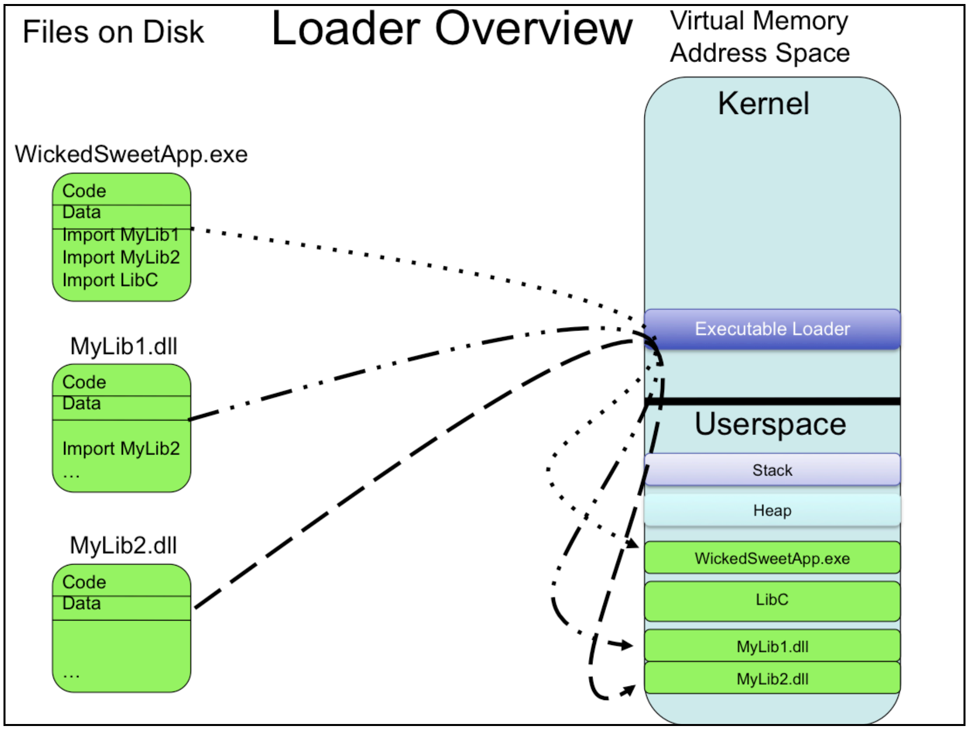
- Headers
- Code chunk
- Data chunk
- Other chunk 1
- Other chunk 2

Linker

Executable or library

- Headers
- Code chunk 1a
- Code chunk 2a
- Code chunk 1b
- Data chunk a
- Data chunk b
- Other chunk a
- Other chunk 1b
- Other chunk 2b

Loader Overview



Files on Disk

Virtual Memory Address Space

WickedSweetApp.exe

- Code
- Data
- Import MyLib1
- Import MyLib2
- Import LibC

MyLib1.dll

- Code
- Data
- Import MyLib2
- …

MyLib2.dll

- Code
- Data
- …

Kernel

Executable Loader

Userspace

Stack

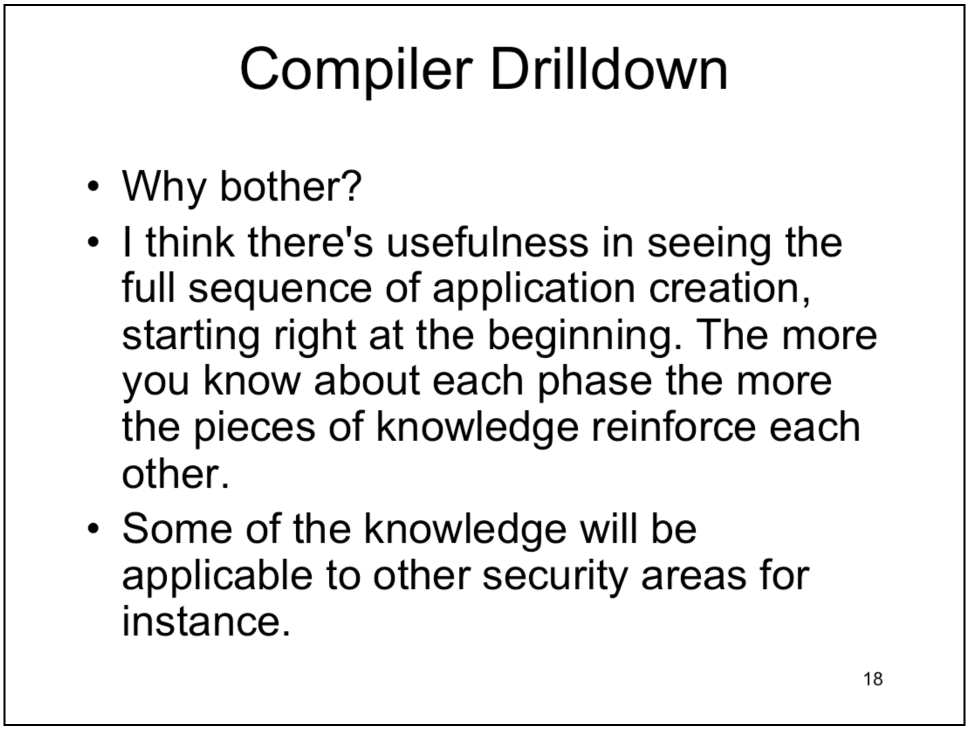
Heap

WickedSweetApp.exe

LibC

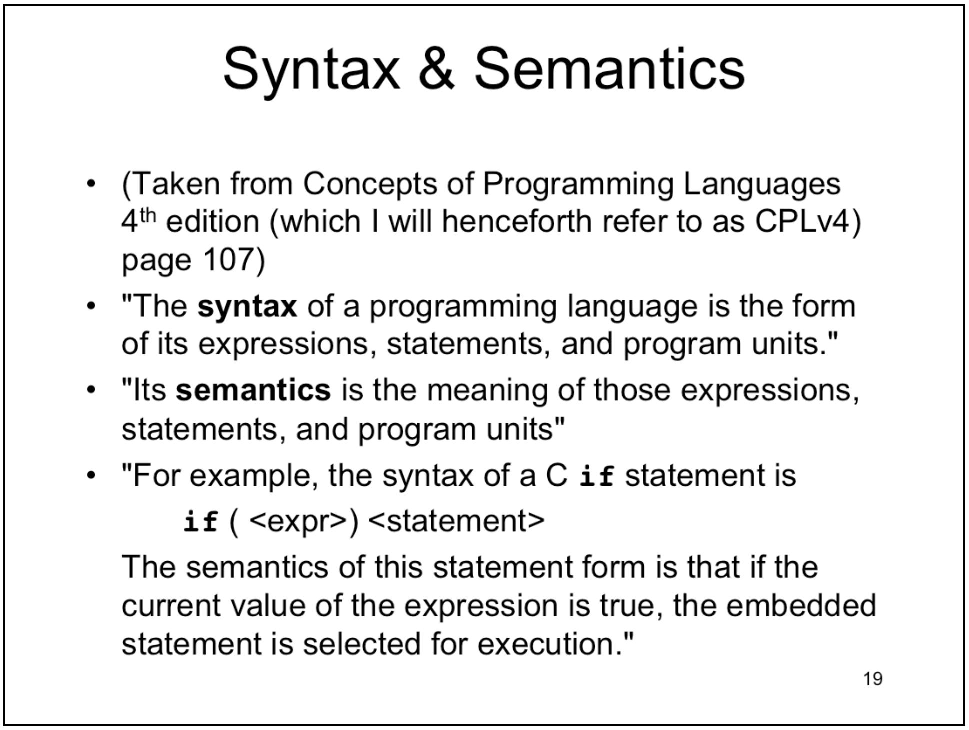
MyLib1.dll

MyLib2.dll



Compiler Drilldown

- Why bother?
- I think there's usefulness in seeing the full sequence of application creation, starting right at the beginning. The more you know about each phase the more the pieces of knowledge reinforce each other.
- Some of the knowledge will be applicable to other security areas for instance.

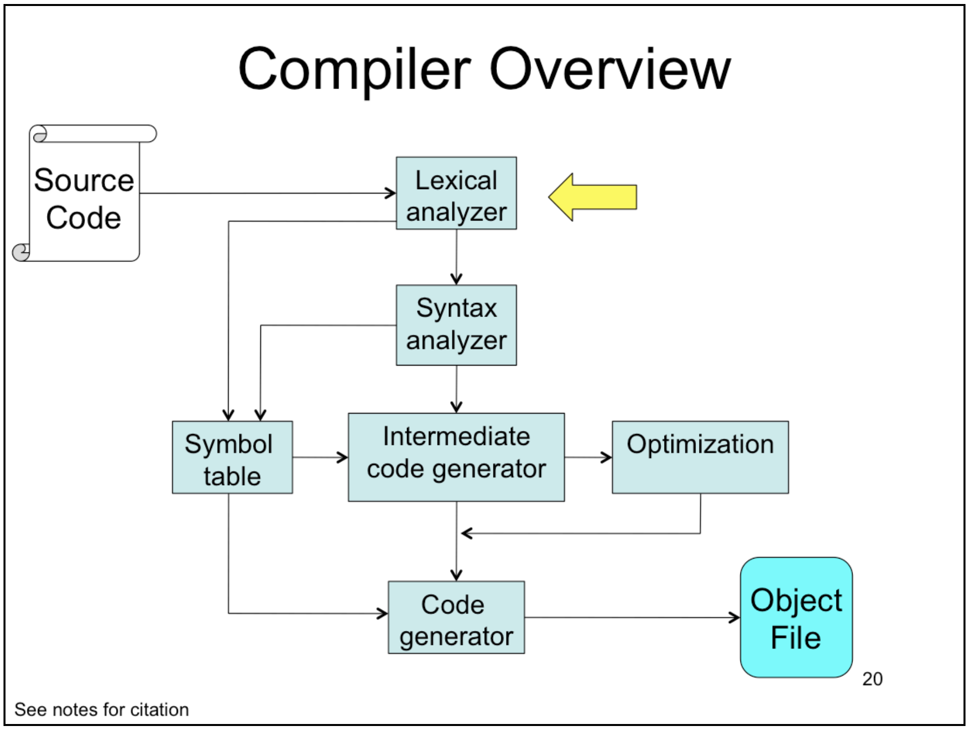


Syntax & Semantics

- (Taken from Concepts of Programming Languages 4th edition (which I will henceforth refer to as CPLv4) page 107)
- "The **syntax** of a programming language is the form of its expressions, statements, and program units."
- "Its **semantics** is the meaning of those expressions, statements, and program units"
- "For example, the syntax of a C `if` statement is

  `if` (<expr>) <statement>

  The semantics of this statement form is that if the current value of the expression is true, the embedded statement is selected for execution."



Compiler Overview

Source Code

Lexical analyzer

Syntax analyzer

Symbol table

Intermediate code generator

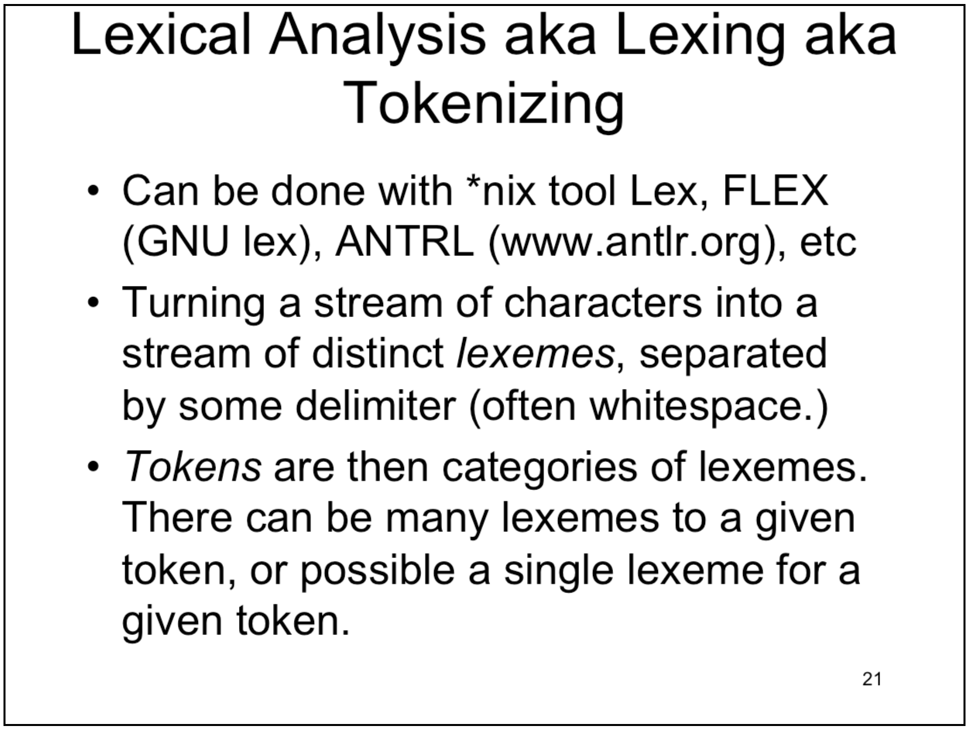
Optimization

Code generator

Object File

See notes for citation

Diagram outline adapted from "Concepts of Programming Languages 4th ed." Figure 1.3 page 28



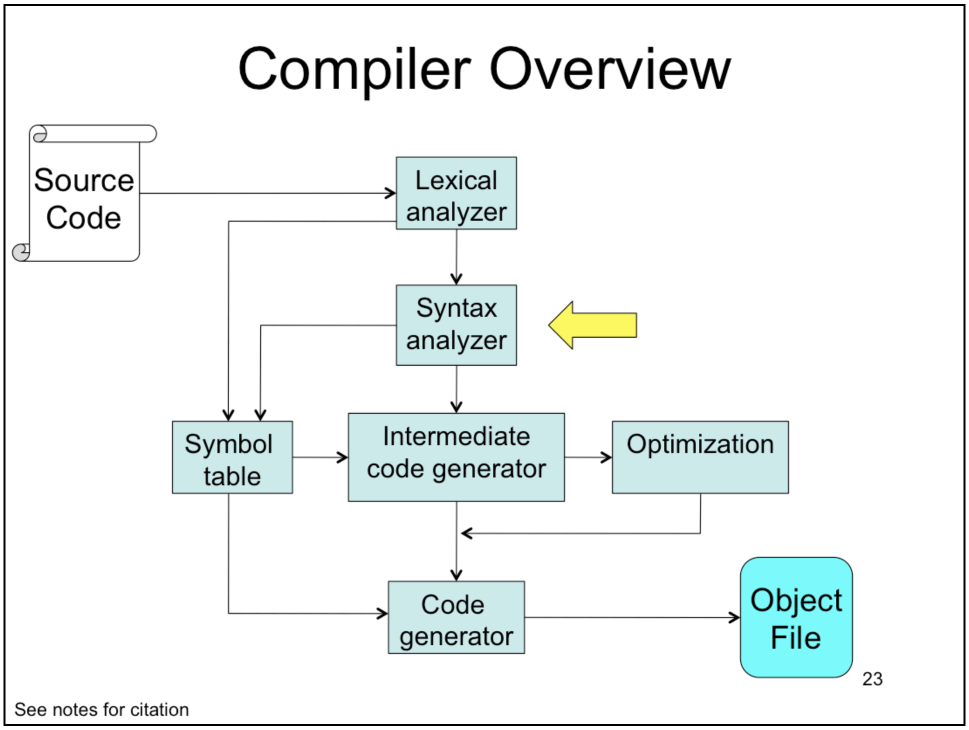
Lexical Analysis aka Lexing aka Tokenizing

- Can be done with *nix tool Lex, FLEX (GNU lex), ANTRL (www.antlr.org), etc
- Turning a stream of characters into a stream of distinct *lexemes*, separated by some delimiter (often whitespace.)
- *Tokens* are then categories of lexemes. There can be many lexemes to a given token, or possible a single lexeme for a given token.

