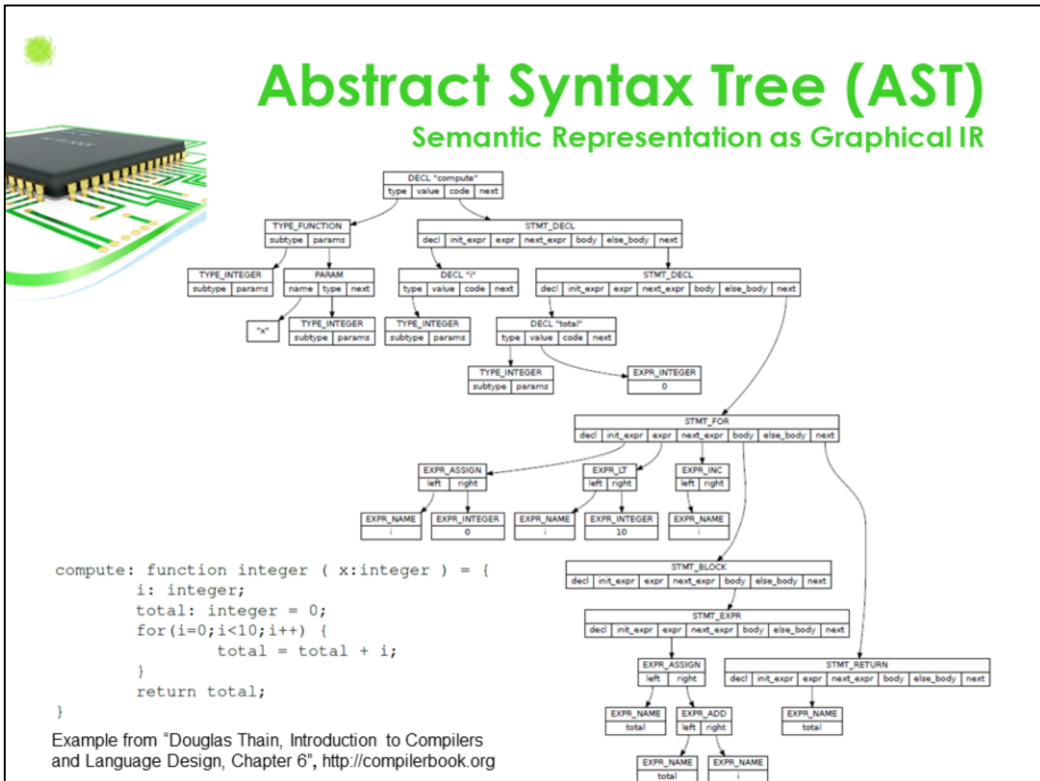


## Definition

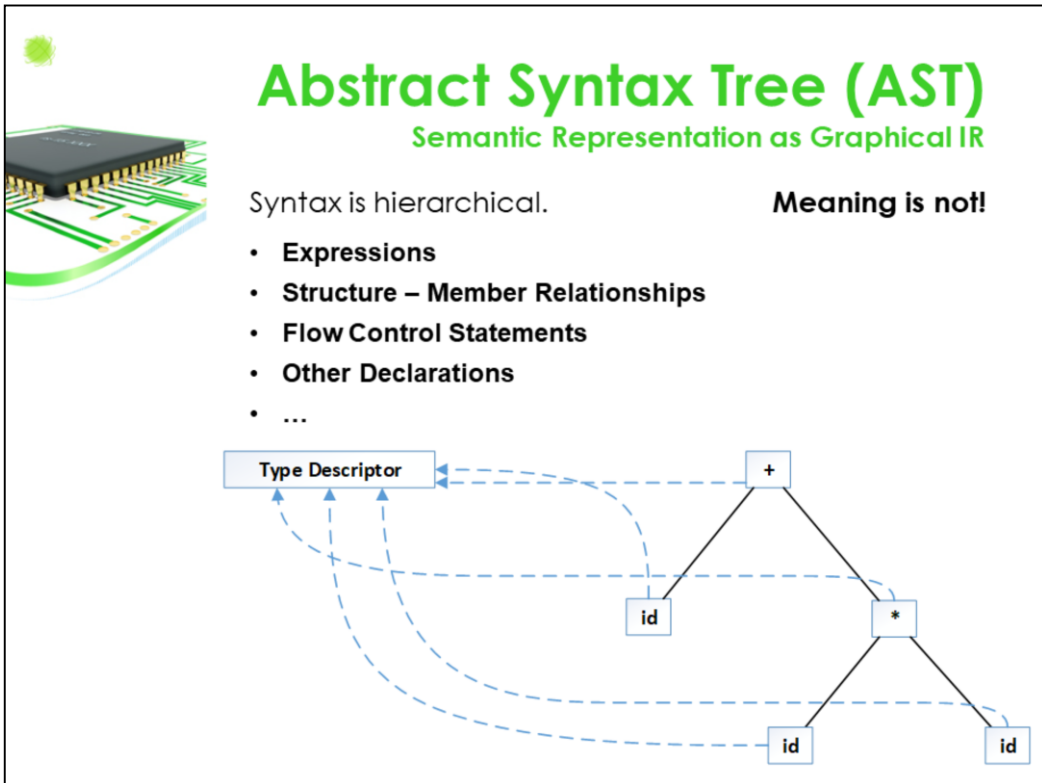
An intermediate representation is a composite derivative structure that supports generation of target code.

