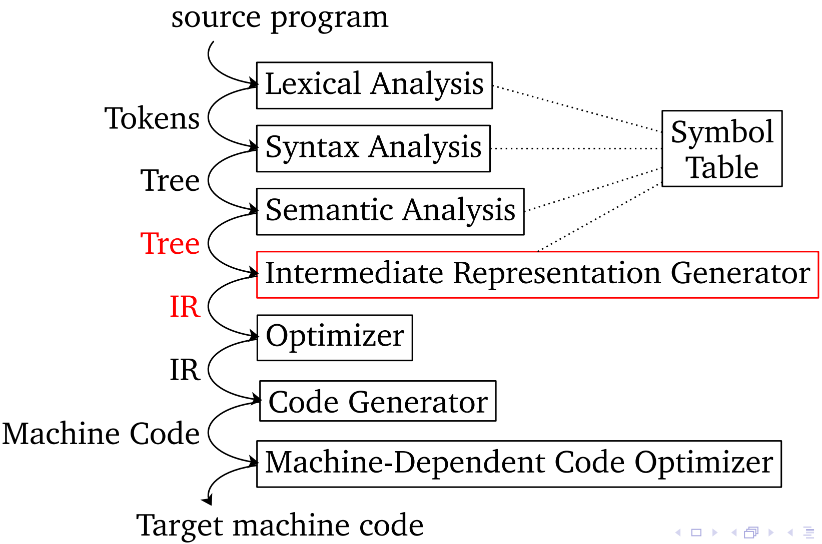
Intermediate Code Generation

Eva Rose Kristoffer Rose

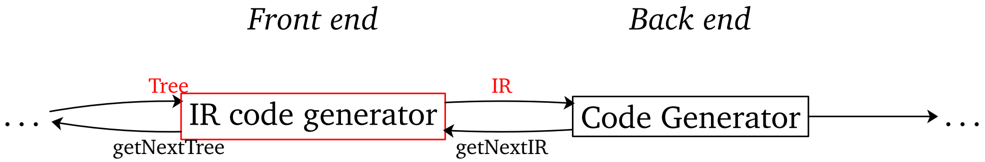
NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-7.pdf

October 16, 2014

Fourth Compilation Phase



IR generator: front-end bordering back-end

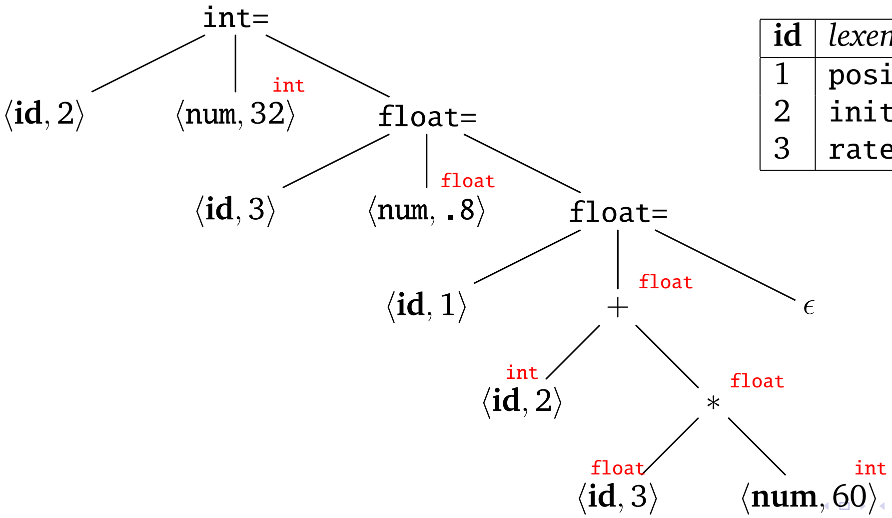


$m \times n$ compilers can be built by writing just m front-ends and n back-ends.

Canonical example

Back to the introductory example...

