Compilation

Lecture 7

Getting into the back-end

Noam Rinetzky

Compilation

Lecture 7

Intermediate Representation

Noam Rinetzky

But first, a short reminder

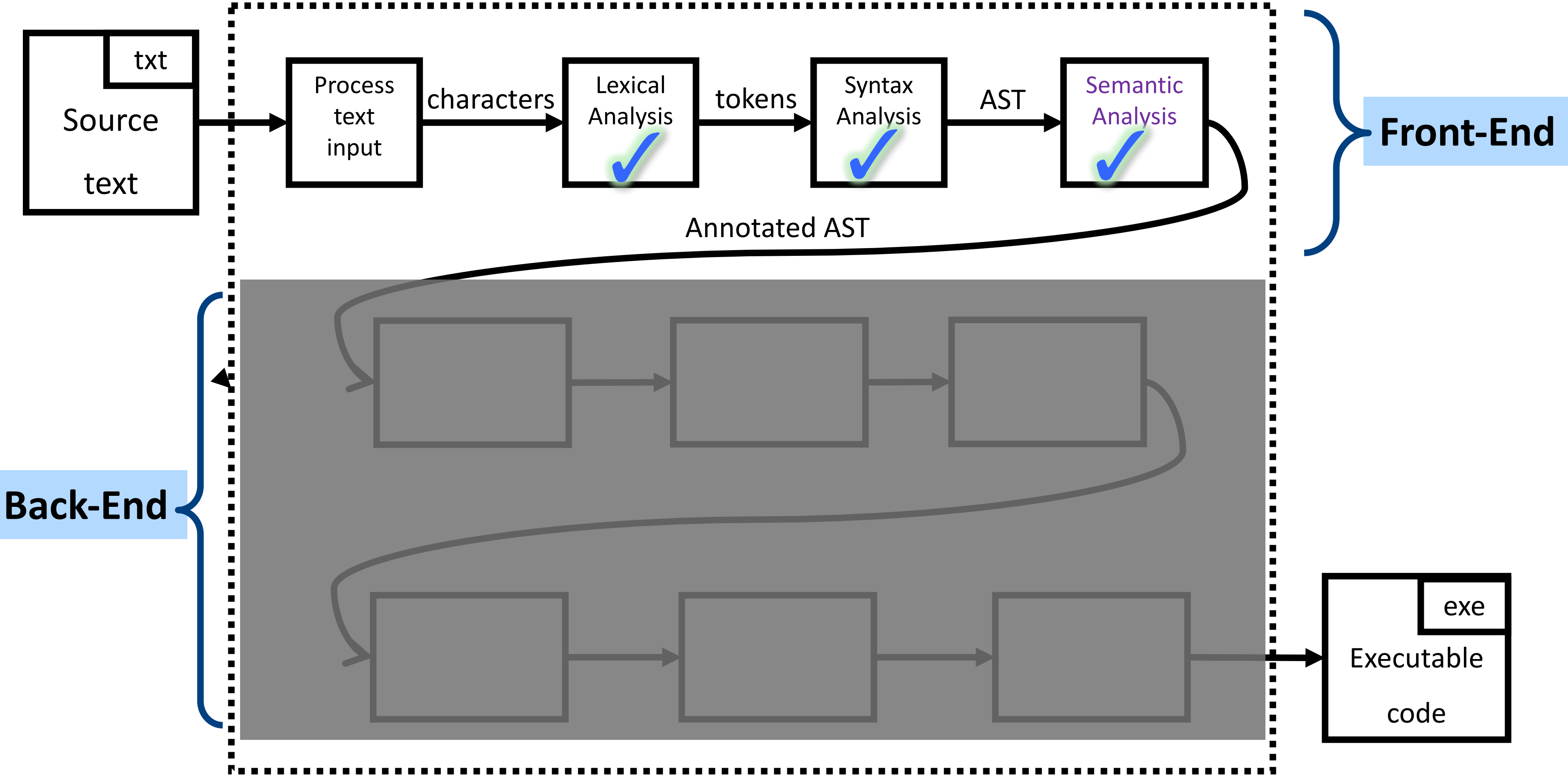
What is a compiler?

“A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

The most common reason for wanting to transform source code is to create an executable program.”

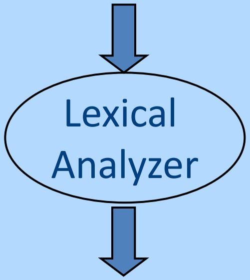
--Wikipedia

Where we were



Lexical Analysis

program text **`((23 + 7) * x)`**



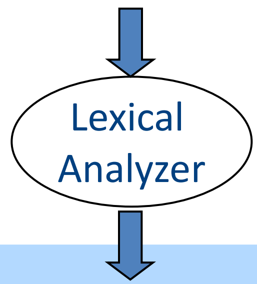
token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

From scanning to parsing

program text

((23 + 7) * x)



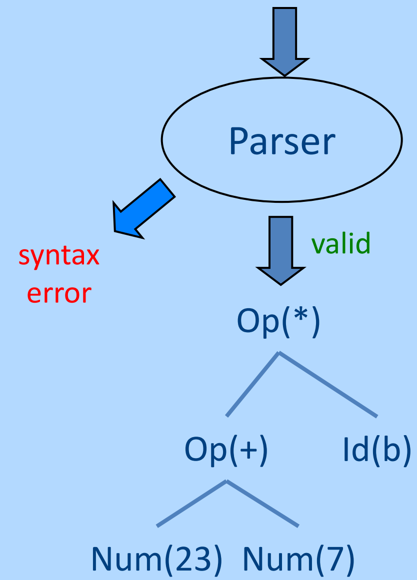
token stream

(	(	23	+	7	)	*	x	)
LP	**LP**	**Num**	**OP**	**Num**	**RP**	**OP**	**Id**	**RP**

Grammar:

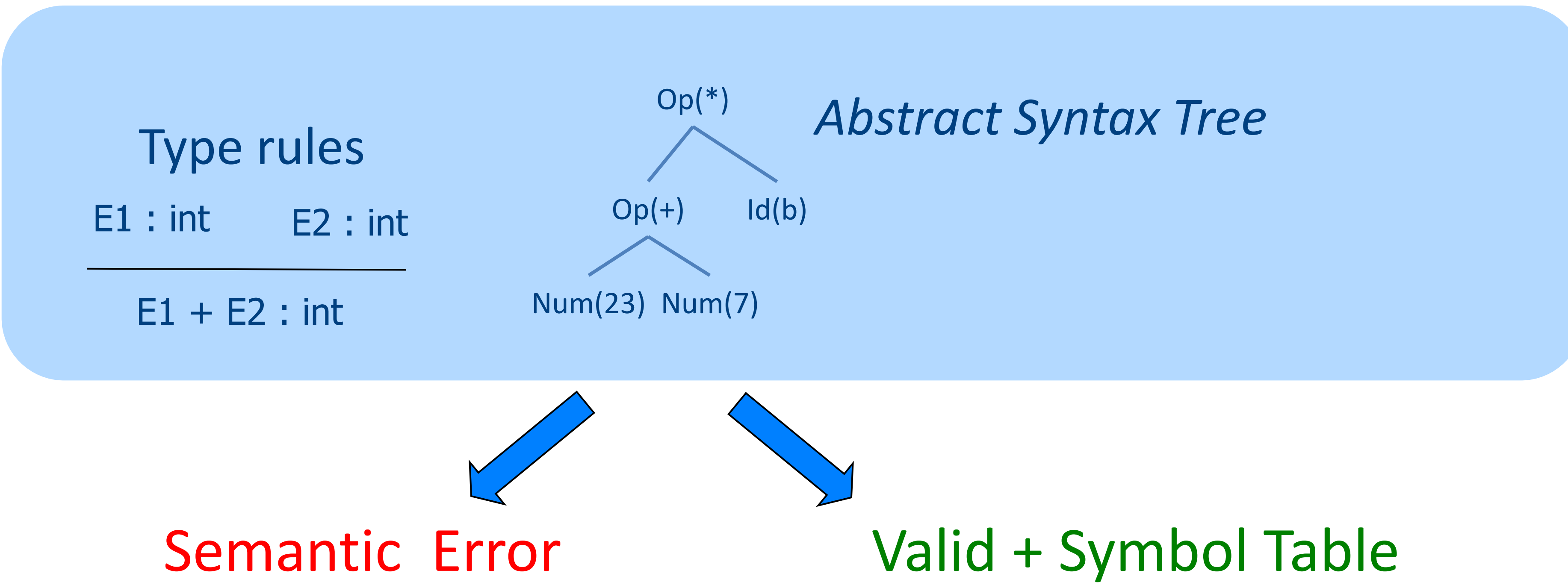
E → ... | Id

Id → ‘a’ | ... | ‘z’



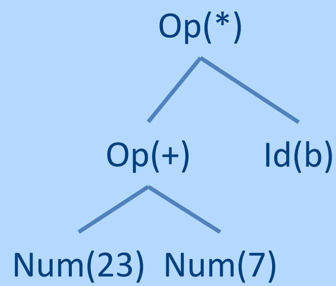
Abstract Syntax Tree

Context Analysis



Code Generation

...



Valid Abstract Syntax Tree
Symbol Table

Verification (possible runtime)
Errors/Warnings

input → Executable Code → output

What is a compiler?

"A **compiler** is a computer program that **transforms** source **code** written in a programming language (source language) into another language (target language).

The most common reason for wanting to transform source code is to create an **executable program**."

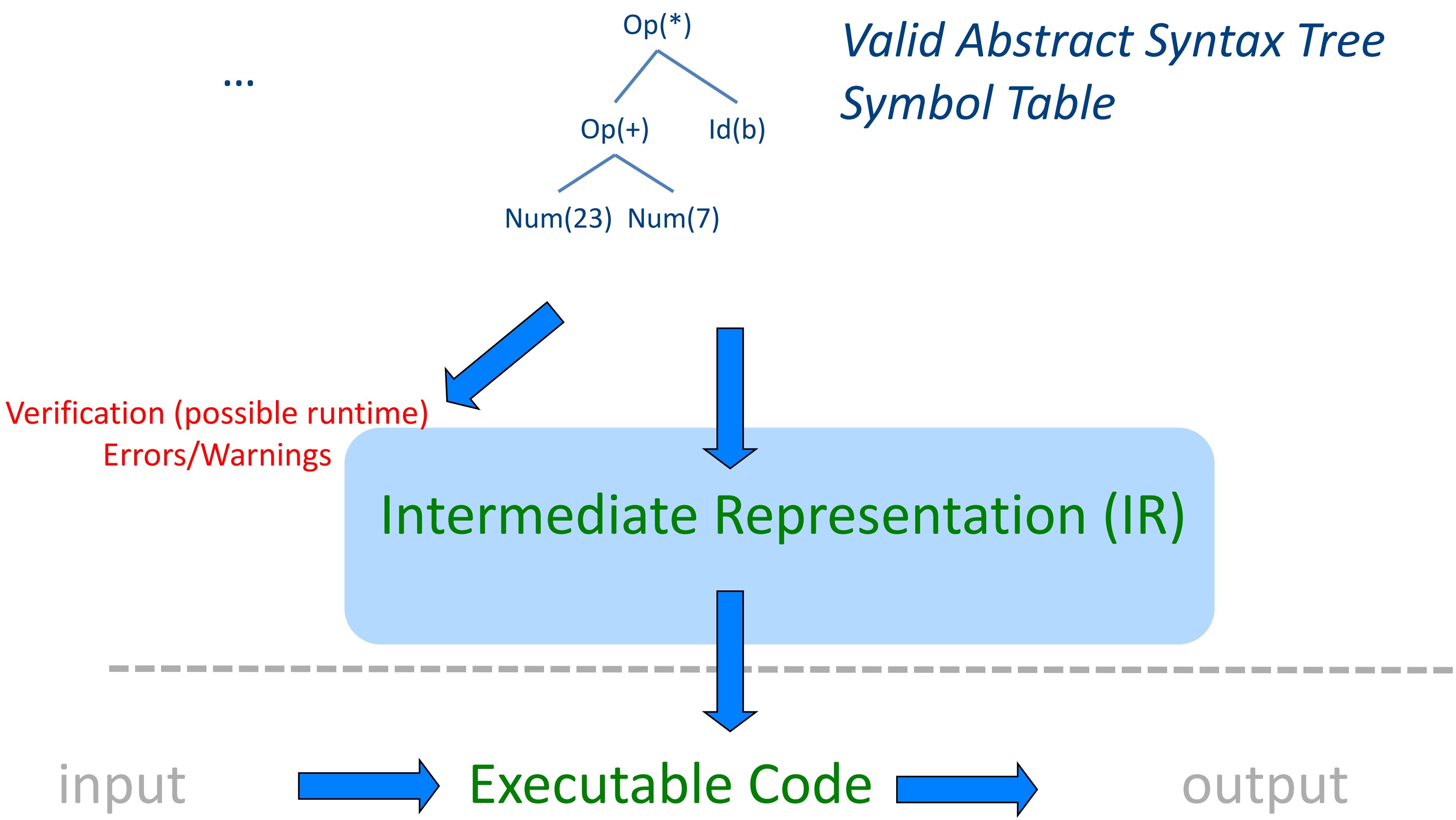
A CPU is (a sort of) an ***Interpreter***

