Compilation

0368-3133

Lecture 6:

Attribute Grammars

IR

Noam Rinetzky

Context Analysis

- Identification
  - Gather information about each named item in the program
  - e.g., what is the declaration for each usage

- Context checking
  - Type checking
  - e.g., the condition in an if-statement is a Boolean

Symbol table

```
month : integer RANGE [1..12];
…
month := 1;
while (month <= 12) {
  print(month_name[month]);
   month : = month + 1;
}
```

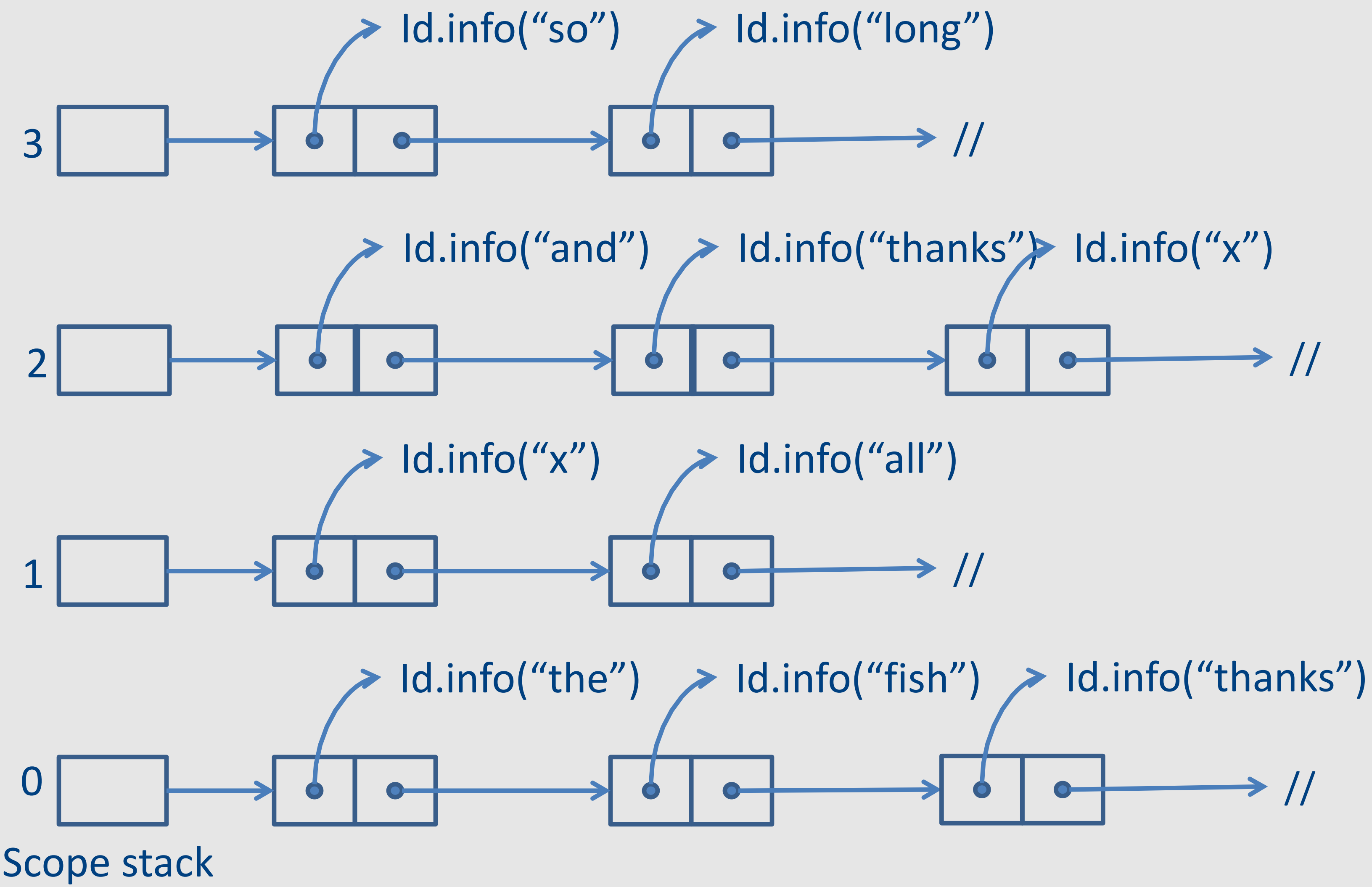
name	pos	type	…
month	1	RANGE[1..12]	
month_name	…	…	
…			

- A table containing information about identifiers in the program
- Single entry for each named item

Semantic Checks

- Scope rules
  - Use symbol table to check that
    - Identifiers defined before used
    - No multiple definition of same identifier
    - …
- Type checking
  - Check that types in the program are consistent
    - How?
    - Why?

Scope Info



(now just pointers to the corresponding record in the symbol table)

Type System

- A type system of a programming language is a way to define how “good” program “behave”
  - Good programs = well-typed programs
  - Bad programs = not well typed

- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime
- Type inference
  - Automatically infer types for a program (or show that there is no valid typing)

Typing Rules

If E1 has type int and E2 has type int,
then E1 + E2 has type int

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$

So far…

- Static correctness checking
  - Identification
  - Type checking
- Identification matches applied occurrences of identifier to its defining occurrence
  - The symbol table maintains this information
- Type checking checks which type combinations are legal
- Each node in the AST of an expression represents either an l-value (location) or an r-value (value)

How does this magic happen?

- We probably need to go over the AST?
- how does this relate to the clean formalism of the parser?

Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - How to update the attributes

- Attribute grammars

Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      Expr → Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise

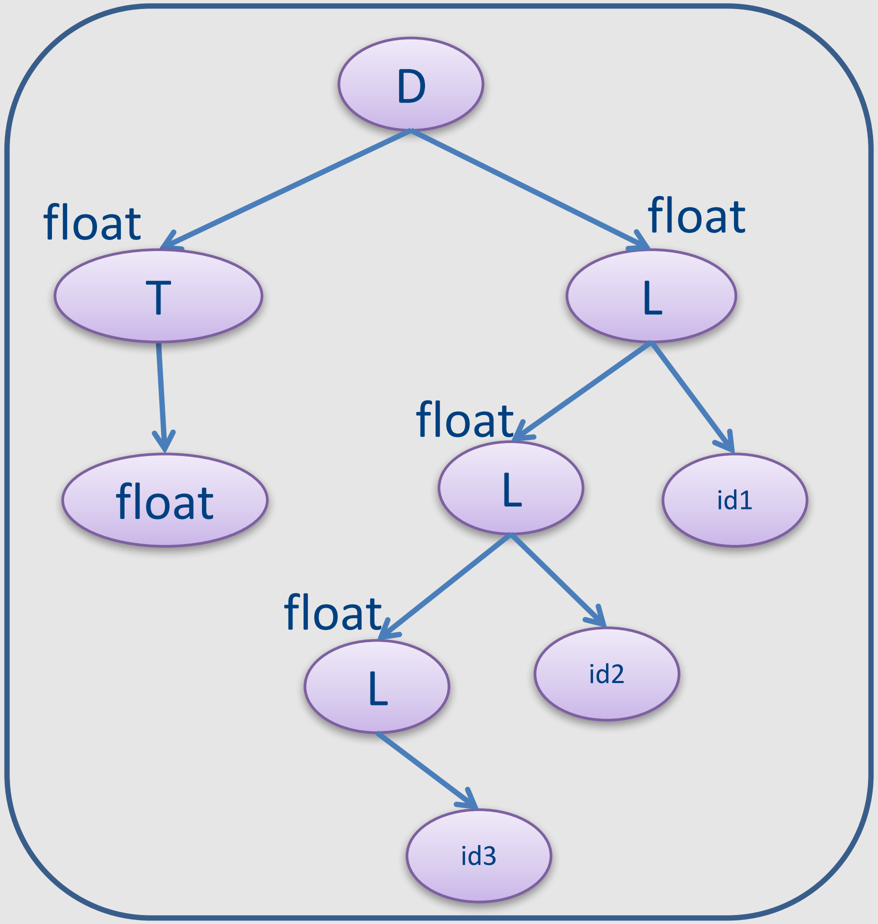
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

- Expr → Expr + Term
  Becomes
  Expr → Expr1 + Term

Example

float x,y,z



Production	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

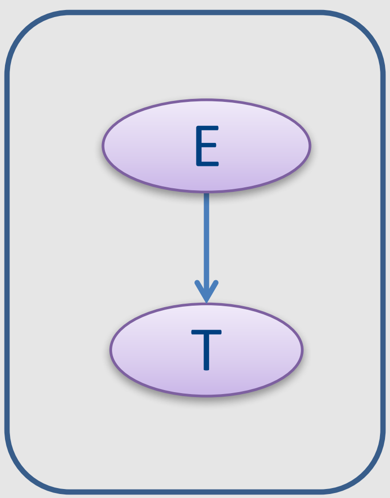
Dependencies

- A semantic equation a = b1,…,bm requires computation of b1,…,bm to determine the value of a

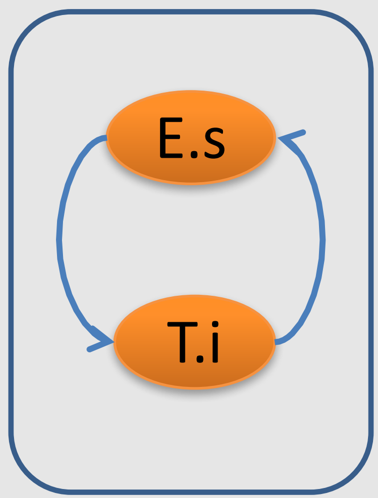
- The value of a depends on b1,…,bm
  - We write a → bi

Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values



AST



Dependence graph

E.s = T.i

T.i = E.s + 1

Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering

- Works as long as there are no cycles

Building Dependency Graph

- All semantic equations take the form

  attr1 = func1(attr1.1, attr1.2,…)
  attr2 = func2(attr2.1, attr2.2,…)

- Actions with side effects use a dummy attribute
- Build a directed dependency graph G
  - For every attribute a of a node n in the AST create a node n.a
  - For every node n in the AST and a semantic action of the form b = f(c1,c2,…ck) add edges of the form (ci,b)

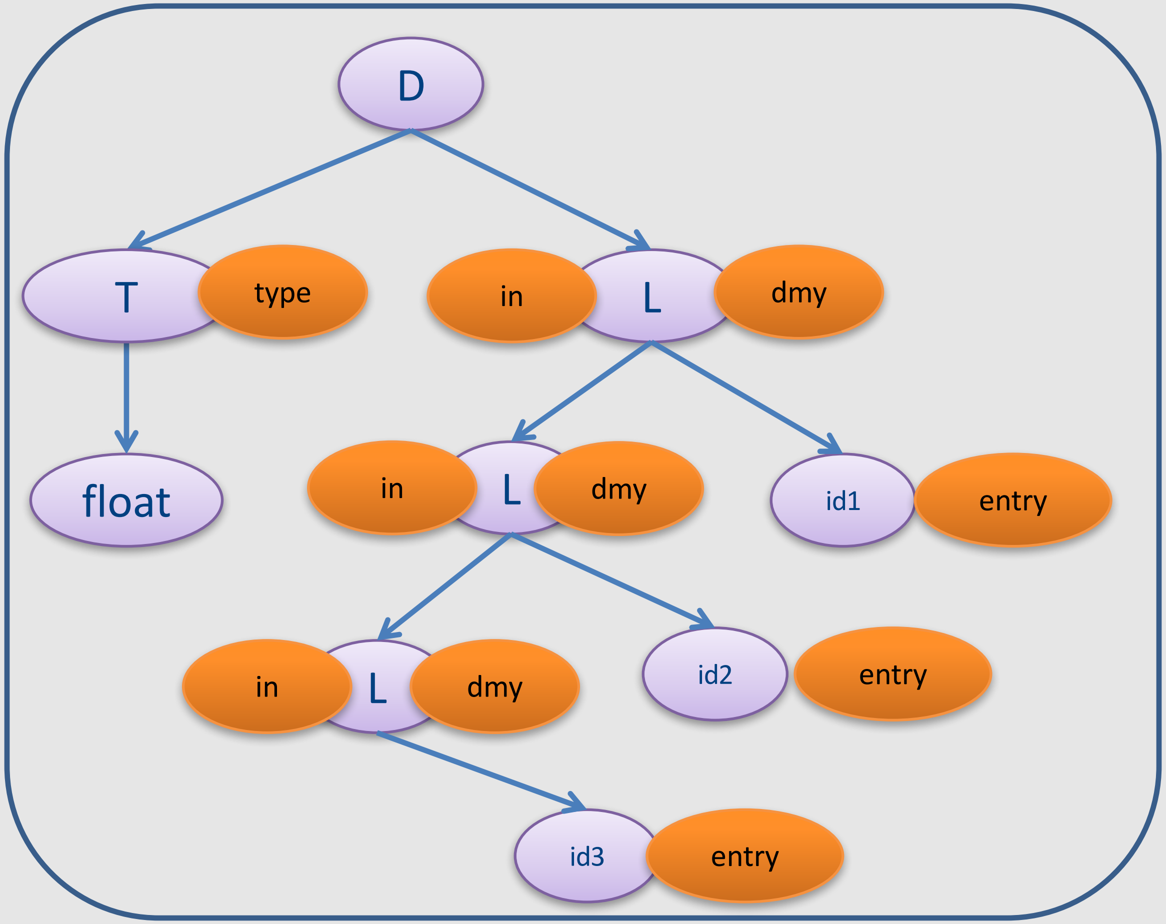
Production	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

Convention:
Add dummy variables for side effects.

Production	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in L.dmy = addType(id.entry,L.in)
L → id	L.dmy = addType(id.entry,L.in)

Example

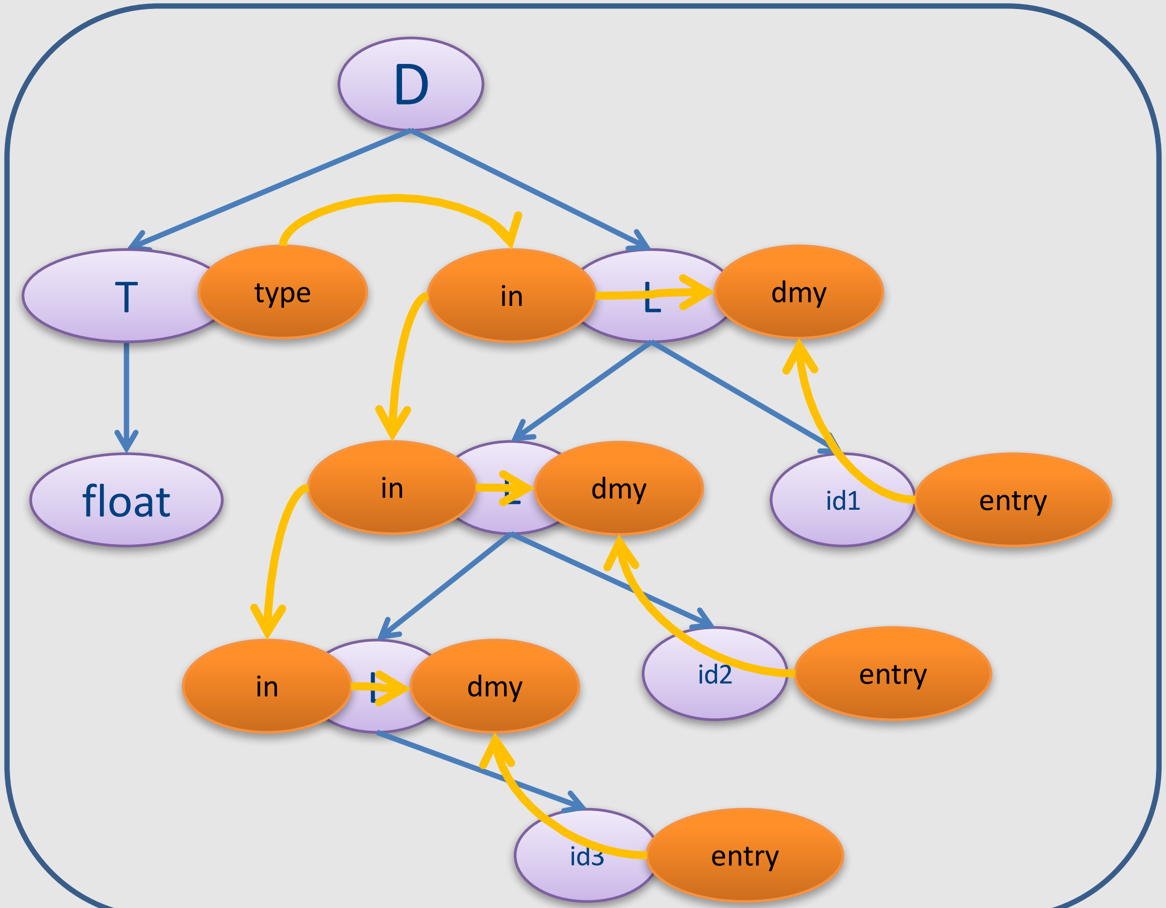
float x,y,z



Prod.	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

Example

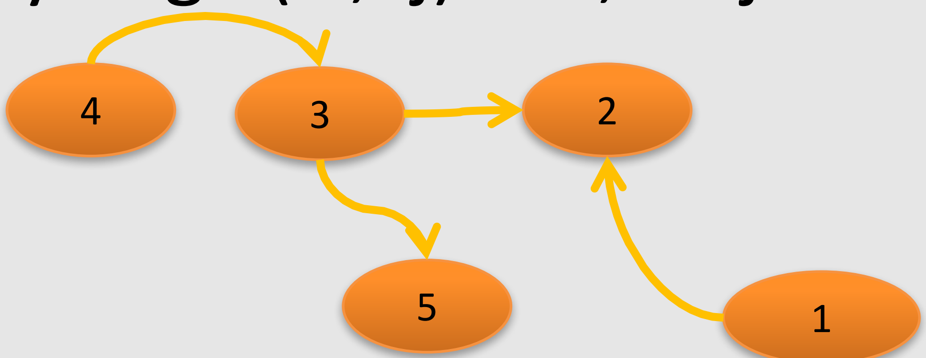
float x,y,z



Prod.	Semantic Rule
D ⟶ T L	L.in = T.type
T ⟶ int	T.type = integer
T ⟶ float	T.type = float
L ⟶ L1, id	L1.in = L.in addType(id.entry,L.in)
L ⟶ id	addType(id.entry,L.in)