lexemes and tokens

- For the following statement (taken from CPLv4 pgs. 107/108)
- index = 2 * count + 17;
- The lexemes and tokens might be:

lexeme	token
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon



Compiler Overview

Source Code

Lexical analyzer

Syntax analyzer

Symbol table

Intermediate code generator

Optimization

Code generator

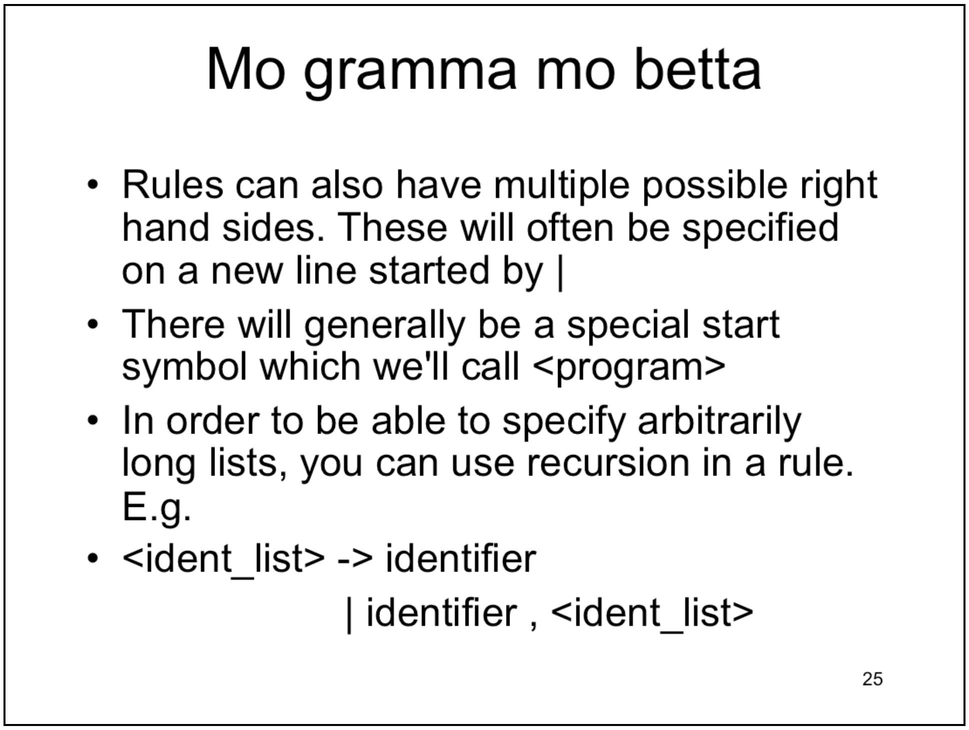
Object File

See notes for citation

Diagram outline adapted from "Concepts of Programming Languages 4th ed." Figure 1.3 page 28

Syntactic Analysis & Context Free Grammars (CFGs)

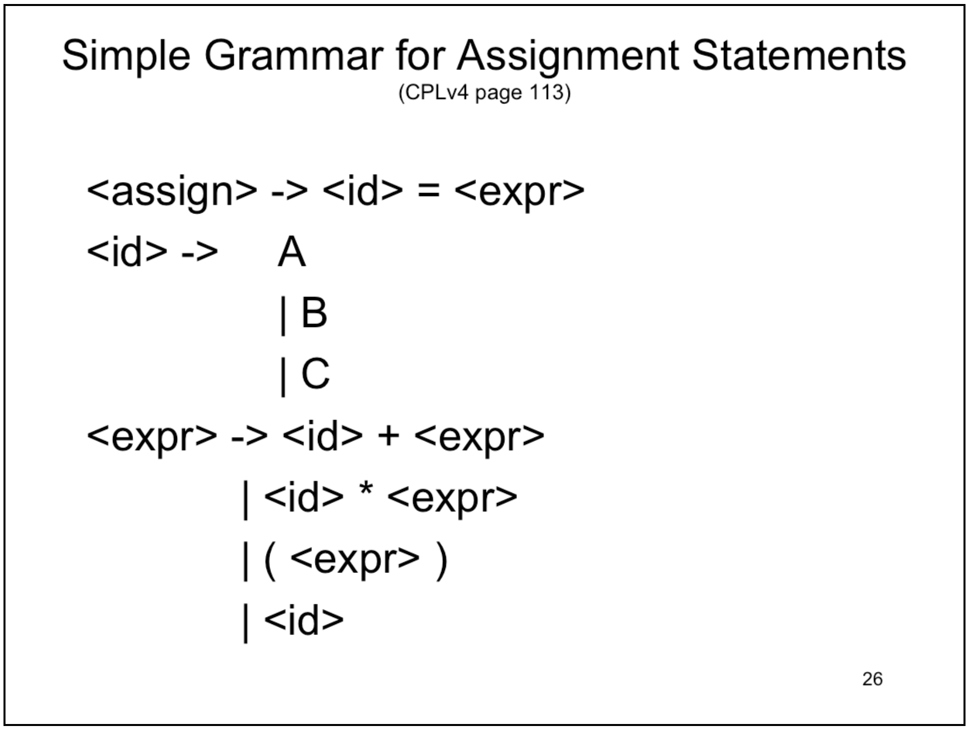
- Done with tools like YACC (yet another compiler compiler), Bison (GNU yacc), ANTLR, or CUP (for java)
- A way to formally specify a syntax
- A commonly used form is Backus-Naur form (BNF)
- A grammar in BNF will be a series of rules, composed of terminal symbols, and non-terminal symbols. An example rule for an assignment statement might look like:
- <assign> -> <var> = <expression>
- The -> will be used to indicate that the symbol on the left hand side can be represented by the statement on the right hand side.
- The < and > are used to enclose a non-terminal
- In the above the = is a terminal. Terminals can also be given by tokens.
- Of course, for the above to be of any use, you then need rules to describe how <var> and <expression> are formed



Mo gramma mo betta

- Rules can also have multiple possible right hand sides. These will often be specified on a new line started by |
- There will generally be a special start symbol which we'll call <program>
- In order to be able to specify arbitrarily long lists, you can use recursion in a rule. E.g.
-
  ```
  <ident_list> -> identifier
                | identifier , <ident_list>
  ```

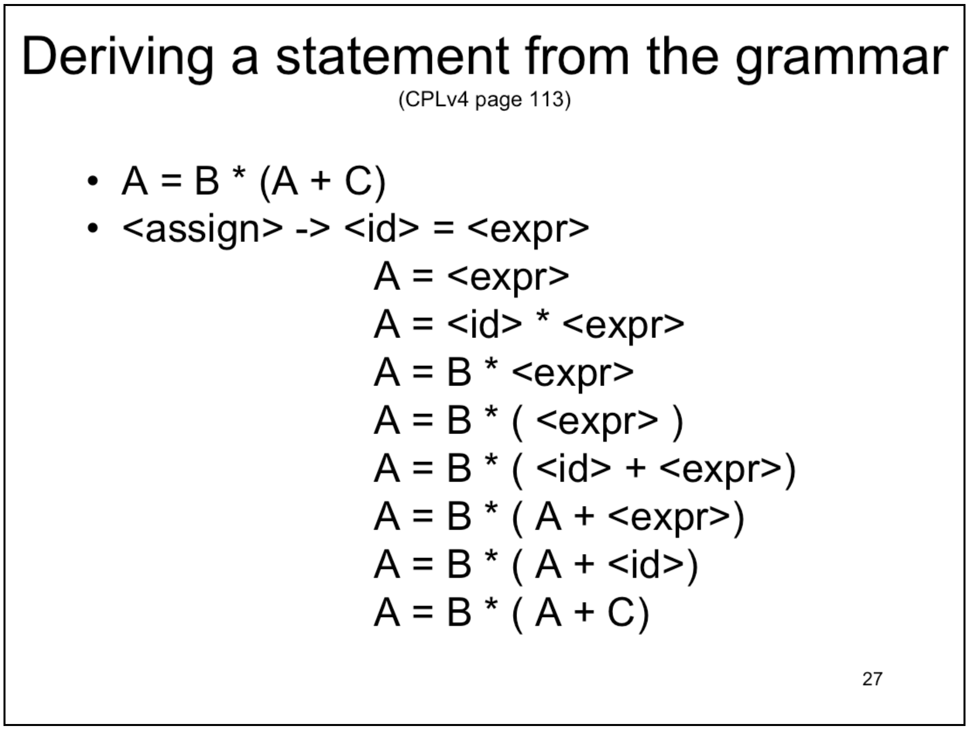
25



Simple Grammar for Assignment Statements

(CPLv4 page 113)

```
<assign> -> <id> = <expr>
<id> ->     A
            | B
            | C
<expr> -> <id> + <expr>
         | <id> * <expr>
         | ( <expr> )
         | <id>
```



Deriving a statement from the grammar

(CPLv4 page 113)

- A = B * (A + C)
- <assign> -> <id> = <expr>

A = <expr>

A = <id> * <expr>

A = B * <expr>

A = B * (<expr>)

A = B * (<id> + <expr>)

A = B * (A + <expr>)

A = B * (A + <id>)

A = B * (A + C)

Parse Trees

(aka Concrete Syntax Tree)
(Picture from CPLv4 page 114)

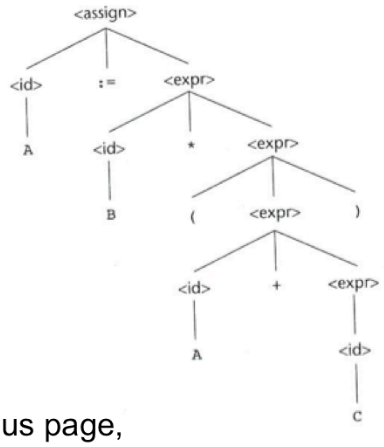


Figure 3.1
A parse tree for the simple statement
`A := B * (A + C)`

This is the parse tree for
the sequence on the previous page,
except I changed stuff to = from :=

Syntax Graphs

(from http://www.json.org)

<number>
- <int>
- | <int> <frac>
- int exp
- int frac exp

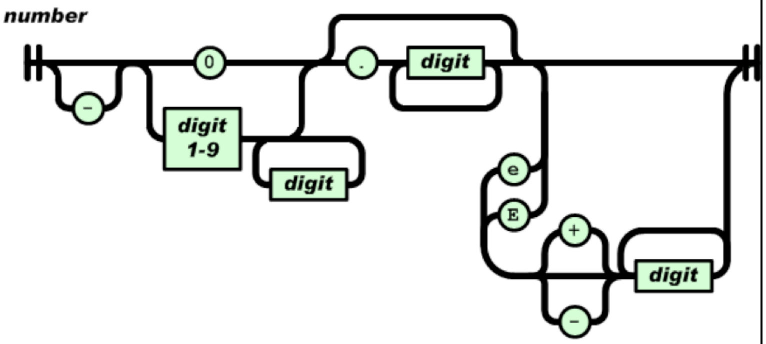
<int>
- digit
- digit1-9 digits
- - digit
- - digit1-9 digits

<frac>
- . digits

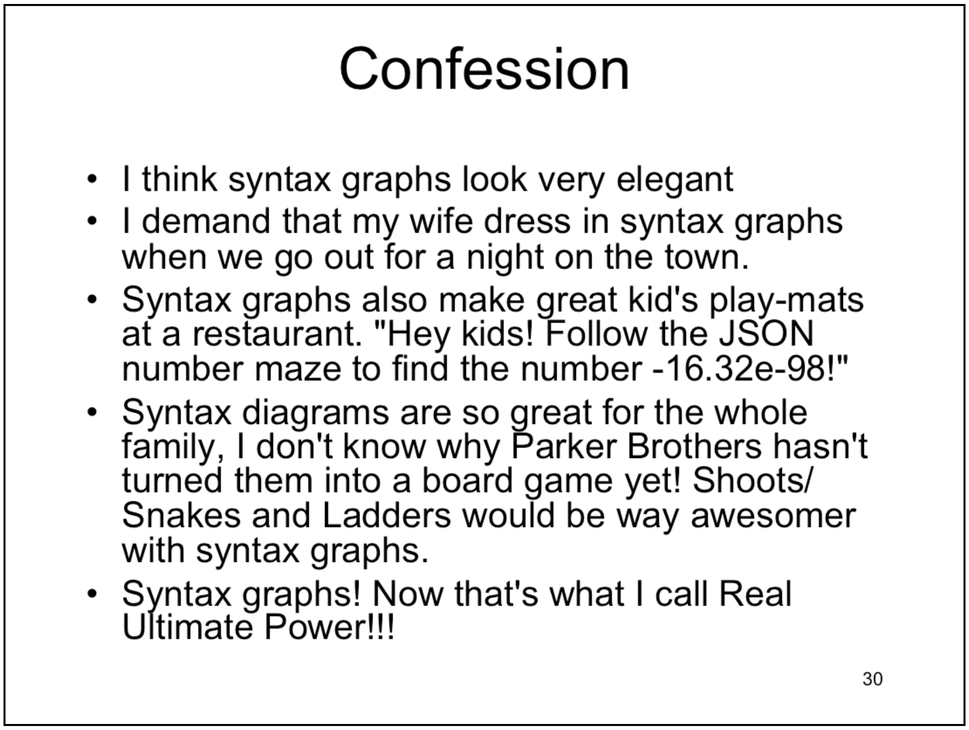
exp
- e digits

digits
- digit
- digit digits

e
- e
- e+
- e-
- E
- E+
- E-

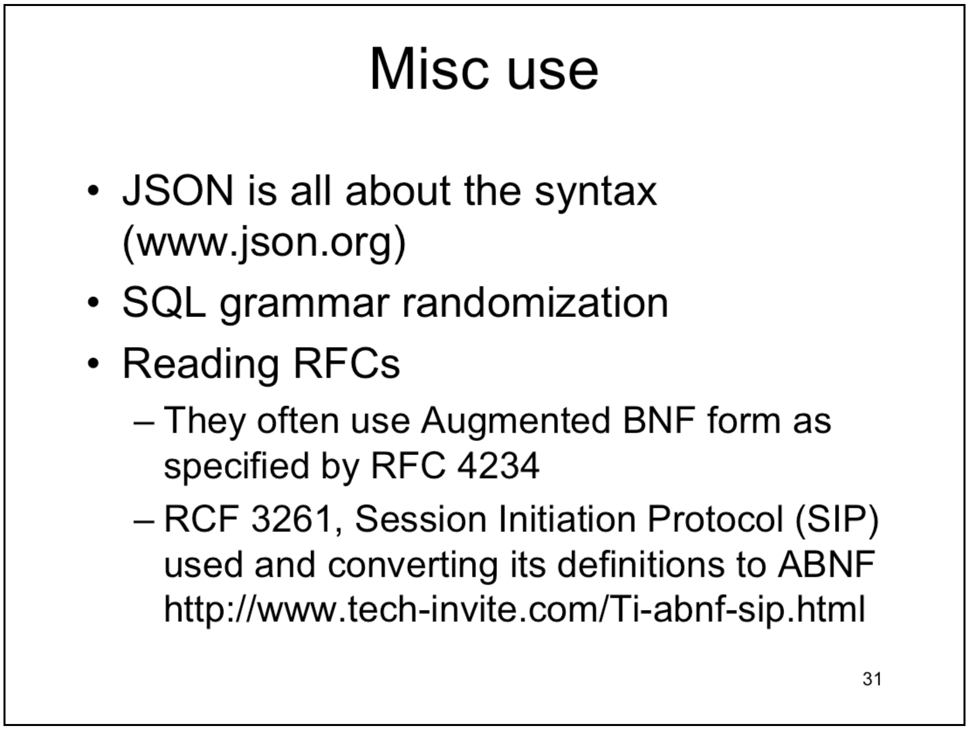


I just did a search for "futuristic rifle" and got this :D
JSON Number syntax graph and FNH FS2000 -
Separated at birth?
From http://en.wikipedia.org/wiki/FN_F2000



