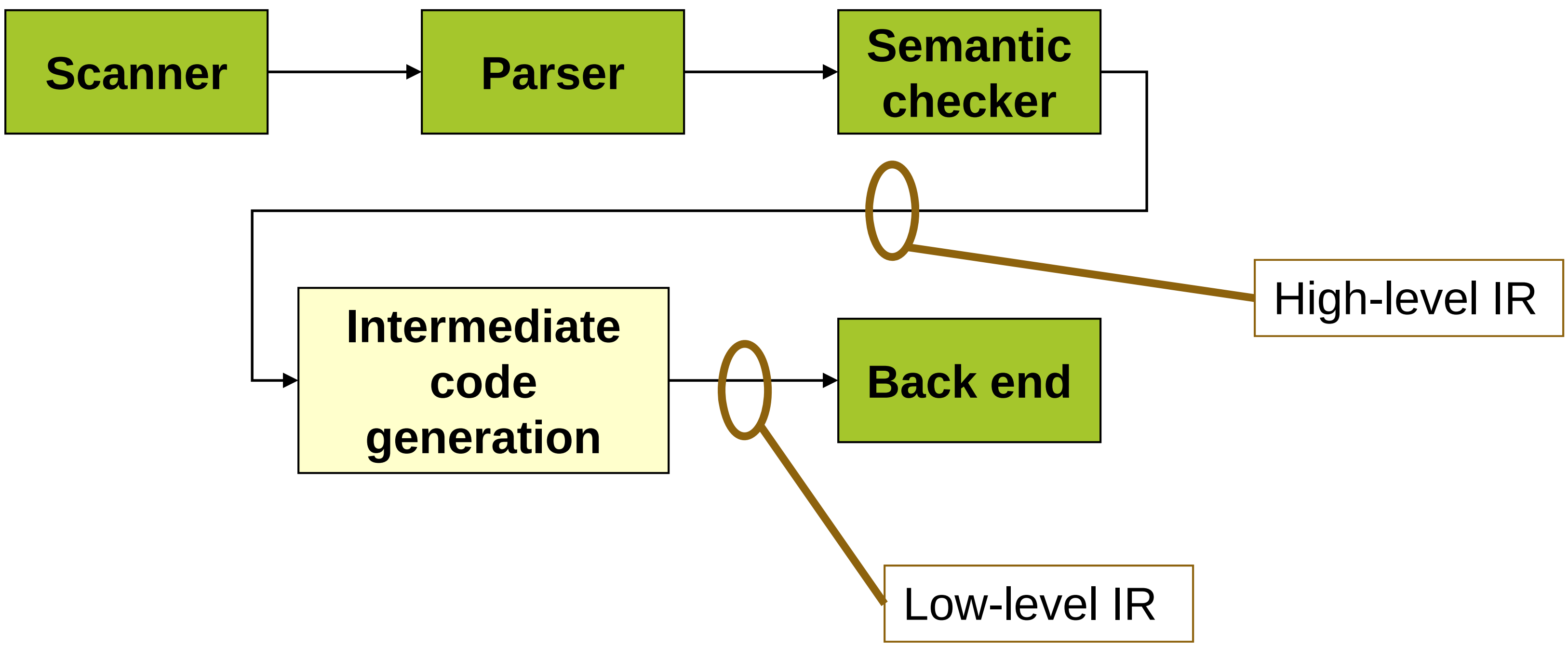
Compilers

Intermediate representations and code generation

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Today

- Intermediate representations and code generation



Intermediate representations

- IR design affects compiler speed and capabilities
- Some important *IR* properties
  - Ease of generation, manipulation, optimization
  - Size of the representation
  - Level of ***abstraction***: level of "detail" in the IR
    - How close is IR to source code? To the machine?
    - What kinds of operations are represented?
- Often, different IRs for different jobs

  Typically:
  - High-level IR: close to the source language
  - Low-level IR: close to the machine assembly code

Types of IRs

Three major categories

- Structural
  - Graph oriented
  - Heavily used in IDEs, language translators
  - Tend to be large

Examples:
Trees, DAGs

- Linear
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

Examples:
3 address code
Stack machine code

- Hybrid
  - Combination of graphs and linear code

Example:
Control-flow graph

High-level IR

- High-level language constructs
  - Array accesses, field accesses
  - Complex control flow

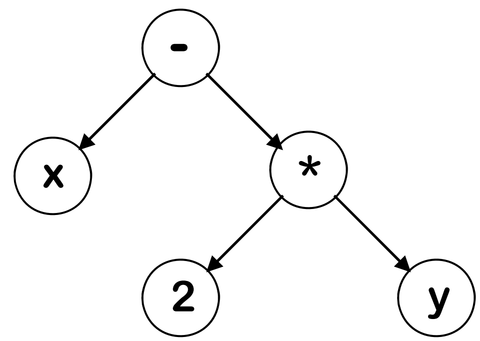
    Loops, conditionals, switch, break, continue
  - Procedures: callers and callees
  - Arithmetic and logic operators

    Including things like short-circuit && and ||
- Often: tree structured
  - Arbitrary nesting of expressions and statements

Abstract Syntax Tree

- AST: parse tree with some intermediate nodes removed

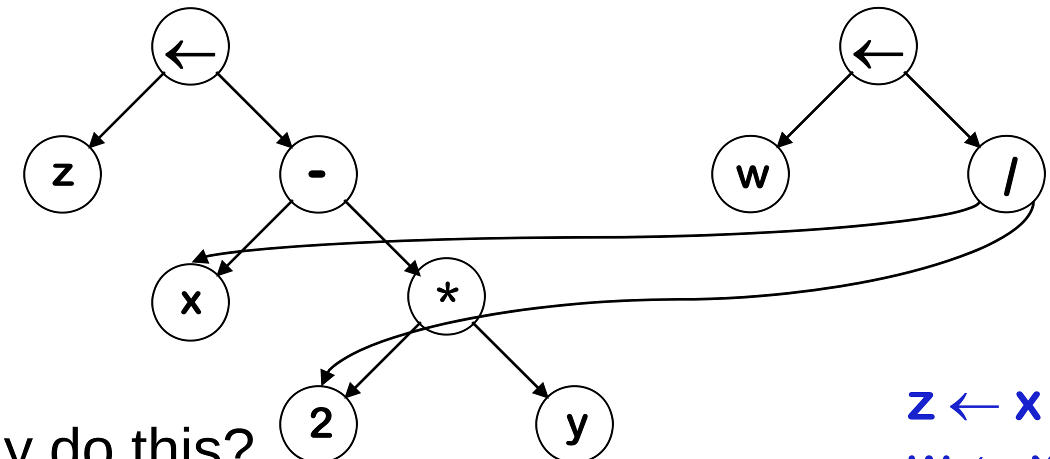
```
x – 2 * y
```



- What is this representation good for?
  - Interpreters
  - We can reconstruct original source
  - Program understanding tools
  - Language translators

Directed Acyclic Graph

- A directed acyclic graph (DAG)
  - AST with a unique node for each value



- Why do this?
  - More compact (sharing)
  - Encodes redundancy

$z \leftarrow x - 2 * y$
$w \leftarrow x / 2$

Same expression twice means that the compiler might arrange to evaluate it just once!

