Lecture 4: ASTs, Interpreters

David Hovemeyer

September 12, 2022

601.428/628 Compilers and Interpreters

Today

- ASTs, how to create them
- Building an interpreter on top of an AST
- Evaluating expressions
- Functions

ASTs

Abstract Syntax Tree (AST)

- An AST is a simplified form of a parse tree
  - Unnecessary information is omitted
  - Structure is simplified
- How do we create an AST? Options:
  - Transform the parse tree
  - Have the parser build AST directly
- Example code: `https://github.com/daveho/astdemo`

Expression grammar

Infix expression grammar with left recursion eliminated (n means “number”, i means “identifier”):

E $\rightarrow$ T E’
E’ $\rightarrow$ $+$ T E’
E’ $\rightarrow$ - T E’
E’ $\rightarrow$ ϵ
T $\rightarrow$ F T’
T’ $\rightarrow$ * F T’
T’ $\rightarrow$ $/$ F T’
T’ $\rightarrow$ ϵ
F $\rightarrow$ n
F $\rightarrow$ i
F $\rightarrow$ (E)

AST node types

AST node types should reflect the *operations* that the input program performs

For the expression grammar:

```
enum ASTKind {
  AST_ADD,
  AST_SUB,
  AST_MULTIPLY,
  AST_DIVIDE,
  AST_VARREF,
  AST_INT_LITERAL,
};
```

These reflect:

- How values are produced (variable references, literal values)
- How values are computed from existing values (add, sub, multiply, divide)

Option 1 (transform the parse tree)

Basic idea: write a function

```
struct Node *buildast(struct Node *t);
```

When passed a pointer to the root of some part of the parse tree, it returns a pointer to the root of an equivalent AST

Main issue for the expression grammar: we need left associativity for additive and multiplicative operators

- E.g., `a - b - c` means `(a - b) - c`
- The transformation from left recursion to right recursion makes the parse trees for left associative operators grow the wrong way

Parse tree to AST transformation

```
struct Node *buildast(struct Node *t) {
  int tag = node_get_tag(t);
  switch (tag) {

  ...cases for various kinds of parse nodes...

  default:
    err_fatal("Unknown parse node type %d\n", tag);
    return NULL;
  }
}
```

Easy cases

Identifiers and integer literals become AST_VARREF and AST_INT_LITERAL nodes:

```
case TOK_IDENTIFIER: // variable reference
  return node_alloc_str_copy(AST_VARREF, node_get_str(t));

case TOK_INTEGER_LITERAL: // integer literal
  {
    const char *lexeme = node_get_str(t);
    struct Node *ast = node_alloc_str_copy(AST_INT_LITERAL, lexeme);
    node_set_ival(ast, atol(lexeme));
    return ast;
  }
```

These are the base cases of the recursion

Primary expressions

Primary expressions are occurrences of the F nonterminal, productions:

$$F \rightarrow n$$
$$F \rightarrow i$$
$$F \rightarrow (E)$$

Recursively build AST from n (integer literal), i (identifier), or E (arbitrary expression) child:

```
case NODE_F: // parenthesized expression, identifier, or integer literal
  return buildast(node_get_kid(t, node_get_num_kids(t) == 3 ? 1 : 0));
```

Interesting cases

Occurrences of E and T nonterminals are expressions involving left associative (additive and multiplicative) operators. We need to fix the structure of the tree:

```
case NODE_E:
case NODE_T: // restructure for left associativity
  return buildast_left(buildast(node_get_kid(t, 0)), node_get_kid(t, 1));
```

Productions are:

$$E \rightarrow T\ E'$$
$$T \rightarrow F\ T'$$

Start by building an AST for T or F occurrence, then continue recursively if the expression continues at the same precedence level

Fixing associativity

Productions for continuations of additive and multiplicative expressions:

E' $\rightarrow +$ T E'
E' $\rightarrow$ - T E'
E' $\rightarrow \epsilon$
T' $\rightarrow$ * F T'
T' $\rightarrow /$ F T'
T' $\rightarrow \epsilon$

Epsilon production means the expression is finished

Otherwise, form will be an operator ($+$, -, *, or $/$), followed by an operand (T or F), followed by a recursive continuation

Fixing associativity

```
static struct Node *buildast_left(struct Node *ast, struct Node *right) {
  if (node_get_num_kids(right) == 0) {
    return ast; // done with expression
  }

  // first child of right parse tree is the operator
  struct Node *op = node_get_kid(right, 0);
  int op_tag = node_get_tag(op);

  // second child is an operand (T or F), convert it to AST
  struct Node *operand_ast = buildast(node_get_kid(right, 1));

  // join current expression AST with new operand
  int ast_tag = buildast_operator_tag(op_tag);
  ast = node_build2(ast_tag, ast, operand_ast);

  // continue recursively
  return buildast_left(ast, node_get_kid(right, 2));
}
```

