7. Intermediate Code Generation

Eva Rose  Kristoffer Rose

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

March 23, 2015
1 Introduction
2 Directed Acyclic Graphs
3 Three-Address Code
4 Translations of Computation Expressions
5 Control Flow
6 Procedure Calls
7 Translations of Arrays
Introduction

Directed Acyclic Graphs

Three-Address Code

Translations of Computation Expressions

Control Flow

Procedure Calls

Translations of Arrays
Fourth Compilation Phase

source program

Tokens

Lexical Analysis

Syntax Analysis

Symbol Table

Tree

Semantic Analysis

Intermediate Representation Generator

Tree

IR

Optimizer

Code Generator

Machine Code

Target machine code

Machine-Dependent Code Optimizer
IR generator: front-end bordering back-end

Front end

Back end

\[ \text{IR code generator} \]

\[ \text{Code Generator} \]

\[ \text{getNextTree} \]

\[ \text{getNextIR} \]

\[ m \times n \] compilers can be built by writing just \( m \) front-ends and \( n \) back-ends.
Canonical example

Back to the introductory example...
Example Code

```c
int initial = 32;
float rate = .8;
float position = initial + rate * 60;
```
Example Abstract Syntax Tree (AST)

```
int =
   \langle id, 2 \rangle
   \langle num, 32 \rangle

float =
   \langle id, 3 \rangle
   \langle num, .8 \rangle

float =
   \langle id, 1 \rangle
   \langle id, 2 \rangle
   \langle id, 3 \rangle
   \langle num, 60 \rangle

\epsilon
```

<table>
<thead>
<tr>
<th>id</th>
<th>lexeme</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>position</td>
<td>float</td>
</tr>
<tr>
<td>2</td>
<td>initial</td>
<td>int</td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
<td>float</td>
</tr>
</tbody>
</table>

Eva Rose, Kristoffer Rose
Compiler Construction (CSCI-GA.2130-001) 7. Intermediate Code Generation
March 23, 2015 8 / 61
Example Intermediate Representation (code)

```plaintext
int    t_1 = 32
int    initial = t_1
float  t_2 = .8
float  rate = t_2
int    t_3 = initial
float  t_4 = rate
int    t_5 = 60
float  t_6 = (float) t_3
float  t_7 = t_4 * t_6
float  t_8 = t_6 + t_7
float  position = t_8
```
Intermediate representation

There are essentially 2 steps:

- High level IR (DAG tree)
- Low level IR (Three-address code)
## Syntax trees

Recall AST construction of simple expressions (lecture 5):

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $E \rightarrow E_1 + T_2$</td>
<td>$E$.node = new Node('+', $E_1$.node, $T_2$.node)</td>
</tr>
<tr>
<td>2. $E \rightarrow E_1 - T_2$</td>
<td>$E$.node = new Node('-', $E_1$.node, $T_2$.node)</td>
</tr>
<tr>
<td>3. $E \rightarrow E_1 * T_2$</td>
<td>$E$.node = new Node('*', $E_1$.node, $T_2$.node)</td>
</tr>
<tr>
<td>4. $E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>5. $T \rightarrow (E)$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>6. $T \rightarrow id$</td>
<td>$T$.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>7. $T \rightarrow num$</td>
<td>$T$.node = new Leaf(num, num.entry)</td>
</tr>
</tbody>
</table>

- Draw up an AST for $a + a * (b - c) + (b - c) * d$
AST for $a + a \ast (b - c) + (b - c) \ast d$
DAG for $a + a \times (b - c) + (b - c) \times d$
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:

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T_2$</td>
<td>$E.node = \textbf{new} \ Node(\ '+', E_1.node, T_2.node)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 - T_2$</td>
<td>$E.node = \textbf{new} \ Node(\ '-', E_1.node, T_2.node)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \ast T_2$</td>
<td>$E.node = \textbf{new} \ Node(\ '*', E_1.node, T_2.node)$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.node = T.node$</td>
</tr>
<tr>
<td>$T \rightarrow (E)$</td>
<td>$T.node = E.node$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T.node = \textbf{new} \ Leaf(\text{id}, \text{id}.entry)$</td>
</tr>
<tr>
<td>$T \rightarrow \text{num}$</td>
<td>$T.node = \textbf{new} \ Leaf(\text{num}, \text{num}.entry)$</td>
</tr>
</tbody>
</table>

