Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• LC

• Parsing

• Translating high level control structures
p ::= f*
f ::= T name ( (type name)* ) scope
scope ::= { i* }
i ::= type names | name <- s | name <- cond |
       label | if (cond) label label | br label | return (t)? |
       while (cond) label label | continue | break |
       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) | scope
T ::= type | void
type ::= int64([])* | tuple | code
args ::= t | t (, t)*
s ::= t | label
t ::= name | N
N ::= (+|-)? [1-9][0-9]*
op ::= + | - | * | & | << | >> | < | <= | = | >= | >
name ::= [a-zA-Z_][a-zA-Z_0-9]*
label ::= :name
cond ::= t op t
names ::= name | name (, name)*
p ::= f^*
f ::= T name ( (type name)* ) scope
scope ::= { i^* }
i ::= i1 | i2 | scope
i1 ::= name <- s | name <- cond | 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)
i2 ::= type names | if (cond) scope else scope | return (t)? |
       while (cond) scope | do scope while (cond) | for (i1?; cond? ; i1?) scope | continue | break
T ::= type | void
type ::= int([])* | tuple | code
args ::= t | t (, t)*
s ::= t | t
name
name
N ::= (+|-)? [1-9][0-9]*
op ::= + | - | * | & | << | >> | < | <= | = | >= | >
name ::= [a-zA-Z_][a-zA-Z_0-9]*
cond ::= t op t
names ::= name | name (, name)*

- No labels
- High level loops
LC example 0: if then else

```c
void main (){
    int index
    index <- 0
    if (index < 10) {
        index <- index + 1
    } else {
        index <- index – 1
    }
    return
}
```
p ::= f*
f ::= T name ( (type name)* ) scope
scope ::= { i* }
i ::= i1 | i2 | scope
i1 ::= name <- s | name <- cond | 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)
i2 ::= type names | if (cond) scope else scope | return (t)? |
    while (cond) scope | do scope while (cond) | for (i1?; cond? ; i1?) scope | continue | break
T ::= type | void
type ::= int([[]]* | tuple | code
args ::= t | t (, t)*
s ::= t | name
t ::= name | N
N ::= (+|-)? [1-9][0-9]*
op ::= + | - | * | & | << | >> | < | <= | = | >= | >
name ::= [a-zA-Z_] [a-zA-Z_0-9]*
cond ::= t op t
names ::= name | name (, name)*
LC example 1: while loop

```java
void main (){
    int index
    index <- 0
    while (index < 10) {
        index <- index + 1
    }
    return
}
```
p ::= f+
f ::= T name ( (type name)* ) scope
scope ::= { i* }
i ::= i1 | i2 | scope
i1 ::= name <- s | name <- cond | 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)
i2 ::= type names | if (cond) scope else scope | return (t)? |
   while (cond) scope | do scope while (cond) | for (i1; cond; i1) scope | continue | break
T ::= type | void
type ::= int([[]])* | tuple | code
args ::= t | t (, t)*
s ::= t | name
t ::= name | N
N ::= (+|-)? [1-9][0-9]*
op ::= + | - | * | & | << | >> | < | <= | = | => | >
name ::= [a-zA-Z_] [a-zA-Z_0-9]*
cond ::= t op t
names ::= name | name (, name)*
LC example 2: do while loop

void main (){
    int index
    index <- 0
    do {
        index <- index + 1
    } while (index < 10)
    return
}