Low-level IR

- Linear stream of ***abstract instructions***
- Instruction: single operation and assignment

```
x = y op z
```

```
x ← y op z
```

```
op x, y, z
```

- Must break down high-level constructs
  - Example:

```
z = x – 2 * y
```

```
t ← 2 * y
z ← x - t
```

  - Introduce temps as necessary: called ***virtual registers***
- Simple control-flow
  - Label and goto

```
label1:
goto label1
if_goto x, label1
```

Jump to label1 if x has non-zero value

Stack machines

Now, Java VM

- Originally for stack-based computers

```
x – 2 * y
```

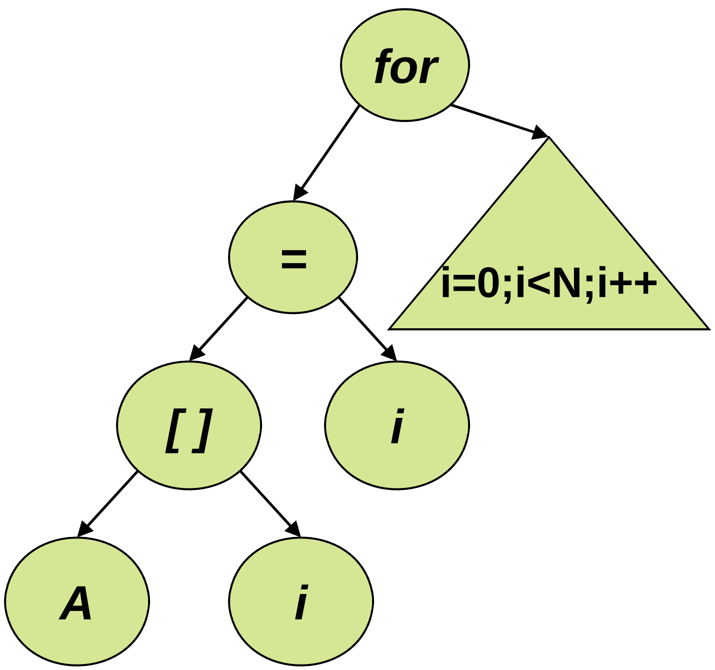
```
push x
push 2
push y
multiply
subtract
```

Post-fix notation

- What are advantages?
  - Introduced names are *implicit*, not *explicit*
  - Simple to generate and execute code
  - Compact form – who cares about code size?
    - Embedded systems
    - Systems where code is transmitted (the 'Net)

IR Trade-offs

```
for (i=0; i<N; i++)
  A[i] = i;
```



Loop invariant

Strength reduce to temp2 += 4

```
loop:
temp1 = &A
temp2 = i * 4
temp3 = temp1 + temp2
store [temp3] = i
...
goto loop
```

IR Summary

- Intermediate representations
  - High-level
    - Rich structure, close to source
    - Good for source-level tools, IDEs
    - **Example**: abstract syntax tree
  - Low-level
    - Linear sequence, close to the machine
    - Good for optimization
    - **Example**: abstract machine code, bytecode
- Essence of compilation:

  Translating from the high-level IR to low-level IR

Towards code generation

```
if (c == 0) {
  while (c < 20) {
    c = c + 2;
  }
}
else
  c = n * n + 2;
```

```
t1 = c == 0
if_goto t1, lab1
t2 = n * n
c = t2 + 2
goto end
lab1:
t3 = c >= 20
if_goto t3, end
c = c + 2
goto lab1
end:
```

Code generation

- Convert from high-level IR to low-level IR
  - HIR is complex, with nested structures
  - LIR is low-level, with *everything* explicit
  - Often called ***lowering*** or ***linearizing*** the code
  - Result is a sequence of LIR instructions
  - Need a systematic algorithm
- **Idea**:
  - Define translation for each AST node, *assuming* we can get code to implement children
  - Come up with a scheme to stitch them together
  - Recursively descend the AST

Lowering scheme

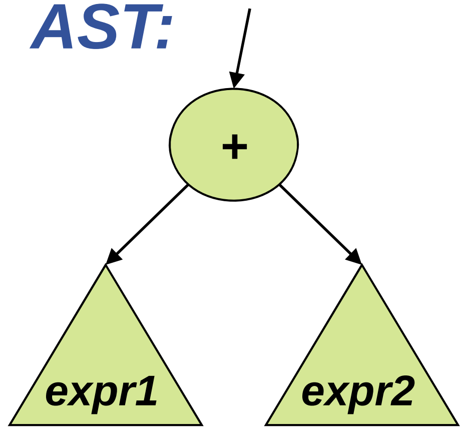
- General scheme
  - Code "template" for each AST node
    - Captures key semantics of each construct
    - Has "holes" for the children of the node
    - Implemented in a function called ***generate***
  - To fill in the template:
    - Call generate function recursively on children
    - Plug code into the holes
- How to stitch code together?
  - Generate returns a temporary that holds the result
  - Emit code that combines the results

Example: add expression

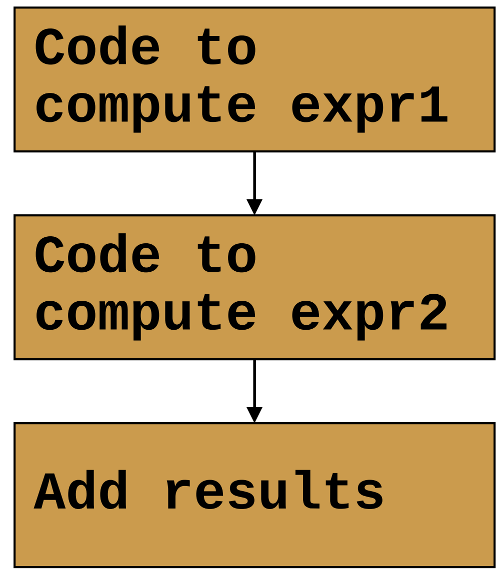
Source:

```
expr1 + expr2
```

AST:



