Computer Languages

Code generation: instruction Selection

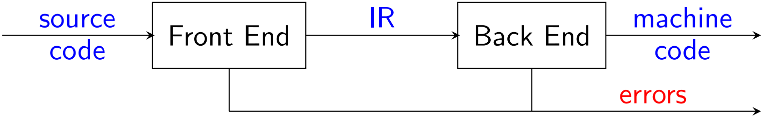
Verónica Gaspes

School of Information Science, Computer and Electrical Engineering

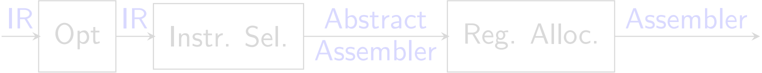
March 4

www2.hh.se/staff/vero/languages

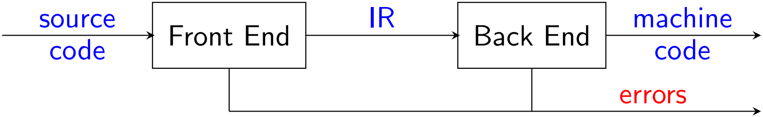
The back-end



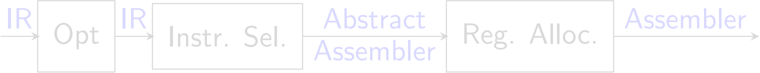
The back-end is also structured in phases!



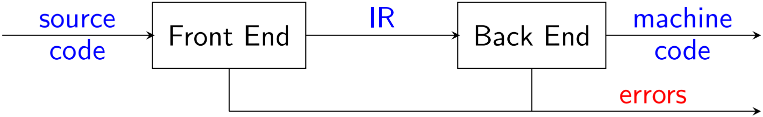
The back-end



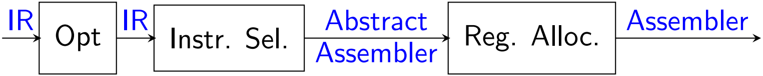
The back-end is also structured in phases!



The back-end



The back-end is also structured in phases!



Observation

For the back-end we no longer look at *minijava*! We compile IR-trees to assembler.

A small minijava program

```
class A{
   public static void main(String[] a){
      System.out.println(new B().f(3).f());
   }
}
class B{
   int x;
   int y;
   public C f(int z){
      if(x<y) x=z+1; else x=z+x;
      return new C();
   }
}
class C{
   public int f(){return 3;}
}
```

The result of translating it

```
PROCEDURE :main
EXPS(
 CALL(
  NAME _printint,
  CONST 0,
  CALL(
   NAME C_f,
   CALL(
    NAME B_f,
    CALL(
     NAME _malloc,
     CONST 0,
     CONST 8),
    CONST 3))))
```

```
PROCEDURE :B_f
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
PROCEDURE :C_f
MOVE(
 TEMP t32,
 CONST 3)
```

The result of translating it

```
PROCEDURE :main
EXPS(
 CALL(
  NAME _printint,
  CONST 0,
  CALL(
   NAME C_f,
   CALL(
    NAME B_f,
    CALL(
     NAME _malloc,
     CONST 0,
     CONST 8),
    CONST 3))))
```

```
PROCEDURE :B_f
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
PROCEDURE :C_f
MOVE(
 TEMP t32,
 CONST 3)
```

The result of translating it

```
PROCEDURE :main
EXPS(
 CALL(
  NAME _printint,
  CONST 0,
  CALL(
   NAME C_f,
   CALL(
    NAME B_f,
    CALL(
     NAME _malloc,
     CONST 0,
     CONST 8),
    CONST 3))))
```

```
PROCEDURE :B_f
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
PROCEDURE :C_f
MOVE(
 TEMP t32,
 CONST 3)
```

(almost)All of B_f

```
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
SEQ(
 SEQ(
  SEQ(
   LABEL L0,
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)),
    BINOP(PLUS,
     TEMP t65,
     CONST 1))),
  JUMP(
   NAME L2)),
```

```
SEQ(
 SEQ(
  LABEL L1,
  MOVE(
   MEM(
    BINOP(PLUS,
     TEMP t64,
     CONST 0)),
   BINOP(PLUS,
    TEMP t65,
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)))))),
 JUMP(
  NAME L2)))),
```

