Compilation

0368-3133 (Semester A, 2013/14)

Lecture 7: Intermediate Representation
(Target Architecture Agnostic Code Generation)

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

Admin

- Mobiles …

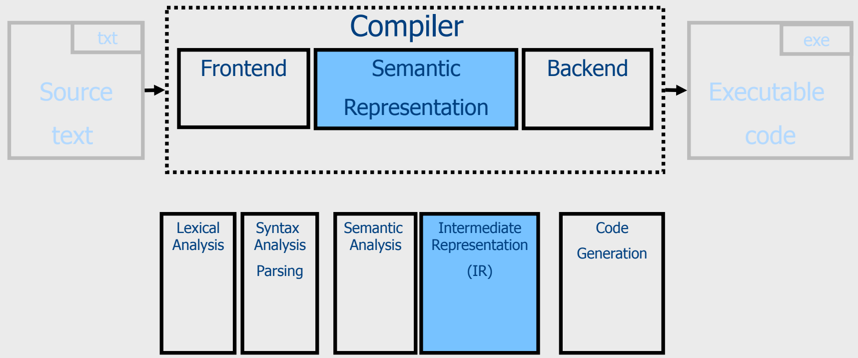
What is a Compiler?

"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

Conceptual Structure of a Compiler



From scanning to parsing

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

Lexical Analyzer

token stream

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

Grammar:
E → … | Id
Id → ‘a’ | … | ‘z’

Parser

syntax error

valid

Op(*)
Op(+) Id(b)
Num(23) Num(7)

Abstract Syntax Tree

Context Analysis

Type rules

E1 : int E2 : int
―――――――――――
E1 + E2 : int

Op(*)
Op(+) Id(b)
Num(23) Num(7)

Abstract Syntax Tree

Semantic Error

Valid + Symbol Table

Code Generation

…

Op(*)
Op(+) Id(b)
Num(23) Num(7)

Valid Abstract Syntax Tree
Symbol Table

Verification (possible runtime)
Errors/Warnings

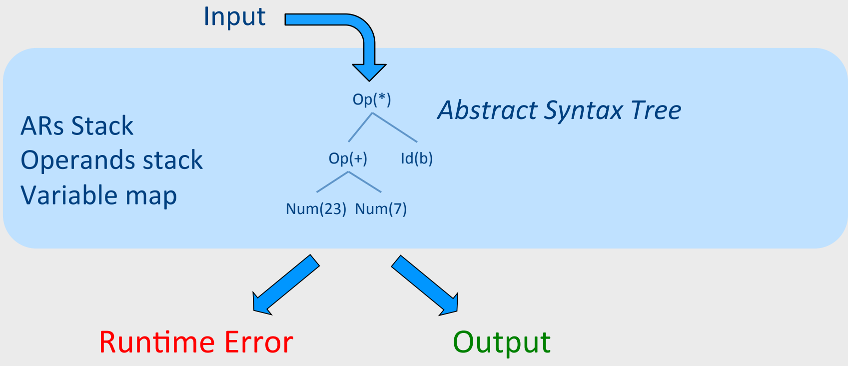
Intermediate Representation (IR)

input → Executable Code → output

Compile Time vs Runtime

- Compile time: Data structures used during program compilation
- Runtime: Data structures used during program execution

[Interpretation]



[Interpretation]

"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**."

- The frontend generates the AST from source
- The interpreter "executes" the AST
  - Recursive interpreter
  - Iterative interpreter
- Are we done?