Flow chart:



Code template:

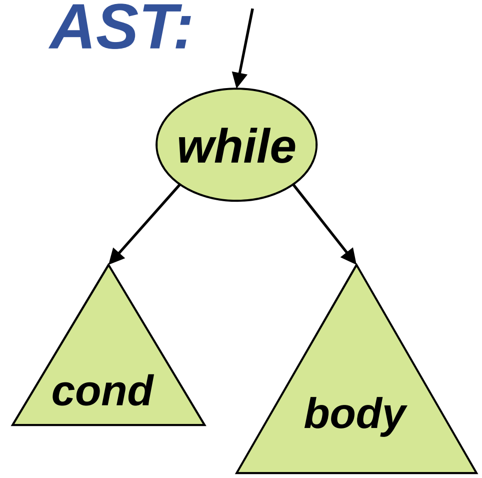
```
   <code for expr1>
   <code for expr2>
<result> = <expr1> + <expr2>
```

Example: while loop

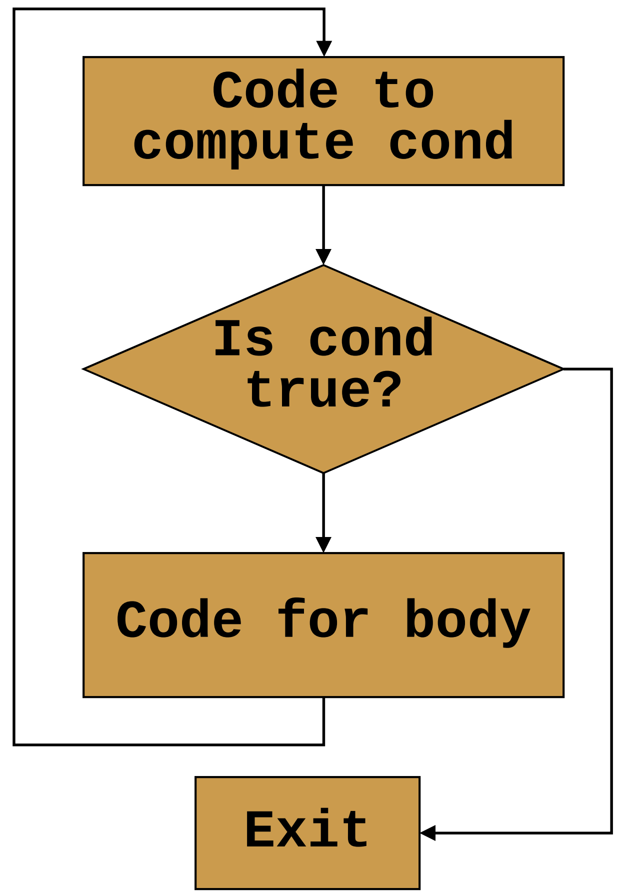
Source:

```
while (cond)
    body;
```

AST:



Flow chart:



Code template:

```
top_label:
   <code for condition>
ifnot_goto   <cond>, end_label
   <code for body>
goto top_label
end_label:
```

Generation scheme

- Two problems:
  - Getting the order right
  - How to pass values between the pieces of code
- **Solution**: order
  - Append each instruction to a global buffer
  - Emit instructions in the desired order
- **Solution**: passing values
  - Request a new (unique) temporary variable name
  - Generate code that computes value into the temp
  - Return the name of the temp to higher-level generate call

While loop

Compiler:

```
generate(WhileNode w) {
  E = new_label()
  T = new_label()
  emit( $T: )
  t = generate(w.Condition)
  emit( ifnot_goto $t, $E )
  generate(w.Body)
  emit( goto $T )
  emit( $E: )
}
```

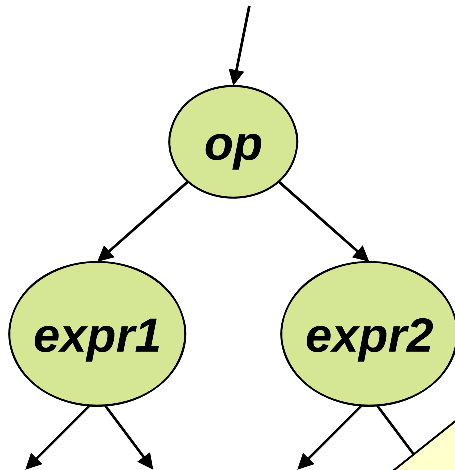
Code template:

```
top:
   <code for condition>
ifnot_goto <cond>, end
   <code for body>
goto top
end:
```

Lowering expressions

- Arithmetic operations

```
expr1 op expr2
```




```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Generate code for left and right children, get the registers holding the results

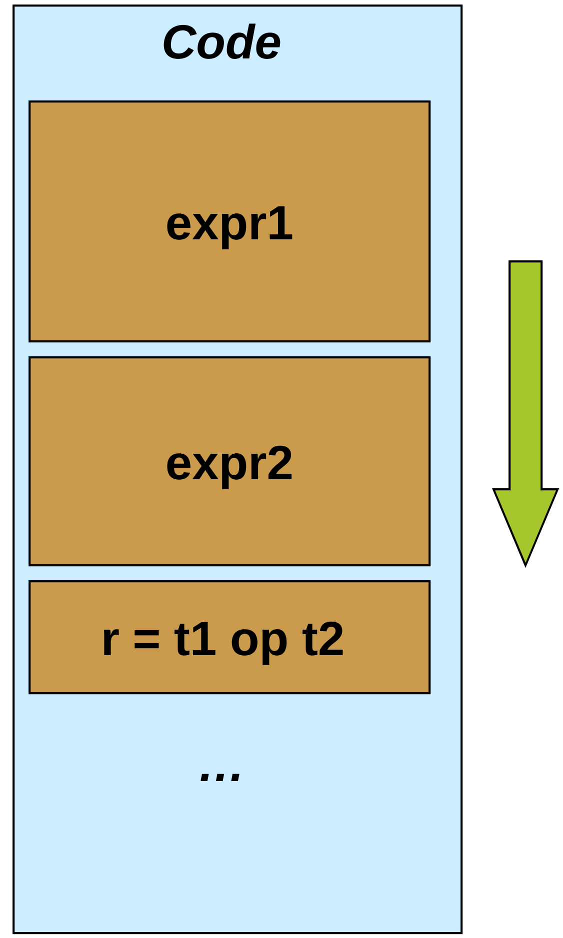
Obtain a fresh register name

Emit code for this operation

Return the register to *generate* call above

Lowering scheme

- Emit function
  - Appends low-level abstract instructions to a global buffer
  - **Order** of calls to emit is important!
- Scheme works for:
  - Binary arithmetic
  - Unary operations
  - Logic operations
- What about && and ||?
  - In C and Java, they are “short-circuiting”
  - Need control flow…

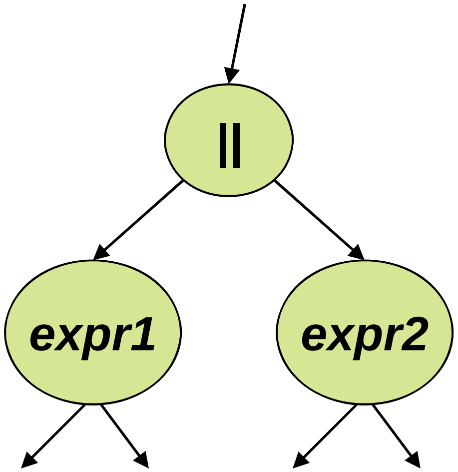


Short-circuiting ||

- If expr1 is true, don't eval expr2