Example parse tree

```
$ echo "a - b - c*3" | ./astdemo -p
E
+--T
|  +--F
|  |  +--IDENTIFIER[a]
|  +--T'
+--E'
   +--MINUS[-]
   +--T
   |  +--F
   |  |  +--IDENTIFIER[b]
   |  +--T'
   +--E'
      +--MINUS[-]
      +--T
      |  +--F
      |  |  +--IDENTIFIER[c]
      |  +--T'
      |     +--TIMES[*]
      |     +--F
      |     |  +--INTEGER_LITERAL[3]
      |     +--T'
      +--E'
```

Note how in expansion of E/E', subtrees grow to the right

Example AST

```
$ echo "a - b - c*3" | ./astdemo -b
SUB
+--SUB
|   +--VARREF[a]
|   +--VARREF[b]
+--MULTIPLY
    +--VARREF[c]
    +--INT_LITERAL[3]
```

In AST, the - (SUB) operator now associates to the left

Option 2 (have parser build AST)

We could avoid the need for a separate AST-building step by having the parser construct an AST directly:

- Omit unnecessary nodes
- Restructure tree as required

Primary expressions

```
struct Node *Parser2::parse_F() {
  struct Node *next_tok = lexer_peek(m_lexer);
  if (!next_tok) { error }

  int tag = node_get_tag(next_tok);
  if (tag == TOK_INTEGER_LITERAL || tag == TOK_IDENTIFIER) {
    struct Node *tok = expect(static_cast<enum TokenKind>(tag));
    const char *lexeme = node_get_str(tok);
    int ast_tag = tag == TOK_INTEGER_LITERAL ? AST_INT_LITERAL : AST_VARREF;
    struct Node *ast = node_alloc_str_copy(ast_tag, lexeme);
    if (ast_tag == AST_INT_LITERAL) { node_set_ival(ast, atol(lexeme)); }
    node_destroy(tok);
    return ast;
  } else if (tag == TOK_LPAREN) {
    expect_and_discard(TOK_LPAREN);
    struct Node *ast = parse_E();
    expect_and_discard(TOK_RPAREN);
    return ast;
  } else { error }
}
```

Handling of identifiers and integer literals is straightforward

Primary expressions

```
struct Node *Parser2::parse_F() {
  struct Node *next_tok = lexer_peek(m_lexer);
  if (!next_tok) { error }

  int tag = node_get_tag(next_tok);
  if (tag == TOK_INTEGER_LITERAL || tag == TOK_IDENTIFIER) {
    struct Node *tok = expect(static_cast<enum TokenKind>(tag));
    const char *lexeme = node_get_str(tok);
    int ast_tag = tag == TOK_INTEGER_LITERAL ? AST_INT_LITERAL : AST_VARREF;
    struct Node *ast = node_alloc_str_copy(ast_tag, lexeme);
    if (ast_tag == AST_INT_LITERAL) { node_set_ival(ast, atol(lexeme)); }
    node_destroy(tok);
    return ast;
  } else if (tag == TOK_LPAREN) {
    expect_and_discard(TOK_LPAREN);
    struct Node *ast = parse_E();
    expect_and_discard(TOK_RPAREN);
    return ast;
  } else { error }
}
```

Parentheses omitted from AST for parenthesized subexpression

Additive expressions

Production $\boxed{\text{E} \rightarrow \text{T E'}}$

Idea is to parse and build an AST for one term, then handle possible continuation recursively, building up a left-associative AST

- Multiplicative expressions ($\boxed{\text{T} \rightarrow \text{F T'}}$) are handled similarly

```
struct Node *Parser2::parse_E() {
  struct Node *ast = parse_T();
  return parse_EPrime(ast);
}
```

Additive expressions

As additive operators and terms are parsed, build left-leaning AST

```
struct Node *Parser2::parse_EPrime(struct Node *ast) {
  struct Node *next_tok = lexer_peek(m_lexer);
  if (next_tok) {
    int next_tok_tag = node_get_tag(next_tok);
    if (next_tok_tag == TOK_PLUS || next_tok_tag == TOK_MINUS)  {
      struct Node *op = expect(static_cast<enum TokenKind>(next_tok_tag));
      struct Node *term_ast = parse_T();
      ast = node_build2(next_tok_tag == TOK_PLUS
                          ? AST_ADD : AST_SUB, ast, term_ast);
      node_destroy(op);
      return parse_EPrime(ast); // continue recursively
    }
  }
  return ast; // end of additive expression
}
```

parse_TPrime is very similar

Example AST

```
$ echo "a - b - c*3" | ./astdemo -2
SUB
+--SUB
|   +--VARREF[a]
|   +--VARREF[b]
+--MULTIPLY
    +--VARREF[c]
    +--INT_LITERAL[3]
```

Which approach to use?