[Types of Interpreters]

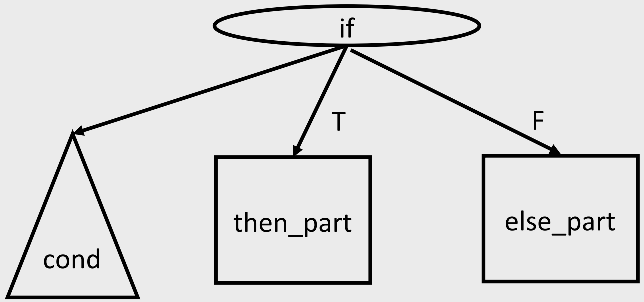
- Recursive
  - Recursively traverse the tree
  - Uniform data representation
  - Conceptually clean
  - Excellent error detection
  - **1000x slower than executing compiled code**
- Iterative (Threaded AST)
  - Closer to CPU
  - One flat loop
  - Explicit stack
  - Good error detection
  - Can invoke compiler on code fragments
  - **30x slower than executing compiled code**

[Interpreters: What did we learn?]

- "compilation"
  - Lexer; parser
- "Executable code"
  - AST
- Runtime environment + execution
  - States (memory)
    - Operand stack (for expression evaluation)
    - Variable map (left + right values)
    - Activation Records (functions)
  - Interpretation
    - Expressions (e.g., `x + 4`)
    - Assignments (e.g., `x := a + 4`)
    - Control (e.g., `if (0 < x) then x := a + 4 ; z := x`)
    - Procedure invocation + parameter passign (e.g., `f(3)`)

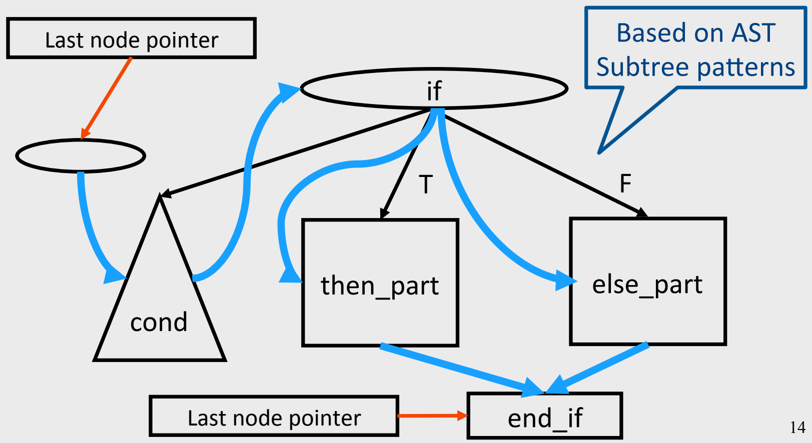
[Interpreters: What did we learn?]

Creating “executable” code from AST



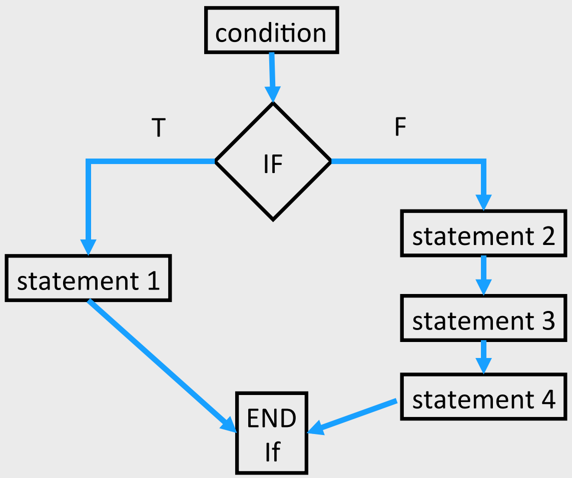
[Interpreters: What did we learn?]

Creating “executable” code from AST



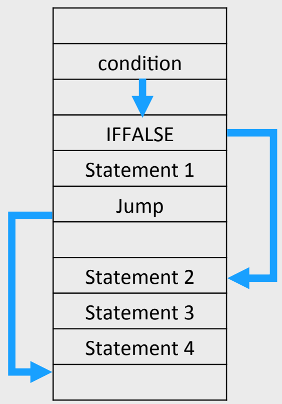
[Interpreters: What did we learn?]

Code representation



[Interpreters: What did we learn?]

Code representation: Threaded AST as Array of Instructions



[Interpreters: What did we learn?]

