Intermediate Representations

The Glue That Holds a Compiler Together

Comp 412
Taxonomy of Intermediate Representations

Three major categories

• Structural IRs
  – Graphically oriented
  – Heavily used in source-to-source translators
  – Tend to be large

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

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
- SSA Form
Several different representations of three address code

• In general, three address code has statements of the form:

\[ x \leftarrow y \ op \ z \]

With 1 operator (\( \text{op} \)) and, at most, 3 names (\( x, y, \) & \( z \))

Example:

\[ z \leftarrow x \ 2 \ast y \]

becomes

\[ t \leftarrow 2 \ast y \]
\[ z \leftarrow x - t \]

Advantages:

• Resembles many real machines
• Introduces a new set of names
• Compact form

See Lab 1 and Lab 3

The concept of “three address code” has many implementations.
Three Address Code: As Quadruples

Naïve representation of three address code

- Table of \(k \times 4\) small integers
- Simple record structure
- Easy, albeit slow, to reorder
- Explicit names

<table>
<thead>
<tr>
<th>Opcode</th>
<th>(Op_1)</th>
<th>(Op_2)</th>
<th>(Op_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

RISC assembly code

store opcode & operands as small integers, of course.
Three Address Code: As Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<table>
<thead>
<tr>
<th>Implicit Name</th>
<th>Opcode</th>
<th>Op₁</th>
<th>Op₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>loadI</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>(4)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Remember, for a long time, 640 KB was a lot of RAM
Three Address Code: As Indirect Triples

- List first triple in each statement
- Implicit name space for statements
- Uses more space than triples, but easier to reorder

<table>
<thead>
<tr>
<th>Stmt List</th>
<th>Implicit Names</th>
<th>Indirect Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100)</td>
<td>(100)</td>
<td>load y</td>
</tr>
<tr>
<td>(105)</td>
<td>(101)</td>
<td>loadI 2</td>
</tr>
<tr>
<td></td>
<td>(102)</td>
<td>mult (100) (101)</td>
</tr>
<tr>
<td>(103)</td>
<td></td>
<td>load x</td>
</tr>
<tr>
<td>(104)</td>
<td></td>
<td>sub (103) (102)</td>
</tr>
</tbody>
</table>

Standard trick: when you need the ability to rearrange objects in memory, add a level of indirection.

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important
BEGIN INTEGER K;
  ARRAY A[1:I-J];
  K := 0;
L: IF I > J
  THEN K := K + A[I-J]*6
  ELSE BEGIN I := I+1; I := I+1; GO TO L END
END

FIGURE 11.1. An ALGOL Program Segment.

(1) BLOCK
(2)  - I, J
(3)  BOUNDS 1, (2)
(4)  ADEC A
(5)  := 0, K
(6)  - I, J
(7)  BMZ (13), (6)
(8)  - I, J
(9)  * A[(8)], 6
(10) + K, (9)
(11) := (10), K
(12) BR (18)
(13) + I, 1
(14) := (13), I
(15) + I, 1
(16) := (15), I
(17) BRL L
(18) BLCEND

FIGURE 11.7. Triples for Program Segment of Figure 11.1.

BEGIN INTEGER K;
ARRAY A[1:I-J];
K := 0;
L: IF I > J
THEN K := K + A[I-J]*6
ELSE BEGIN I := I+1; I := I+1; GO TO L END
END

FIGURE 11.1. An ALGOL Program Segment.

1. (1) 10. (8)  
2. (2) 11. (9)  
2. (3) 12. (10)  
4. (4) 13. (11)  
5. (5) 14. (12)  
6. (2) 15. (11)  
7. (6) 16. (12)  
8. (2) 17. (13)  
9. (7) 18. (14)  

(1) BLOCK  
(2) - I, J  
(3) BOUNDS 1, (2)  
(4) ADEC A  
(5) := 0, K  
(6) BMZ 13, (2)  
(7) * A[ (2) ], 6  
(8) + K, (7)  
(9) := (8), K  
(10) BR 18  
(11) + I, 1  
(12) := (11), I  
(13) BRL L  
(14) BLCKEND

FIGURE 11.8. Indirect Triples for Figure 11.1.

Two-Address Code

Two-address code allows statements of the form

\[ x \leftarrow x \, op \, y \]

Each operations has 1 operator \((op)\) and, at most, 2 names \((x \text{ and } y)\)

Example

\[
\begin{align*}
z & \leftarrow x - 2 \times y \\
\text{becomes} & \\
\end{align*}
\]

\[
\begin{align*}
\text{We write:} & \\
r1 + r2 & \Rightarrow r2 \\
\text{as:} & \\
\text{add } r1, r2 & \\
\end{align*}
\]

- Can be very compact

Problems

- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11, x86)
- We would like destructive operations to become a thing of the past

Not many arguments in favor of two-address code
Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
  \[ x - 2 * y \] becomes

  | push x |
  | push 2 |
  | push y |
  | multiply |
  | subtract |

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted over slow communication links or where memory is limited

- Java bytecode was designed for transmission over slow links
- Follows a long line of bytecode-like IRs designed to be compact

In a stack machine, most operations are destructive.
**Taxonomy of Intermediate Representations**

**Three major categories**

- **Structural IRs**
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large
  
  **Examples:**
  - Trees, DAGs

- **Linear IRs**
  - 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 IRs**
  - Combination of graphs and linear code
  - Example: control-flow graph

  **Examples:**
  - Control-flow graph
  - SSA Form
Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

Edges represent the branches & jumps at the ends of blocks.