Extends the meaning,  
regards the target architectures.

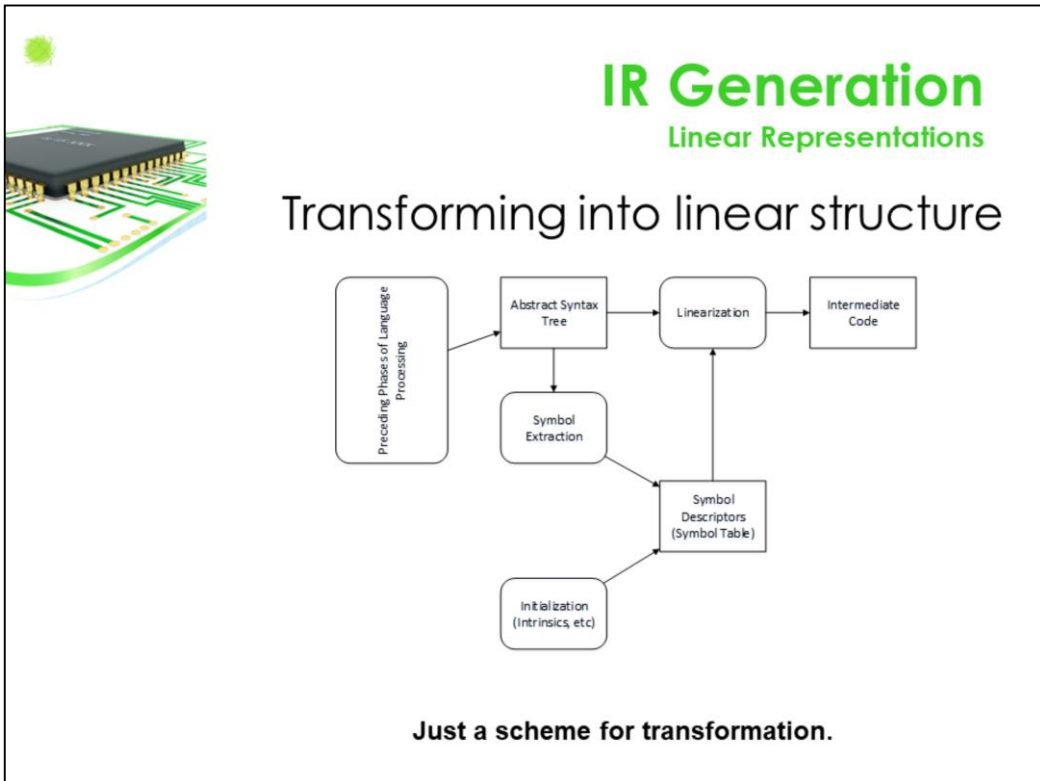
An intermediate representation uses the results of the preceding phases (lexical analysis, syntax analysis, semantic analysis) and performs more processing to make it easy generation of targets. While the structures generated by the semantic analysis phase describes what is in the input, the IR generation phase regards the target architecture so that target generation is possible.