Optimization: Tail Call Elimination

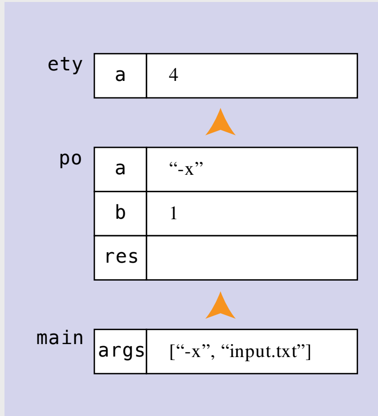
```
void a(…)
{
  …
  b();
}

void b(){
  code;
}
```

→

```
void a(…)
{
  …
  code;
}

void b(){
  code;
}
```


[Interpreters: What did we learn?]

State (Runtime) environment

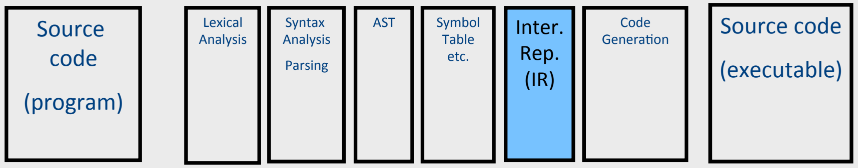


Compilers

- Code generation
- Optimization
- State (runtime) layout + management
- Evaluation

"we'll be back"

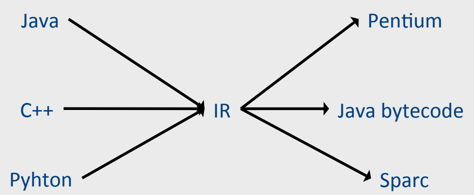
Code Generation: IR



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

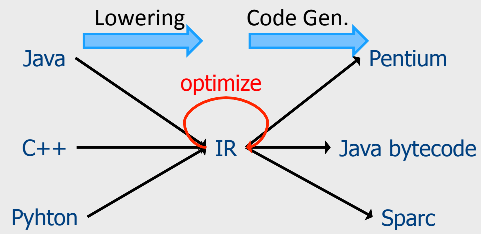
Intermediate representation

- 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



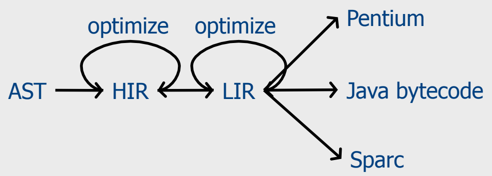
Intermediate representation

- 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 node types (instructions)



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

- Rich set of language constructs
- Rich type system
- Declarations: types (classes, interfaces), functions, variables
- Control flow statements: if-then-else, while-do, break-continue, switch, exceptions
- Data statements: assignments, array access, field access
- Expressions: variables, constants, arithmetic operators, logical operators, function calls

LIR

- An abstract machine language
- Very limited type system
- Only computation-related code
- Labels and conditional/ unconditional jumps, no looping
- Data movements, generic memory access statements
- No sub-expressions, logical as numeric, temporaries, constants, function calls – explicit argument passing

Sub-expressions example

Source

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

LIR (unoptimized)

Where have the declarations gone?

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


Sub-expressions example

Source

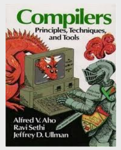
```
int a;
int b;
int c;
int d;
a = b + c + d;
b = a * a + b * b;
```

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

Three-Address Code IR



Chapter 8

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

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 **+, -, *, /, %**

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 1, 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