```
expr1 || expr2
```

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```



Details...

```
E = new_label()
r = new temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

```
t1 = expr1
r = t1
if_goto t1, E
t2 = expr2
r = t2
E:
. . .
```

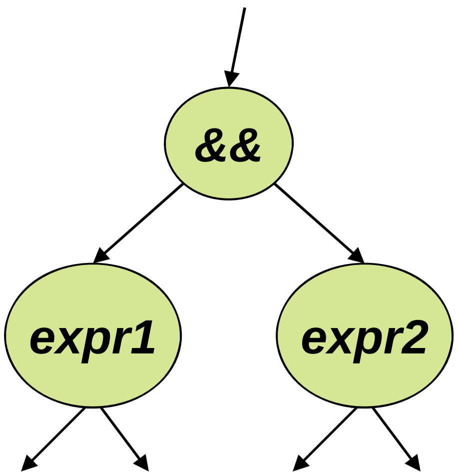
```
t3 =
L:
ifgoto
t1 =
```

Helper functions

- ***emit***()
  - The only function that generates instructions
  - Adds instructions to end of buffer
  - At the end, buffer contains code
- ***new_label***()
  - Generate a unique label name
  - Does not update code
- ***new_temp***()
  - Generate a unique temporary name
  - May require type information (from where?)

Short-circuiting &&

```
expr1 && expr2
```

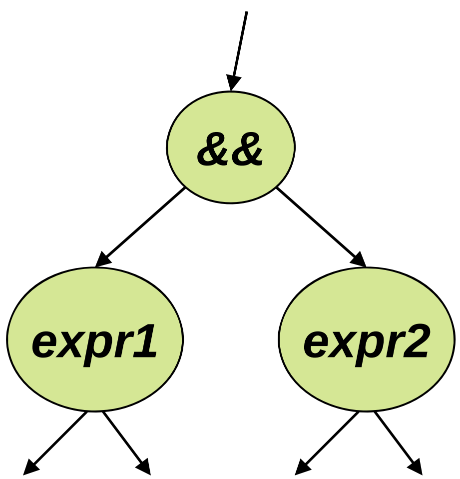



```
N = new_label()
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, N )
emit( goto E )
emit( N: )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Short-circuiting &&

- Can we do better?

```
expr1 && expr2
```




```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( ifnot_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Array access

- Depends on abstraction

```
expr1 [ expr2 ]
```

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

- OR:
  - Emit array op
  - Lower later

Type information from the symbol table

Statements

- Simple sequences

```
statement1;
statement2;
. . .
statementN;
```

```
generate(statement1)
generate(statement2)
. . .
generate(statementN)
```

- Conditionals

```
if (expr)
  statement;
```

```
E = new_label()
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( E: )
```

Loops

- Emit label for top of loop
- Generate condition and loop body

```
while (expr)
  statement;
```

```
E = new_label()
T = new_label()
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

For loop

- How does “for” work?

```
for (expr1; expr2; expr3)
  statement
```

Assignment

- How should we generate **`x = y`** ?
- ***Problem***
  - Difference between right-side and left-side
  - Right-side: a value *r-value*
  - Left-side: a location *l-value*

- Example: array assignment

```
A[i] = B[j]
```

Store to this location → `A[i]`

Load from this location → `B[j]`

Special generate

- Define special generate for l-values
  - ***lgenerate()*** returns register containing address
  - Use **lgenerate()** for left-side
- Return *r-value* for nested assignment

```
expr1 = expr2;
```

⇒

```
r = generate(expr2)
l = lgenerate(expr1)
emit( store *l = r )
return r
```

Example: arrays

- Code: `A[i] = B[j]`
- Two versions of generate:

```
r = new_temp()
a = generate(arr)
o = generate(index)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

r-value case

```
a = generate(arr)
o = generate(index)
emit( o = o * size )
emit( a = a + o )
return a
```

l-value case

At leaves

- Depends on level of abstraction
- ***generate***(v) – for variables
  - All virtual registers: return v
  - Strict register machine: ***emit***(r = load &v)
  - **Note**: may introduce many temporaries
  - **Later**: where is storage for v?

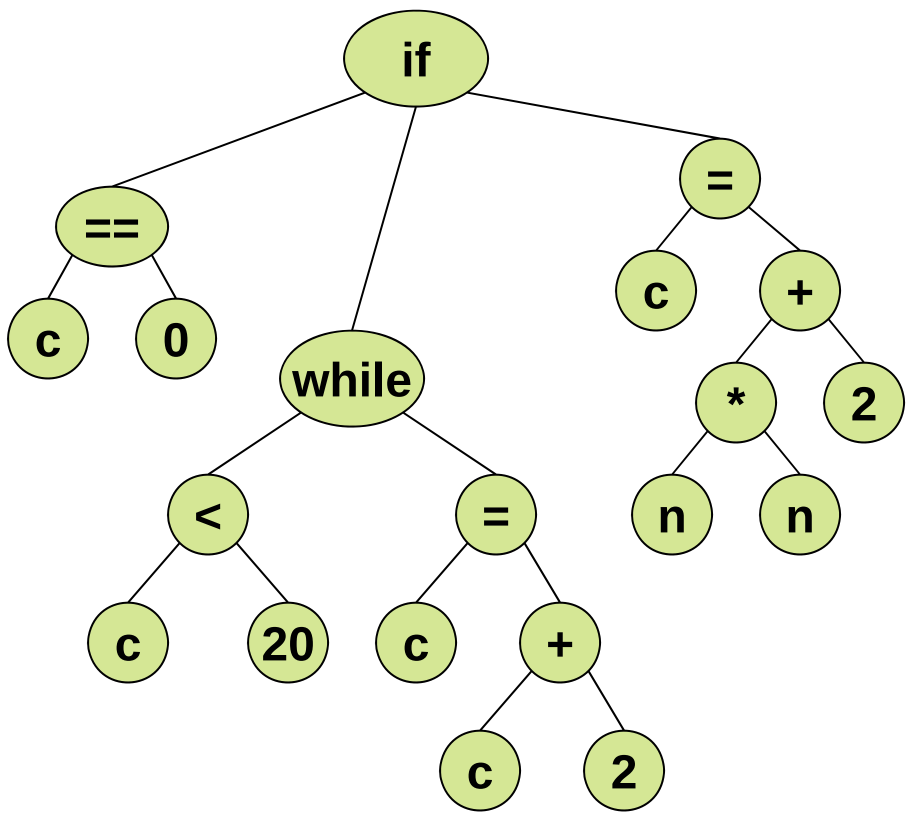
    More specifically: how the does the compiler set aside space for v and compute its address?
- ***generate***(c) – for constants
  - Special cases to avoid r = #

