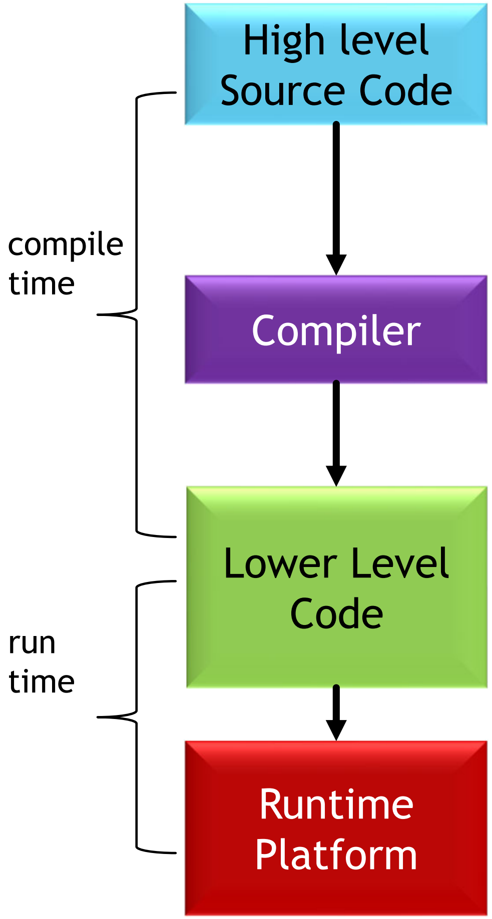
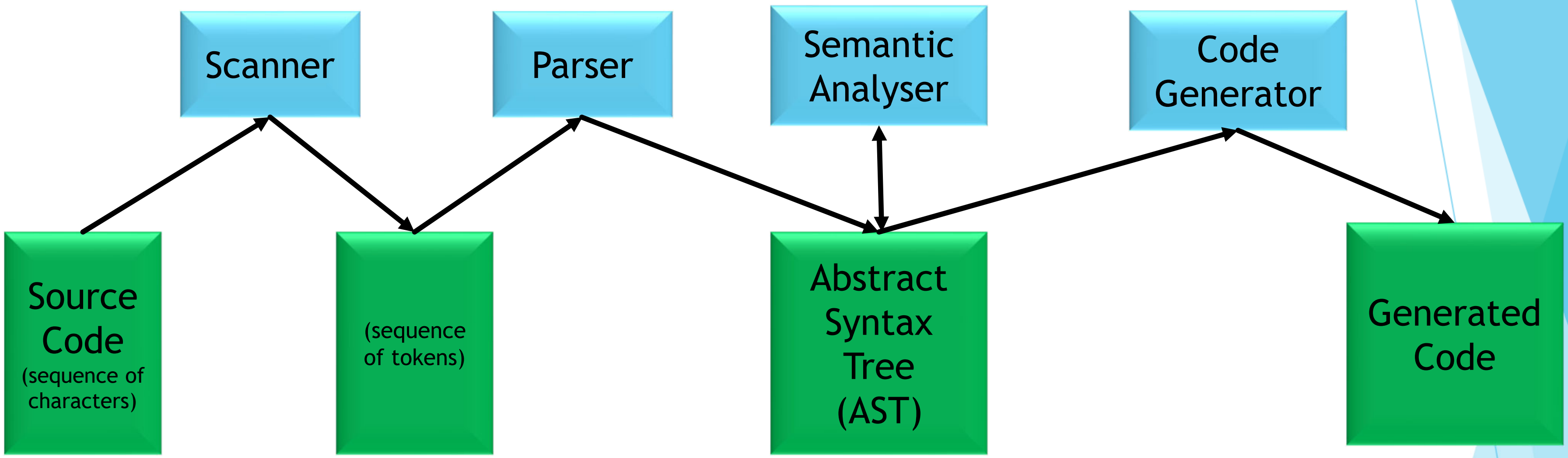
Week 12

Compiling

Compiling

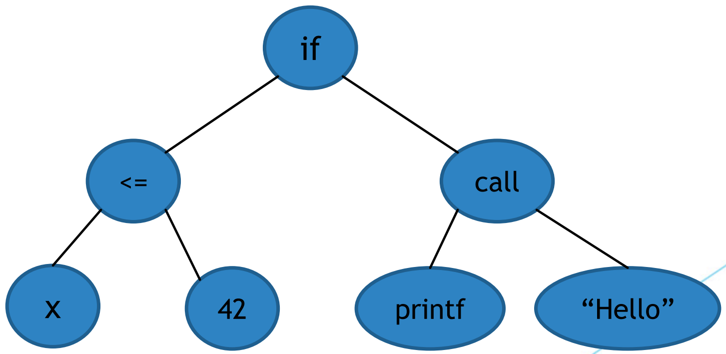

High level Source Code
compile time
Compiler
Lower Level Code
run time
Runtime Platform

Compiler Passes/Phases



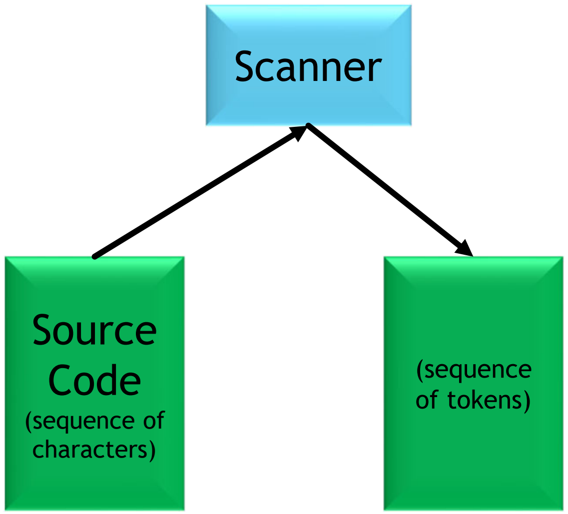

```
if (x <= 41)
    printf("Hello");
```

```
ifkeyword
lparen
identifier ("x")
leq
integerLiteral (42)
rparen
identifier ("printf")
lparen
stringLiteral ("Hello")
rparen
semicolon
```




```
00B513E5 cmp  dword ptr [x],2Ah
00B513E9 jg   main+42h (0B51402h)
00B513EB mov  esi,esp
00B513ED push 0B55858h
00B513F2 call dword ptr ds:[0B59114h]
00B513F8 add  esp,4
00B513FB cmp  esi,esp
```

Scanner (Lexical Analysis)




```
if (x <= 41)
    printf("Hello");
```

```
ifkeyword
lparen
identifier ("x")
leq
integerLiteral (42)
rparen
identifier ("printf")
lparen
stringLiteral ("Hello")
rparen
semicolon
```

Scanning

- Eliminate whitespace [spaces(' '), tabs (\t), new line (\n), carriage return(\r)]
- Eliminate comments
- Group characters:
  - "<=" -> less than or equals operator
  - "if " -> if keyword
  - 42 -> integer literal
  - 2.99e+8 -> float literal
  - "Hello World\n" -> string literal
  - sub_total_2 -> identifier

Regular Expressions

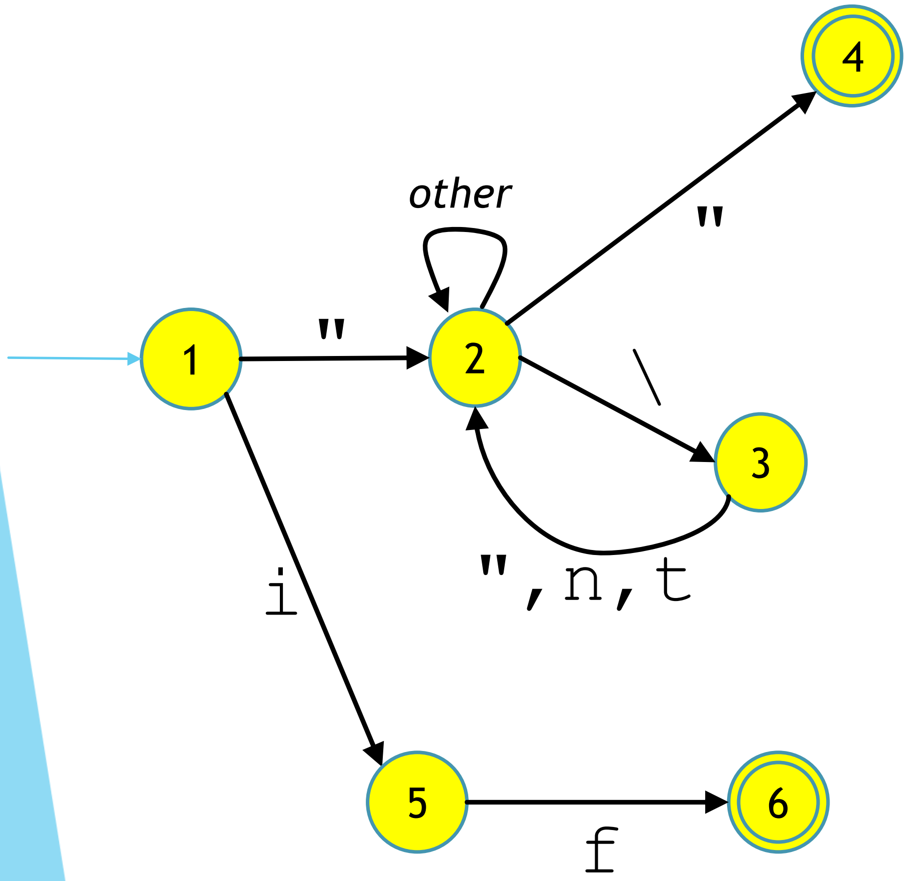
- ϵ *empty string*
- **c** *literal character*
- $r_1 \mid r_2$ *r_1 or r_2*
- $r_1\ r_2$ *r_1 followed by r_2*
- r^{*} *zero or more occurrences of r*
- $r+ \Leftrightarrow r\ r^{*}$ *one or more occurrences of r*
- $r? \Leftrightarrow (r \mid \epsilon)$ *r is optional*
- $[1\text{-}9] \Leftrightarrow (1|2|3|4|5|6|7|8|9)$ *range of values*

```
if                          if keyword
(                           open parenthesis
)                           close parenthesis
<=                          less than or equal
;                           semi colon
[a-zA-Z][a-zA-Z0-9]*        identifier
[0-9]+                      integer literal
"([^"]|\[nt"])*"            string literal
```

