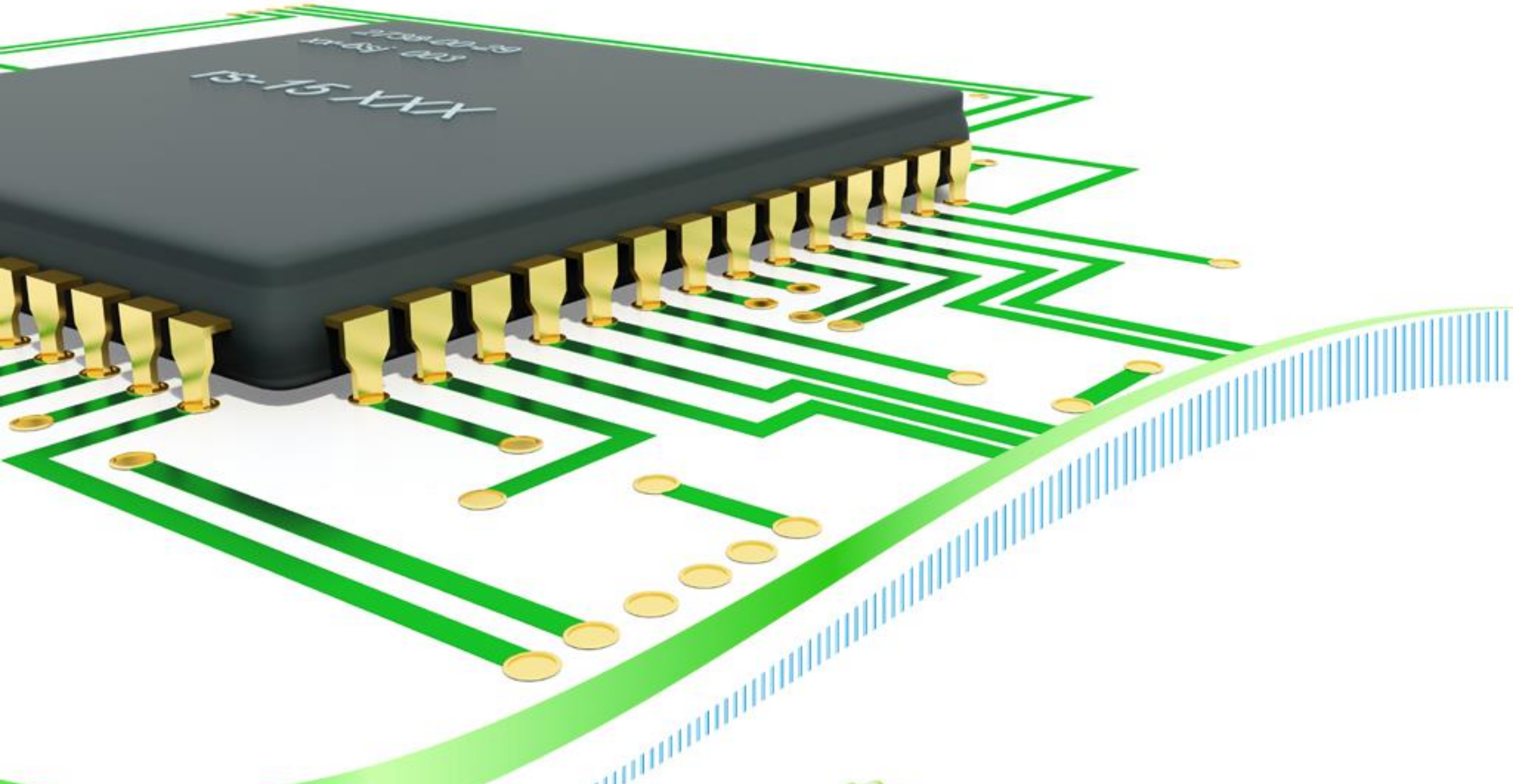
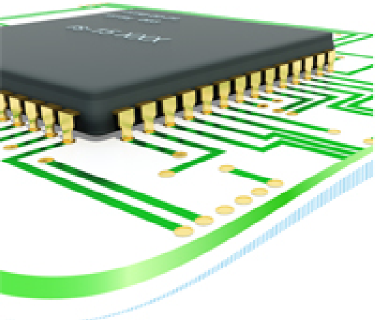