(almost)All of B_f

```
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
SEQ(
 SEQ(
  SEQ(
   LABEL L0,
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)),
    BINOP(PLUS,
     TEMP t65,
     CONST 1))),
  JUMP(
   NAME L2)),
```

```
SEQ(
 SEQ(
  LABEL L1,
  MOVE(
   MEM(
    BINOP(PLUS,
     TEMP t64,
     CONST 0)),
   BINOP(PLUS,
    TEMP t65,
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)))))),
 JUMP(
  NAME L2))),
```

(almost)All of B_f

```
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
SEQ(
 SEQ(
  SEQ(
   LABEL L0,
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)),
    BINOP(PLUS,
     TEMP t65,
     CONST 1))),
  JUMP(
   NAME L2)),
```

```
SEQ(
 SEQ(
  LABEL L1,
  MOVE(
   MEM(
    BINOP(PLUS,
     TEMP t64,
     CONST 0)),
   BINOP(PLUS,
    TEMP t65,
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0))))),
 JUMP(
  NAME L2)))),
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

Instruction Selection

- The assembler instructions of the target machine can be studied from the documentation following with the SPIM simulator (linked from the course's homepage).
- To develop the program assigning assembler instructions to IR statements, we follow the algorithm *suitable for RISC architectures* presented in the book:

Maximal Munch

- Identify patterns of trees that are implemented by assembler instructions
- Tile the IR tree using these patterns so that all nodes in the tree are *covered*
- Output the sequence of instructions corresponding to the tiling

Instruction Selection

- The assembler instructions of the target machine can be studied from the documentation following with the SPIM simulator (linked from the course's homepage).
- To develop the program assigning assembler instructions to IR statements, we follow the algorithm *suitable for RISC architectures* presented in the book:

Maximal Munch

- Identify patterns of trees that are implemented by assembler instructions
- Tile the IR tree using these patterns so that all nodes in the tree are covered
- Output the sequence of instructions corresponding to the tiling

Instruction Selection

- The assembler instructions of the target machine can be studied from the documentation following with the SPIM simulator (linked from the course's homepage).
- To develop the program assigning assembler instructions to IR statements, we follow the algorithm *suitable for RISC architectures* presented in the book:

Maximal Munch

- Identify patterns of trees that are implemented by assembler instructions
- Tile the IR tree using these patterns so that all nodes in the tree are covered
- Output the sequence of instructions corresponding to the tiling

Instruction Selection

- The assembler instructions of the target machine can be studied from the documentation following with the SPIM simulator (linked from the course's homepage).
- To develop the program assigning assembler instructions to IR statements, we follow the algorithm *suitable for RISC architectures* presented in the book:

Maximal Munch

- Identify patterns of trees that are implemented by assembler instructions
- Tile the IR tree using these patterns so that all nodes in the tree are covered
- Output the sequence of instructions corresponding to the tiling

Instruction Selection

- The assembler instructions of the target machine can be studied from the documentation following with the SPIM simulator (linked from the course's homepage).
- To develop the program assigning assembler instructions to IR statements, we follow the algorithm *suitable for RISC architectures* presented in the book:

Maximal Munch

- Identify patterns of trees that are implemented by assembler instructions
- Tile the IR tree using these patterns so that all nodes in the tree are covered
- Output the sequence of instructions corresponding to the tiling

Instruction Selection

- The assembler instructions of the target machine can be studied from the documentation following with the SPIM simulator (linked from the course's homepage).
- To develop the program assigning assembler instructions to IR statements, we follow the algorithm *suitable for RISC architectures* presented in the book:

Maximal Munch

- Identify patterns of trees that are implemented by assembler instructions
- Tile the IR tree using these patterns so that all nodes in the tree are covered
- Output the sequence of instructions corresponding to the tiling

A simple instruction set

ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j + c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

There are only arithmetic and memory instructions (no jumps!).

r_0 is allways 0.

A simple instruction set

ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j + c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

There are only arithmetic and memory instructions (no jumps!).

r_0 is allways 0.

A simple instruction set

ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j + c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

There are only arithmetic and memory instructions (no jumps!).

r_0 is allways 0.

A simple instruction set

ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j + c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

There are only arithmetic and memory instructions (no jumps!).

r_0 is allways 0.

A simple instruction set

ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j + c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