"A **compiler** is a computer program that **transforms** source **code** written in a programming language (source language) into another language (target language).

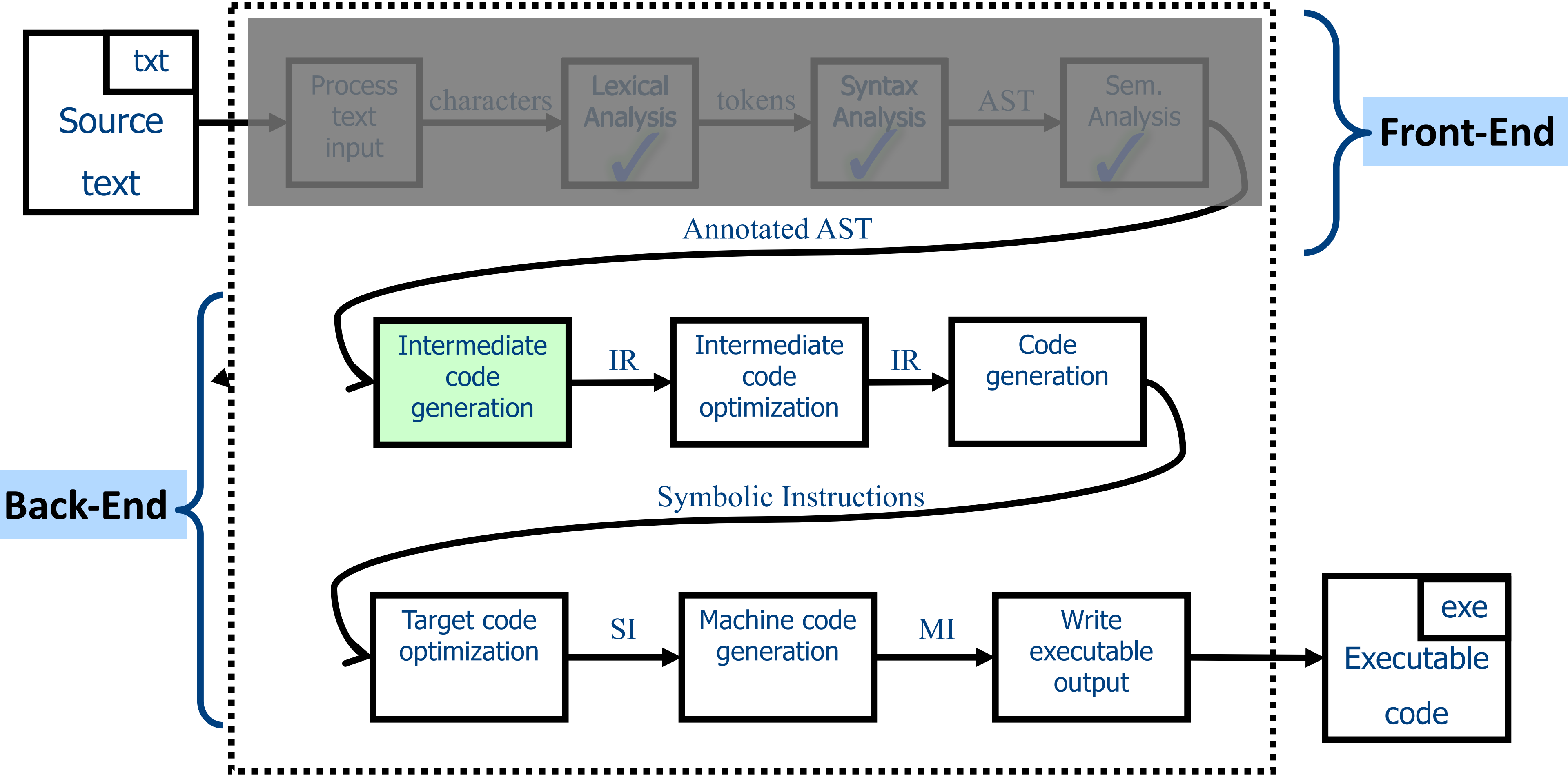
The most common reason for wanting to transform source code is to create an **executable program**."

- Interprets machine code …
  - Why not AST?
- Do we want to go from AST directly to MC?
  - We can, but …
    - Machine specific
    - Very low level

Code Generation in Stages



Where we are

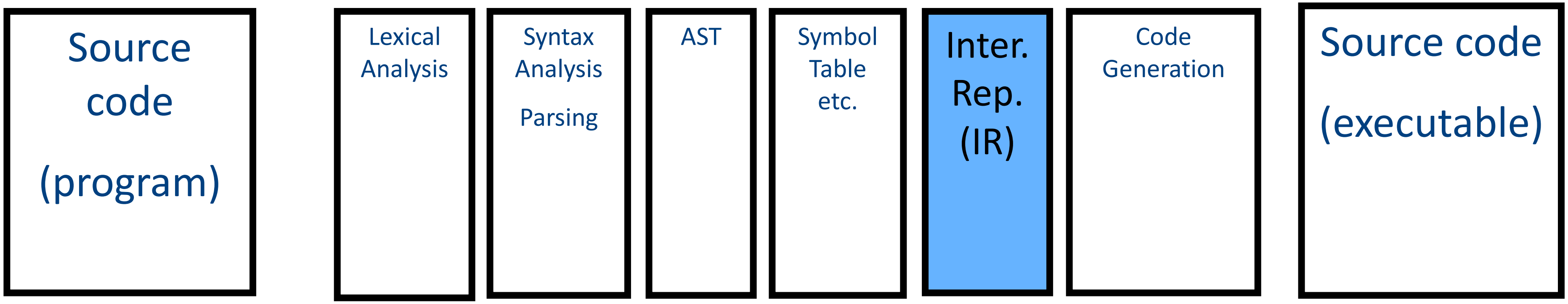

txt
Source text
Process text input
characters
Lexical Analysis
tokens
Syntax Analysis
AST
Sem. Analysis
Front-End
Annotated AST
Intermediate code generation
IR
Intermediate code optimization
IR
Code generation
Back-End
Symbolic Instructions
Target code optimization
SI
Machine code generation
MI
Write executable output
exe
Executable code

1 Note: Compile Time vs Runtime

- Compile time: Data structures used during program compilation
- Runtime: Data structures used during program execution
  - Activation record stack
  - Memory management
- The compiler generates code that allows the program to interact with the runtime

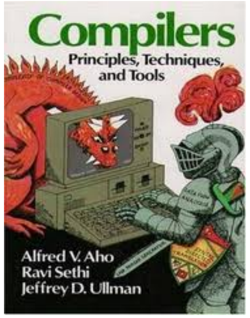
Intermediate Representation

Code Generation: IR



- Translating from abstract syntax (AST) to intermediate representation (IR)
  - **T**hree-**A**ddress **C**ode
- …

Three-Address Code IR



Chapter 8

- A popular form of IR
- High-level assembly where instructions have at most three operands

IR by example

Sub-expressions example

Source	**IR**
int a;	
int b;	
int c;	
int d;	
a = b + c + d;	_t0 = b + c; a = _t0 + d;

b = a * a + b * b;	_t1 = a * a;
	_t2 = b * b;
	b = _t1 + _t2;

Sub-expressions example

Source

```
int a;
int b;
int c;
int d;
a = b + c + d;
b = a * a + b * b;
```

IR (not optimized)

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

Temporaries explicitly store intermediate values resulting from sub-expressions

Variable assignments

- var = constant ;
- var_1 = var_2 ;
- var_1 = var_2 **op** var_3 ;
- var_1 = constant **op** var_2 ;
- var_1 = var_2 **op** constant ;
- var = $constant_1$ **op** $constant_2$;
- Permitted operators are **+, -, *, /, %**

In the impl. var is replaced by a pointer to the symbol table

A compiler-generated temporary can be used instead of a var

Booleans

- Boolean variables are represented as integers that have zero or nonzero values
- In addition to the arithmetic operator, TAC supports <, ==, ||, and &&
- How might you compile the following?

```
b = (x <= y);
```

```
_t0 = x < y;
_t1 = x == y;
b = _t0 || _t1;
```

Unary operators

- How might you compile the following assignments from unary statements?