Intermediate Representations



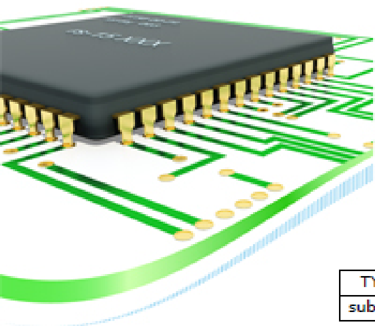
Definition

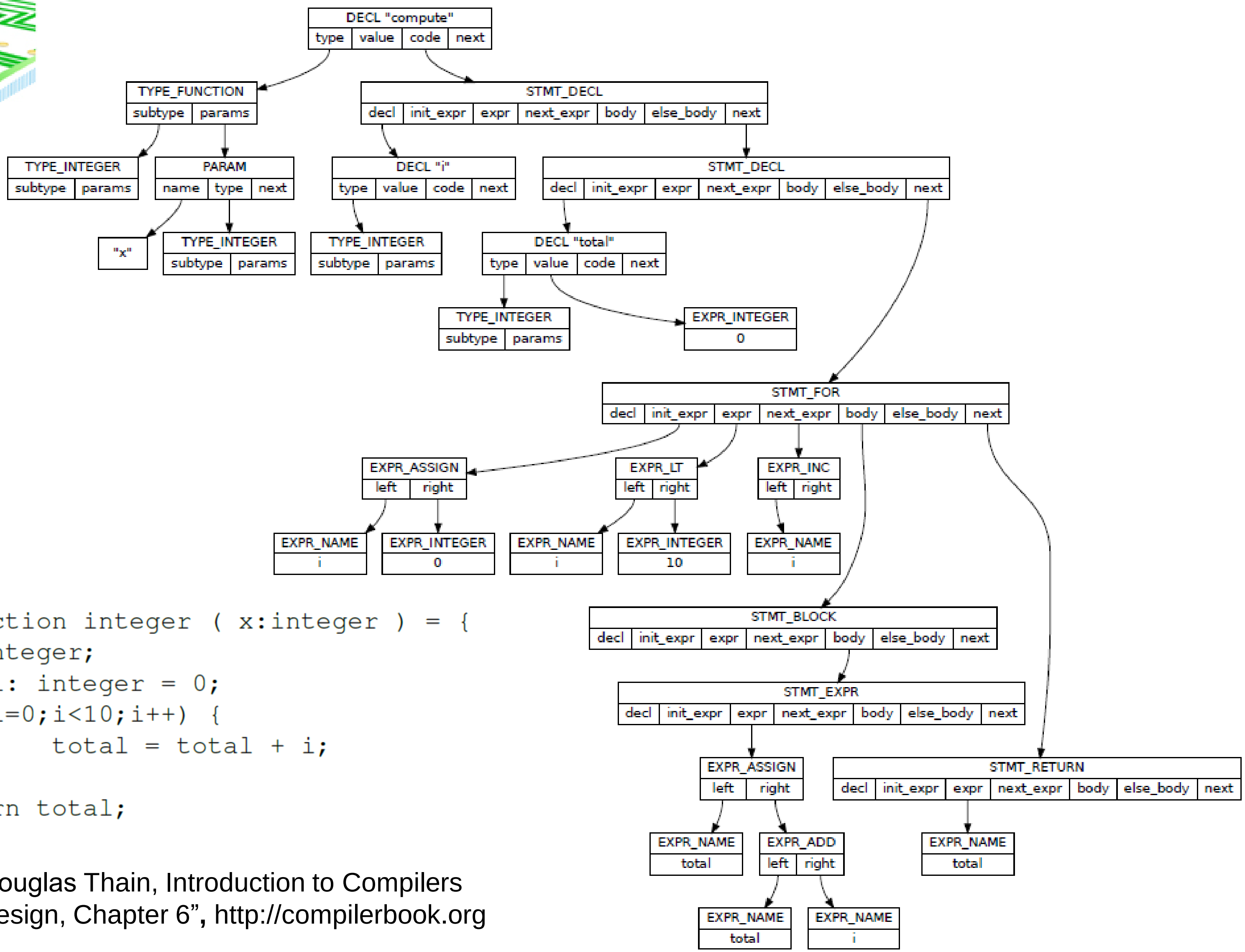
An intermediate representation is a composite derivative structure that supports generation of target code.

Extends the meaning,
regards the target architectures.

Abstract Syntax Tree (AST)

Semantic Representation as Graphical IR






```
compute: function integer ( x:integer ) = {
        i: integer;
        total: integer = 0;
        for(i=0;i<10;i++) {
                total = total + i;
        }
        return total;
}
```

Example from “Douglas Thain, Introduction to Compilers and Language Design, Chapter 6”, http://compilerbook.org