\[ p ::= f^+ \]
\[ f ::= T \text{name } ( (\text{type name})^* ) \text{ scope} \]
\[ \text{scope} ::= \{ i^* \} \]
\[ i ::= i1 \mid i2 \mid \text{scope} \]
\[ i1 ::= \text{name } <- s \mid \text{name } <- \text{cond} \mid \text{name } <- \text{name }([t])^+ \mid \text{name }([t])^+ <- s \mid \text{name } <- \text{length } \text{name } t \mid \text{name }((\text{args}?) ) \mid \text{name } <- \text{name }((\text{args}?) ) \mid \text{print}(t) \mid \text{name } <- \text{new } \text{Array}(\text{args}) \mid \text{name } <- \text{new } \text{Tuple}(t) \]
\[ i2 ::= \text{type } \text{names} \mid \text{if } (\text{cond}) \text{ scope } \text{else } \text{scope} \mid \text{return } (t)? \mid \text{while } (\text{cond}) \text{ scope } \mid \text{do scope } \text{while } (\text{cond}) \mid \text{for } (i1?; \text{cond}?; i1?) \text{ scope } \mid \text{continue } \mid \text{break} \]
\[ T ::= \text{type} \mid \text{void} \]
\[ \text{type} ::= \text{int}([])^* \mid \text{tuple} \mid \text{code} \]
\[ \text{args} ::= t \mid t (, t)^* \]
\[ s ::= t \mid \text{name} \]
\[ t ::= \text{name} \mid N \]
\[ N ::= (+|-)? [1-9][0-9]^* \]
\[ \text{op} ::= + \mid - \mid * \mid & \mid << \mid >> \mid < \mid <= \mid = \mid >= \mid > \]
\[ \text{name} ::= [a-zA-Z_][a-zA-Z_0-9]^* \]
\[ \text{cond} ::= t \text{op } t \]
\[ \text{names} ::= \text{name} \mid \text{name } (, \text{name})* \]
null
Now that you know LC

• Rewrite all your LB programs in LC
Outline

• LC

• Parsing

• Translating high level control structures
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

```cpp
if (index < 10) {
    index <- index + 1
} else {
    index <- index – 1
}
```

• Problem: you want to generate an instruction “if” before anything that is inside the two scopes

```cpp
struct if_else_rule:
    pegtl::seq<
        str_if,
        ‘(‘, condition_rule, ‘)’,
        scope,
        str_else,
        scope
    > {};
```
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

```c
if (index < 10) {
    index <- index + 1
} else {
    index <- index - 1
}
```

Solution:
• Create a high-level-control-structure (HLCS) stack
• The "if" instruction class includes pointers to the two scopes (then, else, branches)
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

```c
if (index < 10) {
    index <- index + 1
} else {
    index <- index – 1
}
```

```cpp
struct if_else_begin_rule: str_if,
    pegtl::seq<
    str_if,
    {'(', condition_rule, ')'},
    > {};
```

```cpp
struct if_else_rule: if_else_begin_rule,
    scope,
    str_else,
    scope
    > {};
```

• Create an “if” instruction and append it to the current scope
• Push the “if” instruction just created on top of the HLCS stack
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

```c
if (index < 10) {
    index <- index + 1
} else {
    index <- index - 1
}
```

```cpp
struct if_else_begin_rule:
  pegtl::seq<str_if, '(', condition_rule, ')'>;
struct if_else_rule:
  pegtl::seq<if_else_begin_rule, scope, str_else, scope, '>';}
```

At “{“:
• Check if the opening scope should be attached to an “if” instruction:
  • Is there an “if” on the HLCS stack?
  • If no: append the scope to the parent scope
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

if (index < 10) {
    index <- index + 1
} else {
    index <- index - 1
}

At “{“:
• Check if the opening scope should be attached to an “if” instruction:
  • If yes: attach the opening scope to the “if” on top of the HLCS stack
  • If that “if” has both branches with attached scopes, then pop it from the HLCS stack
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

```cpp
if (index < 10) {
    index <- index + 1
} else {
    index <- index - 1
}
```

```cpp
struct if_else_begin_rule:
    pegtl::seq<
        str_if,
        if_else_begin_rule,,
    >{};

struct if_else_rule:
    pegtl::seq<
        if_else_begin_rule,,
        scope,
        str_else,
        scope
    >{};
```

Attaching the opening scope to the “if” on top of the HLCS stack

• If the top “if” doesn’t have a scope on the ”then” branch: attach the opening scope to the “then” branch
LC parser almost the same as the LB one

