

# Tentative Schedule

- 20/12 Interpreter+ Code Generation
- 27/12 Code Generation for Control Flow
- 3/1 Activation Records
- 10/1 Program Analysis
- 17/1 Register Allocation
- 24/1 Assembler/Linker/Loader
- 31/1 Garbage Collection
- Object Oriented

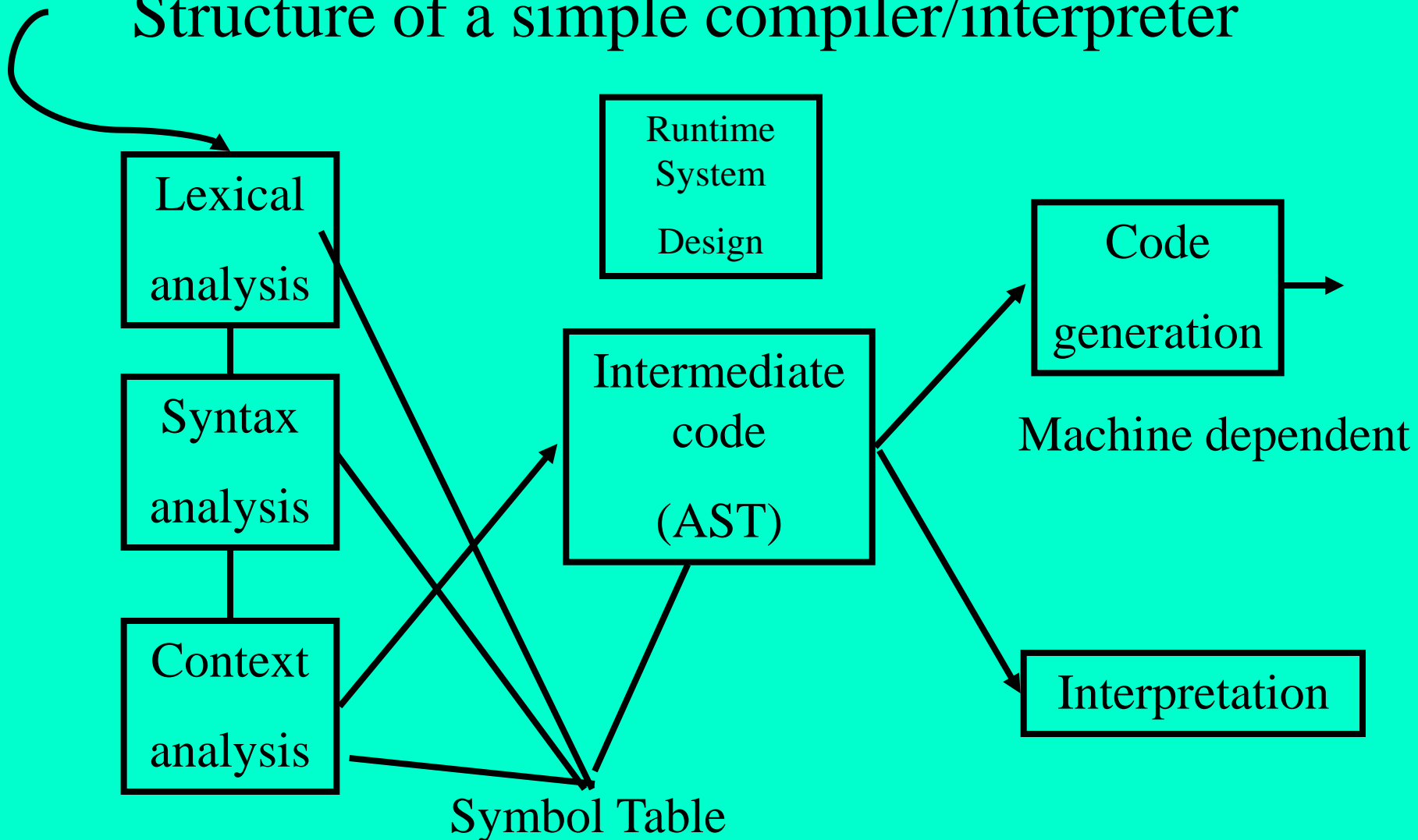
# Building Interpreters

Mooly Sagiv

<http://www.cs.tau.ac.il/~msagiv/courses/wcc11-12.html>

Chapter 4

# Structure of a simple compiler/interpreter



PL dependent

PL+paradigm dependent

# Types of Interpreters

- Recursive
  - Recursively traverse the tree
  - Uniform data representation
  - Conceptually clean
  - Excellent error detection
  - 1000x slower than compiler
- Iterative
  - Closer to CPU
  - One flat loop
  - Explicit stack
  - Good error detection
  - 30x slower than compiler
  - Can invoke compiler on code fragments

# Input language (Overview)

- Fully parameterized expressions
- Arguments can be a single digit

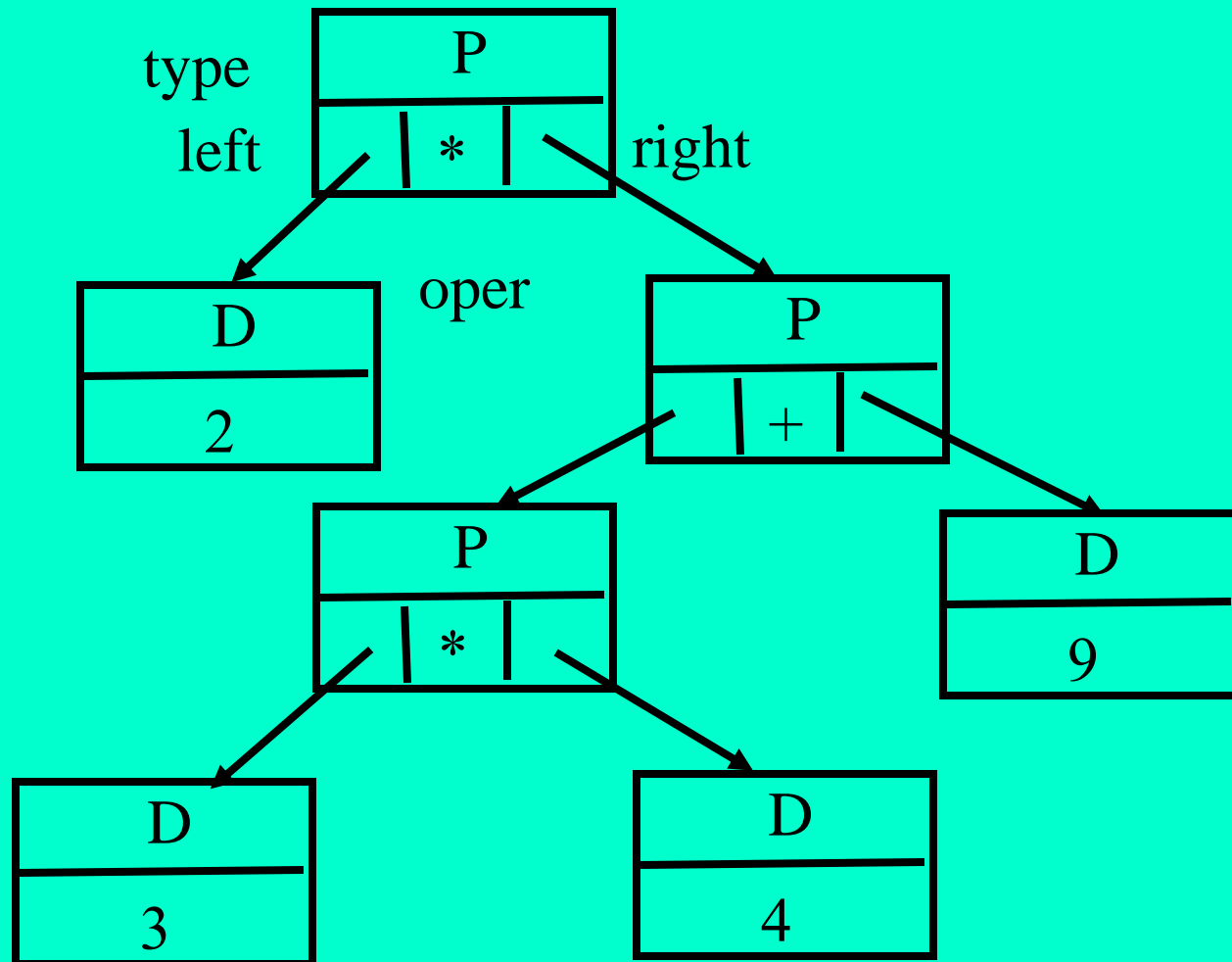
expression  $\rightarrow$  digit | ‘(‘ expression operator expression ‘)’

operator  $\rightarrow$  ‘+’ | ‘\*’

digit  $\rightarrow$  ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

```
#include "parser.h"
#include "backend.h"
static int Interpret_expression(Expression *expr) {
    switch (expr->type) {
        case 'D':
            return expr->value;
            break;
        case 'P': {
            int e_left = Interpret_expression(expr->left);
            int e_right = Interpret_expression(expr->right);
            switch (expr->oper) {
                case '+': return e_left + e_right;
                case '*': return e_left * e_right;
            }
            break;
        }
    }
}
void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
```

# AST for $(2 * ((3 * 4) + 9))$



# Uniform self-identifying data representation

- The types and sizes of program data values are not known when the interpreter is written
- Uniform representation of data types
  - Type
  - Size
- The value is a pointer



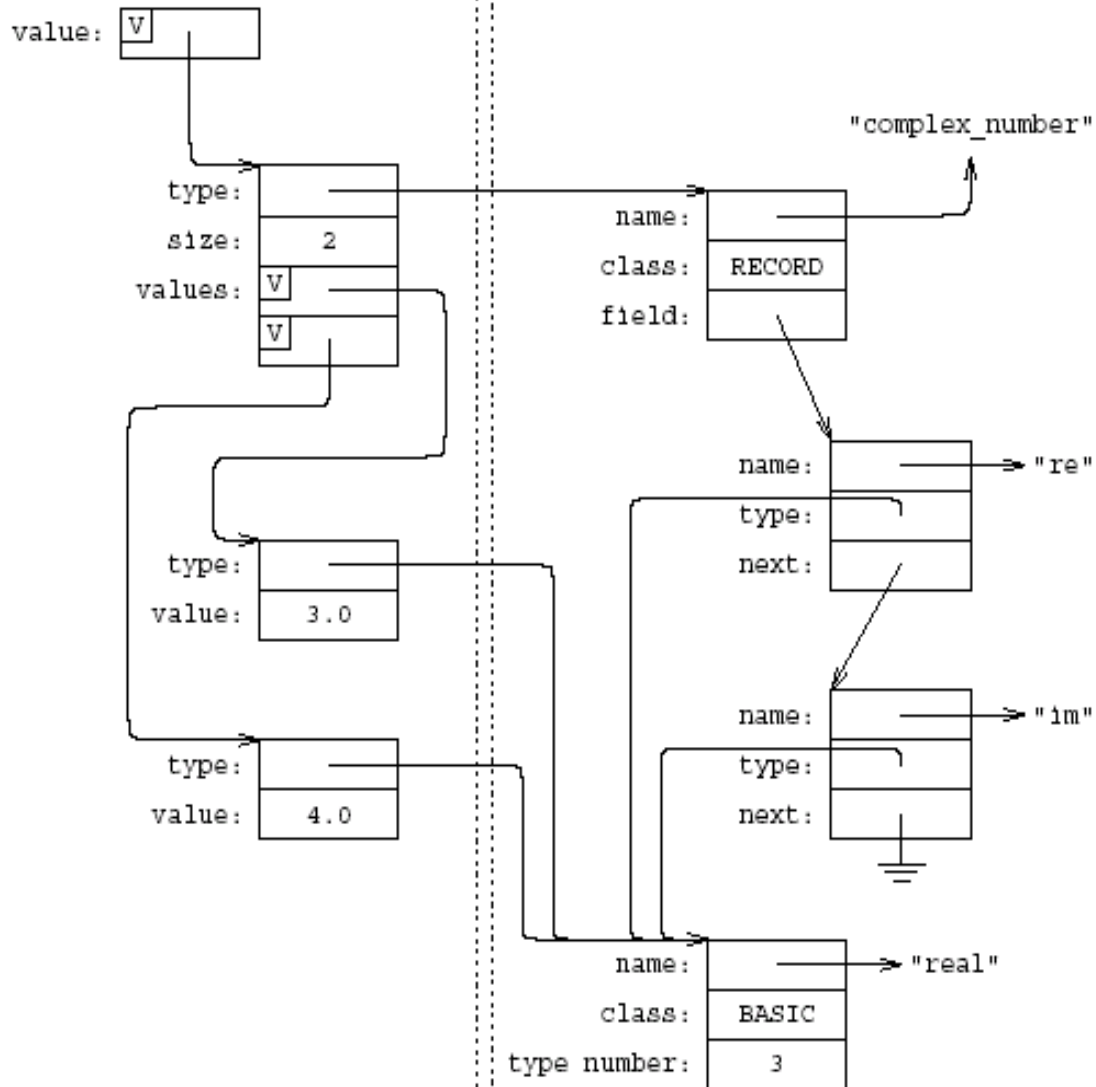
# Example: Complex Number

re:

3.0

im:

4.0



Specific to the given value of type  
complex\_number

Common to all values of type  
complex\_number

# Status Indicator

- Direct control flow of the interpreter
- Possible values
  - Normal mode
  - Errors
  - Jumps
  - Exceptions
  - Return

# Example: Interpreting C Return

PROCEDURE Elaborate return with expression statement (RWE node):

SET Result To Evaluate expression (RWE node . expression);

IF Status . mode /= Normal mode: Return mode;

SET Status . mode To Return mode;