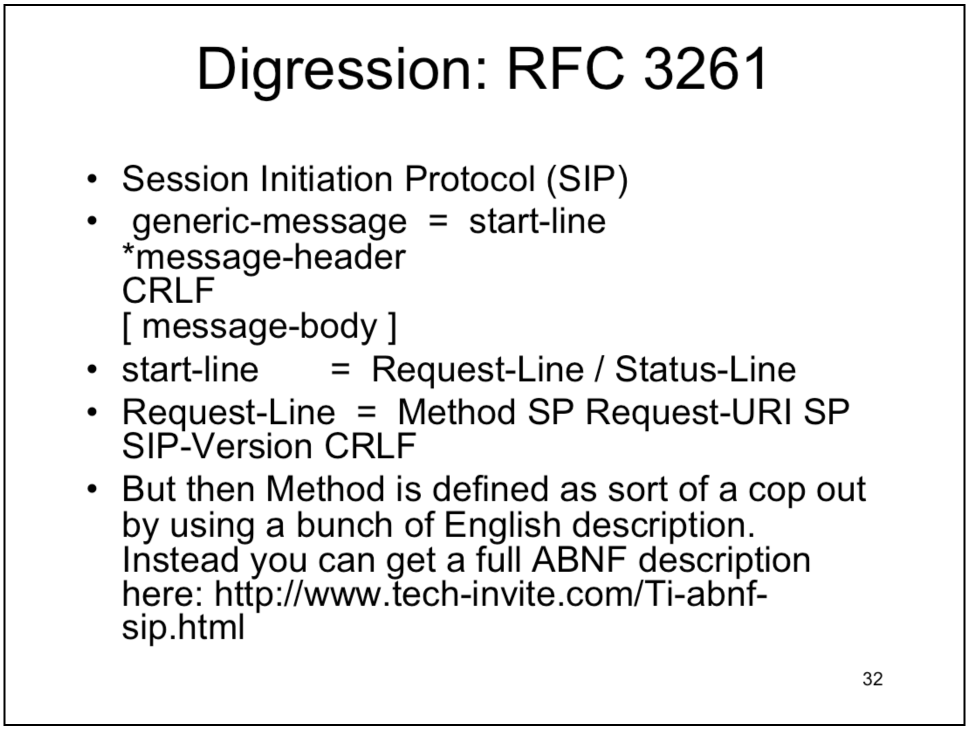
Confession

- I think syntax graphs look very elegant
- I demand that my wife dress in syntax graphs when we go out for a night on the town.
- Syntax graphs also make great kid's play-mats at a restaurant. "Hey kids! Follow the JSON number maze to find the number -16.32e-98!"
- Syntax diagrams are so great for the whole family, I don't know why Parker Brothers hasn't turned them into a board game yet! Shoots/ Snakes and Ladders would be way awesomer with syntax graphs.
- Syntax graphs! Now that's what I call Real Ultimate Power!!!



Misc use

- JSON is all about the syntax (www.json.org)
- SQL grammar randomization
- Reading RFCs
  - – They often use Augmented BNF form as specified by RFC 4234
  - – RCF 3261, Session Initiation Protocol (SIP) used and converting its definitions to ABNF http://www.tech-invite.com/Ti-abnf-sip.html



Digression: RFC 3261

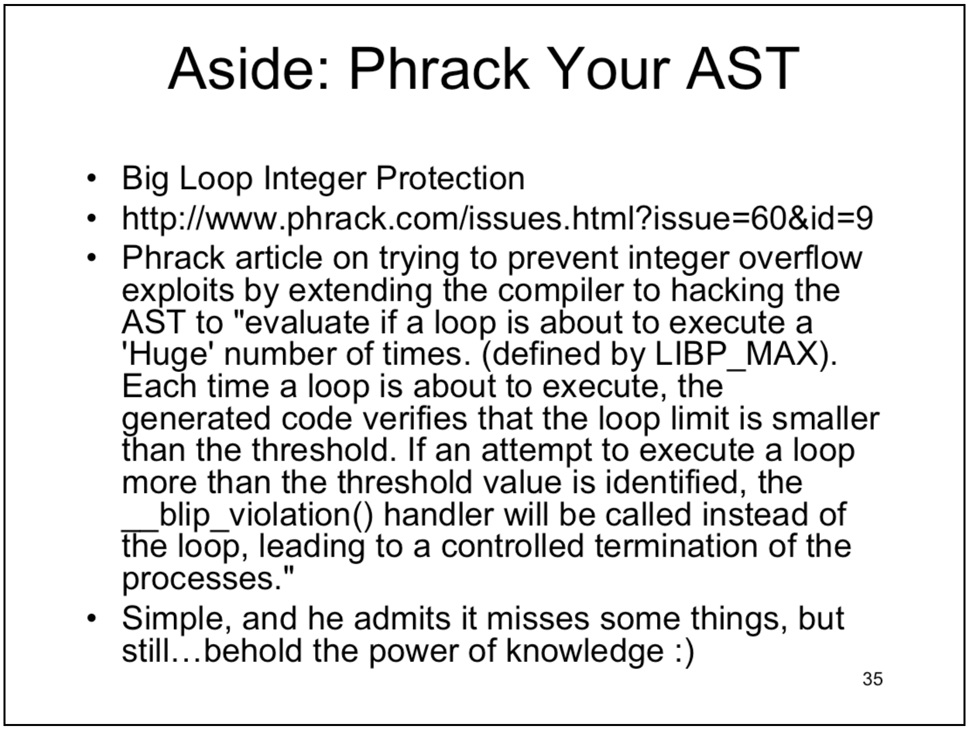
- Session Initiation Protocol (SIP)
- generic-message = start-line
  *message-header
  CRLF
  [message-body]
- start-line = Request-Line / Status-Line
- Request-Line = Method SP Request-URI SP SIP-Version CRLF
- But then Method is defined as sort of a cop out by using a bunch of English description. Instead you can get a full ABNF description here: http://www.tech-invite.com/Ti-abnf-sip.html

More Info on Source to assembly to object file transition

- Take a compilers class :P
- http://www.cs.usfca.edu/~galles/compilerdesign/cimplementation.pdf
- http://en.wikipedia.org/wiki/Dragon_Book_(computer_science)

Abstract Syntax Trees (ASTs)

- ASTs are a condensed/simplified form of the parse tree where the operators are internal nodes and never leaves.
- ASTs are more convenient form for subsequent stages to work with.
- Rather than having a grammar parser which just generates a parse tree and then converts it to an AST, you can create a parser which converts the input directly to an AST. This is called syntax-directed translation (and it's the shield the knight on the Dragon Book is holding ;)). In order to do that, the parser needs to be able to call some code to perform some action when it recognizes things.



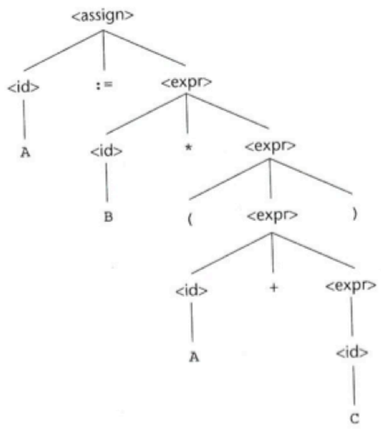
Aside: Phrack Your AST

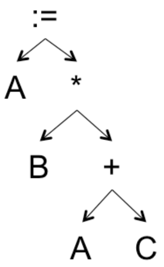
- Big Loop Integer Protection
- http://www.phrack.com/issues.html?issue=60&id=9
- Phrack article on trying to prevent integer overflow exploits by extending the compiler to hacking the AST to "evaluate if a loop is about to execute a 'Huge' number of times. (defined by LIBP_MAX). Each time a loop is about to execute, the generated code verifies that the loop limit is smaller than the threshold. If an attempt to execute a loop more than the threshold value is identified, the __blip_violation() handler will be called instead of the loop, leading to a controlled termination of the processes."
- Simple, and he admits it misses some things, but still…behold the power of knowledge :)

AST Vs. Parse Tree

Figure 3.1
A parse tree for the simple statement
`A := B * (A + C)`

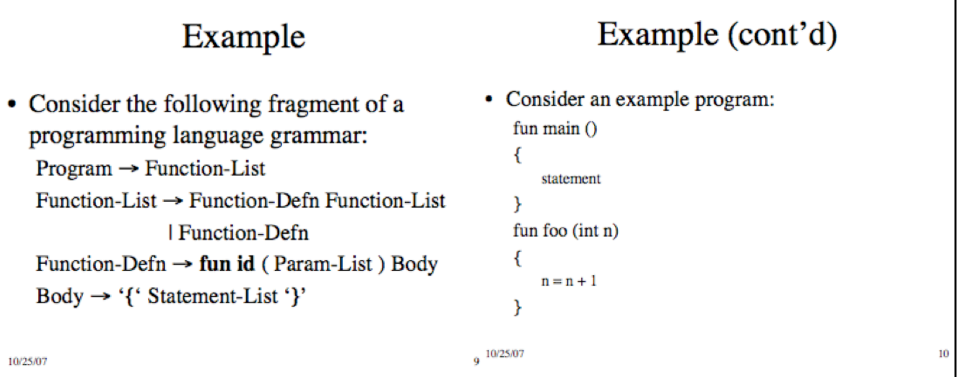
(CPLv4 page 114)





AST

A slightly more complex example before we move on



Example

- Consider the following fragment of a programming language grammar:

  Program → Function-List
  Function-List → Function-Defn Function-List
  | Function-Defn
  Function-Defn → **fun id** (Param-List) Body
  Body → '{' Statement-List '}'

10/25/07 9

Example (cont'd)

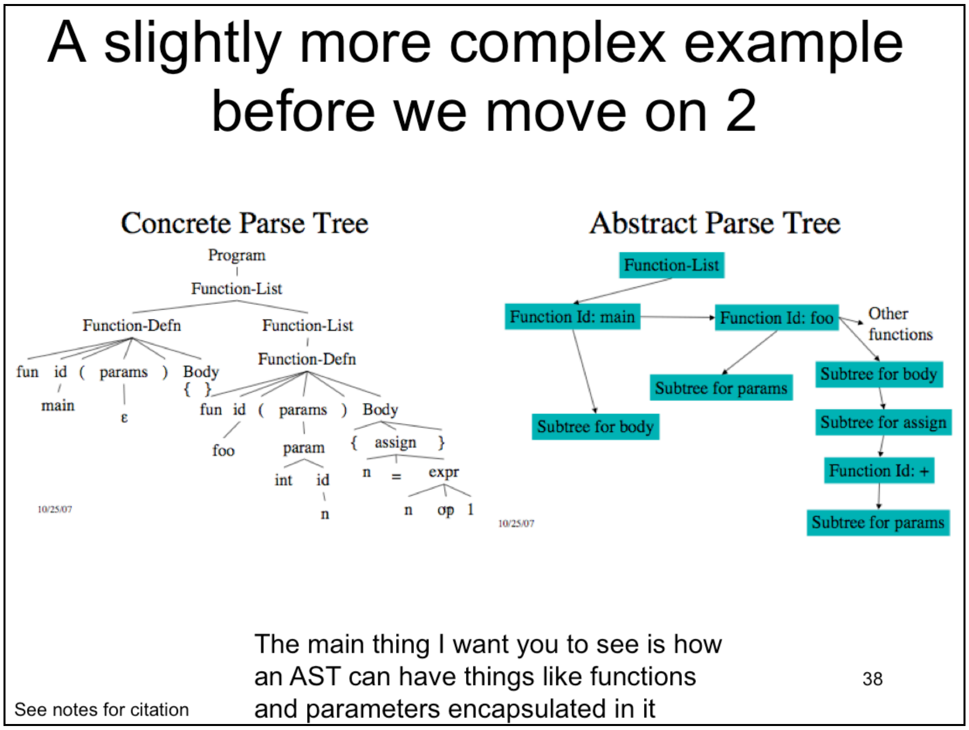
- Consider an example program:

```
fun main ()
{
    statement
}
fun foo (int n)
{
    n = n + 1
}
```

10/25/07 10

See notes for citation

from http://www.cs.sfu.ca/~anoop/courses/CMPT-379-Fall-2007/abstract.pdf



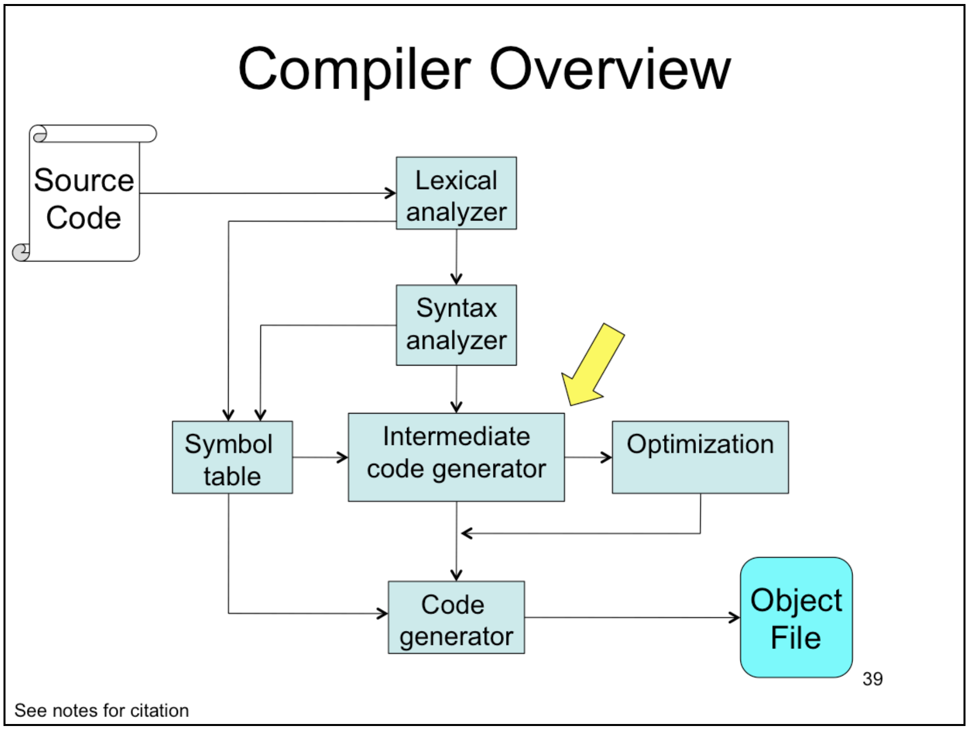
A slightly more complex example before we move on 2

The main thing I want you to see is how an AST can have things like functions and parameters encapsulated in it

See notes for citation

38

from http://www.cs.sfu.ca/~anoop/courses/CMPT-379-Fall-2007/abstract.pdf



Compiler Overview

Source Code

Lexical analyzer

Syntax analyzer

Symbol table

Intermediate code generator

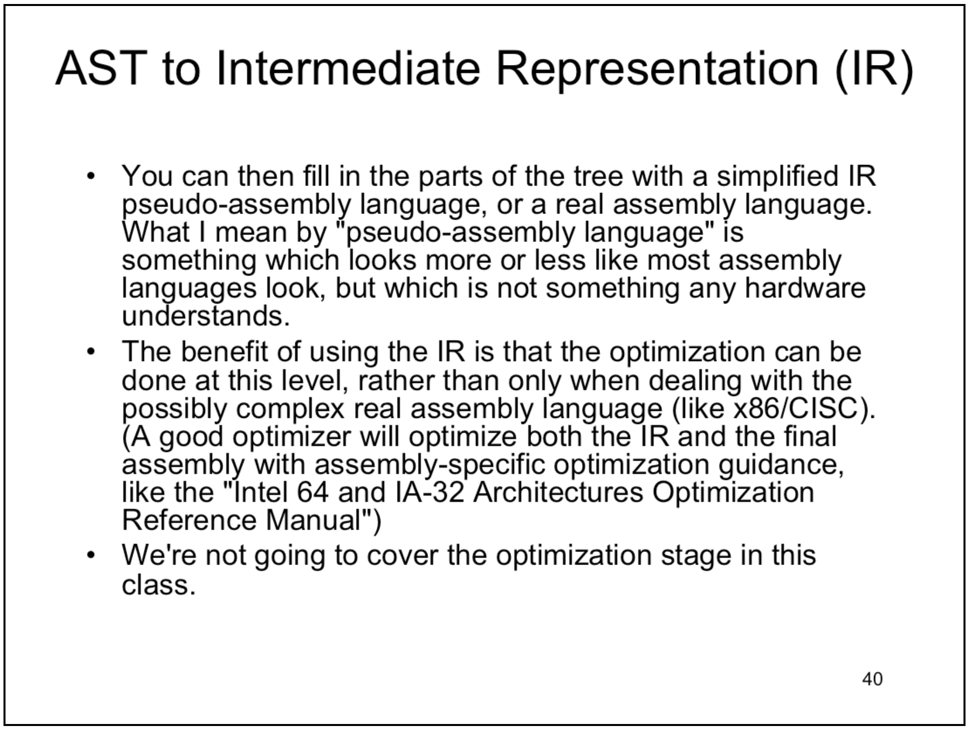
Optimization

Code generator

Object File

See notes for citation

Diagram outline adapted from "Concepts of Programming Languages 4th ed." Figure 1.3 page 28



AST to Intermediate Representation (IR)

- You can then fill in the parts of the tree with a simplified IR pseudo-assembly language, or a real assembly language. What I mean by "pseudo-assembly language" is something which looks more or less like most assembly languages look, but which is not something any hardware understands.
- The benefit of using the IR is that the optimization can be done at this level, rather than only when dealing with the possibly complex real assembly language (like x86/CISC). (A good optimizer will optimize both the IR and the final assembly with assembly-specific optimization guidance, like the "Intel 64 and IA-32 Architectures Optimization Reference Manual")
- We're not going to cover the optimization stage in this class.