Example Abstract Syntax Tree (AST)



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

Example AST as Annotated Code

$\texttt{int initial} = \texttt{32}^{\color{red}\texttt{int}}\texttt{;}$

$\texttt{float rate} = \texttt{.8}^{\color{red}\texttt{float}}\texttt{;}$

$\texttt{float position} = \quad \texttt{initial}^{\color{red}\texttt{int}}$

$\qquad +^{\color{red}\texttt{float}}$

$\qquad \texttt{rate}^{\color{red}\texttt{float}}$

$\qquad *^{\color{red}\texttt{float}}$

$\qquad \texttt{8}^{\color{red}\texttt{int}}$

;

Example Intermediate Representation (code)

```
int   t₁ = 32
int   initial = t₁
float t₂ = .8
float rate = t₂
int   t₃ = initial
float t₄ = rate
int   t₅ = 8
float t₆ = (float) t₃
float t₇ = t₄ * t₆
float t₈ = t₆ + t₇
float position = t₈
```

Intermediate representation

There are essentially 2 steps:

High level IR (DAG tree) + Lowlevel IR (Three-address code)

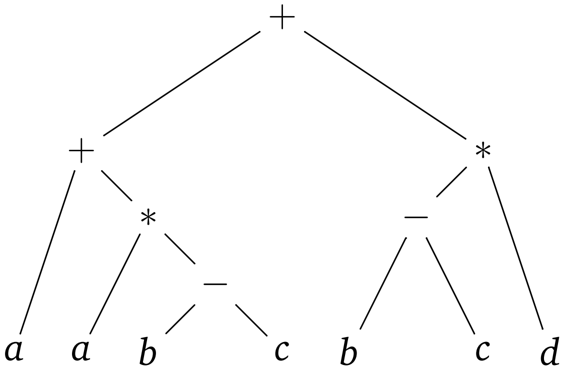
Syntax trees

Recall AST construction of simple expressions (lecture 4):

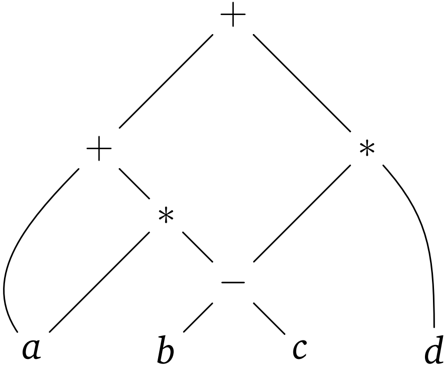
	Production	Semantic Rule
1.	$E \rightarrow E_1 + T_2$	$E.node = \textbf{new}\ Node('+', E_1.node, T_2.node)$
2.	$E \rightarrow E_1 - T_2$	$E.node = \textbf{new}\ Node('-', E_1.node, T_2.node)$
3.	$E \rightarrow E_1 * T_2$	$E.node = \textbf{new}\ Node('*', E_1.node, T_2.node)$
4.	$E \rightarrow T$	$E.node = T.node$
5.	$T \rightarrow (E)$	$T.node = E.node$
6.	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7.	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.entry)$

- Draw up an AST for $a + a * (b - c) + (b - c) * d$

AST for $a + a * (b - c) + (b - c) * d$



DAG for $a + a * (b - c) + (b - c) * d$



Directed Acyclic Graph (DAG)

- No repetition of patterns.
- Node can have more than one parent.
- More compact representation than AST.
- Gives clues regarding generation of efficient code...

Example

Construct the DAG for:

- ((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))

DAG from SDD

How to generate DAGs from Syntax-Directed Definitions:

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T_2$	$E.node = \textbf{new}\ Node('+', E_1.node, T_2.node)$
$E \to E_1 - T_2$	$E.node = \textbf{new}\ Node('-', E_1.node, T_2.node)$
$E \to E_1 * T_2$	$E.node = \textbf{new}\ Node('*', E_1.node, T_2.node)$
$E \to T$	$E.node = T.node$
$T \to (\ E\ )$	$T.node = E.node$
$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.entry)$

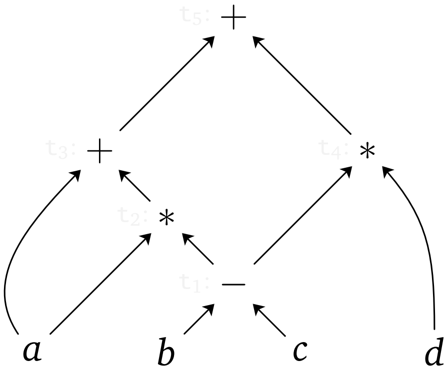
All that is needed are functions such as **Node** and **Leaf** above, that checks if the node has been created before. If a node already exists, a pointer to that node is returned.