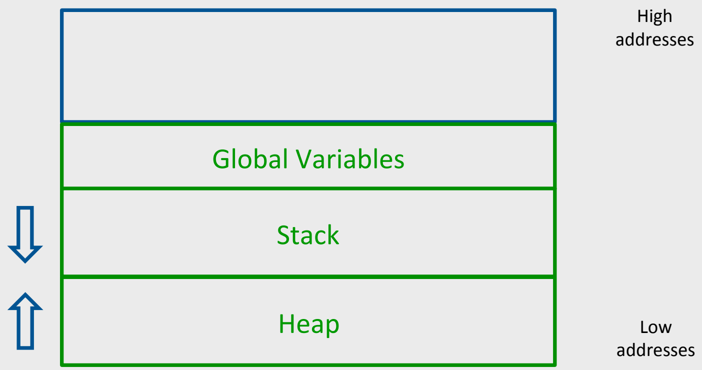
```
int x;
int y;

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

y = x;
```

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

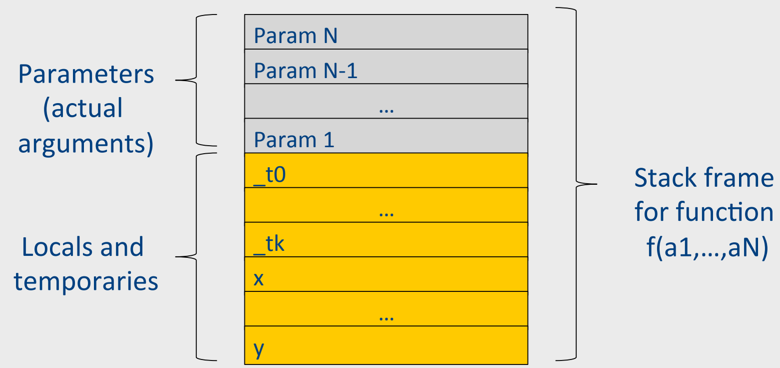

Procedures / Functions

- 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;
  ```


Memory Layout
(popular convention)



A logical stack frame



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)

Lowering AST to TAC

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

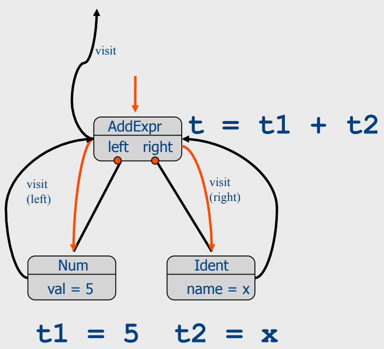
cgen(5 + x) = {
  Choose a new temporary *t*
  Let t_1 = {
    Choose a new temporary *t*
    Emit(*t = 5;*)
    Return *t*
  }
  Let t_2 = {
    Choose a new temporary *t*
    Emit(*t = x;*)
    Return *t*
  }
  Emit($t = t_1 + t_2;$)
  Return *t*
}

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

Inefficient translation, but we will improve this later

cgen as recursive AST traversal

cgen(5 + x)




```
t1 = 5;

t2 = x;

t = t1 + t2;
```


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

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

Code

```
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
}
```


Example

Code
```
_t0=a;
```

```
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
}
```


Example

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

```
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
}
```

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}:)

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 = 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
```

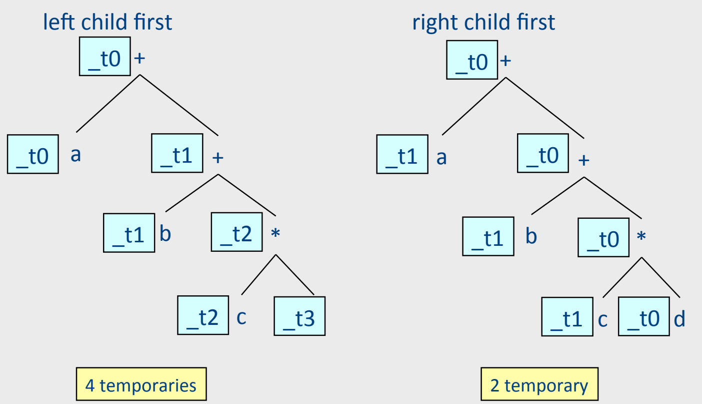

Weighted register allocation

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

*+ and * are commutative operators*



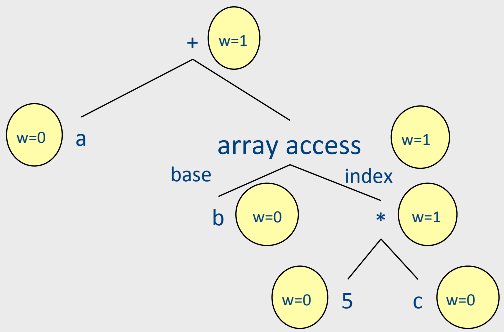
Weighted register allocation

- 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])