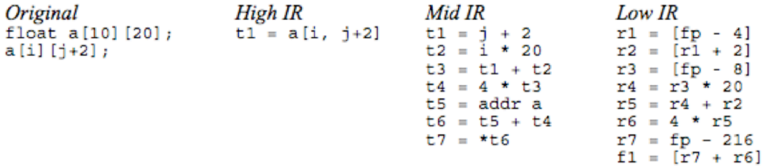
Different Levels of IR

- http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf

Fairly language dependent

Ideally language and machine independent

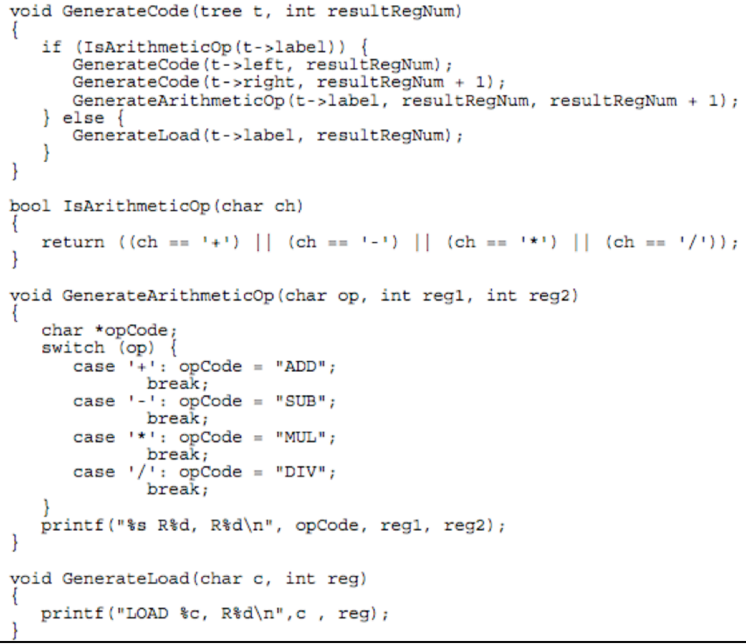
Fairly machine dependent



```
Original             High IR             Mid IR            Low IR
float a[10][20];     t1 = a[i, j+2]      t1 = j + 2        r1 = [fp - 4]
a[i][j+2];                               t2 = i * 20       r2 = [r1 + 2]
                                         t3 = t1 + t2      r3 = [fp - 8]
                                         t4 = 4 * t3       r4 = r3 * 20
                                         t5 = addr a       r5 = r4 + r2
                                         t6 = t5 + t4      r6 = 4 * r5
                                         t7 = *t6          r7 = fp - 216
                                                           f1 = [r7 + r6]
```

Code to Generate IR from AST

http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf

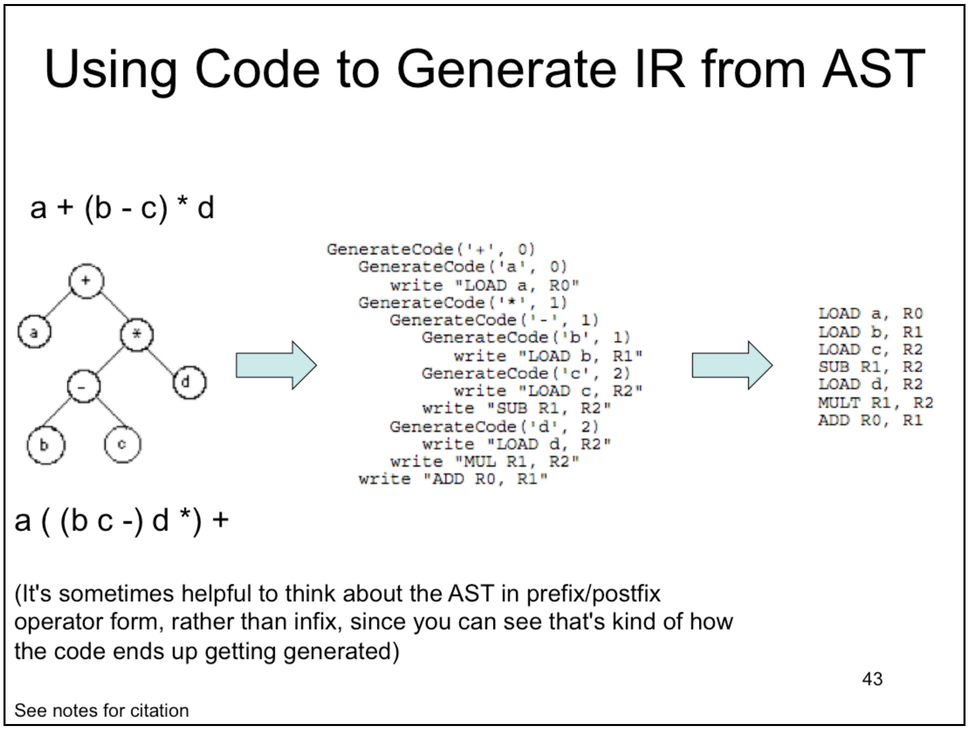


```
void GenerateCode(tree t, int resultRegNum)
{
   if (IsArithmeticOp(t->label)) {
      GenerateCode(t->left, resultRegNum);
      GenerateCode(t->right, resultRegNum + 1);
      GenerateArithmeticOp(t->label, resultRegNum, resultRegNum + 1);
   } else {
      GenerateLoad(t->label, resultRegNum);
   }
}

bool IsArithmeticOp(char ch)
{
   return ((ch == '+') || (ch == '-') || (ch == '*') || (ch == '/'));
}

void GenerateArithmeticOp(char op, int reg1, int reg2)
{
   char *opCode;
   switch (op) {
      case '+': opCode = "ADD";
             break;
      case '-': opCode = "SUB";
             break;
      case '*': opCode = "MUL";
             break;
      case '/': opCode = "DIV";
             break;
   }
   printf("%s R%d, R%d\n", opCode, reg1, reg2);
}

void GenerateLoad(char c, int reg)
{
   printf("LOAD %c, R%d\n",c , reg);
}
```



Using Code to Generate IR from AST

a + (b - c) * d

```
GenerateCode('+', 0)
   GenerateCode('a', 0)
      write "LOAD a, R0"
   GenerateCode('*', 1)
      GenerateCode('-', 1)
         GenerateCode('b', 1)
            write "LOAD b, R1"
         GenerateCode('c', 2)
            write "LOAD c, R2"
         write "SUB R1, R2"
      GenerateCode('d', 2)
         write "LOAD d, R2"
      write "MUL R1, R2"
   write "ADD R0, R1"
```

```
LOAD a, R0
LOAD b, R1
LOAD c, R2
SUB R1, R2
LOAD d, R2
MULT R1, R2
ADD R0, R1
```

a ((b c -) d *) +

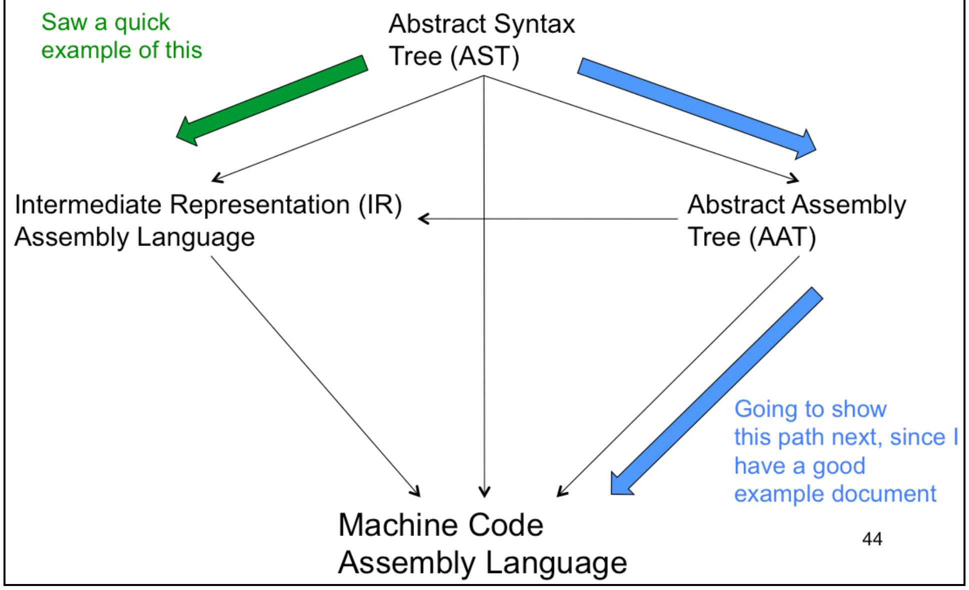
(It's sometimes helpful to think about the AST in prefix/postfix operator form, rather than infix, since you can see that's kind of how the code ends up getting generated)

43

See notes for citation

http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf

Different Paths to Code Generation



Saw a quick example of this

Abstract Syntax Tree (AST)

Intermediate Representation (IR) Assembly Language

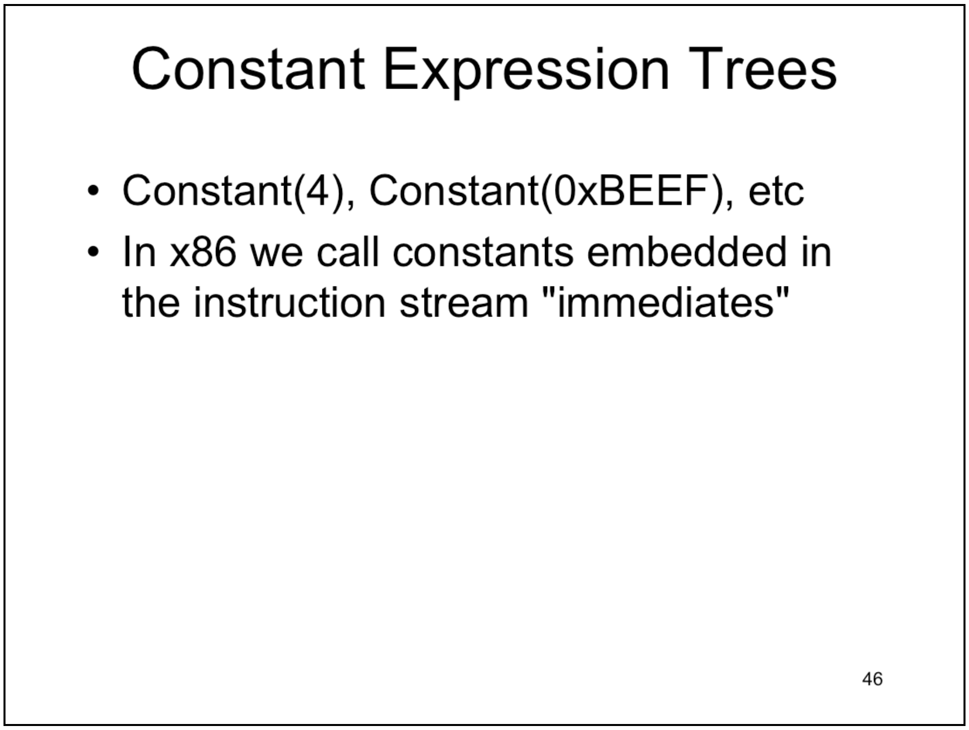
Abstract Assembly Tree (AAT)

Going to show this path next, since I have a good example document

Machine Code Assembly Language

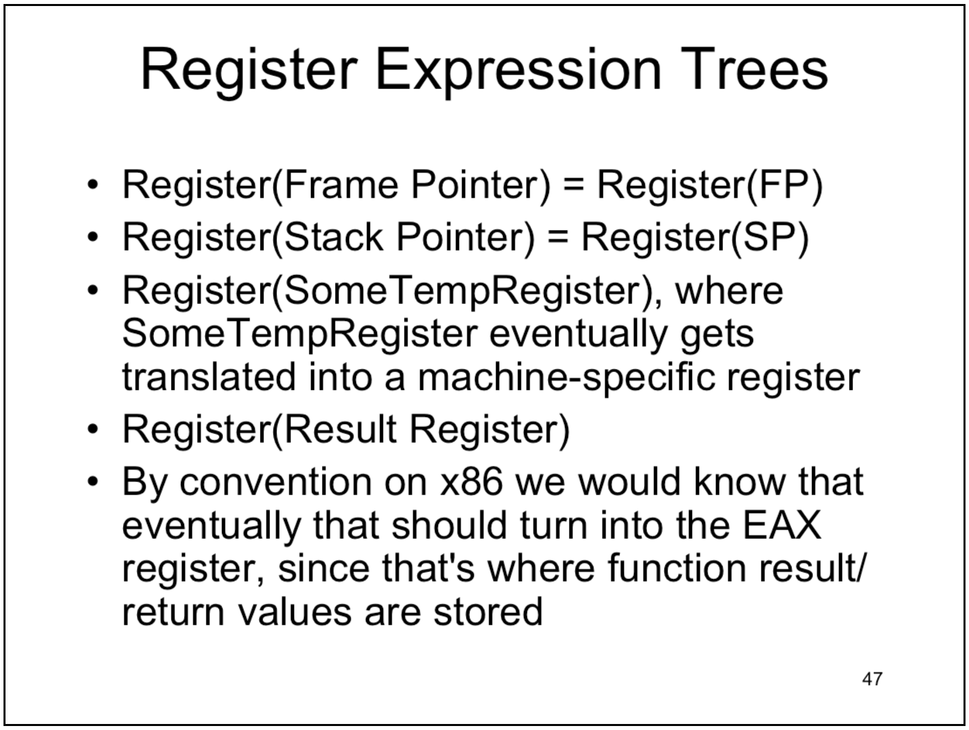
Abstract Assembly Trees (AATs)

- Once you have an AST, it is useful to generate an AAT
- There are two subtypes of an AAT, Expression Trees (for values) and Statement Trees (for actions). You can think of the result of expression trees as being some value which is put on the stack when the tree is processed and collapsed. And the result of statement trees are the organization of which code goes where.
- Highly recommend the more detailed AAT examples here: http://www.uogonline.com/drlee/CS410/Original-Slides/Chap8.java.pdf which I am partially drawing from for the following discussion.