Topological Order

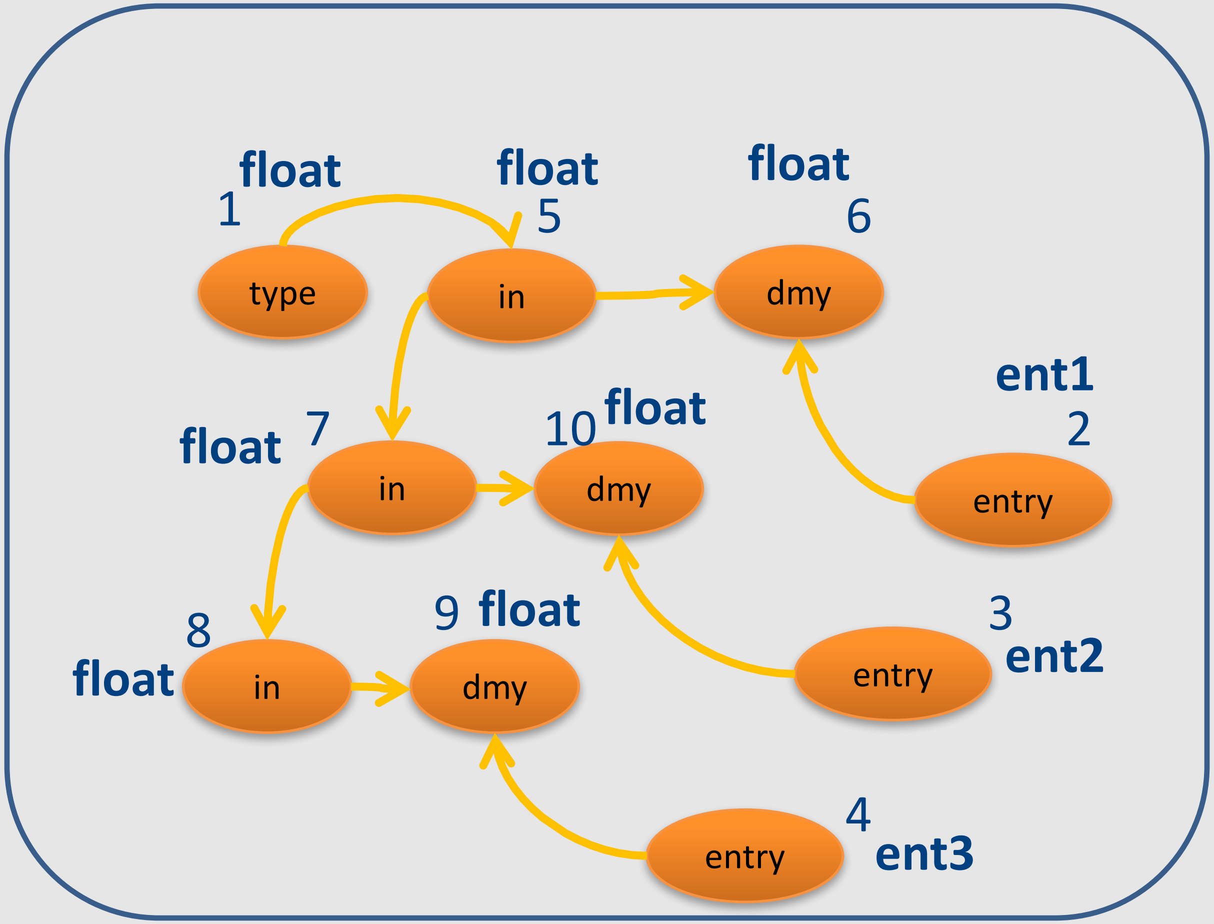
- For a graph G=(V,E), |V|=k
- Ordering of the nodes v1,v2,…vk such that for every edge (vi,vj) $\in$ E, i < j



Example topological orderings: 1 4 3 2 5, 4 1 3 5 2

Example

float x,y,z



But what about cycles?

- For a given attribute grammar hard to detect if it has cyclic dependencies
  - Exponential cost

- Special classes of attribute grammars
  - Our “usual trick”
  - sacrifice generality for predictable performance

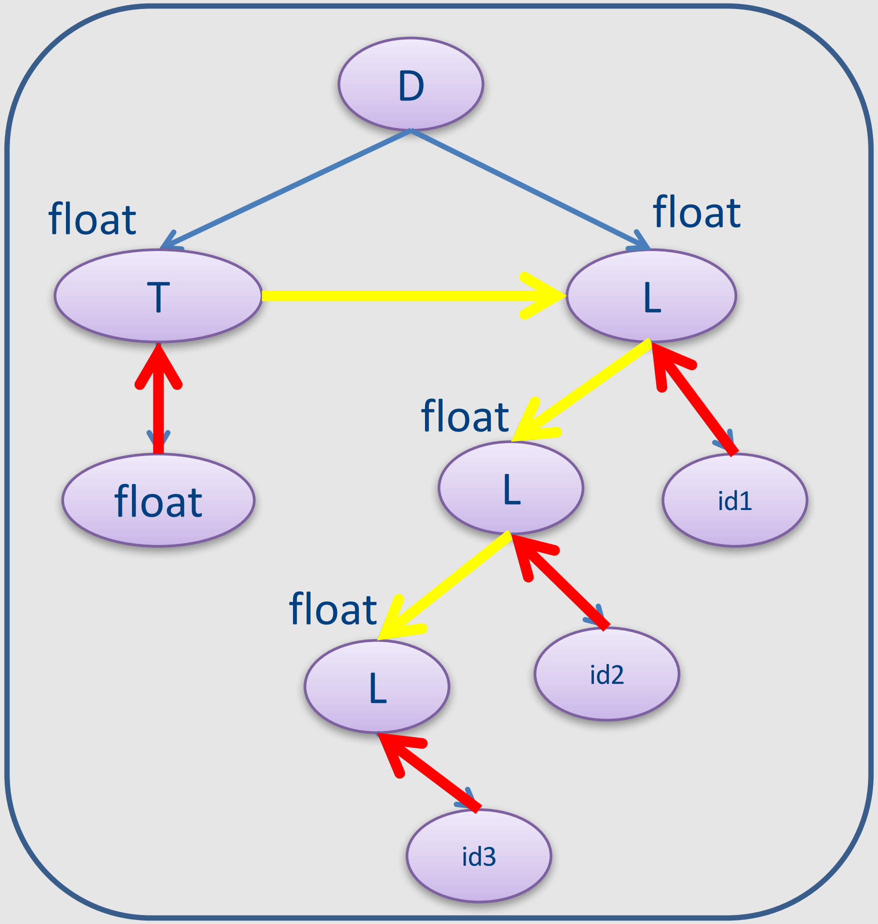
Inherited vs. Synthesized Attributes

- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node

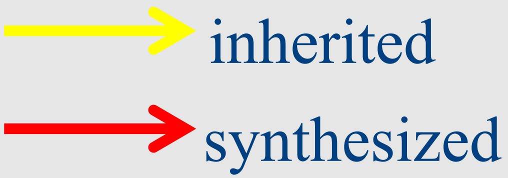
- Attributes of tokens are technically considered as synthesized attributes

example

float x,y,z



Production	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)



S-attributed Grammars

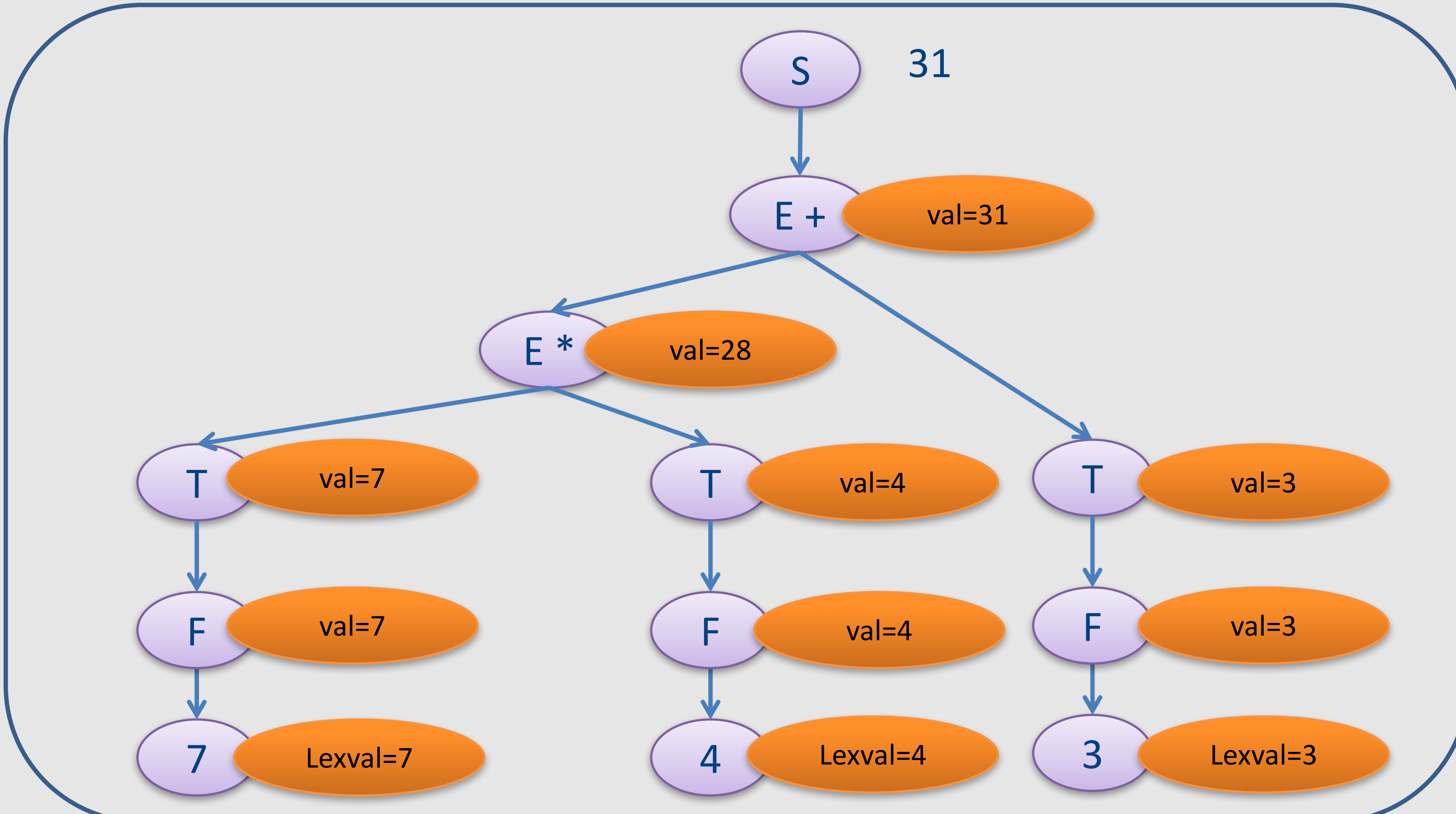
- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes

- Can be computed by any bottom-up parser during parsing
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children

S-attributed Grammar: example

Production	Semantic Rule
S→ E ;	print(E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

example

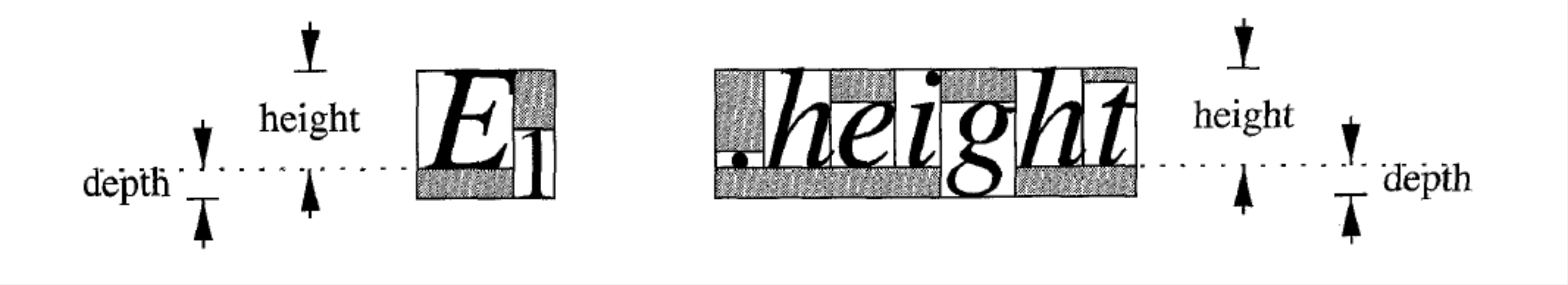


S
31
E +
val=31
E *
val=28
T
val=7
T
val=4
T
val=3
F
val=7
F
val=4
F
val=3
7
Lexval=7
4
Lexval=4
3
Lexval=3

L-attributed grammars

- L-attributed attribute grammar when every attribute in a production A → X1…Xn is
  - A synthesized attribute, or
  - An inherited attribute of Xj, 1 <= j <=n that only depends on
    - Attributes of X1…Xj-1 to the left of Xj, or
    - Inherited attributes of A

Example: typesetting



- Each box is built from smaller boxes from which it gets the height and depth, and to which it sets the point size.
- pointsize (ps) – size of letters in a box. Subscript text has smaller point size of o.7p.
- height (ht) – distance from top of the box to the baseline
- depth (dp) – distance from baseline to the bottom of the box.

Example: typesetting

production	semantic rules
S → B	B.ps = 10
B → B1 B2	B1.ps = B.ps B2.ps = B.ps B.ht = max(B1.ht,B2.ht) B.dp = max(B1.dp,B2.dp)
B → B1 sub B2	B1.ps = B.ps B2.ps = 0.7*B.ps B.ht = max(B1.ht,B2.ht – 0.25*B.ps) B.dp = max(B1.dp,B2.dp– 0.25*B.ps)
B → text	B.ht = getHt(B.ps,text.lexval) B.dp = getDp(B.ps,text.lexval)

Computing the attributes from left to right during a DFS traversal

```
procedure dfvisit (n: node);
begin
    for each child m of n, from left to right
      begin
         evaluate inherited attributes of m;
         dfvisit (m)
      end;
    evaluate synthesized attributes of n
end
```

Summary

- Contextual analysis can move information between nodes in the AST
  - Even when they are not “local”
- Attribute grammars
  - Attach attributes and semantic actions to grammar
- Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- Special classes with pre-determined evaluation order: S-attributed, L-attributed

The End

- Front-end

Compilation

0368-3133 2014/15a

Lecture 6a

Getting into the back-end

Noam Rinetzky

But first, a short reminder

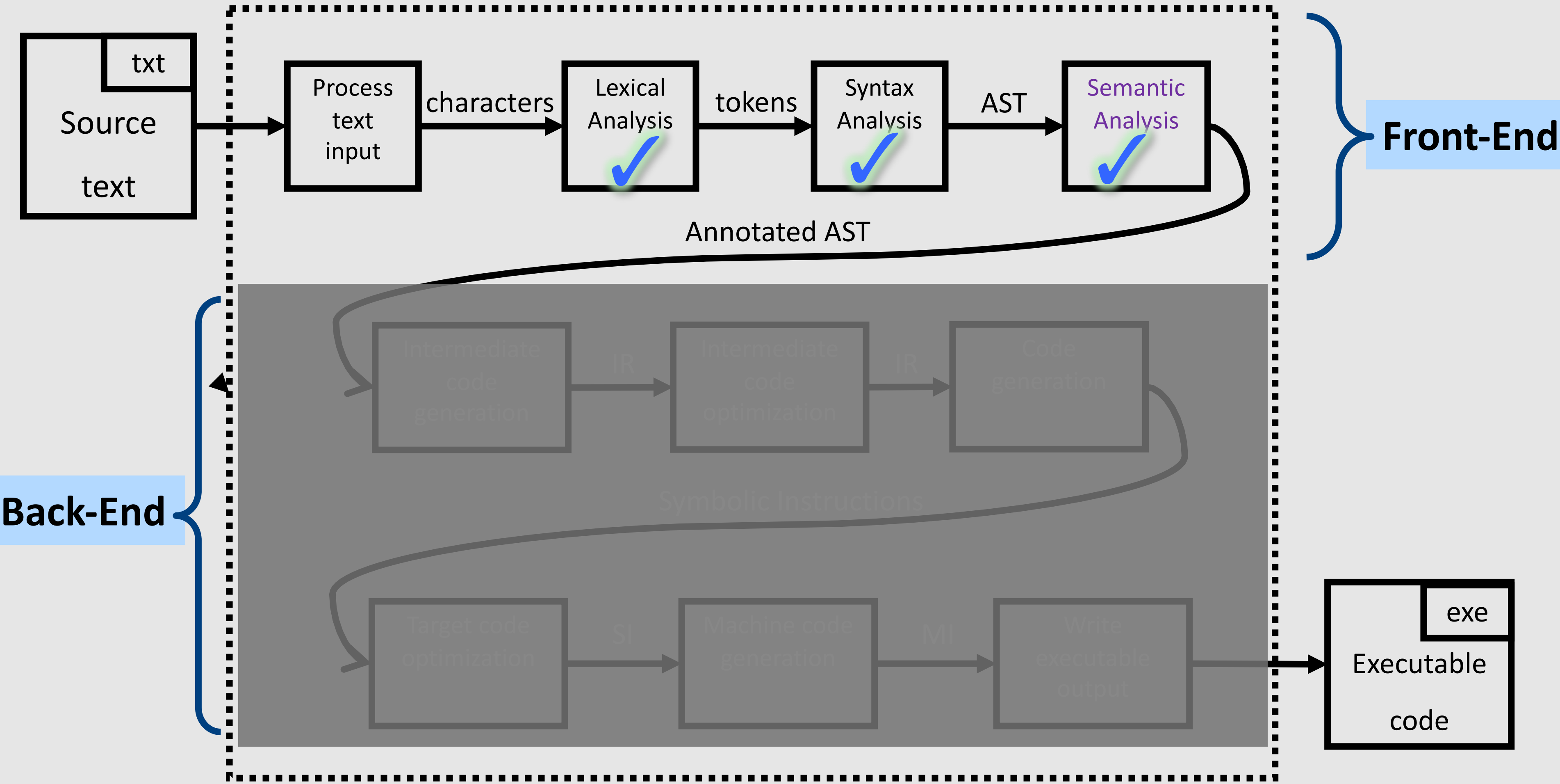
What is a compiler?