Chomsky Language Hierarchy

Grammar Type	Machine Required to Recognize
3) Regular (most Scanners)	Finite State Automaton
2) Context-Free (most Parsers)	Pushdown Automaton
1) Context-Sensitive	Linear Bounded Automaton
0) Unrestricted	Turing Machines

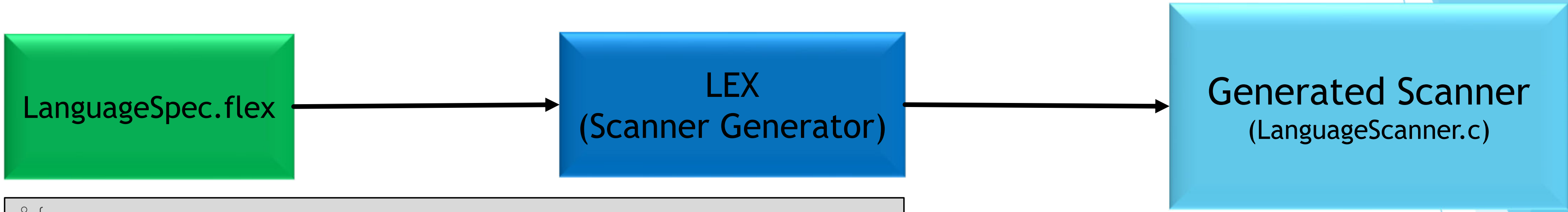
Finite State Automata (FSA)



`"([^\"]|\[nt"])*"`

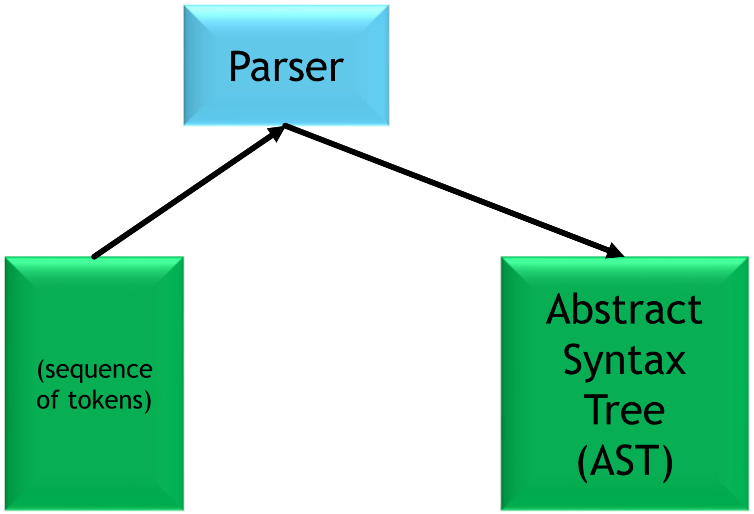
state	"	\	n	t	i	f	...
1	2				5		
2	4	3	2	2	2	2	2
3	2		2	2			
4	accept string						
5						6	
6	accept ifkeyword						

Scanner Generators (e.g. lex/flex)

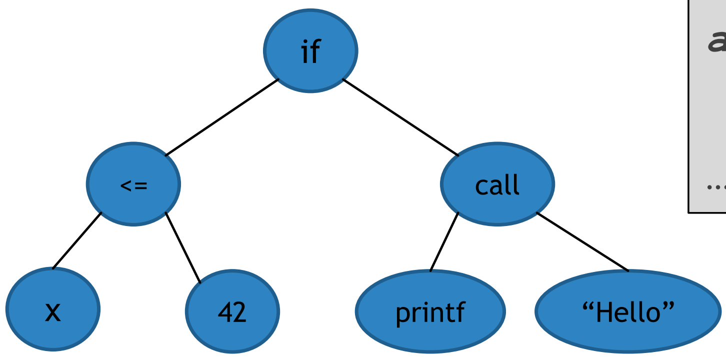



```
%{
#include "y.tab.h"
%}
%%
[ \t\n]                      { }
"if"                         { return ifkeyword; }
"("                          { return lparen; }
")"                          { return rparen; }
"<="                         { return leq; }
";"                          { return semicolon; }
[a-zA-Z][a-zA-Z0-9]*         { return identifier; }
[0-9]+                       { return integerLiteral; }
\"([^\\"]|\\[nt\"])*\"       { return stringLiteral; }
%%
```

Parsing



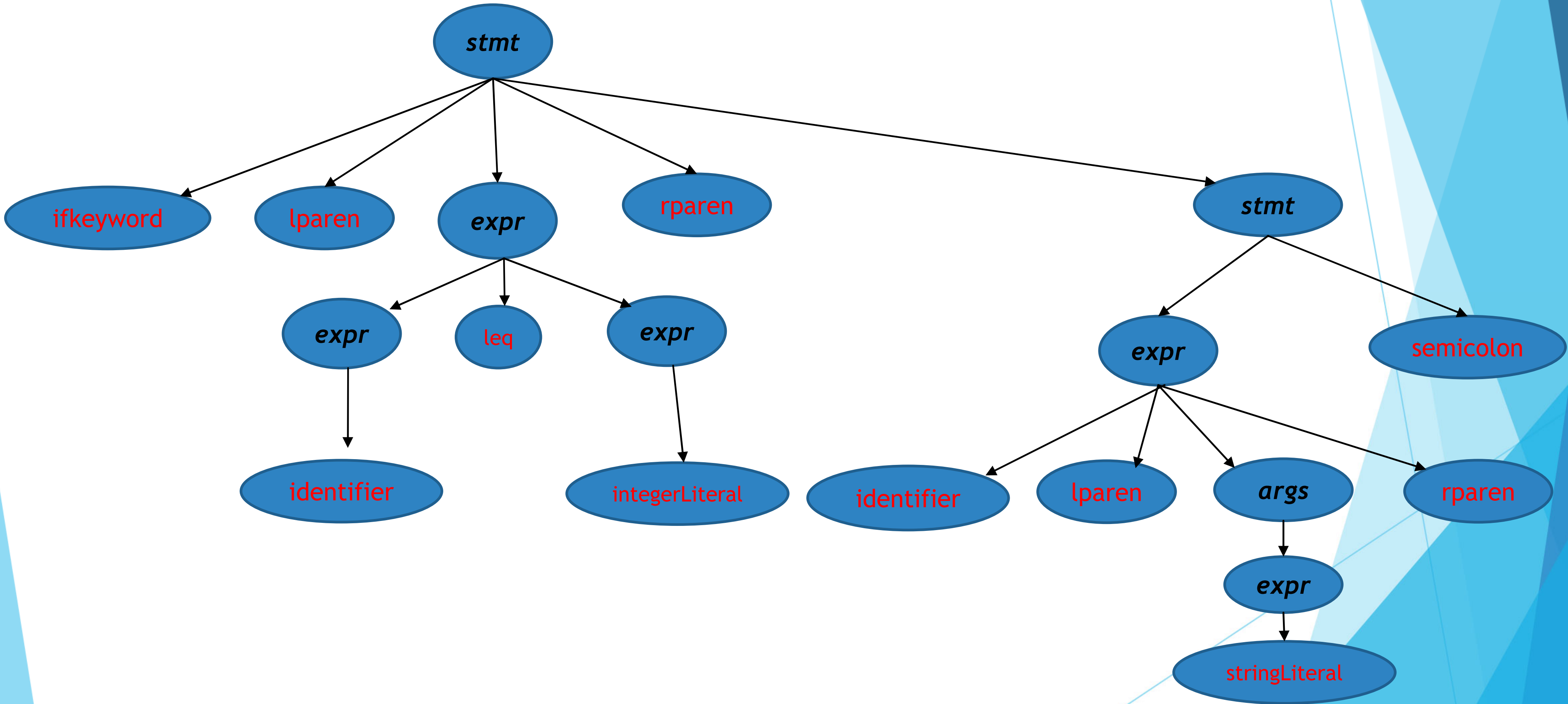

```
ifkeyword
lparen
identifier ("x")
leq
integerLiteral (42)
rparen
identifier ("printf")
lparen
stringLiteral ("Hello")
rparen
semicolon
```