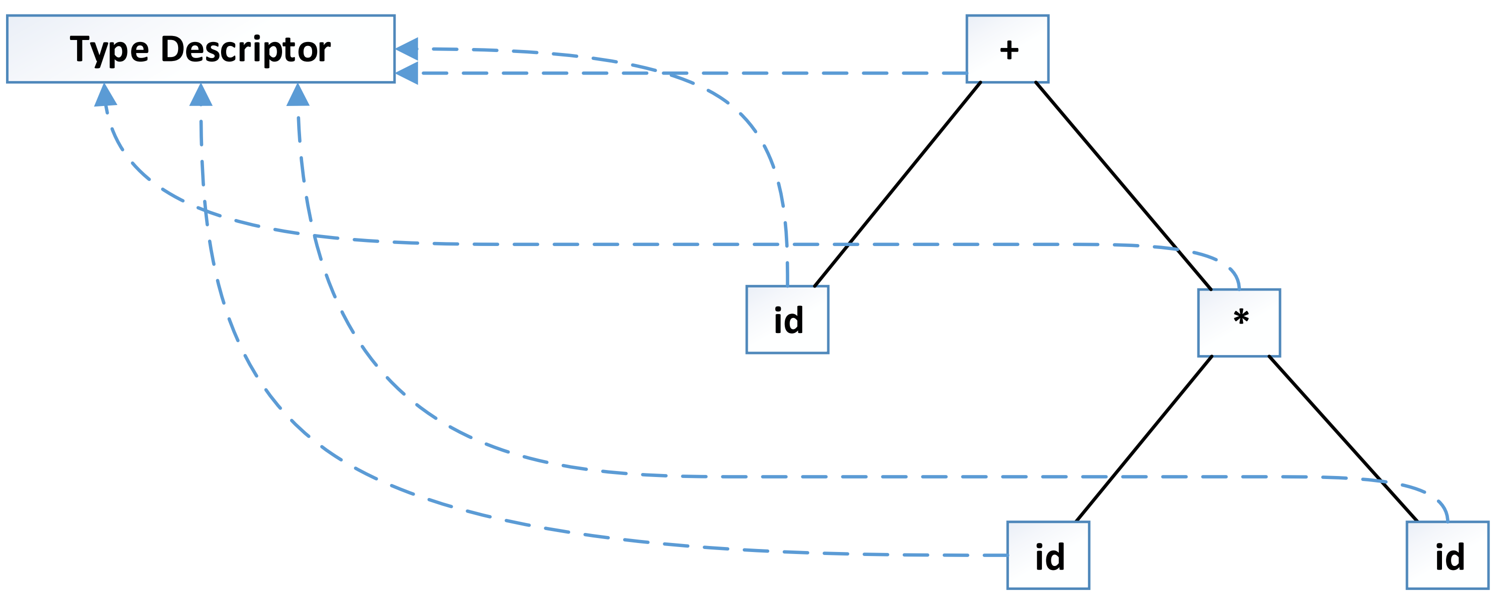
Abstract Syntax Tree (AST)

Semantic Representation as Graphical IR

Syntax is hierarchical.

Meaning is not!

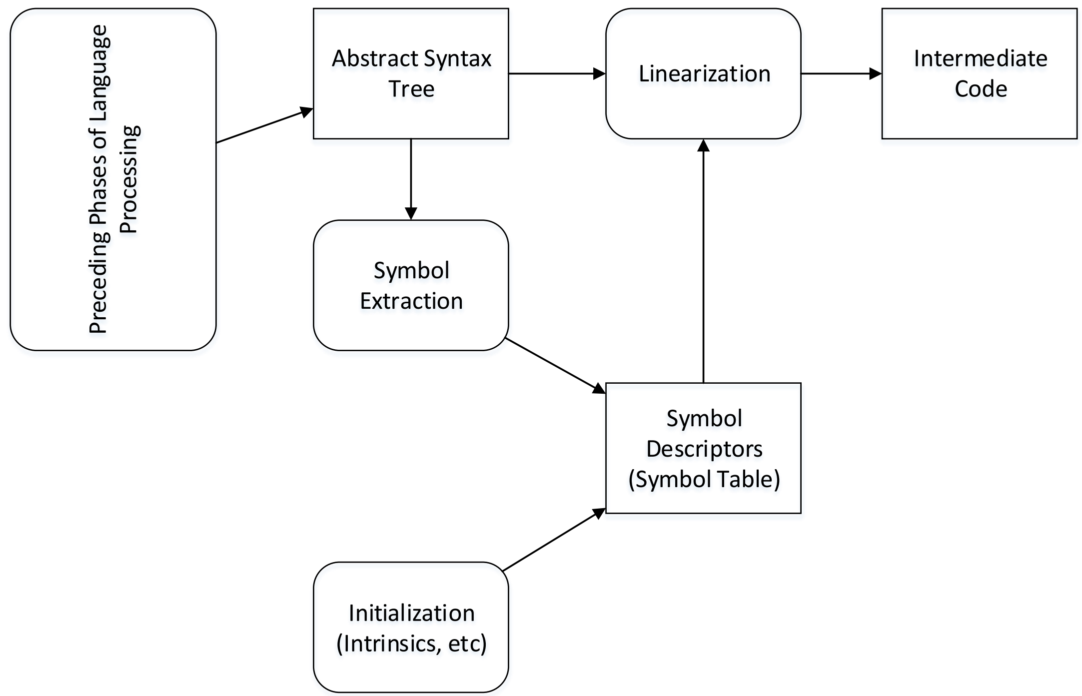
- **Expressions**
- **Structure – Member Relationships**
- **Flow Control Statements**
- **Other Declarations**
- **…**