"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."

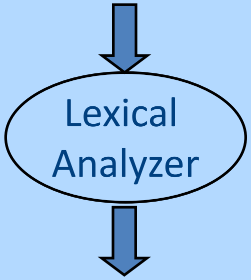
--Wikipedia

Where we were



Lexical Analysis

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

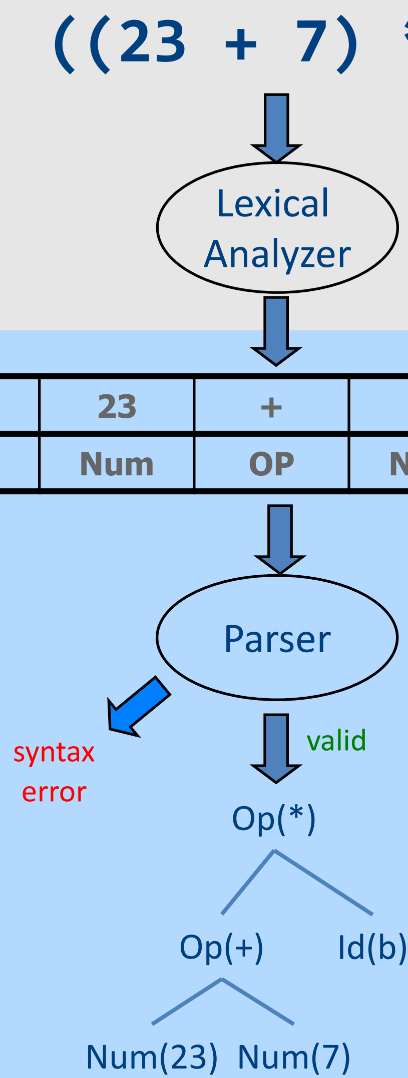


token stream

(	(	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

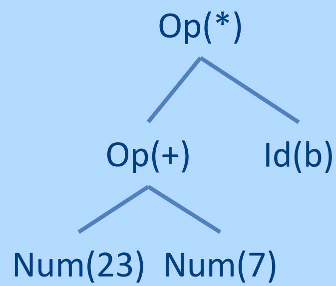
Id → 'a' | ... | 'z'

Abstract Syntax Tree

Context Analysis

Type rules

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$



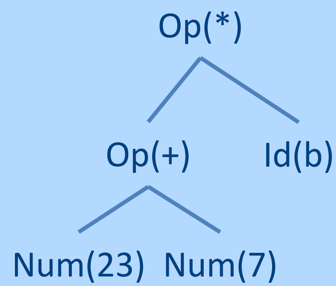
Abstract Syntax Tree

Semantic Error

Valid + Symbol Table

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***

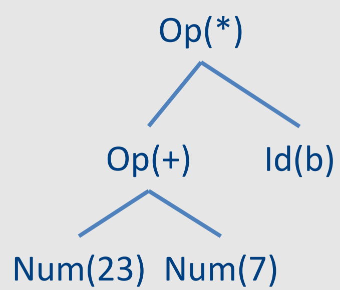
"A **compiler** is a computer program that **transforms** source **code** written in a programming language (source language) into another language (target language).

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

…



Valid Abstract Syntax Tree

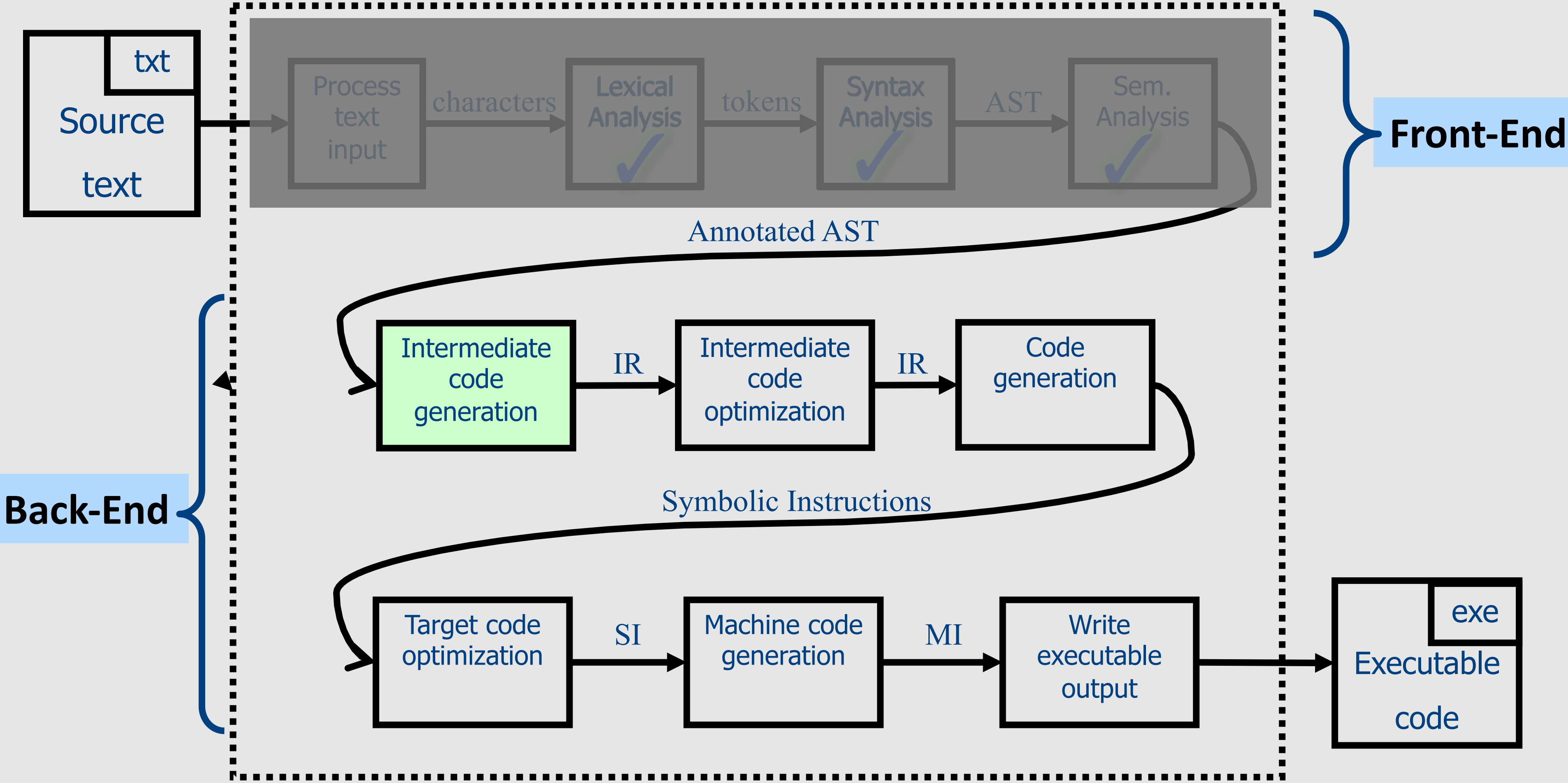
Symbol Table

Verification (possible runtime)
Errors/Warnings

Intermediate Representation (IR)

input → Executable Code → output

Where we are



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

1 Note: Compile Time vs Runtime

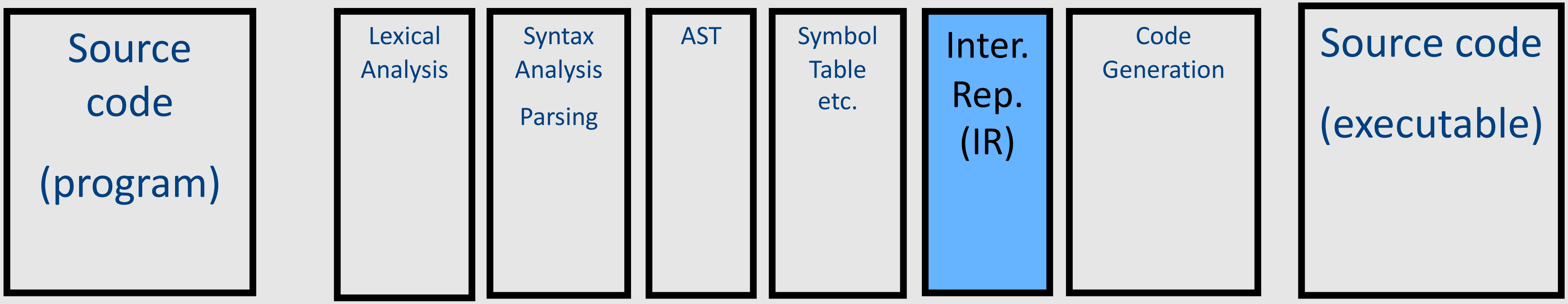
- Compile time: Data structures used during program compilation

- 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

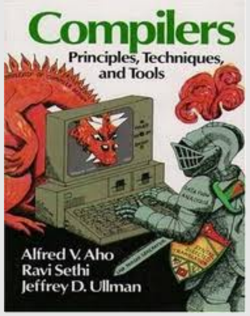
Intermediate Representation

Code Generation: IR



- Translating from abstract syntax (AST) to intermediate representation (IR)
  - Three-**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

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

IR

```
_t0 = b + c;
a = _t0 + d;
_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;
```

LIR (unoptimized)

```
_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

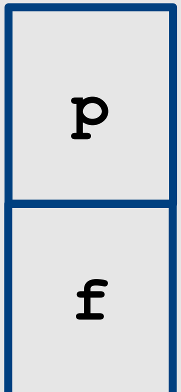
```
int x;
int y;

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

y = x;
```

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

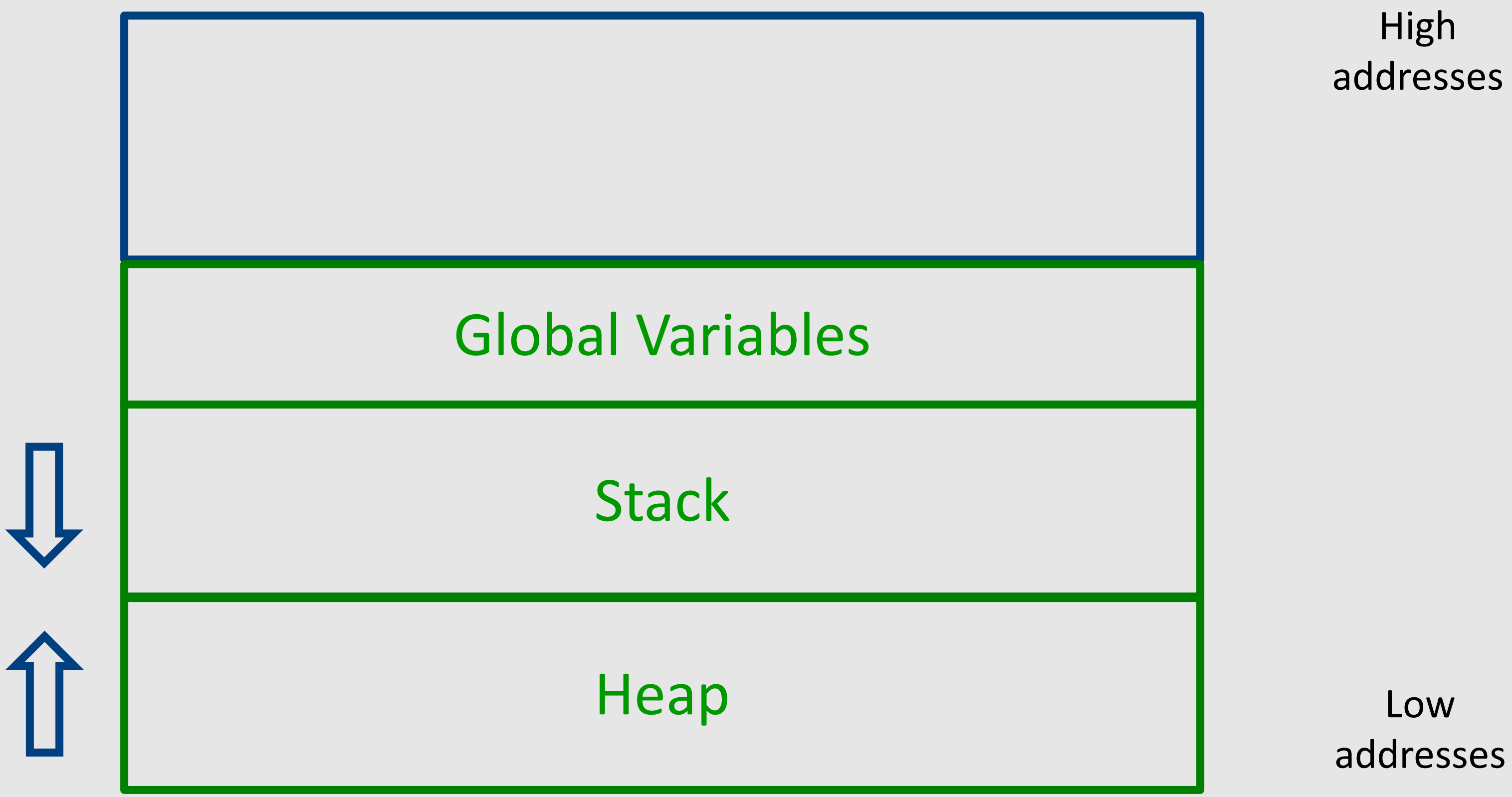
Procedures / Functions

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

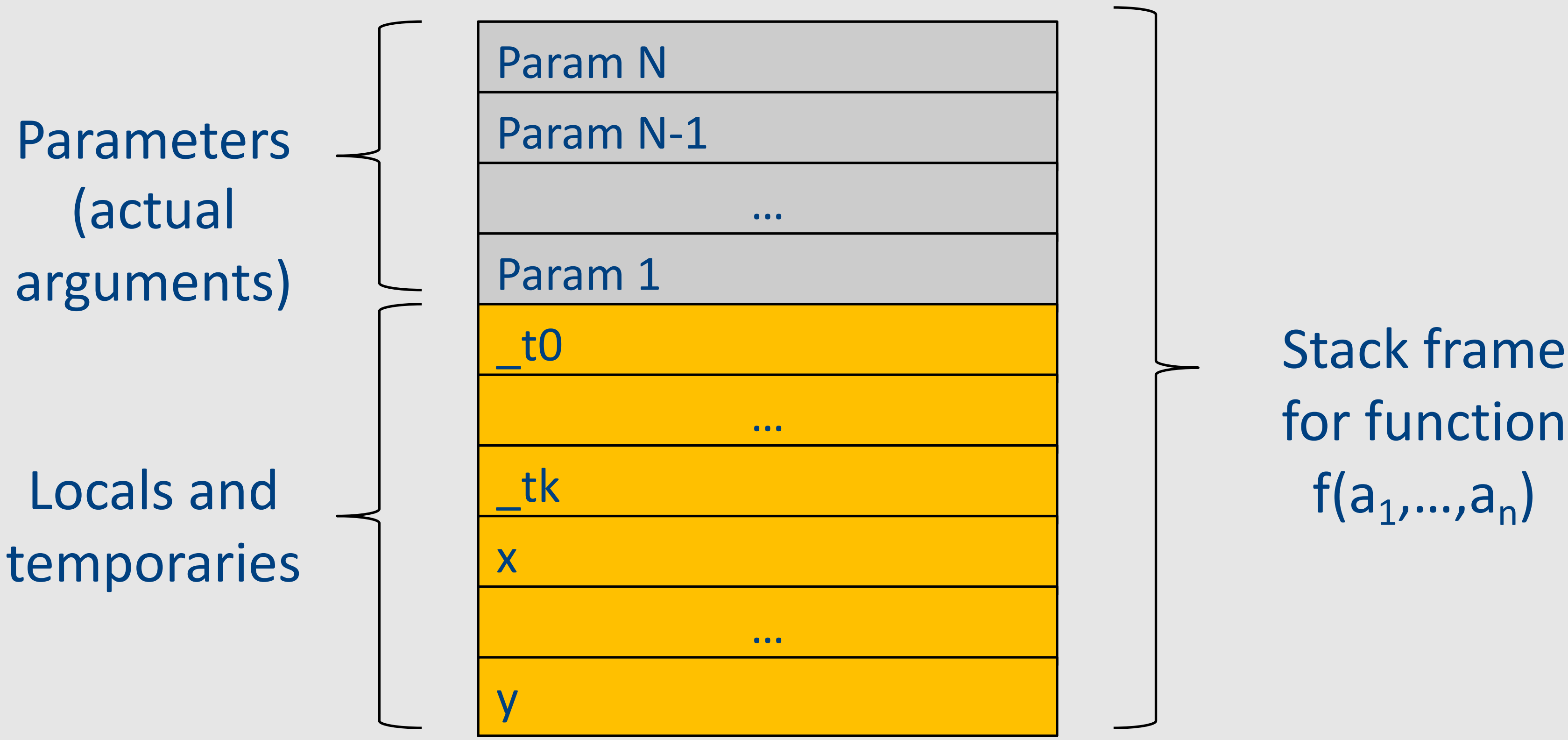
- What happens in runtime?



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]
```