Context Free Grammar

```
stmt -> if lparen expr rparen stmt
     -> expr semicolon
     -> …
expr -> integerLiteral
     -> stringLiteral
     -> identifier
     -> identifier lparen args rparen
     -> expr leq expr
     -> …
args -> expr
     -> args comma expr
…
```

Derivation Trees




```
ifkeyword,lparen,identifier,leq,integerLiteral,rparen,identifier,lparen,stringLiteral,rparen,semicolon
```

LL and LR Parsers

- **LL**
  - Left to Right traversal of input
  - Leftmost derivation
  - Top down parser
    - Predictive/Recursive descent
  - ANTLR generates top-down LL parsers
- **LR**
  - Left to Right traversal of the input
  - Rightmost derivation (reverse/bottom up)
  - Bottom up parser
  - more powerful than LL parsers
  - Look-ahead: LR(0), LALR(1), LR(1)
  - YACC generates bottom-up LALR parsers

Recursive Descent

```
void stmt()
{
    if (next == ifkeyword)
    {
        match(ifkeyword); match(lparen); expr(); match(rparen); stmt();
    }
    else
    {
        expr(); match(semicolon);
    }
}

void expr()
{
    if (next == integerLiteral)
        match(integerLiteral);
    else if (next == stringLiteral)
        match(stringLiteral);
    else if (next == identifier);
        match(identifier);
    else
    {
        expr(); match(leq); expr();
    }
}
void args()
{
    expr();
    if (next == comma)
    {
        match(comma);
        args();
    }
}
```

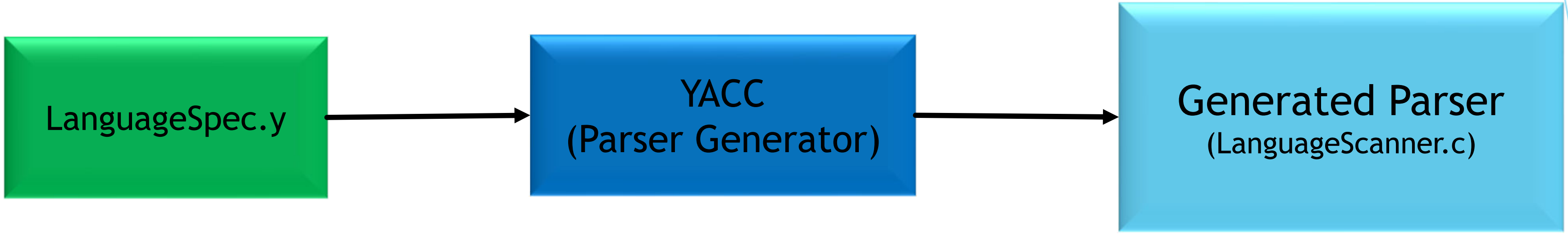
```
stmt -> if lparen expr rparen stmt
     -> expr semicolon
     -> …
expr -> integerLiteral
     -> stringLiteral
     -> identifier
     -> identifier lparen args rparen
     -> expr leq expr
     -> …
args -> expr
     -> args comma expr
…
```

Shift/Reduce Parsing

▶ **Shift** ifkeyword	ifkeyword
▶ **Shift** lparen	ifkeyword lparen
▶ **Shift** identifier	ifkeyword lparen identifier
▶ **Reduce** identifier -> expr	ifkeyword lparen expr
▶ **Shift** leq	ifkeyword lparen expr leq
▶ **Shift** integerLiteral	ifkeyword lparen expr leq integerLiteral
▶ **Reduce** integerLiteral -> expr	ifkeyword lparen expr leq expr
▶ **Reduce** expr leq expr -> expr	ifkeyword lparen expr
▶ **Shift** rparen	ifkeyword lparen expr rparen
▶ **Shift** identifier	ifkeyword lparen expr rparen identifier
▶ **Shift** lparen	ifkeyword lparen expr rparen identifier lparen
▶ **Shift** stringLiteral	ifkeyword lparen expr rparen identifier lparen stringLiteral
▶ **Reduce** stringLiteral -> expr	ifkeyword lparen expr rparen identifier lparen expr
▶ **Reduce** expr -> args	ifkeyword lparen expr rparen identifier lparen args
▶ **Shift** rparen	ifkeyword lparen expr rparen identifier lparen args rparen
▶ **Reduce** identifier lparen args rparen -> expr	ifkeyword lparen expr rparen expr
▶ **Shift** semicolon	ifkeyword lparen expr rparen expr semicolon
▶ **Reduce** expr semicolon -> stmt	ifkeyword lparen expr rparen stmt
▶ **Reduce** ifkeyword lparen expr rparen stmt -> stmt	stmt

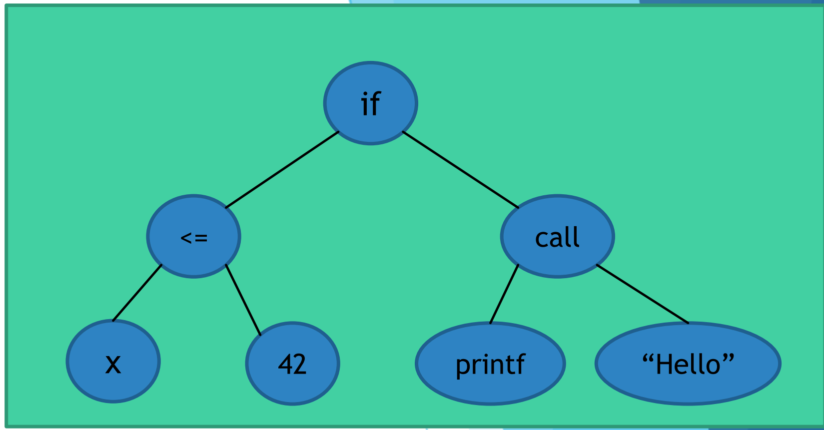
```
ifkeyword,lparen,identifier,leq,integerLiteral,rparen,identifier,lparen,stringLiteral,rparen,semicolon
```

