5 Intermediate Representations

Chapter Contents

Intermediated Representations

As the compiler derives knowledge about the code it compiles, it must convey that information from one pass to another.

- **intermediate representation** (IR) passes this knowledge along compilation phases
- either a single or a series of IRs may be used as program code is transformed

To record all of the detail that must be encoded, most compilers augment the IR with **tables** and **sets** that record additional information. We consider these tables part of the IR.

Selecting an appropriate IR requires an understanding of the source language, the target machine, and the properties of the applications that the compiler will translate.

A Taxonomy of Intermediate Representations

We will organize our discussion of irs along three axes

- structural organization
- level of abstraction
- naming discipline

In general, these three attributes are **independent**. Most combinations of organization, abstraction, and naming have been used in some compiler.

A Taxonomy of Intermediate Representations

IRs fall into three **structural categories**

- **Graphical IRs** encode the compiler's knowledge in a graph
  Algorithms are expressed in terms of graphical objects: nodes, edges, lists, or trees
- **Linear IRs** resemble pseudo-code for some abstract machine
  The algorithms iterate over simple, linear sequences of operations.
- **Hybrid IRs** combine elements of both graphical and linear IRs
  A common hybrid representation uses a low-level linear IR to represent blocks of straight-line code and a graph to represent the flow of control among those blocks

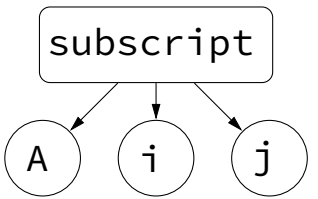
A Taxonomy of Intermediate Representations

IRs can represent operations at different **levels of abstraction**

- **near-source representation**
  A single node might represent an array access or a procedure call
- **low-level representation**
  Several IR operations together form a single source-level operation

A Taxonomy of Intermediate Representations

Example Assume that `A[1...10, 1...10]` is an array of four-byte elements stored in row-major order. Consider how the compiler might represent the array reference `A[i,j]` in a source-level tree and in ILOC.




```
subI   r_i, 1    ⇒  r_1
multI  r_1, 10   ⇒  r_2
subI   r_j, 1    ⇒  r_3
add    r_2, r_3  ⇒  r_4
multI  r_4, 4    ⇒  r_5
loadI  @A        ⇒  r_6
add    r_5, r_6  ⇒  r_7
load   r_7       ⇒  r_ij
```

Note In the source-level tree, the compiler can easily recognize the computation as an array reference. However, the ILOC code lets the compiler optimize details that remain implicit in the source-level tree.

A Taxonomy of Intermediate Representations

IRs can use different **naming schemes** to represent values in the code

- **unique names**: each expression gets its own unique name
- **name reuse**: if possible, names are reused

$$
\begin{aligned}
t_1 &\leftarrow b \\
t_2 &\leftarrow 2 \times t_1 \\
t_3 &\leftarrow a \\
t_4 &\leftarrow t_3 - t_2
\end{aligned}
\qquad\qquad
\begin{aligned}
t_1 &\leftarrow b \\
t_1 &\leftarrow 2 \times t_1 \\
t_2 &\leftarrow a \\
t_1 &\leftarrow t_2 - t_1
\end{aligned}
$$

Note If the subexpression $2 \times b$ has a unique name, other instances of $2 \times b$ could be replaced with a reference to the existing value. However, using fewer names leads to smaller compile-time data structures and faster compilation.

Graphical IRs

Many compilers use IRs that represent the underlying code as a graph.

All graphical IRs consist of nodes and edges, but differ in their **level of abstraction**, in the **relationship** between the graph and the code, and in the **structure** of the graph.

