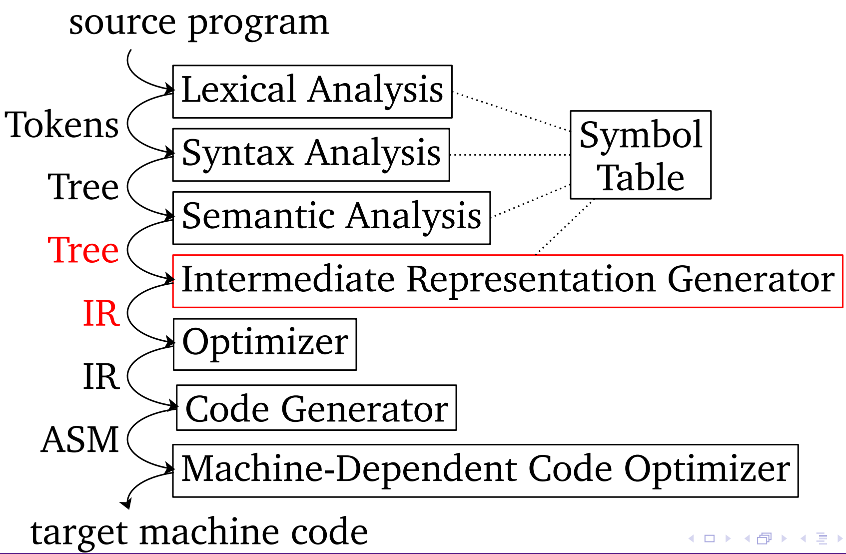
Intermediate Code Generation

Kristoffer H. Rose   Eva Rose
{*krisrose,evarose*}@*cs.nyu.edu*

Compiler Construction (CSCI-GA.2130-001) Spring 2014
NYU Courant Institute

March 24, 2014

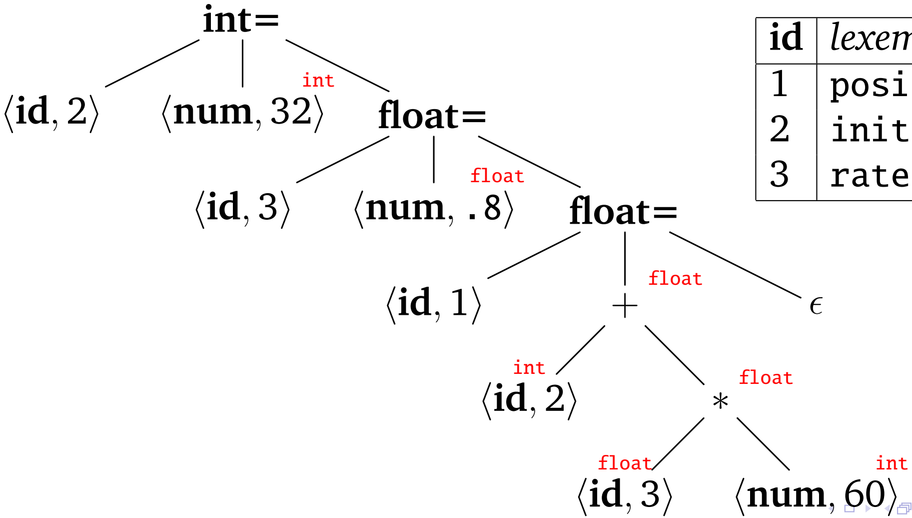
Fourth Compilation Phase



Not to worry. . .

IRs are Trees, too!