`y = -x;`	`y = 0 - x;` `y = -1 * x;`
`z := !w;`	`z = w == 0;`

Control flow instructions

- Label introduction

  ```
  _label_name:
  ```

  Indicates a point in the code that can be jumped to
- Unconditional jump: go to instruction following label L

  ```
  Goto L;
  ```
- Conditional jump: test condition variable t;
  if 0, jump to label L

  ```
  IfZ t Goto L;
  ```
- Similarly : test condition variable t;
  if not zero, jump to label L

  ```
  IfNZ t Goto L;
  ```

Control-flow example – conditions

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;
z = z * z;
```

```
        _t0 = x < y;
        IfZ _t0 Goto _L0;
        z = x;
        Goto _L1;
_L0:
        z = y;
_L1:
        z = z * z;
```

Control-flow example – loops

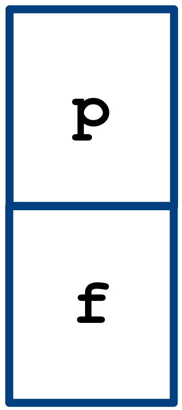
```
int x;
int y;

while (x < y) {
   x = x * 2;
}

y = x;
```

```
_L0:
      _t0 = x < y;
      IfZ _t0 Goto _L1;
      x = x * 2;
      Goto _L0;
_L1:
      y = x;
```

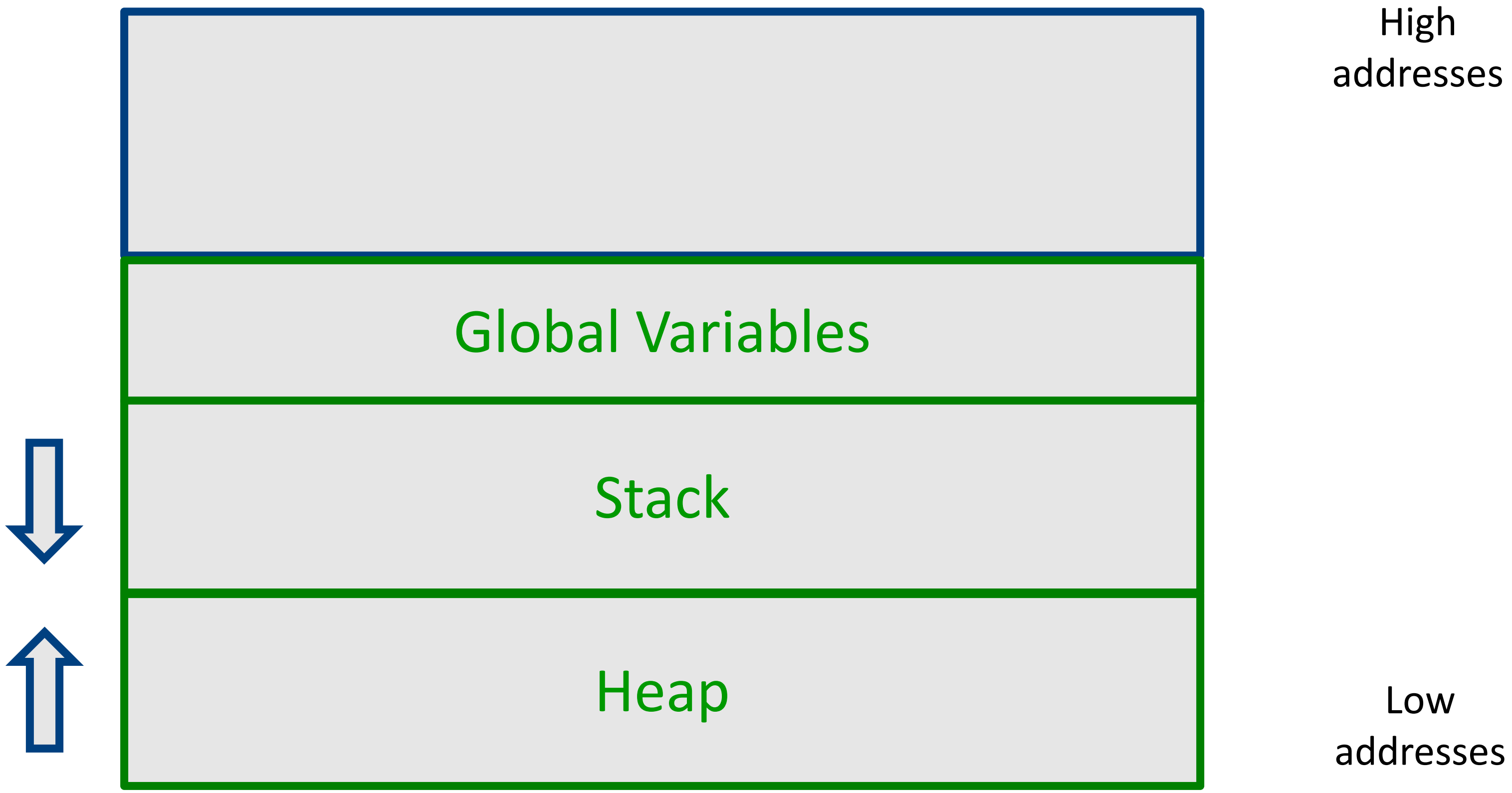
Procedures / Functions

```
p(){
  int y=1, x=0;
  x=f(a1,…,an);
  print(x);
}
```

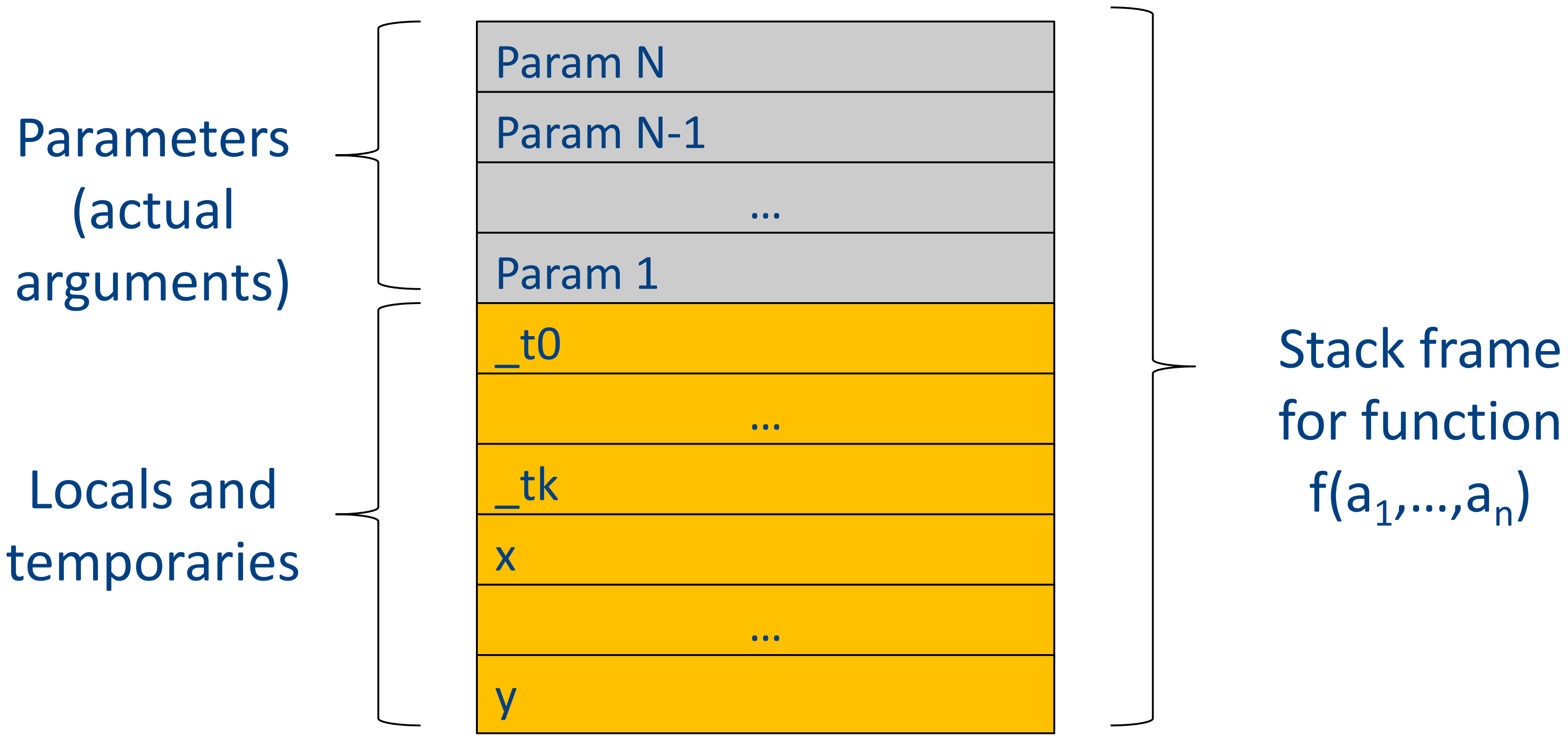
- What happens in runtime?



Memory Layout
(popular convention)



A logical stack frame



Procedures / Functions

- A procedure call instruction **pushes** arguments to stack and **jumps** to the function label
  A statement **`x=f(a1,…,an);`** looks like
  ```
  Push a1; … Push an;
  Call f;
  Pop x; // pop returned value, and copy to it
  ```
- Returning a value is done by **pushing** it to the stack (**`return x;`**)
  ```
  Push x;
  ```
- **Return control** to caller (and **roll up stack**)
  ```
  Return;
  ```

Functions example

```
int SimpleFn(int z) {
    int x, y;
    x = x * y * z;
    return x;
}

void main() {
   int w;
   w = SimpleFunction(137);
}
```

```
_SimpleFn:
_t0 = x * y;
_t1 = _t0 * z;
x = _t1;
Push x;
Return;

main:
_t0 = 137;
Push _t0;
Call _SimpleFn;
Pop w;
```

Memory access instructions

- **Copy** instruction: a = b
- **Load/store** instructions:
  a = *b   *a = b
- **Address of** instruction a=&b
- **Array accesses**:
  a = b[i]   a[i] = b
- **Field accesses**:
  a = b[f]   a[f] = b
- **Memory allocation** instruction:
  a = alloc(size)
  - Sometimes left out (e.g., malloc is a procedure in C)

Memory access instructions

- **Copy** instruction: a = b
- **Load/store** instructions:
  a = *b   *a = b
- **Address of** instruction a=&b
- **Array accesses**:
  a = b[i]   a[i] = b
- **Field accesses**:
  a = b[f]   a[f] = b
- **Memory allocation** instruction:
  a = alloc(size)
  - Sometimes left out (e.g., malloc is a procedure in C)

Array operations

x := y[i]

```
t1 := &y      ; t1 = address-of y
t2 := t1 + i  ; t2 = address of y[i]
x  := *t2     ; loads the value located at y[i]
```

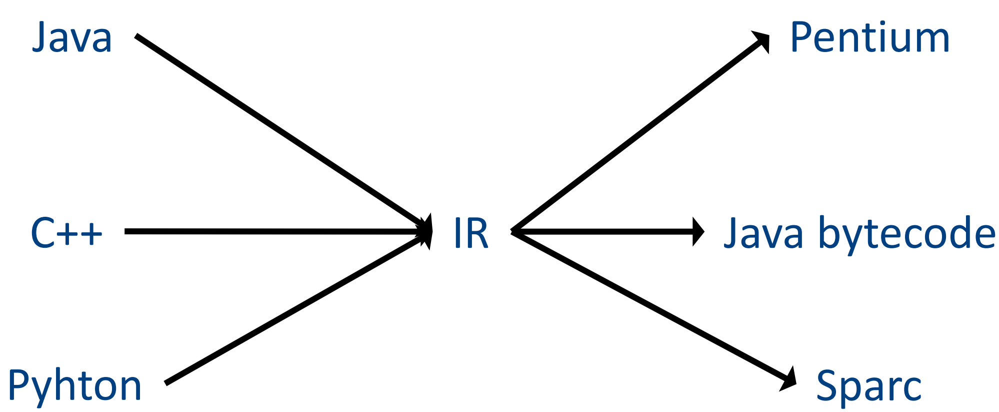
x[i] := y