SET Status . value TO Result;

# Interpreting If-Statement

```
PROCEDURE Elaborate if statement (If node):
  SET Result TO Evaluate condition (If node .condition);
  IF Status .mode /= Normal mode: RETURN;
  IF Result .type /= Boolean:
    ERROR "Condition in if-statement is not of type Boolean";
    RETURN;
  IF Result .boolean .value = True:
    Elaborate statement (If node .then part);
  ELSE Result .boolean .value = False:
    // Check if there is an else-part at all:
    IF If node .else part /= No node:
      Elaborate statement (If node .else part);
    ELSE If node .else part = No node:
      SET Status .mode TO Normal mode;
```

# Symbol table

- Stores content of variables, named constants, ...
- For every variable  $V$  of type  $T$ 
  - A pointer to the name of  $V$
  - The file name and the line it is declared
  - Kind of declaration
  - A pointer to  $T$
  - A pointer to newly allocated space
  - Initialization bit
  - Language dependent information (e.g. scope)

# Summary Recursive Interpreters

- Can be implemented quickly
  - Debug the programming language
- Not good for heavy-duty interpreter
  - Slow
  - Can employ general techniques to speed the recursive interpreter
    - Memoization
    - Tail call elimination
    - Partial evaluation

# Memoization

```
int fib(int n)
{
    if (n == 0) return 0 ;
    if (n==1) return 1;
    return fib(n-1) + fib(n-2) ;
}
```

⇒

```
int sfib[100] = {-1, -1, ..., -1}
int fib(int n)
{
    if (sfib[n] > 0) return sfib[n];
    if (n == 0) return 0 ;
    if (n==1) return 1;
    sfib[n] = fib(n-1) + fib(n-2) ;
    return sfib[n];
}
```



# Tail Call Elimination

```
void a(...)  
{  
  ...  
  b();  
}  
void b(){  
code;  
}
```

⇒

```
void a(...)  
{  
  ...  
  code;  
}  
void b(){  
code;  
}
```

# Tail Call Elimination

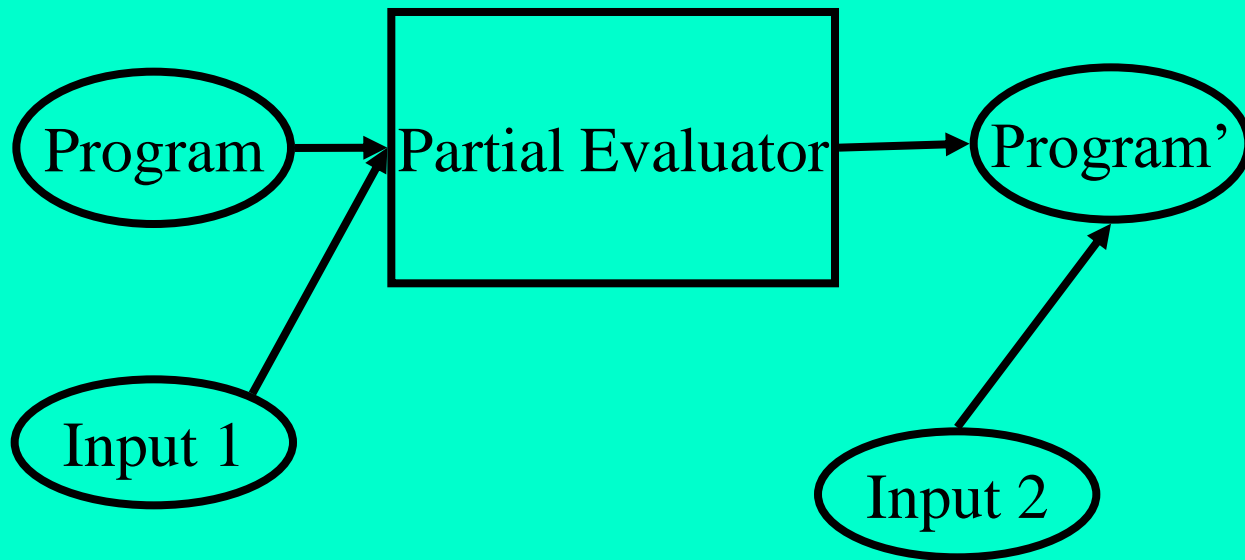
```
void a(int n)
{
  code
  if (n > 0) a(n-1);
}
```

⇒

```
void a(int n)
{
  loop:
  code
  if (n > 0) {
    n = n - 1 ;
    goto loop
  }
```

# Partial Evaluation

- Partially interpret static parts in a program
- Generates an equivalent program



# Example

```
int pow(int n, int e)
{
    if (e==0)
        return 1;

    else return n * pow(n, e-1);
}
```

```
int pow4(int n)
{
    return n * n * n *n;
}
```

e=4

# Example2

```
Bool match(string, regexp)
```

```
{
```

```
  switch(regexp) {
```

```
    ....
```

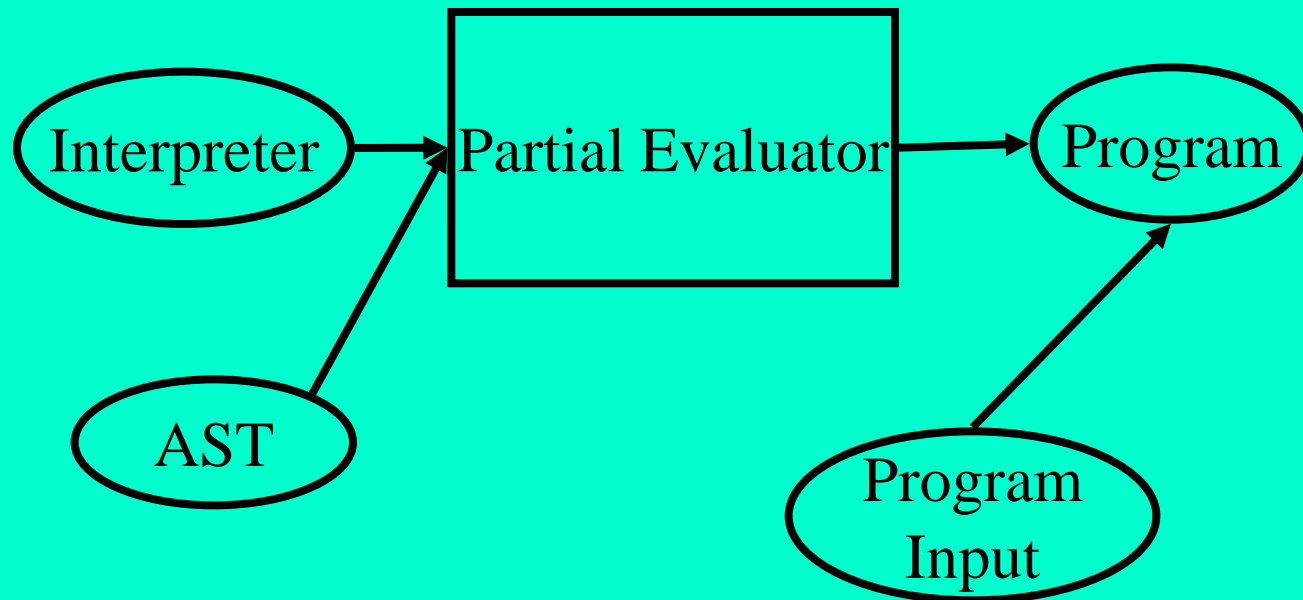
```
  }
```

```
}
```

regexp=a b\*

# Partial Evaluation

## Generalizes Compilation



But . . . .

# Iterative Interpretation

- Closed to CPU
- One flat loop with one big case statement
- Use explicit stack
  - Intermediate results
  - Local variables
- Requires fully annotated threaded AST
  - Active-node-pointer (interpreted node)



# Demo Compiler

```
#include "parser.h" /* for types AST_node and Expression */
#include "thread.h" /* for self check */
                        /* PRIVATE */

static AST_node *Last_node;

static void Thread_expression(Expression *expr) {
    switch (expr->type) {
        case 'D':
            Last_node->successor = expr; Last_node = expr;
            break;
        case 'P':
            Thread_expression(expr->left);
            Thread_expression(expr->right);
            Last_node->successor = expr; Last_node = expr;
            break;
    }
}

                        /* PUBLIC */

AST_node *Thread_start;

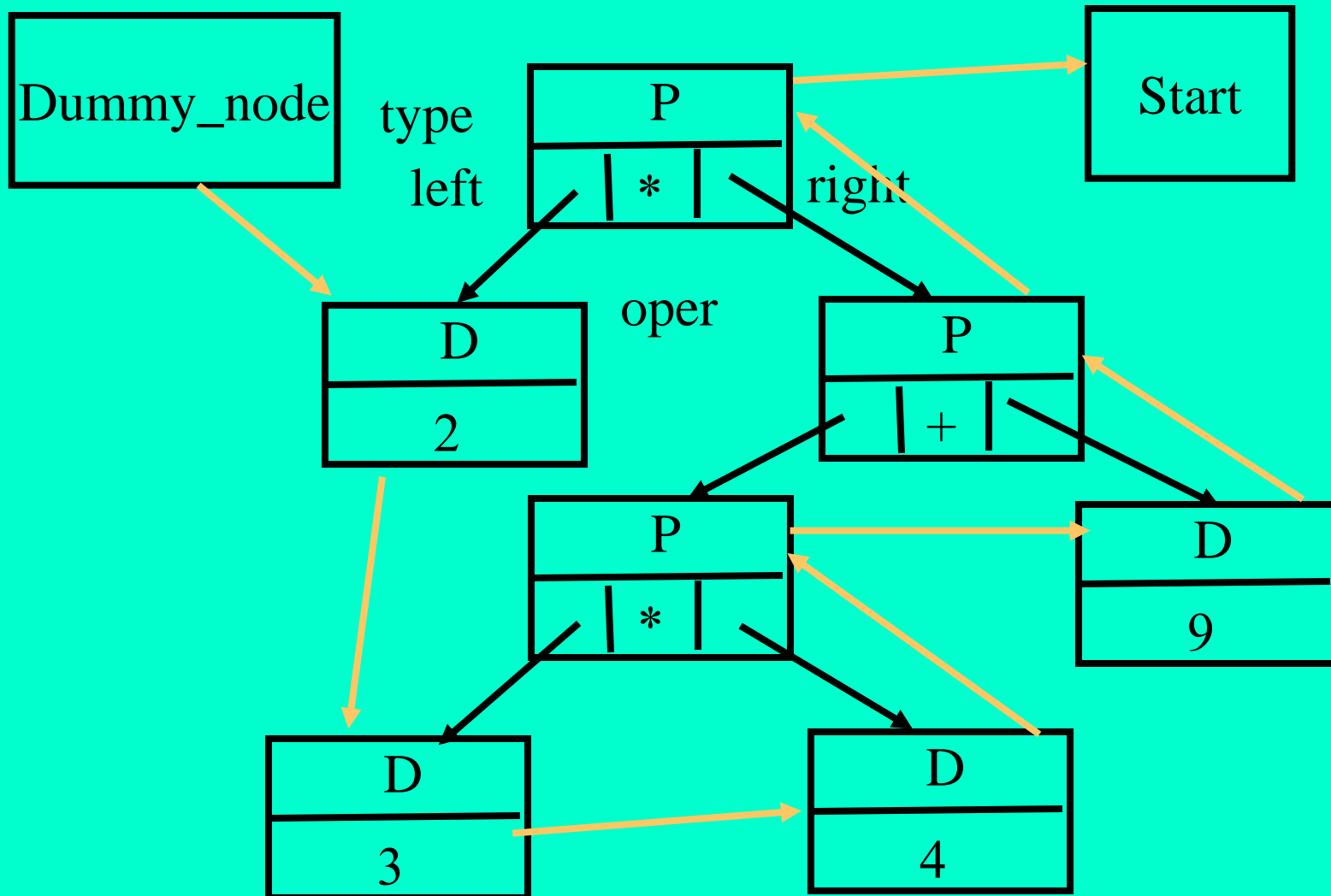
void Thread_AST(AST_node *icode) {
    AST_node Dummy_node;

    Last_node = &Dummy_node; Thread_expression(icode);
    Last_node->successor = (AST_node *)0;
    Thread_start = Dummy_node.successor;
}
```

# Threaded AST

- Annotated AST
- Every node is connected to the immediate successor in the execution
- Control flow graph
  - Nodes
    - Basic execution units
      - expressions
      - assignments
  - Edges
    - Transfer of control
      - sequential
      - while
      - ...