Constant Expression Trees

- Constant(4), Constant(0xBEEF), etc
- In x86 we call constants embedded in the instruction stream "immediates"

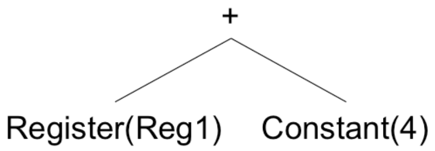


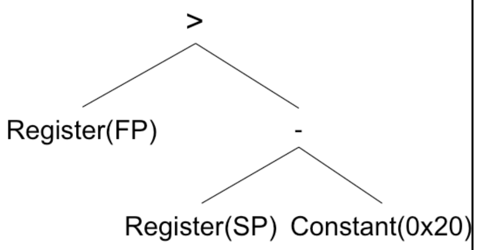
Register Expression Trees

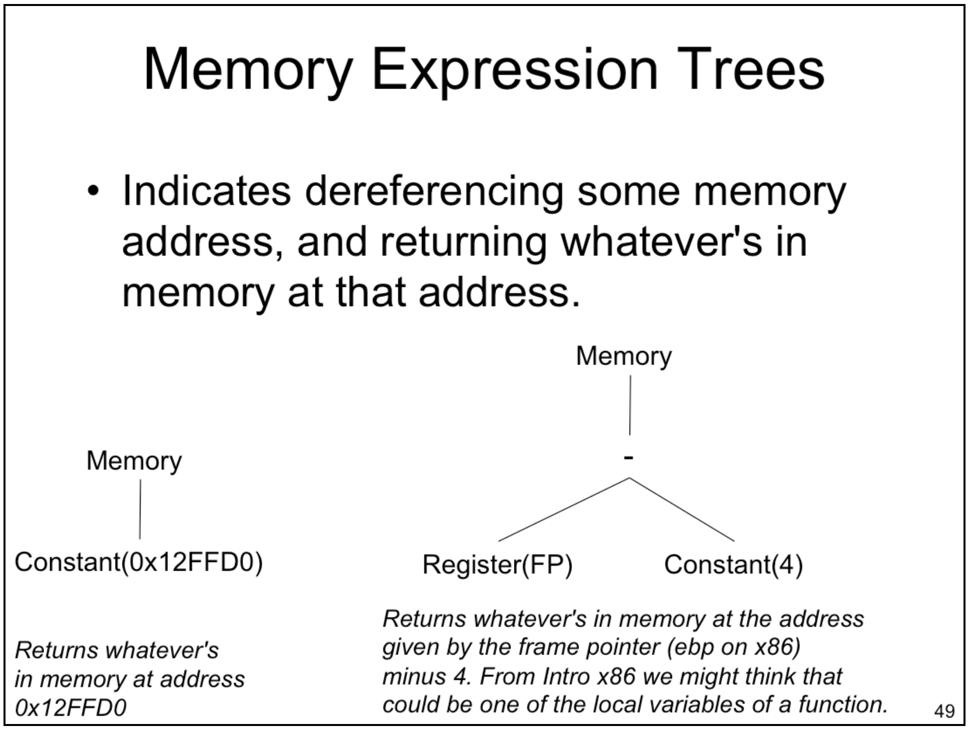
- Register(Frame Pointer) = Register(FP)
- Register(Stack Pointer) = Register(SP)
- Register(SomeTempRegister), where SomeTempRegister eventually gets translated into a machine-specific register
- Register(Result Register)
- By convention on x86 we would know that eventually that should turn into the EAX register, since that's where function result/ return values are stored

Operator Expression Trees

- Now we actually have a tree. The root would be the operator, and the right and left subtrees/leaves are the operands
- Operators are things like +, -, *, /, <, ≤, >, ≥, &&, ||, !, ==







Memory Expression Trees

- Indicates dereferencing some memory address, and returning whatever's in memory at that address.

Memory

Constant(0x12FFD0)

Returns whatever's in memory at address 0x12FFD0

Memory

-

Register(FP)

Constant(4)

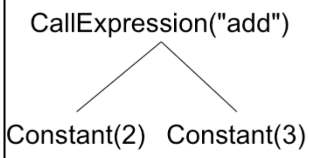
Returns whatever's in memory at the address given by the frame pointer (ebp on x86) minus 4. From Intro x86 we might think that could be one of the local variables of a function.

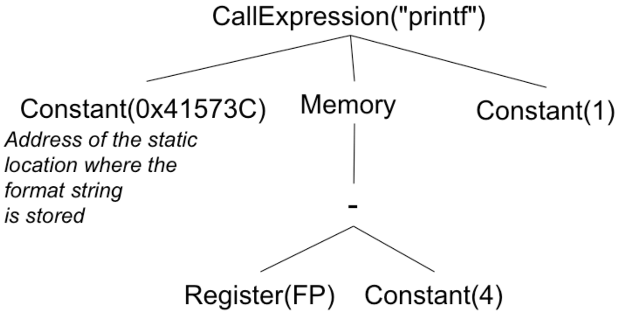
Call Expression Trees

- Shown with the name, and then a sub-expression for each of the input parameters.
- Recall that all expression trees put some value onto the stack, so remember that the return value from the call is put onto the stack.

Example: printf("a = %d, %d", a, 1)

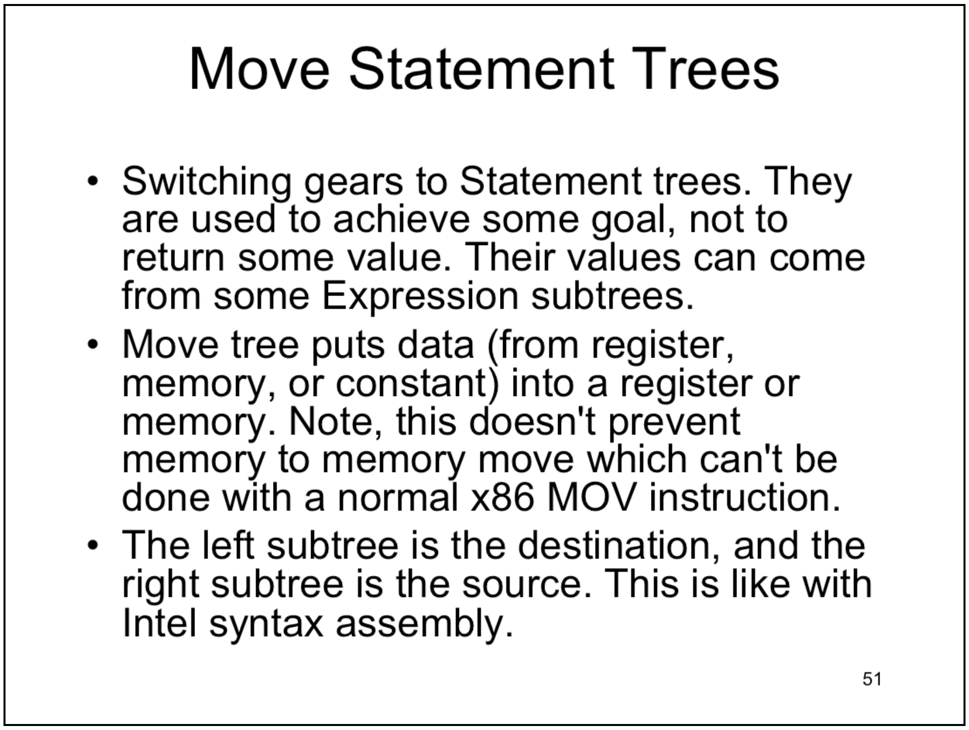
Example: add(2, 3)





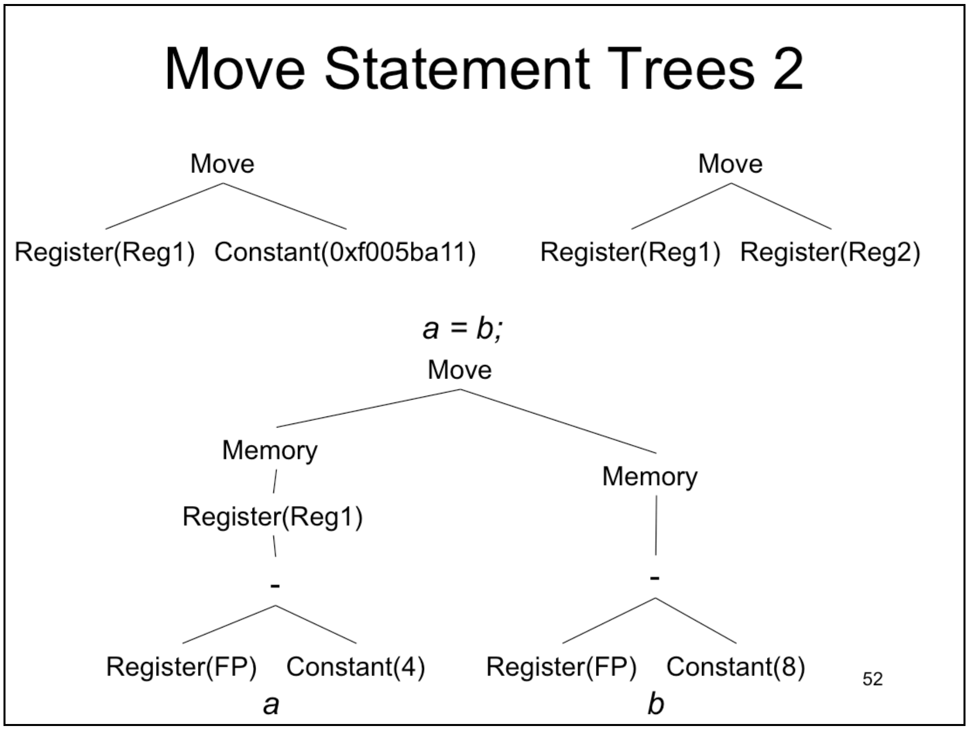
Address of the static location where the format string is stored

Address of the local variable a



Move Statement Trees

- Switching gears to Statement trees. They are used to achieve some goal, not to return some value. Their values can come from some Expression subtrees.
- Move tree puts data (from register, memory, or constant) into a register or memory. Note, this doesn't prevent memory to memory move which can't be done with a normal x86 MOV instruction.
- The left subtree is the destination, and the right subtree is the source. This is like with Intel syntax assembly.



Move Statement Trees 2

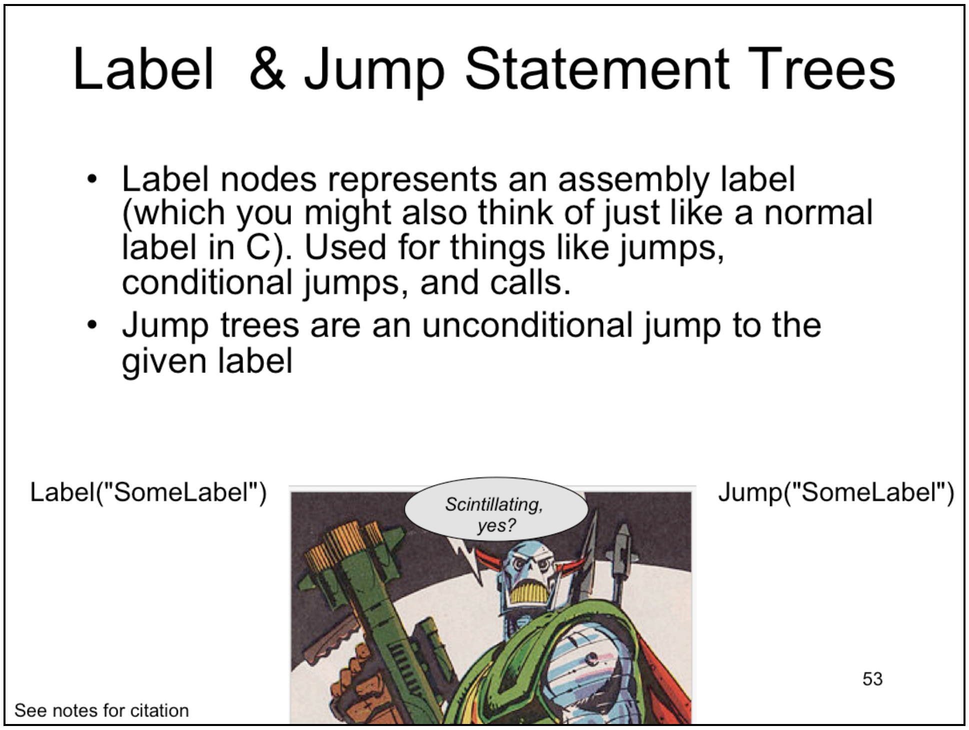
Move
Register(Reg1) Constant(0xf005ba11)

Move
Register(Reg1) Register(Reg2)

a = b;
Move
Memory
Register(Reg1)
-
Register(FP) Constant(4)
a

Memory
-
Register(FP) Constant(8)
b

52



Label & Jump Statement Trees

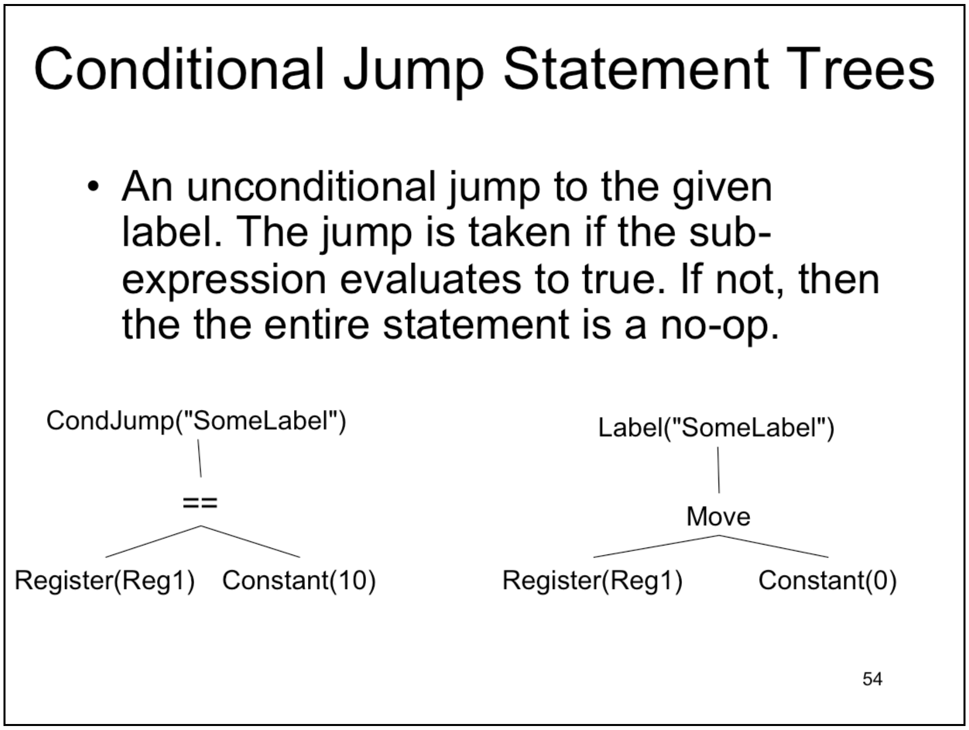
- Label nodes represents an assembly label (which you might also think of just like a normal label in C). Used for things like jumps, conditional jumps, and calls.
- Jump trees are an unconditional jump to the given label

Label("SomeLabel")

Jump("SomeLabel")

See notes for citation

from http://tfwiki.net/wiki/Image:Deathshead113heardofme.jpg



Conditional Jump Statement Trees

- An unconditional jump to the given label. The jump is taken if the sub-expression evaluates to true. If not, then the the entire statement is a no-op.