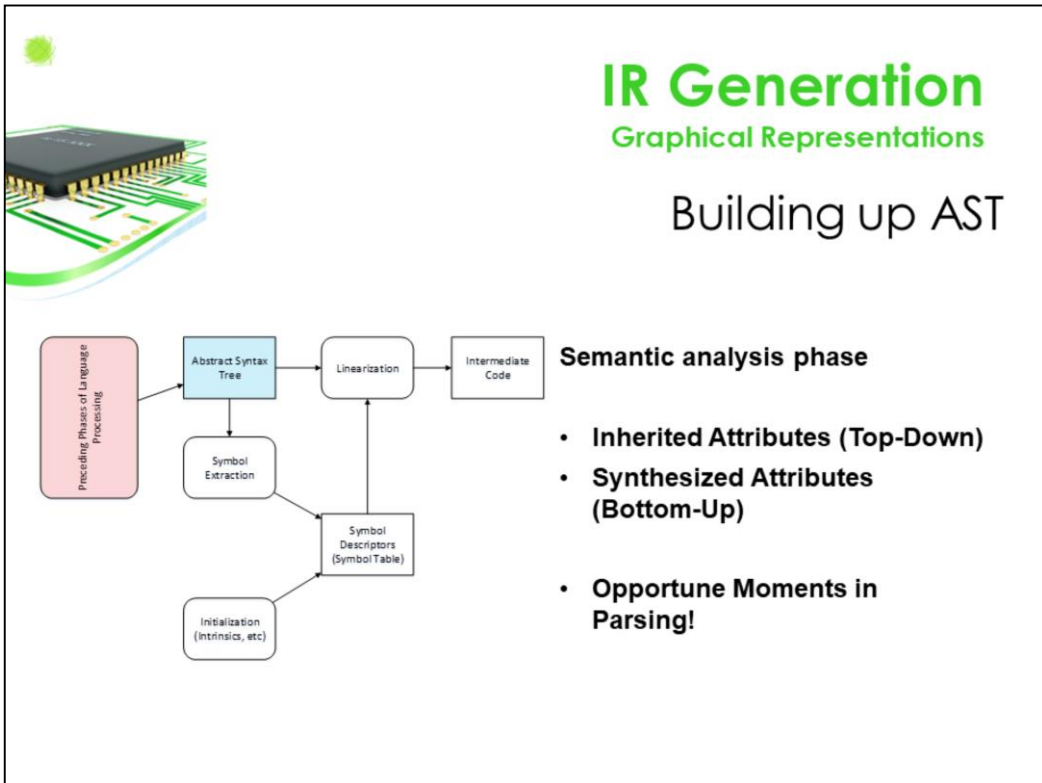
Abstract syntax tree is a component of Intermediate representation. It falls in the category of graphical IR.



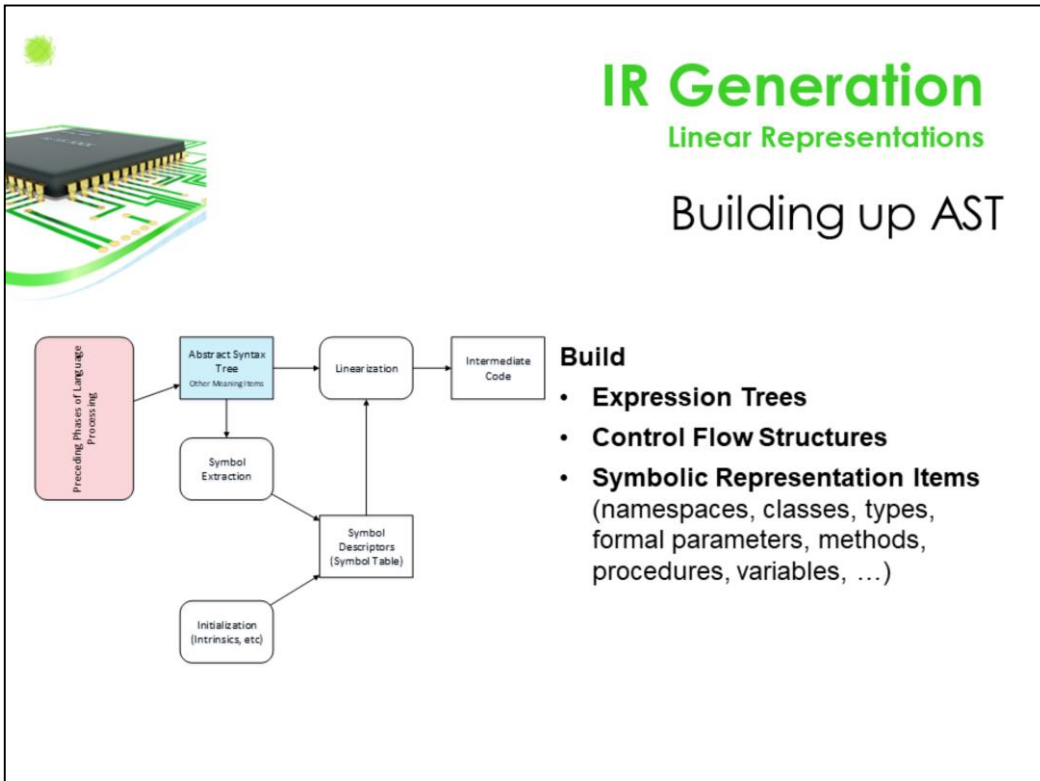
Representation of the meaning transcends the limitations of tree structure. When the input is organized as sequence of symbols, which is a one dimensional composition, the limits of the hierarchical declarative organization cannot be overcome. But, the semantics require more complex representations. This problem is commonly solved by symbolic references turning the IR into a composite derivative structure.