There are only arithmetic and memory instructions (no jumps!).

r_0 is allways 0.

Identifying patterns

Instruction	Patterns
r_i	TEMP()
$r_i \leftarrow r_j + r_k$	BINOP(+,_,_)
$r_i \leftarrow r_j + c$	BINOP(+,_,CONST()) BINOP(+,CONST(),_) CONST()
$r_i \leftarrow M[r_j + c]$	MEM(BINOP(+,_,CONST())) MEM(CONST())
$r_i \leftarrow M[r_j + c]$	MOVE(MEM(BINOP(+,_,CONST())),_) MOVE(_,MEM(BINOP(+,_,CONST())))
$M[r_j] \leftarrow M[r_i]$	MOVE(MEM(_),MEM(_))
...	...

Important

All IR nodes should be covered by some pattern!

Identifying patterns

Instruction	Patterns
r_i	TEMP()
$r_i \leftarrow r_j + r_k$	BINOP(+,_,_)
$r_i \leftarrow r_j + c$	BINOP(+,_,CONST()) BINOP(+,CONST(),_) CONST()
$r_i \leftarrow M[r_j + c]$	MEM(BINOP(+,_,CONST())) MEM(CONST())
$r_i \leftarrow M[r_j + c]$	MOVE(MEM(BINOP(+,_,CONST())),_) MOVE(_,MEM(BINOP(+,_,CONST()))))
$M[r_j] \leftarrow M[r_i]$	MOVE(MEM(_),MEM(_))
...	...

Important

All IR nodes should be covered by some pattern!

Identifying patterns

Instruction	Patterns
r_i	TEMP()
$r_i \leftarrow r_j + r_k$	BINOP(+,_,_)
$r_i \leftarrow r_j + c$	BINOP(+,_,CONST()) BINOP(+,CONST(),_) CONST()
$r_i \leftarrow M[r_j + c]$	MEM(BINOP(+,_,CONST())) MEM(CONST())
$r_i \leftarrow M[r_j + c]$	MOVE(MEM(BINOP(+,_,CONST())),_) MOVE(_,MEM(BINOP(+,_,CONST())))
$M[r_j] \leftarrow M[r_i]$	MOVE(MEM(_),MEM(_))
...	...

Important

All IR nodes should be covered by some pattern!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
  - Starting with the root of the IR tree find the largest tile that fits,
  - cover the root and subnodes matching the tile,
  - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
    - Starting with the root of the IR tree find the largest tile that fits,
    - cover the root and subnodes matching the tile,
    - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
  - Starting with the root of the IR tree find the largest tile that fits,
  - cover the root and subnodes matching the tile,
  - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
  - Starting with the root of the IR tree find the largest tile that fits,
  - cover the root and subnodes matching the tile,
  - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
  - Starting with the root of the IR tree find the largest tile that fits,
  - cover the root and subnodes matching the tile,
  - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
    - Starting with the root of the IR tree find the largest tile that fits,
    - cover the root and subnodes matching the tile,
    - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Tiling

- View the tree patterns as tiles and cover the IR-tree that is being translated with nonoverlapping tiles.
- Greedy algorithm:
    - Starting with the root of the IR tree find the largest tile that fits,
    - cover the root and subnodes matching the tile,
    - recursively apply the algorithm on all non-covered subtrees.
- Examples on the whiteboard!

Representing instructions

- The result of instruction selection is a list of instructions.
- An instruction can
  - be formatted: made into a string to contribute one line to the assembler file
  - return the list of registers it uses (sources) and the list of registers it modifies (destinations). These 2 lists are later used by the register allocator.
  - return the list of labels it jumps to.

Representing instructions

- The result of instruction selection is a list of instructions.
- An instruction can
  - be formatted: made into a string to contribute one line to the assembler file
  - return the list of registers it uses (sources) and the list of registers it modifies (destinations). These 2 lists are later used by the register allocator.
  - return the list of labels it jumps to.

Representing instructions

- The result of instruction selection is a list of instructions.
- An instruction can
  - be formatted: made into a string to contribute one line to the assembler file
  - return the list of registers it uses (sources) and the list of registers it modifies (destinations). These 2 lists are later used by the register allocator.
  - return the list of labels it jumps to.

Representing instructions

- The result of instruction selection is a list of instructions.
- An instruction can
  - be formatted: made into a string to contribute one line to the assembler file
  - return the list of registers it uses (sources) and the list of registers it modifies (destinations). These 2 lists are later used by the register allocator.
  - return the list of labels it jumps to.

Representing instructions

- The result of instruction selection is a list of instructions.
- An instruction can
  - be formatted: made into a string to contribute one line to the assembler file
  - return the list of registers it uses (sources) and the list of registers it modifies (destinations). These 2 lists are later used by the register allocator.
  - return the list of labels it jumps to.

Representing instructions

- The result of instruction selection is a list of instructions.
- An instruction can
  - be formatted: made into a string to contribute one line to the assembler file
  - return the list of registers it uses (sources) and the list of registers it modifies (destinations). These 2 lists are later used by the register allocator.
  - return the list of labels it jumps to.

Representing instructions