CondJump("SomeLabel")

==

Register(Reg1) Constant(10)

Label("SomeLabel")

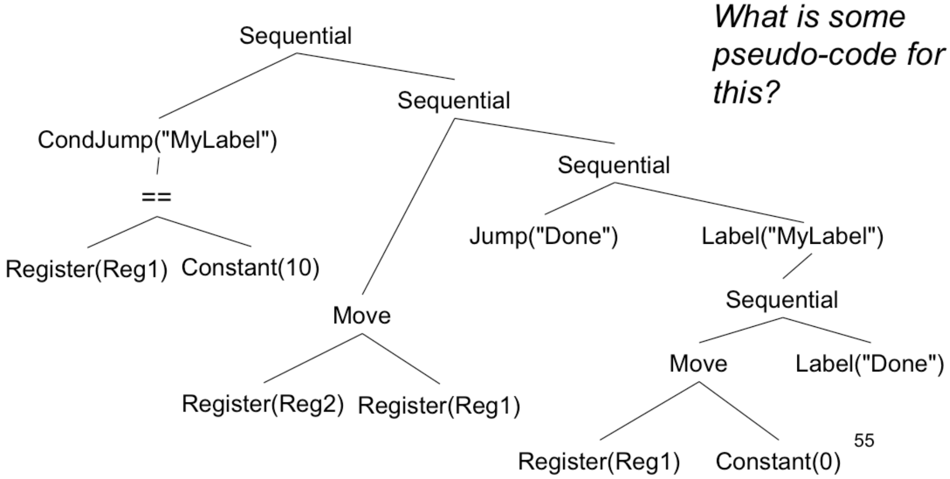
Move

Register(Reg1) Constant(0)

Sequential Statement Trees

- Just execute the left subtree and then the right subtree. Used to maintain ordering of statements. This differs from the AST where whatever was at the leaves furthest from the root was supposed to occur first. But one can see how we would just order from AST to AAT appropriately.

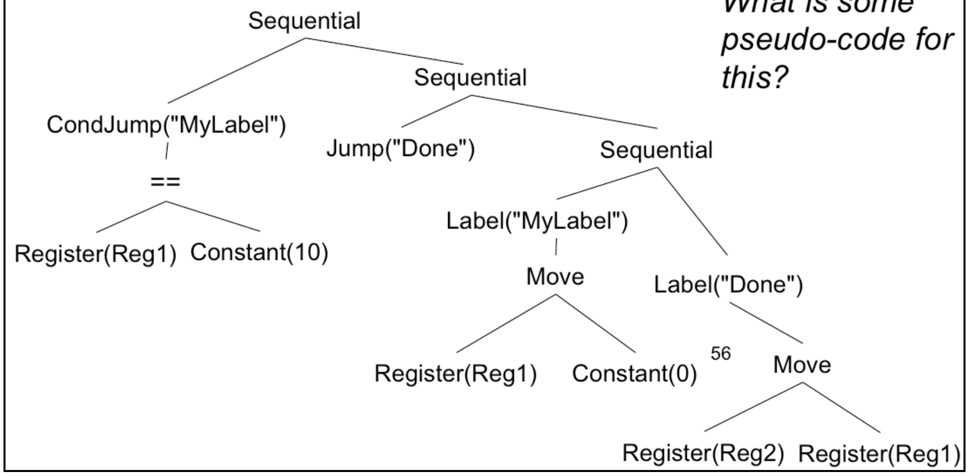
What is some pseudo-code for this?



Sequential Statement Trees

- Just execute the left subtree and then the right subtree. Used to maintain ordering of statements. This differs from the AST where whatever was at the leaves furthest from the root was supposed to occur first. But one can see how we would just order from AST to AAT appropriately.

What is some pseudo-code for this?

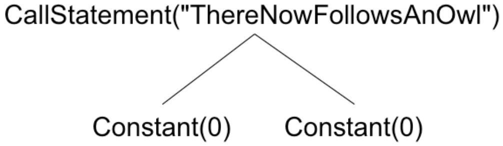


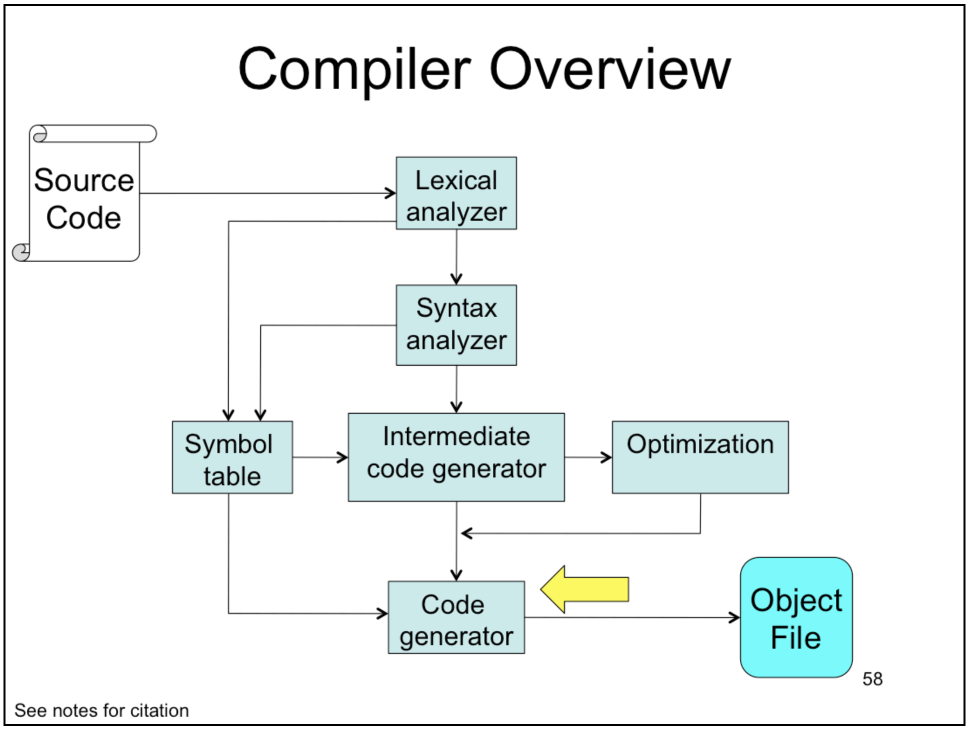
Sequential
CondJump("MyLabel")
==
Register(Reg1)
Constant(10)
Sequential
Jump("Done")
Sequential
Label("MyLabel")
Move
Register(Reg1)
Constant(0)
Label("Done")
Move
Register(Reg2)
Register(Reg1)

Call Statement Trees

- Basically the exact same thing as call expression trees, except that statements don't return a value and expressions do. So you can think of this like calls to functions which return void.
- Now that you've seen Labels, you can think of the function name being called as just a target label.
- Documentation we see later calls the Call Statement a "Procedure Call" and Call Expressions as "Function Call". Meh. I prefer to continue with the statement vs. expression differentiation.

void ThereNowFollowsAnOwl(int a, int b); //declaration/prototype
ThereNowFollowsAnOwl(0,0); //actual use





Compiler Overview

Source Code

Lexical analyzer

Syntax analyzer

Symbol table

Intermediate code generator

Optimization

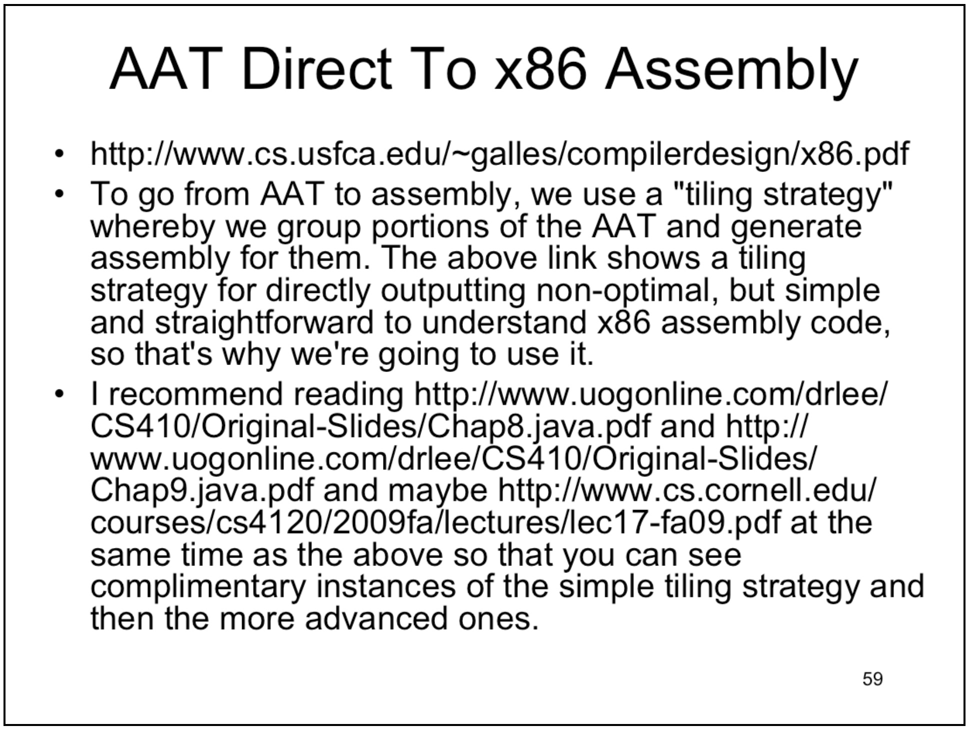
Code generator

Object File

58

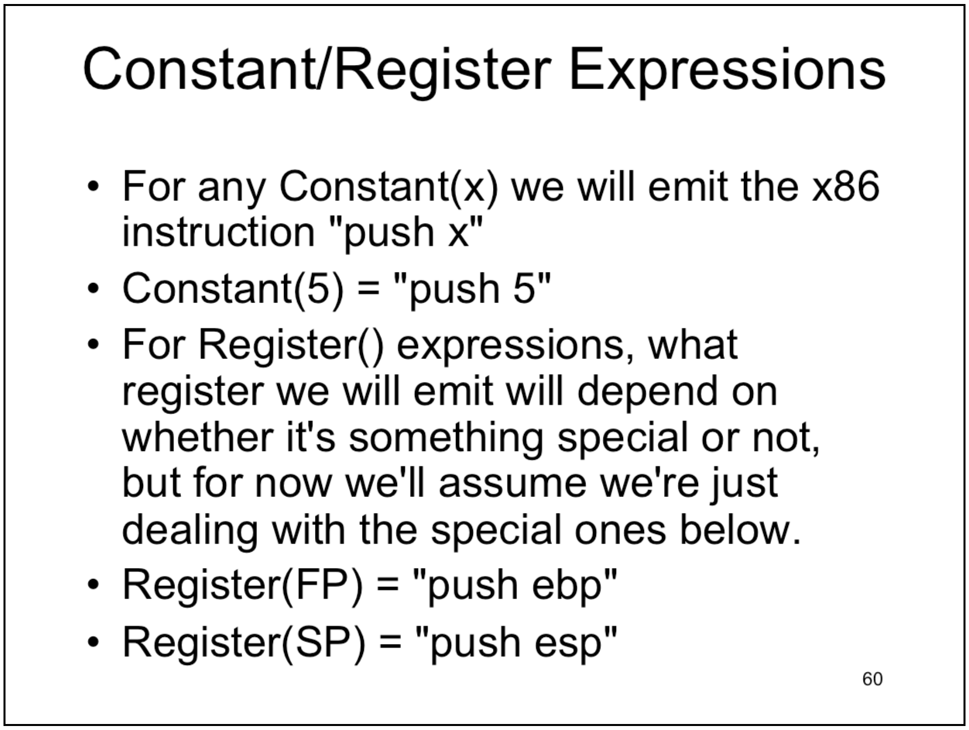
See notes for citation

Diagram outline adapted from "Concepts of Programming Languages 4th ed." Figure 1.3 page 28



AAT Direct To x86 Assembly

- http://www.cs.usfca.edu/~galles/compilerdesign/x86.pdf
- To go from AAT to assembly, we use a "tiling strategy" whereby we group portions of the AAT and generate assembly for them. The above link shows a tiling strategy for directly outputting non-optimal, but simple and straightforward to understand x86 assembly code, so that's why we're going to use it.
- I recommend reading http://www.uogonline.com/drlee/CS410/Original-Slides/Chap8.java.pdf and http://www.uogonline.com/drlee/CS410/Original-Slides/Chap9.java.pdf and maybe http://www.cs.cornell.edu/courses/cs4120/2009fa/lectures/lec17-fa09.pdf at the same time as the above so that you can see complimentary instances of the simple tiling strategy and then the more advanced ones.

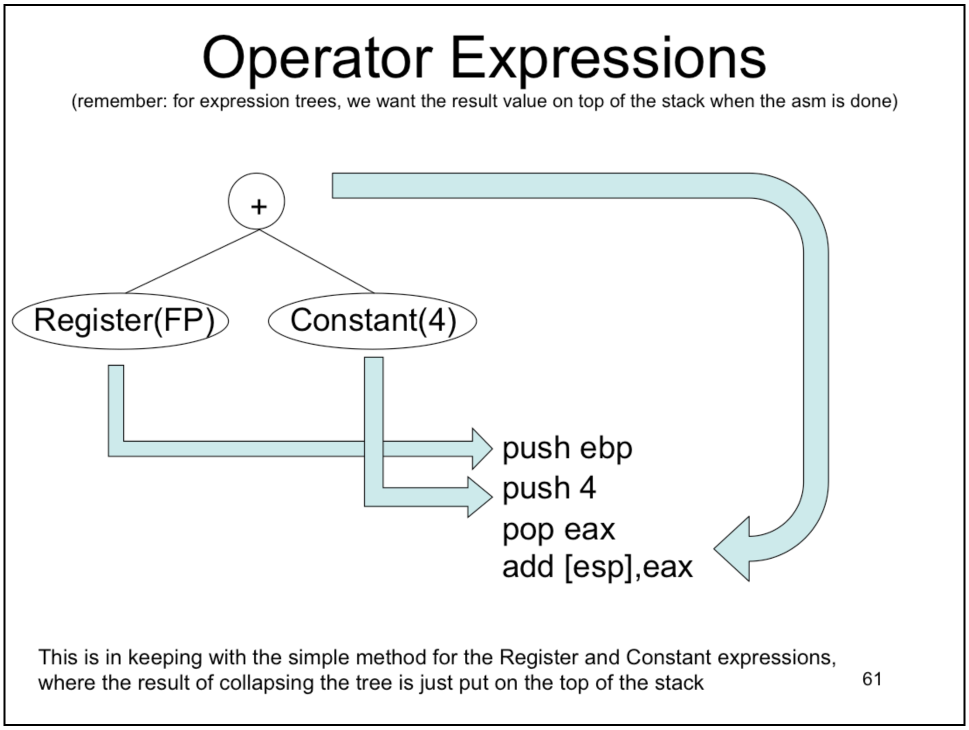


Constant/Register Expressions

- For any Constant(x) we will emit the x86 instruction "push x"
- Constant(5) = "push 5"
- For Register() expressions, what register we will emit will depend on whether it's something special or not, but for now we'll assume we're just dealing with the special ones below.
- Register(FP) = "push ebp"
- Register(SP) = "push esp"

Operator Expressions

(remember: for expression trees, we want the result value on top of the stack when the asm is done)