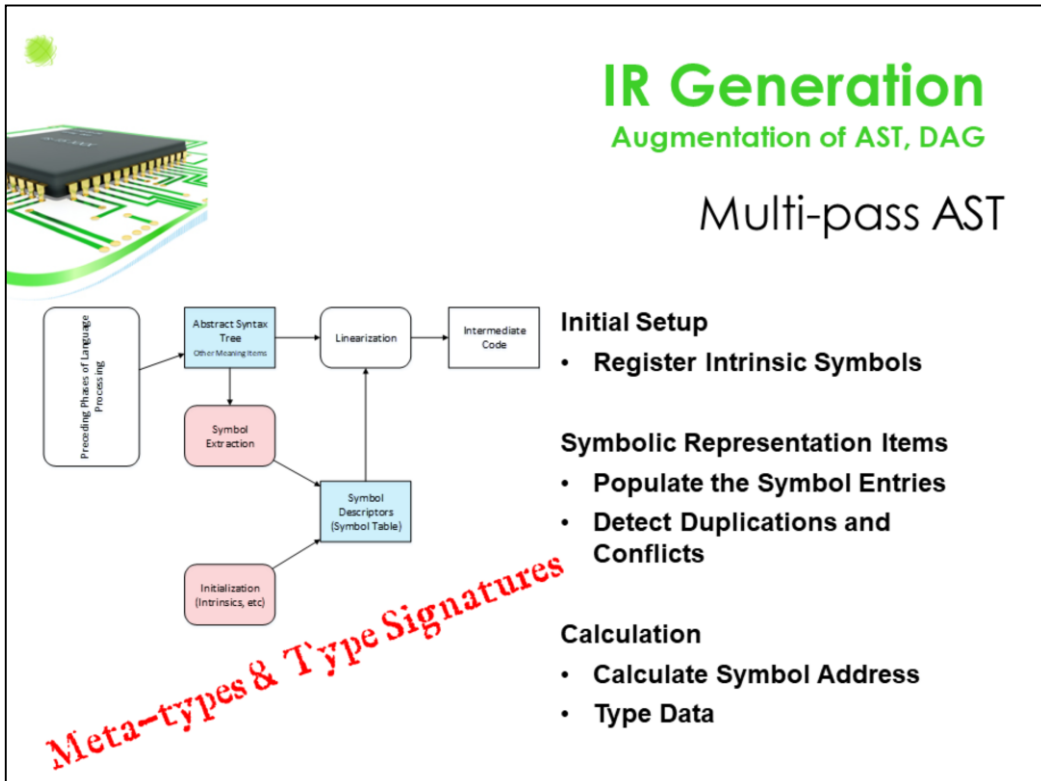
The diagram presented in this slide is a model for transforming an imperative language into an intermediate code. As pointed out earlier in this lecture alternative diagrams can be proposed and justified so long as they meet implementation specific or educational needs. The scheme proposed in this diagram describes also the implementation plan for the rest of the applied track of this course.



Building up abstract syntax tree is dependent on the parsing strategy, which can be determined by the underlying methods and tools. In bison, the capturing the semantics is achieved by integrating pseudo-variables with developer defined types and actions. In a top down scheme, use of inherited attribute, entry, and exit actions will enable alternative implementations. Or, it is possible to have the entire the parse tree built once, which incurs memory and CPU costs, and processing the whole AST.