Generation: Big picture

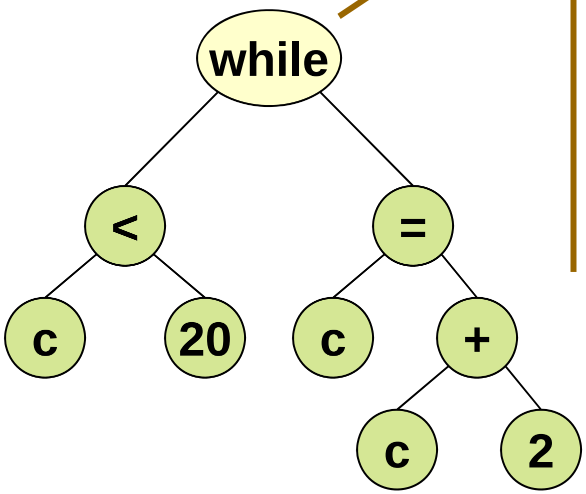
```
Reg generate(ASTNode node)
{
  Reg r;
  switch (node.getKind()) {
  case BIN: t1 = generate(node.getLeft());
            t2 = generate(node.getRight());
            r = new_temp();
            emit( r = t1 op t2 );
            break;
  case NUM: r = new_temp();
            emit( r = node.getValue() );
            break;
  case ID:  r = new_temp();
            o = symtab.getOffset(node.getID());
            emit( r = load <address of o> );
            break;
  }
  return r
}
```

Example

```
if (c == 0) {
  while (c < 20) {
    c = c + 2;
  }
}
else
  c = n * n + 2;
```



Example

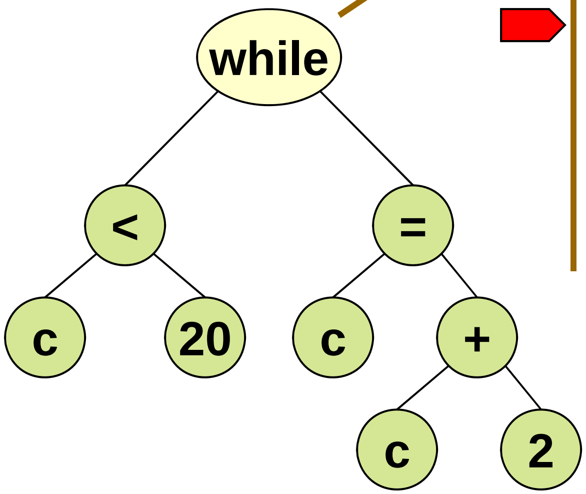



```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

Code

```
L1:
```

Example

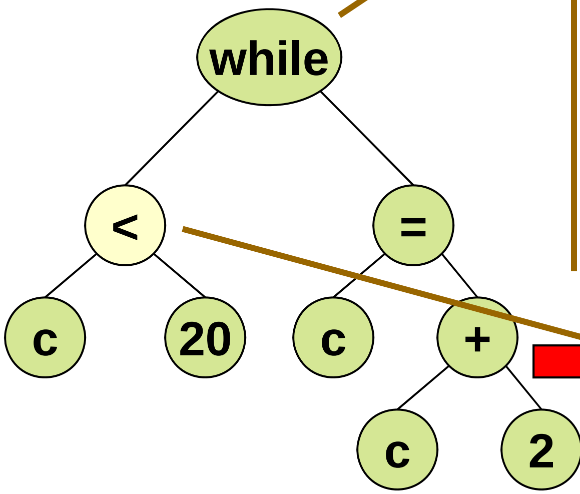



```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

Code

```
L1:
```

Example



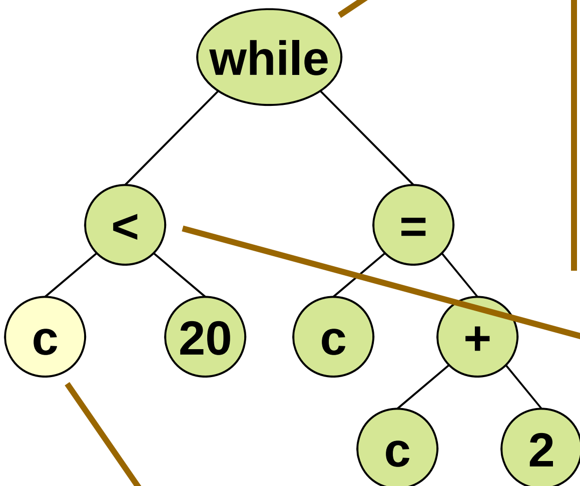

```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Code

```
L1:
```

Example




```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

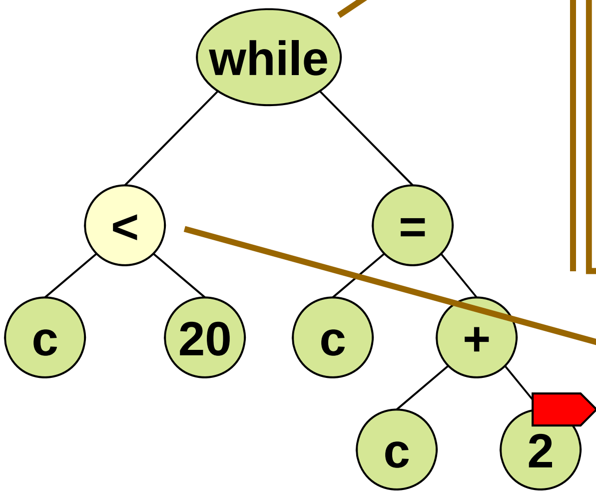
```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