Example Abstract Syntax Tree (AST)



id	*lexeme*	*type*
1	position	float
2	initial	int
3	rate	float

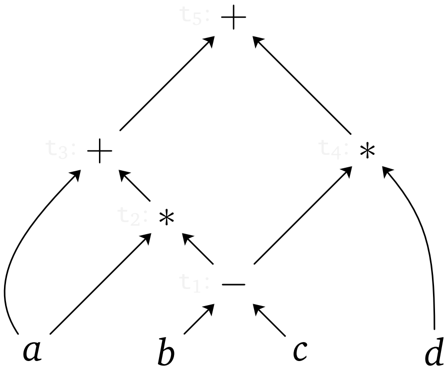
Example AST as Annotated Code

```
int initial = 32^int;
float rate = .8^int;
float position =    initial^int
                +^float
                          rate^float
                      *^float
                          8^int
;
```

Example Intermediate (Three-Address) Representation

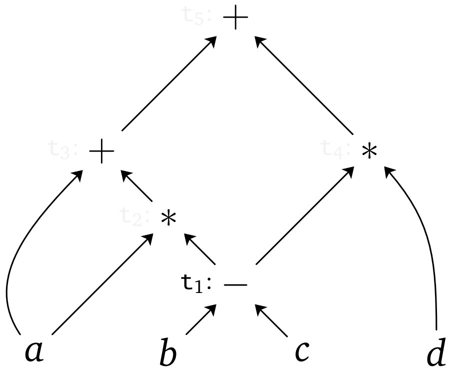
$$
\begin{array}{lll}
\texttt{int} & t_1 & = 32 \\
\texttt{int} & \texttt{initial} & = t_1 \\
\texttt{float} & t_2 & = .8 \\
\texttt{float} & \texttt{rate} & = t_2 \\
\texttt{int} & t_3 & = \texttt{initial} \\
\texttt{float} & t_4 & = \texttt{rate} \\
\texttt{int} & t_5 & = 8 \\
\texttt{float} & t_6 & = (\texttt{float})\ t_3 \\
\texttt{float} & t_7 & = t_4 * t_6 \\
\texttt{float} & t_8 & = t_6 + t_7 \\
\texttt{float} & \texttt{position} & = t_8
\end{array}
$$

A Value Graph (DAG)



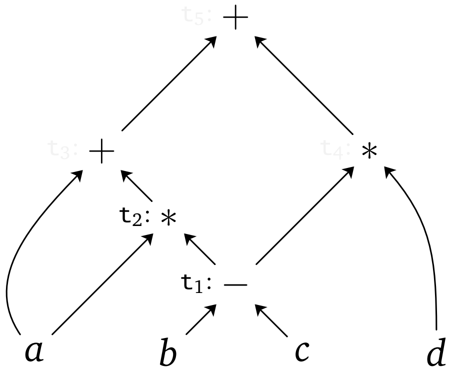
- $t_1 = b - c$
- $t_2 = a * t_1$
- $t_3 = a + t_2$
- $t_4 = t_1 * d$
- $t_5 = t_3 + t_4$

A Value Graph (DAG) and Code



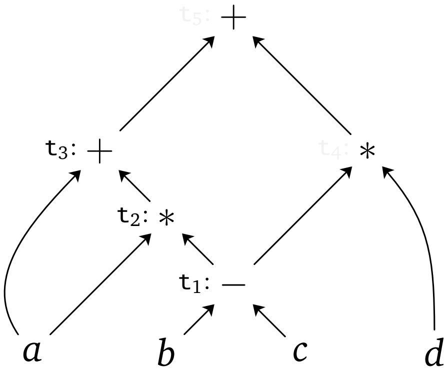

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

A Value Graph (DAG) and Code



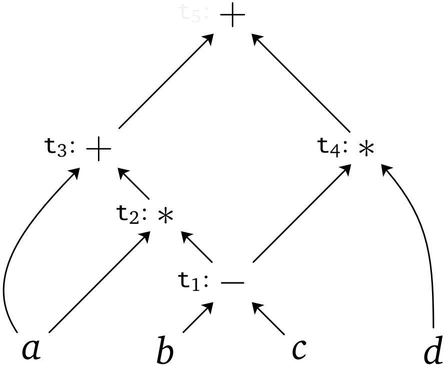
- t_1 = b - c
- t_2 = a * t_1
- t_3 = a + t_2
- t_4 = t_1 * d
- t_5 = t_3 + t_4