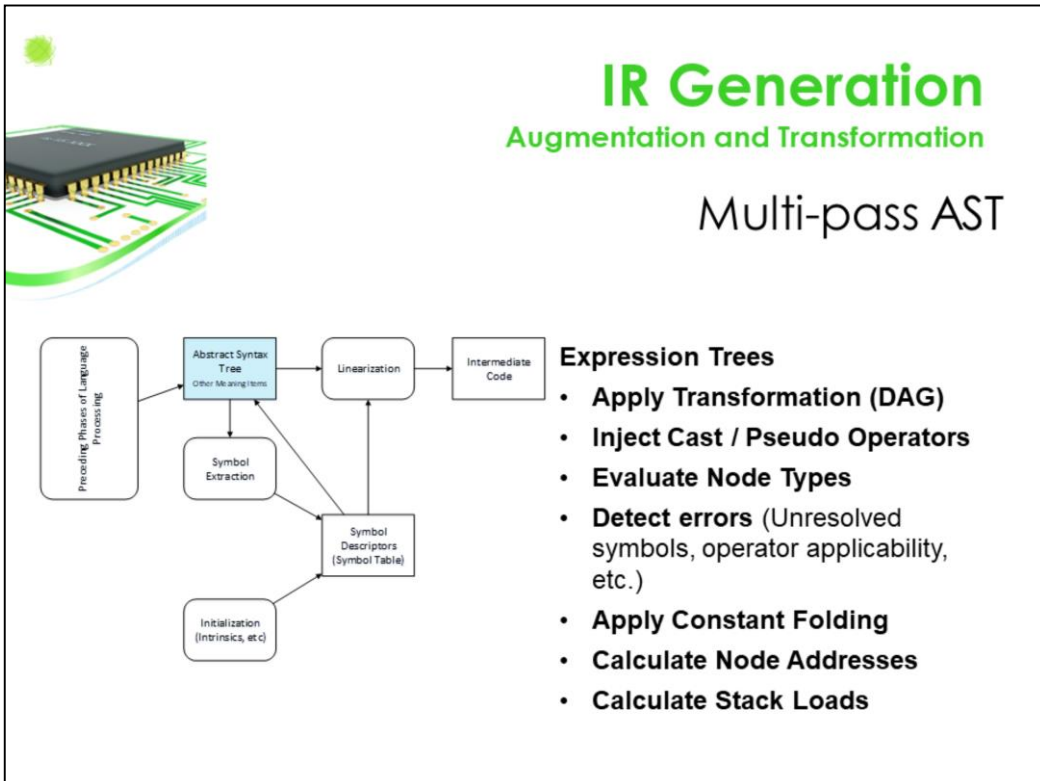
The AST may need to be accompanied by other supporting structures such as preprocessing data, compilation unit bindings (think about C, C++ modules). Availability of this structures may be critical for code generation (think about code generation for debugging and release modes), error reporting, debug information generation, runtime type information, and similar.



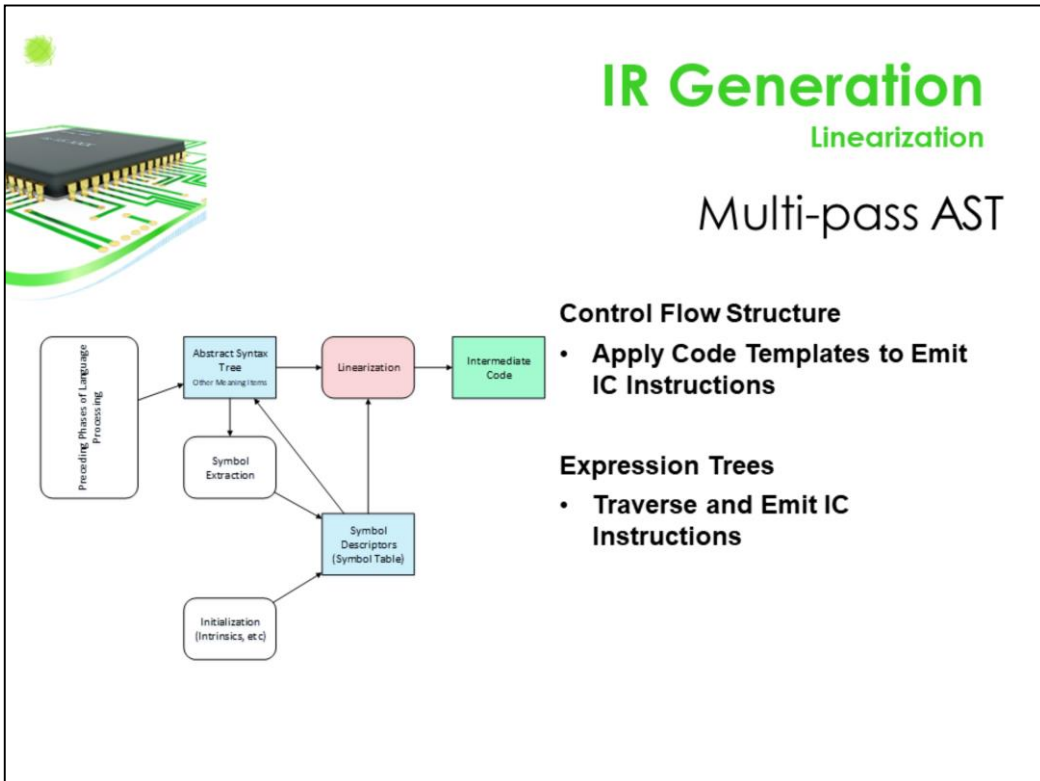
Symbolic representation will be implemented with the symbol management strategy. It may be based on maintenance of a global symbol table that is capable of representing each symbol with associated metatype (type, variable, parameter, class, etc), scope, and similar. It is also possible to manage the symbols in association with scope entities that may be found in AST. In this case, symbol collections are associated with scope entities such as execution blocks, class structures, module structures. There are advantages of scanning the AST (and other meaning items) once and building the symbol representation once. This strategy separates the mutating operations (insertion) with reading operations. So the complexities become clear and minimal. Populating, lexicographic sorting, and look up has the complexities of  $O(1)$ ,  $O(n \log n)$ , and  $O(\log n)$  respectively.