```
r = new_temp() = R0
emit( r = load v )
return r
```

Code

```
L1:
R0 = load c
```

Example



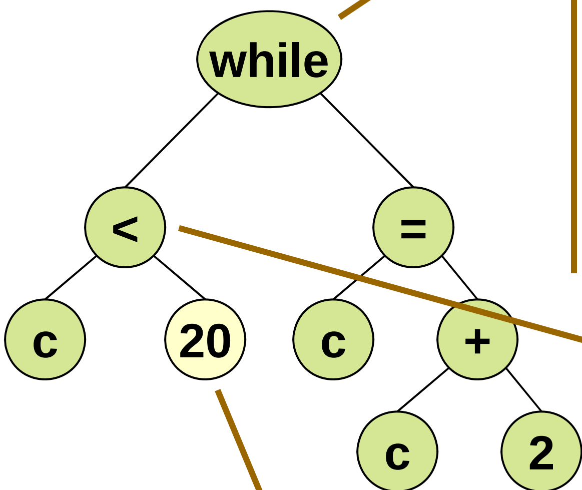

```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
t1 = generate(expr1)=R0
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Code

```
L1:
R0 = load c
```

Example




```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

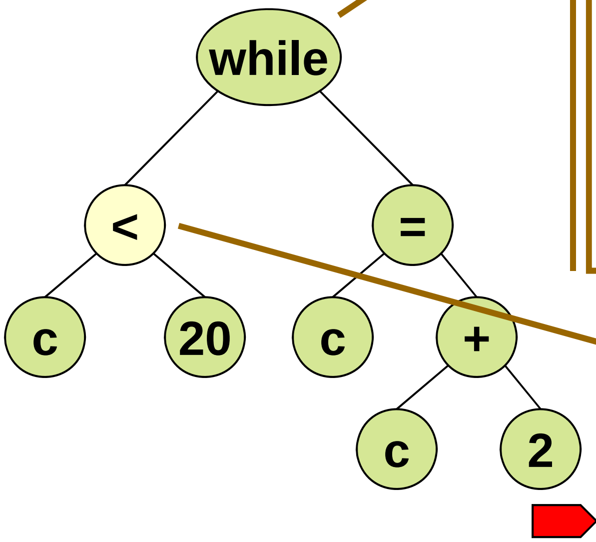
```
t1 = generate(expr1)=R0
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

```
r = new temp() = R1
emit( r = 20 )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
```

Example



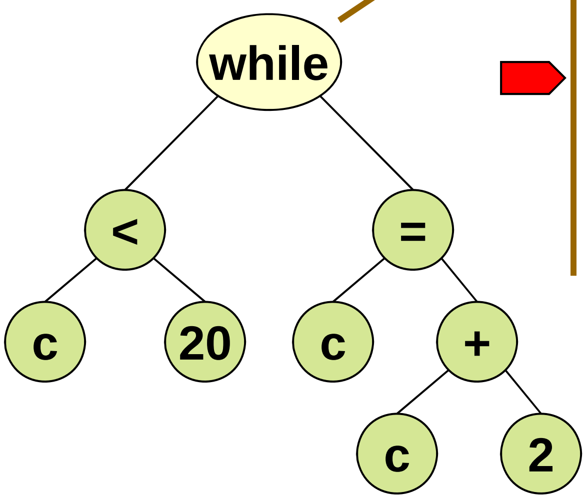

```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
t1 = generate(expr1)=R0
t2 = generate(expr2)=R1
r = new_temp() = R2
emit( r = t1 op t2 )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
```

Example

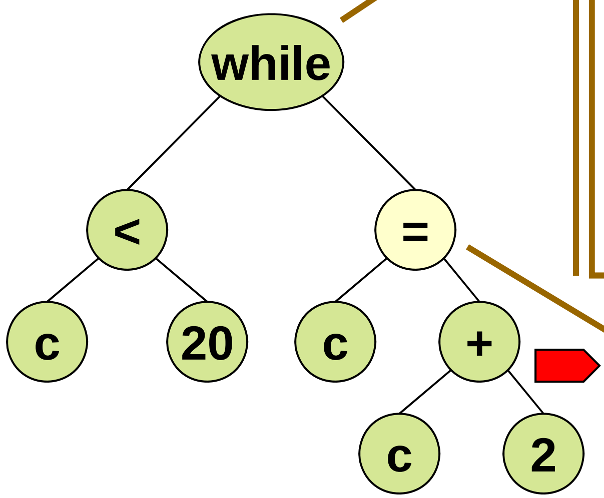



```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
```

Example



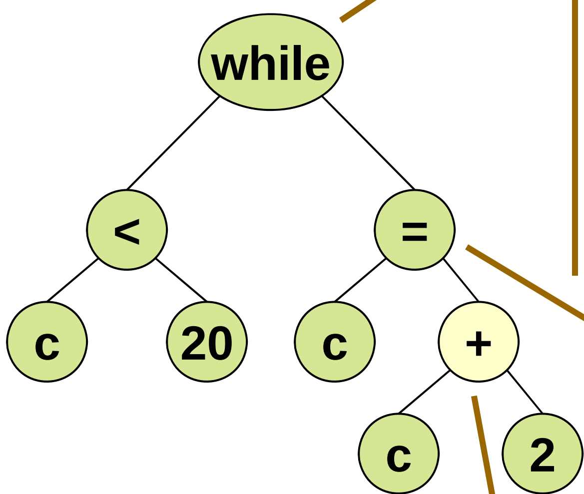

```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
r = generate(expr2)
l = lgenerate(expr2)
emit( store *l = r )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
```

Example




```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

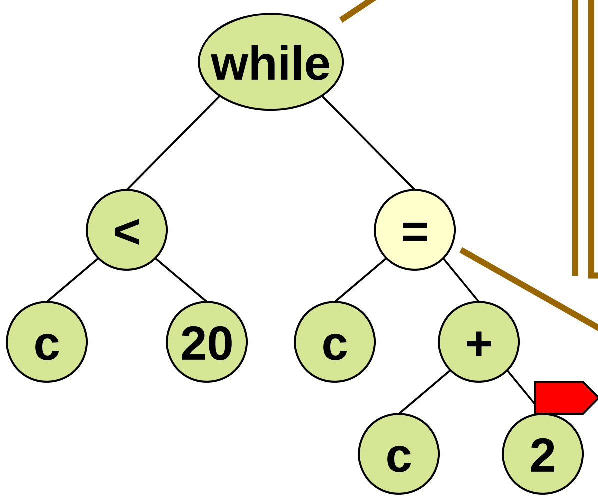
```
r = generate(expr2)
l = lgenerate(expr2)
emit( store *l = r )
return r
```