A Value Graph (DAG) and Code



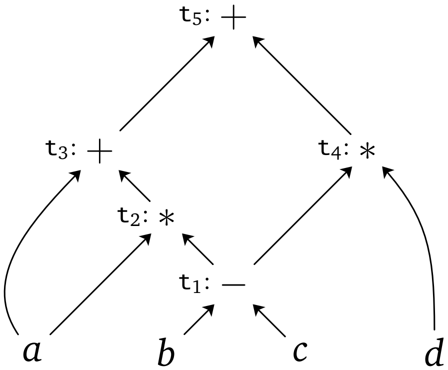
- $t_1 = b - c$
- $t_2 = a * t_1$
- $t_3 = a + t_2$
- $t_4 = t_1 * d$
- $t_5 = t_3 + t_4$

A Value Graph (DAG) and Code



- t_1 = b - c
- t_2 = a * t_1
- t_3 = a + t_2
- t_4 = t_1 * d
- t_5 = t_3 + t_4

A Value Graph (DAG) and Code



- $t_1 = b - c$
- $t_2 = a * t_1$
- $t_3 = a + t_2$
- $t_4 = t_1 * d$
- $t_5 = t_3 + t_4$

What is "Address" in Three-Address Code?

Name (from the source program)

Constant (with explicit primitive type)

Compiler-generated temporary ("register")

What are the Instructions of Three-Address Code?

1. $x = y \, op \, z$: with *op* a binary operation
2. $x = op \, y$: with *op* a unary operation
3. $x = y$: copy operation
4. goto L: unconditional jump to label L
5. if x goto L: jump to L is x is true (for ifFalse when false)
6. if $x \, relop \, y$ goto L: jump to L if *relop*-comparison holds
7. param x and call P: push x on parameter stack then call P
8. $x = y[i]$ and $x[i] = y$: indexed copy instructions
9. $x = \&y$, $x = *y$, and $*x = y$: address/pointer assignments

Variations on Three-Address Code

- Some label scheme – we use L: instructions
- Some temporary management – we write the explicit type when needed.

Static Single-Assignment Form

Every distinct assignment must be to a distinct temporary:

```
if (f) x=1; else x=2;  y=x*a;
```

is changed to

if (f) x_1=1; else x_2=2; y=ϕ(x_1,x_2)*a;

1. Three-Address Code
2. **Translations of Expressions**
3. Translations of Arrays
4. Control Flow
5. Procedure Calls
6. HACS

Syntax-Directed Definition

PRODUCTION	RULES
$S \to$ **id** **=** E_1 **;** S_2	$E_1.e = S.e; S_2.e = S.e; S.c = E_1.c \| \| [\![\textbf{id} = E_1.a]\!] \| \| S_2.c$
$\mid \epsilon$	$S.c = [\![\,]\!]$
$E \to E_1$ **+** E_2	$E_1.e = E.e; E_2.e = E.e; E.a = \text{newTemp}$ $E.c = E_1.c \| \| E_2.c \| \| [\![E.a = E_1.a + E_2.a]\!]$
$\mid$ **-** E_1	$E_1.e = E.e; E.a = \text{newTemp}$ $E.c = E_1.c \| \| [\![E.a = -E_1.a]\!]$
$\mid$ **(** E_1 **)**	$E_1.e = E.e; E.a = E_1.a; E.c = E_1.c$
$\mid$ **id**	$E.a = \textbf{id}; E.c = [\![\,]\!]$

with inherited environments $S.e$ and $E.e$, synthesized addresses $E.a$, and synthesized code $S.c$ and $E.c$.

Variations...

Global Symbol Table. The symbol table is managed by global updates to data structure.

Incremental Translation. Each semantic rule includes an action that describes what code is appended to the global code stream.

In both cases depends on evaluation order of semantic rules.