```
t1  := &x      ; t1 = address-of x
t2  := t1 + i  ; t2 = address of x[i]
*t2 := y       ; store through pointer
```

IR Summary

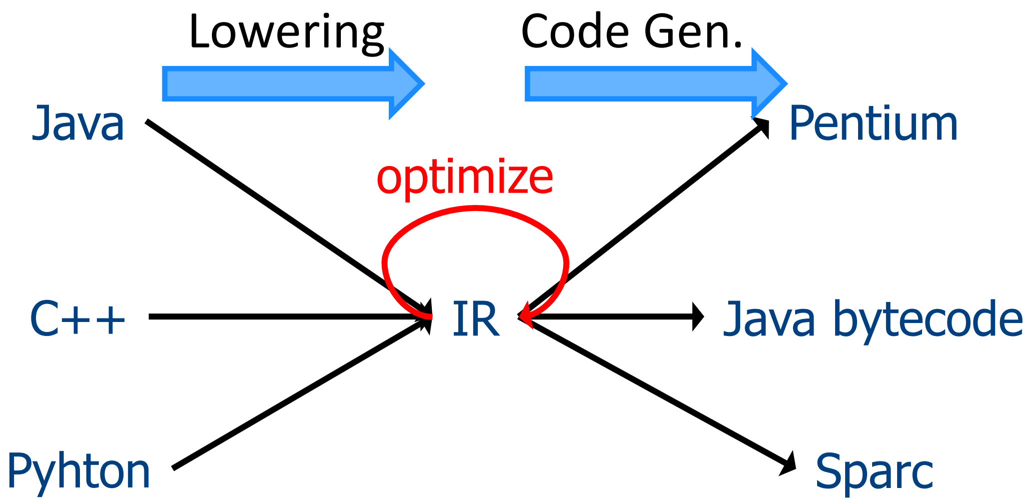
Intermediate representation

- A language that is between the source language and the target language – not specific to any machine
- Goal 1: retargeting compiler components for different source languages/target machines



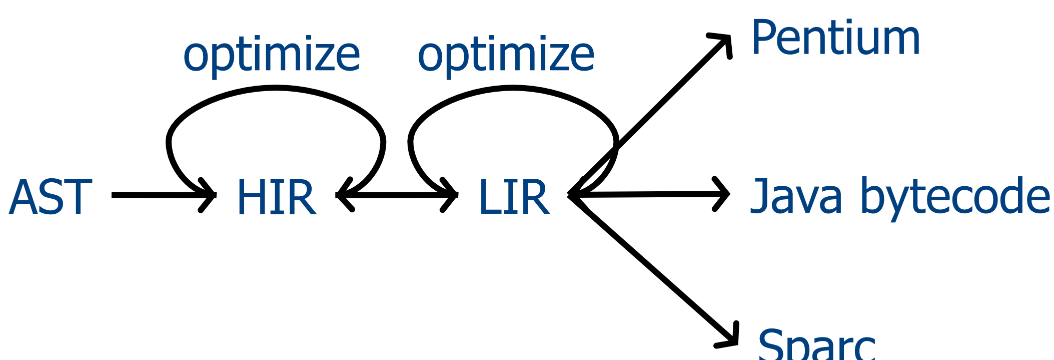
Intermediate representation

- A language that is between the source language and the target language – not specific to any machine
- Goal 1: retargeting compiler components for different source languages/target machines
- Goal 2: machine-independent optimizer
  - Narrow interface: small number of instruction types



Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code
- Solution: use multiple IR stages

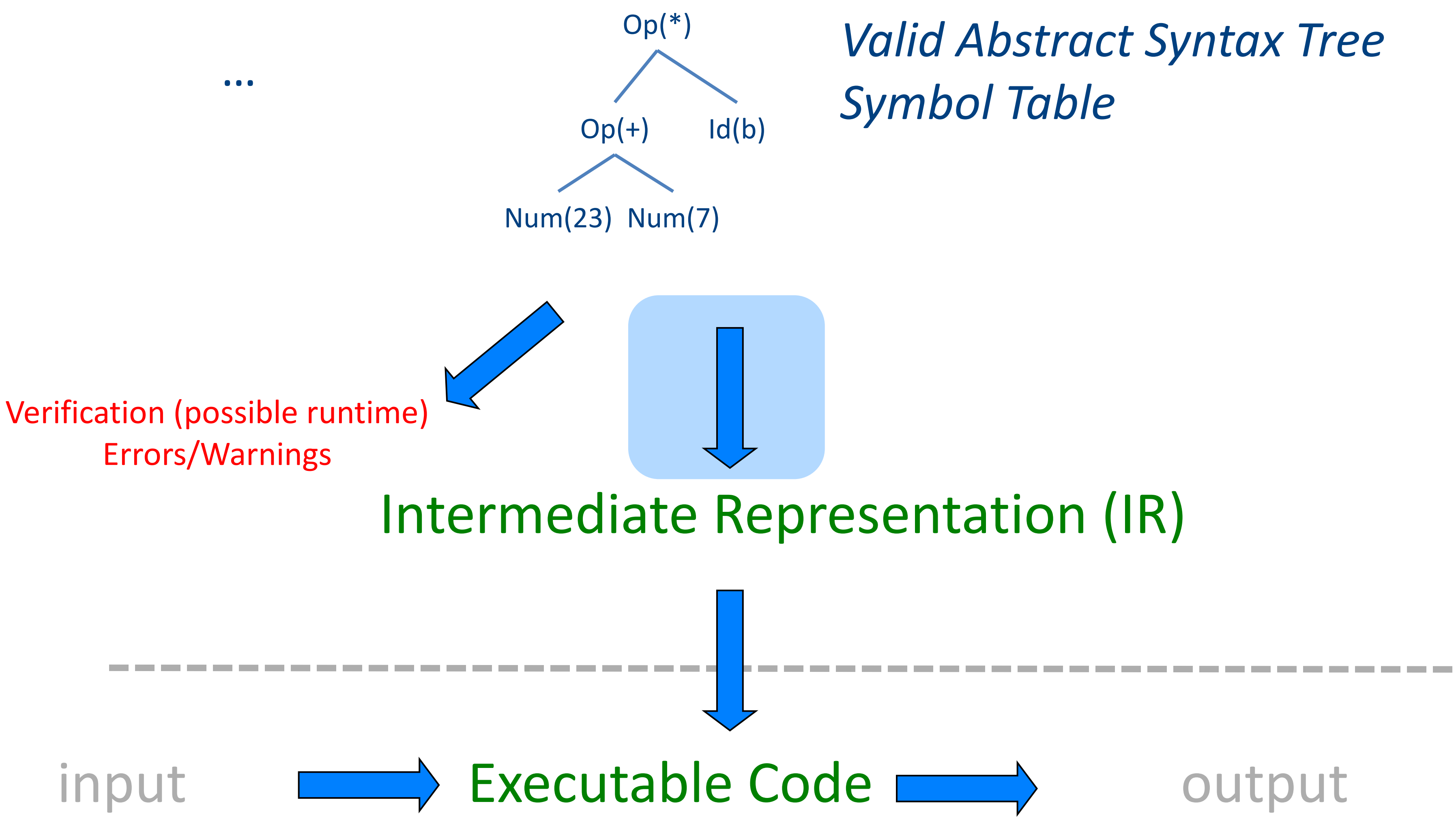


AST vs. LIR for imperative languages

AST	LIR
Rich set of language constructs	An abstract machine language
Rich type system	Very limited type system
Declarations: types (classes, interfaces), functions, variables	Only computation-related code
Control flow statements: if-then-else, while-do, break-continue, switch, exceptions	Labels and conditional/ unconditional jumps, no looping
Data statements: assignments, array access, field access	Data movements, generic memory access statements
Expressions: variables, constants, arithmetic operators, logical operators, function calls	No sub-expressions, logical as numeric, temporaries, constants, function calls – explicit argument passing

Lowering AST to TAC

IR Generation



TAC generation

- At this stage in compilation, we have
  - an AST
  - annotated with scope information
  - and annotated with type information
- To generate TAC for the program, we do recursive tree traversal
  - Generate TAC for any subexpressions or substatements
  - Using the result, generate TAC for the overall expression

TAC generation for expressions

- Define a function **cgen**(*expr) that generates* TAC that computes an expression, stores it in a temporary variable, then hands back the name of that temporary
  - Define **cgen** directly for atomic expressions (constants, this, identifiers, etc.)
- Define **cgen** recursively for compound expressions (binary operators, function calls, etc.)

cgen for basic expressions

cgen(*k) = { // k is a constant*
  Choose a new temporary *t*
  Emit(*t = k)*
  Return *t*
}

cgen(*id) = { // id is an identifier*
  Choose a new temporary *t*
  Emit(*t = id)*
  Return *t*
}

cgen for binary operators

cgen(e_1 + e_2) = {

Choose a new temporary *t*

Let t_1 = **cgen**(e_1)

Let t_2 = **cgen**(e_2)

Emit($t = t_1 + t_2$)

Return *t*

}

cgen example

cgen(5 + x) = {

Choose a new temporary t

Let t_1 = **cgen**(5)

Let t_2 = **cgen**(x)

Emit($t = t_1 + t_2$)

Return t

}

cgen example

cgen(5 + x) = {

  Choose a new temporary *t*

  Let t_1 = {

    Choose a new temporary *t*

    Emit(*t = 5;*)

    Return *t*

  }

  Let t_2 = **cgen**(x)

  Emit($t = t_1 + t_2$)

  Return *t*

}

cgen example

cgen(5 + x) = {
  Choose a new temporary *t*
  Let t_1 = {
    Choose a new temporary *t*
    Emit(*t = 5;*)
    Return *t*
  }
  Let t_2 = {
    Choose a new temporary *t*
    Emit(*t = x;*)
    Return *t*
  }
  Emit($t = t_1 + t_2;$)
  Return *t*
}

Returns an **arbitrary fresh** name

```
t1 = 5;
t2 = x;
t = t1 + t2;
```

cgen example

cgen(5 + x) = {
  Choose a new temporary *t*
  Let t_1 = {
    Choose a new temporary *t*
    Emit(*t = 5;*)
    Return *t*
  }
  Let t_2 = {
    Choose a new temporary *t*
    Emit(*t = x;*)
    Return *t*
  }
  Emit($t = t_1 + t_2;$)
  Return *t*
}

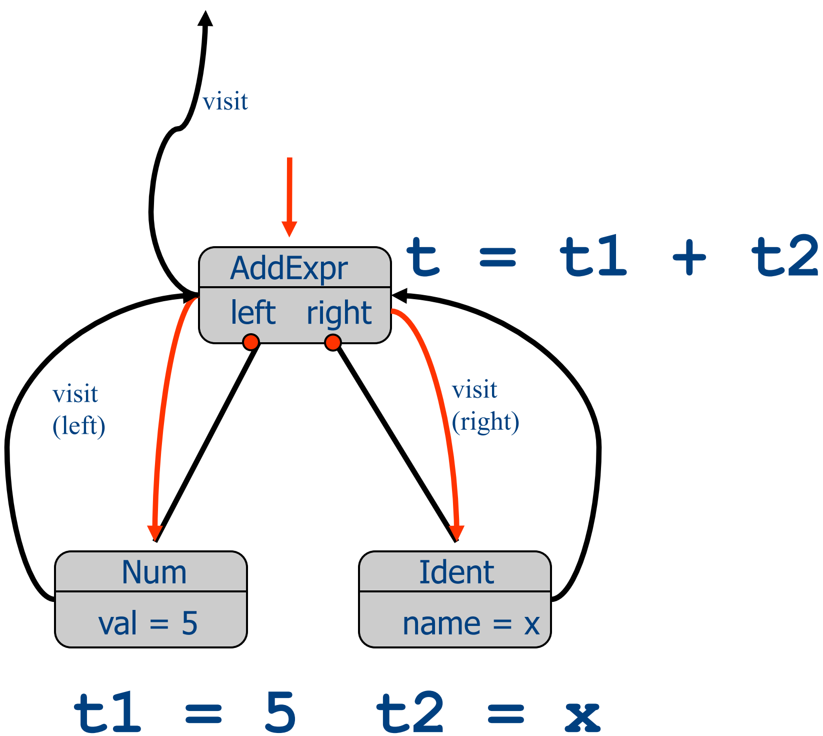
Returns an **arbitrary fresh** name