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: \( a + a \times (b - c) + (b - c) \times d \)

\[
\begin{align*}
p_1 &= \text{Leaf}(id, entry-a) \\
p_2 &= \text{Leaf}(id, entry-a) = p_1 \\
p_3 &= \text{Leaf}(id, entry-b) \\
p_4 &= \text{Leaf}(id, entry-c) \\
p_5 &= \text{Node}(', p_3, p_4) \\
p_6 &= \text{Node}('+', p_1, p_5) \\
p_7 &= \text{Node}('+', p_1, p_6) \\
p_8 &= \text{Leaf}(id, entry-b) = P_3 \\
p_9 &= \text{Leaf}(id, entry-c) = P_4 \\
p_{10} &= \text{Node}(', p_3, p_4) = P_5 \\
p_{11} &= \text{Leaf}(id, entry-d) \\
p_{12} &= \text{Node}('+', p_5, p_{11}) \\
p_{13} &= \text{Node}('+', p_5, p_{12})
\end{align*}
\]
Introduction Directed Acyclic Graphs Three-Address Code Translations of Computation Expressions Control Flow Procedure Calls Translations of Arrays
A Value Graph (DAG)

\[
\begin{align*}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{align*}
\]
A Value Graph (DAG) and Code

\[ t_1 = b - c \]
\[ t_2 = a \times t_1 \]
\[ t_3 = a + t_2 \]
\[ t_4 = t_1 \times d \]
\[ t_5 = t_3 + t_4 \]
A Value Graph (DAG) and Code

\[ \begin{align*}
\text{\texttt{t1}} &= \text{\texttt{b}} - \text{\texttt{c}} \\
\text{\texttt{t2}} &= \text{\texttt{a}} \times \text{\texttt{t1}} \\
\text{\texttt{t3}} &= \text{\texttt{a}} + \text{\texttt{t2}} \\
\text{\texttt{t4}} &= \text{\texttt{t1}} \times \text{\texttt{d}} \\
\text{\texttt{t5}} &= \text{\texttt{t3}} + \text{\texttt{t4}}
\end{align*} \]
A Value Graph (DAG) and Code

\[ \begin{align*}
  t_1 &= b - c \\
  t_2 &= a \times t_1 \\
  t_3 &= a + t_2 \\
  t_4 &= t_1 \times d \\
  t_5 &= t_3 + t_4 
\end{align*} \]
A Value Graph (DAG) and Code

A directed acyclic graph (DAG) and corresponding code for a simple arithmetic expression. The expression is:

\[ t_5 = t_3 + t_4 \]

with:

- \( t_1 = b - c \)
- \( t_2 = a \times t_1 \)
- \( t_3 = a + t_2 \)
- \( t_4 = t_1 \times d \)
- \( t_5 = t_3 + t_4 \)

The DAG shows the dependencies and order of operations. The nodes represent intermediate values, and the edges indicate the computation steps.
A Value Graph (DAG) and Code

\[
\begin{align*}
\text{\texttt{t1}} &= \texttt{b} \ - \ \texttt{c} \\
\text{\texttt{t2}} &= \texttt{a} \ * \ \texttt{t1} \\
\text{\texttt{t3}} &= \texttt{a} \ + \ \texttt{t2} \\
\text{\texttt{t4}} &= \texttt{t1} \ * \ \texttt{d} \\
\text{\texttt{t5}} &= \texttt{t3} \ + \ \texttt{t4}
\end{align*}
\]
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) e.g., $x, y, z$

Constant  (with explicit primitive type) e.g., 1, 2, ’a’, ’b’,

Compiler-generated temporary  (“register”) e.g., $t_1, t_2, t_3$
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

\[
\text{do } i = i + 1; \text{ while } (a[i] < v); \\
\]

\[
L:\quad t_1 = i + 1 \\
i = t_1 \\
t_2 = i \times 8 \\
t_3 = a[t_2] \\
\text{if } t_3 < v \text{ goto } L
\]

\[
100:\quad t_1 = i + 1 \\
101:\quad i = t_1 \\
102:\quad t_2 = i \times 8 \\
103:\quad t_3 = a[t_2] \\
104:\quad \text{if } t_3 < v \text{ goto } 100
\]
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

<table>
<thead>
<tr>
<th>OP</th>
<th>ARG1</th>
<th>ARG2</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>c</td>
<td>t1</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>b</td>
<td>t1</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>c</td>
<td>t3</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>b</td>
<td>t3</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>t2</td>
<td>t4</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>t5</td>
<td>a</td>
</tr>
</tbody>
</table>
**Triples**

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

![Diagram of a tree expression]

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>(1)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>a</td>
<td>(4)</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>OP</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>0</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>2</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>3</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>4</td>
<td>(1)</td>
<td>(3)</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>a</td>
<td>(4)</td>
</tr>
</tbody>
</table>
Static Single-Assignment Form

Helps certain code optimizations.