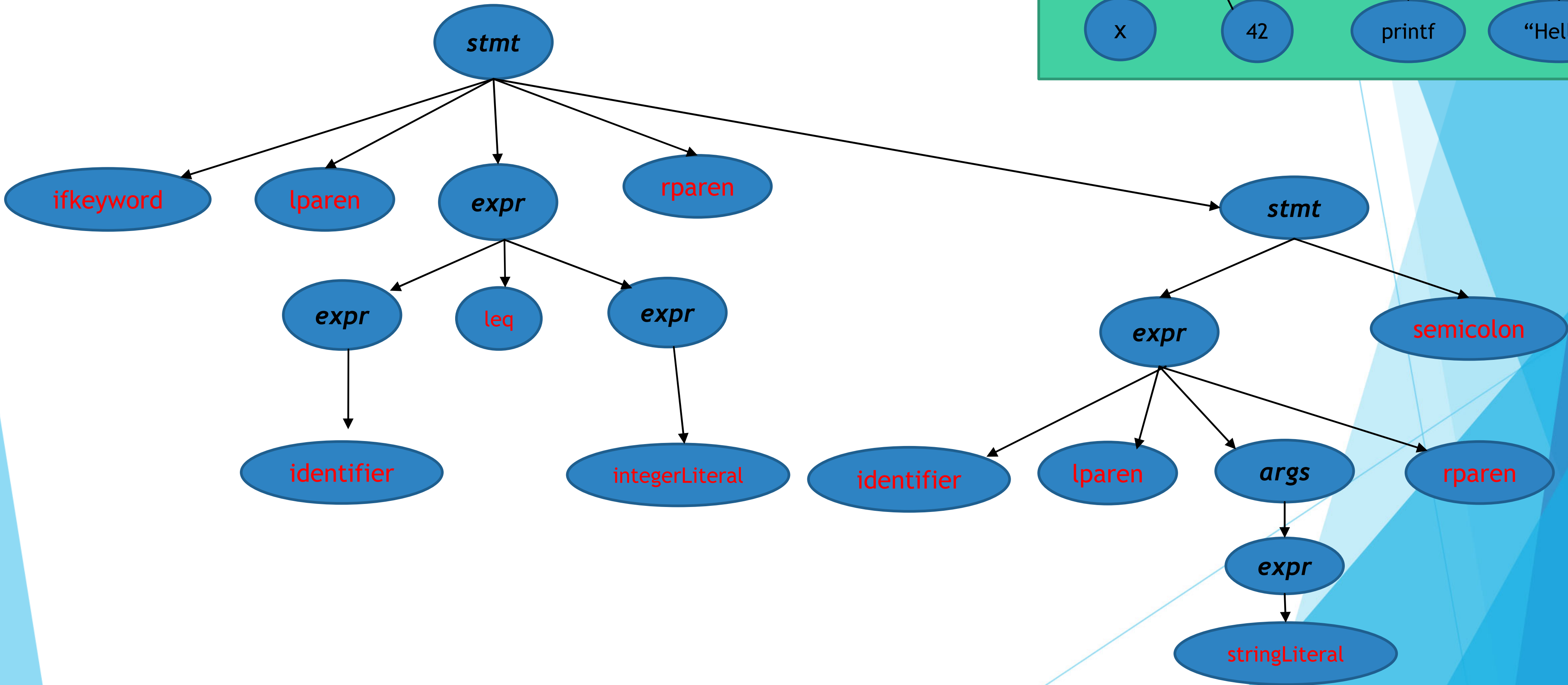
Parser Generators (e.g. YACC)




```
%token ifkeyword leq identifier integer string
%nonassoc leq
%%
stmt: if '(' expr ')' stmt
    | expr ';'
    ;
expr: integer
    | string
    | identifier
    | identifier '(' args ')'
    | expr leq expr
    ;
args: expr
    | args',' expr
    ;
%%
```

Concrete vs Abstract Syntax Tree



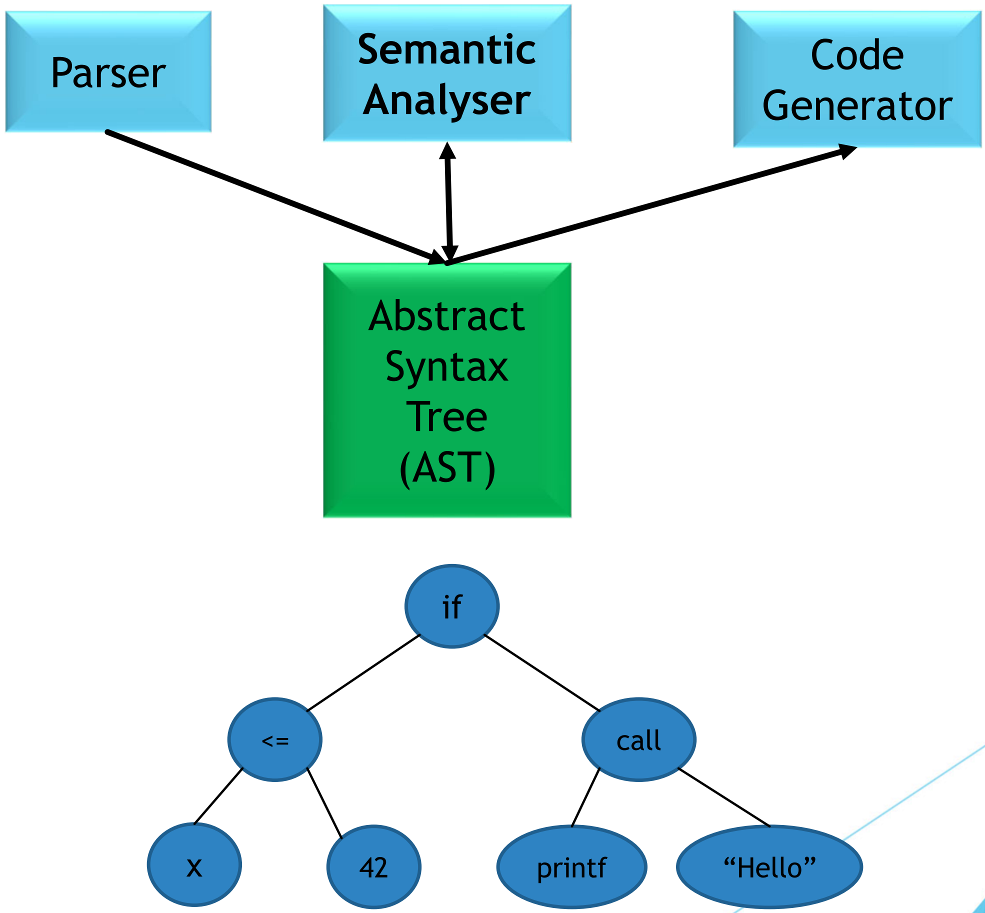


Creating an AST during Parsing

- http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

Semantic Analysis

1. Name Resolution
2. Type Checking
3. Data Flow Analysis



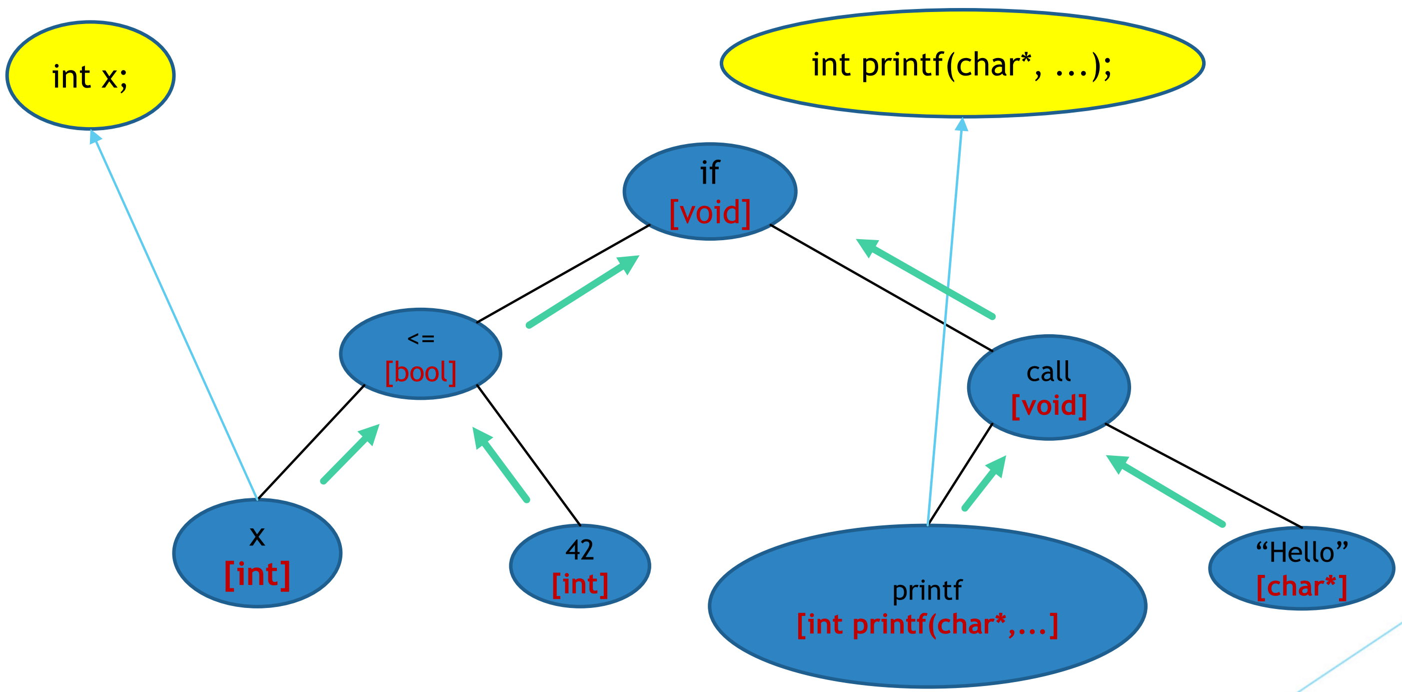
Name Resolution

- For each ***name*** (identifier):
  - Locate matching ***declaration***:
    - local variable declaration
    - parameter declaration
    - field or method declaration
    - class or type declaration
    - package declaration
  - Nested Lexical Scopes
  - Declaration may come after use
- For compound names:
  - e.g. `Console.WriteLine`
  - First resolve Console within current scope
    - locates class System.Console.
  - Then resolve WriteLine within located scope
    - within System.Console class