```
_t18 = 5;
_t29 = x;
_t6 = _t18 + _t29;
```

Inefficient translation, but we will improve this later

cgen as recursive AST traversal

cgen(5 + x)




```
t1 = 5;

t2 = x;

t = t1 + t2;
```

Naive **cgen** for expressions

- Maintain a counter for temporaries in c
- Initially: c = 0
- **cgen**(e_1 *op* e_2) = {

  Let A = **cgen**(e_1)
  c = c + 1
  Let B = **cgen**(e_2)
  c = c + 1
  Emit(_tc = A *op* B;)
  Return _tc

  }

Example

cgen((a*b)-d)

Example

c = 0

cgen((a*b)-d)

Example

```
c = 0
cgen( (a*b)-d) = {
   Let A = cgen(a*b)
   c = c + 1
   Let B = cgen(d)
   c = c + 1
   Emit( _tc = A - B; )
   Return _tc
}
```

Example

```
c = 0
cgen( (a*b)-d) = {
  Let A = {
      Let A = cgen(a)
      c = c + 1
      Let B = cgen(b)
      c = c + 1
      Emit( _tc = A * B; )
      Return tc
   }
   c = c + 1
   Let B = cgen(d)
   c = c + 1
   Emit( _tc = A - B; )
   Return _tc
}
```

Example

```
c = 0
cgen( (a*b)-d) = {
  Let A = {                                   here A=_t0
      Let A = { Emit(_tc = a;), return _tc }
      c = c + 1
      Let B = { Emit(_tc = b;), return _tc }
      c = c + 1
      Emit( _tc = A * B; )
      Return _tc
  }
  c = c + 1
  Let B = { Emit(_tc = d;), return _tc }
  c = c + 1
  Emit( _tc = A - B; )
  Return _tc
}
```

Code

Example

```
c = 0
cgen( (a*b)-d) = {
    Let A = {
        Let A = { Emit(_tc = a;), return _tc }
        c = c + 1
        Let B = { Emit(_tc = b;), return _tc }
        c = c + 1
        Emit( _tc = A * B; )
        Return _tc
    }
    c = c + 1
    Let B = { Emit(_tc = d;), return _tc }
    c = c + 1
    Emit( _tc = A - B; )
    Return _tc
}
```

here A=_t0

Code

```
_t0=a;
```

Example

```
c = 0
cgen( (a*b)-d) = {
   Let A = {
      Let A = { Emit(_tc = a;), return _tc }
      c = c + 1
      Let B = { Emit(_tc = b;), return _tc }
      c = c + 1
      Emit( _tc = A * B; )
      Return _tc
   }
   c = c + 1
   Let B = { Emit(_tc = d;), return _tc }
   c = c + 1
   Emit( _tc = A - B; )
   Return _tc
}
```

here A=_t0

Code

```
_t0=a;
_t1=b;
```

Example

```
c = 0
cgen( (a*b)-d) = {
   Let A = {
      Let A = { Emit(_tc = a;), return _tc }
      c = c + 1
      Let B = { Emit(_tc = b;), return _tc }
      c = c + 1
      Emit( _tc = A * B; )
      Return _tc
   }
   c = c + 1
   Let B = { Emit(_tc = d;), return _tc }
   c = c + 1
   Emit( _tc = A - B; )
   Return _tc
}
```

here A=_t0

Code

```
_t0=a;
_t1=b;
_t2=_t0*_t1
```

Example

```
c = 0
cgen( (a*b)-d) = {
  Let A = {
     Let A = { Emit(_tc = a;), return _tc }
     c = c + 1
     Let B = { Emit(_tc = b;), return _tc }
     c = c + 1
     Emit( _tc = A * B; )
     Return _tc
  }
  c = c + 1
  Let B = { Emit(_tc = d;), return _tc }
  c = c + 1
  Emit( _tc = A - B; )
  Return _tc
}
```

here A=_t2

here A=_t0

```
Code
_t0=a;
_t1=b;
_t2=_t0*_t1
```

Example

```
c = 0
cgen( (a*b)-d) = {
 Let A = {
    Let A = { Emit(_tc = a;), return _tc }
       c = c + 1
       Let B = { Emit(_tc = b;), return _tc }
       c = c + 1
       Emit( _tc = A * B; )
       Return _tc
    }
    c = c + 1
    Let B = { Emit(_tc = d;), return _tc }
    c = c + 1
    Emit( _tc = A - B; )
    Return _tc
}
```

here A=_t2

here A=_t0

Code

```
_t0=a;
_t1=b;
_t2=_t0*_t1
_t3=d;
```

Example

```
c = 0
cgen( (a*b)-d) = {
 Let A = {
    Let A = { Emit(_tc = a;), return _tc }
      c = c + 1
      Let B = { Emit(_tc = b;), return _tc }
      c = c + 1
      Emit( _tc = A * B; )
      Return _tc
   }
   c = c + 1
   Let B = { Emit(_tc = d;), return _tc }
   c = c + 1
   Emit( _tc = A - B; )
   Return _tc
}
```

here A=_t2

here A=_t0

Code

```
_t0=a;
_t1=b;
_t2=_t0*_t1
_t3=d;
_t4=_t2-_t3
```

cgen for statements

- We can extend the **cgen** function to operate over statements as well
- Unlike **cgen** for expressions, **cgen** for statements does not return the name of a temporary holding a value.
  - *(Why?)*

cgen for simple statements

```
cgen(expr;) = {
   cgen(expr)
}
```

cgen for `if-then-else`

cgen(if (e) s_1 else s_2)

Let _t = **cgen**(e)

Let L_{true} be a new label

Let L_{false} be a new label

Let L_{after} be a new label

Emit(IfZ _t Goto L_{false};)

cgen(s_1)

Emit(Goto L_{after};)

Emit(L_{false}:)

cgen(s_2)

Emit(Goto L_{after};)

Emit(L_{after}:)

cgen for **`while`** loops

cgen(while (*expr) stmt)*

Let L_{before} be a new label.
Let L_{after} be a new label.
Emit(L_{before}:)
Let t = **cgen**(expr)
Emit(IfZ t Goto Lafter;)
cgen(stmt)
Emit(Goto L_{before};)
Emit(L_{after}:)

cgen for short-circuit disjunction

cgen(e1 || e2)

Emit(_t1 = 0; _t2 = 0;)

Let L_{after} be a new label

Let _t1 = **cgen**(e1)

Emit(IfNZ _t1 Goto L_{after})

Let _t2 = **cgen**(e2)

Emit(L_{after}:)

Emit(_t = _t1 || _t2;)

Return _t

Our first optimization

Naive **cgen** for expressions

- Maintain a counter for temporaries in c
- Initially: c = 0
- **cgen**(e_1 *op* e_2) = {

    Let A = **cgen**(e_1)
    c = c + 1
    Let B = **cgen**(e_2)
    c = c + 1
    Emit(_tc = A *op* B;)
    Return _tc

  }

Naïve translation

- **cgen** translation shown so far very inefficient
  - Generates (too) many temporaries – one per sub-expression
  - Generates many instructions – at least one per sub-expression
- Expensive in terms of running time and space
- Code bloat
- We can do much better …

Naive **cgen** for expressions

- Maintain a counter for temporaries in c
- Initially: c = 0
- **cgen**(e_1 *op* e_2) = {
  Let A = **cgen**(e_1)
  c = c + 1
  Let B = **cgen**(e_2)
  c = c + 1
  Emit(_tc = A *op* B;)
  Return _tc
  }
- **Observation: temporaries in cgen(e_1) can be reused in cgen(e_2)**

Improving **cgen** for expressions

- Observation – naïve translation needlessly generates temporaries for leaf expressions
- **Observation – temporaries used exactly once**
  - **Once a temporary has been read it can be reused for another sub-expression**
- **cgen**(e_1 *op* e_2) = {
  Let _t1 = **cgen**(e_1)
  Let _t2 = **cgen**(e_2)
  Emit(_t =_t1 *op* _t2;)
  Return t
  }
- Temporaries **cgen**(e_1) can be reused in **cgen**(e_2)

Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
  - Minimizes number of temporaries
- Main data structure in algorithm is a stack of temporaries
  - Stack corresponds to recursive invocations of _t = **cgen**(e)
  - All the temporaries on the stack are live
    - Live = contain a value that is needed later on

Live temporaries stack

- Implementation: use counter c to implement live temporaries stack
  - Temporaries _t(0), … , _t(c) are alive
  - Temporaries _t(c+1), _t(c+2)… can be (re)used
  - Push means increment c, pop means decrement c
- In the translation of _t(c)=**cgen**(e_1 *op* e_2)

```
_t(c) = cgen(e1)
            --------------- c = c + 1
_t(c) = cgen(e2)
            --------------- c = c - 1
_t(c) = _t(c) op  _t(c+1)
```

Using stack of temporaries example

_t0 = cgen(((c*d)-(e*f))+(a*b))

```
            ------- c = 0

                          _t0 = c*d
                                         ------- c = c + 1
_t0 = cgen(c*d)-(e*f))    _t1 = e*f
                                         ------- c = c - 1
                          _t0 = _t0 -_t1