Basic blocks: Maximal length sequences of straightline code

Implementations:
See Figures B.3 and B.4 in Appendix B of EaC2e
The Main Idea: each name defined by exactly one operation

• Introduce $\phi$-functions to make it work

**Original**

\[
\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\text{while } (x < k) & \\
x & \leftarrow x + 1 \\
y & \leftarrow y + x
\end{align*}
\]

**SSA-form**

\[
\begin{align*}
x_0 & \leftarrow \ldots \\
y_0 & \leftarrow \ldots \\
\text{if } (x_0 \geq k) & \text{ goto next} \\
\text{loop: } & \\
x_1 & \leftarrow \phi(x_0, x_2) \\
y_1 & \leftarrow \phi(y_0, y_2) \\
x_2 & \leftarrow x_1 + 1 \\
y_2 & \leftarrow y_1 + x_2 \\
\text{if } (x_2 < k) & \text{ goto loop} \\
\text{next: } & \\
& \ldots
\end{align*}
\]

**Strengths of SSA-form**

• Sharper analysis

• $\phi$-functions give hints about placement

• (sometimes) faster algorithms

SSA is often interpreted as a graph, with edges running from def to use.
Combination of IRs

Some compilers use a CFG to represent control flow and a linear IR to represent code in blocks

• This hybrid IR with the advantages of a graph
  – Easy navigation between blocks

• And the advantages of a linear IR
  – Explicit, low-level detail & operation sequence

Strengths

• Good for understanding control-flow issues
• Good for understanding flow of data  
  (program analysis)

In lab 3, you will build and use a dependence graph, linked to a basic block. The dependence graph records the flow of values, while the block records both the details of those operations & their relative order.
Using Multiple Representations

- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations

- Example: the Open64 compiler
  - WHIRL intermediate format
    - Consists of 5 different IRs that are progressively more detailed and less abstract
    - Each successive IR focuses on a different set of challenges & opportunities
  - Translation is a monotonic lowering of level of abstraction
    - Compilers are good at lowering level of abstraction & not so good at raising it
Memory Models

An IR usually incorporates a memory model, explicit or implicit.

• **Register-to-register model**
  – Keep all values that can legally be stored in a register in registers
  – Ignore machine limitations on number of registers
  – Compiler back-end must insert loads and stores

• **Memory-to-memory model**
  – Keep all values in memory
  – Only promote values to registers directly before they are used
  – Compiler back-end can remove loads and stores

**Compilers for RISC machines usually use a register-to-register model**

• Closely reflects **RISC** instruction sets
• Register use is explicit and well-modelled
The Rest of the Story...

Representing the code is only part of an IR

Other components are necessary

• Symbol table
  – Every name in the program

• Constant table
  – Representation, type
  – Storage class, offset

• Storage map
  – Overall storage layout
  – Overlap information
  – Virtual register assignments

We will return to these tables, their motivation, and their use over the next several lectures.
A parser’s other activities can be termed “semantic elaboration.”

- It may perform many tasks, including
  - Type checking
  - Code generation
  - Storage layout
  - Error checking

- It might create an IR & interpret that IR to produce answers

- It might pretty-print the code back to a file in some modified form.

- Semantic elaboration makes parsers quite versatile tools.
Example of SDT

Consider a simple example: building an AST in a recursive-descent parser

**Simple little grammar**
(similar to a ProductionList)

```
Stmts → Assign StmtList
StmtList → Assign StmtList
           | epsilon
Assign → LHS ← RHS
```

```cpp
def Stmts() -> bool:
    if Assign():
        return StmtList()
    else:
        print("Looking for assignment, found none")
    return False;
```

**Duplicate RHS for Stmts & StmtList ensures that Stmts contains at least one Assign**
Example of SDT

Consider a simple example: building an AST in a recursive-descent parser

```
Stmts  →  Assign  StmtList
StmtList → Assign  StmtList
           |  epsilon
Assign  →  LHS ← RHS
```

Simple little grammar (similar to a ProductionList)

The Plan is easy:
1. Make Stmts() return a (bool, tree) pair (and Assign, and StmtList, ...)
2. Build the appropriate tree structure before the “returns”

bool Stmts( ) {
    if ( Assign() )
        then return StmtList()
    else
        print “Looking for assignment, found none”
        return false;
}

Duplicate RHS for Stmts & StmtList ensures that Stmts contains at least one Assign
Example of SDT

Building the IR in \textit{Stmts()} makes matters somewhat more complex

\begin{verbatim}
pair(bool,tree) Stmts() {
    result1 = Assign()
    if (result1.bool) then {
        result2 = StmtList()
        if (result2.bool) then {
            if (result2.tree == empty) then return result1;
            else return (true, cons(result1,tree,result2.tree))
        }
    }
    else
        print “Looking for assignment, found none”
        return (false, empty)
}
\end{verbatim}

The Plan is easy:
1. Make \textit{Stmts()}, \textit{Assign()}, \textit{StmtList()}, ...
   return a (bool,tree) pair
2. Build the appropriate tree structure before the “returns”

The Implementation is a little messy

Duplicate RHS for \textit{Stmts} & \textit{StmtList} ensures that \textit{Stmts} contains at least one \textit{Assign}
What About AHSDT In an LR(1) Parser?

