Building Abstract Syntax Trees

The files associated with this lecture are

- flex input file exp.lex
- bison input file exp.yacc
- class declarations exp.h
- class definitions exp.cpp
- makefile

Classes that Need to be Declared

In order to construct a parse tree you will need to declare a class for each non-terminal and each production in your grammar. The class for each non-terminal should be declared as abstract and each of the non-terminal's productions should be declared as a subclass. If a terminal carries information, such as a number or id, then that terminal should also have its own top-level class. If a terminal has only one possible value or is a punctuation character, there is no need to store it since you will know its value based on its production. For example, if you have the production `E -> E - E`, there is no need to store the minus sign since you will know that the production represents a minus expression.

In each production's subclass you will need to have pointers to nodes that represent non-terminals on the right hand side of the production. If there are terminals that carry information, such as numbers or ids, then you will need to pointers to those nodes as well. The reason that productions should be subclasses of their left hand side nonterminal is that they expand that nonterminal and therefore represent one of the potential subtrees rooted at that nonterminal.

As an example of how you might construct a parse tree, consider the following expression grammar:

```
    pgm -> stmt*
    stmt -> id = exp
         |  print id
    exp -> exp + exp | exp - exp | exp * exp | exp / exp
         | ( exp ) | - exp | id | number
```

The nonterminals are *pgm*, *stmt* and *exp* so we need abstract classes for these three nonterminals:

```
class pgm {}
```

```
class statement {}
class exp_node {}
```

We will start by defining the subclasses for the productions associated with an expression. The easiest subclasses are those that represent the terminals **number** and **id**. The subclass for **number** needs to store the value of the number:

```
class number_node : public exp {
  protected:
    int num;

  public:
    number_node::number_node(float value) {
      num = value;
    }
};
```

Likewise, the subclass for **id** needs to store the string value of the id:

```
class id_node : public exp {
  protected:
    string id;

  public:
    id_node(string value) : id(value) {}
};
```

Next we define subclasses for the arithmetic productions (+, -, *, /). I am only showing the subclasses for Plus and Times:

```
class plus_node : public node {
  protected:
    node *left;
    node *right;

  public:
    plus_node(node *L, node *R): left(L), right(R) {}
};

class times_node : public node {
  protected:
    node *left;
    node *right;

  public:
    times_node(node *L, node *R): left(L), right(R) {}
};
```

The plus and times nodes look pretty similar, and in fact they are so similar that we can factor some of their common node into a shared superclass, called `operator_node`:

```
class operator_node : public node {
  protected:
    node *left;
    node *right;
  public:
    operator_node(node *L, node *R): left(L), right(R) {}
};

class plus_node : public operator_node {
  public:
    plus_node(node *L, node *R): operator_node(L, R) {}
};
class times_node : public operator_node {
  public:
    times_node(node *L, node *R): operator_node(L, R) {}
};
```

For now we will continue to create classes for our remaining productions. The next class is for the unary minus production:

```
class unary_minus_node : public node {
  protected:
    node *exp;

  public:
    unary_minus_node(node *expToNegate): exp(expToNegate) {}
};
```

Finally we come to the parenthesized expression production. This production is one that is useful for deriving the syntactic meaning of the program, but is useless for deriving the semantic meaning. Hence we will not explicitly represent this production in the syntax tree, but instead pass its expression up to the next node in the chain. We will see how to do that below. For now the important thing is that we do not need to create a class for the parenthesized expression production.

Adding Attributes and Attribute Evaluation Rules

After we have defined the classes and subclasses for our grammar, we need to consider the set of attributes that we will need. For our simple expression grammar we need only one attribute, which is the value of the expression. Remember that attributes are declared in the class associated with the non-terminal, and that the subclasses define methods for evaluating these attributes. Hence the *value* attribute is declared in the **exp** class:

```
class exp {
  protected:
    float num;
```

```
};
```

Next we define a method for evaluating the *value* attribute. I will call it `evaluate`. Each production will have its own rule for evaluating the *value* attribute, so we need to declare the `evaluate` method to be a pure virtual method:

```
class exp {
  protected:
    float num;
  public:
    virtual evaluate() = 0;
};
```

Each expression subclass will now define an appropriate evaluate method. Here is a representative sample:

```
// a number just returns its value
float number_node::evaluate() {
  return num;
}

// an id looks itself up in the idTable
float id_node::evaluate() {
  return idTable[id];
}

// a plus operator evaluates its left and right operands and
// then adds them together
float plus_node::evaluate() {
  float left_num, right_num;

  left_num  = left->evaluate();
  right_num = right->evaluate();

  num = left_num + right_num;
  return num;
}
```

Building Abstract Syntax Trees in Bison

Now we are ready to use Bison to build an abstract syntax tree for strings that can be generated using this expression grammar. There are two steps:

1. Declare the types of the nodes in the abstract syntax tree. Since nodes in the abstract syntax tree represent the non-terminals on the left hand sides of productions, we will be declaring the nodes to be pointers to classes. This is done via the `%union` and `%type` directives. We add a field to the `%union` that declares a pointer to an Exp node, and then use the `%type` directive to make the `exp` non-terminal be a pointer to an Exp node:

```
%union {
  float num;
  char *id;
  exp_node *expnode;
}

%type <expnode> exp
```

2. For each production in our grammar that has a subclass defined for it, write an action rule to create an instance of that class. Typically the action rule will simply pass pointers to the production's children to the constructor. For example, to construct a plus_node, we pass it the pointers to its two operands:

```
exp:    exp PLUS exp {
          $$ = new plus_node($1, $3); }
```

Flattening Lists

We frequently want to represent productions that represent lists as a single root node with a list of children, as opposed to a spindly "vine" of nodes. For example, our production for a program is ideally written as:

```
pgm -> stmt*
```

and is ideally represented in an abstract syntax tree as:

```
     ---- pgm-----
    /    /        \
stmt1  stmt2 ...  stmt_n
```

However, to get Bison to recognize the grammar, we must rewrite it as:

```
pgm -> stmtlist
stmtlist -> stmtlist stmt
         |  ε
```

which creates a viney looking tree:

```
                 pgm
                  |
              stmtlist
              /      \
          stmtlist stmt_n
             ...
          /      \
      stmtlist  stmt2
```

```
      /     \
   stmtlist stmt1
      |
      ε
```

We can flatten a tree by starting a list when we encounter the &epsilon production, and simply appending children to this list thereafter. For example:

```
%union {
  ...
  list *stmts;
  pgm *prog;
}

%type <stmts> stmtlist
%type <prog> program

// $1 is a list of stmts
program : stmtlist { $$ = new pgm($1); }
;

stmtlist : stmtlist stmt NEWLINE
            { // copy up the list and add the stmt to it
              $$ = $1;
              $1->push_back($2);
            }
         |  { $$ = new list(); }
```

Returning the Abstract Syntax Tree from the Parser

You will need to store a pointer to the root of your abstract syntax tree in a global variable, since `yyparse` always returns an `int` (0 for success, 1 for failure). Here's how I did it in the expression grammar:

```
%{
// definitions section
// the root of the abstract syntax tree
pgm *root;
%}

%%

  /* rules section */
program : stmtlist { $$ = new pgm($1); root = $$; }
;
```