            ------- c = c + 1
 _t1 = a*b
            ------- c = c - 1
 _t0 = _t0 + _t1
```

Weighted register allocation

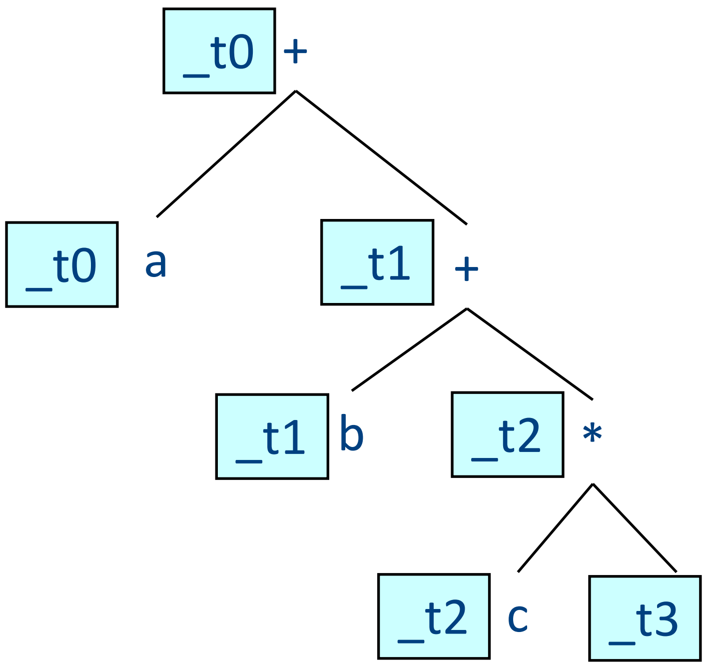
- Suppose we have expression e_1 *op* e_2
  - e_1, e_2 without side-effects
    - That is, no function calls, memory accesses, ++x
  - **cgen**(e_1 *op* e_2) = **cgen**(e_2 *op* e_1)
  - *Does order of translation matter?*
- Sethi & Ullman's algorithm translates heavier sub-tree first
  - Optimal local (per-statement) allocation for side-effect-free statements

Example

$\_t0 = \textbf{cgen}(\ a+(b+(c*d))\ )$

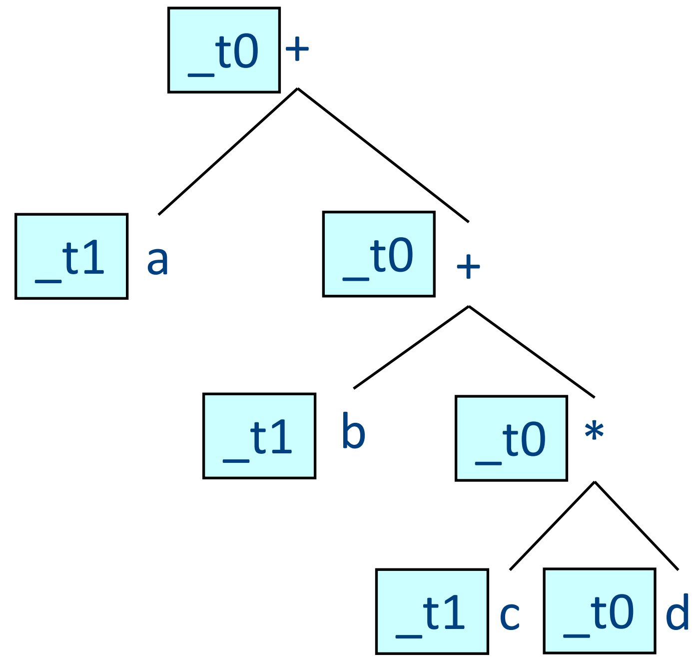
*+ and * are commutative operators*

left child first



4 temporaries

right child first



2 temporary

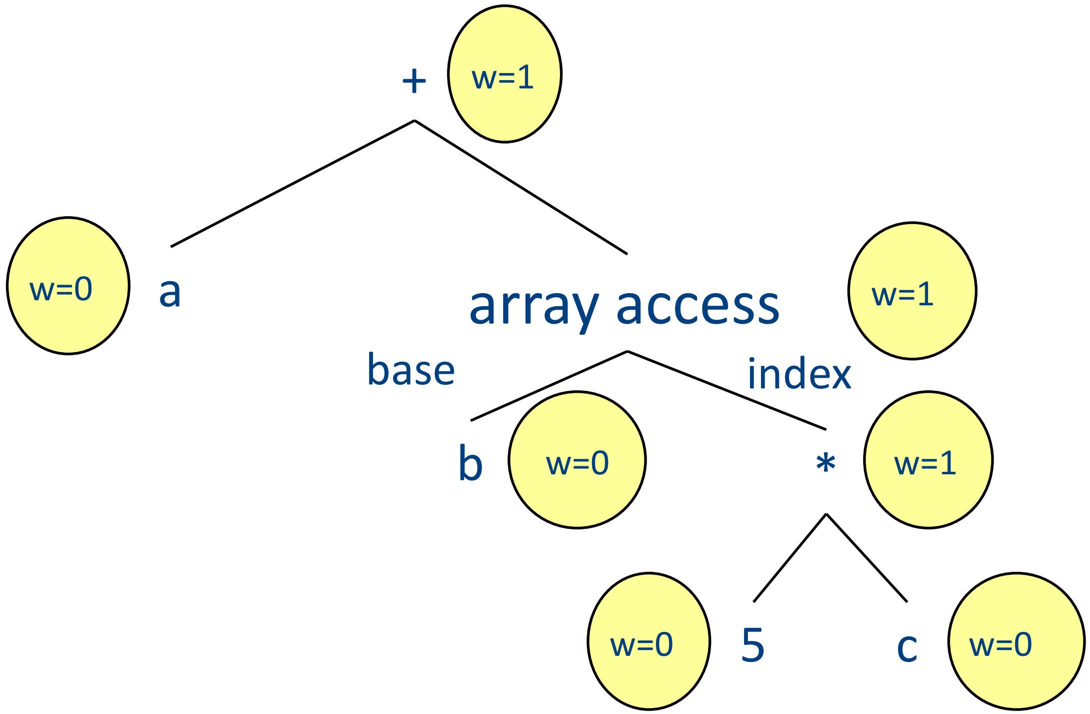
Weighted register allocation

- Can save registers by **re-ordering** subtree **computations**
- Label each node with its **weight**
  - Weight = number of registers needed
  - Leaf weight known
  - Internal node weight
    - w(left) > w(right) then w = left
    - w(right) > w(left) then w = right
    - w(right) = w(left) then w = left + 1
- Choose **heavier** child as first to be translated
- WARNING: have to check that no side-effects exist before attempting to apply this optimization
  - pre-pass on the tree

Weighted reg. alloc. example

_t0 = **cgen**(a+b[5*c])

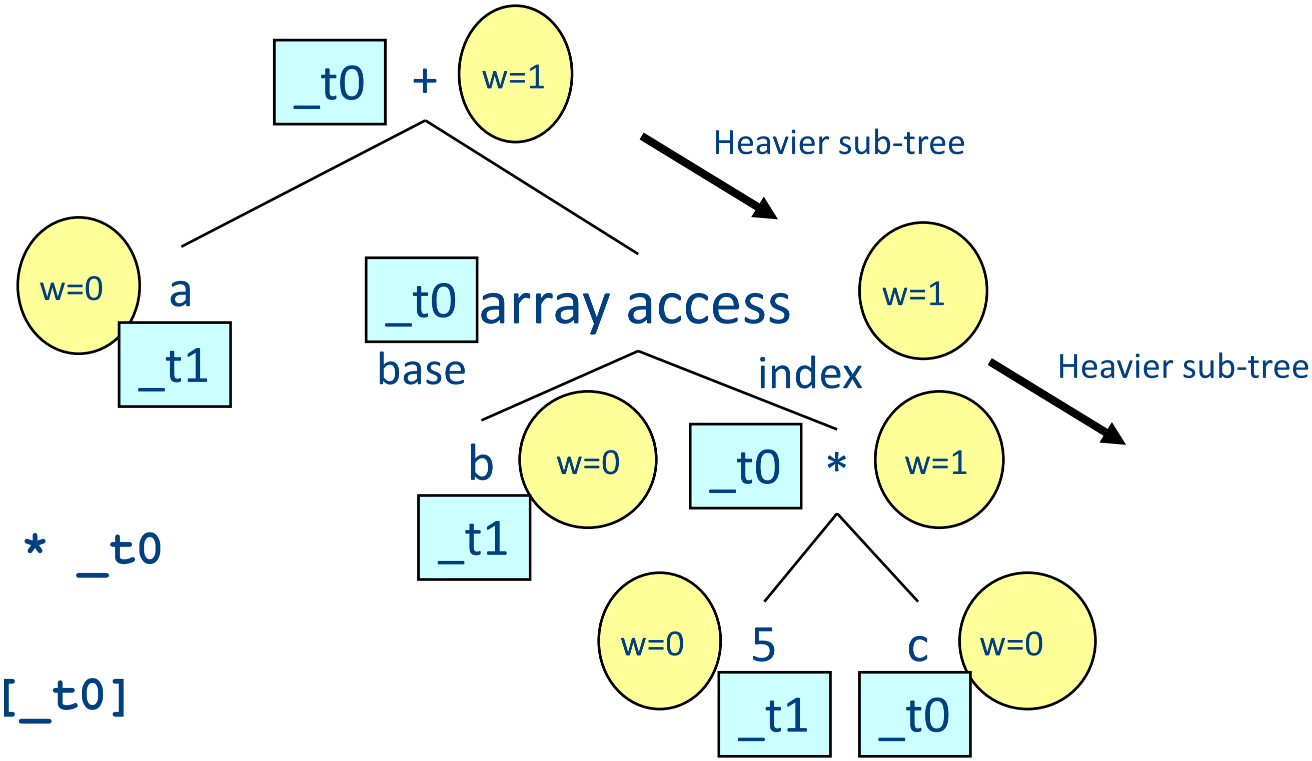
Phase 1: - check absence of side-effects in expression tree
- assign weight to each AST node



Weighted reg. alloc. example

_t0 = **cgen**(a+b[5*c])

Phase 2: - use weights to decide on order of translation




```
_t0 = c
_t1 = 5
_t0 = _t1 * _t0
_t1 = b
_t0 = _t1[_t0]
_t1 = a
_t0 = _t1 + _t0
```

Note on weighted register allocation

- **Must** reset temporaries counter after every statement: **x=y; y=z**
  - should **not** be translated to