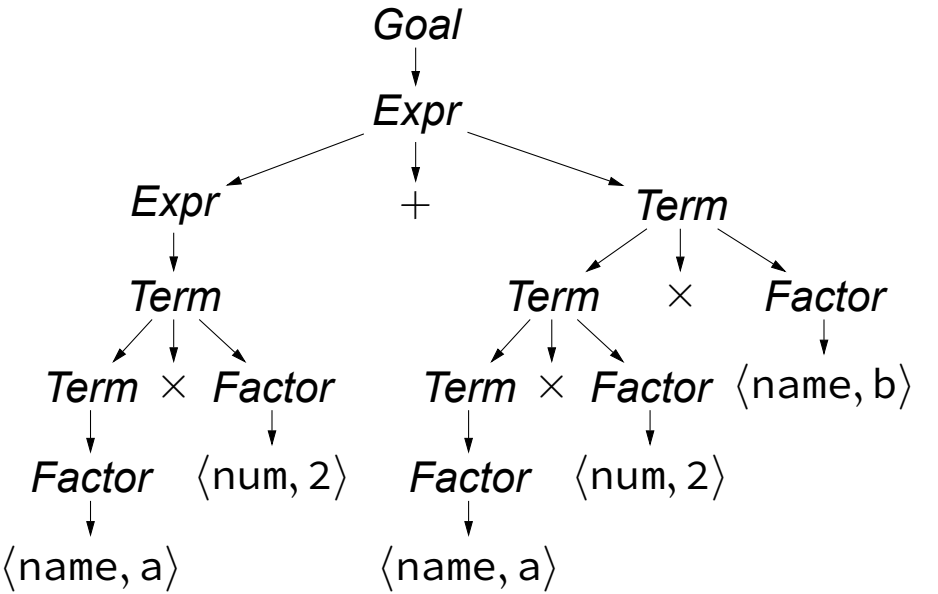
Examples

1. Syntax-Related Trees
   - Parse Trees
   - Abstract Syntax Trees
   - Directed Acyclic Graphs
2. Graphs
   - Control-Flow Graph
   - Dependence Graph
   - Call Graph

Parse Trees

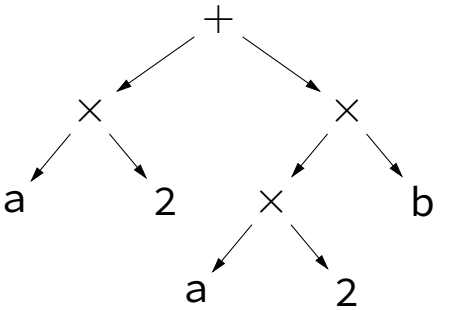
The **parse tree** is a graphical representation for complete derivation, with a node for each grammar symbol in the derivation.

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr* $+$ *Term*
2		$\mid$	*Expr* $-$ *Term*
3		$\mid$	*Term*
4	*Term*	$\rightarrow$	*Term* $\times$ *Factor*
5		$\mid$	*Term* $\div$ *Factor*
6		$\mid$	*Factor*
7	*Factor*	$\rightarrow$	(*Expr*)
8		$\mid$	num
9		$\mid$	name



Abstract Syntax Trees

The **abstract syntax tree** (AST) is a contraction of the parse tree that omits most nodes for nonterminals, but retains its essential structure.



The AST is a near-source-level representation. Because of its rough correspondence to a parse tree, the parser can build an AST directly.

ASTs have been used in many practical compiler systems.

Directed Acyclic Graphs

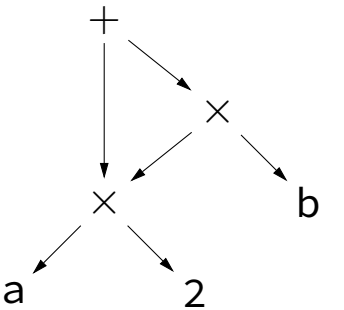
A **directed acyclic graph** (DAG) is a contraction of the AST that avoids this duplication. Identical subtrees are instantiated once, with multiple parents.

DAGs are used in real systems for **two** reasons

- reduce the memory footprint of compiler
- expose and prevent redundancies in compiled code

However, the compiler must **prove** that a shared expression's value cannot change between uses.

If the expression contains neither assignment nor calls to other procedures, the proof is easy.



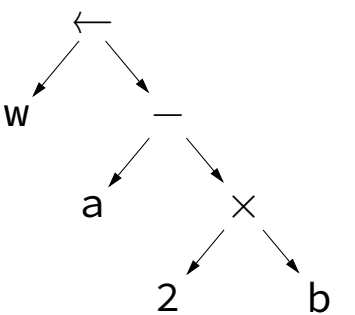
Level of Abstraction

Apart from source-level trees, compilers also use **low-level** trees. Tree-based techniques for optimization and code generation may require such detail.

Example The AST for the statement $\mathtt{w} \leftarrow \mathtt{a} - 2 \times \mathtt{b}$ lacks much of the detail needed to translate it into assembly code.