# Threaded AST for $(2 * ((3 * 4) + 9))$



# Demo Compiler

```
static AST_node *Active_node_pointer;

static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
            case 'D':
                Push(expr->value);
                break;
            case 'P': {
                int e_left = Pop(); int e_right = Pop();
                switch (expr->oper) {
                    case '+': Push(e_left + e_right); break;
                    case '*': Push(e_left * e_right); break;
                }
                break;
            }
            Active_node_pointer = Active_node_pointer->successor;
        }
        printf("%d\n", Pop());      /* print the result */
    }
}

/* PUBLIC */

void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
}
```

# C Example

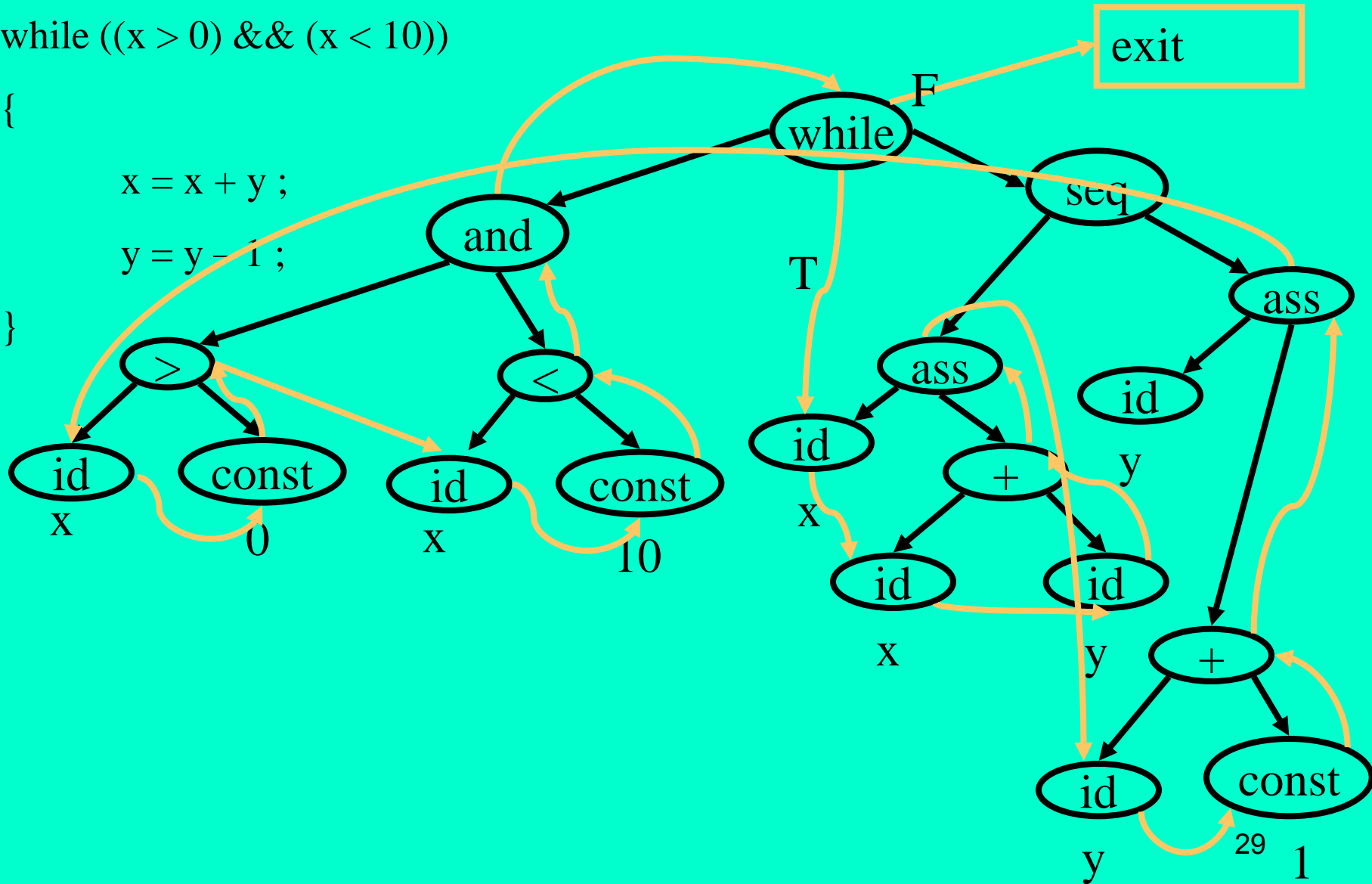
```
while ((x > 0) && (x < 10))
```

```
{
```

```
  x = x + y ;
```

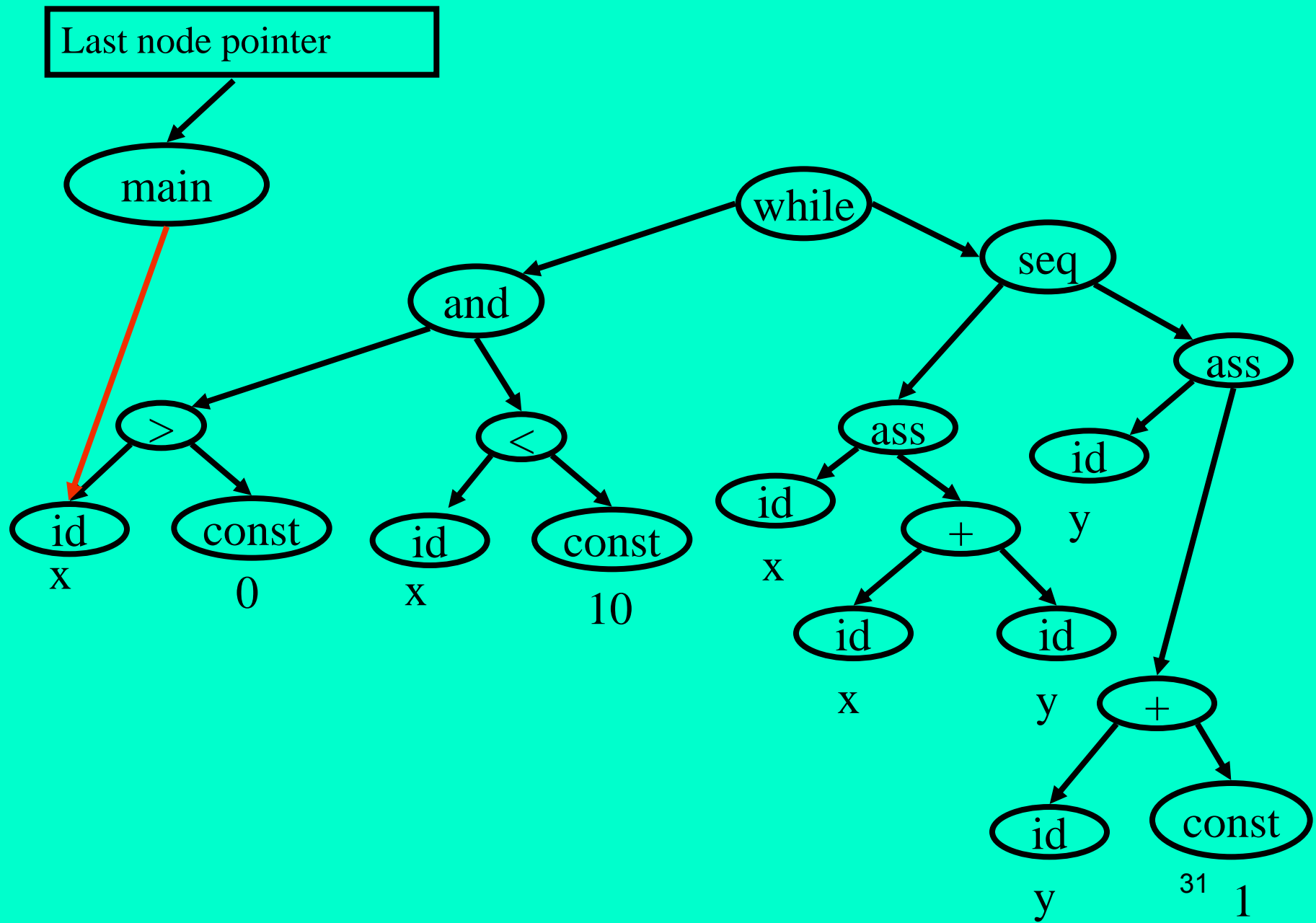
```
  y = y - 1 ;
```

```
}
```

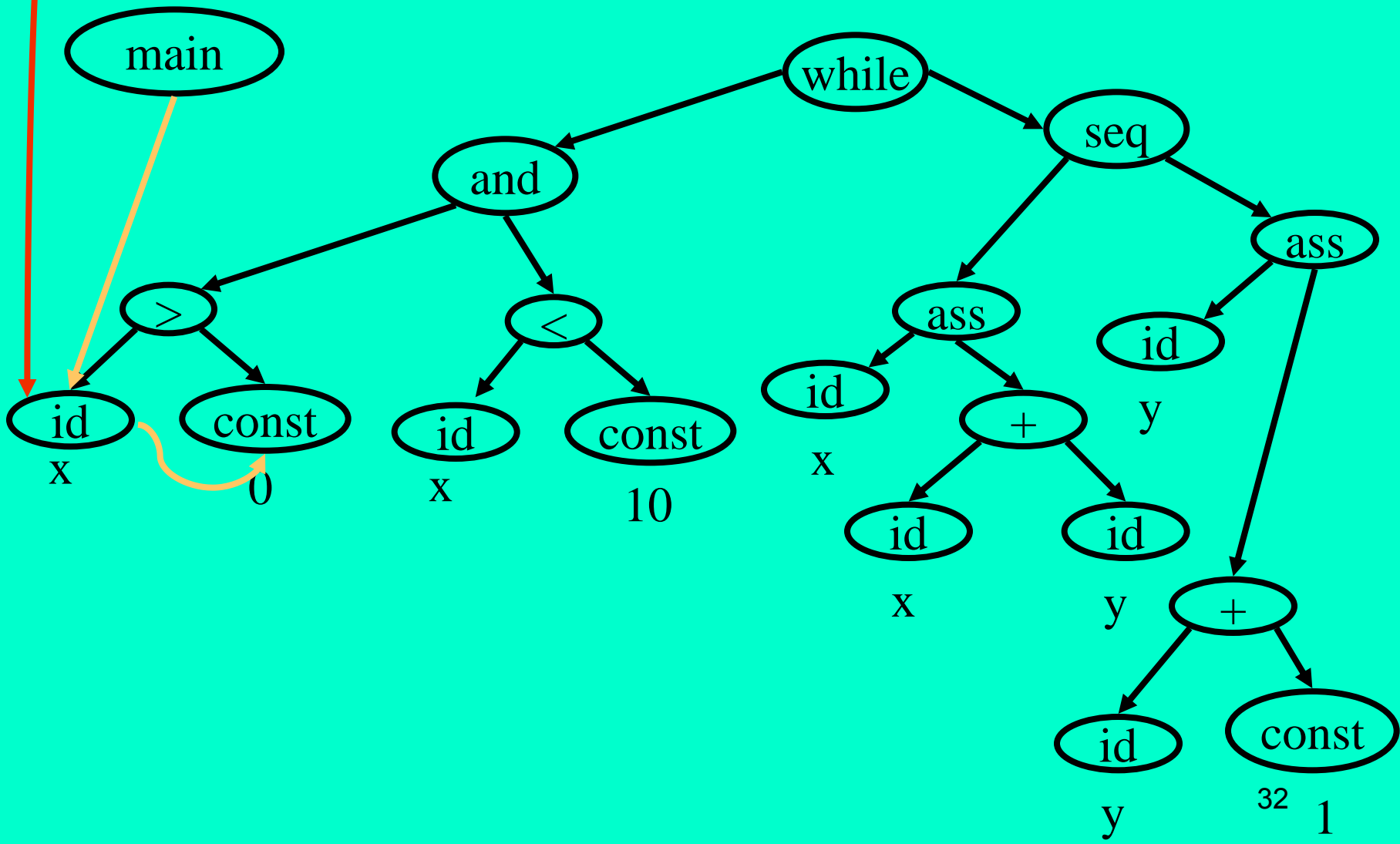


# Threading the AST(3.2.1)

- One preorder AST pass
- Every type of AST has its threading routine
- Maintains **Last node pointer**
  - Global variable
- Set successor of **Last pointer** when node is visited