SDD to DAG

Input string: $a + a * (b - c) + (b - c) * d$

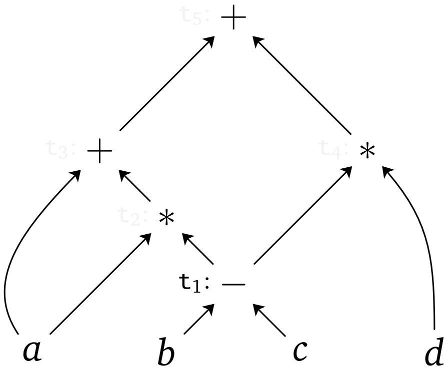
$$
\begin{array}{lll}
p_1 & = Leaf(id, entry{-}a) & \\
p_2 & = Leaf(id, entry{-}a) & = p_1 \\
p_3 & = Leaf(id, entry{-}b) & \\
p_4 & = Leaf(id, entry{-}c) & \\
p_5 & = Node('-', p_3, p_4) & \\
p_6 & = Node('*', p_1, p_5) & \\
P_7 & = Node('+', p_1, p_6) & \\
P_8 & = Leaf(id, entry{-}b) & = P_3 \\
P_9 & = Leaf(id, entry{-}c) & = P_4 \\
P_{10} & = Node('-', p_3, p_4) & = P_5 \\
P_{11} & = Leaf(id, entry{-}d) & \\
P_{12} & = Node('*', p_5, p_{11}) & \\
P_{13} & = Node('*', p_7, p_{12}) &
\end{array}
$$

A Value Graph (DAG)



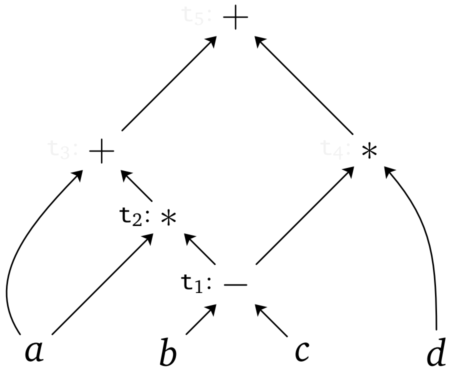
- $t_1 = b - c$
- $t_2 = a * t_1$
- $t_3 = a + t_2$
- $t_4 = t_1 * d$
- $t_5 = t_3 + t_4$

A Value Graph (DAG) and Code



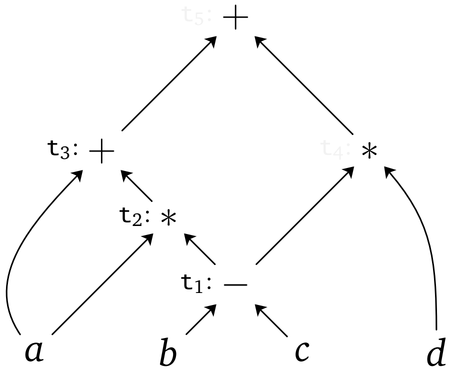
- `t1 = b - c`
- `t2 = a * t1`
- `t3 = a + t2`
- `t4 = t1 * d`
- `t5 = t3 + t4`

A Value Graph (DAG) and Code



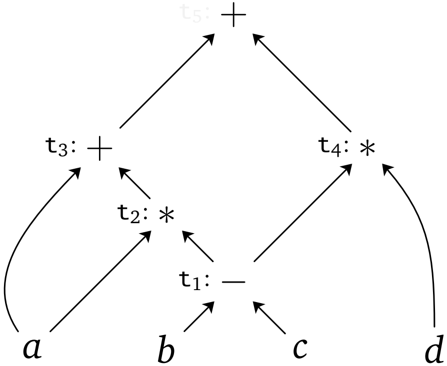
- t_1 = b - c
- t_2 = a * t_1
- t_3 = a + t_2
- t_4 = t_1 * d
- t_5 = t_3 + t_4