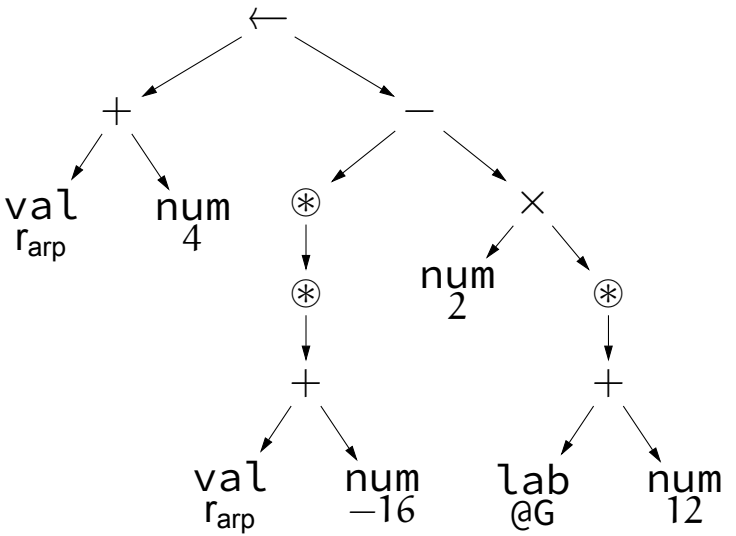
A low-level tree can make that detail explicit by introducing **new** node types.

- `val` nodes represent values already in a register
- `num` nodes represent known constants
- `lab` nodes represent an assembly-level label
- $\circledast$ is an operator that dereferences a value



Level of Abstraction

The low-level tree reveals the address calculations for the three variables.



Note

- w is stored at offset 4 **after** the pointer in r_{arp}
- r_{arp} holds the pointer to the **data area** for the current procedure
- double dereferencing of a shows that it is a call-by-reference parameter accessed through a pointer stored 16 bytes **before** r_{arp}
- b is stored at offset 12 after the label @G

Graphs

Trees provide a natural representation for the grammatical structure of the source code discovered by parsing.

Their rigid structure makes them less useful for representing other properties of programs

- flow of control between blocks of a program
- dependences of uses of a value and its definition
- relationships between procedure calls

To model these aspects of program behavior, compilers often use more **general graphs** as IRs.

Note The DAG introduced on Slide 240 is one example of a graph.

Control-Flow Graph

The simplest unit of control flow in a program is a **basic block**

- maximal length sequence of straightline, or branch-free, code
- sequence of operations that always execute together, unless an excaption is raised
- control always enters a basic block at its first operation and exits at its last operation

Control-Flow Graph

A control-flow graph (CFG) models the flow of control between the basic blocks in a program. A CFG is a directed graph $G = (N, E)$.

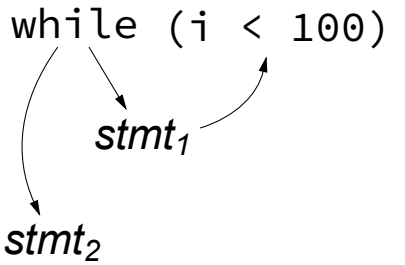
- each node $n \in N$ corresponds to a basic block
- each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from block n_i to block n_j

We assume that each CFG has a unique entry node n_0 and a unique exit node n_f.

Control-Flow Graph

Example Consider the following CFG for a `while` loop.

```
while (i < 100)
  begin
    stmt1
  end
stmt2
```

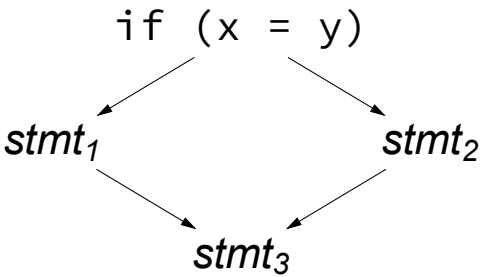


The edge from $stmt_1$ back to the loop header creates a **cycle**. The AST for this fragment would be acyclic.

Control-Flow Graph

Example For an `if-then-else` construct, the CFG is acyclic.

```
if (x = y)
  then stmt1
  else stmt2
stmt3
```



The CFG shows that control always flows from $stmt_1$ and $stmt_2$ to $stmt_3$. In an AST, that connection is implicit, rather than explicit.

Control-Flow Graph

Compilers typically use a CFG **in conjunction** with another IR

- graph represents the relationships among blocks
- operations inside a block are represented with another IR

Using **single-statement blocks**, *i.e.*, code blocks that corresponds to a single source-level statement, as nodes can simplify algorithms for analysis and optimization.

Many parts of the compiler rely on a CFG, either explicitly or implicitly

- control-flow analysis
- instruction scheduling
- global register allocation

Dependence Graph

Compilers also use graphs to encode the **flow of values** from the point where a value is created, a definition, to any point where it is used, a use.