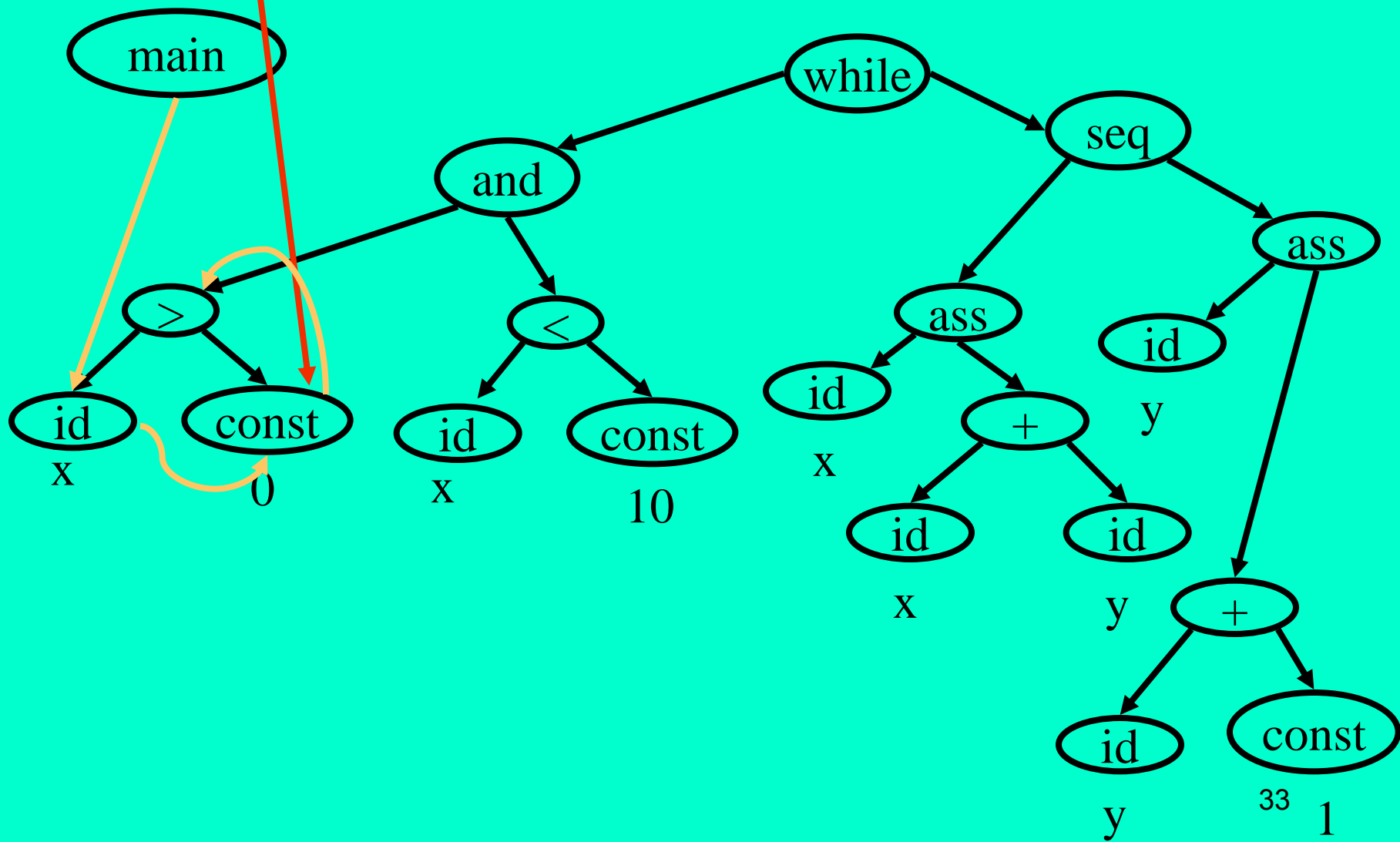


Last node pointer

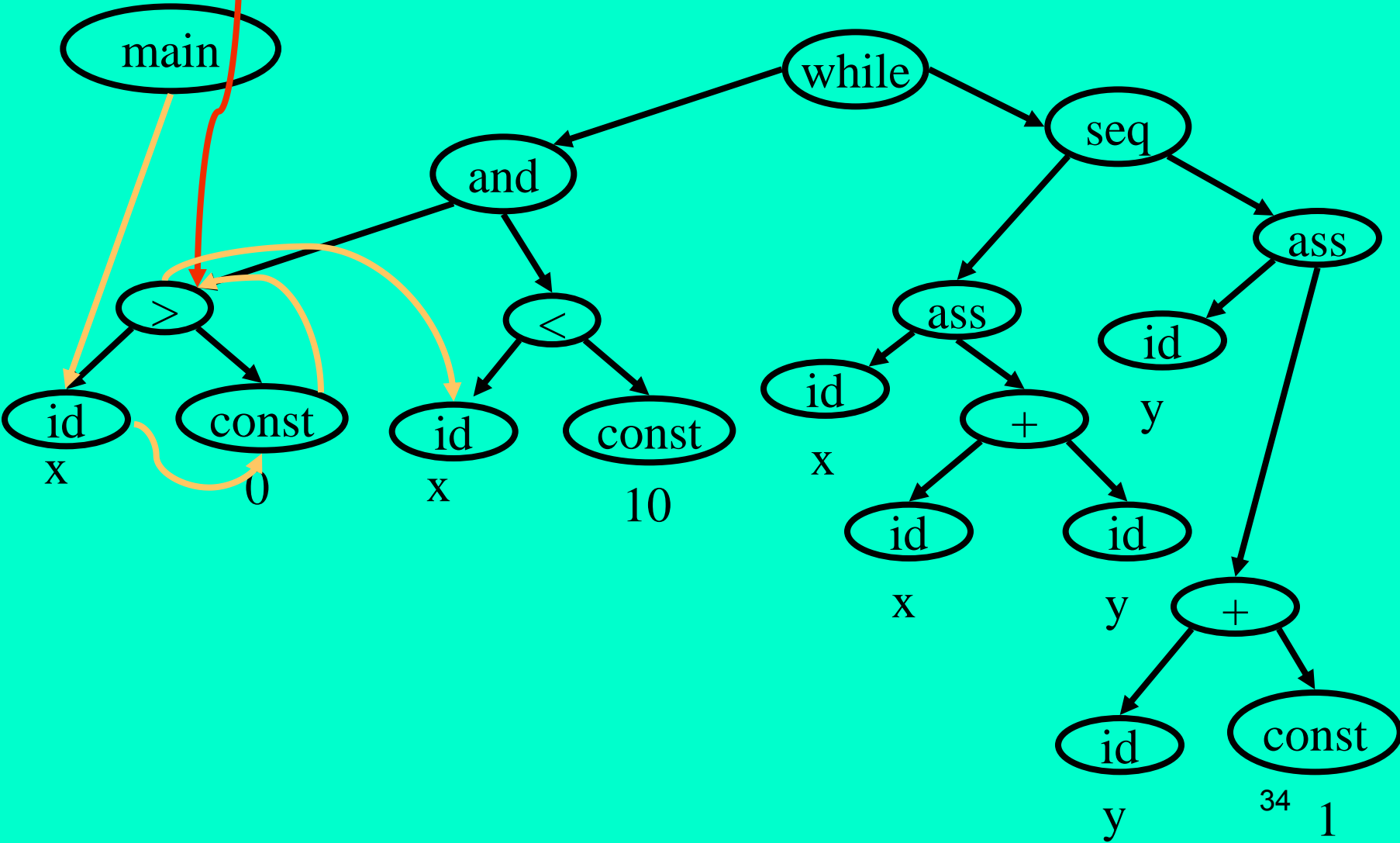




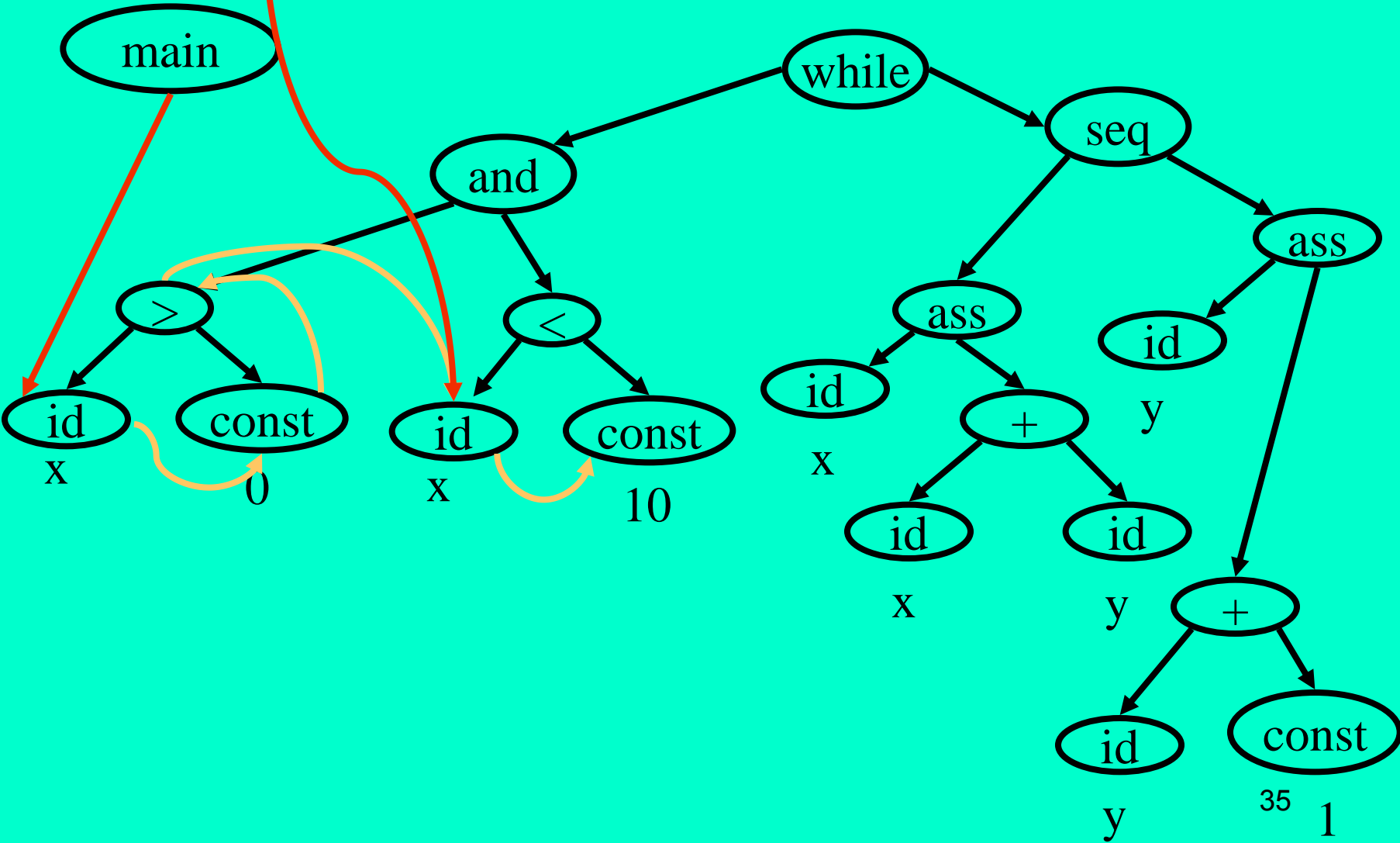
Last node pointer



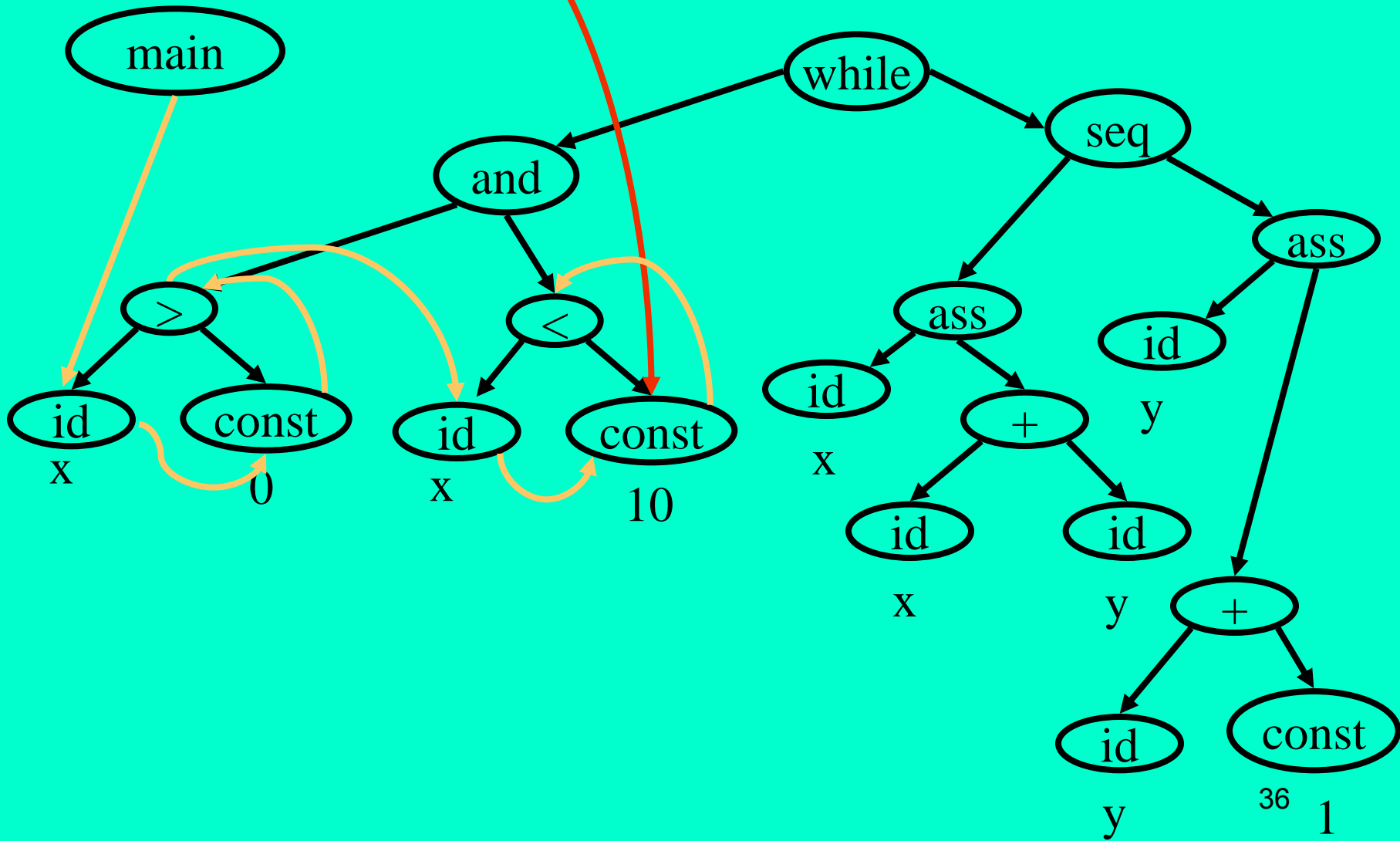
Last node pointer



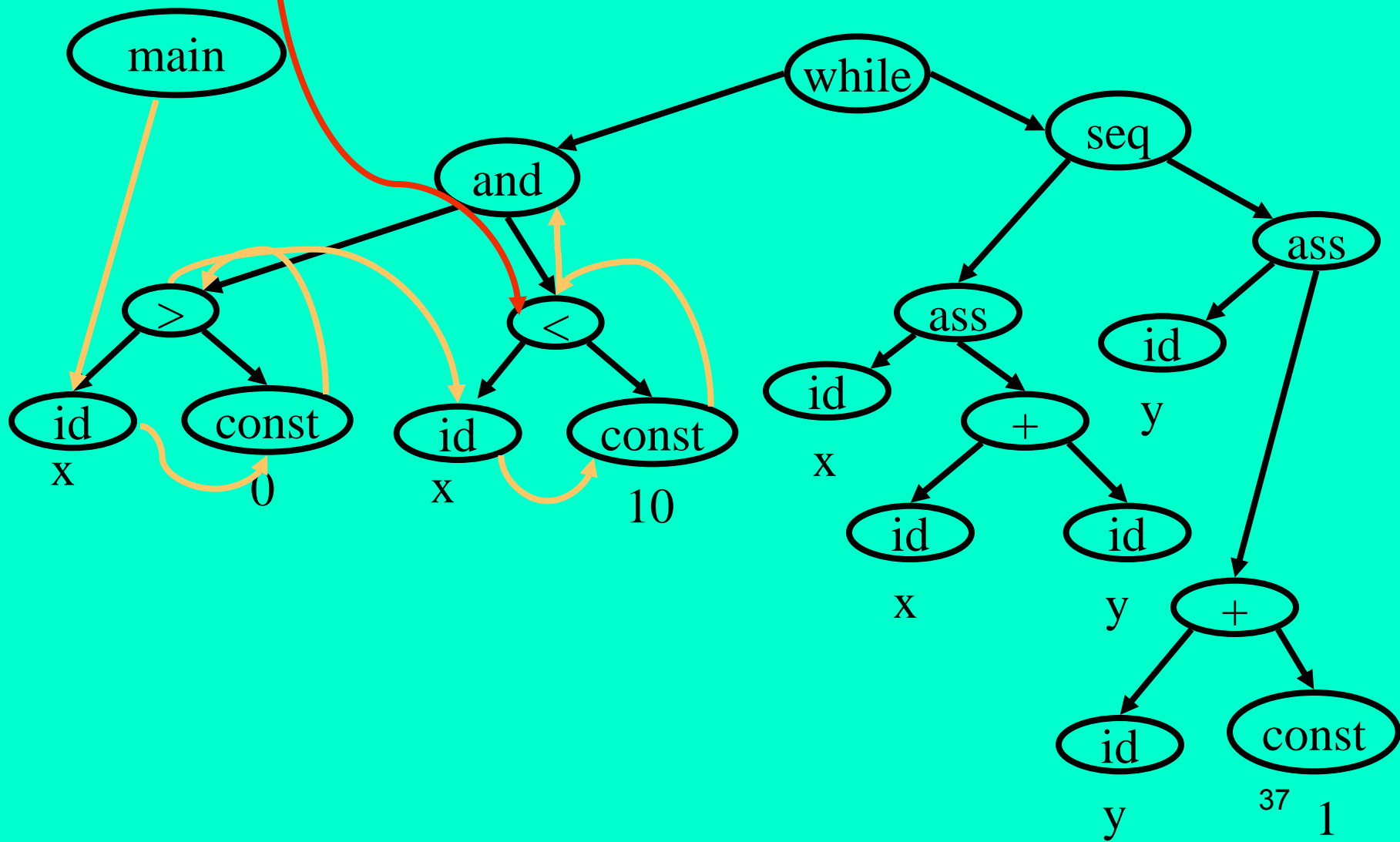
Last node pointer



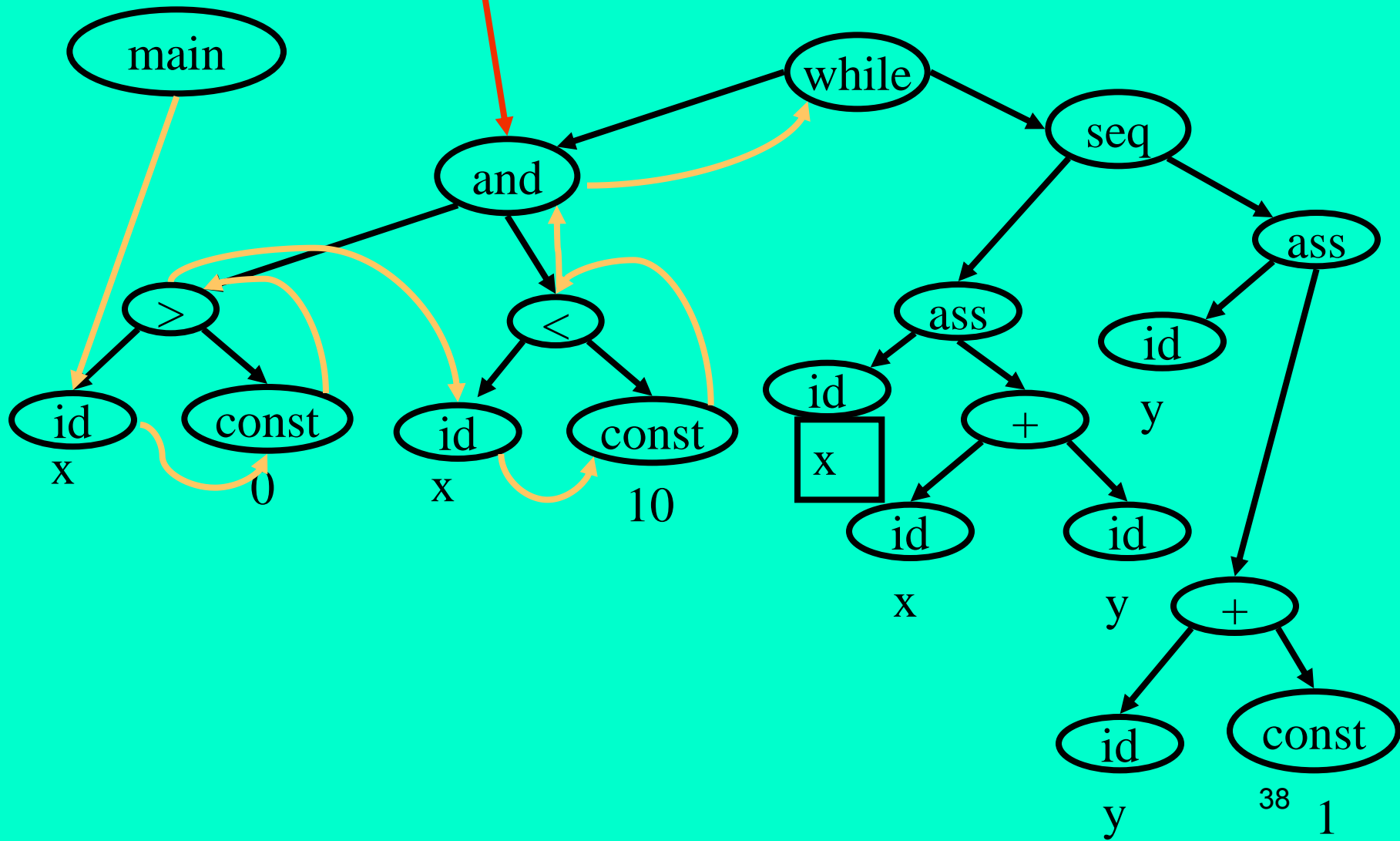
Last node pointer



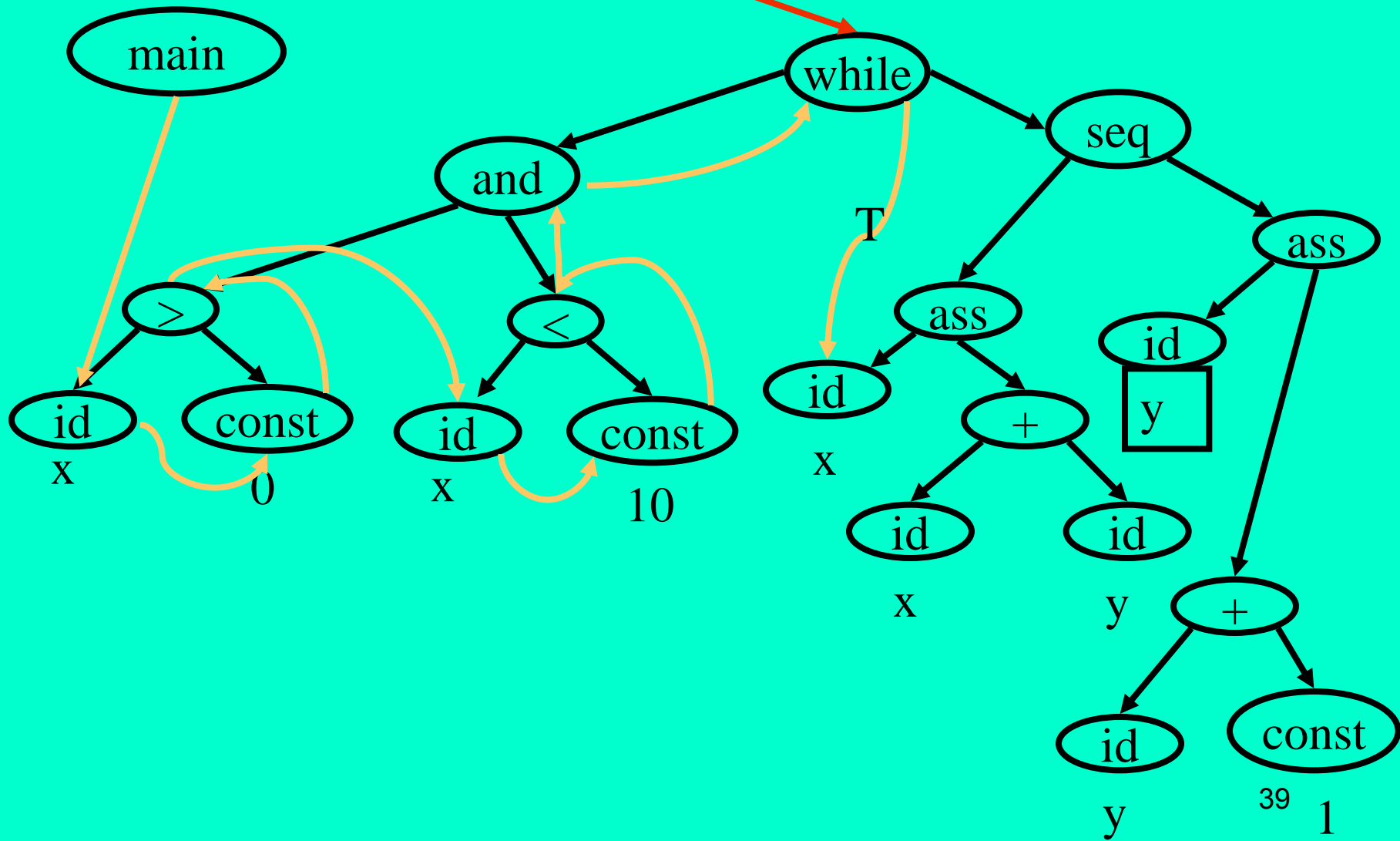
Last node pointer