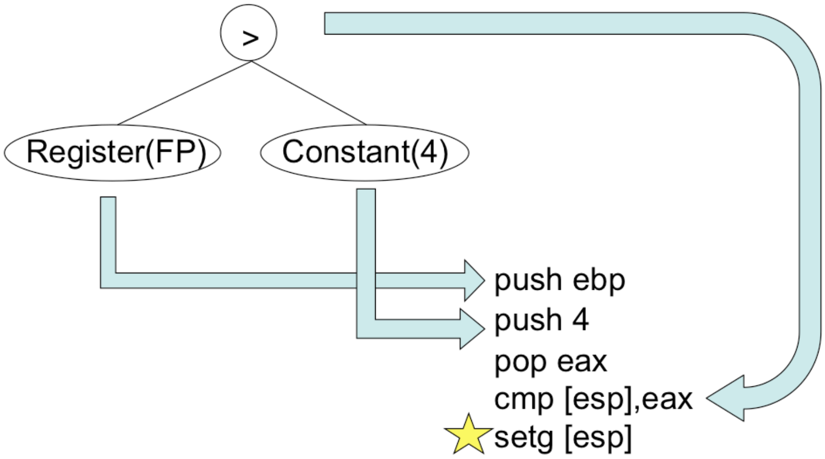


```
push ebp
push 4
pop eax
add [esp],eax
```

This is in keeping with the simple method for the Register and Constant expressions, where the result of collapsing the tree is just put on the top of the stack

Operator Expressions

(remember: for expression trees, we want the result value on top of the stack when the asm is done)

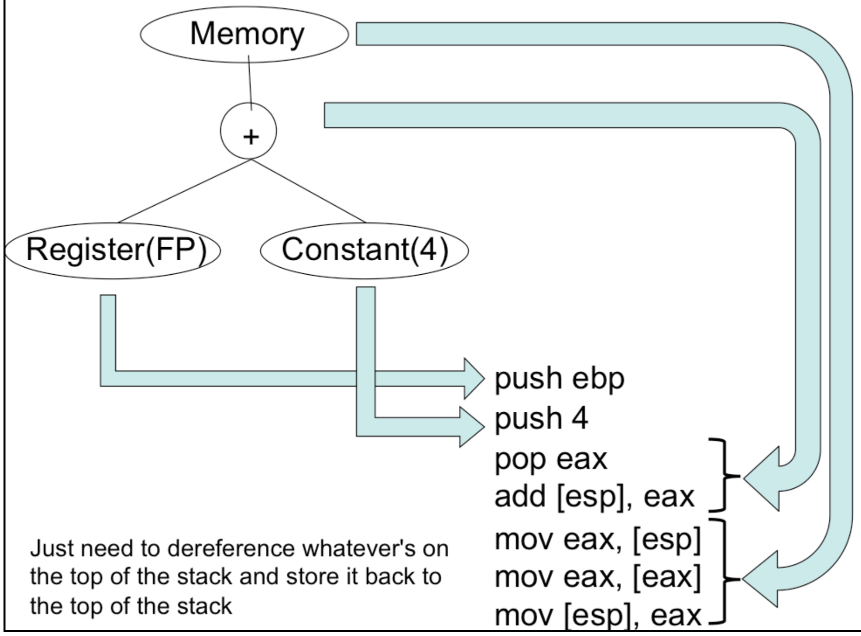


```
push ebp
push 4
pop eax
cmp [esp],eax
setg [esp]
```

We didn't learn about the "SETcc" group of instructions in the x86 classes, but all it does is set the specified destination to 1 if the condition holds, and 0 if it doesn't. setg like a jg (jump if greater) is a "set if greater", and then it's obviously putting the result on the top of the stack

Memory Expressions

(remember: for expression trees, we want the result value on top of the stack when the asm is done)

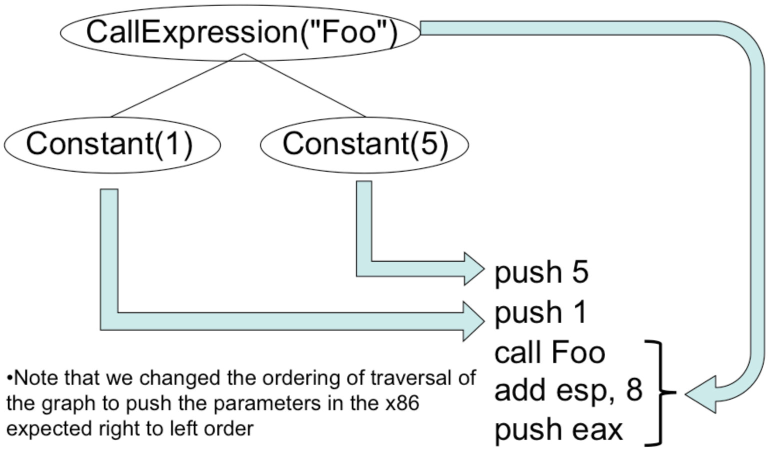


```
push ebp
push 4
pop eax
add [esp], eax
mov eax, [esp]
mov eax, [eax]
mov [esp], eax
```

Just need to dereference whatever's on the top of the stack and store it back to the top of the stack

Call Expressions

(remember: for expression trees, we want the result value on top of the stack when the asm is done)

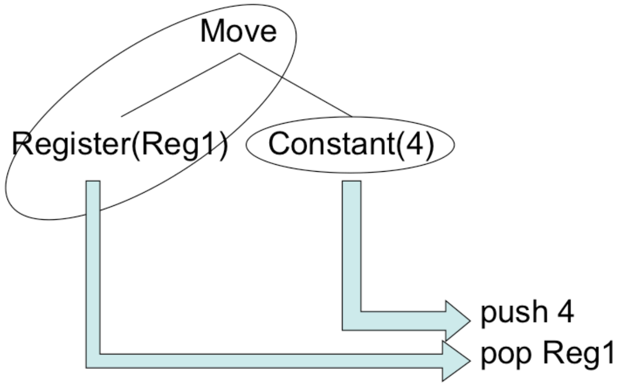


```
push 5
push 1
call Foo
add esp, 8
push eax
```

- Note that we changed the ordering of traversal of the graph to push the parameters in the x86 expected right to left order
- The "add esp, 8" is indicative of the cdecl calling convention (as we learned in Intro x86), and the 8 would just be an assumed 4 byte size per parameters multiplied by the number of parameters pushed.
- *How would the assembly be different if this was a call statement?*
- *How would the assembly be different if this was a stdcall calling convention?*

Move (to Register) Statements

(remember: for statement trees, we just want to perform some action, no net changes to the stack should occur)



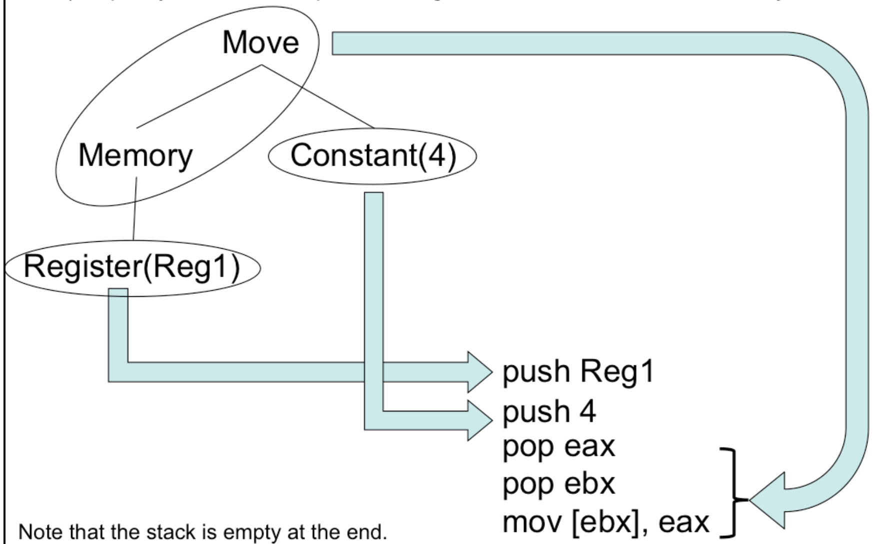
```
push 4
pop Reg1
```

Note that the stack is empty at the end.
Because Statement trees don't result in values,
they just cause actions.

Move (to Memory) Statements

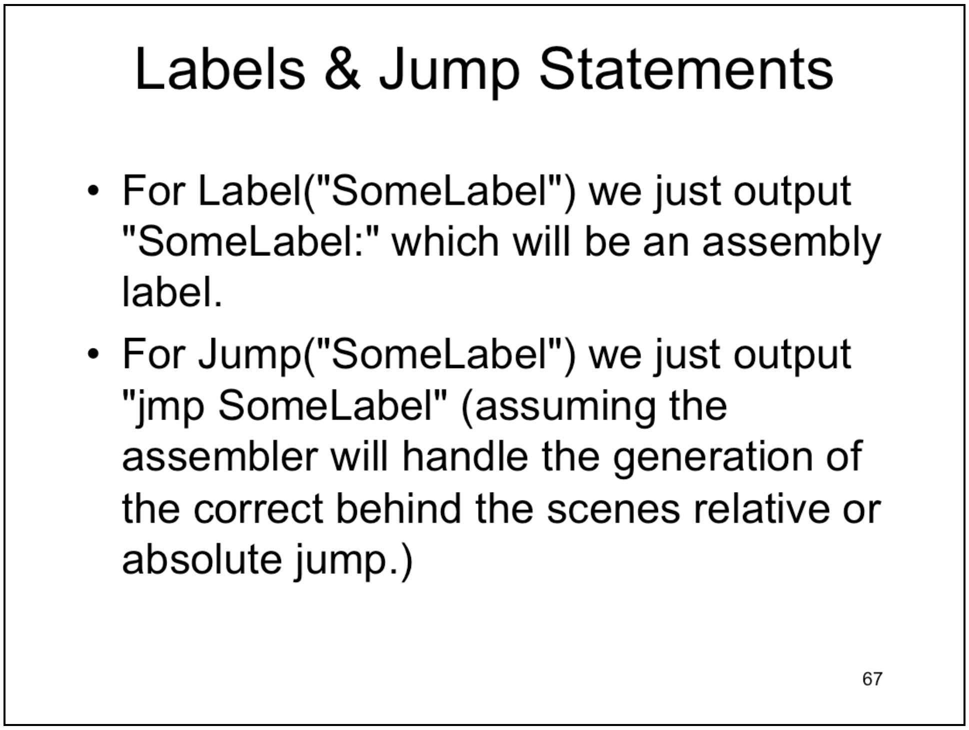
(remember: for statement trees, we just want to perform some action, no net changes to the stack should occur)

(I'm pretty sure the x86.pdf is wrong for this case so I substituted my own code)



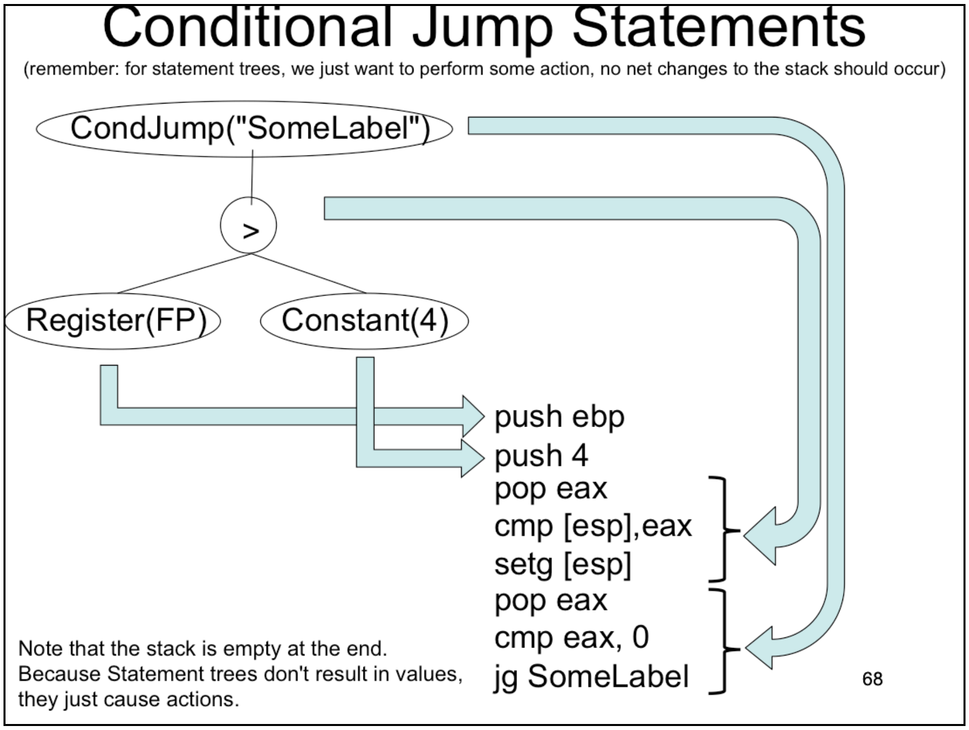
```
push Reg1
push 4
pop eax
pop ebx
mov [ebx], eax
```

Note that the stack is empty at the end.
Because Statement trees don't result in values,
they just cause actions.



Labels & Jump Statements

- For Label("SomeLabel") we just output "SomeLabel:" which will be an assembly label.
- For Jump("SomeLabel") we just output "jmp SomeLabel" (assuming the assembler will handle the generation of the correct behind the scenes relative or absolute jump.)



Conditional Jump Statements

(remember: for statement trees, we just want to perform some action, no net changes to the stack should occur)

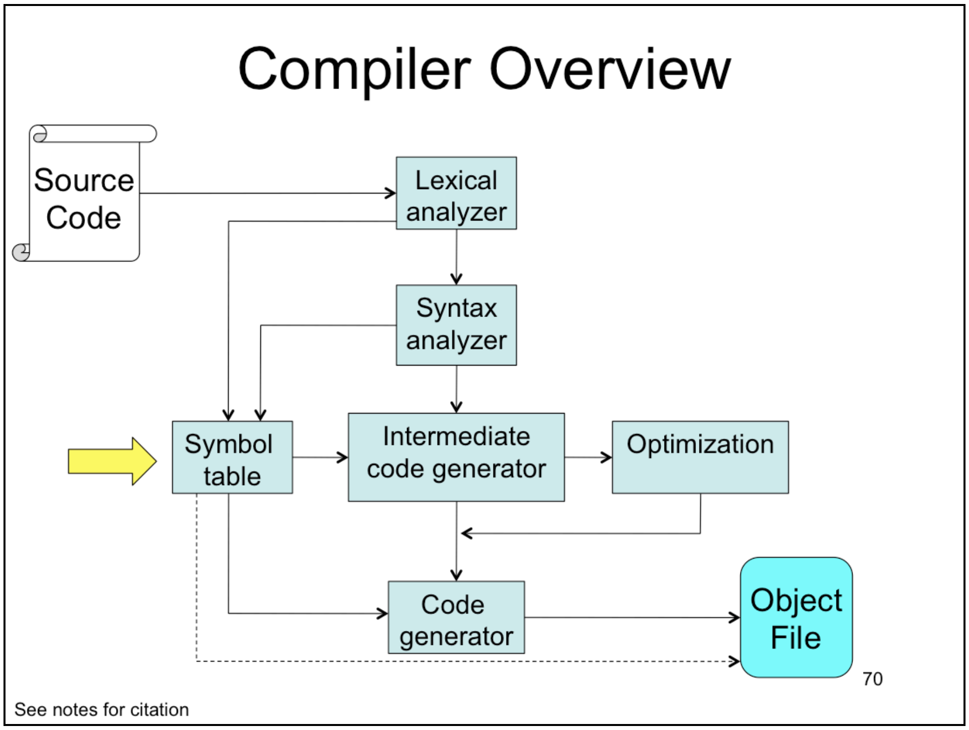
```
push ebp
push 4
pop eax
cmp [esp],eax
setg [esp]
pop eax
cmp eax, 0
jg SomeLabel
```

Note that the stack is empty at the end.
Because Statement trees don't result in values,
they just cause actions.