On the applied track, we will initialize the symbols table with the pre-defined functions. We will populate symbol descriptors with object (table) sub-types with simple signatures.

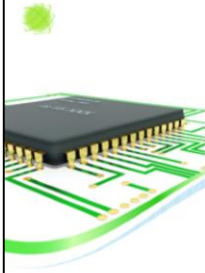




On the applied track, we will apply node type evaluation, error detection and constant folding.



On the applied track we will use case specific simple IC instructions.



# IR Generation

## Intermediate Code

### Purpose

- Architecture Independence (!)
- Standard / Improved Target Code Quality
- Reducing Cost of Overall Implementation

### Basic Types

- Stack Machine Code
- Three Address Code

### Examples

- Gimple / LLVM / JVM

Decoupling Target Code Generation from Previous Stages



# IC Generation

## Example: Gimple

```
float f( int a, int b, float x ) {  
    float y = a*x*x + b*x + 100;  
    return y;  
}
```

```
f (int a, int b, float x)  
{  
    float D.1597D.1597;  
    float D.1598D.1598;  
    float D.1599D.1599;  
    float D.1600D.1600;  
    float D.1601D.1601;  
    float D.1602D.1602;  
    float D.1603D.1603;  
    float y;  
  
    D.1597D.1597 = (float) a;  
    D.1598D.1598 = D.1597D.1597 * x;  
    D.1599D.1599 = D.1598D.1598 * x;  
    D.1600D.1600 = (float) b;  
    D.1601D.1601 = D.1600D.1600 * x;  
    D.1602D.1602 = D.1599D.1599 + D.1601D.1601;  
    y = D.1602D.1602 + 1.0e+2;  
    D.1603D.1603 = y;  
    return D.1603D.1603;  
}
```

Example from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 8", <http://compilerbook.org>



# IC Generation

Example: LLVM

```
float f( int a, int b, float x ) {  
    float y = a*x*x + b*x + 100;  
    return y;  
}
```

```
define float @f(i32 %a, i32 %b, float %x) #0 {  
    %1 = alloca i32, align 4  
    %2 = alloca i32, align 4  
    %3 = alloca float, align 4  
    %y = alloca float, align 4  
    store i32 %a, i32* %1, align 4  
    store i32 %b, i32* %2, align 4  
    store float %x, float* %3, align 4  
    %4 = load i32* %1, align 4  
    %5 = sitofp i32 %4 to float  
    %6 = load float* %3, align 4  
    %7 = fmul float %5, %6  
    %8 = load float* %3, align 4  
    %9 = fmul float %7, %8  
    %10 = load i32* %2, align 4  
    %11 = sitofp i32 %10 to float  
    %12 = load float* %3, align 4  
    %13 = fmul float %11, %12  
    %14 = fadd float %9, %13  
    %15 = fadd float %14, 1.000000e+02  
    store float %15, float* %y, align 4  
    %16 = load float* %y, align 4  
    ret float %16  
}
```

Example from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 8", <http://compilerbook.org>



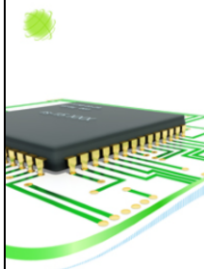
# IC Generation

Example: JVM

```
float f( int a, int b, float x ) {  
    float y = a*x*x + b*x + 100;  
    return y;  
}
```

```
0: iload 1  
1: i2f  
2: fload 3  
4: fmul  
5: fload 3  
7: fmul  
8: iload 2  
9: i2f  
10: fload 3  
12: fmul  
13: fadd  
14: ldc #2  
16: fadd  
17: fstore 4  
19: fload 4  
21: freturn
```

Example from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 8", <http://compilerbook.org>



# IC Generation

## Example: Proprietary

```
float f(int32 a, int32 b, float x)
begin
    float y=a*x*x + b*x + 100;

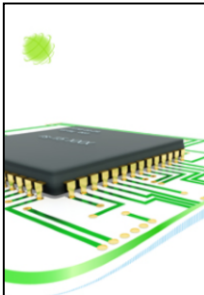
    return y;
end

0 ssr 1[1 0x00000001]
1 ssr 29[52 0x00000034]
2 psh mwp base pointer offset 0
3 pmw base pointer offset -5
4 pmw base pointer offset -3
5 cvt_F_i -2 regs
6 mul_F
7 pmw base pointer offset -3
8 mul_F
9 pmw base pointer offset -4
10 pmw base pointer offset -3
11 cvt_F_i -2 regs
12 mul_F
13 add_F
14 psh int8 100 0x64
15 cvt_F_t -1 regs
16 add_F
17 ssr 4[0 0x00000000]
18 ssr 3[1 0x00000001]
19 ssr 29[89 0x00000059]
20 pmw base pointer offset 0
21 ssr 5[1 0x00000001]
22 ssr 2[1 0x00000001]
23 rtf 3
24 hit

; Prologue f S0n(F0ni0ni0nF0n)
; Debug expression prologue y=a*x*x + b*x + 100;

; flat
; Debug expression prologue y;

; Epilogue f S0n(F0ni0ni0nF0n)
```