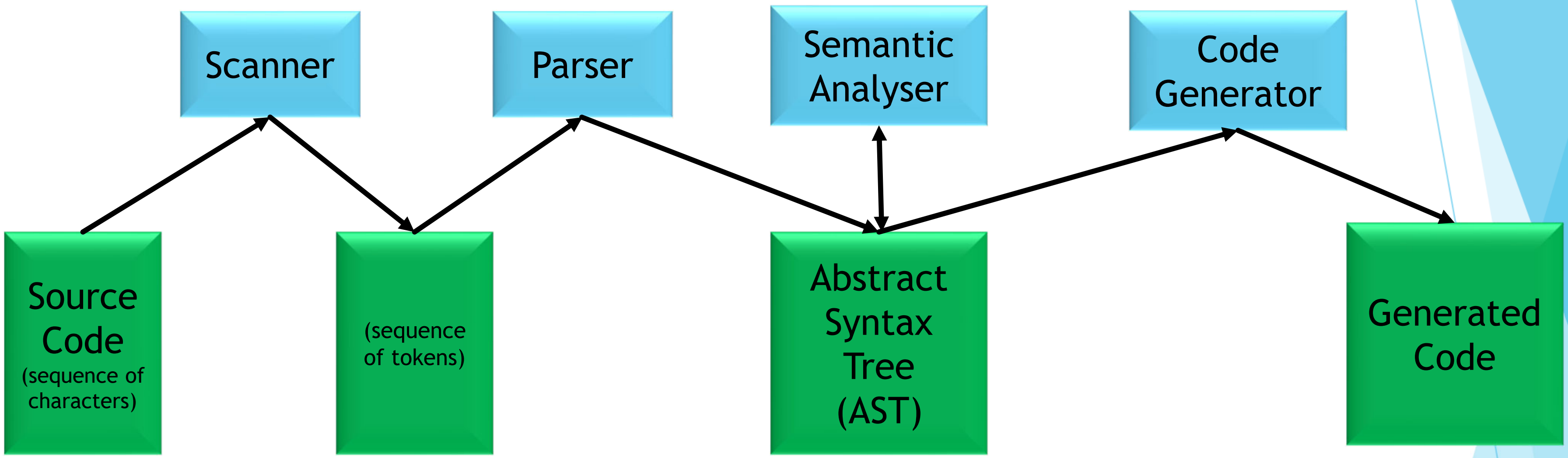
```
using System;
class Other
{
    static public int z;
}
namespace Foo
{
    class Program
    {
        void Foo(int x)
        {
            int a = 42;
            while (a > 19)
            {
                int b = a * 2;
                Console.WriteLine(Bar(a + b + x + y + Other.z));
            }
        }
        private int y;
        int Bar(int z)
        {
            return z;
        }
    }
}
```

Type Checking

- Type check by performing bottom up traversal of AST.
  - check/evaluate types of children prior to parent

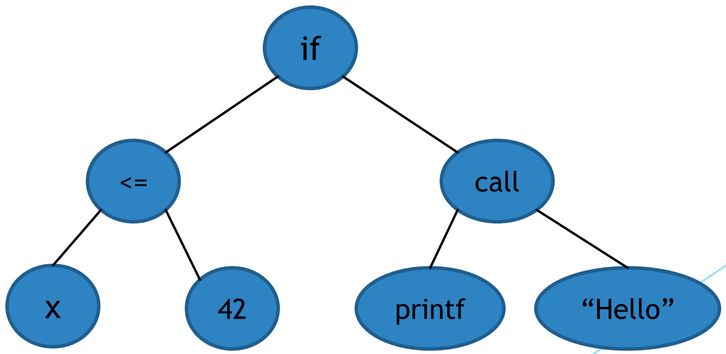


Code Generation



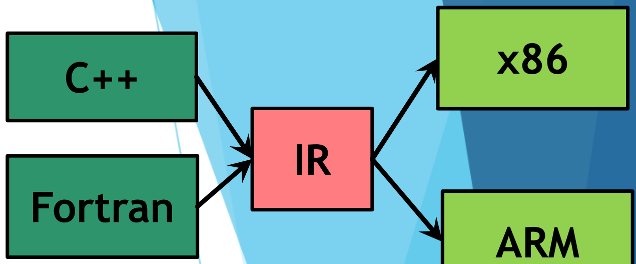

```
if (x <= 41)
    printf("Hello");
```

```
ifkeyword
lparen
identifier ("x")
leq
integerLiteral (42)
rparen
identifier ("printf")
lparen
stringLiteral ("Hello")
rparen
semicolon
```




```
00B513E5 cmp  dword ptr [x],2Ah
00B513E9 jg   main+42h (0B51402h)
00B513EB mov  esi,esp
00B513ED push 0B55858h
00B513F2 call dword ptr ds:[0B59114h]
00B513F8 add  esp,4
00B513FB cmp  esi,esp
```

Intermediate Code



- Many compilers first translate to an intermediate code and then from intermediate code to machine code.
- Intermediate code is generally low level, but machine independent
  - Just need to add a new "back end" to support additional machines types.
- Intermediate code is generally language independent
  - Different "front-end" compilers can share the same code generation infrastructure.