68

And the rest

- I have left out a ton of things, and didn't go over any of the more optimal tiling strategies. So again, if you're interested in more details, I really like these slides and encourage you to read them:
- http://www.uogonline.com/drlee/CS410/Original-Slides/Chap8.java.pdf
- http://www.uogonline.com/drlee/CS410/Original-Slides/Chap9.java.pdf
- http://www.cs.usfca.edu/~galles/compilerdesign/x86.pdf
- http://www.cs.cornell.edu/courses/cs4120/2009fa/lectures/lec17-fa09.pdf



Compiler Overview

Source Code

Lexical analyzer

Syntax analyzer

Symbol table

Intermediate code generator

Optimization

Code generator

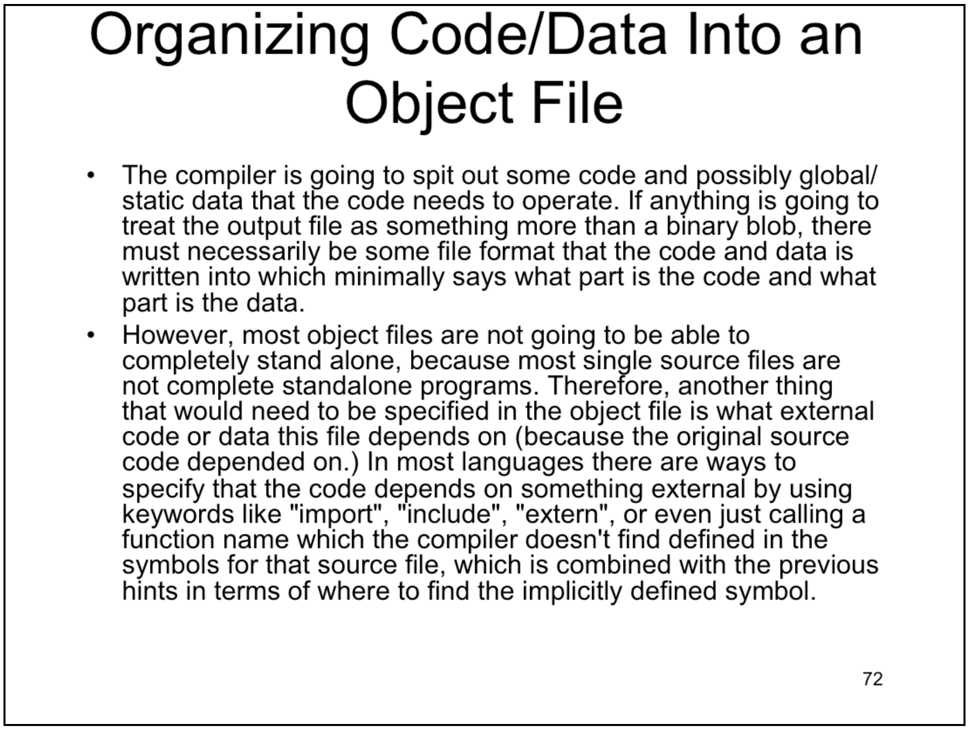
Object File

See notes for citation

Diagram outline adapted from "Concepts of Programming Languages 4th ed." Figure 1.3 page 28

Symbols

- I added the dashed line from the symbol table to the object file just to say that generally there will be *some* form of symbol table in the object file. It need not be exactly the same as is used in the other stages, but there will be something.
- The symbols table is basically a little database where the various stages can store information about the names & types of variables & functions.



Organizing Code/Data Into an Object File

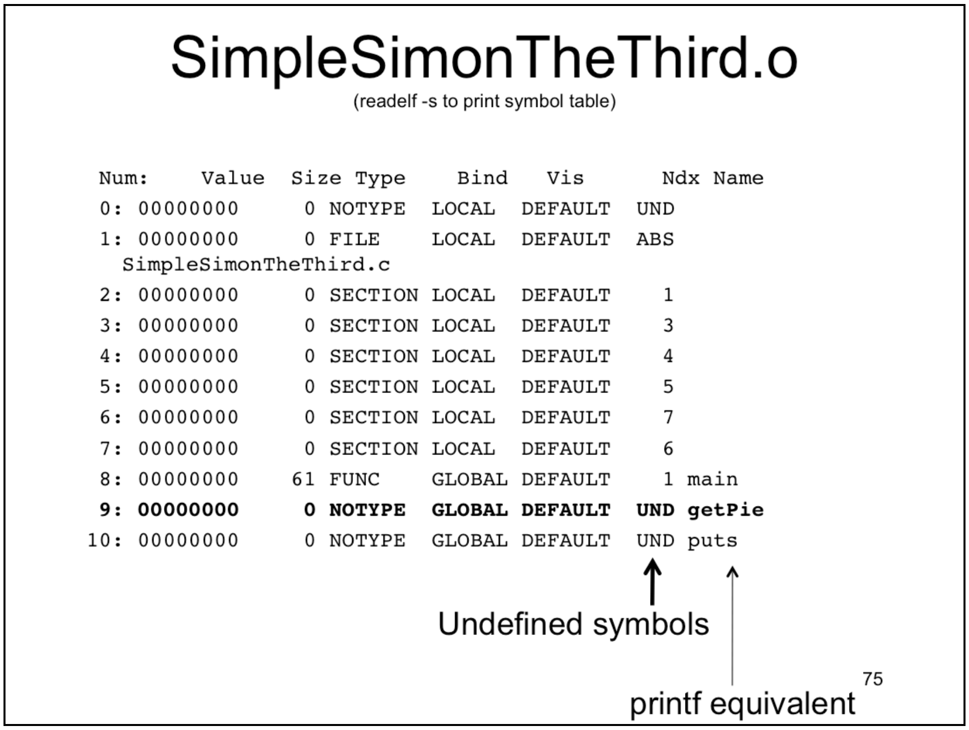
- The compiler is going to spit out some code and possibly global/static data that the code needs to operate. If anything is going to treat the output file as something more than a binary blob, there must necessarily be some file format that the code and data is written into which minimally says what part is the code and what part is the data.
- However, most object files are not going to be able to completely stand alone, because most single source files are not complete standalone programs. Therefore, another thing that would need to be specified in the object file is what external code or data this file depends on (because the original source code depended on.) In most languages there are ways to specify that the code depends on something external by using keywords like "import", "include", "extern", or even just calling a function name which the compiler doesn't find defined in the symbols for that source file, which is combined with the previous hints in terms of where to find the implicitly defined symbol.

Organizing Code/Data Into an Object File 2

- So the compiler and linker must therefore have some protocol/format specification embedded in the object file whereby the compiler knows it can say "this code needs to access this symbol" (whether the symbol is code or data), and the linker then knows how to search for these unresolved symbols at link time when it's putting all the objects together into a final binary.
- When the linker can't that's obviously where "unresolved symbol <bla>" errors come from.
- Also this sort of implies that the object files need to be able to say "yeah, I have that symbol, and it's located within my binary here"

Quick Example: SimpleSimonTheThird

- This is jumping ahead slightly because it shows some info about binary formats, but I just felt like justifying the claim about contained and uncontained symbols.



SimpleSimonTheThird.o

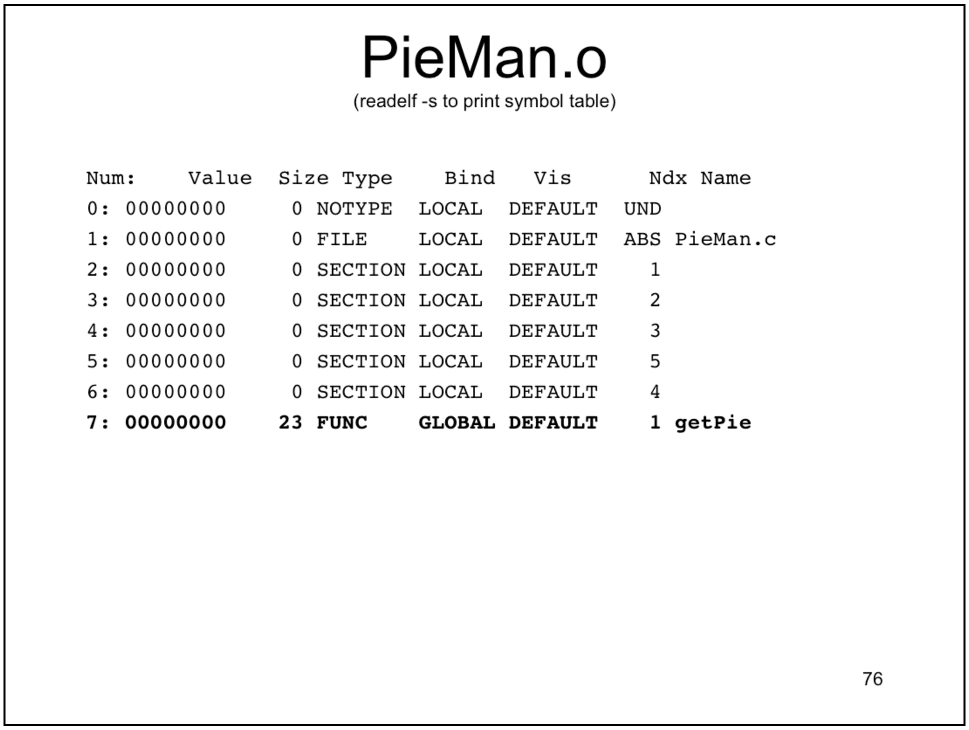
(readelf -s to print symbol table)

```
 Num:    Value  Size Type    Bind   Vis      Ndx Name
  0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 00000000     0 FILE    LOCAL  DEFAULT  ABS
   SimpleSimonTheThird.c
  2: 00000000     0 SECTION LOCAL  DEFAULT    1
  3: 00000000     0 SECTION LOCAL  DEFAULT    3
  4: 00000000     0 SECTION LOCAL  DEFAULT    4
  5: 00000000     0 SECTION LOCAL  DEFAULT    5
  6: 00000000     0 SECTION LOCAL  DEFAULT    7
  7: 00000000     0 SECTION LOCAL  DEFAULT    6
  8: 00000000    61 FUNC    GLOBAL DEFAULT    1 main
  9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND getPie
 10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

Undefined symbols

printf equivalent

75

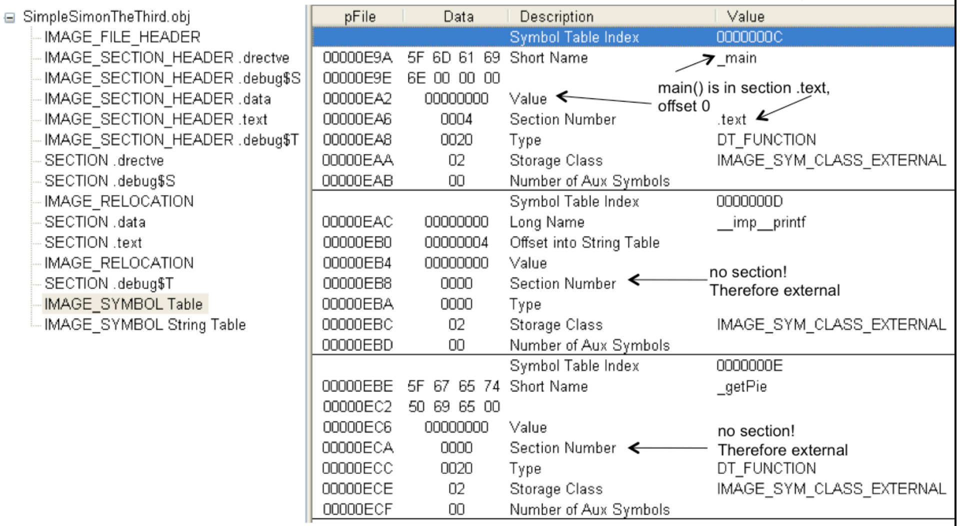


PieMan.o

(readelf -s to print symbol table)

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
1: 00000000     0 FILE    LOCAL  DEFAULT  ABS PieMan.c
2: 00000000     0 SECTION LOCAL  DEFAULT    1
3: 00000000     0 SECTION LOCAL  DEFAULT    2
4: 00000000     0 SECTION LOCAL  DEFAULT    3
5: 00000000     0 SECTION LOCAL  DEFAULT    5
6: 00000000     0 SECTION LOCAL  DEFAULT    4
7: 00000000    23 FUNC    GLOBAL DEFAULT    1 getPie
```

SimpleSimonTheThird.obj



- SimpleSimonTheThird.obj
  - IMAGE_FILE_HEADER
  - IMAGE_SECTION_HEADER .drectve
  - IMAGE_SECTION_HEADER .debug$S
  - IMAGE_SECTION_HEADER .data
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .debug$T
  - SECTION .drectve
  - SECTION .debug$S
  - IMAGE_RELOCATION
  - SECTION .data
  - SECTION .text
  - IMAGE_RELOCATION
  - SECTION .debug$T
  - IMAGE_SYMBOL Table
  - IMAGE_SYMBOL String Table

pFile	Data	Description	Value
		Symbol Table Index	0000000C
00000E9A	5F 6D 61 69	Short Name	_main
00000E9E	6E 00 00 00		
00000EA2	00000000	Value	
00000EA6	0004	Section Number	.text
00000EA8	0020	Type	DT_FUNCTION
00000EAA	02	Storage Class	IMAGE_SYM_CLASS_EXTERNAL
00000EAB	00	Number of Aux Symbols	
		Symbol Table Index	0000000D
00000EAC	00000000	Long Name	__imp__printf
00000EB0	00000004	Offset into String Table	
00000EB4	00000000	Value	
00000EB8	0000	Section Number	
00000EBA	0000	Type	
00000EBC	02	Storage Class	IMAGE_SYM_CLASS_EXTERNAL
00000EBD	00	Number of Aux Symbols	
		Symbol Table Index	0000000E
00000EBE	5F 67 65 74	Short Name	_getPie
00000EC2	50 69 65 00		
00000EC6	00000000	Value	
00000ECA	0000	Section Number	
00000ECC	0020	Type	DT_FUNCTION
00000ECE	02	Storage Class	IMAGE_SYM_CLASS_EXTERNAL
00000ECF	00	Number of Aux Symbols	

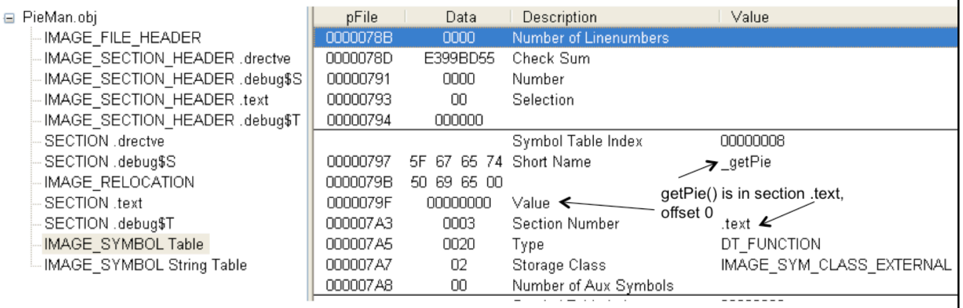
main() is in section .text, offset 0

no section! Therefore external

no section! Therefore external

77

PieMan.obj



- PieMan.obj
  - IMAGE_FILE_HEADER
  - IMAGE_SECTION_HEADER .drectve
  - IMAGE_SECTION_HEADER .debug$S
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .debug$T
  - SECTION .drectve
  - SECTION .debug$S
  - IMAGE_RELOCATION
  - SECTION .text
  - SECTION .debug$T
  - IMAGE_SYMBOL Table
  - IMAGE_SYMBOL String Table

pFile	Data	Description	Value
0000078B	0000	Number of Linenumbers	
0000078D	E399BD55	Check Sum	
00000791	0000	Number	
00000793	00	Selection	
00000794	000000		
		Symbol Table Index	00000008
00000797	5F 67 65 74	Short Name	_getPie
0000079B	50 69 65 00		
0000079F	00000000	Value	
000007A3	0003	Section Number	.text
000007A5	0020	Type	DT_FUNCTION
000007A7	02	Storage Class	IMAGE_SYM_CLASS_EXTERNAL
000007A8	00	Number of Aux Symbols	

getPie() is in section .text,
offset 0

Git along lil doggie!

- We're going to get back into more about compiler options, linker options, and linking after we see more about the binary formats which are used to store binaries pre and post-linking.