```
package assem;
public abstract class Instr {
   public String assem;
   public Temp[] use;
   public Temp[] def;
   public String format(TempMap m) {...}
   }
```

- The method for formatting takes as argument a table saying how registers should be represented as a string and produces a string for the instruction.
- In the string containing the assembler instruction registers are named as strings but they have to be related to the registers in the lists *use* and *def*

Representing instructions

```
package assem;
public abstract class Instr {
   public String assem;
   public Temp[] use;
   public Temp[] def;
   public String format(TempMap m) {...}
   }
```

- The method for formatting takes as argument a table saying how registers should be represented as a string and produces a string for the instruction.
- In the string containing the assembler instruction registers are named as strings but they have to be related to the registers in the lists *use* and *def*

Representing instructions

```
package assem;
public abstract class Instr {
   public String assem;
   public Temp[] use;
   public Temp[] def;
   public String format(TempMap m) {...}
   }
```

- The method for formatting takes as argument a table saying how registers should be represented as a string and produces a string for the instruction.
- In the string containing the assembler instruction registers are named as strings but they have to be related to the registers in the lists *use* and *def*

Representing instructions

```
package assem;
public abstract class Instr {
  public String assem;
  public Temp[] use;
  public Temp[] def;
  public String format(TempMap m) {...}
  }
```

- The method for formatting takes as argument a table saying how registers should be represented as a string and produces a string for the instruction.
- In the string containing the assembler instruction registers are named as strings but they have to be related to the registers in the lists `use` and `def`

Representing instructions

- One way of creating instructions is by using

```
public class OPER extends Instr {
   public OPER(String a, Temp[] d, Temp[] s,
           List<Label> j) {
...}
}
```

- By providing
  - a string with an assembler instruction with registers using codes, e.g. "add ’d0 ’s0 ’s1"
  - The lists of temporaries with sources and destinations to be matched with the codes. Some of the temporaries will be the result of the translation of IR-trees of subexpressions.

Representing instructions

- One way of creating instructions is by using

```
public class OPER extends Instr {
   public OPER(String a, Temp[] d, Temp[] s,
          List<Label> j) {
...}
}
```

- By providing
    - a string with an assembler instruction with registers using *codes*, e.g. `"add 'd0 's0 's1"`
    - The lists of temporaries with sources and destinations to be matched with the codes. Some of the temporaries will be the result of the translation of IR-trees of subexpressions.

Representing instructions

- One way of creating instructions is by using

```
public class OPER extends Instr {
   public OPER(String a, Temp[] d, Temp[] s,
          List<Label> j) {
...}
}
```

- By providing
  - a string with an assembler instruction with registers using *codes*, e.g. `"add 'd0 's0 's1"`
  - The lists of temporaries with sources and destinations to be matched with the codes. Some of the temporaries will be the result of the translation of IR-trees of subexpressions.

Representing instructions

- One way of creating instructions is by using

  ```
  public class OPER extends Instr {
     public OPER(String a, Temp[] d, Temp[] s,
            List<Label> j) {
  ...}
  }
  ```

- By providing
  - a string with an assembler instruction with registers using *codes*, e.g. `"add ’d0 ’s0 ’s1"`
  - The lists of temporaries with sources and destinations to be matched with the codes. Some of the temporaries will be the result of the translation of IR-trees of subexpressions.

Representing instructions

- One way of creating instructions is by using