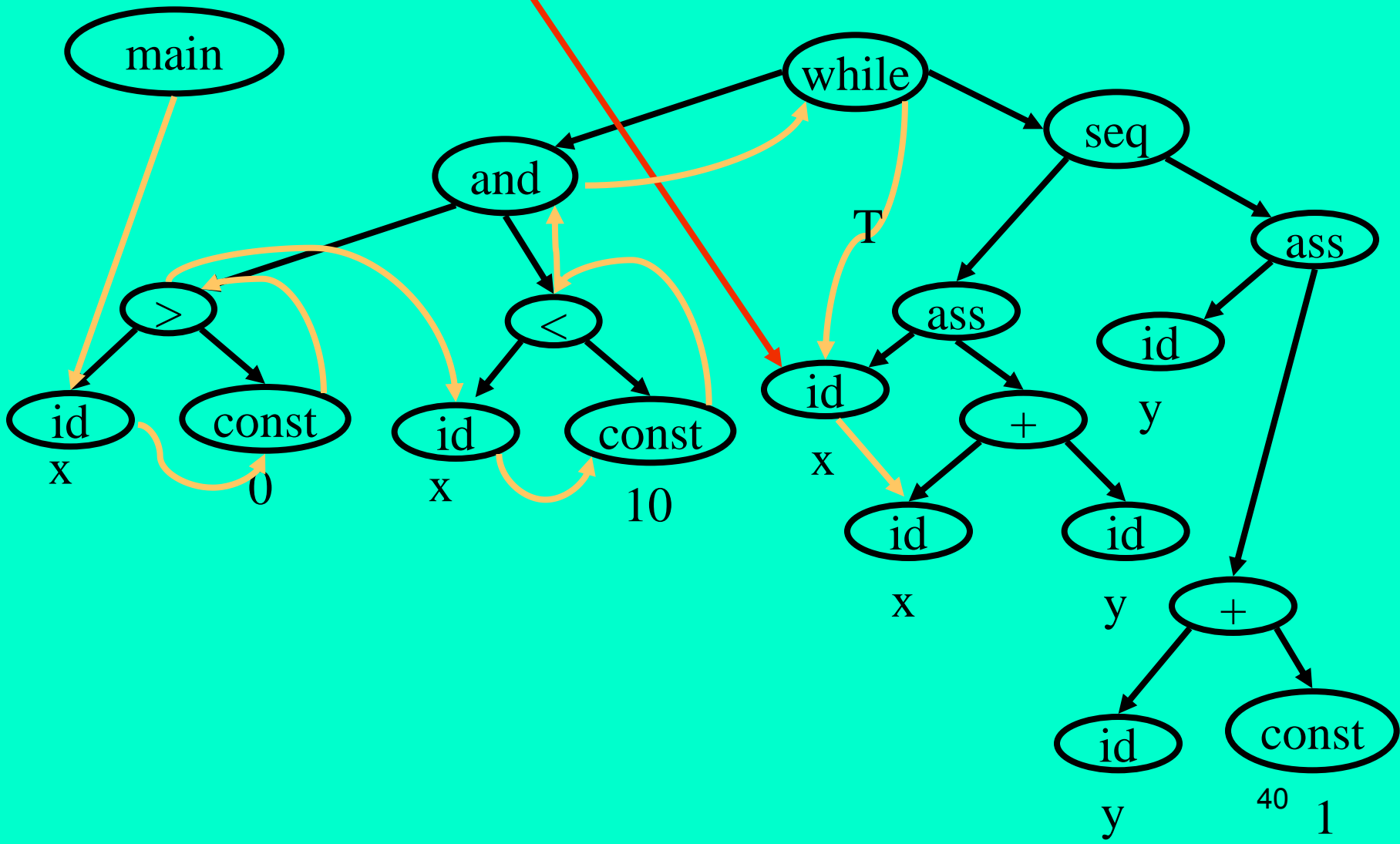
Last node pointer



Last node pointer

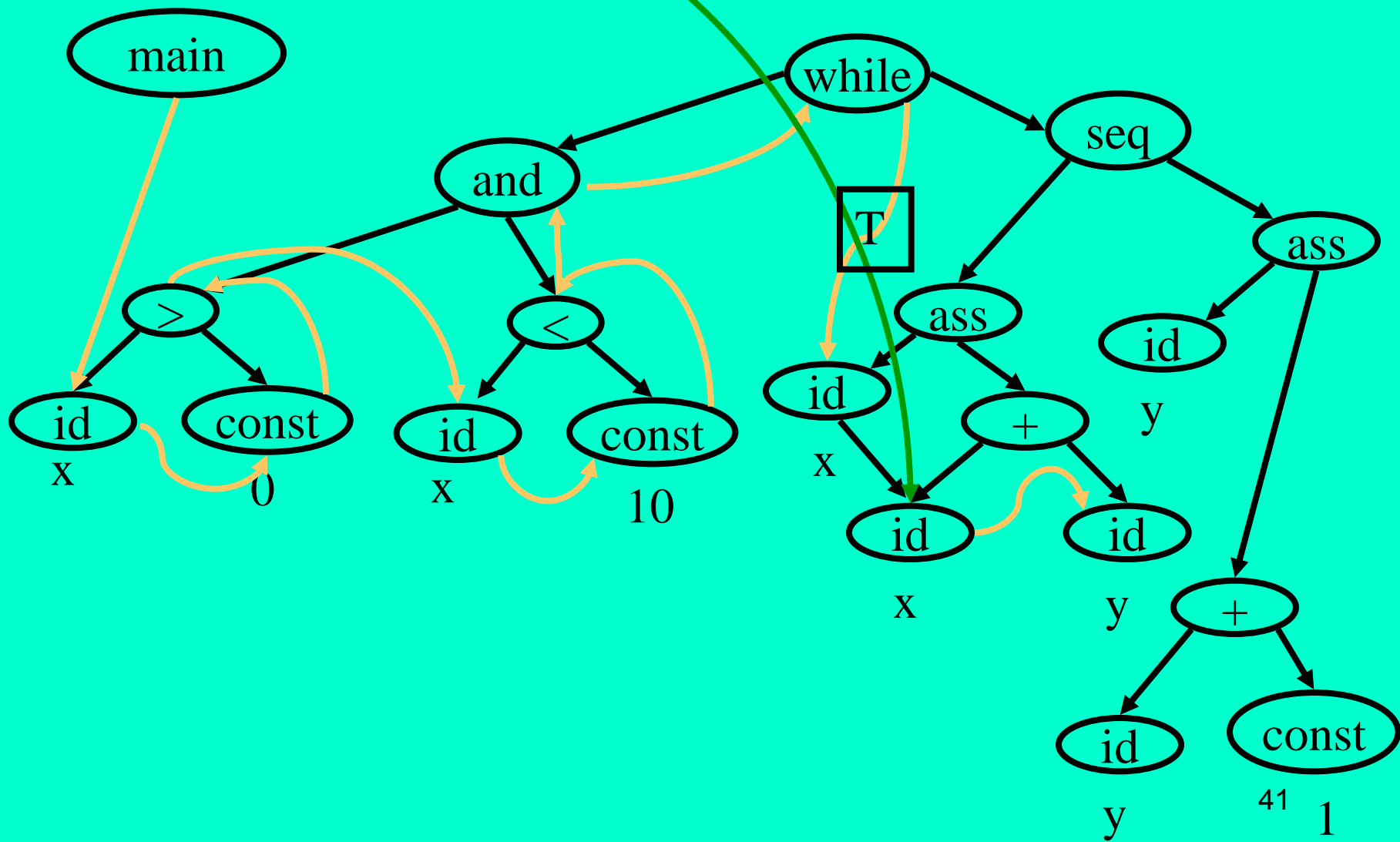


Last node pointer

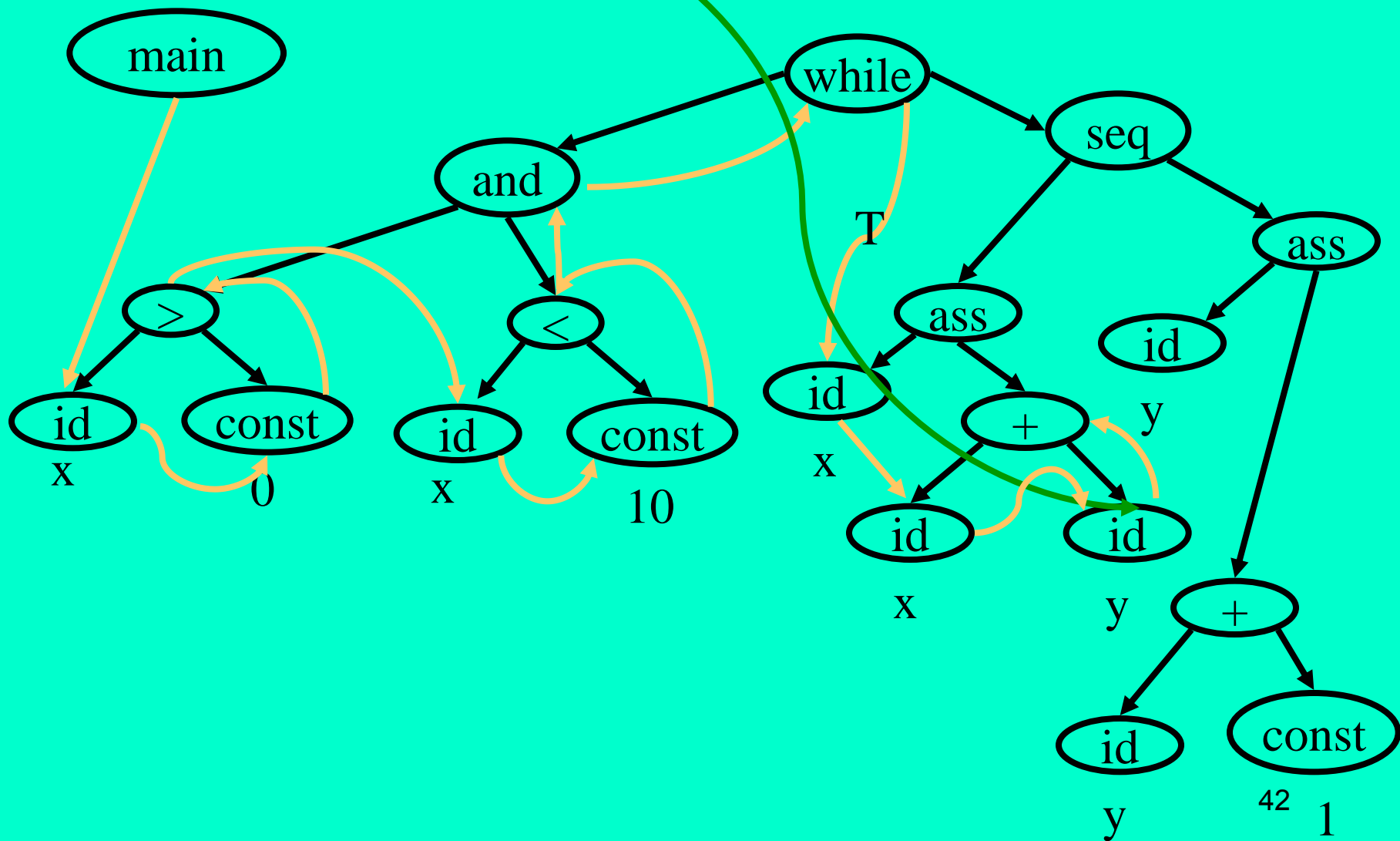


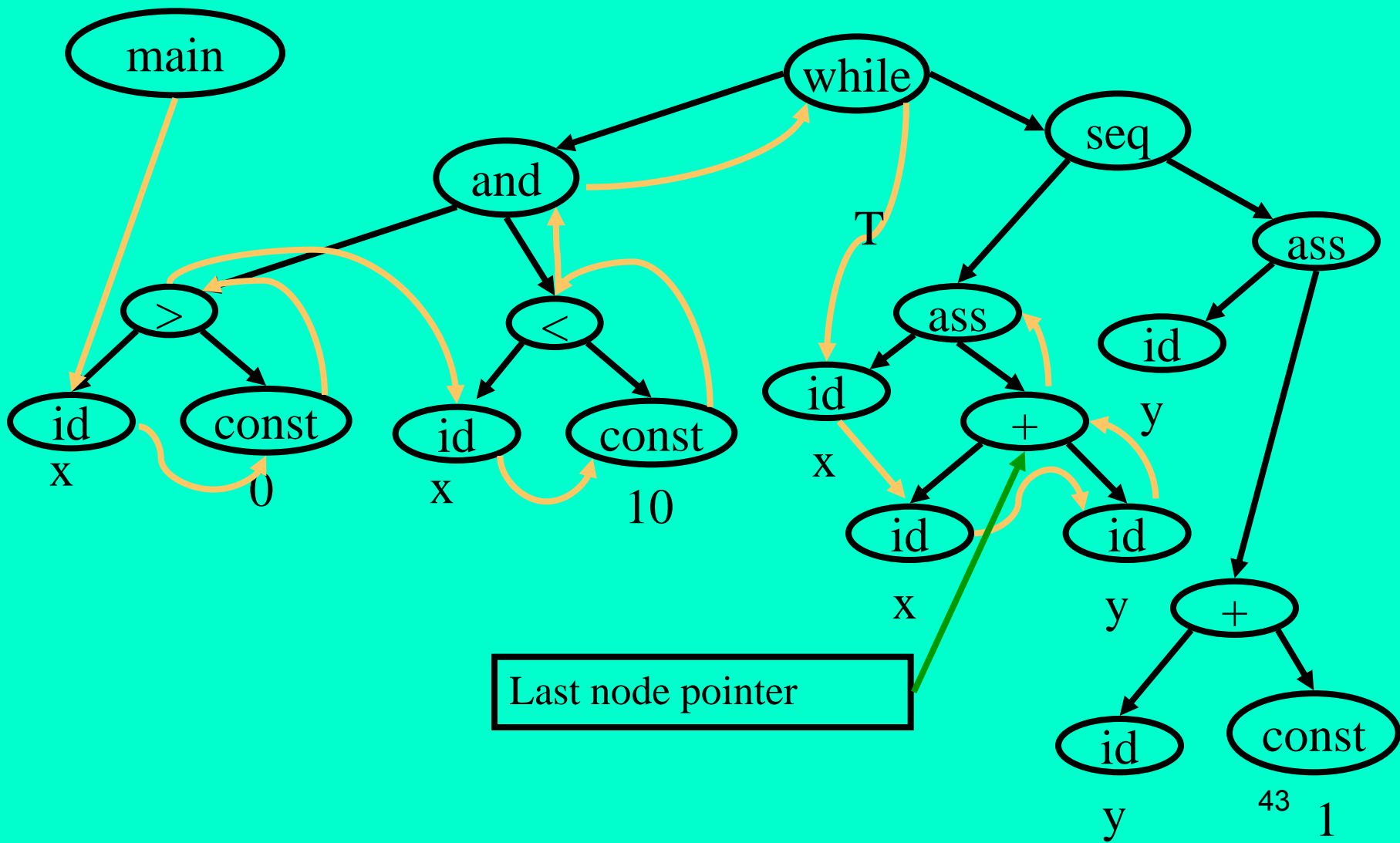


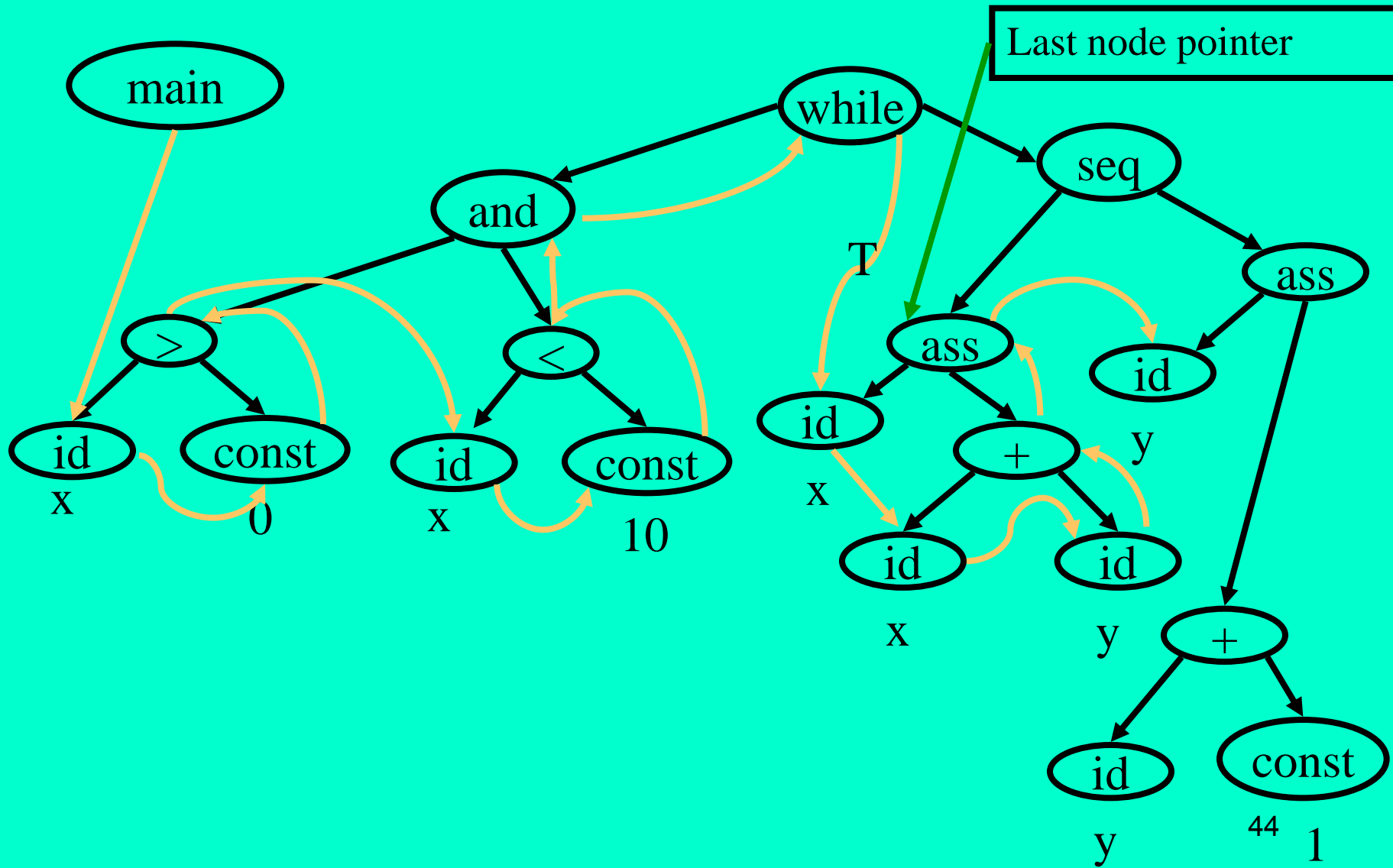
Last node pointer

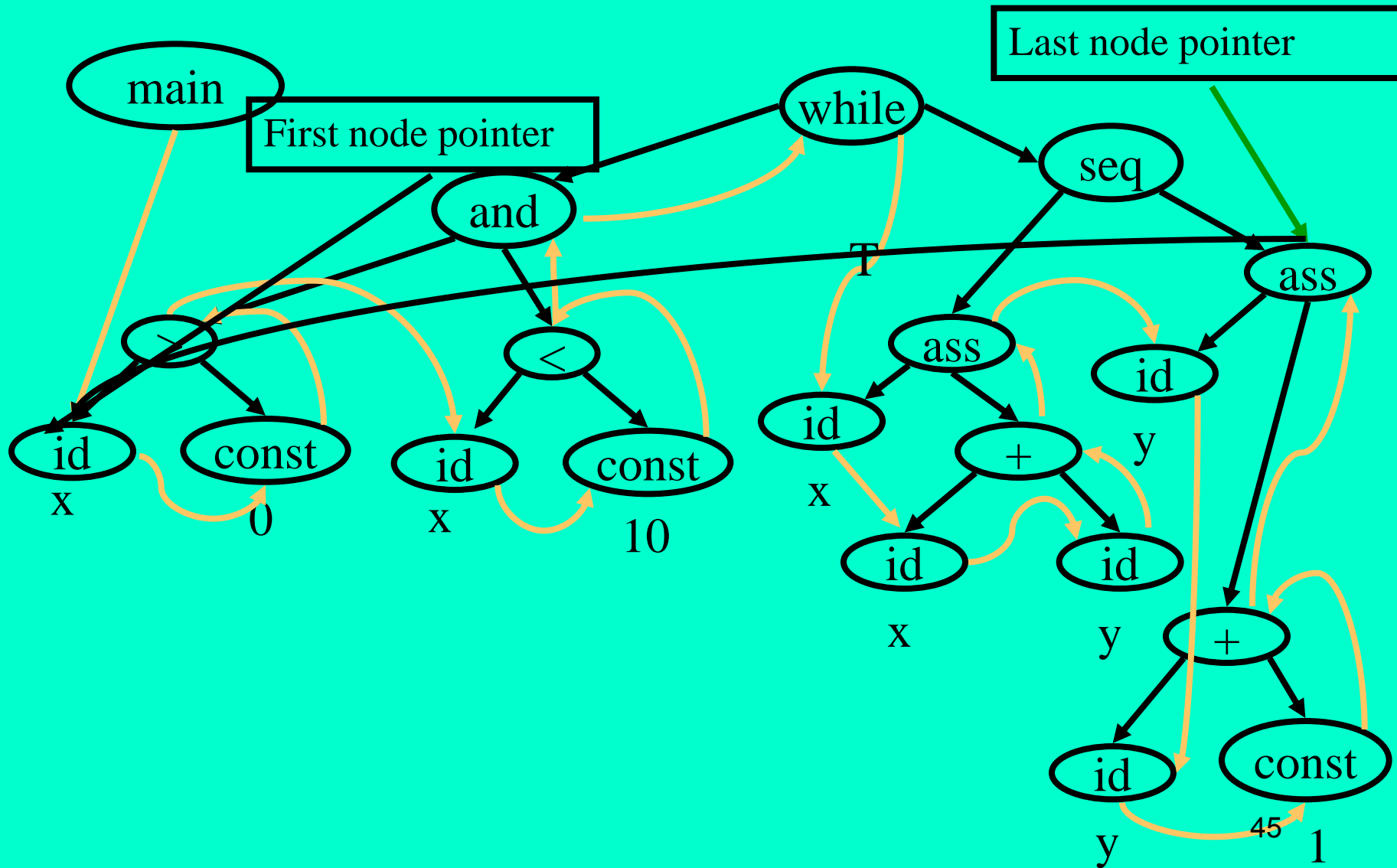


Last node pointer









# Demo Compiler

```
#include "parser.h" /* for types AST_node and Expression */