```
t1 = generate(expr1)=R3
t2 = generate(expr2)=R4
r = new_temp() = R5
emit( r = t1 op t2 )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
```

Example



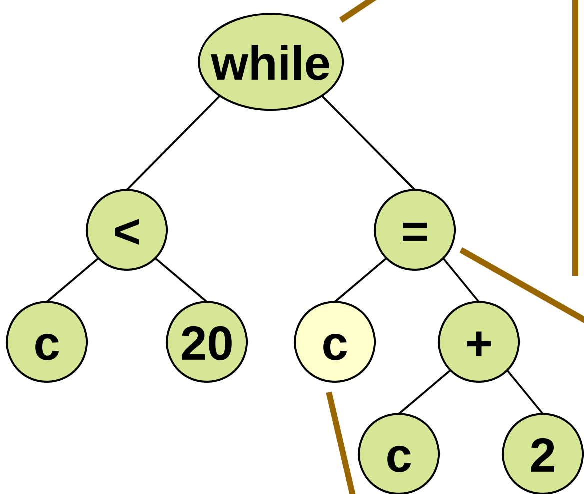

```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
r = generate(expr2)=R5
l = lgenerate(expr2)
emit( store *l = r )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
```

Example




```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
r = generate(expr2)=R5
l = lgenerate(expr2)
emit( store *l = r )
return r
```

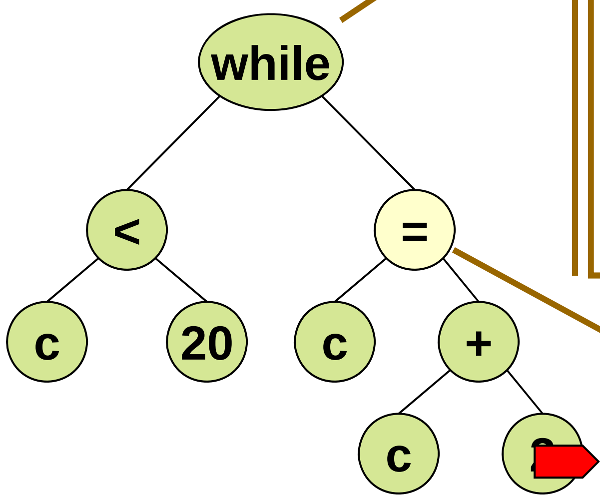
```
r = new_temp()= R6
emit( r = & v )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
```

Something like:
R6 = base + offset

Example



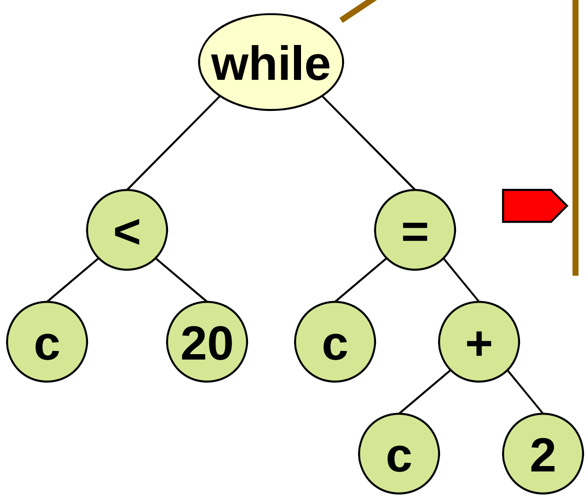

```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

```
r = generate(expr2)=R5
l = lgenerate(expr2)=R6
emit( store *l = r )
return r
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
store [R6]=R5
```

Example

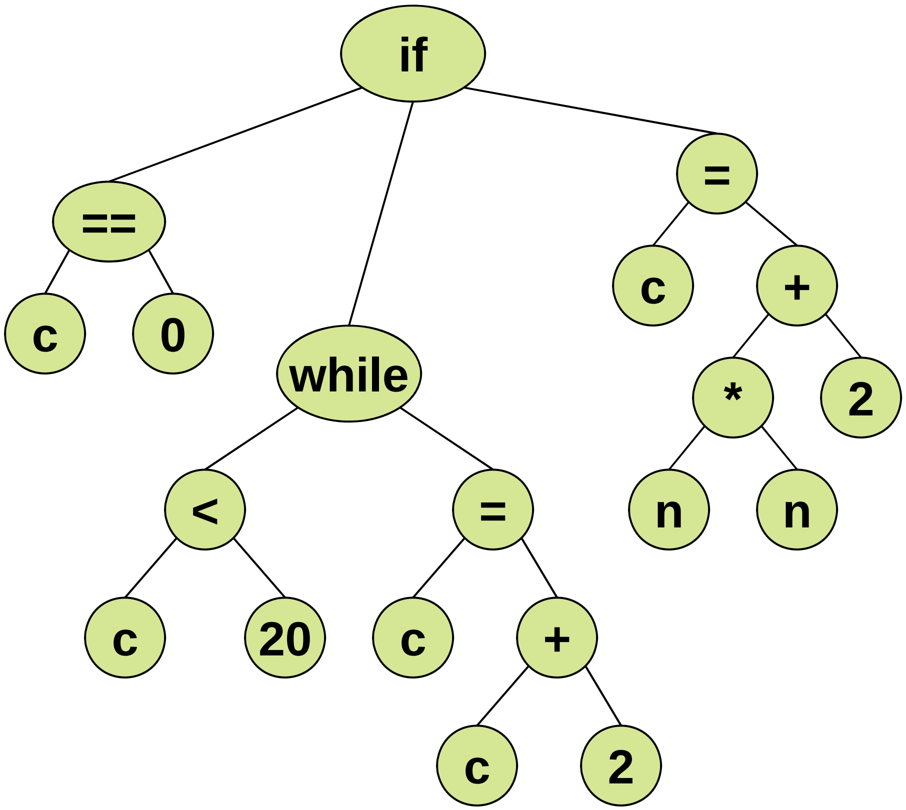



```
E = new_label() = L0
T = new_label() = L1
emit( T: )
t = generate(expr)=R2
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

Code

```
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
store [R6]=R5
goto L1
L0:
```

Example



Code

```
R7 = load c
R8 = 0
R9 = R7 == R8
not_goto R9,L3
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
store [R6]=R5
goto L1
L0:
goto L4
L3:
 . . .
L4:
```

Nesting

while (c<20)
c = c + 2

c < 20

c = c + 2

c + 2