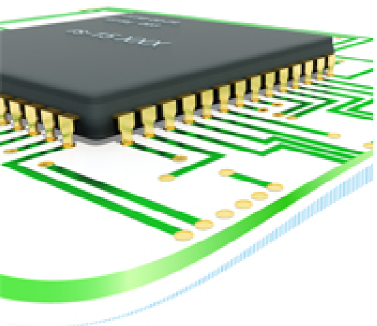
IR Generation

Linear Representations

Transforming into linear structure



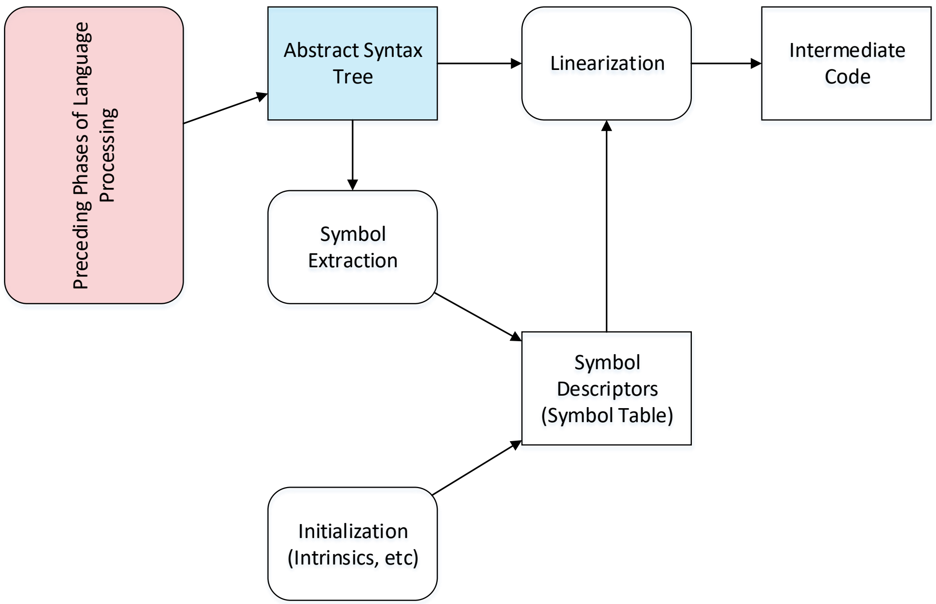
Just a scheme for transformation.



IR Generation

Graphical Representations

Building up AST



Semantic analysis phase

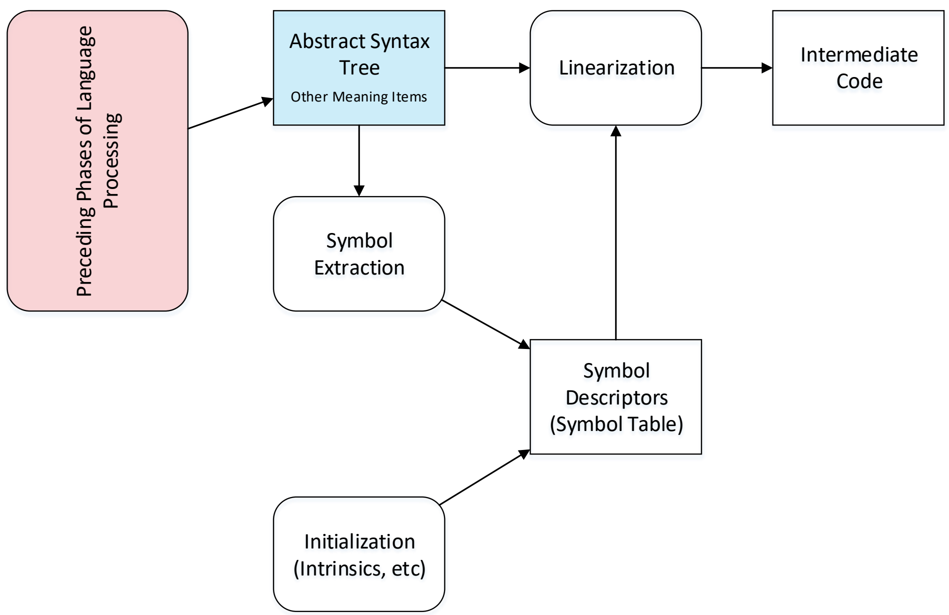
- **Inherited Attributes (Top-Down)**
- **Synthesized Attributes (Bottom-Up)**