• Only difference: you need to parse the high level control structures

```cpp
if (index < 10) {
    index <- index + 1
} else {
    index <- index - 1
}
```

```cpp
struct if_else_begin_rule: pegtl::seq<
    str_if,
    '(', condition_rule, ')
>
struct if_else_rule: pegtl::seq<
    if_else_begin_rule,
    scope,
    str_else,
    scope
>
> {};
```

Attaching the opening scope to the “if” on top of the HLCS stack
• If the top “if” has a scope on the ”then” branch: attach the opening scope to the “else” branch
Outline

• LC

• Parsing

• Translating high level control structures
Translation of the LC "if" to LB code

• Create 3 new LB labels: :LT, :LF, :LE
• Translate the LC condition to LB code and append the LB code to the current innermost scope
• Append the :LT label
• Translate the "then" scope
• Append a jump to :LE
• Append the :LF label
• Translate the "else" scope
• Append the :LE label

```
if (index < 10) {
    index <- index + 1
} else {
    index <- index – 1
}
```
Homework #8

Ignore loops for this homework

Write a compiler that translates an LC program (.c) to an LB one
  • You need to generate prog.b

  • You need to pass all tests in the framework
Extra points for LC

• H9 (1 point): extend H8 to translate all loops
p ::= f^*
f ::= T name ( (type name)* ) scope
scope ::= { i^* }
i ::= i1 | i2 | scope
i1 ::= name <- s | name <- cond | 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)
i2 ::= type names | if (cond) scope else scope | return (t)? |
            while (cond) scope | do scope while (cond) | for (i1?; cond? ; i1?) scope | continue | break
T ::= type | void
type ::= int([[]])^* | tuple | code
args ::= t | t (, t)^*
s ::= t | name
t ::= name | \(N\)
N ::= (+|-)? [1-9][0-9]^*
op ::= + | - | * | & | << | >> | | <= | = | >= | >
name ::= [a-zA-Z_][a-zA-Z_0-9]^*
cond ::= t op t
names ::= name | name (, name)^*
Extra points for LC

• H9 (1 point): extend H8 to translate all loops

• H10 (1 point): extend H9 to translate “if” instruction without the “else” branch (the LC grammar needs to increase to add this case)

```c
void main (){
    int index
    index <- 0
    if (index < 10) {
        index <- index + 1
    }
    return
}
```
LC extended

\[ p ::= f^+ \]

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

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

\[ i ::= i1 \ | \ i2 \ | \ \text{scope} \]

\[ i1 ::= \text{name} <- s | \text{name} <- \text{cond} | \text{name} <- \text{name}([t])^+ | \text{name}([t])^+ <- s | \text{name} <- \text{length name t} | \]

\[ \text{name (args?)} | \text{name <- name (args?)} | \text{print(t)} | \]

\[ \text{name <- new Array(args)} | \text{name <- new Tuple(t)} \]

\[ i2 ::= \text{type names} | \text{if (cond) scope else scope} | \text{if (cond) scope} | \text{return (t)?} | \]

\[ \text{while (cond) scope} | \text{do scope while (cond)} | \text{for (i1?; cond? ; i1?) scope} | \text{continue} | \text{break} \]

\[ T ::= \text{type} | \text{void} \]

\[ \text{type} ::= \text{int([])}^* | \text{tuple} | \text{code} \]

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

\[ s ::= t | \text{name} \]

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

\[ N ::= (+|-)? [1-9][0-9]^* \]

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

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

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

\[ \text{names} ::= \text{name} | \text{name (, name)}^* \]
Extra points for LC

• H9 (1 point): extend H8 to translate all loops

• H10 (1 point): extend H9 to translate “if” instruction without the “else” branch (the LC grammar needs to increase to add this case)

• H11 (1 point): the binary your compilers generate from LC/tests/competition2020.c is faster than mine