```
public class OPER extends Instr {
   public OPER(String a, Temp[] d, Temp[] s,
          List<Label> j) {
...}
}
```

- By providing
  - a string with an assembler instruction with registers using *codes*, e.g. `"add 'd0 's0 's1"`
  - The lists of temporaries with sources and destinations to be matched with the codes. Some of the temporaries will be the result of the translation of IR-trees of subexpressions.

Representing instructions

- One way of creating instructions is by using

```
public class OPER extends Instr {
   public OPER(String a, Temp[] d, Temp[] s,
          List<Label> j) {
...}
}
```

- By providing
  - a string with an assembler instruction with registers using *codes*, e.g. `"add 'd0 's0 's1"`
  - The lists of temporaries with sources and destinations to be matched with the codes. Some of the temporaries will be the result of the translation of IR-trees of subexpressions.

Producing assembler instructions

- To recursively traverse an IR-tree we will use a *visitor*

```
package tree;
public interface CodeVisitor {
   public void visit(JUMP n);
   public void visit(CJUMP n);
   public void visit(MOVE n);
   ...
   public Temp visit(BINOP n);
   public Temp visit(MEM n);
   public Temp visit(TEMP n);

   ...
```

- Notice the return type for visiting `tree.Exp`s.
- When we recursively call the instruction generator (a code visitor) on a `tree.Exp` we get the register where it leaves the result!

Producing assembler instructions

- To recursively traverse an IR-tree we will use a *visitor*

```
package tree;
public interface CodeVisitor {
   public void visit(JUMP n);
   public void visit(CJUMP n);
   public void visit(MOVE n);
   ...
   public Temp visit(BINOP n);
   public Temp visit(MEM n);
   public Temp visit(TEMP n);

   ...
```

- Notice the return type for visiting `tree.Exp`s.
- When we recursively call the instruction generator (a code visitor) on a `tree.Exp` we get the register where it leaves the result!

Producing assembler instructions

- To recursively traverse an IR-tree we will use a *visitor*

```
package tree;
public interface CodeVisitor {
   public void visit(JUMP n);
   public void visit(CJUMP n);
   public void visit(MOVE n);
   ...
   public Temp visit(BINOP n);
   public Temp visit(MEM n);
   public Temp visit(TEMP n);

   ...
```

- Notice the return type for visiting `tree.Exp`s.
- When we recursively call the instruction generator (a code visitor) on a `tree.Exp` we get the register where it leaves the result!

Producing assembler instructions

- To recursively traverse an IR-tree we will use a *visitor*

```
package tree;
public interface CodeVisitor {
   public void visit(JUMP n);
   public void visit(CJUMP n);
   public void visit(MOVE n);
   ...
   public Temp visit(BINOP n);
   public Temp visit(MEM n);
   public Temp visit(TEMP n);

   ...
```

- Notice the return type for visiting `tree.Exp`s.
- When we recursively call the instruction generator (a code visitor) on a `tree.Exp` we get the register where it leaves the result!

Producing assembler instructions

- To recursively traverse an IR-tree we will use a *visitor*

```
package tree;
public interface CodeVisitor {
   public void visit(JUMP n);
   public void visit(CJUMP n);
   public void visit(MOVE n);
   ...
   public Temp visit(BINOP n);
   public Temp visit(MEM n);
   public Temp visit(TEMP n);

   ...
```

- Notice the return type for visiting `tree.Exp`s.
- When we recursively call the instruction generator (a code visitor) on a `tree.Exp` we get the register where it leaves the result!

Producing assembler instructions

- In the class generating instructions, say *`class Codegen implements tree.CodeVisitor`*
- the patterns must be *programmed*
- In Java we do this by *restricting* the argument to the visit method (longest patterns first!):