- **Opportune Moments in Parsing!**

IR Generation

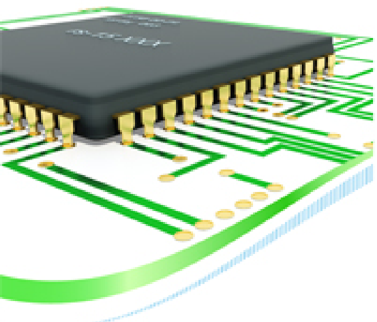
Linear Representations

Building up AST



Build

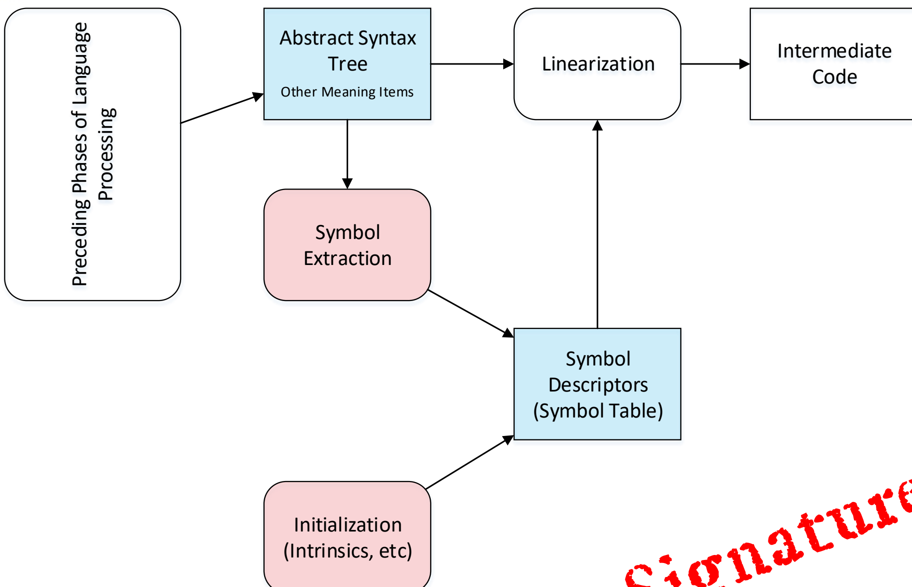
- **Expression Trees**
- **Control Flow Structures**
- **Symbolic Representation Items** (namespaces, classes, types, formal parameters, methods, procedures, variables, …)



IR Generation

Augmentation of AST, DAG

Multi-pass AST



Initial Setup

- **Register Intrinsic Symbols**

Symbolic Representation Items

- **Populate the Symbol Entries**
- **Detect Duplications and Conflicts**

Calculation

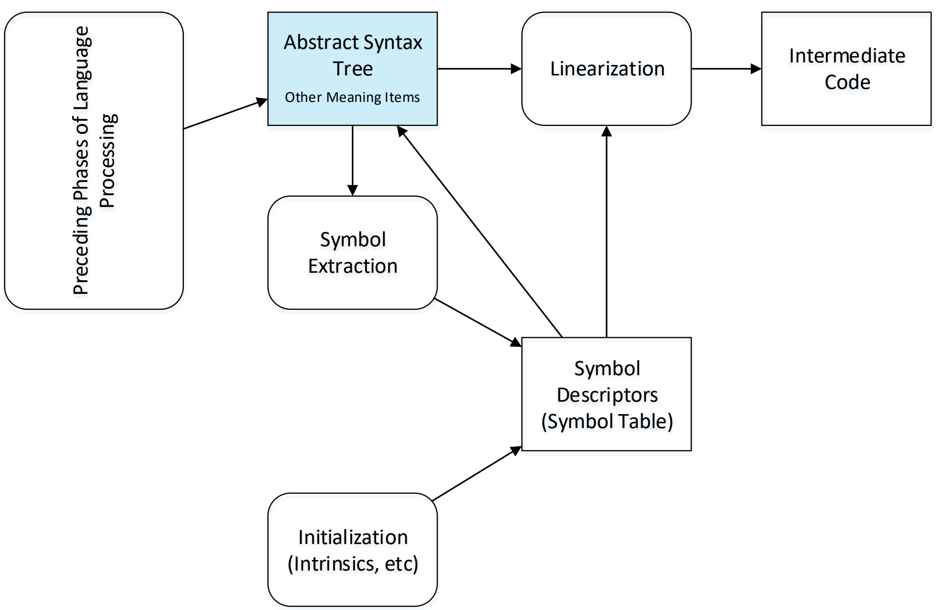
- **Calculate Symbol Address**
- **Type Data**