Every distinct assignment must be to a distinct temporary:

\[
\text{if (f) } x = 1; \text{ else } x = 2; \quad y = x \ast a;
\]

is changed to

\[
\text{if (f) } x_1 = 1; \text{ else } x_2 = 2; \quad x_3 = \phi(x_1, x_2); \quad y = x_3 \ast a;
\]
1. Introduction
2. Directed Acyclic Graphs
3. Three-Address Code
4. Translations of Computation Expressions
5. Control Flow
6. Procedure Calls
7. Translations of Arrays
Example Code with Expression

```c
int initial = 32;
float rate = .8;
float position = initial + rate * 60;
```
Example Intermediate Representation with Code for Expression

```plaintext
int t1 = 32
int initial = t1
float t2 = .8
float rate = t2
int t3 = initial
float t4 = rate

int t5 = 60
float t6 = (float) t3
float t7 = t4 * t6
float t8 = t6 + t7
```

```
float position = t8
```

(E.addr)
Code Generation Patterns: Expressions

Expression $E$

Value is in (synthesized) address $E.addr$.

$E \rightarrow E_1 + E_2$

newTemp = $E_1.addr + E_2.addr$

$E.addr$
**Code Generation Patterns: Expressions**

Expression: $E$

Value is in (synthesized) address $E.addr$.

$E \rightarrow E_1 + E_2$

$E_1$

$E_2$

newTemp = $E_1.addr + E_2.addr$

$E.addr$
# SDD: expression translations

<table>
<thead>
<tr>
<th>Production</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \mathbf{id} = E_1 ;$</td>
<td>$S.code = E_1.code</td>
</tr>
</tbody>
</table>
| $E \rightarrow E_1 + E_2$ | $E.addr = \text{newTemp}()$  
| | $E.code = E_1.code || E_2.code || \text{gen}(E.addr \leftarrow E_1.addr + E_2.addr)$ |
| | $-E_1$ | $E.addr = \text{newTemp}()$  
| | | $E.code = E_1.code || \text{gen}(E.addr \leftarrow -E_1.addr)$ |
| | $(E_1)$ | $E.addr = E_1.addr; E.code = E_1.code$ |
| | $\mathbf{id}$ | $E.addr = \mathbf{id}; E.code = \epsilon$ |

- where $\text{gen}(\ldots)$ builds the instruction for $\ldots$,  
- $E.addr$, $S.code$, and $E.code$ are synthesized attributes.
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 (in book)

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.
Introduction

Directed Acyclic Graphs

Three-Address Code

Translations of Computation Expressions

Control Flow

Procedure Calls

Translations of Arrays
## Statement Generation as SDD

<table>
<thead>
<tr>
<th><strong>Production</strong></th>
<th><strong>Semantic Rule</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \rightarrow S )</td>
<td>( S\text{.next} = \text{newLabel()} )&lt;br&gt;( P\text{.code} = S\text{.code} \parallel \text{label}(S\text{.next}) \parallel \text{gen('halt')} )</td>
</tr>
<tr>
<td>( S \rightarrow S_1 \ S_2 )</td>
<td>( S_1\text{.next} = \text{newLabel()} )&lt;br&gt;( S_2\text{.next} = S\text{.next} )&lt;br&gt;( S\text{.code} = S_1\text{.code} \parallel \text{label}(S_1\text{.next}) \parallel S_2\text{.code} )</td>
</tr>
</tbody>
</table>

Label attribute \( S\text{.next} \) is inherited.
Code Generation Patterns: Statements

Statement

\[ S \]

\[ S.next \]

newLabel:

Subsequent label \( S.next \) is \textit{inherited} into statement code from the context.
Code Generation Patterns: Statements