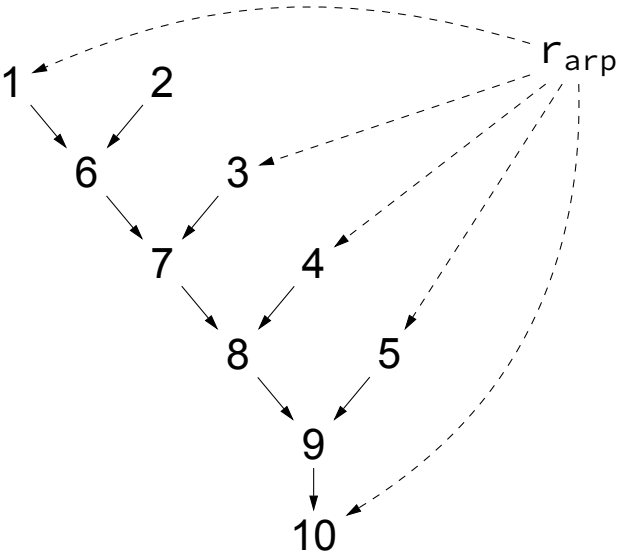
A **data dependence graph** embodies this relationship

- nodes represent operations that may contain both definitions and uses
- edges connect two nodes, one that defines a value and another that uses it

We draw dependence graphs with edges that run **from** definition **to** use.

Dependence Graph

```
 1 loadAI   r_arp, @a ⇒ r_a
 2 loadI    2         ⇒ r_2
 3 loadAI   r_arp, @b ⇒ r_b
 4 loadAI   r_arp, @c ⇒ r_c
 5 loadAI   r_arp, @d ⇒ r_d
 6 mult     r_a, r_2  ⇒ r_a
 7 mult     r_a, r_b  ⇒ r_a
 8 mult     r_a, r_c  ⇒ r_a
 9 mult     r_a, r_d  ⇒ r_a
10 storeAI  r_a       ⇒ r_arp, @a
```



Note Uses of r_{arp} (starting address of the local data area) refer to its implicit definition at the start of the procedure and are shown with dashed lines.

Dependence Graph

The edges in the graph represent real **constraints** on the sequencing of operations: a value cannot be used until it has been defined.

However, the dependence graph does not fully capture the program's control flow.

Example The graph on the previous slide requires that 1 and 2 precede 6. Nothing, however, requires that 1 or 2 precedes 3.

Many execution sequences preserve the dependences shown in the code, including

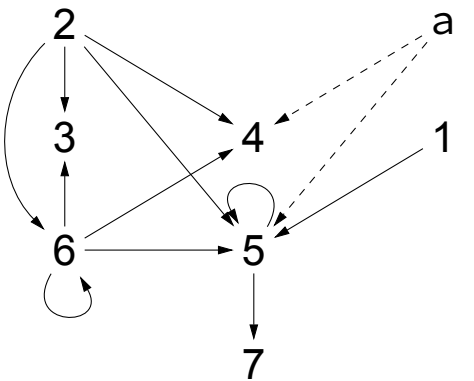
- $\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$
- $\langle 2, 1, 6, 3, 7, 4, 8, 5, 9, 10 \rangle$

The freedom in this partial order is precisely what an "out-of-order" processor exploits.

Dependence Graph

Dependence graphs can also be derived from high-level program code.

```
1 x ← 0
2 i ← 1
3 while i < 100 do
4 |  if a[i] > 0 then
5 |  |  x ← x + a[i]
6 |  i ← i + 1
7 print(x)
```



Note References to `a[i]` are shown deriving their values from a node representing prior definitions of `a`. This connects all uses of a together through a single node.

Dependence Graph

Data-dependence graphs are often used as a **derivative IR**, constructed from the definitive IR for a specific task, used, and then discarded.

- instruction scheduling
- reordering loops to expose parallelism and to improve memory behavior

These typically require sophisticated **analysis of array subscripts** to determine more precisely the patterns of access to arrays.

Call Graph

To perform **interprocedural** analysis and optimization, compilers use a call graph.

A **call graph** represents the runtime transfers of control between procedures

- a node for each procedure
- an edge for each distinct procedure call site

Example If procedure `q` is called from three textually distinct sites in `p`, the call graph has three edges `(p, q)`, one for each call site.