Meta-types & Type Signatures

IR Generation

Augmentation and Transformation

Multi-pass AST

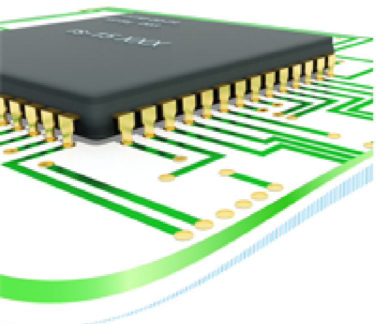


Expression Trees

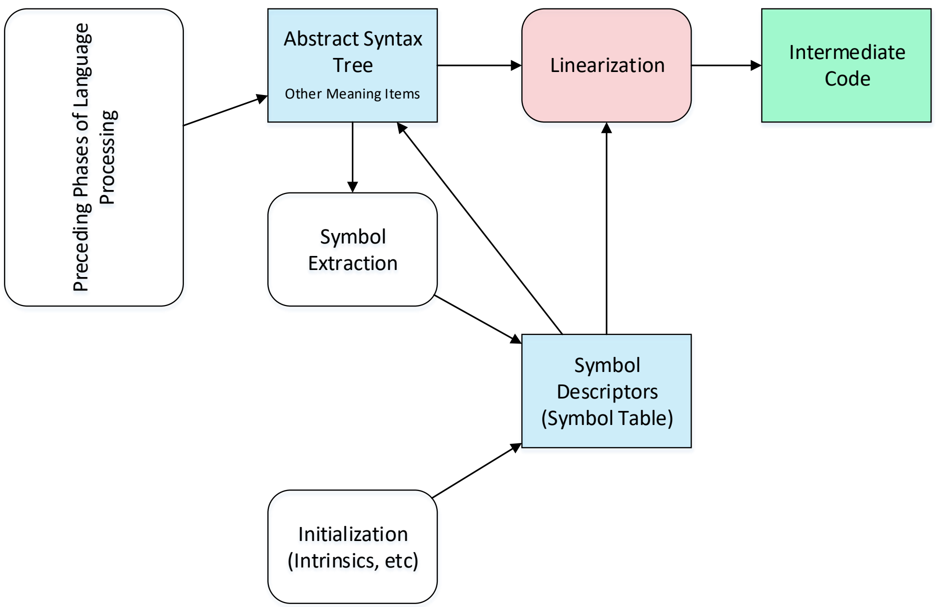
- **Apply Transformation (DAG)**
- **Inject Cast / Pseudo Operators**
- **Evaluate Node Types**
- **Detect errors** (Unresolved symbols, operator applicability, etc.)
- **Apply Constant Folding**
- **Calculate Node Addresses**
- **Calculate Stack Loads**

IR Generation

Linearization



Multi-pass AST

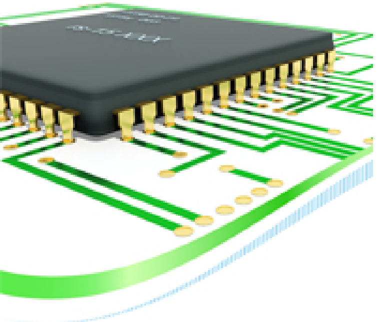


Control Flow Structure

- **Apply Code Templates to Emit IC Instructions**

Expression Trees

- **Traverse and Emit IC Instructions**



IR Generation

Intermediate Code

Purpose

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

Basic Types

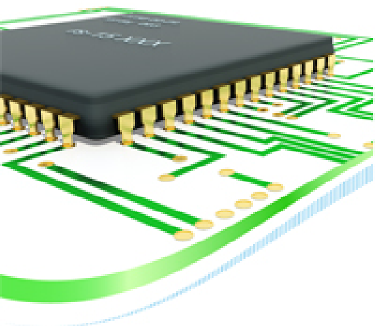
- **Stack Machine Code**
- **Three Address Code**

Examples

- **Gimple / LLVM / JVM**

IC Generation

Example: Gimple



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

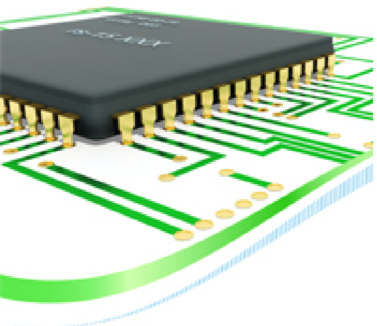
```
f (int a, int b, float x)
{
  float D.1597D.1597;
  float D.1598D.1598;
  float D.1599D.1599;
  float D.1600D.1600;
  float D.1601D.1601;
  float D.1602D.1602;
  float D.1603D.1603;
  float y;

  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

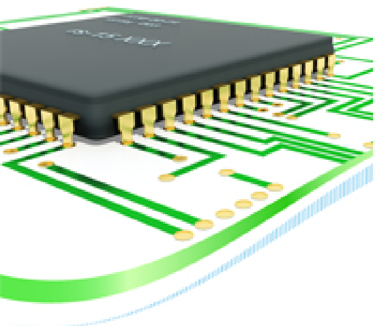
```
float f( int a, int b, float x ) {
    float y = a*x*x + b*x + 100;
    return y;
}
```

