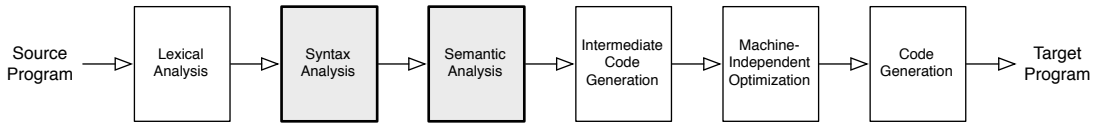
HW 4: Abstract Syntax Trees and Symbol Tables

CSCI 434T
Fall, 2011

Overview



This week's goal is to explore the internal data structures used by a compiler to organize and manipulate a source program. These data structures are the core of a compiler, and developing a good set of programming abstractions is essential for managing implementation complexity.

There are several specific topics for the week:

1. Building abstract syntax trees (ASTs) during parsing, and examining other intermediate representations.
2. Implementing the Visitor Design Pattern to make AST manipulations simpler.
3. Developing a general structure for symbols information.

We will also breifly look at language features called multimethods and open classes, which enable implementation strategies that exhibit a number of interesting benefits and drawbacks over standard AST and Visitor patterns.

Readings

The readings are from a variety of sources. I have grouped them according to topic, roughly in the order in which they will be most useful. Some parts are pretty light and fill in background material; others are more directly related to the problems below.

- LR Parser Generators and Attribute Grammars.
  - *Java CUP manual*. *(online)*
  - *Engineering a Compiler*, Cooper and Torczon, Ch. 4.1–4.3. *(mostly background material – skim for the basic ideas. They will be recurring themes for us.)*
- Intermediate Representations, Abstract Syntax Trees, Visitor Design Pattern.
  - *Engineering a Compiler*, Cooper and Torczon, Ch. 5.1–5.4.
  - *Modern Compiler Implementation in Java*, Appel, Ch. 4.
  - *Design Patterns*, Gamma et al., 331–344. *(optional, book in lab)*
- Scoping and Symbol Tables.
  - *Engineering a Compiler*, Cooper and Torczon, Ch. 5.7.
- Multi-Methods.
  - "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java", Clifton *et al.*, *Conference on Object-Oriented Programming: Systems, Languages, and Architectures*, 2000. Sections 1 – 3. *(online)*
  - "Practical Predicate Dispatch," Todd Millstein, *Conference on Object-Oriented Programming: Systems, Languages, and Architectures*, 2004. *(optional, online)*

Exercises

1. This question explores how CUP and other LR parser generators enable one to embed semantic actions in a grammar definition. Each production in a grammar can be associated with a semantic action:

```
A  ::=  body_1  {: semantic-action_1 :}
    |   body_2  {: semantic-action_2 :}
        ...
    |   body_k  {: semantic-action_k :}
    ;
```

   The semantic action i, which is just Java code, is executed whenever the parser reduces the body of production i to the non-terminal A. The parser also associates an attribute value with each terminal and non-terminal on the parsing stack. The name `RESULT` refers to the attribute for the head (ie, A), and we can give names to the attributes for the symbols in the body of the production, as seen below with the names `e` and `val`:

```
terminal Integer NUM;
terminal PLUS;

nonterminal Integer E;

precedence left PLUS;

E ::=  E:e1 PLUS E:e2 {: RESULT = e1 + e2; :}
   |   NUM:val {: RESULT = val; :}
   ;
```

   In essence, the parser stack contains ⟨ Symbol, Attribute ⟩ tuples. It uses the symbols to parse and mantains the attributes for you to use.

   For each terminal and non-terminal, we declare the attribute type, if any. The scanner must create the attribute values for terminals, as we did in PA 1. The semantic actions in the parser synthesize the attribute values for non-terminals during parsing.

   In the last ten years, there has been a major shift away from using semantic actions to perform any sort of type checking or code generation inside a compiler. Instead, we simply use the semantic actions to build an abstract syntax tree, and we use subsequent tree operations to perform analysis. Thus, we could build an AST for the above example as follows:

```
terminal Integer NUM;
terminal PLUS;

nonterminal Expr E;

precedence left PLUS;

E ::=  E:e1 PLUS E:e2 {: RESULT = new Add(e1, e2); :}
   |   NUM:val {: RESULT = new Number(val); :}
   ;
```

   where we have the following AST node definitions:

```
abstract class Expr {}

class Add extends Expr {
  Expr left, right;
  Add(Expr left, Expr right) { this.left = left; this.right = right; }
}

class Number extends Expr {
  int val;
  Number(int val) { this.val = val; }
}
```

(a) Download and import the project for this problem into Eclipse. In a terminal window, cd into the project directory and run "make dump" to see the JavaCUP description of the state machine — the details are not important now, but it will be useful to do this later if you ever have conflicts or JavaCup errors.