- Most dynamic languages first convert to an intermediate code which is then either interpreted or just in time compiled.

```
@.str = internal constant [12 x i8] c"hello world\00"

define i32 @main() nounwind {
entry:
    %tmp1 = getelementptr ([12 x i8]* @.str, i32 0, i32 0)
    %tmp2 = call i32 (i8*, ...)* @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

Common Instruction Sets

- Processor market:
  - Mobile phones
  - Tablets
  - Embedded (Network controllers, Industrial, Consumer appliances)
  - PCs, Servers and Mainframes

- **x86** (most PCs and servers)
- **ARM** (Mobile phones, tablets and embedded, low power)

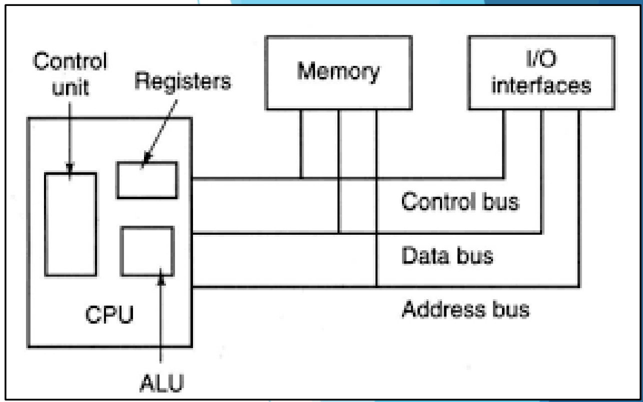
Instruction Sets

- RISC vs CISC vs VLIW
  - Reduced Instruction Set Computer (e.g. ARM)
  - Complex Instruction Set Computer (e.g. Intel)
  - Very Long Instruction Word (instruction parallel RISC)

- a set of registers
- a set of instructions (arithmetic/logic, memory and control flow)
  - opcode + zero or more operands (registers, memory locations or constants)
  - addressing modes

Instructions

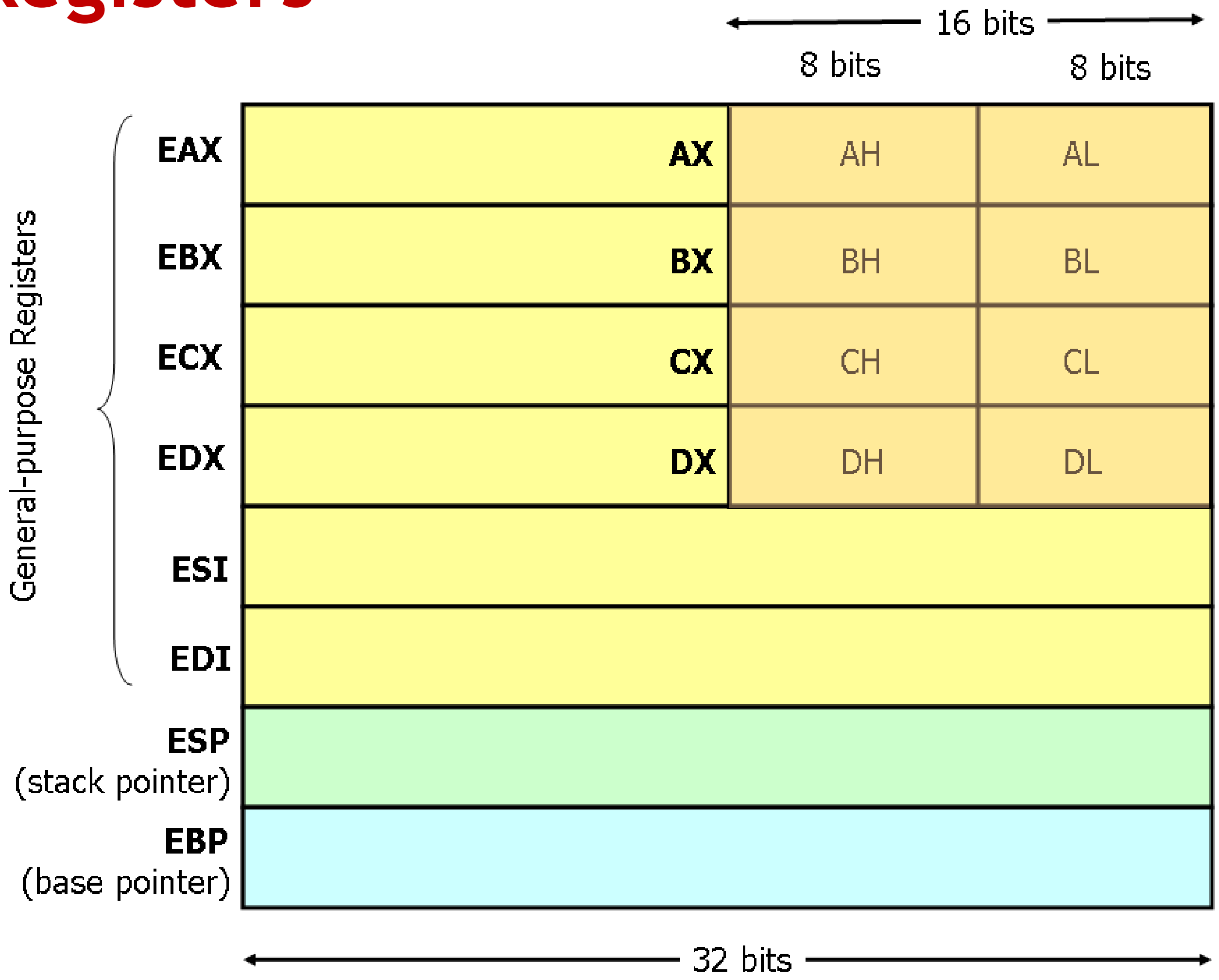
- Arithmetic/Logic
  - add, subtract, multiply, divide
  - and, or, shift, compare
  - Typically inputs and outputs are registers
- Memory Load/Store
  - load contents of a memory location into a register
  - store contents of a register into a memory location.
  - Memory address typically comes from a register.
  - Various addressing modes
- Control Flow
  - conditional and unconditional jump and function call/return
  - condition typically based on a register
  - jump to a new memory address (either absolute or relative address)
  - Changes the Program Counter (PC) register.



32bit vs 64bit

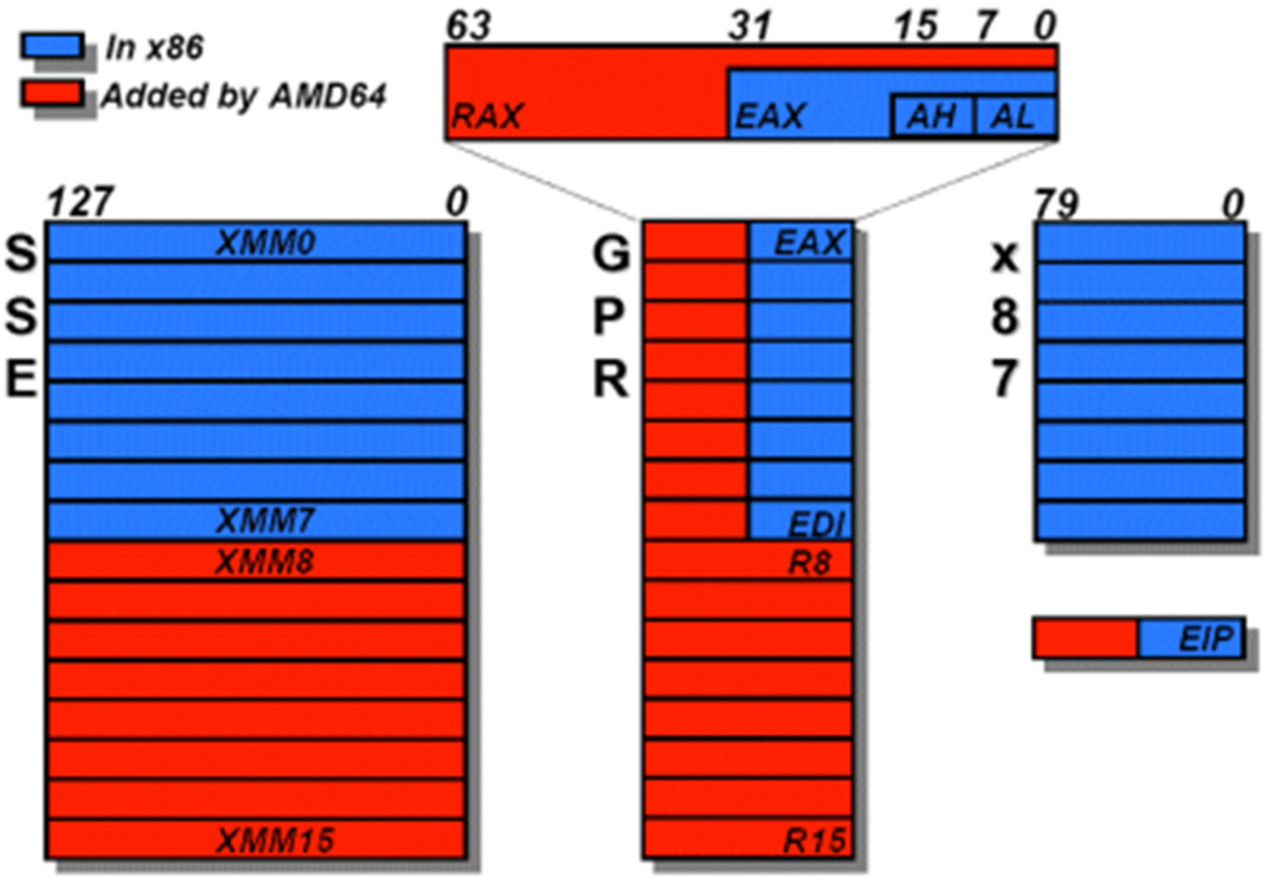
- 32bit vs 64bit refers to the size of the registers
  - Having bigger registers means you arithmetic operations can work on larger numbers and your logical operations can work on more bits in parallel.
  - Having a bigger register means you can point to a larger range of memory addresses and therefore use larger physical memory (> 4GB).
- 32bit x86 code will execute on a x86-64 processor
- 64bit x86 code will not execute on a 32bit processor.

x86 Registers



1978	16-bit
1989	32-bit
2001	64-bit

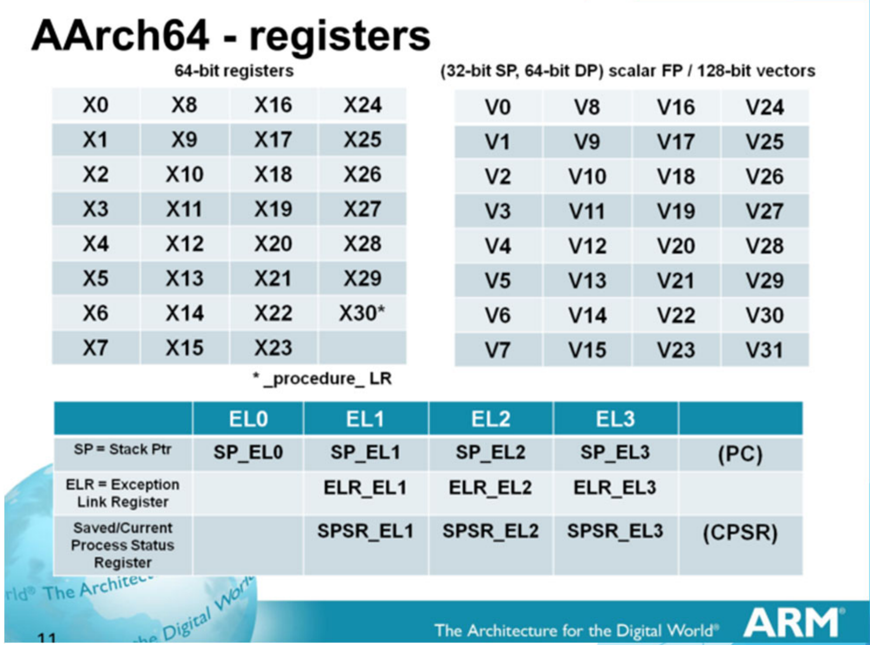
x86-64 Registers



ARM Registers

1985	32-bit
2011	64-bit

Register	
R0	General Purpose
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	Stack Pointer (SP)
R14	Link Register (LR) [return address]
R15	Program Counter (PC)



AArch64 - registers

64-bit registers

X0	X8	X16	X24
X1	X9	X17	X25
X2	X10	X18	X26
X3	X11	X19	X27
X4	X12	X20	X28
X5	X13	X21	X29
X6	X14	X22	X30*
X7	X15	X23	

* _procedure_ LR

(32-bit SP, 64-bit DP) scalar FP / 128-bit vectors

V0	V8	V16	V24
V1	V9	V17	V25
V2	V10	V18	V26
V3	V11	V19	V27
V4	V12	V20	V28
V5	V13	V21	V29
V6	V14	V22	V30
V7	V15	V23	V31

	EL0	EL1	EL2	EL3	
SP = Stack Ptr	SP_EL0	SP_EL1	SP_EL2	SP_EL3	(PC)
ELR = Exception Link Register		ELR_EL1	ELR_EL2	ELR_EL3	
Saved/Current Process Status Register		SPSR_EL1	SPSR_EL2	SPSR_EL3	(CPSR)

The Architecture for the Digital World® ARM®

https://www.anandtech.com/show/9766/the-apple-ipad-pro-review/3

int i = (x + y) * (x + y) - z;

x86 Un-optimized