Variations. . .

Global Symbol Table. The symbol table is managed by global updates to data structure.

Incremental Translation. Each semantic rule includes an action that describes what code is appended to the global code stream.

In both cases depends on evaluation order of semantic rules.

1. Three-Address Code
2. Translations of Expressions
3. Translations of Arrays
4. Control Flow
5. Procedure Calls
6. HACS

Arrays

- One Dimension:

$$addr = base + i \times w$$

- Two dimensions, row-major (n_2 is size of second dimension):

$$addr = base + (i_1 \times n_2 + i_2) \times w$$

- k dimensions, row-major:

$$addr = base + ((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$$

Arrays

- One Dimension:

$$addr = base + i \times w$$

- Two dimensions, row-major (n_2 is size of second dimension):

$$addr = base + (i_1 \times n_2 + i_2) \times w$$

- k dimensions, row-major:

$$addr = base + ((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w$$

Arrays

- One Dimension:

$$addr = base + i \times w$$

- Two dimensions, row-major (n_2 is size of second dimension):

$$addr = base + (i_1 \times n_2 + i_2) \times w$$

- k dimensions, row-major:

$$addr = base + ((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$$

SDT

PRODUCTIONS	RULES
$S \to \mathbf{id} = E_1$;	$\{ gen(top.get(\mathbf{id}.lexeme) = E.addr);\ \}$
$\mid L = E_1$;	$\{ gen(L.array.base\ [\ L.addr\ ] = E.addr;\ \}$
$E \to E_1 + E_2$	$\{ E.addr = \mathbf{new}\ Temp(); gen(E.addr = E_1.addr + E_2.addr);\ \}$
$\mid \mathbf{id}$	$\{ E.addr = top.get(\mathbf{id}.lexeme);\ \}$
$\mid L_1$	$\{ E.addr = \mathbf{new}\ Temp(); gen(E.addr = L_1.array.base\ [\ L.addr\ ]);\ \}$
$L \to \mathbf{id}\ [\ E_1\ ]$	$\{ L.array = top.get(\mathbf{id}.lexeme); L.type = L_1.type.elem;$ $L.addr = \mathbf{new}\ Temp(); gen(L.addr = E_1.addr * L.type.width);\ \}$
$\mid L_1\ [\ E_1\ ]$	$\{ L.array = L_1.array; L.type = L_1.type.elem;$ $t = \mathbf{new}\ Temp(); L.addr = \mathbf{new}\ Temp();$ $gen(t = E_1.addr * L.type.width); gen(L.addr = E_1.addr + t);\ \}$

Note: This is in “action” form, assuming sequential (post-order) runs of *gen*.

1 Three-Address Code

2 Translations of Expressions

3 Translations of Arrays

4 Control Flow

5 Procedure Calls

6 HACS

Conditionals

if (B_1)
S_2
else S_3

ifFalse B_1 goto L_3
S_2
goto L_2
L_3:
S_3
L_2:

Conditionals

```
if (B1)
S2
else S3
```

```
     ifFalse [B1] goto L3
     [S2]
     goto L2
L3:
     [S3]
L2:
```

Conditionals, Example

`if ((a+1)>b)`
S_2
`else` S_3

$\texttt{t}_1$ `= a + 1`
`ifFalse t > b goto` L_3
S_2
`goto` L_2
L_3`:`
S_3
L_2`:`

Conditionals, Example

```
if ((a+1)>b)
S2
else S3
```

```
      t1 = a + 1
      ifFalse t > b goto L3
      [S2]
      goto L2
L3:
      [S3]
L2:
```

Loops

`while` (B_1)
S_2

`goto` L_2
L_1:
S_2
L_2:
`if` B_1 `goto` L_1

Loops

```
while (B₁)
S₂
```

$\texttt{while}\ (B_1)$
S_2

$\texttt{goto}\ L_2$

L_1:

$\boxed{S_2}$

L_2:

$\texttt{if}\ \boxed{B_1}\ \texttt{goto}\ L_1$

Booleans

B	$B.BJump(L)$	$B.\overline{BJump}(L)$
`false`	ϵ	`goto` L
`true`	`goto` L	ϵ
`!`B_1	$B_1.\overline{BJump}(L)$	$B_1.BJump(L)$
B_1 `||` B_2	$B_1.BJump(L)$ $B_2.BJump(L)$	$B_1.\overline{BJump}(L')$ $B_2.\overline{BJump}(L')$ `goto` L L':
B_1&&B_2	$B_1.\overline{BJump}(L')$ $B_2.\overline{BJump}(L')$ `goto` L L':	$B_1.BJump(L)$ $B_2.BJump(L)$

Booleans

B	$B.BJump(L)$	$B.\overline{BJump}(L)$
`false` `true`	ϵ `goto` L	`goto` L ϵ
`!`B_1	$B_1.\overline{BJump}(L)$	$B_1.BJump(L)$
B_1 `||` B_2	$B_1.BJump(L)$ $B_2.BJump(L)$	$B_1.\overline{BJump}(L')$ $B_2.\overline{BJump}(L')$ `goto` L L':
B_1`&&`B_2	$B_1.\overline{BJump}(L')$ $B_2.\overline{BJump}(L')$ `goto` L L':	$B_1.BJump(L)$ $B_2.BJump(L)$

Booleans

B	$B.BJump(L)$	$B.\overline{BJump}(L)$
`false`	ϵ	`goto` L
`true`	`goto` L	ϵ
`!`B_1	$B_1.\overline{BJump}(L)$	$B_1.BJump(L)$
B_1`||`B_2	$B_1.BJump(L)$ $B_2.BJump(L)$	$B_1.\overline{BJump}(L')$ $B_2.\overline{BJump}(L')$ `goto` L L'`:`
B_1&&B_2	$B_1.\overline{BJump}(L')$ $B_2.\overline{BJump}(L')$ `goto` L L'`:`	$B_1.BJump(L)$ $B_2.BJump(L)$

Comparisons

B	$B.BJump(L)$	$B.\overline{BJump}(L)$
$E_1 < E_2$	$E_1.c$ $E_2.c$ `if` $E_1.a < E_2.a$ `goto` L	$E_1.c$ $E_2.c$ `ifFalse` $E_1.a < E_2.a$ `goto` L

Comparisons

B	$B.BJump(L)$	$B.\overline{BJump}(L)$
$E_1 < E_2$	$E_1.c$ $E_2.c$ `if` $E_1.a < E_2.a$ `goto` L	$E_1.c$ $E_2.c$ `ifFalse` $E_1.a < E_2.a$ `goto` L

Comparisons

B	$B.BJump(L)$	$B.\overline{BJump}(L)$
$E_1 < E_2$	$E_1.c$ $E_2.c$ `if` $E_1.a < E_2.a$ `goto` L	$E_1.c$ $E_2.c$ `ifFalse` $E_1.a < E_2.a$ `goto` L

1. Three-Address Code
2. Translations of Expressions
3. Translations of Arrays
4. Control Flow
5. **Procedure Calls**
6. HACS