(b) Extend the example CUP grammar above with the following:

```
terminal OPAREN, CPAREN, COMMA; /* '(', ')', and ',' */

nonterminal Vector<Expr>  EList;
nonterminal Vector<Expr>  ES;

EList ::= OPAREN ES CPAREN;
ES    ::= ES COMMA E | E;
```

Add semantic actions to these new non-terminals so that the parser constructs a vector of `Expr` objects when parsing input of the form "`(4,1+7)`."

(c) Describe the sequence of actions performed by the parser when parsing "`(4,1+7)`." Be sure to describe the attributes for each symbol on the parsing stack each time a production for `ES` is reduced, and draw the final attribute created for the `EList` You need not build the parsing table, etc. Simply describe the actions at a high level (ie, "shift `NUM` onto stack, with attribute value . . ."; "reduce . . . to . . ., popping off attribute values . . . and pushing attribute value . . ."; and so on).

(d) The grammar above uses left recursion in the `ES` non-terminal. Lists like this could also be written with right recursion, as in:

```
ES ::= E COMMA ES | E ;
```

Add semantic actions to these productions to produce the same result as above.

(e) It is often considered bad form to use right recursion in CUP grammars, if it can be avoided. Why do you think left recursion is preferable to right recursion? (Hint: think about how "(1,2,3,4,5,. . . ,1000000)" would be parsed.)

2. [Adapted from Cooper and Torczon]

- Show how the code fragment

```
if (c[i] != 0) {
  a[i] = b[i] / c[i];
} else {
  a[i] = b[i];
}
println(a[i]);
```

might be represented in an abstract syntax tree, in a control flow graph, and in quadruples (or three-address code — the web page has a brief overview of TAC).

- Discuss the advantages of each representation.
- For what applications would one representation be preferable to the others?

3. This question explores the basic idea behind the Visitor pattern, as described in Appel, Chapter 4. (The *Design Patterns* book also discusses this pattern in detail, although Appel hits the most important points.) Consider Programs 4.7 and 4.8 in Appel.

   (a) Suppose we create the following expression `e`:

   ```
   Expr i3 = new IntegerLiteral("3");
   Expr i4 = new IntegerLiteral("4");
   Expr i6 = new IntegerLiteral("6");
   Expr i8 = new IntegerLiteral("8");
   Expr e1 = new MinusExp(i3, i4);
   Expr e2 = new TimesExp(e1, i6);
   Expr e = new PlusExp(i8, e2);
   ```

   If we create an `Interpreter` object `interp` and invoke

   ```
   e.accept(interp);
   ```

   what sequence of `accept` and `visit` calls will occur?

   (b) What are the primary benefits of using a visitor pattern over a pattern like that found in Program 4.5? What are the disadvantages?

4. The following grammar describes the language of regular expressions, with several unusual characteristics described below:

$$
\begin{array}{rcl}
R & \rightarrow & R \,'|'\, R \\
 & | & R \,'.'\, R \\
 & | & R \,'*' \\
 & | & R \,'?' \\
 & | & R \,'+' \\
 & | & '(' \, R \, ')' \\
 & | & \textit{letter} \\
 & | & '[' \, L \, ']' \\
 & | & \texttt{'let'} \; \textit{id} \; '=' \; R \; \texttt{'in'} \; R \\
 & | & \textit{id} \\
 & | & \epsilon \\
 & & \\
L & \rightarrow & L \; \textit{letter} \\
 & | & \textit{letter}
\end{array}
$$

   The `*`, `?`, and `+` operators have higher precedence than concatenation (`'.'`); and, in turn, concatenation has higher precedence than alternation. A *letter* can be any lower-case letter in `'a'`–`'z'`. The term `'[' L ']'` indicates an alternation between all of the letters in the letter list L. Here are some examples:

   - `a.b+`: `a` followed by one or more `b`'s.
   - `[abc]`: any of `a`, `b`, or `c`
   - `a.(b|c)*`: `a` followed by any number of `b`'s and `c`'s.
   - `a|@`: either `a` or ϵ, which is represented by `@`.