```
 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

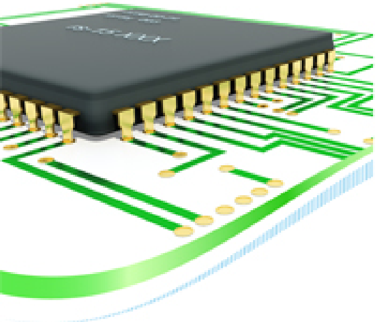
Example: Proprietary



```
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]                          ; Prologue f S0n(F0ni0ni0nF0n)
    1 ssr 29[52 0x00000034]                        ; Debug expression prologue y=a*x*x + b*x + 100;
    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]                          ; flat
   18 ssr 3[1 0x00000001]
   19 ssr 29[89 0x00000059]                        ; Debug expression prologue y;
   20 pmw base pointer offset 0
   21 ssr 5[1 0x00000001]
   22 ssr 2[1 0x00000001]                          ; Epilogue f S0n(F0ni0ni0nF0n)
   23 rtf 3
   24 hlt
```



IC Generation

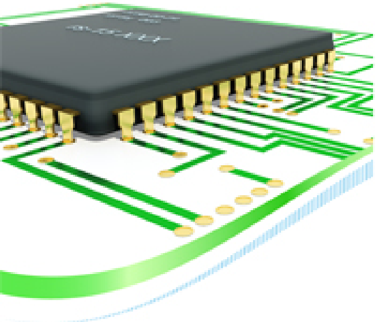
Multi-pass IR Traversal

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 …**

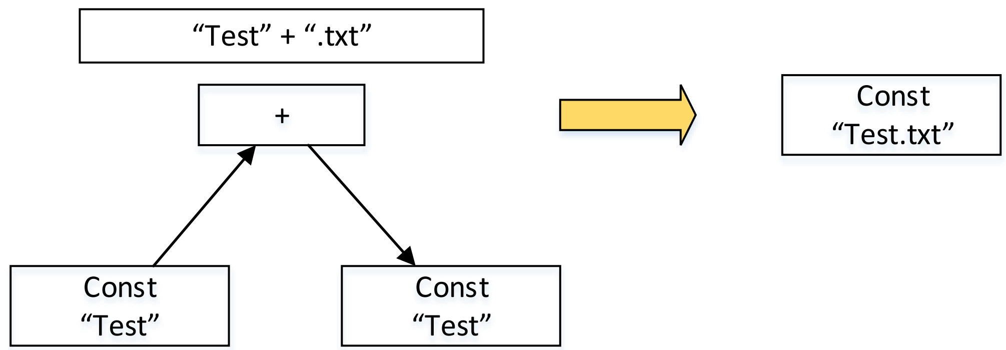


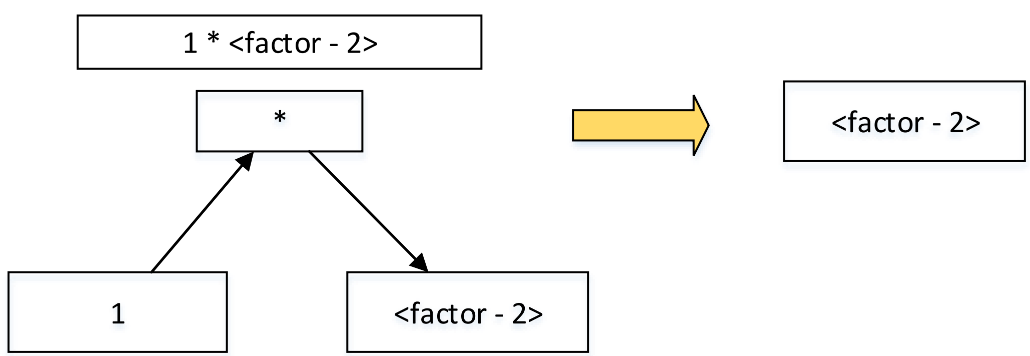
Simple Optimizations on IR

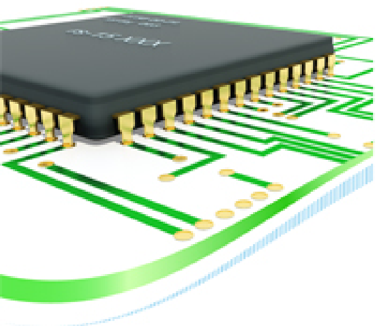
Constant Folding

Arithmetic Logic Operation Properties

- **Constant operands**
- **Identity elements**





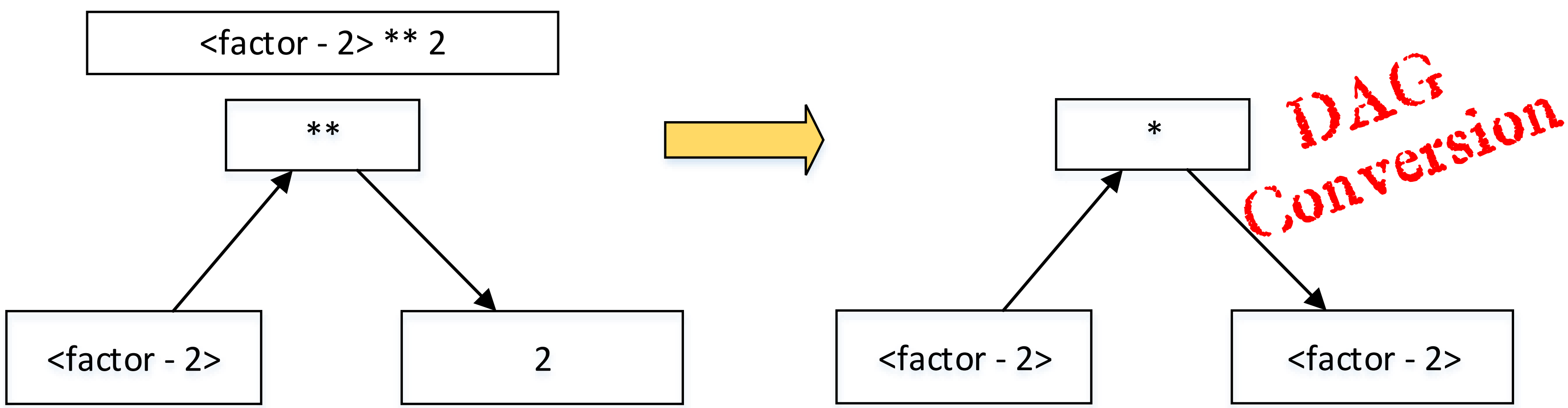


Simple Optimizations on IR

Strength Reduction