```
mov     eax,dword ptr [x]
add     eax,dword ptr [y]
mov     ecx,dword ptr [x]
add     ecx,dword ptr [y]
imul    eax,ecx
sub     eax,dword ptr [z]
mov     dword ptr [i],eax
```

x86 Optimized

```
add     esi,eax     ; esi = esi + eax
imul    esi,esi     ; esi = esi * esi
sub     esi,eax     ; esi = esi - eax
```

Addressing Modes

- An addressing mode describes what kind of operands an instruction can take.
- Different processors support different addressing modes
- Common Addressing modes:
  - Immediate   operand = a specific address
  - Register   operand = register[r]
  - Indirect   operand = memory[register[base] + register[offset]*constant_size + constant_offset]

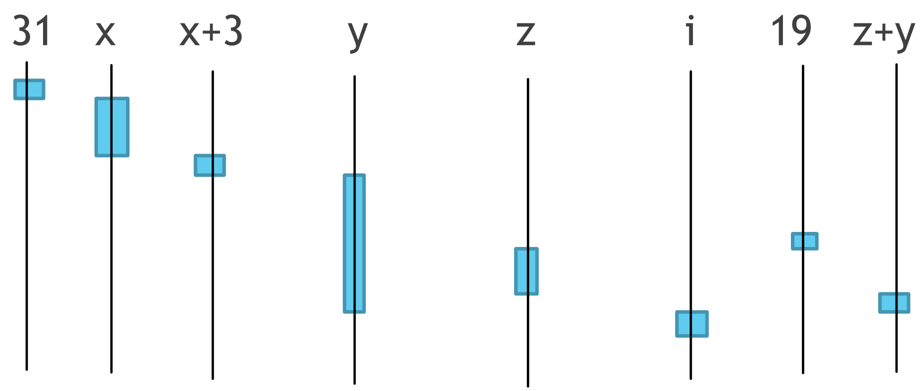
- **x64 Examples:**
  - `ADD EAX, 14`   *; add 14 into 32-bit EAX*
  - `ADD R8L, AL`   ; add 8 bit AL into R8L
  - `MOV R8W, 1234[8*RAX+RCX]`   ;move word at address 8*RAX+RCX+1234 into R8W

`A[i].f`

Register Allocation

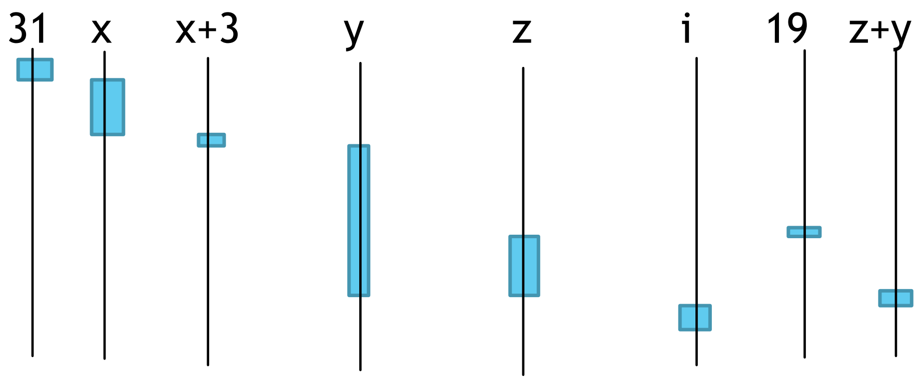
- Only a small finite number of registers available (precious resource).
- Need to decide which registers to store values in.
- Want to minimize unnecessary loading and storing to memory (costly).
- Need to determine the lifetime of each value:
  - when is the value computed
  - when is it last used.
- If the lifetime of two values does not overlap then we can store them both in the same register

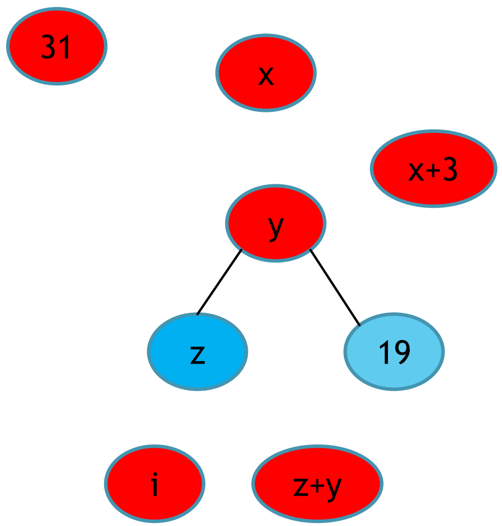
```
x = 31;
y = x + 3;
z = 19;
i = z + y;
```



Graph Colouring

```
x = 31;
y = x + 3;
z = 19;
i = z + y;
```





r1 r2

```
r1 = 31
r1 = r1 + 3
r2 = 19
r1 = r2 + r1
```

- Create a node for each value
- Add an edge between nodes if their lifetimes overlap
- Allocate a colour(register) to each node such that if nodes are connected they have a different colour
- Optimal graph colouring is NP-complete, so approximate/heuristic algorithms are used.

Instruction Selection

- For each high level operation, decide which machine code instruction(s) to translate it to.
  - Often more than one choice
  - Best choice may depend on context

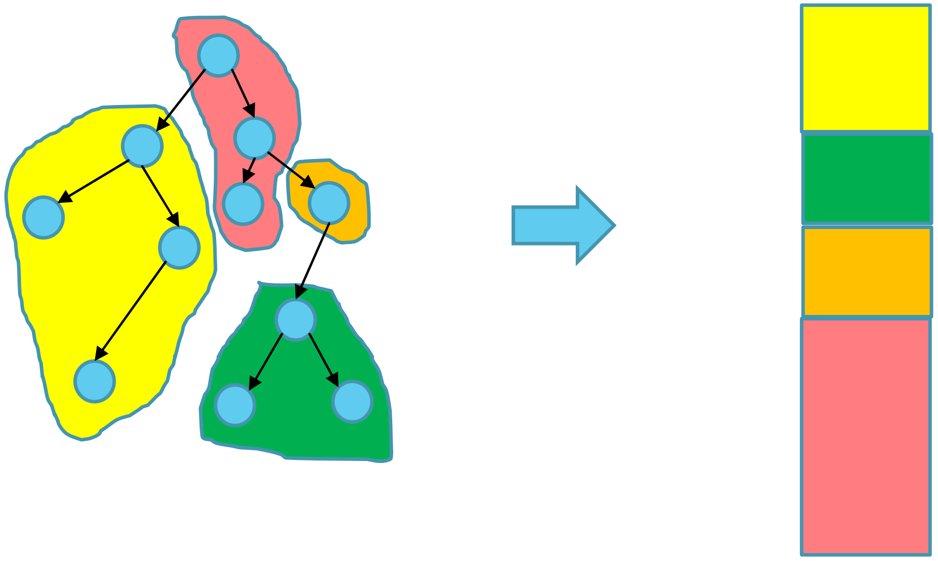
Simple Recursive Code Generation

```
class AddExpression : Expression
{
    Expression lhs, rhs;
    Operand GenCode()
    {
        Operand r1 = lhs.GenCode();
        Operand r2 = rhs.GenCode();
        emit("add %s,%s", r1, r2);
        return r1;
    }
}

class Variable : Expression
{
    string name;
    Operand GenCode()
    {
        Operand r1 = GetNewVirtualRegister();
        emit("mov %s,dword ptr [%s]", r1, name);
        return r1;
    }
}
```

Tree Matching

- Instead of replacing individual nodes in the AST with specific instructions.
- Define a bunch of *patterns* that match subtrees of the AST and generate a sequence of related instructions.
  - Need to find an (optimal) tiling of patterns that completely covers the AST



Generating Code for Control Flow