```
public void visit(Tree.MOVE s) {
   // MOVE(MEM, Exp)
   if (s.dst instanceof Tree.MEM) {
      Tree.MEM mem = (Tree.MEM)s.dst;
      // MOVE(MEM(+ Exp CONST), Exp)
      if (mem.exp instanceof Tree.BINOP) {
         Tree.BINOP b = (Tree.BINOP)mem.exp;
         if (b.binop == Tree.BINOP.PLUS && immediate(b)) {
         ...
               emit(OPER("sw `s0 " + off + "(`s1)", null,

                  new Temp[]{s.src.accept(this), left}));
```

Producing assembler instructions

- In the class generating instructions, say

  class Codegen implements tree.CodeVisitor

- the patterns must be *programmed*

- In Java we do this by *restricting* the argument to the visit method (longest patterns first!):

```
public void visit(Tree.MOVE s) {
   // MOVE(MEM, Exp)
   if (s.dst instanceof Tree.MEM) {
      Tree.MEM mem = (Tree.MEM)s.dst;
      // MOVE(MEM(+ Exp CONST), Exp)
      if (mem.exp instanceof Tree.BINOP) {
         Tree.BINOP b = (Tree.BINOP)mem.exp;
         if (b.binop == Tree.BINOP.PLUS && immediate(b)) {
         ...
               emit(OPER("sw `s0 " + off + "(`s1)", null,

                  new Temp[]{s.src.accept(this), left}));
```

Producing assembler instructions

- In the class generating instructions, say
  `class Codegen implements tree.CodeVisitor`
- the patterns must be *programmed*
- In Java we do this by *restricting* the argument to the visit method (longest patterns first!):

```
public void visit(Tree.MOVE s) {
   // MOVE(MEM, Exp)
   if (s.dst instanceof Tree.MEM) {
      Tree.MEM mem = (Tree.MEM)s.dst;
      // MOVE(MEM(+ Exp CONST), Exp)
      if (mem.exp instanceof Tree.BINOP) {
         Tree.BINOP b = (Tree.BINOP)mem.exp;
         if (b.binop == Tree.BINOP.PLUS && immediate(b)) {
         ...
               emit(OPER("sw `s0 " + off + "(`s1)", null,

                  new Temp[]{s.src.accept(this), left}));
```

Producing assembler instructions

- In the class generating instructions, say
  `class Codegen implements tree.CodeVisitor`
- the patterns must be *programmed*
- In Java we do this by *restricting* the argument to the visit method (longest patterns first!):

```
public void visit(Tree.MOVE s) {
   // MOVE(MEM, Exp)
   if (s.dst instanceof Tree.MEM) {
      Tree.MEM mem = (Tree.MEM)s.dst;
      // MOVE(MEM(+ Exp CONST), Exp)
      if (mem.exp instanceof Tree.BINOP) {
         Tree.BINOP b = (Tree.BINOP)mem.exp;
         if (b.binop == Tree.BINOP.PLUS && immediate(b)) {
         ...
               emit(OPER("sw `s0 " + off + "(`s1)", null,

                  new Temp[]{s.src.accept(this), left}));
```

Producing assembler instructions

- In the class generating instructions, say
  `class Codegen implements tree.CodeVisitor`
- the patterns must be *programmed*
- In Java we do this by *restricting* the argument to the visit method (longest patterns first!):

```
public void visit(Tree.MOVE s) {
   // MOVE(MEM, Exp)
   if (s.dst instanceof Tree.MEM) {
      Tree.MEM mem = (Tree.MEM)s.dst;
      // MOVE(MEM(+ Exp CONST), Exp)
      if (mem.exp instanceof Tree.BINOP) {
         Tree.BINOP b = (Tree.BINOP)mem.exp;
         if (b.binop == Tree.BINOP.PLUS && immediate(b)) {
         ...
               emit(OPER("sw ‘s0 " + off + "(‘s1)", null,

                  new Temp[]{s.src.accept(this), left}));
```

Laboration 5

- You will have to program *`class Codegen implements tree.CodeVisitor`*
- You get all classes in the package *`assem`*
- You get a *`Main`* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)

Laboration 5

- You will have to program *class Codegen implements tree.CodeVisitor*
- You get all classes in the package *assem*
- You get a *Main* that calls on all the phases:
  - parsing
  - typechecking
  - intermediate code generation
  - transformations
  - instruction selection
- You have to read the description of SPIM assembler (starting on page 54 of the manual).
- You have to read the calling convention to know what registers to use (SPIM has dedicated registers!!)