```
_t0 = y;
x = _t0;
_t1 = z;
y = _t1;
```

  - But rather to

```
_t0 = y;
x = _t0;   # Finished translating statement. Set c=0
_t0 = z;
y= _t0;
```

Code generation for procedure calls (+ a few words on the runtime system)

Code generation for procedure calls

- Compile time generation of code for procedure invocations

- Activation Records (aka Stack Frames)

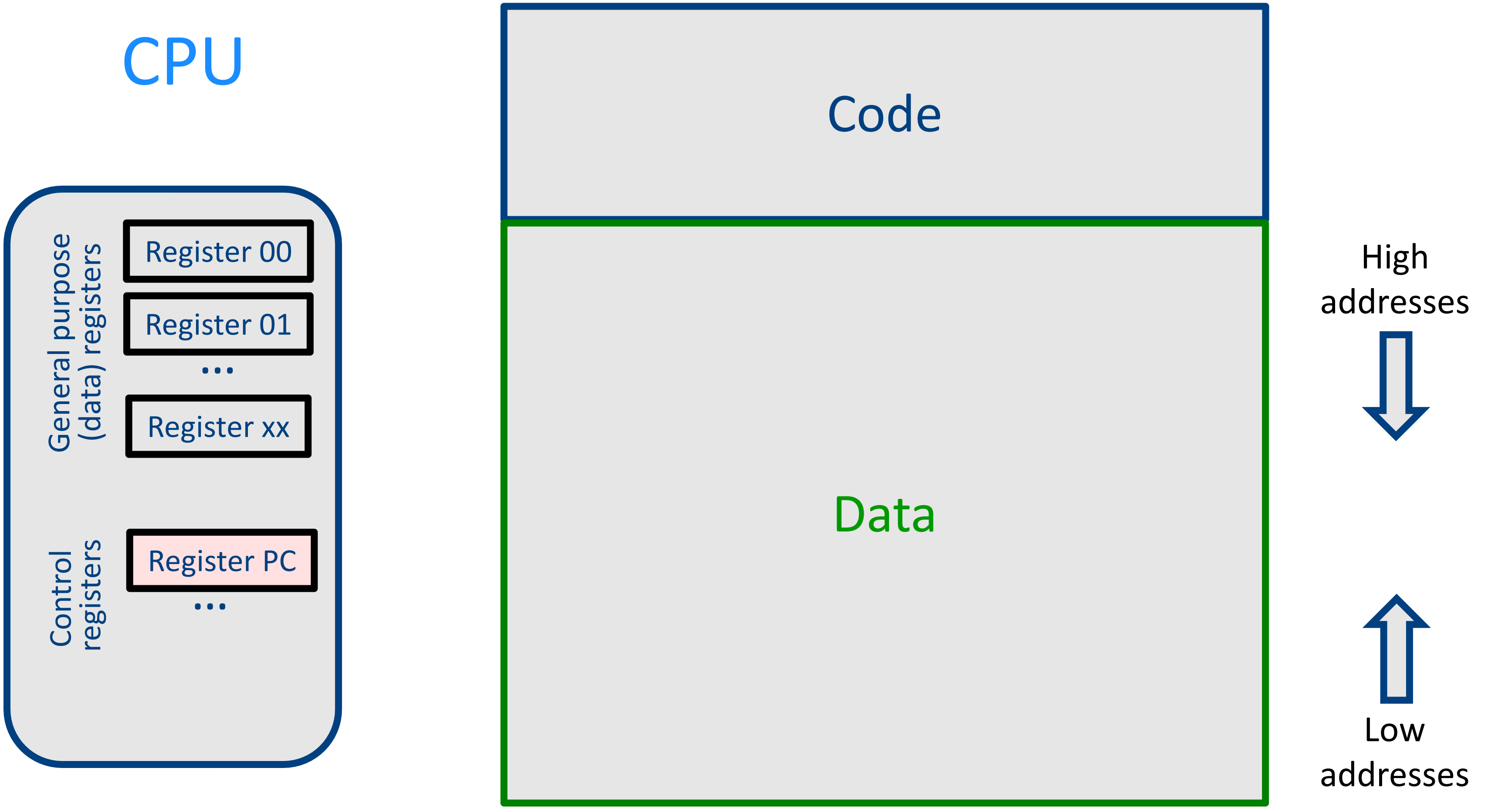
Supporting Procedures

- **Stack**: a new computing environment
  - e.g., temporary memory for **local variables**
- Passing information into the new environment
  - **Parameters**
- **Transfer** of **control** to/from procedure
- Handling return values

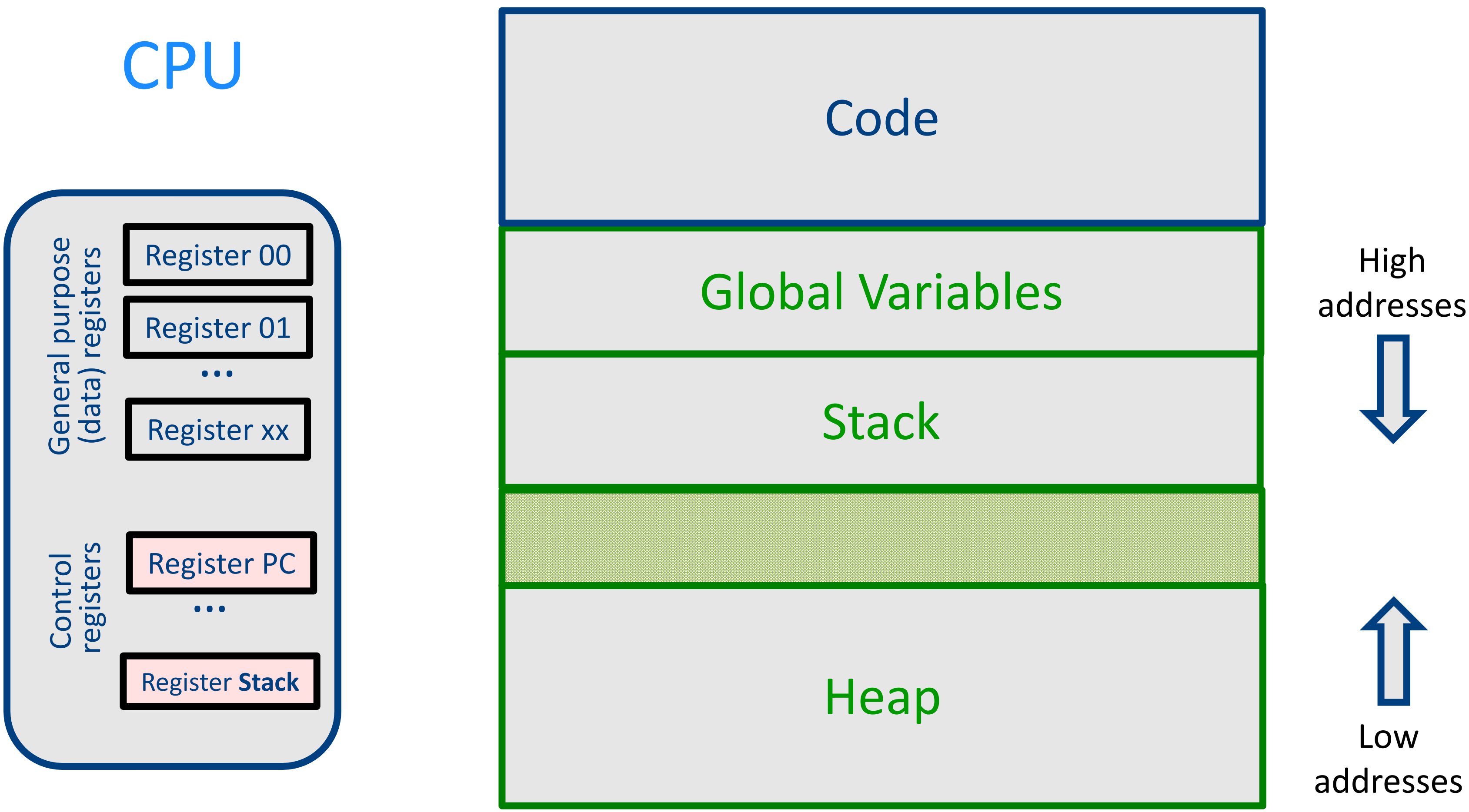
Calling Conventions

- In general, compiler can use any convention to handle procedures
- In practice, CPUs specify standards
    - Aka calling conventios
  - Allows for compiler interoperability
    - Libraries!

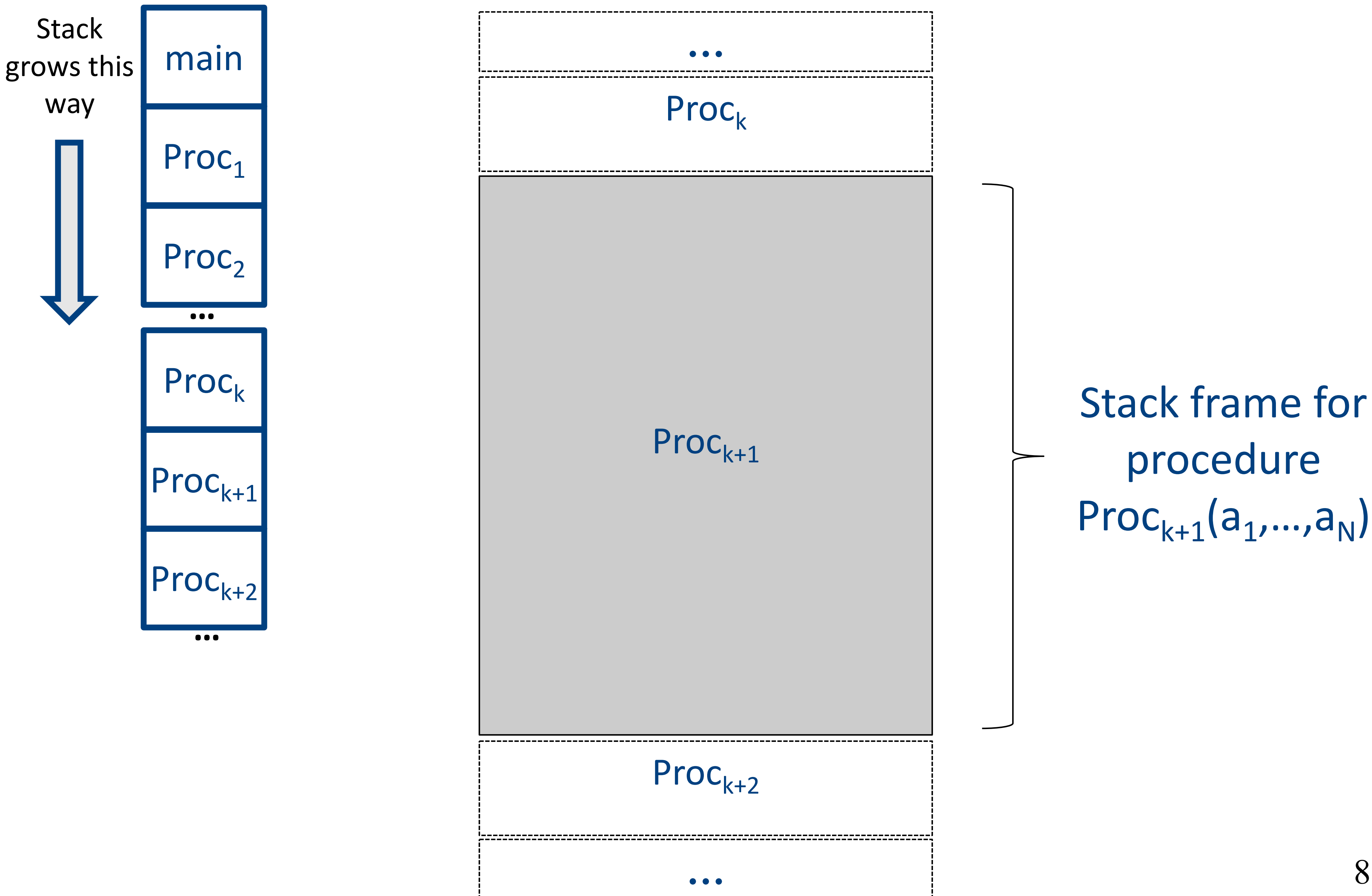
Abstract Register Machine
(High Level View)



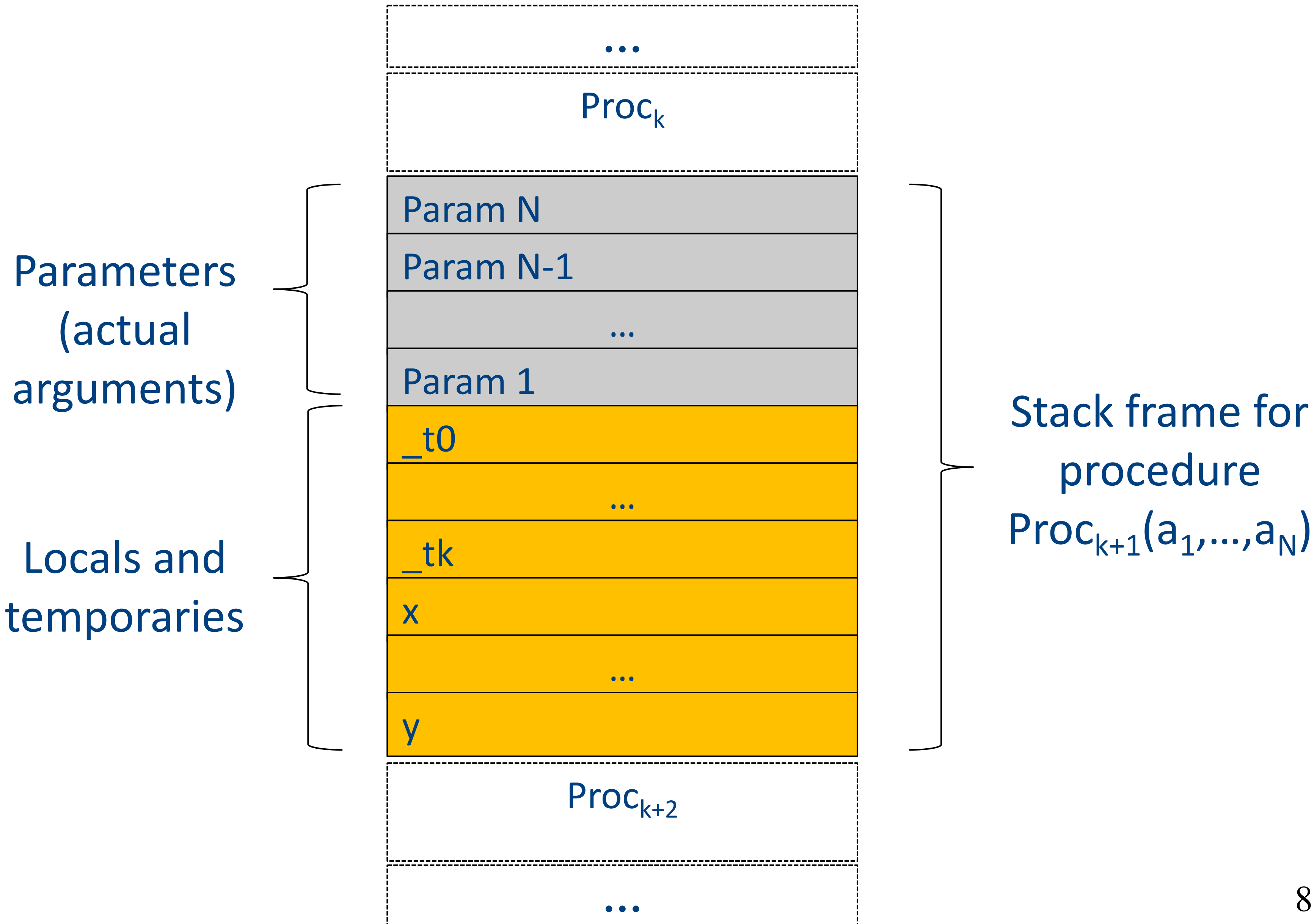
Abstract Register Machine
(High Level View)



Abstract Activation Record Stack



Abstract Stack Frame



Handling Procedures

- Store local variables/temporaries in a stack
- A function call instruction pushes arguments to stack and jumps to the function label
  A statement **`x=f(a1,…,an);`** looks like