Both software-engineering practice and language features complicate the construction of a call graph

- seperate compilation
- procedure-valued parameters
- object-oriented programming

Linear IRs

The alternative to a graphical IR is a **linear IR**. The linear IRs used in compilers resemble the assembly code for an abstract machine.

- sequence of instructions
- executed in their order of appearance (or in an order consistent with that order)
- instruction may contain more than one operation that then execute in parallel

Many kinds of linear IRs have been used in compilers

- **one-address codes**: accumulator machines and stack machines
- **two-address codes**: machine with destructive operations
- **three-address codes**: most operations take two operands and produce a result

Linear IRs

Control flow in a linear IR usually models the implementation of control flow on the target machine with operations for **conditional branches** and **jumps**.

Control flow **demarcates** the basic blocks in a linear IR: blocks end at branches, at jumps, or just before labelled operations.

Taken Branch

Typically, conditional branches use one label. Control flows either to the label, called the taken branch, or to the operation that follows the label, called the not-taken or fall-through branch.

Stack-Machine Codes

Stack-machine code, a form of one-address code, assumes the presence of a stack of operands.

The stack creates an **implicit name space** and eliminates many names from the IR

- shrinks the size of a program in IR form
- all results and arguments are transitory, unless the code explicitly moves them to memory

Stack-machine code is simple to generate and to execute.

Stack-machine code for expression $a - 2 \times b$

```
push 2
push b
multiply
push a
subtract
```

Stack-Machine Codes

Bytecode

Bytecode is an IR designed specifically for its compact form. Typically it represents code for an abstract stack machine.

The name derives from its limited size: opcodes are limited to one byte or less.

- → compact form of the program for distribution
- → reasonably simple scheme for porting the language to a new target machine

Three-Address Codes

In **three-address code**, most operations have the form
$i \leftarrow j \text{ op } k$

- one operator (op)
- two operands (j and k)
- one result (i)

Note Some operators, such as an immediate load and a jump, will need fewer arguments

Three-address code for expression $a - 2 \times b$

$$
\begin{aligned}
t_1 &\leftarrow 2 \\
t_2 &\leftarrow b \\
t_3 &\leftarrow t_1 \times t_2 \\
t_4 &\leftarrow a \\
t_5 &\leftarrow t_4 - t_3
\end{aligned}
$$

Advantages of three-address code

- reasonably compact: an operation (1 or 2 bytes) and three names (4 bytes)
- freedom to control the reuse of names and values
- many modern processors implement three-address operations

Representing Linear Codes

Since a compiler spends most of its time manipulating the IR form of the code, the costs associated with the **data structure** used to represent it deserve some attention.

A compiler writer needs to make the following **design decisions**

- how to represent an instruction
- how to represent a basic block

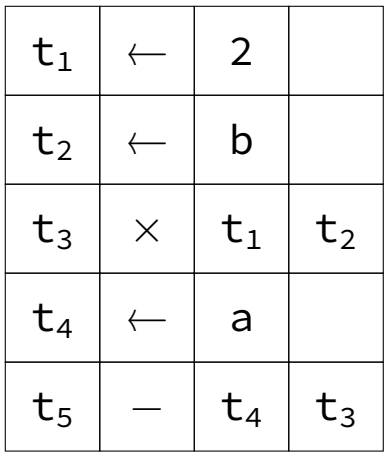
The resulting data structure needs to be **compact**, **efficient** to generate, and support operations to **replace** or **reorder** instructions.

Examples

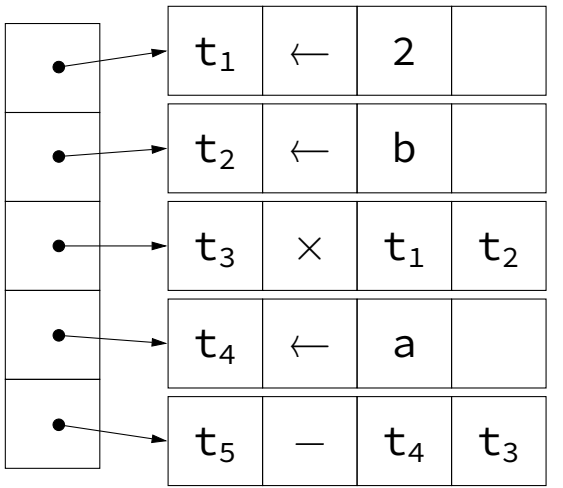
- simple array
- array of pointers
- linked list