#include "thread.h" /* for self check */
                        /* PRIVATE */

static AST_node *Last_node;

static void Thread_expression(Expression *expr) {
    switch (expr->type) {
        case 'D':
            Last_node->successor = expr; Last_node = expr;
            break;
        case 'P':
            Thread_expression(expr->left);
            Thread_expression(expr->right);
            Last_node->successor = expr; Last_node = expr;
            break;
    }
}

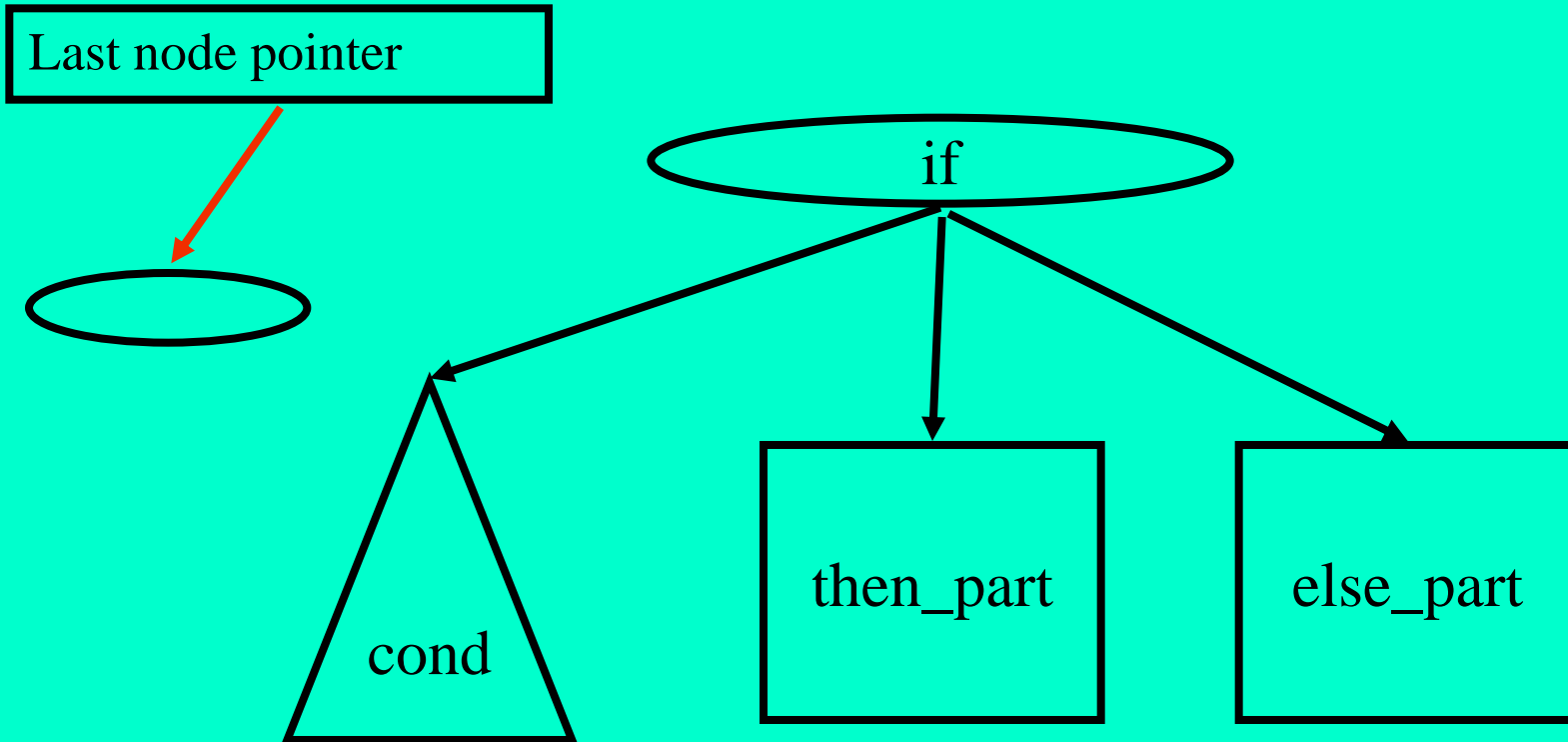
                        /* PUBLIC */

AST_node *Thread_start;

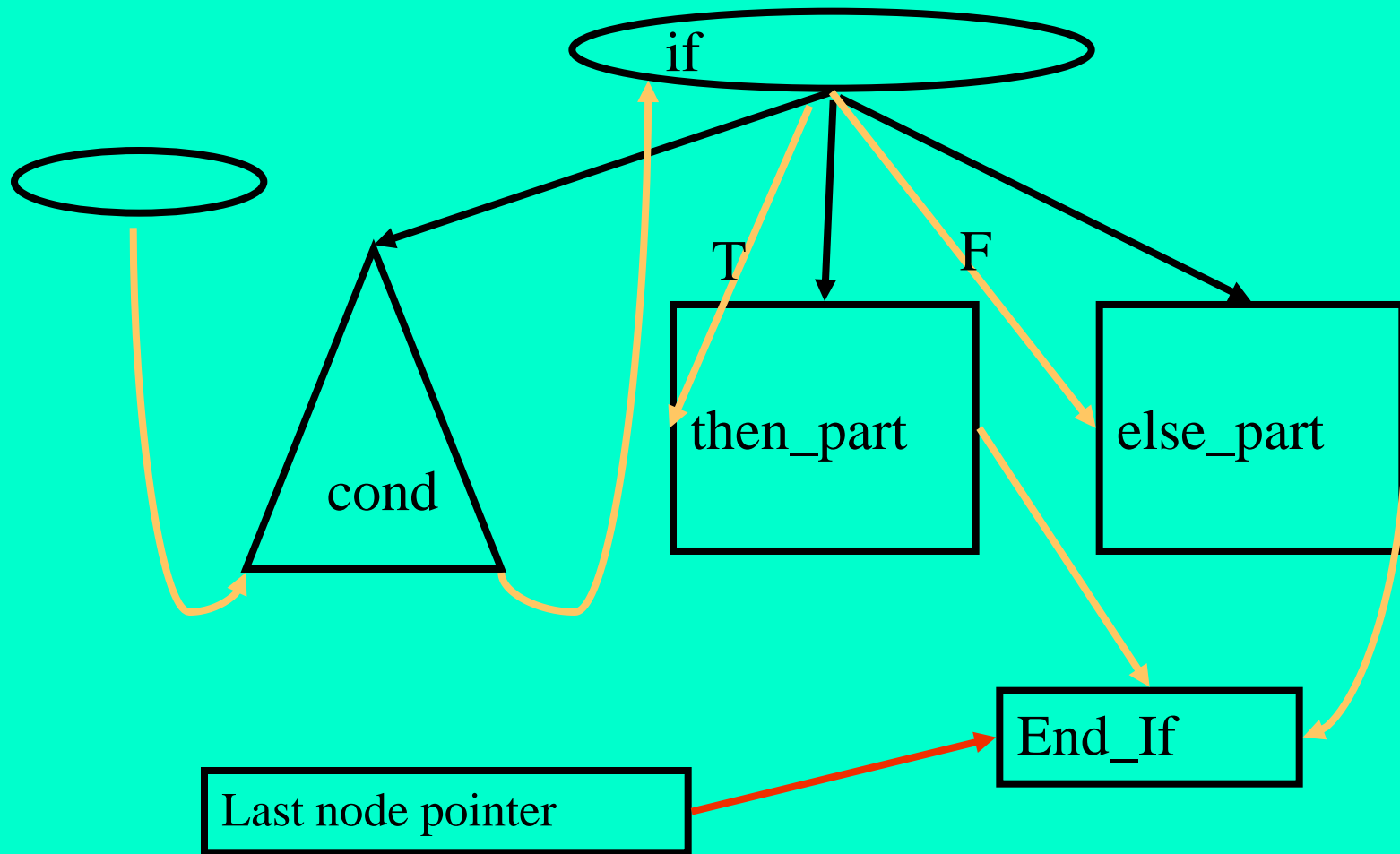
void Thread_AST(AST_node *icode) {
    AST_node Dummy_node;

    Last_node = &Dummy_node; Thread_expression(icode);
    Last_node->successor = (AST_node *)0;
    Thread_start = Dummy_node.successor;
}
```