A Value Graph (DAG) and Code



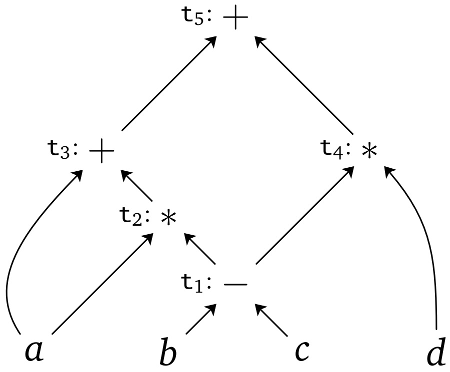
- `t1 = b - c`
- `t2 = a * t1`
- `t3 = a + t2`
- `t4 = t1 * d`
- `t5 = t3 + t4`

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



- $\texttt{t}_1$ = b - c
- $\texttt{t}_2$ = a * $\texttt{t}_1$
- $\texttt{t}_3$ = a + $\texttt{t}_2$
- $\texttt{t}_4$ = $\texttt{t}_1$ * d
- $\texttt{t}_5$ = $\texttt{t}_3$ + $\texttt{t}_4$

Three-address code

Characteristics:

- Lineralized representation of a syntax tree/DAG.
- Explicit names for interior nodes of the graph.
- Two concepts: addresses and instructions.
- At most one operator on the right side of instruction.

Address

What is an "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. $x = y \; op \; z$: where *op* is a binary operation
2. $x = op \; y$: where *op* is 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.
6. ifFalse x goto L : jump to L is x is false.
7. if $x \; relop \; y$ goto L : jump to L if *relop*-comparison holds
8. param x and call P : push x on parameter stack then call P
9. $x = y[i]$ and $x[i] = y$: indexed copy instructions
10. $x = \&y$, $x = *y$, and $*x = y$: address/pointer assignments

Variations on Three-Address Code

- label scheme – we use *L*: instructions for jumps.
- temporary register management – we write the explicit type when needed.

Example: some label scheme

```
do i = i+1; while (a[i] < v);
```

$$
\begin{array}{rl}
L: & t_1 = i + 1 \\
 & i = t_1 \\
 & t_2 = i * 8 \\
 & t_3 = a[t_2] \\
 & \mathit{if}\ t_3 < v\ \mathit{goto}\ L
\end{array}
\qquad
\begin{array}{rl}
100: & t_1 = i + 1 \\
101: & i = t_1 \\
102: & t_2 = i * 8 \\
103: & t_3 = a[t_2] \\
104: & \mathit{if}\ t_3 < v\ \mathit{goto}\ 100
\end{array}
$$

Intermediate Operators

Choice of operator set:

- Rich enough to implement the operations of the source language.
- Close enough to machine instructions to simplify code generation.

So far, we have deployed the operators from the source language (grammar operands). We could, e.g., use operator ‘inc’ instead of ‘+’ through additional graph/code conversion.

Data representation of Three-Address Instructions

What are the canonical data structures for representing the instructions?

- Quadruples.
- Triples.
- Indirect triples.

Quadruples

A quadruple data structure has the characteristics:

- Has four fields: op, arg1, arg2, result.
- Exceptions:
  1. Unary operators: no arg2.
  2. operators like *param*: no arg2, no result.
  3. (Un)conditional jumps: target label is the result.

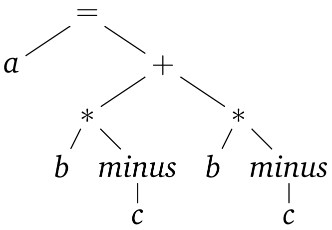
Example: Quadruples

$$t_1 = minus\ c$$
$$t_2 = b * t_1$$
$$t_3 = minus\ c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

	OP	ARG1	ARG2	RESULT
0	*minus*	c		t_1
1	$*$	b	t_1	t_2
2	*minus*	c		t_3
3	$*$	b	t_3	t_4
4	$+$	t_2	t_4	t_5
5	$=$	t_5		a

Triples

- Has three fields: op, arg1, arg2. NO result field!
- Results referred to by their position.



	OP	ARG1	ARG2
0	*minus*	c	
1	$*$	b	(0)
2	*minus*	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)

Indirect triples

- Triples more compact/efficient representation than quadruples.
- When instructions are moving around during optimizations: quadruples better than triples.
- Indirect triples have both advantages.