Representing Linear Codes

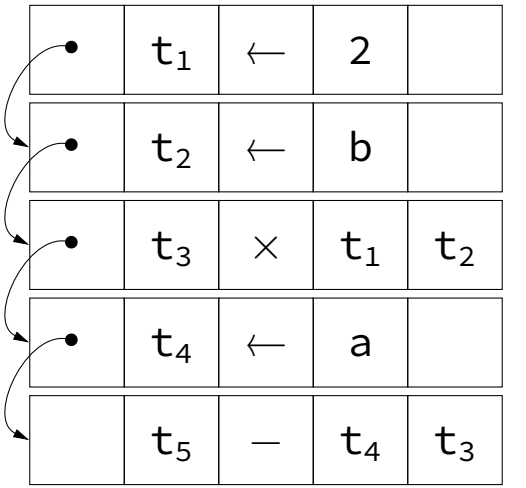
Simple Array



Array of Pointers



Linked List



Note Some information is not shown. For example, no labels are shown because labels are a property of the block rather than any individual quadruple.

Mapping Names to Values

A **naming scheme** can expose opportunities for optimization or it can obscure them.

- compiler must invent names for many, if not all, of the intermediate results
- these choices influence, which computations can be analyzed and optimized

Examples

- **source names**: generated names correspond to source names
- **value names**: generated names correspond to values, *i.e.*, textually identical expressions compute the same value
- **static single-assignment form** encodes both control and value flow

Naming Temporary Values

Source Code

$$
\begin{aligned}
a &\leftarrow b + c \\
b &\leftarrow a - d \\
c &\leftarrow b + c \\
d &\leftarrow a - d
\end{aligned}
$$

Source Names

$$
\begin{aligned}
t_1 &\leftarrow b \\
t_2 &\leftarrow c \\
t_3 &\leftarrow t_1 + t_2 \\
a &\leftarrow t_3 \\
t_4 &\leftarrow d \\
t_1 &\leftarrow t_3 - t_4 \\
b &\leftarrow t_1 \\
t_2 &\leftarrow t_1 + t_2 \\
c &\leftarrow t_2 \\
t_4 &\leftarrow t_3 - t_4 \\
d &\leftarrow t_4
\end{aligned}
$$

Value Names

$$
\begin{aligned}
t_1 &\leftarrow b \\
t_2 &\leftarrow c \\
t_3 &\leftarrow t_1 + t_2 \\
a &\leftarrow t_3 \\
t_4 &\leftarrow d \\
t_5 &\leftarrow t_3 - t_4 \\
b &\leftarrow t_5 \\
t_6 &\leftarrow t_5 + t_2 \\
c &\leftarrow t_6 \\
t_5 &\leftarrow t_3 - t_4 \\
d &\leftarrow t_5
\end{aligned}
$$

Static Single-Assignment Form

Static single-assignment form (SSA) is a naming scheme that many modern compilers use to encode information about both the control and data flow in the program.

A program is in SSA form when it meets two constraints

- each definition has a distinct name
- each use refers to a single definition

In SSA form, names correspond uniquely to specific **definition points** in the code. Each name is defined by one operation, hence the name static single assignment.

Static Single-Assignment Form

To reconcile this single-assignment naming scheme with the effects of **control flow**, a special operation, called ϕ-functions, is inserted at points where control-flow paths meet.

```
x ← ...
y ← ...
while (x < 100)
    x ← x + 1
    y ← y + x
```

```
      x0 ← ...
      y0 ← ...
      if (x0 ≥ 100) goto next
loop: x1 ← φ(x0, x2)
      y1 ← φ(y0, y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < 100) goto loop
next: x3 ← φ(x0, x2)
      y3 ← φ(y0, y2)
```

Static Single-Assignment Form

ϕ-function

A ϕ-function takes several names and merges them, defining a **new** name.

The ϕ-function's behavior **depends on context**: it defines the name with the value of its argument that corresponds to the edge along which control entered the block.

On entry to a basic block, all of its ϕ-functions execute **concurrently**, before any other statement

- first, they **all** read the values of the appropriate arguments
- then they **all** define their target SSA names

Note ϕ-functions do not comform to the three-address model!

Memory Models

For each value computed by the program, the compiler must determine, where in the memory hierarchy that value will reside.

Compilers typically work from one of two memory models

- **Register-to-Register Model**: the compiler keeps values in registers aggressively, ignoring any limitations imposed by the size of the machine's physical register set.
- **Memory-to-Memory Model**: the compiler assumes that all values are kept in memory locations

While the choice of memory model is mostly **orthogonal** to the choice of IR, it has an impact on the rest of the compiler and, in particular, on the register allocator.