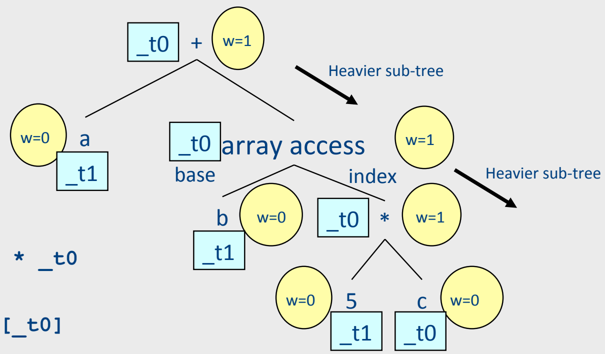
Phase 1: - check absence of side-effects in expression tree
- assign weight to each AST node



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;
```


Once Again

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
  }

Improved cgen for expressions

- Maintain temporaries stack by counter c
- Initially: c = 0
- cgen(e1 op e2) = {

  Let _tc = cgen(e1)
  c = c + 1
  Let _tc = cgen(e2)
  c = c - 1
  Emit(_tc = _tc op _tc+1;)
  Return tc
  }

Example

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

Code

Example

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

```
Code
_t0=a;
```


Example

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

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


Example

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

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

Example

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

c=0 (arrow at "Return _tc" of the inner block)

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


Example

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

c=1 (arrow at "Let _tc = { Emit(_tc = d;), return _tc }")

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


Example

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

c=1 (arrow at "c = c - 1")

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


Example

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

c=0 (arrow at "Emit(_tc = _tc - _tc+1;)")

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

Example

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

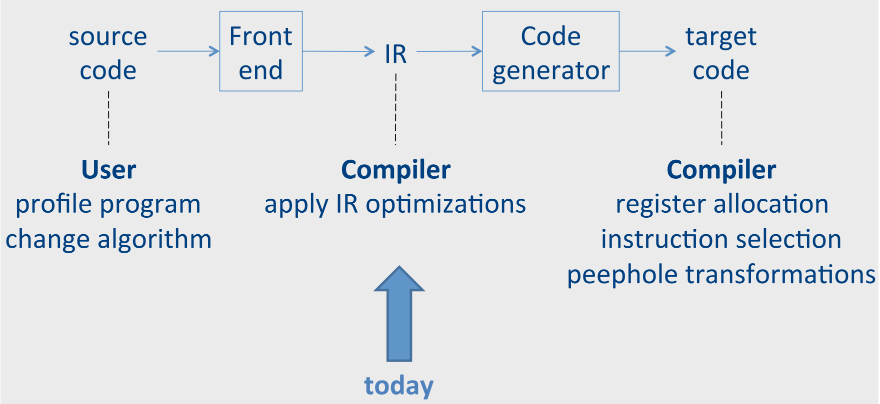
c=0

```
Code
_t0=a;
_t1=b;
_t0=_t0*_t1;
_t1=d;
_t0=_t0-_t1;
```


Weighted register allocation for trees

- Sethi-Ullman's algorithm generates code for side-effect-free expressions yields minimal number of registers
- Phase 0: check side-effect-free condition
- Phase 1: Assign weights (weight = number of registers needed)
  - Leaf weight known (usually 0 or 1)
  - 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
- Phase 2: translate heavier child first
  - Can be done by rewriting the expression such that heavier expressions appear first and then using improved **cgen**

Optimization points



The End