Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• LB

• Scope

• Control structures
A compiler

High level programming language

Front-end

- Name binding
- Control structure translation
- Values encoding
- Checking memory accesses
- Basic block generation

IR

Middle-end

IR

Back-end

- Explicit control flow and data types
- Instruction selection
- Register allocation
- Peephole optimizations
- Instruction scheduling

Machine code
int64 myF (int64 p1) {
    int64 v1, v2
    v1 = -1
    v2 = -2
    print(v2)
    print(v1)
    int64 c
    c = v1 = p1
    if (v1 = p1) :true :false
        :true
        return 1
        :false
        return 2
    }
}
p ::= f^+
f ::= T name ( (type name)* ) { i^* }
i ::= type name | name <- s | name <- t op t |
     label | br label | br t label label | return | return t
     name <- name([t])^+ | name([t])^+ <- s |
     name <- length name t |
     name( args? ) | name <- name( args? ) | print(t) |
     name <- new Array(args) | name <- new Tuple(t)
T ::= type | void
type ::= int64([])* | tuple | code
vars ::= var | var (, var)*
args ::= t | t (, t)*
s ::= t | label
t ::= name | N
op ::= + | - | * | & | << | >> | < | <= | = | >= | >
name ::= [a-zA-Z_][a-zA-Z_0-9]*
label ::= :name
\[
p := f^+
\]

\[
f := T \text{name} \ ((\text{type name})^* ) \ \text{scope}
\]

\[
\text{scope} := \{i^+\}
\]

\[
i := \text{type names} \mid \text{name} \leftarrow s \mid \text{name} \leftarrow \text{cond} \mid
\]

\[
\left. \begin{array}{l}
\text{label} \mid \text{if} \ (\text{cond}) \ \text{label} \ \text{label} \mid \text{br} \ \text{label} \mid \text{return} \ (t) ? \mid
\end{array} \right\} \\
\left. \begin{array}{l}
\text{while} \ (\text{cond}) \ \text{label} \ \text{label} \mid \text{continue} \mid \text{break} \mid
\end{array} \right\}
\]

\[
\text{name} \leftarrow \text{name}([t])^+ \mid \text{name}([t])^+ \leftarrow s \mid \text{name} \leftarrow \text{length} \ \text{name} \ t \mid
\]

\[
\text{name} (\text{args} ?) \mid \text{name} \leftarrow \text{name} (\text{args}?) \mid \text{print} (t) \mid
\]

\[
\text{name} \leftarrow \text{new\ Array} (\text{args}) \mid \text{name} \leftarrow \text{new\ Tuple}(t) \mid \text{scope}
\]

\[
T := \text{int64}([[]])^* \mid \text{tuple} \mid \text{code} \mid \text{void}
\]

\[
\text{cond} := t \ \text{op} \ t
\]

\[
\text{names} := \text{name} \mid \text{name} (, \ \text{name})^*
\]

\[
\text{args} := t \mid t (, t)^*
\]

\[
\text{s} := t \mid \text{label}
\]

\[
\text{t} := \text{name} \mid N
\]

\[
\text{op} := + \mid - \mid * \mid \& \mid << \mid >> \mid < \mid <= \mid = \mid >= \mid >
\]

\[
\text{name} := [a-zA-Z_][a-zA-Z_0-9]^*
\]

\[
\text{label} := :\text{name}
\]
void main ( ){
    int64 v1, v2, v3
    v1 <- 1
    v2 <- 2
    v3 <- v1 + v2
    if (v3 > 1) :true :false
    :true
      print(v3)
    :false
      return
}
LB example: if statement

```c
void myF (int64 v3){
    if (v3 > 1) :true :false
      :true
        print(v3)
        br :exit
    :false
        print(1)
        :exit
        return
}
```

Output if v3 is 2:

```
2
```
void myF (int64 v3){
    if (v3 > 1) : true : false

    : true
    print(v3)

    : false
    print(1)
    : exit
    return
}

Output if v3 is 2:
2
1
LB example: while statement with continue

```c
void myF (int64 v) {
  int64 c
  c <- 0
  while (c < v) :body :exit
    :body
    print(c)
    c <- c + 1
    continue
  :exit
  return
}
```
LB example: while statement w/o continue

```c
void myF (int64 v) {
    int64 c
    c <- 0
    while (c < v) :body :exit
        :body
        print(c)
        c <- c + 1
    :exit
    return
}
```
void myF (int64 v) {
    int64 c
    c <- 0
    while (c < v) :body :exit
    {
        :body
        print(c)
        c <- c + 1
        continue
    }
    :exit
    return
}
LB example 3: while statement

void myF (int64 v) {
    int64 c
    c <- 0
    {
        :body
        print(c)
        c <- c + 1
    } while (c < v)
    :exit
    :exit
    return
}
LB example 4: while statement

```c
void myF (int64[] v) {
    int64 l, index
    l <- length v 0
    index <- 0
    while (index < l) :body :exit
        :body
        print(index)
        index <- index + 1
        continue
    :exit
    return
}
```
LB example 5: while statement with break

```c
void myF (int64 v) {
    int64 c
    c <- 0
    {
        :body
        print(c)
        if (c = 42) :WOW : WOW2 :WOW :WOW
        break
        :WOW2
        c <- c + 1
    }  while (c < v) :body :exit :exit
    return
}
```
Now that you know LB

Write an LB program
To translate an LB function to LA

- Translate multiple variable declarations
- Flat the nested scopes
- Translate if and while statements
Translate multiple declarations

... int64 v1, v2 ...