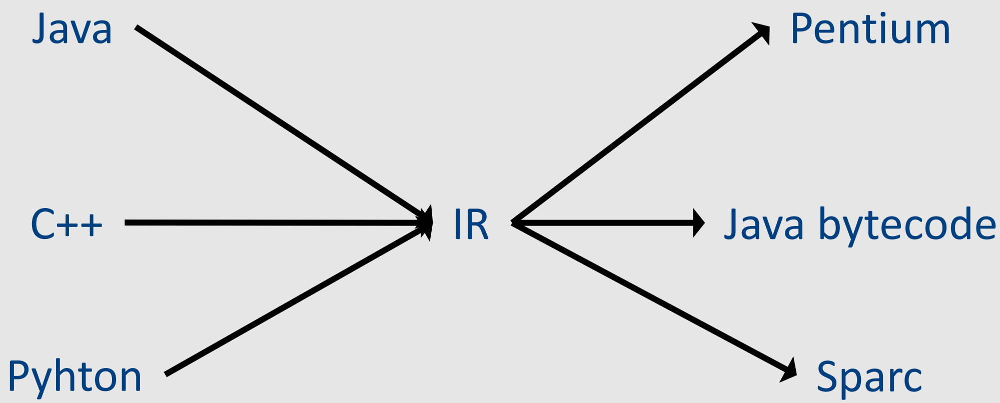
x[i] := y

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

IR Summary

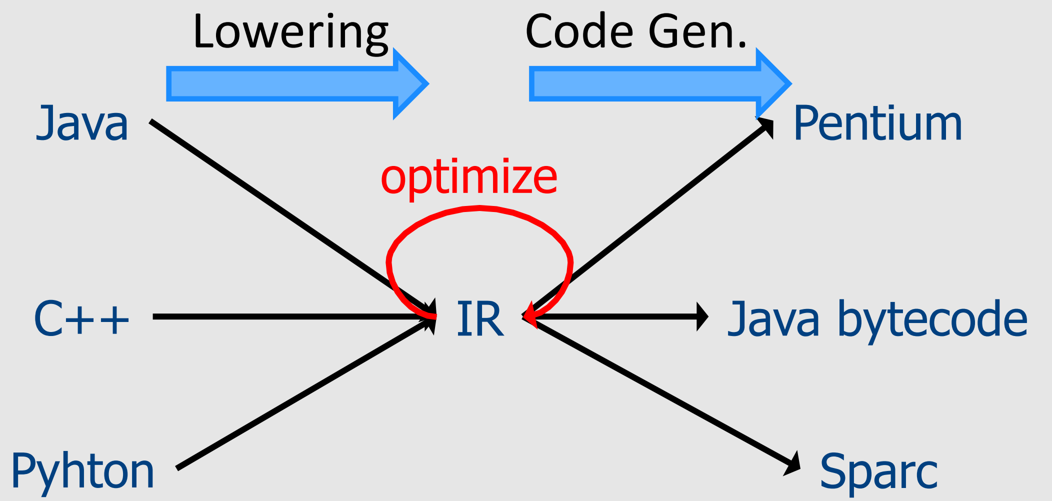
Intermediate representation

- 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



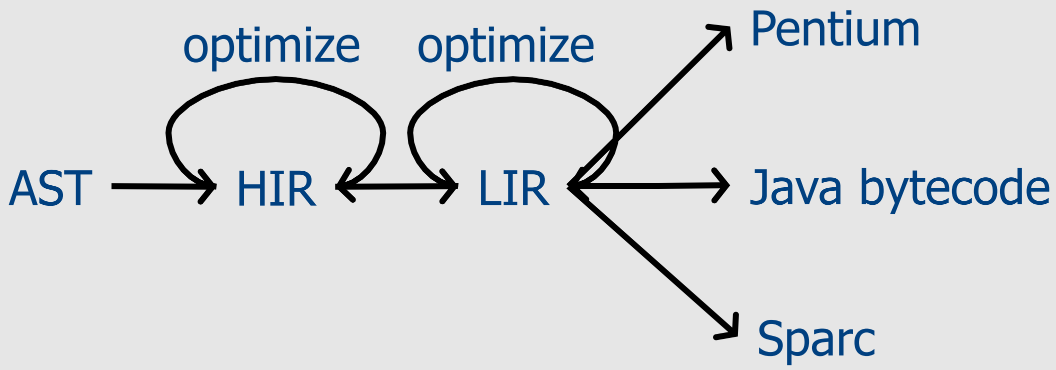
Intermediate representation

- 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

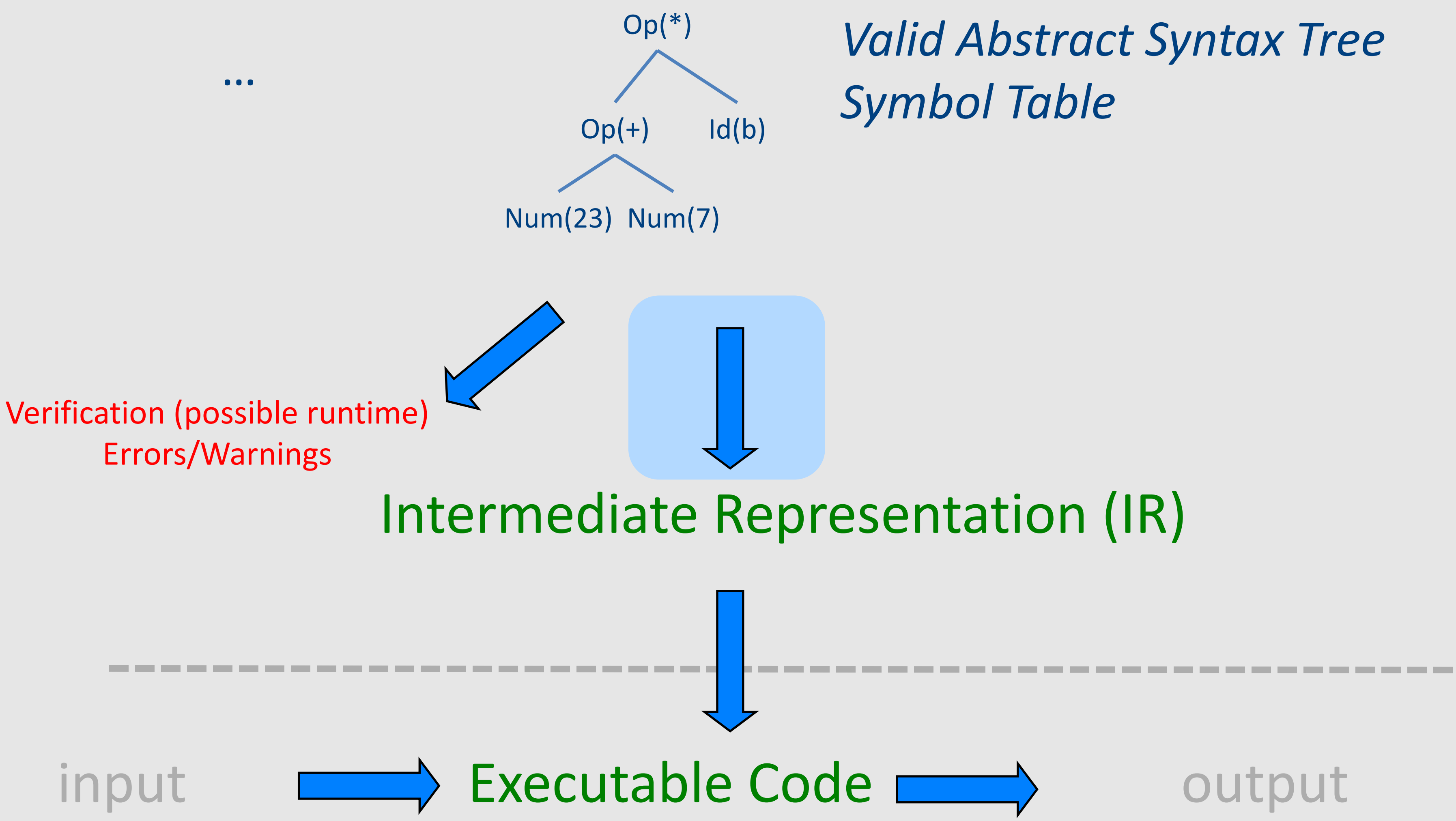


AST vs. LIR for imperative languages

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*
}

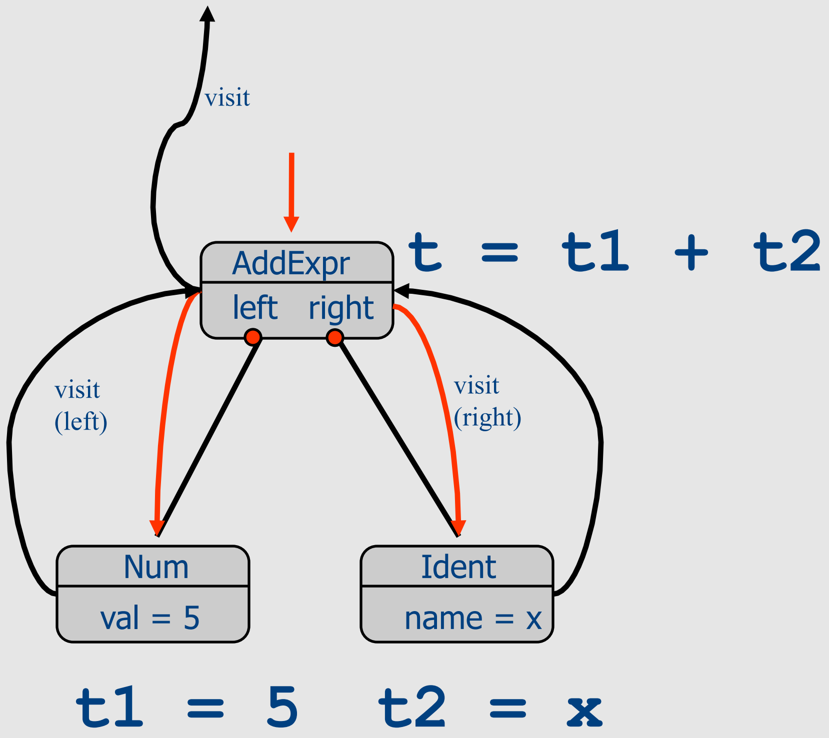
Returns an **arbitrary fresh** name

```
_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 = {
        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

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

```
_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 reused
  - 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 = cgen(c*d)-(e*f))

    _t0 = c*d
            ------- c = c + 1
    _t1 = e*f
            ------- c = c - 1
    _t0 = _t0 -_t1