We specify SDT actions using a simple notation

1. \[ Number \rightarrow DigitList \{ $$ = $1; \} \]
2. \[ DigitList \rightarrow DigitList \text{digit} \{ $$ = $1 * 10 + $2; \} \]
3. \[ | \text{digit} \{ $$ = $2; \} \]

In Bison and Yacc, the compiler writer provides production-specific code snippets that execute when the parser reduces by that production

- Positional notation for the value associated with a symbol
  - $$ is the LHS; $1 is the first symbol in the RHS; $2 the second, ...

- Compiler writer can put arbitrary code in these snippets
  - Solve a travelling salesman problem, compute \( \pi \) to 100 digits, ...
  - More importantly, they can compute on the lexemes of grammar symbols and on information derived and stored earlier in translation
Fitting AHSDT into the \textbf{LR(1)} Skeleton Parser

\begin{verbatim}
stack.push(INVALID);
stack.push(s_0); // initial state
word = scanner.next_word();
loop forever {
  s = stack.top();
  if ( ACTION[s,word] == \textit{reduce A\rightarrow\beta} ) then {
    stack.popnum(2*|\beta|); // pop 2*|\beta| symbols
    s = stack.top();
    stack.push(A); // push LHS, A
    stack.push(GOTO[s,A]); // push next state
  }
  else if ( ACTION[s,word] == \textit{shift} s_i \textit{) then {}
    stack.push(word); stack.push(s_i);
    word ← scanner.next_word();
  }
  else if ( ACTION[s,word] == \textit{accept}
            & word == EOF )
    then break;
  else throw a syntax error;
}
report success;
\end{verbatim}

\textbf{Actions are taken on reductions}

- Insert a call to \textit{Work}() before the call to stack.popnum()
- \textit{Work}() contains a case statement that switches on the production number
- Code in \textit{Work}() can read items from the stack
  \rightarrow That is why it calls \textit{Work}() before stack.popnum()
Fitting AHSDT into the LR(1) Skeleton Parser

```java
stack.push(INVALID);
stack.push(s0); // initial state
word = scanner.next_word();
loop forever {
    s = stack.top();
    if ( ACTION[s,word] == "reduce A→β" ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A); // push LHS, A
        stack.push(GOTO[s,A]); // push next state
    }
    else if ( ACTION[s,word] == "shift s_i" ) then {
        stack.push(word); stack.push(s_i);
        word ← scanner.next_word();
    }
    else if ( ACTION[s,word] == "accept"
             & word == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

**Passing values between actions**

- Tie values to instances of grammar symbols
  - Equivalent to parse tree nodes
- We can pass values on the stack
  - Push / pop 3 rather than 2
  - Work() takes the stack as input (conceptually) and returns the value for the reduction it processes
- Shift creates initial values
Fitting AHSDT into the LR(1) Skeleton Parser

- Modifications are minor
  - Insert call to Work()
  - Change the push() & pop() behavior
- Same asymptotic behavior as the original algorithm.
  - 50% more stack space
- Last obstacle is making it easy to write the code for Work()

Note that, in C, the stack has some odd union type.
Translating Code Snippets Into \textit{Work()}

For each production, the compiler writer can provide a code snippet

\begin{verbatim}
{ value = value * 10 + digit; }
\end{verbatim}

We need a scheme to name stack locations. Yacc introduced a simple one that has been widely adopted.

- \$$\$$ refers to the result, which will be pushed on the stack
- \$1\ is the first item on the productions right hand side
- \$2\ is the second item
- \$3\ is the third item, and so on ...

The digits example above becomes

\begin{verbatim}
{ $$ = $1 * 10 + $2; }
\end{verbatim}
Translating Code Snippets Into `Work()`

How do we implement `Work()`?

- `Work()` takes 2 arguments: the stack and a production number
- `Work()` contains a case statement that switches on production number
  - Each case contains the code snippet for a reduction by that production
  - The $1$, $2$, $3$ ... macros translate into references into the stack
  - The $$ macro translates into the return value

```java
...
if ( ACTION[s,word] == "reduce A→β" ) then {
  r = Work(stack, "A→β")
  stack.popnum(3*|β|); // pop 3*|β| symbols
  s = stack.top(); // save exposed state
  stack.push(A); // push A
  stack.push ( r); // push result of WORK()
  stack.push(GOTO[s,A]); // push next state
}
...
```

$$ translates to r

$i$ translates to the stack location $3 \times (|β| - i + 1)$ units down from stacktop

Note that $β$, $i$, $3$, and $1$ are all constants so $i$ can be evaluated to a compile-time constant