	INSTRUCTION
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	OP	ARG1	ARG2
0	*minus*	*c*	
1	$*$	*b*	(0)
2	*minus*	*c*	
3	$*$	*b*	(2)
4	$+$	(1)	(3)
5	$=$	*a*	(4)

Static Single-Assignment Form

Helps cetain code optimizations.

Every distinct assignment must be to a distinct temporary:

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

is changed to

$$\texttt{if (f) } x_1 = 1;\ \texttt{else } x_2 = 2;\ x_3 = \phi(x_1, x_2);\ x_4 = x_3 * a;$$

SDD: expression translations

PRODUCTION	RULES
$S \to \textbf{id} = E_1\ ;$	$S.code = E_1.code \parallel [\![\textbf{id}' =' E_1.addr]\!]$
$E \to E_1 + E_2$	$E_1.e = E.addr = \text{newTemp}()$ $E.code = E_1.code \parallel E_2.code \parallel [\![E.addr = E_1.addr + E_2.addr]\!]$
$\mid -E_1$	$E.addr = \text{newTemp}()$ $E.code = E_1.code \parallel [\![E.addr = -E_1.addr]\!]$
$\mid (E_1)$	$E.addr = E_1.addr; E.code = E_1.code$
$\mid \textbf{id}$	$E.addr = \textbf{id}; E.code = [\![\,]\!]$

- where $[\![\ldots]\!]$ builds the instruction for ...,
- $E.addr$, $S.code$, and $E.code$ are synthesized attributes.

Translation scheme variation

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

This depends on the evaluation order of semantic rules.

Translation scheme variation

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

This depends on the evaluation order of semantic rules.

Array "flattening"

- 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:

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

Array "flattening"

- 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:

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

Array "flattening"

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

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

Conditionals

$\texttt{if}\ (B_1)$
S_2
$\texttt{else}\ S_3$

$\texttt{ifFalse}\ \boxed{B_1}\ \texttt{goto}\ L_3$
$\boxed{S_2}$
$\texttt{goto}\ L_2$
$L_3\texttt{:}$
$\boxed{S_3}$
$L_2\texttt{:}$

Conditionals

```
if (B₁)
S₂
else S₃
```

$$
\begin{array}{ll}
 & \texttt{ifFalse}\ \boxed{B_1}\ \texttt{goto}\ L_3 \\
 & \boxed{S_2} \\
 & \texttt{goto}\ L_2 \\
L_3\texttt{:} & \\
 & \boxed{S_3} \\
L_2\texttt{:} &
\end{array}
$$

Conditionals, Example

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

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

Conditionals, Example

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

$\quad$ $\mathtt{t}_1$ = a + 1
$\quad$ ifFalse t > b goto L_3
$\quad$ $\boxed{S_2}$
$\quad$ goto L_2
L_3:
$\quad$ $\boxed{S_3}$
L_2:

Loops

`while` (B_1)
S_2

`goto` L_2
L_1:
$\boxed{S_2}$
L_2:
`if` $\boxed{B_1}$ `goto` L_1

Loops

`while` (B_1)
S_2

`goto` L_2
L_1:
$\boxed{S_2}$
L_2:
`if` $\boxed{B_1}$ `goto` L_1

SDD for flow control translation (if and while)

PRODUCTION	SEMANTIC RULE
$P \to S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \to \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \to \textbf{if}\,(B)\,S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$

SDD for flow control translation (if and while)

Production	Semantic Rule
$S \rightarrow \textbf{if}\,(B)\,S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' S.next) \parallel label(B.false) \parallel S_2.code$

SDD for flow control translation (if and while)

PRODUCTION	SEMANTIC RULE
$S \rightarrow \textbf{while}\ (B)\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\quad \parallel label(B.true) \parallel S_1.code \parallel gen('goto' begin)$

SDD for flow control translation (if and while)

Production	Semantic Rule
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

Booleans

Boolean expressions alters the flow of control:

$$B \rightarrow B \,\|\, B \;\mid\; B \,\&\&\, B \;\mid\; !\,B \;\mid\; (B) \;\mid\; E \;\textbf{rel}\; E \;\mid\; \textbf{true} \;\mid\; \textbf{false}$$

where **rel** is $<$, $<=$, $>$, $>=$, $=$, $!=$

- Boolean operators: $\&\&$ higher precedence than $\|$.
- Mathematically: $\&\&$ and $\|$ are associative.
- Evaluation wise: "associates" to the left.