  ```
  Push a1; … Push an;
  Call f;
  Pop x; // copy returned value
  ```

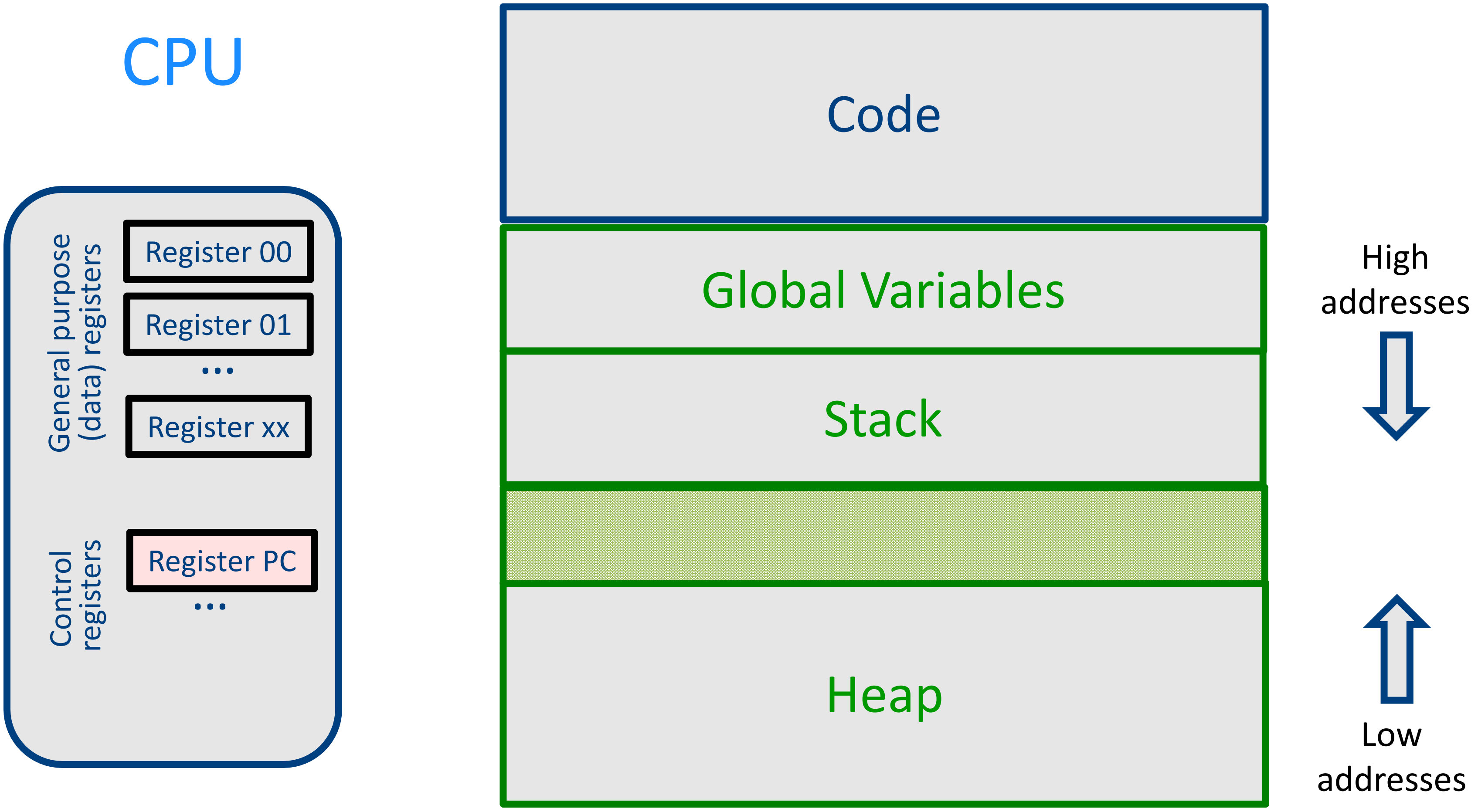
- Returning a value is done by pushing it to the stack (**`return x;`**)

  ```
  Push x;
  ```

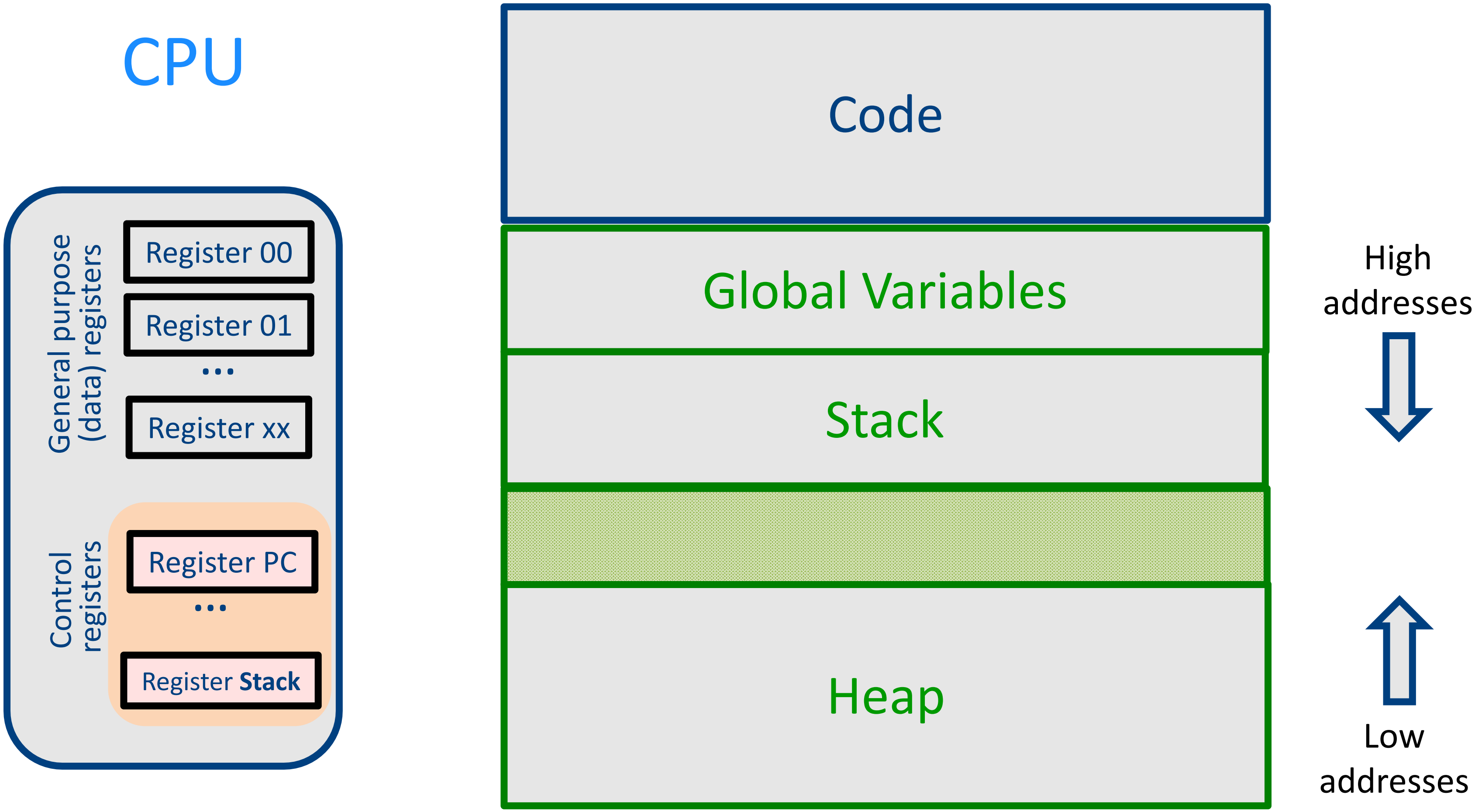
- Return control to caller (and roll up stack)

  ```
  Return;
  ```

Abstract Register Machine



Abstract Register Machine



Intro: Functions Example

```
int SimpleFn(int z) {
    int x, y;
    x = x * y * z;
    return x;
}

void main() {
   int w;
   w = SimpleFunction(137);
}
```

```
_SimpleFn:
_t0 = x * y;
_t1 = _t0 * z;
x = _t1;
Push x;
Return;

main:
_t0 = 137;
Push _t0;
Call _SimpleFn;
Pop w;
```

What Can We Do with Procedures?

- Declarations & Definitions
- Call & Return
- Jumping out of procedures
- Passing & Returning procedures as parameters

Design Decisions

- Scoping rules
  - Static scoping vs. dynamic scoping
- Caller/callee conventions
  - Parameters
  - Who saves register values?
- Allocating space for local variables

Static (lexical) Scoping

```
main ( )
{
    int a = 0 ;
    int b = 0 ;
    {
        int b = 1 ;
        {
            int a = 2 ;
            printf (“%d %d\n”, a, b)
        }
        {
            int b = 3 ;
            printf (“%d %d\n”, a, b) ;
        }
        printf (“%d %d\n”, a, b) ;
    }
    printf (“%d %d\n”, a, b) ;
}
```

a name refers to its (closest) enclosing scope

known at compile time

Declaration	Scopes
a=0	B0,B1,B3
b=0	B0
b=1	B1,B2
a=2	B2
b=3	B3

Dynamic Scoping

- Each identifier is associated with a global stack of bindings
- When entering scope where identifier is declared
  - push declaration on identifier stack
- When exiting scope where identifier is declared
  - pop identifier stack
- **Evaluating the identifier in any context binds to the current top of stack**
- Determined **at runtime**

Example

```
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { return g(); }
```

- What value is returned from main?
  - Static scoping?
  - Dynamic scoping?

Why do we care?

- We need to generate code to access variables

- Static scoping
  - Identifier binding is known at compile time
  - “Address” of the variable is known at compile time
  - Assigning addresses to variables is part of code generation
  - No runtime errors of “access to undefined variable”
  - Can check types of variables

Variable addresses for static scoping: first attempt

```
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { return g(); }
```

identifier	address
x (global)	0x42
x (inside g)	0x73