------- c = c + 1

_t1 = a*b

------- c = c - 1

_t0 = _t0 + _t1
```

Temporaries

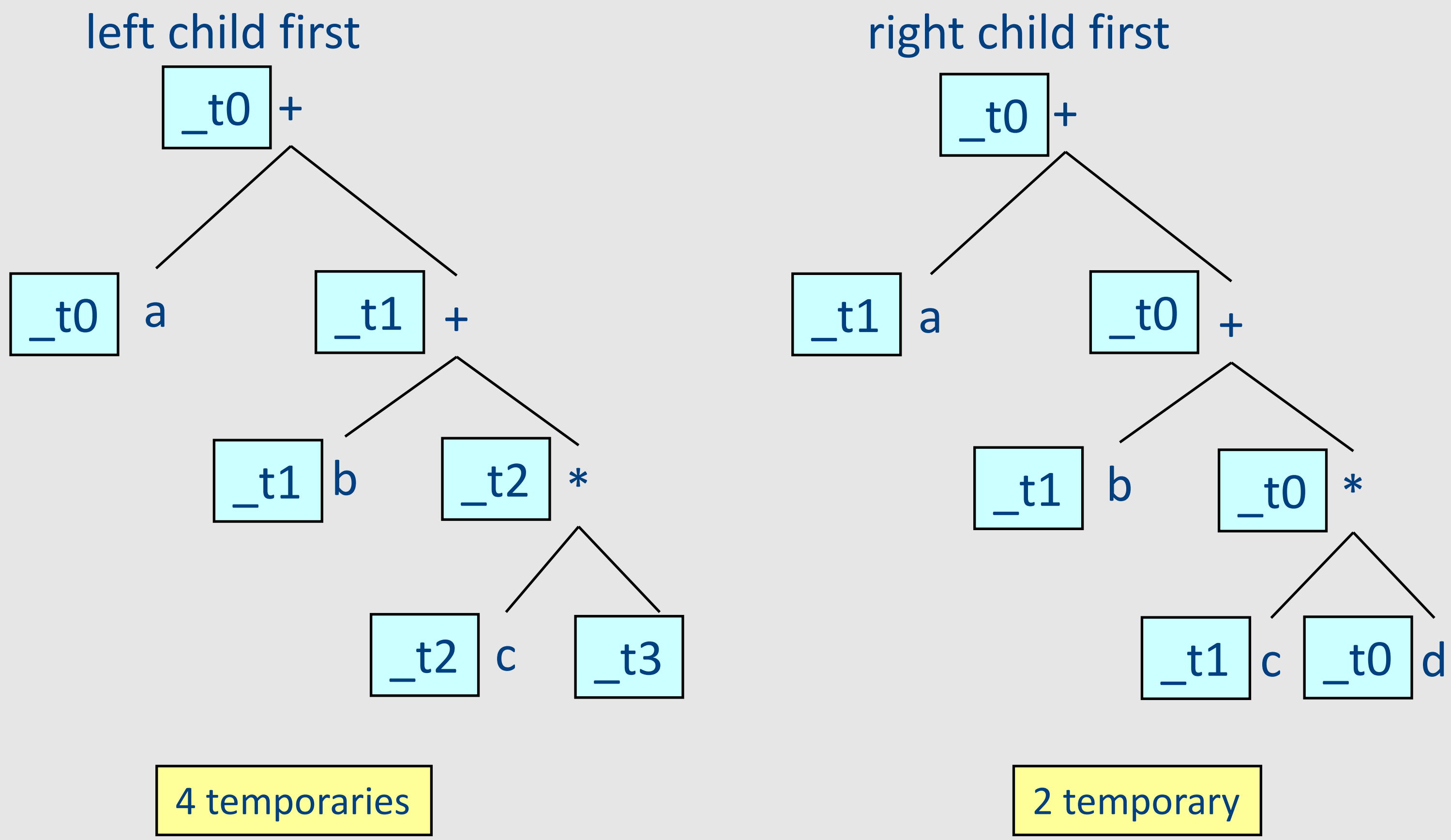
Weighted register allocation

- 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 = **cgen**(a+(b+(c*d)))

*+ and * are commutative operators*



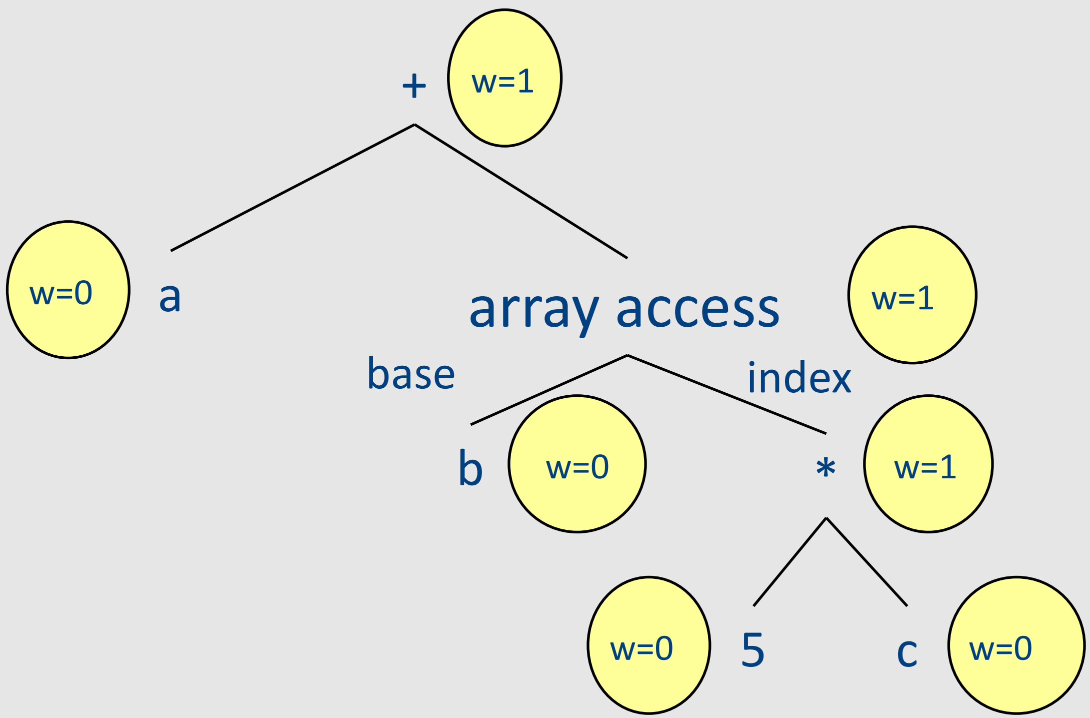
Weighted register allocation

- 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])

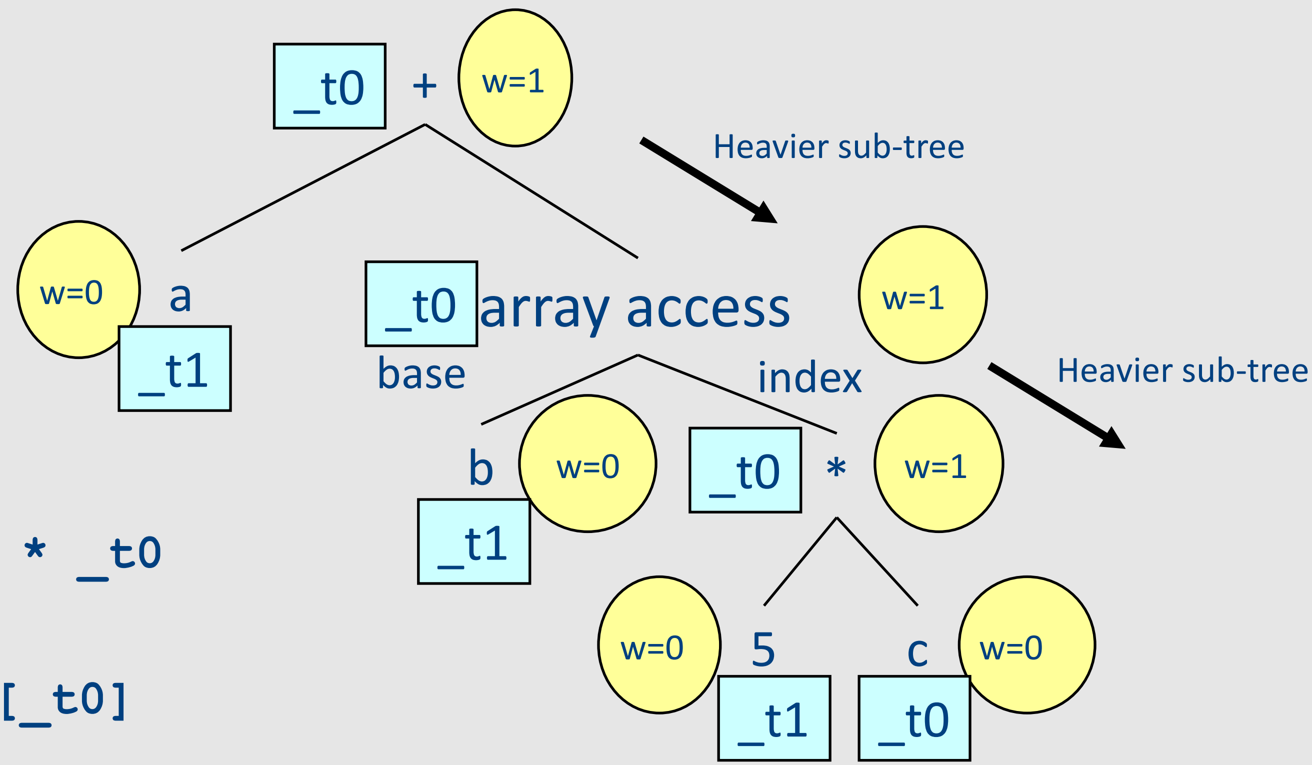
Phase 1: - check absence of side-effects in expression tree
- assign weight to each AST node



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

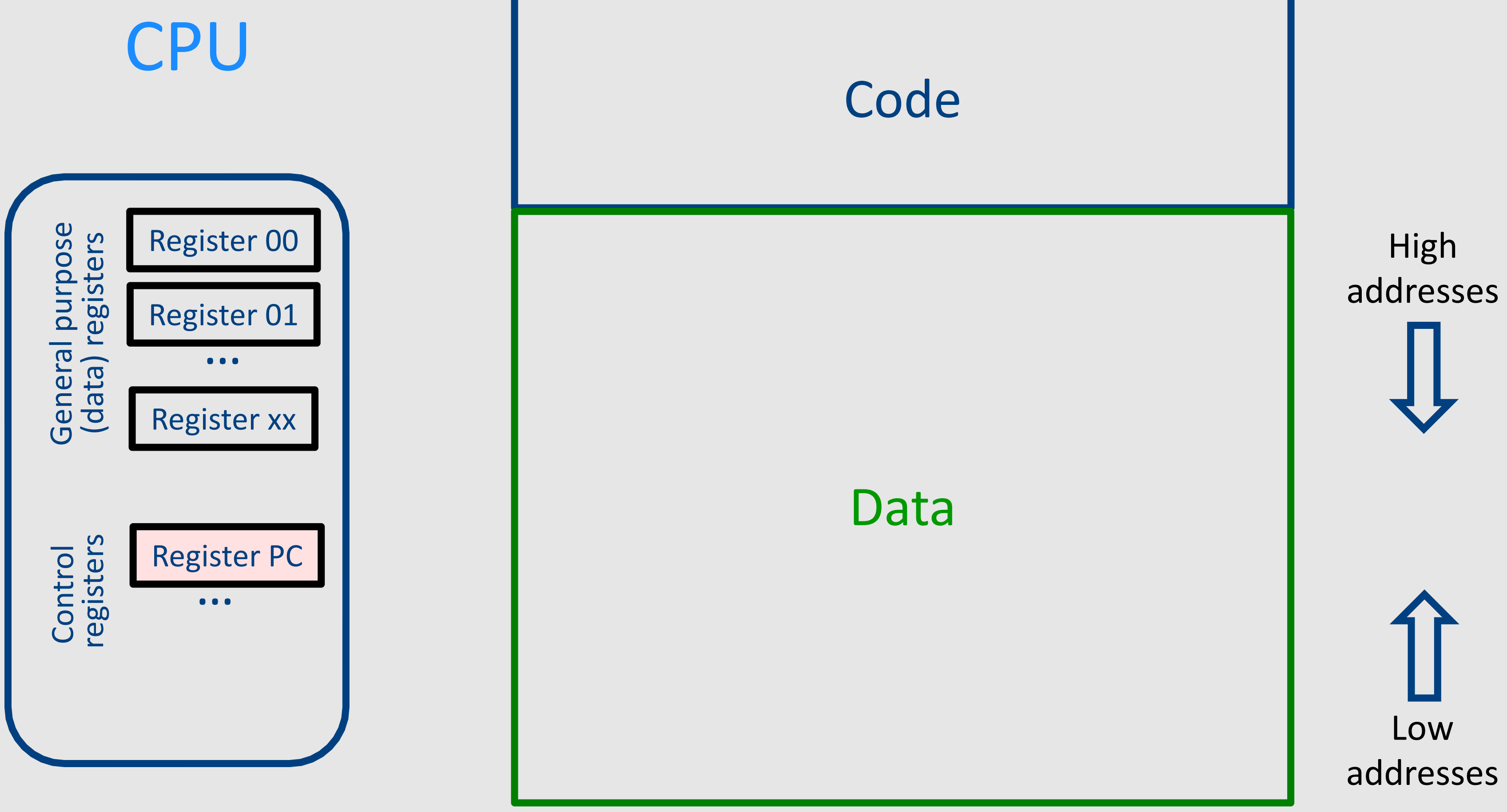
- **Stack**: a new computing environment
  - e.g., temporary memory for **local variables**
- Passing information into the new environment
  - **Parameters**
- **Transfer** of **control** to/from procedure
- Handling return values

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!

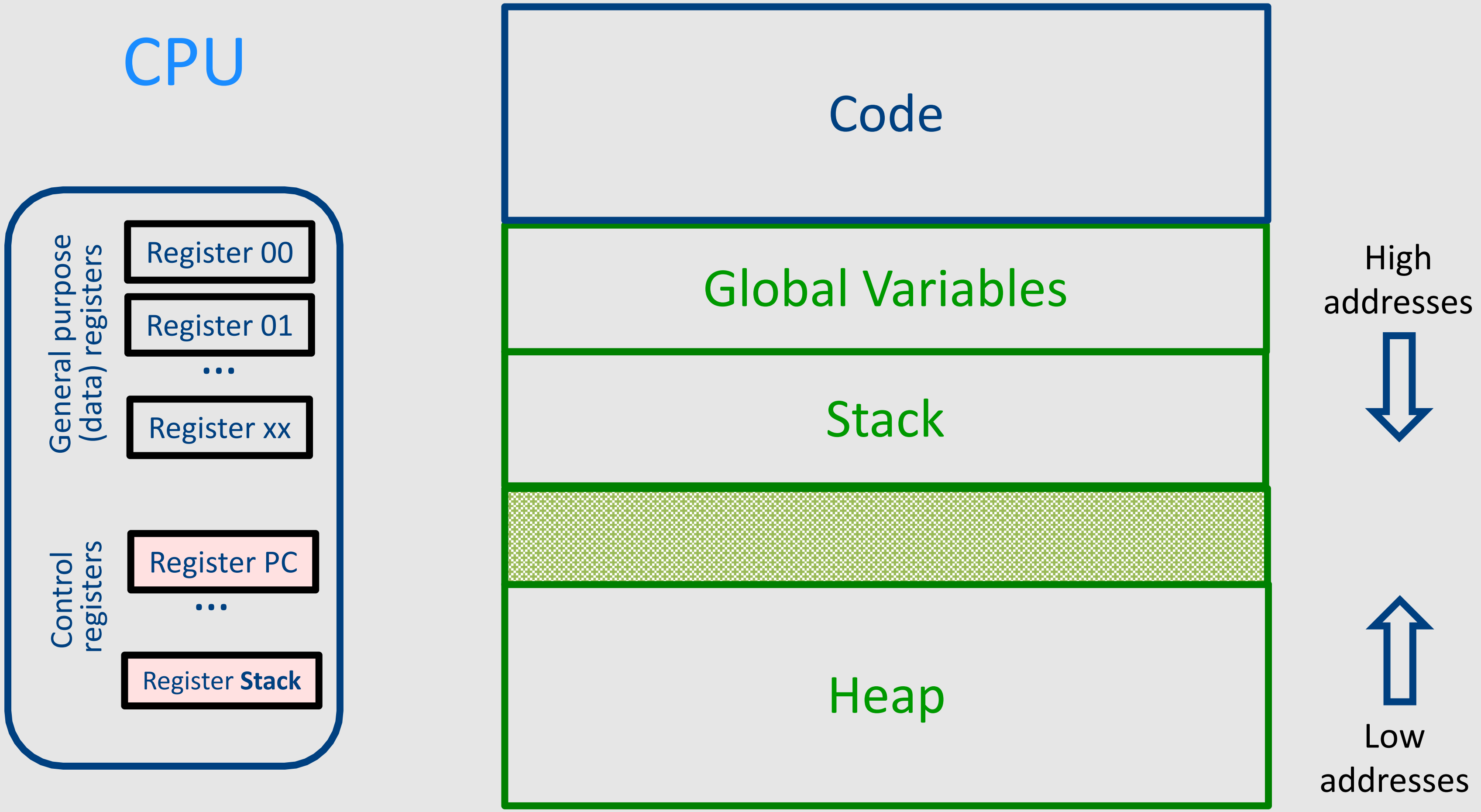
Abstract Register Machine

(High Level View)

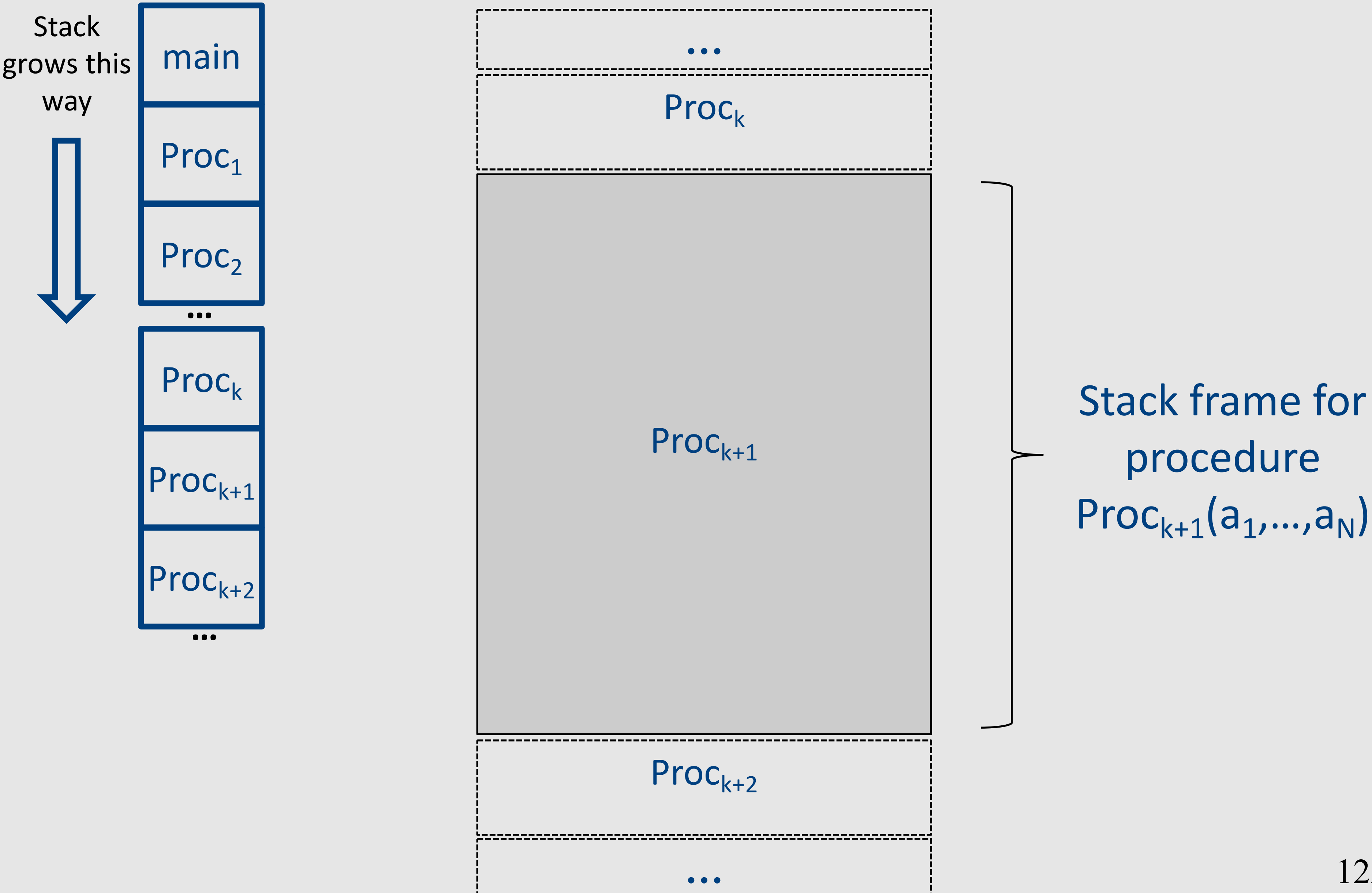


Abstract Register Machine

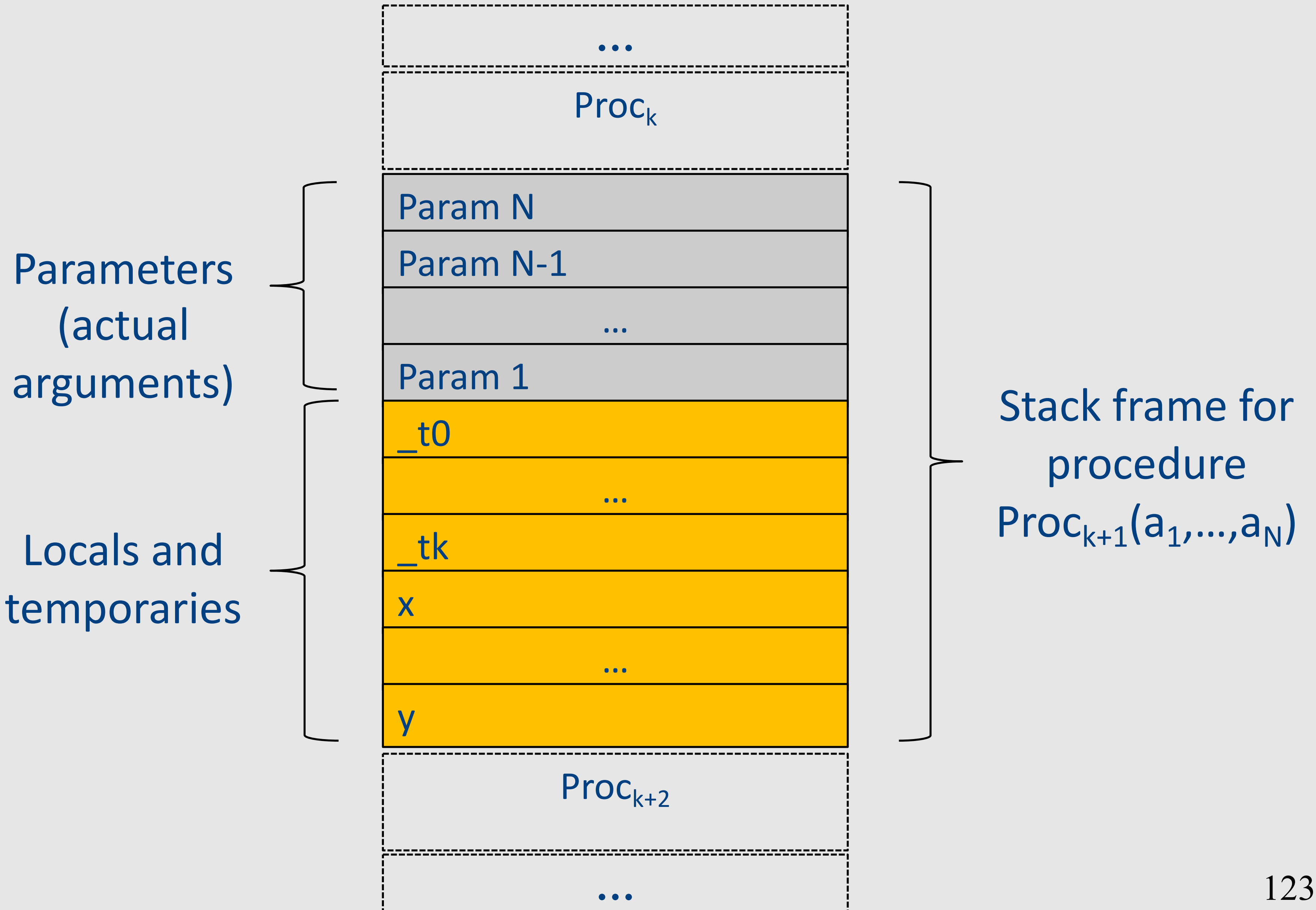
(High Level View)



Abstract Activation Record Stack



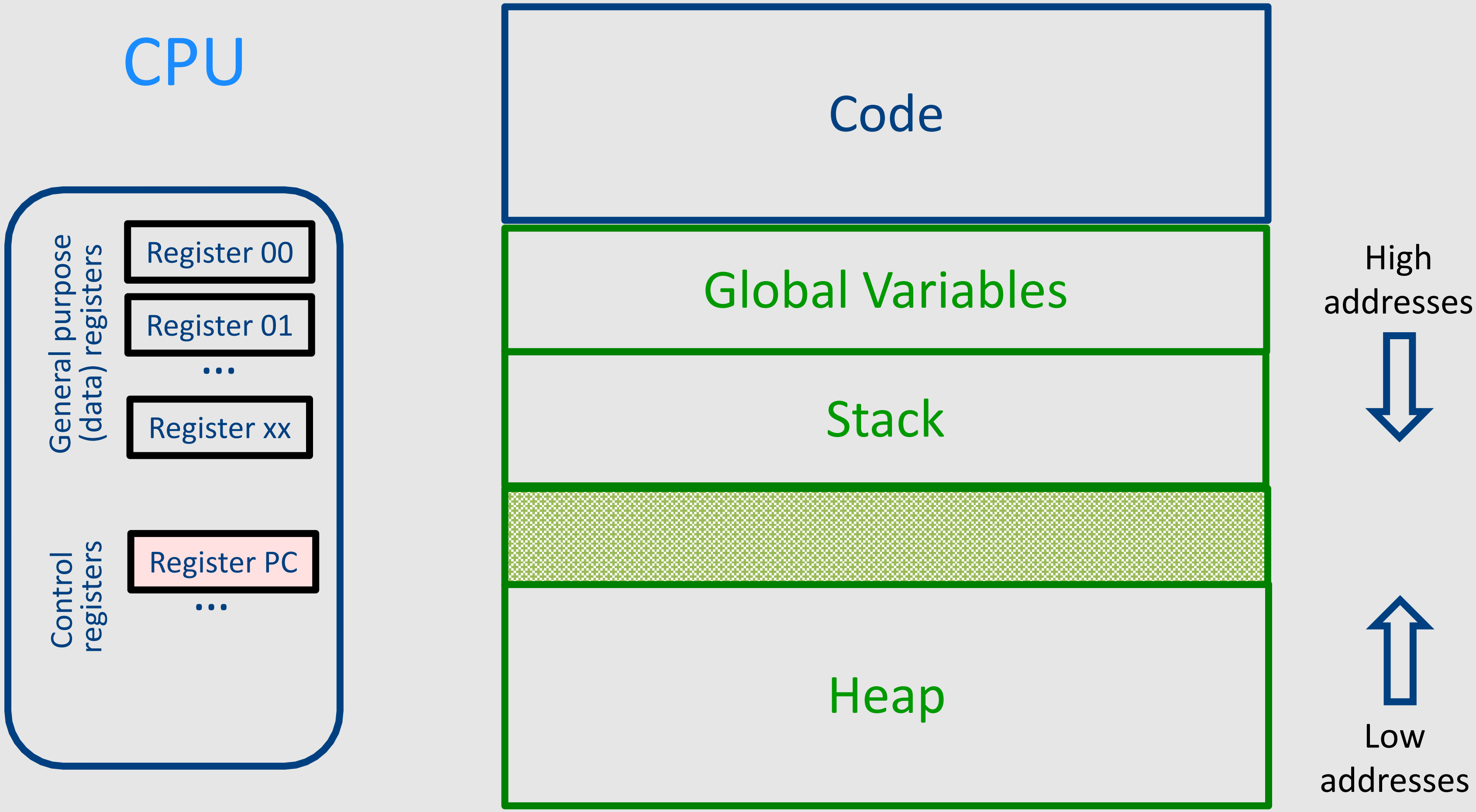
Abstract Stack Frame



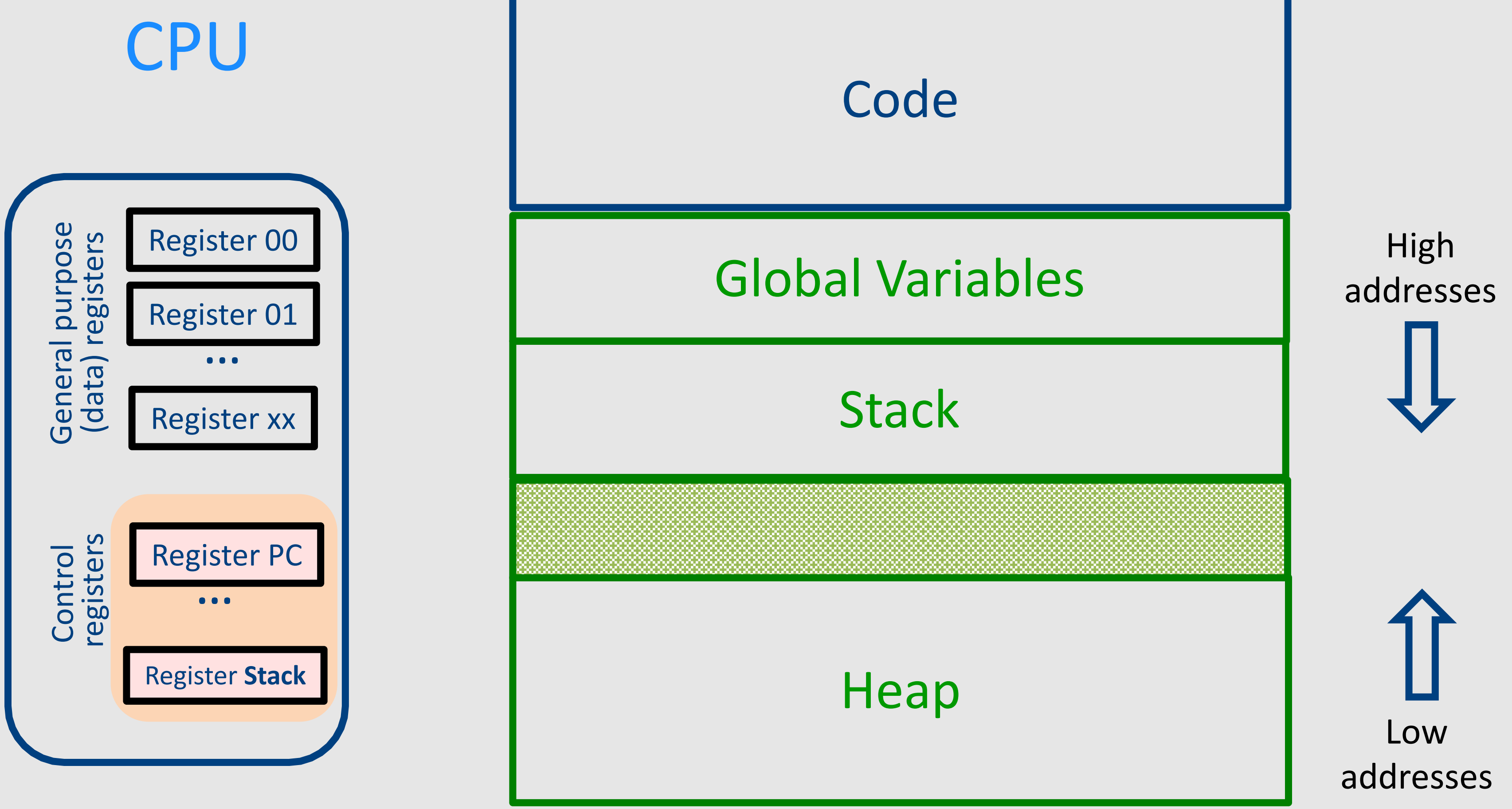
Handling Procedures

- 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) ;
}
```