Calls

$$x = f(E_1, \ldots, E_n)$$

$E_1.c$
...
$E_n.c$
param $E_1.a$
...
param $E_n.a$
x = call f

Calls

$x = f(E_1, \ldots, E_n)$

$E_1.c$
...
$E_n.c$
`param` $E_1.a$
...
`param` $E_n.a$
`x = call` f

Example I

```
token T   | T ('_' ⟨Int⟩)* ; // temporary
sort Tmp | symbol ⟦⟨T⟩ ⟧ ;

// Concrete syntax & abstract syntax sorts.

sort I_Progr | ⟦⟨I_Instr⟩ ⟨I_Progr⟩⟧ | ⟦⟧ ;

sort I_Instr  | ⟦⟨I_Type⟩ ⟨Tmp⟩ = ⟨I_Arg⟩ + ⟨I_Arg⟩;¶⟧
              | ⟦⟨I_Type⟩ ⟨Tmp⟩ = ⟨I_Arg⟩ * ⟨I_Arg⟩;¶⟧
              | ⟦⟨I_Type⟩ ⟨Tmp⟩ = ⟨I_Arg⟩;¶⟧
              | ⟦⟨I_Type⟩ ⟨Name⟩ = ⟨Tmp⟩;¶⟧ ;

sort I_Arg    | ⟦⟨Name⟩⟧ | ⟦⟨Float⟩⟧ | ⟦⟨Int⟩⟧ | ⟦⟨Tmp⟩⟧ ;
```

Example II

```
// Translation scheme.

attribute ↓TmpType{Tmp:Type} ;

sort I_Progr | scheme ⟦ICG { ⟨Stat⟩ } ⟧ ↓TmpType ;

⟦ ICG { id := ⟨Exp#2 ↑t(#t2)⟩; ⟨Stat#3[id]⟩ } ⟧
   → ⟦ { ⟨I_Progr ⟦ICGExp T ⟨Exp#2⟩⟧ ↓TmpType{⟦T⟧:#t2}⟩ }
         ITy ⟨Type#t2⟩ id = T; ICG { ⟨Stat#3[id]⟩ } ⟧ ;

⟦ ICG { { ⟨Stat#1⟩ } ⟨Stat#2⟩ } ⟧→ ⟦ { ICG { ⟨Stat#1⟩ } } ICG { ⟨Stat#2⟩ } ⟧
;
```

Example III

$$[\![\ \text{ICG}\ \{\ \}\ ]\!] \to [\![\ \ ]\!];$$

$$|\ \textbf{scheme}\ [\![\text{ICGExp}\ \langle\text{Tmp}\rangle\ \langle\text{Exp}\rangle\ ]\!];$$

$$[\![\ \text{ICGExp T}\ \langle\text{Int\#1}\rangle\ ]\!] \to [\![\ \text{T} = \langle\text{Int\#1}\rangle;\ ]\!]\ ;$$
$$[\![\ \text{ICGExp T}\ \langle\text{Float\#1}\rangle\ ]\!] \to [\![\ \text{T} = \langle\text{Float\#1}\rangle;\ ]\!]\ ;$$
$$[\![\ \text{ICGExp T id}\ ]\!] \to [\![\ \text{T} = \text{id};\ ]\!]\ ;$$

$$[\![\ \text{ICGExp T}\ \langle\text{Exp\#1}\rangle + \langle\text{Exp\#2}\rangle\ ]\!]$$
$$\to [\![\ \{\text{ICGExp T\_1}\ \langle\text{Exp\#1}\rangle\}\ \{\text{ICGExp T\_2}\ \langle\text{Exp\#2}\rangle\}\ \text{T} = \text{T\_1} + \text{T\_2};\ ]\!]$$
$$;$$

$$[\![\ \text{ICGExp T}\ \langle\text{Exp\#1}\rangle * \langle\text{Exp\#2}\rangle\ ]\!]$$

Example IV

$\rightarrow$ ⟦ {ICGExp T_1 ⟨Exp#1⟩} {ICGExp T_2 ⟨Exp#2⟩} T = T_1 ∗ T_2; ⟧
;

// Helper to flatten code sequence.
| **scheme** ⟦{⟨I_Progr⟩} ⟨I_Progr⟩ ⟧;
⟦ {} ⟨I_Progr#3⟩ ⟧ $\rightarrow$ #3 ;
⟦ {⟨I_Instr#1⟩ ⟨I_Progr#2⟩} ⟨I_Progr#3⟩ ⟧ $\rightarrow$ ⟦⟨I_Instr#1⟩ {⟨I_Progr#2⟩} ⟨I_

Questions?

krisrose@cs.nyu.edu