# Conditional Statement



# Conditional Statement





# Iterative Interpretation

- Closed to CPU
- One flat loop with one big case statement
- Use explicit stack
  - Intermediate results
  - Local variables
- Requires fully annotated threaded AST
  - Active-node-pointer (interpreted node)

# Demo Compiler

```
static AST_node *Active_node_pointer;

static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
            case 'D':
                Push(expr->value);
                break;
            case 'P': {
                int e_left = Pop(); int e_right = Pop();
                switch (expr->oper) {
                    case '+': Push(e_left + e_right); break;
                    case '*': Push(e_left * e_right); break;
                }
                break;
            }
            Active_node_pointer = Active_node_pointer->successor;
        }
        printf("%d\n", Pop());      /* print the result */
    }
}

/* PUBLIC */
void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
}
```

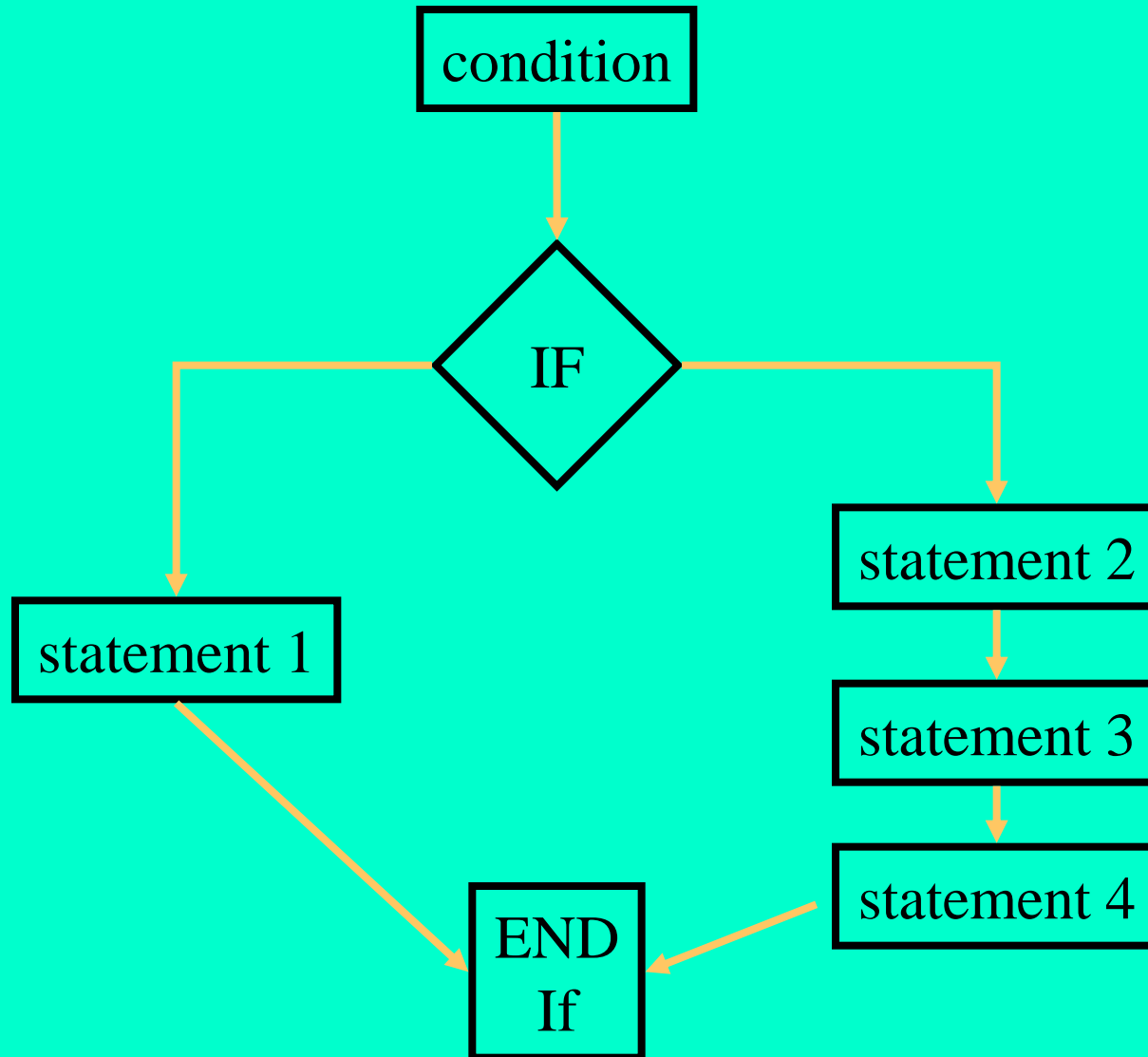
# Conditional Statements

```
WHILE Active node .type /= End of program type:
  SELECT Active node .type:
    CASE ...
    CASE If type:
      // We arrive here after the condition has been evaluated;
      // the Boolean result is on the working stack.
      SET Value TO Pop working stack ();
      IF Value .boolean .value = True:
        SET Active node TO Active node .true successor;
      ELSE Value .boolean .value = False:
        IF Active node .false successor /= No node:
          SET Active node TO Active node .false successor;
        ELSE Active node .false successor = No node:
          SET Active node TO Active node .successor;
    CASE ...
```

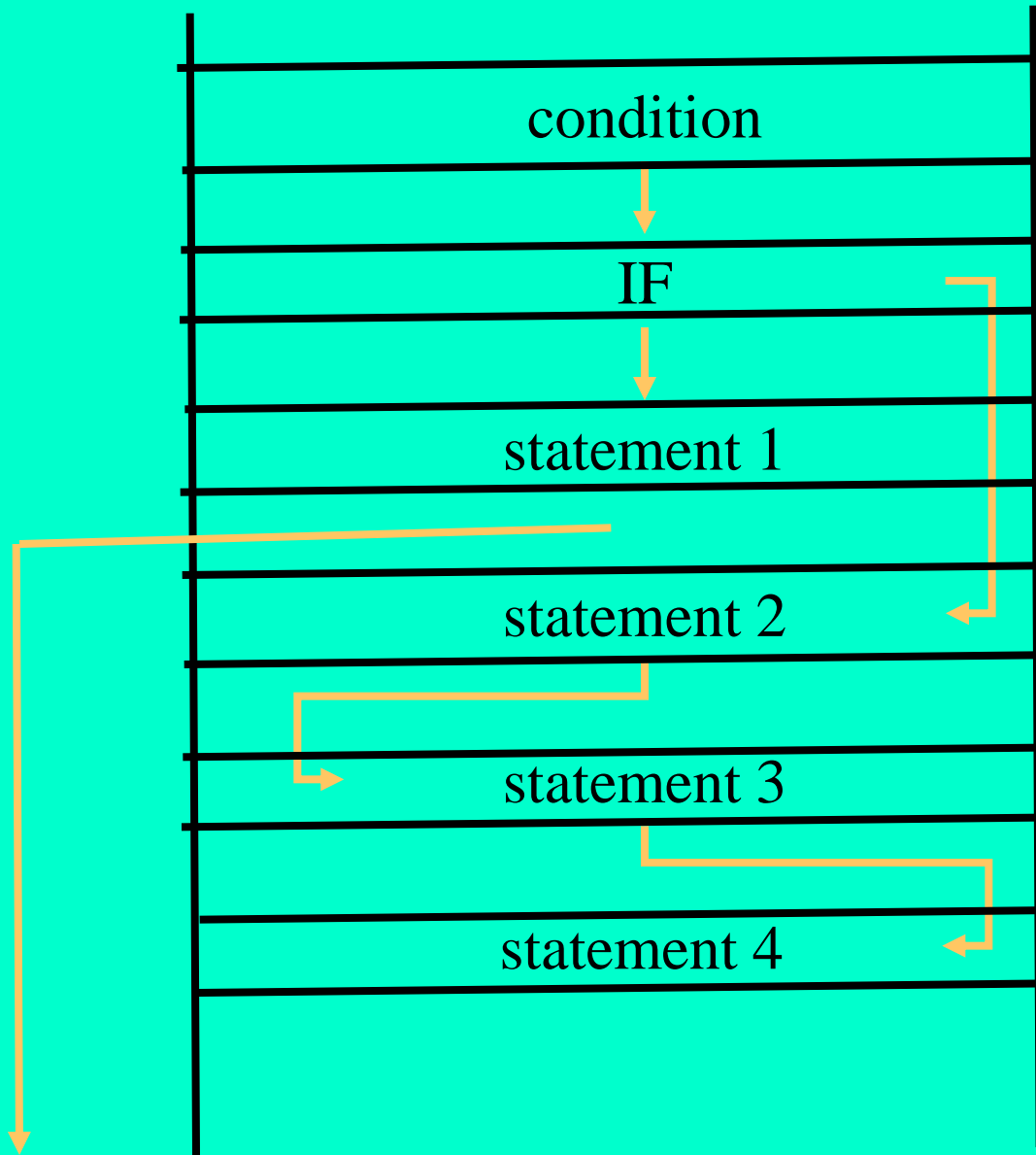
# Storing Threaded AST

- General Graph
- Array
- Pseudo Instructions

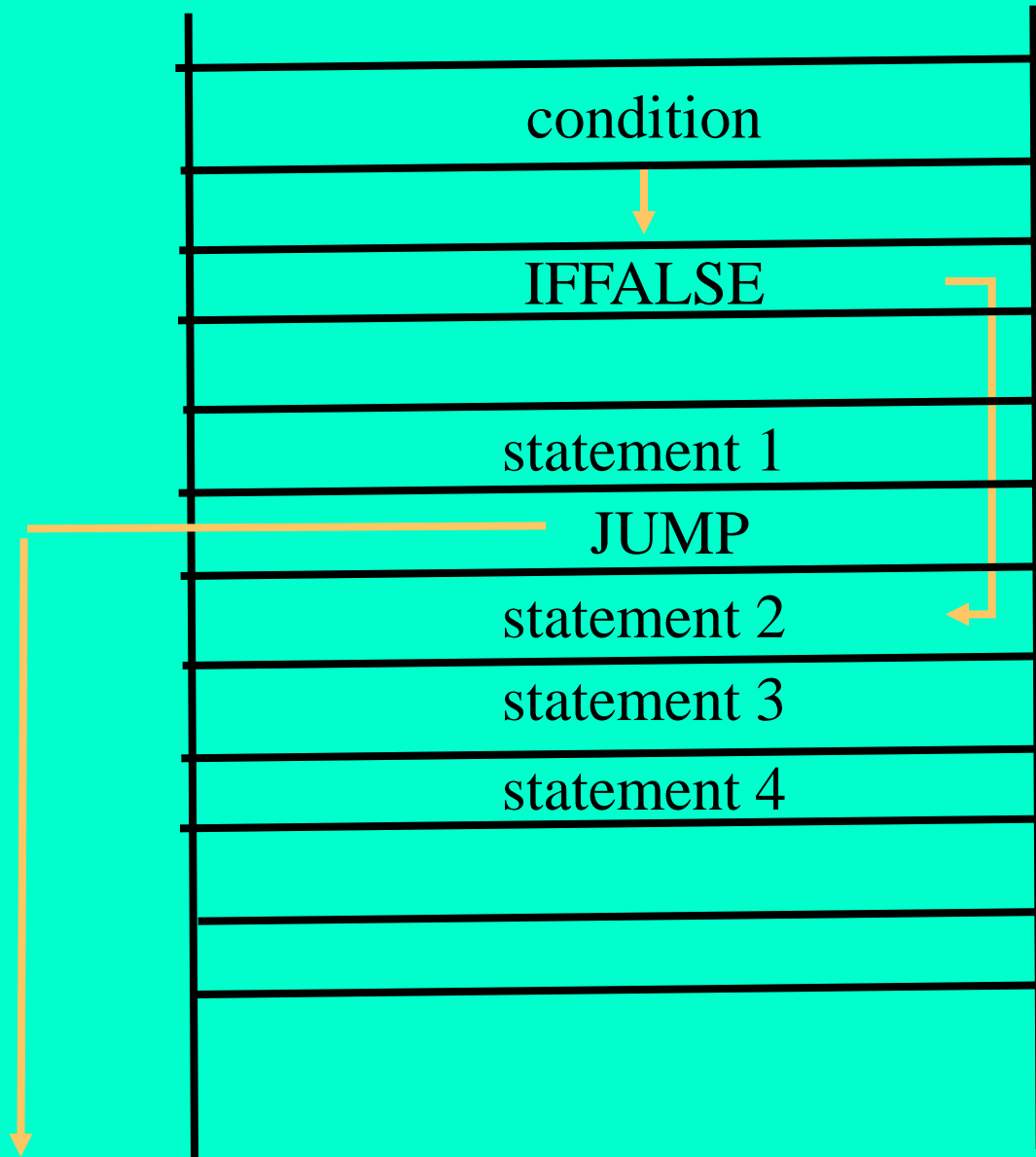
# Threaded AST as General Graph



# Threaded AST as Array



# Threaded AST as Pseudo Instructions



# Iterative Interpreters (Summary)

- Different AST representations
- Faster than recursive interpreters
  - Some interpretative overhead is eliminated
- Portable
- Secure
- Similarities with the compiler