B_0 B_1 B_2 B_3

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

Variable addresses for static scoping: first attempt

```
int a [11] ;

void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m, n);
    quicksort (m, i-1) ;
    quicksort (i+1, n) ;
  }

main() {
...
 quicksort (1, 9) ;
}
```

what is the address of the variable “i” in the procedure quicksort?

Compile-Time Information on Variables

- Name
- Type
- Scope
  - when is it recognized
- Duration
  - Until when does its value exist
- Size
  - How many bytes are required at runtime
- Address
  - Fixed
  - Relative
  - Dynamic

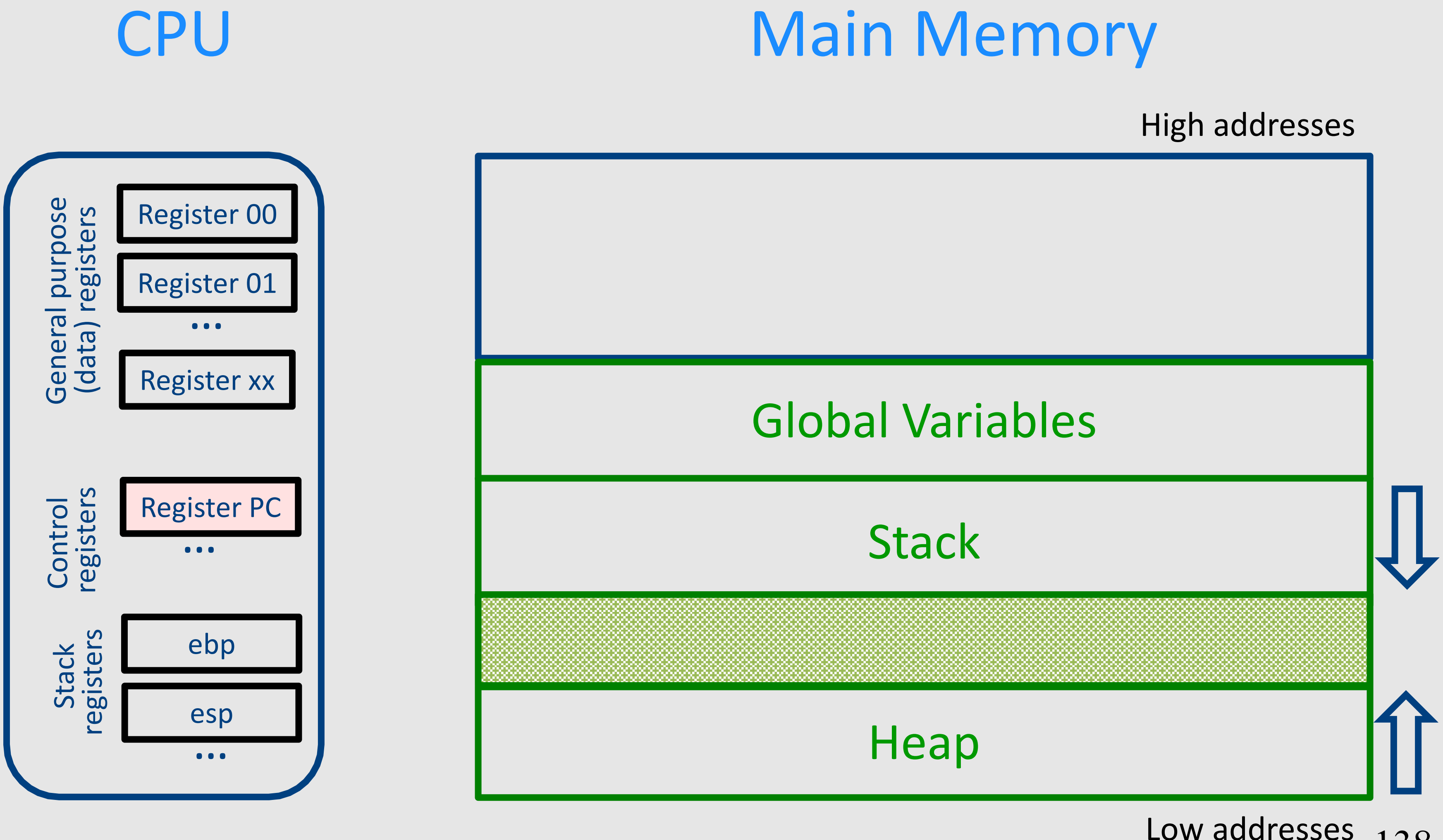
Activation Record (Stack Frames)

- separate space for each procedure invocation

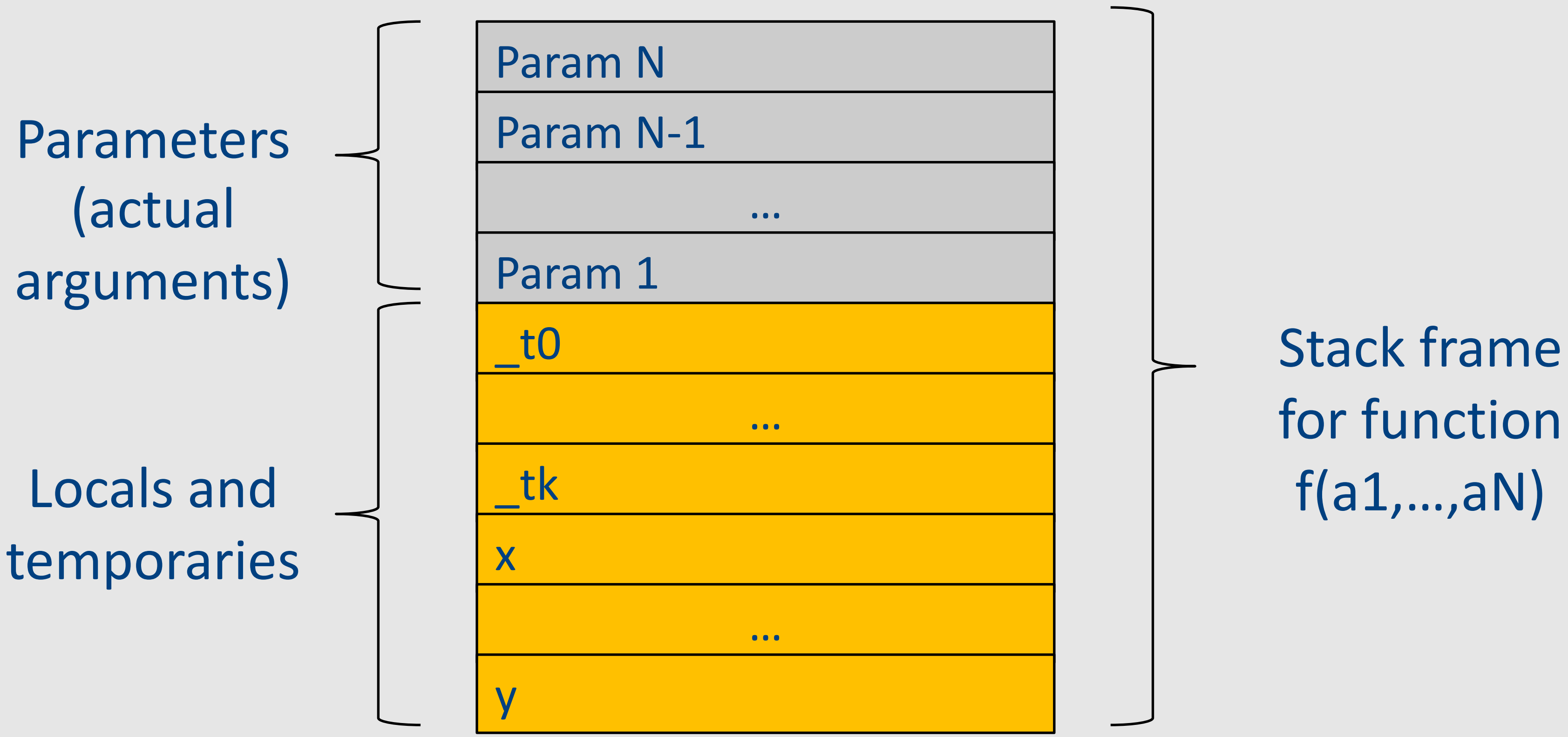
- **managed at runtime**
  - **code for managing it generated by the compiler**

- desired properties
  - efficient allocation and deallocation
    - procedures are called frequently
  - variable size
    - different procedures may require different memory sizes

Semi-Abstract Register Machine



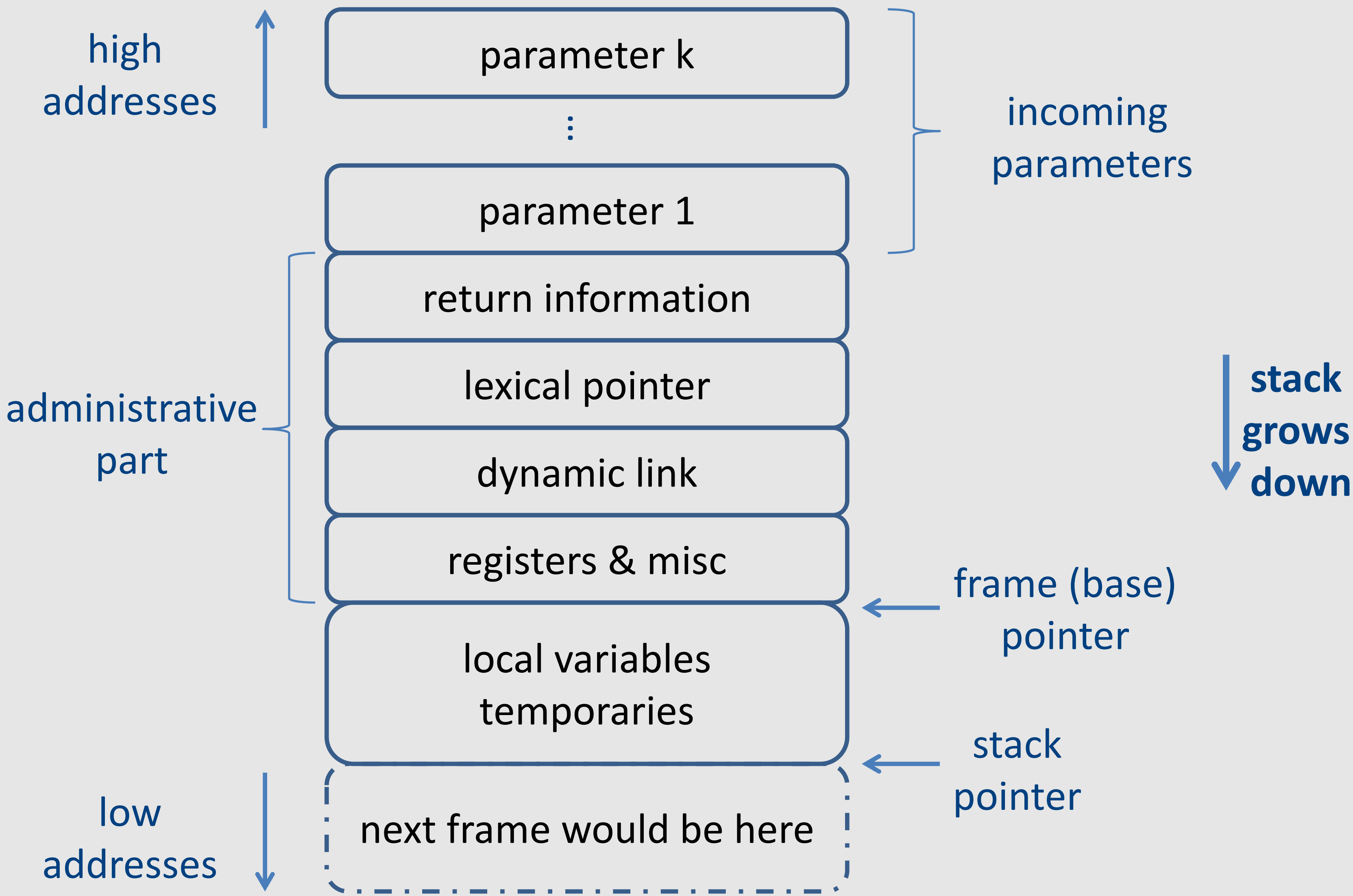
A Logical Stack Frame (Simplified)