# IC Generation

## Multi-pass IR Traversal

### Some Basics

- Multi-Pass IR Node Traversal
- Virtual instructions and types
- Control Flow Code Templates
- Type Calculation
- Address Calculation
- Optimizations
- More...

### Some Advanced Processing

- Type System Operations
- Structured Exception Handling
- Closures
- Concurrency Related Patterns
- More ...

To generate intermediate code, the graphical IR is walked over multiple times to create the linear intermediate representations, which is much closer to the sequence of the instructions that will be generated by the back end. This, addressing many details such as code labeling, code skeletons (code templates or code-macros), virtual types / instructions, and similar. Control statements and expressions can be converted to their corresponding linear forms by using proper IR traversal methods combined with instruction labeling.



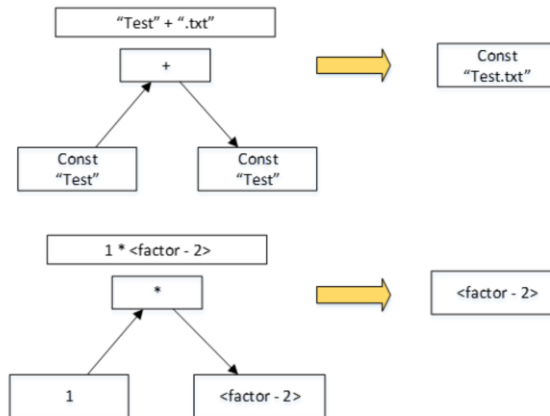


# Simple Optimizations on IR

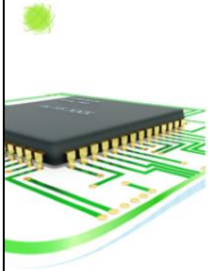
## Constant Folding

### Arithmetic Logic Operation Properties

- Constant operands
- Identity elements



Arithmetic – logic sub-expressions can be calculated during compile time resulting reduced expression trees, more efficient intermediate code. Operators coded with identity elements or constants can be processed by removing the operands and modifying the operator nodes.

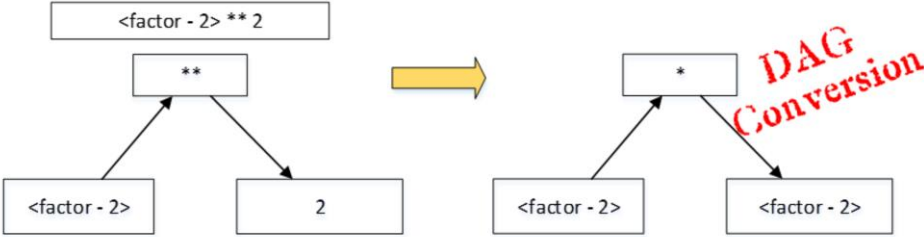


# Simple Optimizations on IR

## Strength Reduction

### Arithmetic Logic Operation Properties


- Exponentials with constants
- Multiplications with constant
- More ...



The diagram illustrates the transformation of an expression tree. On the left, the expression `<factor - 2> ** 2` is shown in a box above a tree with a root node `**` and two children: `<factor - 2>` and `2`. A yellow arrow points to the right, where the transformed tree is shown. The root node is `*`, and both children are `<factor - 2>`. A red label **DAG Conversion** is placed next to the transformed tree.

These properties may have impact on instruction selection, too.

Strength reduction can also be achieved by detecting common sub-expression. There may be subtle properties such as function volatility (if a function returns always the same value without affecting other state or not).