Build AST from parse tree:

- Full represented of source is maintained
- Arguably cleaner from a modularity standpoint
- Disadvantage: slower, uses more memory

Build AST directly in parser:

- Avoid keeping unnecessary information in memory
- Likely more efficient
- Disadvantage: parser is harder to understand

Additional thoughts on construction

Other parsing techniques make AST construction in the parser easier:

- Precedence climbing: essentially produces ASTs for infix expressions "natively"
- Bottom-up parsers that can handle left recursion: avoid the need for tree restructuring

So, building an AST directly from the parser is more straightforward in these cases

Mapping AST nodes to source code

Since the AST will be the starting point for interpretation and/or translation, we'll need to know how AST constructs correspond to source constructs

Basic idea: copy source information produced by lexical analyzer to AST

- Lexer should annotate tokens with this information

Building an interpreter

AST-based interpreters

An AST is an ideal data structure to use as the intermediate representation for an interpreter

- AST(s) represent the program
- *Evaluating* AST(s) executes the program

Values

We will need a data type to represent runtime values:

- values of integer literals
- values loaded from variables
- values stored in variables
- results of computations (e.g., operators in expressions)

Typical approach: tagged variant

- Each runtime value is tagged with its data type
- This approach works well for dynamically typed languages

Example value type

```
enum ValueKind {
  VAL_INT,
  VAL_FLOAT,
  VAL_STRING,
  // etc.
};

struct Value {
  enum ValueKind kind;
  long ival;     // used for VAL_INT
  double fval;   // used for VAL_FLOAT
  char *strval; // used for VAL_STRING
  // etc.
};
```

Tagged unions

Since only one value field at a time will be used, we can use a union to save memory:

```
struct Value {
  enum ValueKind kind;
  union {
    long ival;     // used for VAL_INT
    double fval;   // used for VAL_FLOAT
    char *strval; // used for VAL_STRING
    // etc.
  };
};
```

Storage for all value fields is collapsed

- This is safe as long as code checks `kind` field before accessing a value field

Are values accessed by value or by reference?

If a runtime value representation (e.g., `struct Value` type) stores only small, fixed-sized data values (fixed-precision integer or floating point, etc.), then it can be used *by value* within the interpreter

But, we may want to represent values requiring arbitrary storage to represent! (Strings, arrays, objects, etc.)

This means that runtime values may need to be (at least partially) accessed by reference/pointer

Key issue: how to ensure that memory is reclaimed when no longer used?

- More on this next time...

Evaluating expressions

The core of any programming language is expressions which compute values

Typical approach to representing expressions using ASTs:

- Parent nodes are *operations*
- Child nodes are *operands*
- Leaf nodes are primary expressions (literals, variable references)

Expression evaluation (pseudo code)

```
evaluate(astnode)
    if astnode is literal
        return literal value encoded by astnode
    else if astnode is variable reference
        return result of looking up value of variable
    else if astnode is variable assignment
        childval ← evaluate(astnode.children[0])
        update value of variable
        return childval
    else if astnode is unary operation
        childval ← evaluate(astnode.children[0])
        return result of applying operator to childval
    else if astnode is binary operation
        leftval ← evaluate(astnode.children[0])
        rightval ← evaluate(astnode.children[1])
        return result of applying operator to leftval and rightval
```

Functions

Functions

Functions (a.k.a. procedures, subprograms) are the most fundamental abstraction mechanism in computing

How to support them?

- Syntax
- Semantics

Syntax: function definition

Main issues in function syntax:

- Function name
- Parameters
- Function body

Example grammar production (*italic* means nonterminal, **bold** means terminal):

funcdef → **function identifier** (*opt-parameter-list*) { *statement-list* }

Using a keyword (e.g., **function**) to designate a function definition makes the parser's job easier

Syntax: function call

A function call can be considered as a primary expression

- Along with other kinds of primary expressions, such as literals, variable references

Example grammar production (*italic* means nonterminal, **bold** means terminal):

primary $\rightarrow$ **identifier** (*opt-expression-list*)

In general, this can be parsed easily by both top-down and bottom-up parsers

- If an identifier is immediately followed by a left parenthesis, it's a function call, not a variable reference

Function semantics

Evaluating a function call

```
function add(x, y) {
  x + y;
}

a = add(2+1, 3*4);
```

Value assigned to a should be $\boxed{15}$

- Why?

Evaluating a function call

Steps:

1. Evaluate arguments
2. Create a new *environment* for the function parameters
3. Assign computed argument values to the function parameters in the new environment
4. Evaluate the function body in the new environment
5. Result of evaluating function body becomes the value computed by the function call expression

Function call evaluation example

Evaluating a function call

```
function add(x, y) {
  x + y;
}

a = add(2+1, 3*4);
```

Function call evaluation example

Evaluating a function call

```
function add(x, y) {
  x + y;
}

a = add(2+1, 3*4);
```

```
evaluate arguments: 3, 12
```

Function call evaluation example

Evaluating a function call

```
function add(x, y) {      x=3, y=12
  x + y;
}

a = add(2+1, 3*4);
```

Function call evaluation example

Evaluating a function call

```
function add(x, y) {
  x + y;
}

a = add(2+1, 3*4);
```

`evaluates to 15`

Function call evaluation example

Evaluating a function call

```
function add(x, y) {
  x + y;
}

a = add(2+1, 3*4);
```

`call evaluates to 15`

Function call evaluation example

Evaluating a function call

```
function add(x, y) {
  x + y;
}

a = add(2+1, 3*4);
```

`assign 15 to a`

Making this work

Next time:

- Representing environments
- Variables and scopes
- Representing functions
- Runtime data structures, garbage collection