Subsequent label $S.next$ is *inherited* into statement code from the context.
Conditionals, Intuition

\[
\begin{align*}
\text{if} & \quad (B_1) \\
& \quad \text{else } S_3 \\
S_2 & \\
L_3: & \\
L: & \\
\text{ifFalse} & \quad B_1 \quad \text{goto} \quad L_3 \\
& \quad S_2 \quad \text{goto} \quad L \\
& \quad L_3: \quad S_3 \\
& \quad L:
\end{align*}
\]
**Conditionals, Intuition**

```
if (B₁)
  S₂
else
  S₃
```

```
ifFalse B₁ goto L₃
S₂
goto L
L₃:
  S₃
L:
```
Loops, Intuition

```
while \((B_1)\)
```

```
L_1:
S_2

L_2:
if \(B_1\) goto L_1
```
Loops, Intuition

\[ \text{while } (B_1) \]

\[
\begin{aligned}
L_1 & : \\
S_2 & \\
L_2 & : \\
& \text{if } B_1 \text{ goto } L_1
\end{aligned}
\]
Conditionals, Example

if ((a+1)>b)
S₂
else S₃

\[ t₁ = a + 1 \]
\[ \text{ifFalse } t₁ > b \text{ goto } L₃ \]
\[ S₂ \]
\[ \text{goto } L \]
\[ L₃: \]
\[ S₃ \]
\[ L: \]

Some of the code of \( B₁ \) is before the test!
Conditionals, Example

\[
\begin{align*}
t_1 &= a + 1 \\
\text{ifFalse } t_1 &> b \text{ goto } L_3 \\
\text{if } ((a+1)>b) \\
S_2 & \\
\text{else } S_3 & \\
L_3: & \\
S_3 & \\
L: & \\
\end{align*}
\]

Some of the code of \(B_1\) is before the test!
Conditionals, Example

```
t_1 = a + 1
ifFalse t_1 > b goto L_3
```

```
if ((a+1)>b)
  S_2
else S_3
  L_3:
    S_3
L:
```

Some of the code of \( B_1 \) is before the test!
True and False target labels are *inherited* into test code.
Code Generation Patterns: Boolean Tests

True and False target labels are *inherited* into test code.
Code Generation, Conditional

\[ S \rightarrow \text{if} (B) \ S_1 \ \text{else} \ S_2 \]
Code Generation, Conditional

\[ S \rightarrow \text{if } (B) \text{ else } S_1 \]

\[ S \rightarrow S_2 \]

- \( B \) is the condition.
- \( S_1 \) and \( S_2 \) are the branches.
- \( B\.true \) and \( B\.false \) are the outcomes.
- \( S_1\.next \) and \( S_2\.next \) are the next statements.
- \( \text{newLabel} : \) indicates new label creation.

The diagram illustrates the flow of control and the generation of code for a conditional statement.
**SDD: flow control translation (if)**

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULE</th>
</tr>
</thead>
</table>
| $S \rightarrow \textbf{if} (B) S_1$ | $B.true = \text{newLabel}()$
$B.false = S_1.next = S.next$
$S.code = B.code \parallel \text{label}(B.true) \parallel S_1.code$ |
| $S \rightarrow \textbf{if} (B) S_1 \textbf{else} S_2$ | $B.true = \text{newLabel}()$
$B.false = \text{newLabel}()$
$S_1.next = S_2.next = S.next$
$S.code = B.code \parallel \text{label}(B.true) \parallel S_1.code$
$\parallel \text{gen('goto' S.next)} \parallel \text{label}(B.false) \parallel S_2.code$ |
### SDD: flow control translation (while)

<table>
<thead>
<tr>
<th><strong>Production</strong></th>
<th><strong>Semantic Rule</strong></th>
</tr>
</thead>
</table>
| \( S \rightarrow \text{while} (B) S_1 \) | \[
begin = \text{newLabel}() \\
B.true = \text{newLabel}() \\
B.false = S.next \\
S_1.next = begin \\
S.code = \text{label}(\text{begin}) \parallel B.code \\
\parallel \text{label}(B.true) \parallel S_1.code \parallel \text{gen('goto' begin)}
\] |
Boolean expressions alters the flow of control:

\[ B \rightarrow B \lor B \mid B \&\& B \mid \neg B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false} \]

where rel is \(<\), \(\leq\), \(>\), \(\geq\), \(=\), \(!=\)