# Simple Optimizations on IR

## Invariant Code Motion – Code Hoisting

```

let k=0,
    m=0,
    x=random(),
    y=random();

while (m<1000)
{
    delete(num2str(x*y)+".txt");
    m=m+1;
}

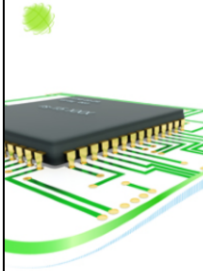
```

00004	setv	1	type:[]
00005	pop	1	type:[]
00006	call	22	type:[number]
00007	setv	2	type:[]
00008	pop	1	type:[]
00009	call	22	type:[number]
00010	setv	3	type:[]
00011	pop	1	type:[]
00012	id	1	type:[number]
00013	const	0	type:[number] 1000
00014	lt	0	type:[number]
00015	pop	1	type:[]
00016	jf	31	type:[]
00017	id	2	type:[number]
00018	id	3	type:[number]
00019	mul	0	type:[number]
00020	call	1	type:[string]
00021	const	0	type:[string] ".txt"
00022	add	0	type:[string]
00023	call	18	type:[string]
00024	pop	1	type:[]
00025	id	1	type:[number]
00026	const	0	type:[number] 1
00027	add	0	type:[number]
00028	asn	1	type:[number]
00029	pop	1	type:[]
00030	jmp	12	type:[]

Move

Example from the TQL assignment

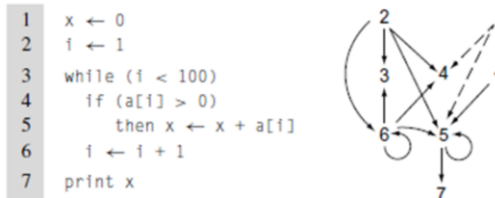
The code generator may decide to move the code fragments by analyzing the data dependencies. In this example, the loop invariant sub-expression can be moved before the code fragment that takes care of loop condition checking. To make the things right a temporary must be allocated to store the result of the moved code.



# Simple Optimizations on IR

## Invariant Code Motion – Code Hoisting


### Data Flow Analysis and Dependence Graphs



■ FIGURE 5.4 Interaction between Control Flow and the Dependence Graph.

Excerpt from "Cooper, K.D., Torczon, L.; Engineering A Compiler, 2<sup>nd</sup> Edition page 233"

Detection of the moveable code can be based on dependence graphs. This analysis reveals the code portions that can be moved outside of the inner statement bodies. Such an approach will save computation cycles when applied on alternative execution paths and loops.



# Simple Optimizations on IR

## Dead Code Elimination

```

let m=0,
    x=0,
    y=random();

x=2*m*y/(m+2);
y=y/3;
x=7/2;

```

00000	const	0	type:[number]	0
00001	setv	0	type:[]	
00002	pop	1	type:[]	
00003	const	0	type:[number]	0
00004	setv	1	type:[]	
00005	pop	1	type:[]	
00006	call	22	type:[number]	
00007	setv	2	type:[]	
00008	pop	1	type:[]	
00009	const	0	type:[number]	2
00010	id	0	type:[number]	
00011	mul	0	type:[number]	
00012	id	2	type:[number]	
00013	mul	0	type:[number]	
00014	id	0	type:[number]	
00015	const	0	type:[number]	2
00016	add	0	type:[number]	
00017	div	0	type:[number]	
00018	asn	1	type:[number]	
00019	pop	1	type:[]	
00020	id	2	type:[number]	
00021	const	0	type:[number]	3
00022	/	0	type:[number]	
00023	asn	2	type:[number]	
00024	pop	1	type:[]	
00025	const	0	type:[number]	3.5
00026	asn	1	type:[number]	
00027	pop	1	type:[]	

**Not Effective**

Example from the TQL assignment

Data analysis is a way to reveal the writes without reads. This helps elimination of ineffective calculations. Control flow graphs help to establish “control paths” which help analyze unreachable code fragments. Control path examination simulates flows with alternatives and identify the unreached statements. “The code behind return”, “The code behind break / continue”, “Ending with no return” are just a few examples you are familiar with.



# Simple Optimizations on IR

## Peephole Optimization

```
let m=0,  
    x=0,  
    y=random();  
  
x=2*m*y/(m+2);  
y=x+1;
```

00000	const	0	type:[number]	0
00001	setv	0	type:[]	
00002	pop	1	type:[]	
00003	const	0	type:[number]	0
00004	setv	1	type:[]	
00005	pop	1	type:[]	
00006	call	22	type:[number]	
00007	setv	2	type:[]	
00008	pop	1	type:[]	
00009	const	0	type:[number]	2
00010	id	0	type:[number]	
00011	mul	0	type:[number]	
00012	id	2	type:[number]	
00013	mul	0	type:[number]	
00014	id	0	type:[number]	
00015	const	0	type:[number]	2
00016	add	0	type:[number]	
00017	/	0	type:[number]	
00018	asn	1	type:[number]	
00019	pop	1	type:[]	
00020	id	1	type:[number]	
00021	const	0	type:[number]	1
00022	add	0	type:[number]	
00023	asn	2	type:[number]	
00024	pop	1	type:[]	
00025	ret	0	type:[]	

**Not  
Effective**

Example from the TQL assignment

Peephole optimization is performed on the linear IR detecting extra code generated by earlier steps. The example on this slide transcends the boundaries of expression and statement contexts.