Arithmetic Logic Operation Properties

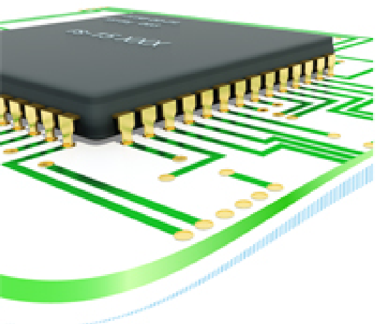
- **Exponentials with constants**
- **Multiplications with constant**
- **More …**



These properties may have impact on instruction selection, too.

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;
}
```

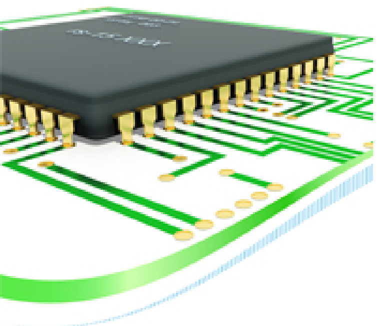
```
00004 setv      1 type:[]
00005 pop       1 type:[]
00006 call     22 type:[number]
00007 setv      2 type:[]
00008 pop       1 type:[]
00009 call     22 type:[number]
00010 setv      3 type:[]
00011 pop       1 type:[]
00012 id        1 type:[number]
00013 const     0 type:[number] 1000
00014 lt        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

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;
```

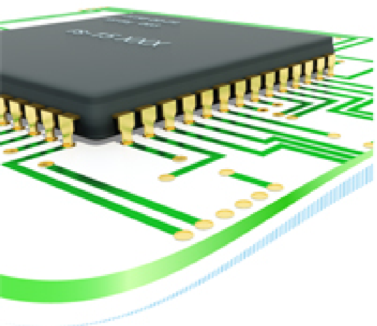
```
00000 const      0 type:[number] 0
00001 setv       0 type:[]
00002 pop        1 type:[]
00003 const      0 type:[number] 0
00004 setv       1 type:[]
00005 pop        1 type:[]
00006 call      22 type:[number]
00007 setv       2 type:[]
00008 pop        1 type:[]
00009 const      0 type:[number] 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

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