Control flow: short-circuit

The operators of a boolean expression does not appear explicitly in the code.

Example: if $(x < 100 \;||\; x > 200 \;\&\&\; x! = y)\; x = 0$

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:  x=0
L1:
```

Control flow: short-circuit

The operators of a boolean expression does not appear explicitly in the code.

Example: if $(x < 100 \mathbin{||} x > 200 \mathbin{\&\&} x! = y)\ x = 0$

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:  x=0
L1:
```

SDD for flow control translation (booleans)

PRODUCTION	SEMANTIC RULE
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

SDD for flow control translation (booleans)

PRODUCTION	SEMANTIC RULE
$B \rightarrow B_1 \mathbin{\&\&} B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.true) \parallel B_2.code$

SDD for flow control translation (booleans)

PRODUCTION	SEMANTIC RULE
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$

SDD for flow control translation (booleans)

PRODUCTION	SEMANTIC RULE
$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$B.code = E_1.code \parallel E_2.code$ $\parallel gen('if'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ 'goto'\ B.true)$ $\parallel gen('goto'\ B.false)$

SDD for flow control translation (booleans)

PRODUCTION	SEMANTIC RULE
$B \rightarrow \mathbf{true}$	$B.code = gen('goto'\ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('goto'\ B.false)$

Calls

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

$E_1.code$
...
$E_n.code$
param $E_1.addr$
...
param $E_n.addr$
x = call f

If parameters are passed **call-by-value** then the *contract* is: $E_1.code, \ldots, E_n.code$ are evaluated before results placed into temporary $E_1.addr, \ldots, E_n.addr$.

Calls

$x = f(E_1, \ldots, E_n)$	$E_1.code$ $\ldots$ $E_n.code$ `param` $E_1.addr$ $\ldots$ `param` $E_n.addr$ `x = call` f

If parameters are passed **call-by-value** then the *contract* is: $E_1.code, \ldots, E_n.code$ are evaluated before results placed into temporary $E_1.addr, \ldots, E_n.addr$.

SDD: expression translations in context

PRODUCTION	RULES
$S \rightarrow \textbf{id} = E_1\ ;\ S_2$	$S.code = E_1.code \parallel [\![\textbf{id}' =' E_1.addr]\!] \parallel S_2.code$
$\mid \epsilon$	$S.code = [\![\ ]\!]$
$E \rightarrow E_1 + E_2$	$E_1.e = E.addr = \text{newTemp}()$ $E.code = E_1.code \parallel E_2.code \parallel [\![E.addr = E_1.addr + E_2.addr]\!]$
$\mid - E_1$	$E.addr = \text{newTemp}()$ $E.code = E_1.code \parallel [\![E.addr = -E_1.addr]\!]$
$\mid (E_1)$	$E.addr = E_1.addr; E.code = E_1.code$
$\mid \textbf{id}$	$E.addr = \textbf{id}; E.code = [\![\ ]\!]$

SDD: expression translations with environments

Production	Rules
$S \to \mathbf{id} = E_1 \; ; \; S_2$	$E_1.e = S.e; S_2.e = S.e; S.c = E_1.c \parallel [\![\mathbf{id} = E_1.a]\!] \parallel S_2.c$
$\mid \epsilon$	$S.c = [\![\,]\!]$
$E \to E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e; E.a = \text{newTemp}$ $E.c = E_1.c \parallel E_2.c \parallel [\![E.a = E_1.a + E_2.a]\!]$
$\mid - E_1$	$E_1.e = E.e; E.a = \text{newTemp}$ $E.c = E_1.c \parallel [\![E.a = -E_1.a]\!]$
$\mid (E_1)$	$E_1.e = E.e; E.a = E_1.a; E.c = E_1.c$
$\mid \mathbf{id}$	$E.a = \mathbf{id}; E.c = [\![\,]\!]$

with inherited environments $S.e$ and $E.e$; attributes $S.c$ abbreviation for $S.code$, $E.c$ for $E.code$, and $E.a$ for $E.addr$.

Questions?