```
      Code
R7 = load c
R8 = 0
R9 = R7 == R8
not_goto R9,L3
L1:
R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
store [R6]=R5
goto L1
L0:
goto L4
L3:
 . . .
L4:
```

Code quality

- Are there ways to make this code better?

Many CPUs have a fast c == 0 test

Can use accumulators: c = c + 2

Label leads to another goto; may have multiple labels

Code

```
R7 = c
R8 = 0
R9 = R7 == R8
not_goto R9,L3
L1:
R0 = c
R1 = 20
R2 = R0 < R1
not_goto R2,L0
R3 = c
R4 = 2
R5 = R3 + R2
c = R5
goto L1
L0:
goto L4
L3:
 . . .
L4:
```

Efficient lowering

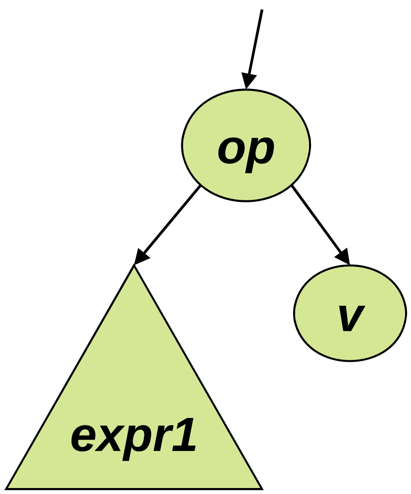
- Reduce number of temporary registers
  - Don’t copy variable values unnecessarily
  - Accumulate values, when possible
  - Reuse temporaries, where possible
    - highly depends on IR (e.g., if load-store, cannot do much)
- Generate more efficient labels
  - Don’t generate multiple adjacent labels
  - Avoid goto-label-goto
  - Typically done later, as a separate control-flow optimization

Avoiding extra copies

- Basic algorithm
  - Recursive generation traverses to leaves
  - At leaves, generate: R = v or R = c
- Improvement
  - Stop recursion one level early
  - Check to see if children are leaves
  - Don't call generate recursively on variables, constants

Avoiding copies

```
expr1 op expr2
```




```
if (expr1 is-a Var)
  t1 = (Var) expr1
else
  t1 = generate(expr1)
if (expr2 is-a Var)
  t2 = (Var) expr2
else
  t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Example

- ***Expr1*** is (a+b)
  - Not a leaf
  - Recursively generate code
  - Return temp
- ***Expr2*** is c
  - Return c
- ***Emit*** (R0 * c)

```
( a + b ) * c
```

Code

```
R0 = a + b
R1 = R0 * c
```

Use accumulation

- **Idea**:
  - We only need 2 registers to evaluate expr1 op expr2
  - Reuse temp assigned to one of the subexpressions

```
if (expr1 is var)
  t1 = (Var) expr1
else
  t1 = generate(expr1)
if (expr2 is var)
  t2 = (Var) expr2
else
  t2 = generate(expr2)
emit( t1 = t1 op t2 )
return t1
```

Example

- Combined:
  - Remove copies
  - Accumulate value
  - Only need one register
  - How many would the original scheme have used?

```
( a + b ) * c
```

Code

```
R0 = a + b
R0 = R0 * c
```

Reuse of temporaries

- **Idea**:

```
expr1 op expr2
```

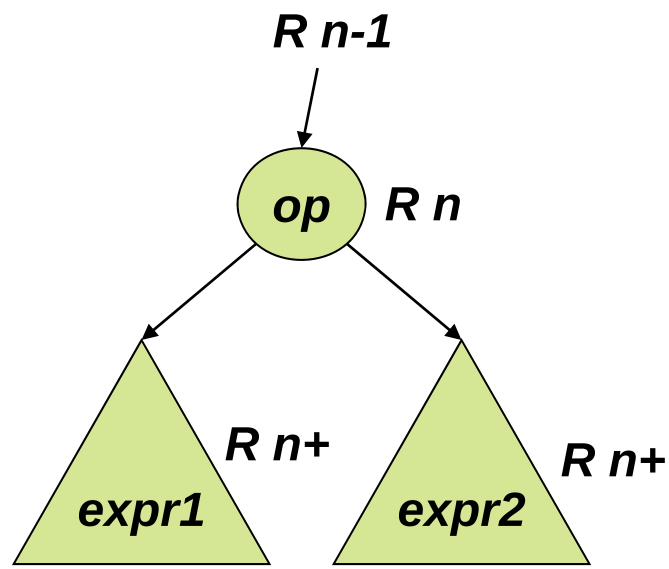
→

```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

- Can ***generate***(expr1) and ***generate***(expr2) share temporaries?
  - Yes, except for t1 and t2
  - Observation: temporaries have a limited lifetime
  - Lifetime confined to a subtree

Reuse of temporaries

- Subtrees can share registers
- Algorithm:
  - Use a stack of registers
  - Start at # = 0
  - Each call to generate:
    - "Push" next number
    - Use any register > #
    - When done, "pop" back up




```
R# = expr1
R# = R# op expr2
```

Miscellaneous

- Code "shape"
  - Consider expression **`x + y + z`**
  - Code:

```
t1 = x + y
t2 = t1 + z
```

```
t1 = x + z
t2 = t1 + y
```

```
t1 = y + z
t2 = t1 + x
```

  - What if x = 3 and y = 2
  - What if y+z evaluated earlier in code?
- Ordering for performance
  - Using associativity and commutativity – very hard
  - Operands
    - op1 must be preserved while op2 is computed
    - Emit code for more intensive one first

Code Generation

- Tree-walk algorithm
  - Notice: generates code for children first
  - Effectively, a bottom up algorithm
  - So that means….
- Right! Use syntax directed translation
  - Can emit LIR code in productions
  - Pass register names in $$, $1, $2, etc.
  - Can generate assembly: one-pass compiler
  - Tricky part: assignment

One-pass code generation

```
Goal::= Expr:e  {: RES = e :}
Expr::= Expr:e + Term:t
      {: r = new_temp();
         emit( r = e + t );
         RES = r; :}
    | Expr:e – Term:t
      {: r = new_temp();
         emit( r = e - t );
         RES = r; :}
```

```
Term::= Term:t * Fact:f
      {: r = new_temp();
         emit( r = t * f );
         RES = r; :}
    | Term:t / Fact:f
      {: r = new_temp();
         emit( r = t / f );
         RES = r; :}
```

```
Fact::= ID:i  {: r = new_temp();
                 o = symtab.getOffset(i);
                 emit( r = load <address of o> );
                 RES = r; :}
    | NUM:n {: r = new_temp();
               emit( r = $n );
               RES = r; :}
```