- Boolean operators: \&\& higher precedence than \lor.
- Mathematically: \&\& and \lor 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 \((x < 100 \, \| \, x > 200 \, \&\& \, x \neq y)\) \(x = 0;\)

```c
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 \((x < 100 \; \mid \; x > 200 \; \&\& \; x \neq y)\) \(x = 0\);

\[
\begin{align*}
\text{if } x < 100 & \; \text{ goto } L_2 \\
\text{ifFalse } x > 200 & \; \text{ goto } L_1 \\
\text{ifFalse } x \neq y & \; \text{ goto } L_1 \\
\end{align*}
\]

\(L_2: \; x = 0\)

\(L_1:\)
## SDD for flow control translation (booleans)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $B \rightarrow B_1 \parallel B_2$ | $B_1.\text{true} = B.\text{true}$  
$B_1.\text{false} = \text{newLabel}()$  
$B_2.\text{true} = B.\text{true}$  
$B_2.\text{false} = B.\text{false}$  
$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$ |
### SDD for flow control translation (booleans)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $B \rightarrow B_1 \&\& B_2$ | $B_1.true = \text{newLabel}()$
| | $B_1.false = B.false$
| | $B_2.true = B.true$
| | $B_2.false = B.false$
| | $B.code = B_1.code \parallel \text{label}(B_1.true) \parallel B_2.code$ |
### SDD for flow control translation (booleans)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $B \rightarrow !B_1$ | $B_1.\text{true} = B.\text{false}$  
$B_1.\text{false} = B.\text{true}$  
$B.\text{code} = B_1.\text{code}$ |
## SDD for flow control translation (booleans)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \rightarrow E_1 \text{ rel } E_2$</td>
<td>$B.code = E_1.code \parallel E_2.code \parallel gen('if' E_1.addr \text{ rel } op E_2.addr 'goto' B.true) \parallel gen('goto' B.false)$</td>
</tr>
</tbody>
</table>
# SDD for flow control translation (booleans)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \rightarrow \text{true}$</td>
<td>$B.code = \text{gen}('goto' B.true)$</td>
</tr>
<tr>
<td>$B \rightarrow \text{false}$</td>
<td>$B.code = \text{gen}('goto' B.false)$</td>
</tr>
</tbody>
</table>
Introduction

Directed Acyclic Graphs

Three-Address Code

Translations of Computation Expressions

Control Flow

Procedure Calls

Translations of Arrays
Calls

\[ x = f(E_1, \ldots, E_n) \]

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

\[ x = f(E_1, \ldots, E_n) \]

\[ E_1.code \]
\[ \ldots \]
\[ E_n.code \]
\[ \text{param } E_1.addr \]
\[ \ldots \]
\[ \text{param } E_n.addr \]
\[ x = \text{call } f \]

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 \( E_1.addr, \ldots, E_n.addr \).
Introduction

Directed Acyclic Graphs

Three-Address Code

Translations of Computation Expressions

Control Flow

Procedure Calls

Translations of Arrays
Array “flattening”

- One Dimension:

\[ \text{addr} = \text{base} + i \times w \]

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

\[ \text{addr} = \text{base} + (i_1 \times n_2 + i_2) \times w \]

- \(k\) dimensions, row-major:

\[ \text{addr} = \text{base} + (\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w \]
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:

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

\[ addr = base + (((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w \]
## Array Translation SDD

<table>
<thead>
<tr>
<th>PRODUCTIONS</th>
<th>RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E_1$ ;</td>
<td>$S.c = E_1.c \parallel \text{gen}(	ext{id} = E.a)$;</td>
</tr>
<tr>
<td>\hline</td>
<td>$S.c = L_1.c \parallel E_2.c \parallel \text{gen}(L_1.base [ L_1.a ] = E_2.a)$;</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E.a = \text{newTemp}(); E.c = E_1.c \parallel E_2.c \parallel \text{gen}(E.a = E_1.a + E_2.a)$;</td>
</tr>
<tr>
<td>\hline</td>
<td>$E.a = \text{id}; E.c = \epsilon$</td>
</tr>
<tr>
<td>\hline</td>
<td>$E.a = \text{newTemp}(); E.c = L_1.c \parallel \text{gen}(E.a = L_1.base [ L_1.a ])$;</td>
</tr>
<tr>
<td>$L \rightarrow \text{id} [ E_1 ]$</td>
<td>$L.base = \text{id}; L.t = \text{id.type}$;</td>
</tr>
<tr>
<td>\hline</td>
<td>$L.a = \text{newTemp}(); L.c = E_1.c \parallel \text{gen}(L.a = E_1.a \ast L.t.width)$;</td>
</tr>
<tr>
<td>\hline</td>
<td>$L.base = L_1.base; L.t = \text{shift}(L_1.t)$;</td>
</tr>
<tr>
<td>\hline</td>
<td>$t = \text{newTemp}(); L.a = \text{newTemp}();$</td>
</tr>
<tr>
<td>\hline</td>
<td>$L.c = L_1.c \parallel E_1.c \parallel \text{gen}(t = L_1.a \ast L_1.t.length) \parallel \text{gen}(L.a = t + E_1.a)$;</td>
</tr>
</tbody>
</table>

Note: we abbreviate code, type, and addr, as c, t, and a, respectively.
Questions?