Your work

... int64 v1
... int64 v2 ...

...
Outline

• LB

• Scope

• Control structures
Name binding

• Association of entities (e.g., variables) with identifiers
• For example:

```c
int64 myF (int64 p1){
    int64 v1
    v1 <- 1
    {
        int64 v1
        v1 <- 2
        ...
    }
}
```
Scope

- Determines which names bind to which entities (e.g., variables)

```cpp
int64 myF (int64 p1){
    int64 v1
    v1 <- 1
    {
        int64 v1
        v1 <- 2
        print(v1)
    }
    print(v1)
}
```
Binding time

- Static (LB variables, C variables, C++ variables, Java variables)
- Dynamic (C++ virtual methods, Java object methods)

class A{
  virtual void myF();
}
class B : A{
  void myF() override;
}
void anotherF(A *obj) {
  obj->myF();
}
Translating an LB function

1. For each variable \( v \) defined by instruction \( i \) in a nested scope \( s \)
   • Rename \( v \) in \( i \) to a new and unique name \( z \)
   • Remember the mapping \( v \rightarrow z \): \( s.map[v] = z \)

2. For each instruction \( i \) of a function
   • For each variable \( v \) either used or defined by \( i \)
     • Bind \( v \) to the variable \( q \) that has been defined in the original code to the most nested scope \( s \)
     • If \( s \) exists, then change the reference \( v \) of \( i \) from \( v \) to \( s.map[q] \)

3. Remove all nested scopes
Example

```c
int64 myF (int64 p1){
    int64 v1
    v1 <- 1
    {
        int64 v1
        v1 <- 2
        ...
    }
}
```

```c
int64 myF (int64 p1){
    int64 v1
    v1 <- 1
    {
        int64 v1_1
        v1_1 <- 2
        ...
    }
}
```

```c
int64 myF (int64 p1){
    int64 v1
    v1 <- 1
    {
        int64 v1_1
        v1_1 <- 2
        ...
    }
}
```
Outline

• LB

• Scope

• Control structures
Translating if structures

\[
\text{... if (v1 = p1) :true :false ...}
\]

Your work

\[
\text{... int64 newV}
\text{newV <- v1 = p1}
\text{br newV :true :false ...}
\]
Translating while structures

1. Identify entry and exit point of each while instruction \( w \)
   \[ \text{while} \ (\%v1 = 3) \ : body_w \ : after_w \]
   
   - beginWhile[\( w \)] = :body_w
   - endWhile[\( w \)] = :after_w

2. Add a new label \( l \) just before a while instruction \( w \)
   
   - condLabels[\( w \)] = \( l \)

3. Map instructions to their more nested loop \( w \)

4. Translate while instructions

5. Translate continue and break instructions
Mapping instructions to their loops

\[ i = F.\text{firstInstruction}(); \text{whileSeen} = \{}; \text{loopStack} = \text{Stack}(); \]

while (i){
    if (we are within the body of a loop w) \( \text{loop}[i] = w; \)
    if (i is a while instruction not included in whileSeen) {
        loopStack.push(i); whileSeen.add(i); continue;
    }
    if (i is a label and it’s the beginning of a while loop w such that \( w \notin \text{whileSeen} \)){
        loopStack.push(w); continue;
    }
    if (i is a label and it’s the end of a loop w) loopStack.pop();
    i = next(i);
}
Translating continue

• Let $i$ be a continue instruction

• Fetch the innermost loop $w$ that $i$ belongs to:
  $w = \text{Loop}[i]$

• Fetch the label placed just before the condition of $w$:
  $l_{\text{cond}} = \text{condLabels}[w]$

• Generate a jump to the condition code of $w$
  $br l_{\text{cond}}$
Translating break

• Let $i$ be a break instruction

• Fetch the innermost loop $w$ that $i$ belongs to:
  $w = Loop[i]$

• Fetch the exit label of $w$:
  $l_{\text{exit}} = endWhile[w]$

• Generate a jump to leave the loop $w$
  $br l_{\text{exit}}$
The LB compiler (LBc)

- LB program
- Your work
- prog.a
- LAc
- a.out

- Competition: During our last class
- Winner of the competition:
  - Get an A
  - His/her/their name(s) go to the Hall of Fame of the class
Homework #7 (and last one)

A. Write a compiler that translates an LB program (.b) to an LA one
   • You need to generate prog.a
   • You need to pass all tests in the framework

B. You can compete only if
   • You pass all L1, L2, L3, IR, LA, LB tests
   • You submit your LBc and LB program before the deadline (hard deadline)

Good luck!
What todo with your LBc

• Testing:
  • Compile all compilers:
    run make from the parent directory of the framework
  • Test LBC: cd LB ; make test

• Submitting
  • Please use “make homework”
  • Submit the archive generated by make homework

• Competing
  • Login in hanlon
  • Compiler your compilers: make from the parent directory
  • Run your compiler: cd LB; make performance
Competition

• It will be live

• All correct LB compilers will participate together with gcc, clang, and our compilers
Workload for the class competition

- LB program: LB/tests/competition2019.b

- After compiling this program, take a look at L1/prog.S 😊
  - Could you have implemented the same workload writing it directly in assembly?

- The official competition will use only the latest LBc compiler that you have submitted before the deadline