Runtime Stack

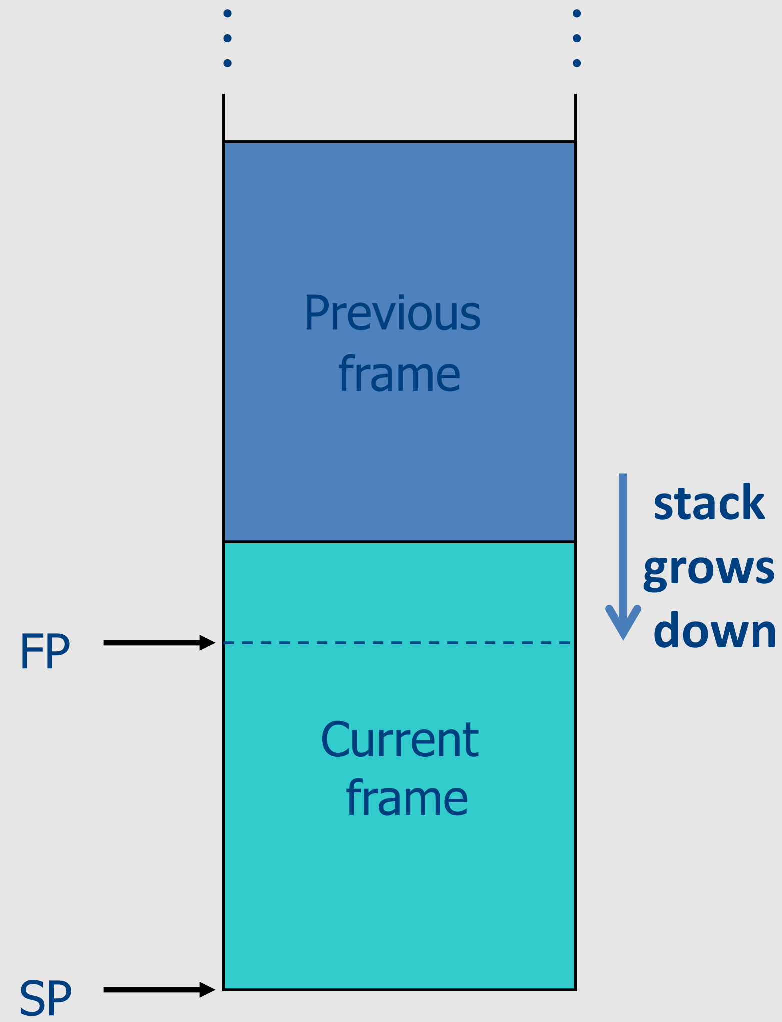
- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one “active” activation record – top of stack
- How do we handle recursion?

Activation Record (frame)



Runtime Stack

- SP – stack pointer
  – top of current frame
- FP – frame pointer
  – base of current frame
  - Sometimes called BP (base pointer)
  - Usually points to a “fixed” offset from the “start” of the frame



Code Blocks

- Programming language provide code blocks

```
void foo()
{
 int x = 8 ; y=9;//1
  { int x = y * y ;//2 }
  { int x = y * 7 ;//3}
      x = y + 1;
 }
```

adminstrative
x1
y1
x2
x3
…

L-Values of Local Variables

- The offset in the stack is known at compile time
- L-val(x) = FP+offset(x)
- x = 5 $\Rightarrow$ Load_Constant 5, R3
  Store R3, offset(x)(FP)

Pentium Runtime Stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

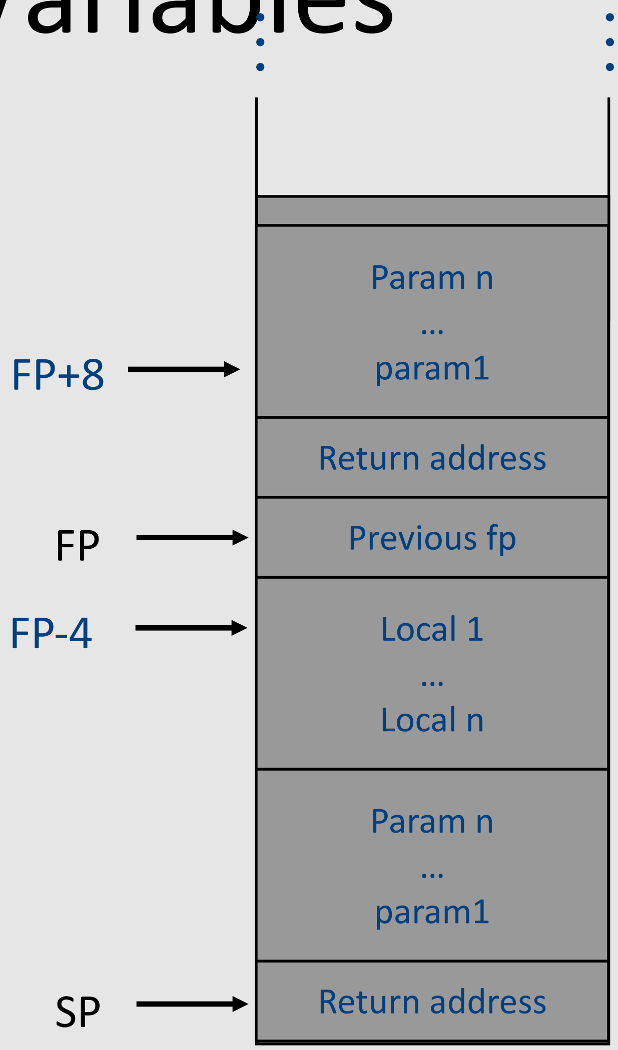
Pentium stack registers

Instruction	Usage
push, pusha,…	push on runtime stack
pop,popa,…	Base pointer
call	transfer control to called routine
return	transfer control back to caller

Pentium stack and call/ret instructions

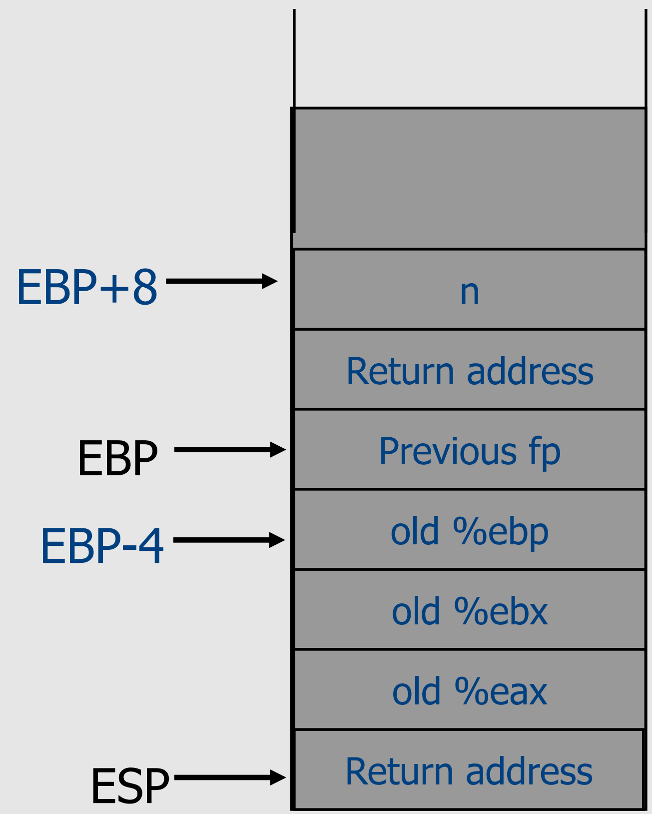
Accessing Stack Variables

- Use offset from FP (%ebp)
  - Remember: stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp – 4 = first local



Factorial – `fact(int n)`

```
fact:
pushl %ebp              # save ebp
movl %esp,%ebp          # ebp=esp
pushl %ebx              # save ebx
movl 8(%ebp),%ebx       # ebx = n
cmpl $1,%ebx            # n = 1 ?
jle .lresult            # then done
leal -1(%ebx),%eax      # eax = n-1
pushl %eax              #
call fact               # fact(n-1)
imull %ebx,%eax         # eax=retv*n
jmp .lreturn            #
.lresult:
movl $1,%eax            # retv
.lreturn:
movl -4(%ebp),%ebx      # restore ebx
movl %ebp,%esp          # restore esp
popl %ebp               # restore ebp
```



(stack in intermediate point)

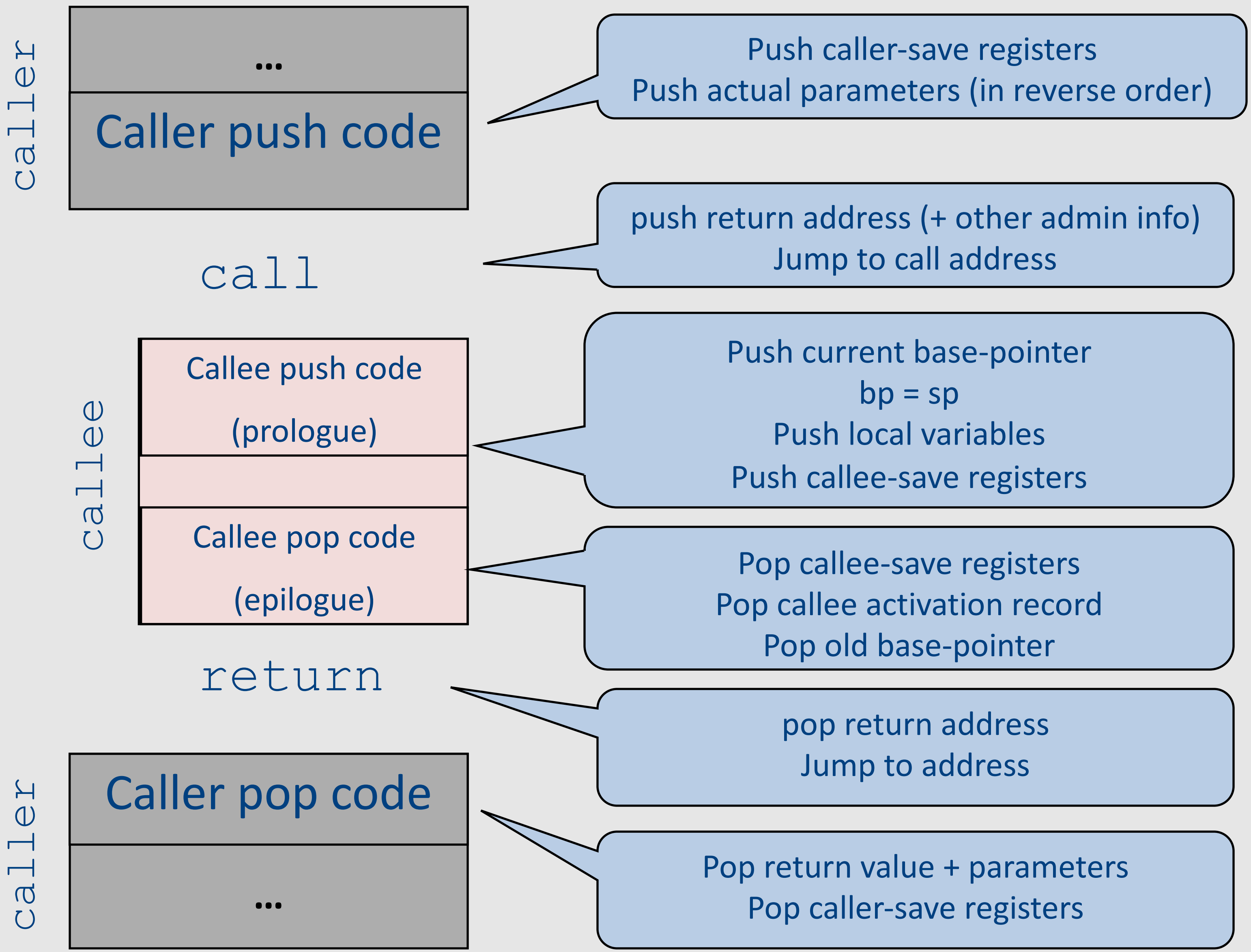
(disclaimer: real compiler can do better than that)

Call Sequences

- The **processor** **does not save** the content of **registers** on procedure calls

- So who will?
  - Caller saves and restores registers
  - Callee saves and restores registers
  - But can also have both save/restore some registers

Call Sequences



“To Callee-save or to Caller-save?”

- Callee-saved registers need only be saved when callee modifies their value
- Some heuristics and conventions are followed

Caller-Save and Callee-Save Registers

- Callee-Save Registers
  - Saved by the callee before modification
  - Values are automatically preserved across calls
- Caller-Save Registers
  - Saved (if needed) by the caller before calls
  - Values are not automatically preserved across calls
- Usually the architecture defines caller-save and callee-save registers

- Separate compilation
- Interoperability between code produced by different compilers/languages
- But compiler writers decide when to use caller/callee registers

Callee-Save Registers

- Saved by the callee before modification
- Usually at procedure prolog
- Restored at procedure epilog
- Hardware support may be available
- Values are automatically preserved across calls

```
int foo(int a)    {
        int b=a+1;
        f1();
        g1(b);
        return(b+2);
}
```

```
.global _foo
              Add_Constant -K, SP //allocate space for foo
              Store_Local  R5, -14(FP) // save R5
              Load_Reg  R5, R0; Add_Constant R5, 1
              JSR f1 ; JSR g1;
              Add_Constant R5, 2; Load_Reg R5, R0
             Load_Local -14(FP), R5 // restore R5
             Add_Constant K, SP; RTS // deallocate
```

Caller-Save Registers

- Saved by the caller before calls when needed
- Values are not automatically preserved across calls

```
void bar (int y) {
    int x=y+1;
    f2(x);
    g2(2);
    g2(8);
}
```

```
.global _bar
    Add_Constant -K, SP //allocate space for bar
    Add_Constant R0, 1
    JSR f2
    Load_Constant  2, R0  ;       JSR g2;
    Load_Constant 8, R0 ;         JSR g2
    Add_Constant K, SP // deallocate space for bar
    RTS
```

Parameter Passing

- 1960s
  - In memory
    - No recursion is allowed
- 1970s
  - In stack
- 1980s
  - In registers
  - First k parameters are passed in registers (k=4 or k=6)
  - Where is time saved?

