Chapter 4
Structure of a simple compiler/interpreter

Lexical analysis
Syntax analysis
Context analysis
Intermediate code (AST)
Symbol Table
Runtime System Design
Code generation
Interpretation
Machine dependent

PL dependent
PL+paradigm dependent
Outline

- Interpreters
- Code Generation
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 → digit | ‘(‘ expression operator expression ‘)’
operator → ‘+’ | ‘*’
digit → ‘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 \((2 \ast ((3 \ast 4)+9))\)
Uniform self-identifying data representation

- The types of the 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

<table>
<thead>
<tr>
<th>re:</th>
<th>3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>im:</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Specific to the given value of type `complex_number`

- **value:** \[v\]
  - **type:**
  - **size:** 2
  - **values:** \[v\] \[v\]
  - **type:**
    - **value:** 3.0
  - **type:**
    - **value:** 4.0

Common to all values of type `complex_number`

- **name:**
- **class:** RECORD
- **field:**
  - **name:**
    - **type:**
    - **next:**
  - **name:**
    - **type:**
    - **next:**
  - **name:**
    - **type:**
    - **next:**

- **name:**
  - **class:** BASIC
  - **type number:** 3

- **name:**
  - **type:** str
    - **name:** "real"
  - **class:** STRING
  - **type number:** 4

- **name:**
  - **type:** list
    - **name:** "re"
  - **class:** LIST
  - **type number:** 5
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;
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
Partial Evaluation

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

```c
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
Example 2

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)
#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;
}
```c
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 */
    }
}

void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
    /* PUBLIC */
```
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 \((2 \times ((3 \times 4) + 9))\)
while ((x > 0) && (x < 10)) {
    x = x + y;
    y = y - 1;
}

C Example

while (true)
    seq
        while
            and
                id x
                const 0
                <
                    id x
                    const 10
                id y
                id x
                +
                    id y
                    const 1
            ass
                id x
                id y
                +
                    id y
                    const 1
        T
            F
exit
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
while

and

seq

const

0

<

>

id

x

const

id

y

x

10

+}

Last node pointer

main
while

and

seq

while

seq

Last node pointer

main

> id

const 0

id x

< id

const

id x

< id

const

+ id x

id y

id y

+ id y

const 1
while and seq

Last node pointer

main

> id x const 0

< id x const 10

> id x

< id x 10

id x

const 10

id x 10

id y

+ id x y

id y

+ id x y 1

id

const 1
While and seq ass id x id x id y const 0 < id x const 10 + id x id y + id const 1 1 Last node pointer main
Last node pointer

main

while

and

seq

const 0

id x

> id

< id

const 10

+ id y

id x

id y

+ id y

id x

const 1

id y

const 1

id x

id y

+ id const
while and id x id y const 0 < id x const 10 + id x id y + id y const 1 const 1

Last node pointer main
while and seq

id x

main

> x

const 0

< x

const 10

Last node pointer

while

const 10

id x

+ id y

id x

const 1

T

id x

y

id y

+ id const 1

id y

+ id const 1
Last node pointer

```
while
  and
    >
      id
      const
    <
      id
      const
  id
  const

seq
  ass
    id
    +
      id
      y
    id
  id
  +
    id
    y
    const
  y
```

main
Last node pointer

main

while

and

seq

const

0

<

id

x

const

10

>
Last node pointer

main

while

and

seq

T

const

0

const

10

+ x y

id

id

id

id

id

id

const

1

while

seq

const

+ x y

id

id

id

id

id

id

+ y

id

const

1

Last node pointer
while

and

seq

id

ass

id

const

id

id

const

id

id

const

id

id

const

id

id

y

+}

Last node pointer
while

and

<

id x

const 0

id x

const 10

while

seq

T

Last node pointer

id x

const 1

id x

id y

const 1

+ x y

id x

+ y 1

id y

id 1

id y

T

First node pointer

id x

Last node pointer
```c
#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;
    }
}

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

Last node pointer

if

cond

then_part

else_part
Conditional Statement

if

cond

T

F

then_part

else_part

End_If

Last node pointer
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)
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 ...
        CASE ...
Storing Threaded AST

- General Graph
- Array
- Pseudo Instructions
Threaded AST as General Graph

- condition
- IF
- statement 1
- statement 2
- statement 3
- statement 4
- END

If
Threaded AST as Array

condition

IF

statement 1

statement 2

statement 3

statement 4
Threaded AST as Pseudo Instructions

condition
IFFALSE
statement 1
JUMP
statement 2
statement 3
statement 4
Iterative Interpreters (Summary)

• Different AST representations
• Faster than recursive interpreters
  – Some interpretative overhead is eliminated
• Portable
• Secure
• Similarities with the compiler
Code Generation

• Transform the AST into machine code
• Machine instructions can be described by tree patterns
• Replace tree-nodes by machine instruction
  – Tree rewriting
  – Replace subtrees
• Applicable beyond compilers
\[
a := (b[4\ast c+d]\ast 2)+9
\]
Figure 4.10 Two sample instructions with their ASTs.

Load_Address M[R₁], C, R_d
Load_Byte (M+R₀) [R₁], C, R_d

leal          movsbl
Load_Byte (b+Rd)[Rc], 4, Rt
Load_byte (b+Rd)[Rc], 4, Rt

Load_address 9[Rt], 2, Ra

Ra
Code generation issues

• Code selection
• Register allocation
• Instruction ordering
Simplifications

- Consider small parts of AST at time
- Simplify target machine
- Use simplifying conventions
Overall Structure

Diagram:
- Intermediate code
  - IC pre-processing
  - Code generation proper
  - Target code post-processing
  - Machine code generation
  - Executable code output

Executable code

exm