```
if (a < 7 || b == 3)
    S1
else
    S2;
S3;
```

```
    if (a < 7)  jmp L1
    if (b == 3) jmp L1
    S2
    jmp L2
L1: S1
L2: S3
```

```
S1;
for (i=0; i<10; i++)
    S2;
S3;
```

```
    S1
    i = 0
    jmp L2
L1: S2
    i++
L2: if (i < 10) jmp L1
    S3
```

Function Calls

- Runtime memory is divided into:
  - Heap – for dynamically allocated data (new/malloc)
  - Call Stack – for local variables, function parameters and return addresses.
- Activation stack contains one activation record for each active function call

```
int fib(int n)
{
        int p, q;
001:    if (n < 2)
002:        return n;
        else
        {
003:        p = fib(n-1);
004:        q = fib(n-2);
005:        return p + q;
        }
}
void main(int argc, char* argv[])
{
006:    int f = fib(2);
007:    printf("%d\n", f);
008:    return 0;
}
```

Program Counter (PC)	**006**
Stack Pointer (SP)	103

Stack

101	**argc: 1**
102	argv: 0x433536332
103	caller: 0x0
104	f:
105	
106	
107	
108	
109	
110	
111	
112	
113	

Function Calls

- Runtime memory is divided into:
  - Heap – for dynamically allocated data (new/malloc)
  - Call Stack – for local variables, function parameters and return addresses.
- Activation stack contains one activation record for each active function call

```
int fib(int n)
{
        int p, q;
001:    if (n < 2)
002:        return n;
        else
        {
003:        p = fib(n-1);
004:        q = fib(n-2);
005:        return p + q;
        }
}
void main(int argc, char* argv[])
{
006:    int f = fib(2);
007:    printf("%d\n", f);
008:    return 0;
}
```

Program Counter (PC)	**001**
Stack Pointer (SP)	106

Stack

101	**argc: 1**
102	argv: 0x433536332
103	caller: 0x0
104	f:
105	n: 2
106	caller: 0x006
107	p:
108	q:
109	
110	
111	
112	
113	

Function Calls

- Runtime memory is divided into:
  - Heap - for dynamically allocated data (new/malloc)
  - Call Stack - for local variables, function parameters and return addresses.
- Activation stack contains one activation record for each active function call

```
int fib(int n)
{
        int p, q;
001:    if (n < 2)
002:        return n;
        else
        {
003:        p = fib(n-1);
004:        q = fib(n-2);
005:        return p + q;
        }
}
void main(int argc, char* argv[])
{
006:    int f = fib(2);
007:    printf("%d\n", f);
008:    return 0;
}
```

Program Counter (PC)	**003**
Stack Pointer (SP)	106

Stack

101	**argc: 1**
102	argv: 0x433536332
103	caller: 0x0
104	f:
105	n: 2
106	caller: 0x006
107	p:
108	q:
109	
110	
111	
112	
113	

Function Calls

- Runtime memory is divided into:
  - Heap – for dynamically allocated data (new/malloc)
  - Call Stack – for local variables, function parameters and return addresses.
- Activation stack contains one activation record for each active function call

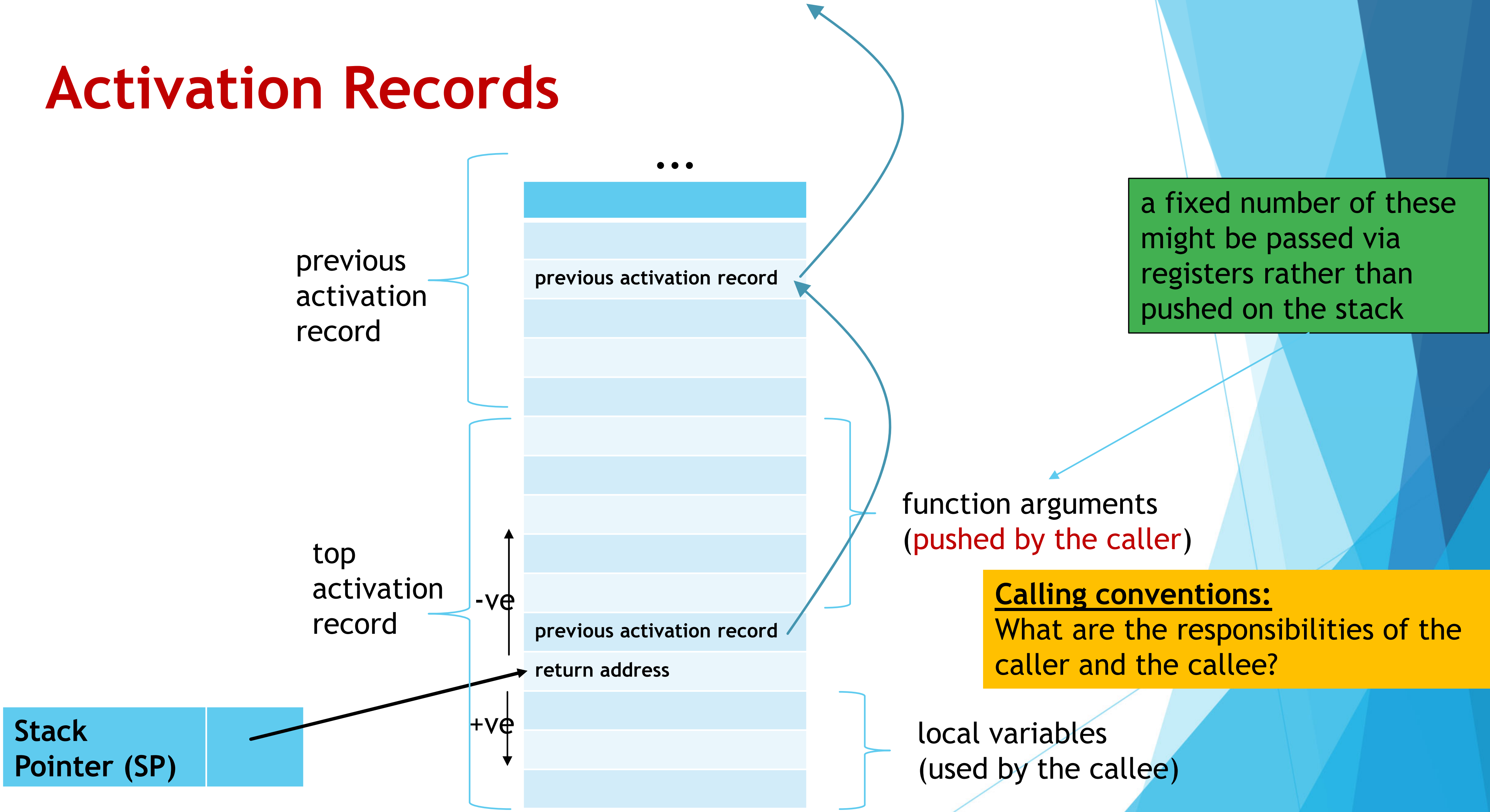
```
int fib(int n)
{
        int p, q;
001:    if (n < 2)
002:        return n;
        else
        {
003:        p = fib(n-1);
004:        q = fib(n-2);
005:        return p + q;
        }
}
void main(int argc, char* argv[])
{
006:    int f = fib(2);
007:    printf("%d\n", f);
008:    return 0;
}
```

Program Counter (PC)	**001**
Stack Pointer (SP)	106

Stack

101	**argc: 1**
102	argv: 0x433536332
103	caller: 0x0
104	f:
105	n: 2
106	caller: 0x006
107	p:
108	q:
109	n: 1
110	caller: 0x003
111	p:
112	q:
113	

Activation Records



Calling conventions:
What are the responsibilities of the caller and the callee?

Optimizations

- Common Sub-Expression Elimination (CSE).

```
x = y*2 + f(y*2 + 1);
```

- Constant Folding and value Propagation.

```
x = 2; y = a * x; z = x * 3;
```

- Loop unrolling

```
for (int i=0; i<4; i++)
    a[i] = i;
```

- Strength Reduction

```
i = x / 8;
```

- Loop invariant code motion.

```
for (int i=0; i<10; i++) {
    int a = sqrt(4) + b;
    x[i] += a;
}
```

- Unreachable/Dead code elimination

```
int foo(int x) {
    int i = sqrt(x);
    return x;
}
```

- Function Inlining

LLVM

- http://llvm.org/
- A popular open source code generation framework
  - Intermediate form is based on Static Single Assignment (SSA) form
  - Supports advanced analysis and optimizations
  - Able to generate machine code for a wide range of target architectures.
  - Instruction selection based on Tree Matching.