- Most procedures are leaf procedures
- Interprocedural register allocation
- Many of the registers may be dead before another invocation
- Register windows are allocated in some architectures per call (e.g., sun Sparc)

Activation Records & Language Design

Compile-Time Information on Variables

- Name, type, size
- Address kind
  - Fixed (global)
  - Relative (local)
  - Dynamic (heap)
- Scope
  - when is it recognized
- Duration
  - Until when does its value exist

Scoping

```
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?

Nested Procedures

- For example – Pascal
- Any routine can have sub-routines
- Any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself
  - “non-local” variables

Example: Nested Procedures

```
program p(){
  int x;
  procedure a(){
    int y;
    procedure b(){ … c() … };
    procedure c(){
      int z;
      procedure d(){
        y := x + z
      };
      … b() … d() …
    }
    … a() … c() …
  }
  a()
}
```

Possible call sequence:
p → a → a → c → b → c → d

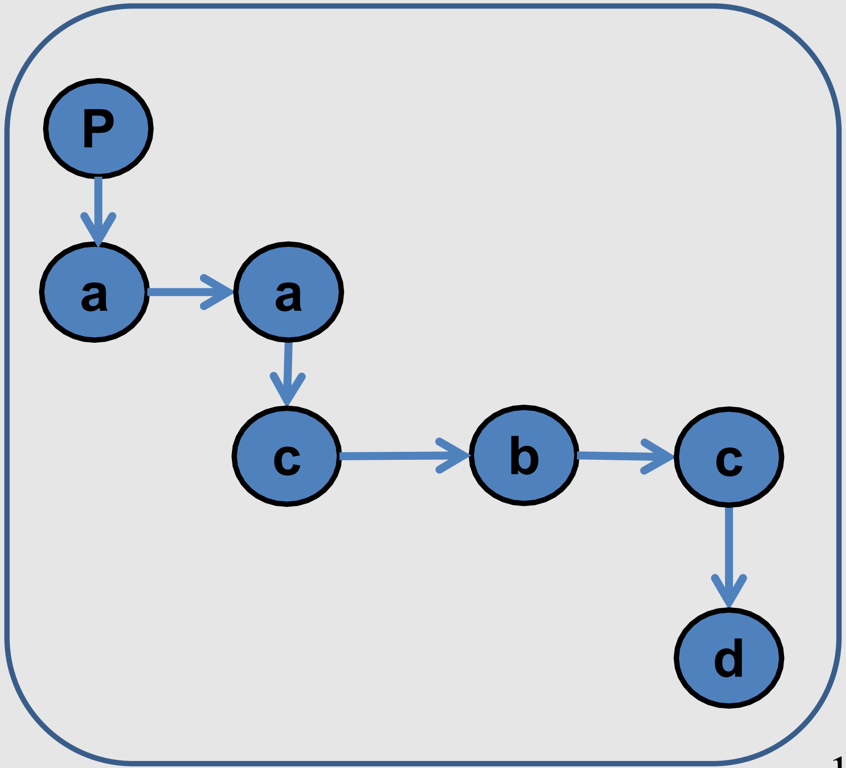
what are the addresses of variables "x," "y" and "z" in procedure d?

Nested Procedures

- **can call a sibling, ancestor**
- when “c” uses (non-local) variables from “a”, which instance of “a” is it?

- how do you find the right activation record at runtime?

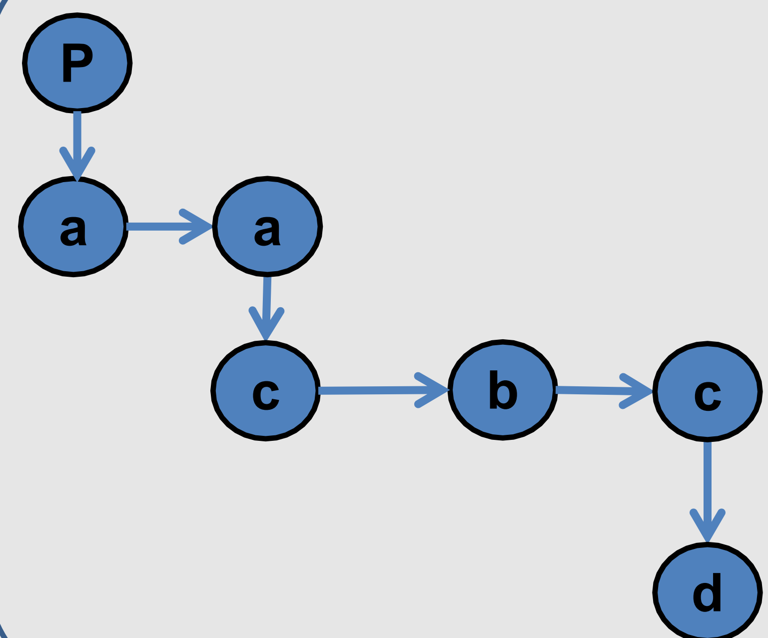
Possible call sequence:
p → a → a → c → b → c → d



Nested Procedures

- goal: **find the closest routine in the stack from a given nesting level**
- if we reached the same routine in a sequence of calls
  - routine of level k uses variables of the same nesting level, it uses its own variables
  - if it uses variables of nesting level j < k then it must be the last routine called at level j
- If a procedure is last at level j on the stack, then it must be ancestor of the current routine

Possible call sequence:
p → a → a → c → b → c → d



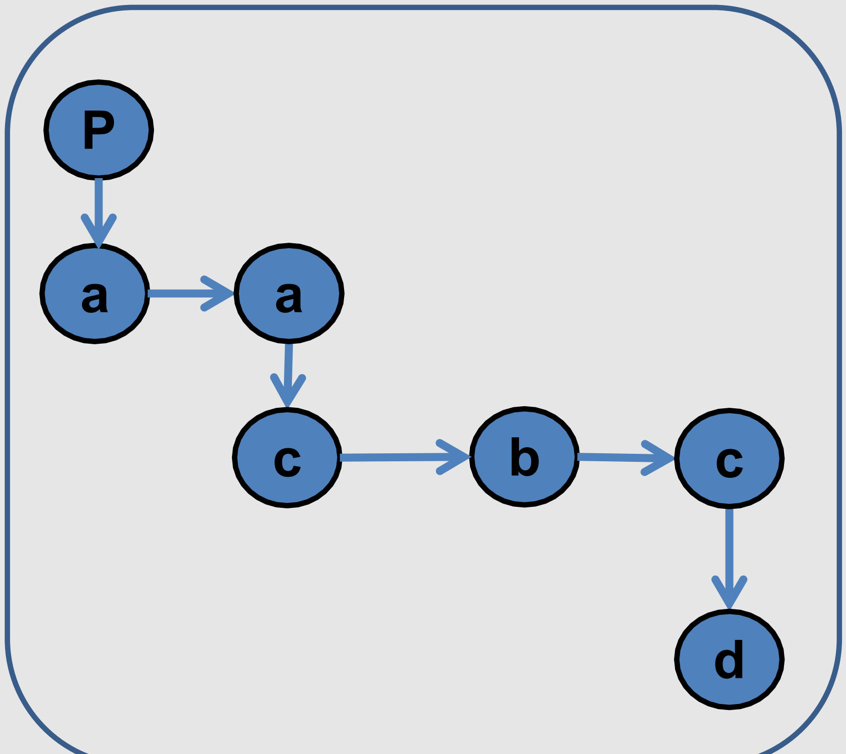
Nested Procedures

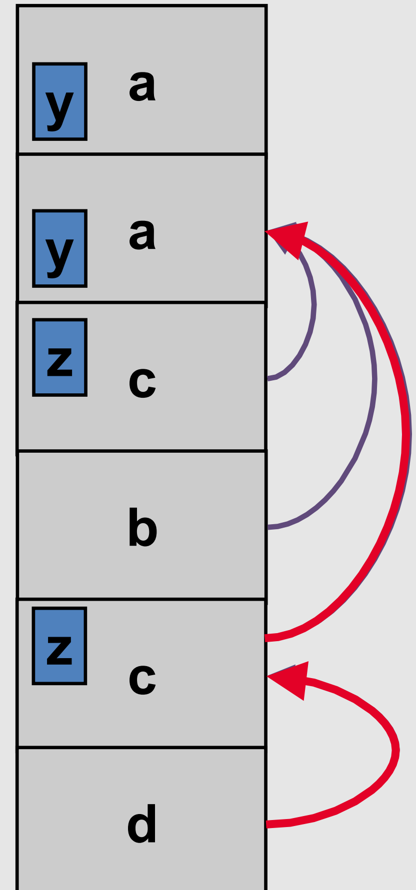
- problem: a routine may need to access variables of another routine that contains it statically
- solution: lexical pointer (a.k.a. access link) in the activation record
- lexical pointer points to the last activation record of the nesting level above it
  - in our example, lexical pointer of d points to activation records of c
- lexical pointers created at runtime
- number of links to be traversed is known at compile time

Lexical Pointers

```
program p(){
  int x;
  procedure a(){
    int y;
    procedure b(){ c() };
    procedure c(){
      int z;
      procedure d(){
        y := x + z
      };
      … b() … d() …
    }
    … a() … c() …
  }
  a()
}
```

Possible call sequence:
p → a → a → c → b → c → d

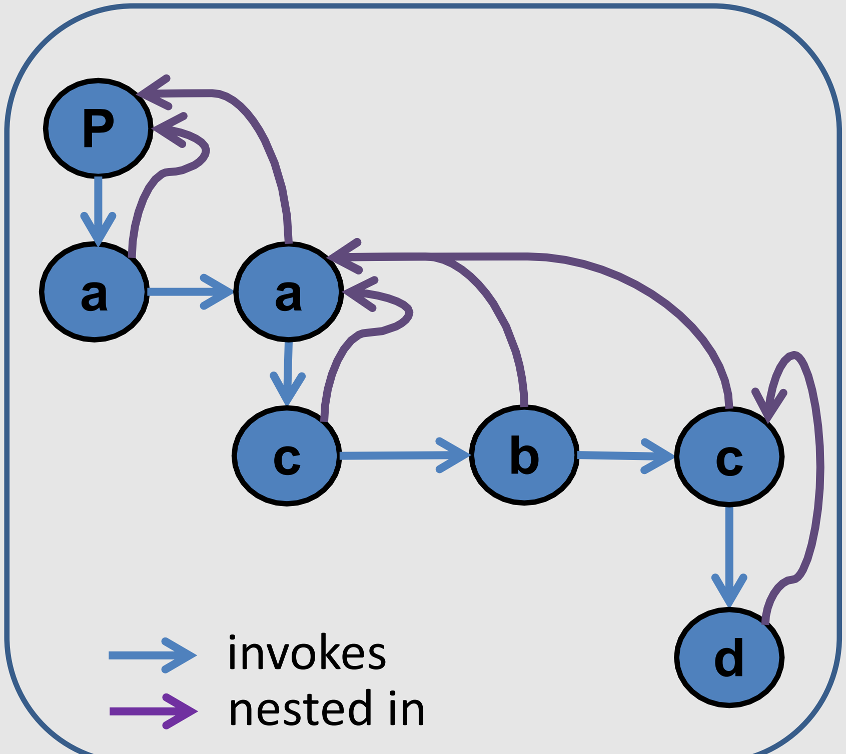


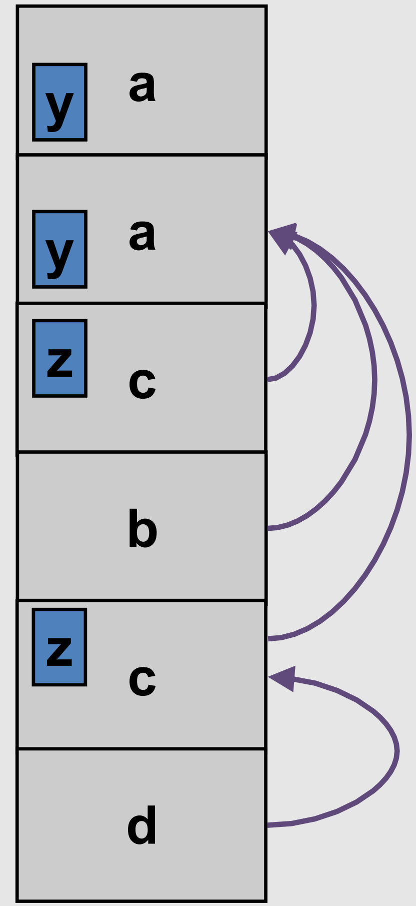


Lexical Pointers

```
program p(){
  int x;
  procedure a(){
    int y;
    procedure b(){ c() };
    procedure c(){
      int z;
      procedure d(){
        y := x + z
      };
      … b() … d() …
    }
    … a() … c() …
  }
  a()
}
```

Possible call sequence:
p → a → a → c → b → c → d





Activation Records: Remarks

Stack Frames

- Allocate a separate space for every procedure incarnation
- Relative addresses
- Provide a simple mean to achieve modularity
- Supports separate code generation of procedures
- Naturally supports recursion
- Efficient memory allocation policy
  - Low overhead
  - Hardware support may be available
- LIFO policy
- Not a pure stack
  - Non local references
  - Updated using arithmetic

Non-Local goto in C syntax

```
void level_0(void) {
    void level_1(void) {
        void level_2(void) {
            ...
            goto L_1;
            ...
        }
        ...
    L_1:...
        ...
    }
    ...
}
```

Non-local gotos in C

- setjmp remembers the current location and the stack frame
- longjmp jumps to the current location (popping many activation records)

Non-Local Transfer of Control in C

```c
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

Variable Length Frame Size

- C allows allocating objects of unbounded size in the stack

```
void p() {
   int i;
   char *p;
   scanf("%d", &i);
   p = (char *) alloca(i*sizeof(int));
 }
```

- Some versions of Pascal allows conformant array value parameters

Limitations

- The compiler may be forced to store a value on a stack instead of registers
- The stack may not suffice to handle some language features

Frame-Resident Variables

- A variable x cannot be stored in register when:
  - x is passed by reference
  - Address of x is taken (&x)
  - is addressed via pointer arithmetic on the stack-frame (C varags)
  - x is accessed from a nested procedure
  - The value is too big to fit into a single register
  - The variable is an array
  - The register of x is needed for other purposes
  - Too many local variables
- An escape variable:
  - Passed by reference
  - Address is taken
  - Addressed via pointer arithmetic on the stack-frame
  - Accessed from a nested procedure

The Frames in Different Architectures

g(x, y, z) where x escapes

	Pentium	MIPS	Sparc
x	InFrame(8)	InFrame(0)	InFrame(68)
y	InFrame(12)	InReg(X_{157})	InReg(X_{157})
z	InFrame(16)	InReg(X_{158})	InReg(X_{158})
View Change	M[sp+0]$\leftarrow$fp fp $\leftarrow$sp sp $\leftarrow$sp-K	sp $\leftarrow$sp-K M[sp+K+0] $\leftarrow r_2$ $X_{157} \leftarrow$ r4 $X_{158} \leftarrow$ r5	save %sp, -K, %sp M[fp+68]$\leftarrow i_0$ $X_{157} \leftarrow i_1$ $X_{158} \leftarrow i_2$

Limitations of Stack Frames

- A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P
- Example 1: Static variables in C
  (own variables in Algol)

```
void p(int x)
{
   static int y = 6 ;
   y += x;
 }
```

- Example 2: Features of the C language

```
int * f()
{ int x ;
   return &x ;
}
```

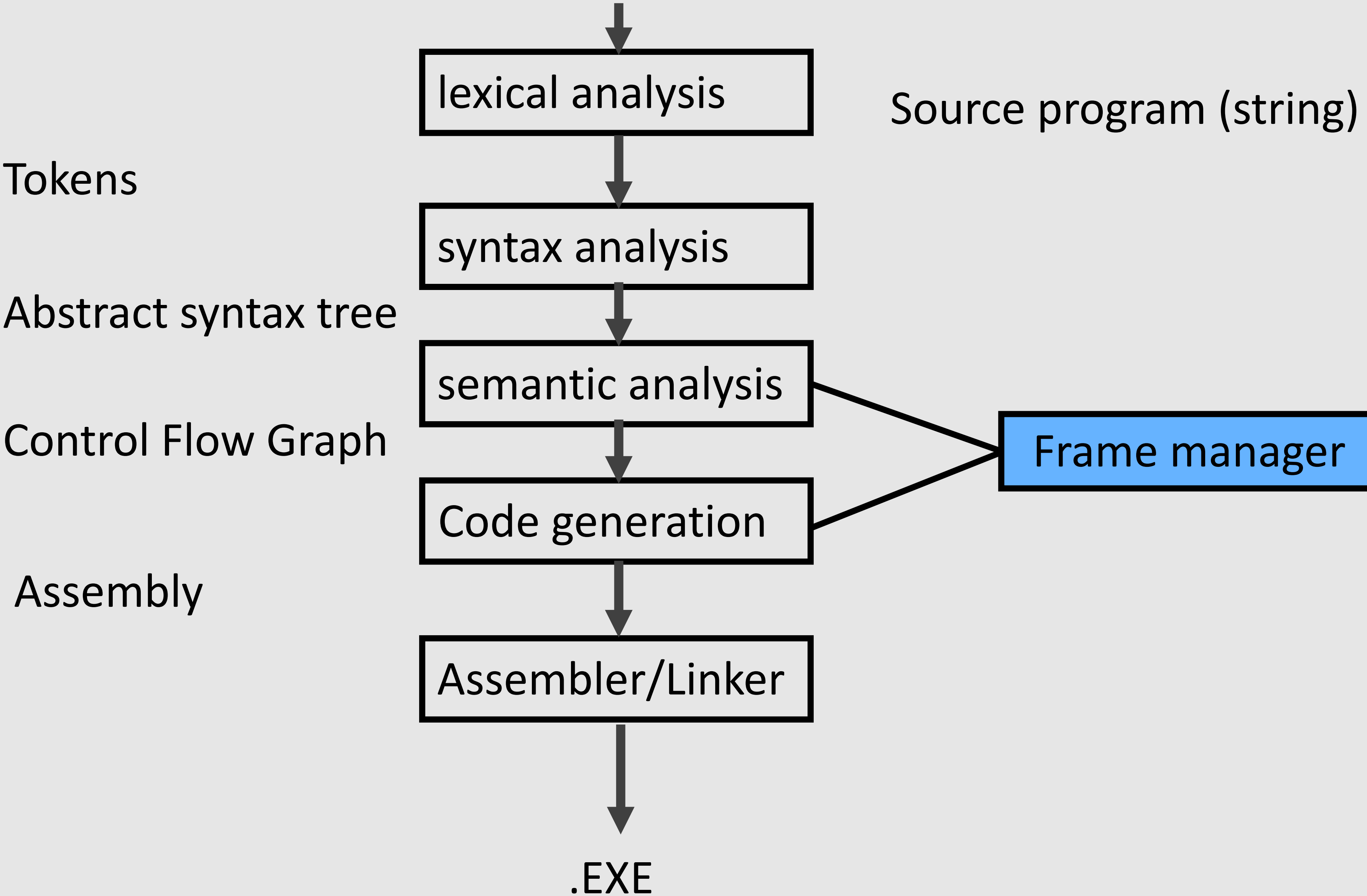
- Example 3: Dynamic allocation

```
int * f()  { return (int *)
malloc(sizeof(int)); }
```

Compiler Implementation

- Hide machine dependent parts
- Hide language dependent part
- Use special modules

Basic Compiler Phases



Hidden in the frame ADT

- Word size
- The location of the formals
- Frame resident variables
- Machine instructions to implement “shift-of-view” (prologue/epilogue)
- The number of locals “allocated” so far
- The label in which the machine code starts

Activation Records: Summary

- compile time memory management for procedure data
- works well for data with well-scoped lifetime
  - deallocation when procedure returns