In order to describe more interesting patterns succinctly, our language has "let"-bindings for *id*'s (which are identifiers starting with capital letters), as in the following:

```
let C = (c.o.w)* in
  C.m.C.m.C
```

which is the same as `(c.o.w)*.m.(c.o.w)*.m.(c.o.w)*`. Bindings can be nested, and one binding can redefine an already bound name:

```
let C = c.o.w in
  C.C.
  let C = m.o.o in
    C*
```

which is equivalent to `c.o.w.c.o.w.(m.o.o)*`.

The starter code for this problem includes a complete Flex specification and a skeletal CUP specification to scan and parse regular expressions, respectively. You are to design an AST package for regular expressions and then write code to translate regular expressions into NFA descriptions that can be executed on our NFA Simulator.

(There is a Scala starter project you may use as well, but please implement the Visitor pattern as described instead of `case` classes. The Visitor pattern is a design pattern you will see in many places and is worth understanding.)

(a) The `main` method in `re.Main` currently reads a regular expression from its first argument. It is executed from the command line as follows:

```
java -classpath .:tools/java-cup-11a.jar re.Main ex1.re
```

Before you can successfully parse expressions, however, you must complete the parser. Please use the CUP precedence rules to do eliminate ambiguity — do not rewrite the grammar.

(b) Design a hierarchy of classes to represent regular expression ASTs. The root of your hierarchy should be the `re.ast.RENode` class that I have provided. You hierarchy should contain a reasonable, *minimal* collection of classes. Not every concrete syntactic form needs to have an analog in the abstract syntax (eg, "`[L]`" can be expressed as an alternation, etc.).

You AST hierarchy should support the visitor design pattern. More specifically, each node class should define the following method from the `RENode` abstract class:

```
public void accept(Visitor visitor)
```

In addition, you should implement the appropriate `Visitor` interface that has a `visit` method for each different type of node.

On some occasions, we will need to use a variation of the visitor that enables us to pass information into and return information from the `visit` methods. To support this, you should also define a `PropogatingVisitor` interface in which the `visit` methods take and return extra values:

```
/**
 * An interface for a propagating AST visitor.
 * The visitor passes down objects of type DownType
 * and propagates up objects of type UpType.
 */
interface PropogatingVisitor<DownType,UpType> {
  UpType visit(REEmpty re, DownType context);
  UpType visit(..., DownType context);
  ...
}
```

The type parameters `DownType` and `UpType` describe the type of data passed into and out of each `visit` method. To support propagating visitors, each `RENode` class will also need a second `accept` method, which propagates the extra information through the traversal:

```
public <DownType, UpType> UpType accept(
        PropagatingVisitor<DownType, UpType> visitor, DownType context)
```

(c) Extend the parser to generate an AST for the parsed regular expression.

(d) Complete the `re.PrettyPrint` visitor to print expressions represented by a `RENode`, and extend the `main` method to print the parsed expression.

(e) Complete the `re.NFABuilder` propogating visitor to build a `NFA` for a regular expression represented by a `RENode`. I have provided the `re.NFA` class to help in this step — your visitor simply needs to create a new `NFA` object and invoke the appropriate methods on it to create states and edges. See the online javadoc for details on the `NFA` class.

Think carefully about what information would be most useful to propagate down the tree and back up during the traversal.

There are a number of choices in how to manage and lookup names during NFA construction. You may use the analog of either static or dynamic scoping to lookup names during translation. Dynamic is simpler. Thus,

```
let A = a in
  let B = A in
    (let A = c in
      B
    ).B
```

would yield `c.a`.

Regardless of your resolution rules, you may assume that no circularities in name definitions will exist to avoid generating infinitely large NFAs.

At very the least, your `NFABuilder` will need to mantain an environment to map *id*'s to their definitions and should generate a `re.error.REError` exception if you encounter an *id* that has not been defined.

(f) Use the `NFA.toString()` to write the resulting NFA to a file, which can then be run with the `nfasim` program that I have provided on the Unix machines. This alphabet for this simulator is `a`–`z`, plus `@` for ϵ.

If `ex1.re` contains the expression "`a.(b|c)*`", your program should generate an NFA similar to the following:

```
7
6
0 a:(1) ;
1 @:(2,3) ;
2 b:(4) ;
3 c:(5) ;
4 @:(6) ;
5 @:(6) ;
6 @:(1) ;
```

Running

```
nfasim ex1.nfa ab cow abccccc a
```

will then result in

```
ab: yes
cow: no
abccccc: yes
a: yes
```

Be sure to test your program on more sophisticated examples.

To help you debug the last two steps, the `NFA` class also contains a `printDot()` method that generates the file "nfa.dot". This is graphical representation of the NFA can be viewed by issuing the following command from the command line to generate a PDF file:

```
dot -Tpdf < nfa.dot > nfa.pdf
```

Please turn in printouts of the following:

- A description of your Abstract Syntax Tree package and the `PrettyPrint` and `NFABuilder` visitors that you wrote.
- All of your code.

5. [Adapted from Cooper and Torczon] You are writing a compiler for a lexically-scoped programming language. Consider the following source program:

```
1    procedure main
2       integer a,b,c;
3       procedure f1(integer w, integer x)
4         integer a;
5         call f2(w,x);
6       end;
7       procedure f2(integer y, integer z)
8         integer a;
9         procedure f3(integer m, integer n)
10          integer b;
11          c = a * b * m * n;
12        end;
13        call f3(c,z);
14      end;
15      ...
16   call f1(a,b);
17   end;
```

As in ML, Pascal, or Scheme (which I'm sure you all remember oh-so-well from 334...), the scope of a nested procedure declaration includes all declarations from the enclosing declarations.

(a) Draw the symbol table and its contents at line 11.

(b) What actions are required for symbol table management when the semantic analyzer enters a new procedure and when it exits a procedure?

(c) The compiler must store information in the IR version of the program that allows it to easily recover the relevant details about each name. In general, what are some of the relevant details for the variable and procedure names that you will need to perform semantic analysis, optimization, and code generation? What issues must you consider when designing the data structures to store that information in the compiler?

(d) This part explores how to extend your symbol table scheme to handle the `with` statement from Pascal. From the Pascal documentation:

> The `with` statement serves to access the elements of a record or object or class, without having to specify the name of the each time. The syntax for a `with` statement is:

```
with variable-reference do
  statement
```

The variable reference must be a variable of a record, object or class type. In the `with` statement, any variable reference, or method reference is checked to see if it is a field or method of the record or object or class. If so, then that field is accessed, or that method is called. Given the declaration:

```
Type Passenger = Record
   Name : String[30];
   Flight : String[10];
end;

Var TheCustomer : Passenger;
```

The following statements are completely equivalent:

```
TheCustomer.Name := 'Michael';
TheCustomer.Flight := 'PS901';
```

and

```
With TheCustomer do
  begin
    Name := 'Michael';
    Flight := 'PS901';
  end;
```

In essence, the `with` statement is a shorthand to access a bunch of fields from a compound structure without fully qualifying each name. Discuss in a few sentences how you would augment the symbol table scheme you followed in parts (a) and (b) to support `with`. In particular, what information would you store about each `record` type definition, and how would you modify the symbol table when the semantic analyzer enters and exits a `with` statement? What information do you attach to any symbol added to the table during these operations?

6. The visitor pattern is just one way to implement AST-like data structures. A number of language features not present in Java can impact how easy it is to write tree structures and operations on them. For example, ML datatypes and pattern matching offer a different idiom with its own set of tradeoffs, and there are others as well.

   Please read the first few sections of the MultiJava paper (Clifton *et al.*, 2000) and consider the following. You need not write more than a few sentences or sketch a few lines of code for each part, but please think about the ideas a little.

   - How could Open Classes be used in place of the Visitor pattern?
   - How could Multimethods be used in place of the Visitor pattern?
   - What are the advantages / disadvantages of these two approaches over the Visitor Pattern?

   You may wish to illustrate the two alternative approaches by implementing a small example, such as the one discussed in question 3. I have installed MultiJava on the Unix machines for you to do this. It can be run with the `mjc` command. I will also put some sample code on the web page.

   You may also wish to read the "Practical Predicate Dispatch" paper, which takes dispatching to the extreme. The first three or four sections are quite interesting from a language design point of view.