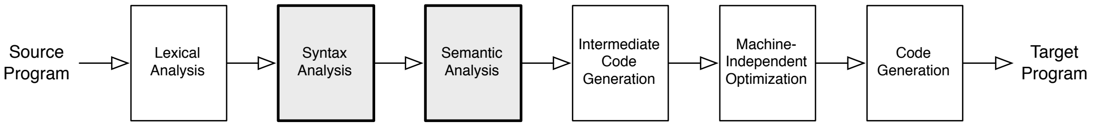
HW 4: Abstract Syntax Trees and Symbol Tables

CSCI 434T
Spring, 2019

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. Developing a general structure for symbols information.

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.
  - *Engineering a Compiler*, Cooper and Torczon, Ch. 5.1–5.4.
  - *Modern Compiler Implementation in Java*, Appel, Ch. 4. *Skim. If you have seen the Visitor design pattern, you can compare what Appel describes about AST visitors to how you envision case classes supporting the type of operations we're likely to perform on trees.*
- Scoping and Symbol Tables.
  - *Engineering a Compiler*, Cooper and Torczon, Ch. 5.7.

Exercises

1. **Will work on this in lab on Tuesday.** This question requires some programming. You may work with a partner if you wish. The problem 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  ::=  $body_1$  {: $semantic\text{-}action_1$ :}
     |  $body_2$  {: $semantic\text{-}action_2$ :}
        ...
     |  $body_k$  {: $semantic\text{-}action_k$ :}
     ;
```

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 $\langle$ Symbol, Attribute $\rangle$ 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 15 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) Clone your `ExpressionParser` from GitLab. If you'd like to work with a partner (or two), choose one repository to work on and share the repository with the group. Instructions for sharing a project are in the GitLab tutorial on the web page.

Open the project in IDEA and be sure it compiles. You may need to configure the JDK as you did last week. Use the same external tool to run Flex and JavaCUP to generate the parser and scanner.

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.

To experiment with this, comment out the `precedence left PLUS` line in `hw4.cup`. Run "make dump" again – examine the reported conflict in the state machine it prints. Add that line back in to remove the conflict.

(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)`." (Note: CUP treats the non-terminal for the first production in the file to be the start symbol, so the line "`EList ::= OPAREN ES CPAREN`" must appear before the definitions of `ES` or `E`.)

(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.)

No need to bring this code to our meetings, but be prepared to talk about the above items.

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 also involves programming. It is more substantial than most of the homework questions, so please don't wait until the last minute. You may work with a partner if you like. The following grammar describes the language of regular expressions, with several unusual characteristics described below:

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

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. You'll use Scala this time.

To begin, clone your "RECompiler" from GitLab. It is set up like the project for the first question and PA 2. You are free to again work with a partner or two on this question.

(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:

```
scala -classpath "bin:tools/java-cup-11a.jar:tools/*" re.Main ex1.re
```

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

(b) Design a hierarchy of case classes to represent regular expression ASTs. The root of your hierarchy should be the `re.ast.RENode` class that I have provided. Your 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.).

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

Your AST classes for this problem, and for the project, will be written in Scala. Those classes should use Scala collections and other classes to store their data. That is, you should not use classes like `Java.util.Vector` in your parsing code. Instead, you should use Scala `Lists` and other Scala classes. You can create objects of those Scala classes in Java code, but it can be a bit ugly when it comes to lists and options, so I have provided a few helper methods in `ParseUtil.scala`. Inside your parser, you can use the following three methods to create an empty list, insert on the front of a list, and append to the end of a list:

```
ParseUtil.empty()
ParseUtil.cons(elem, list)
ParseUtil.append(list, elem)
```

I also provide two other methods for creating values of `Option` types. Options are useful for storing "optional" values. I encourage you to use them where appropriate. More details can be found in our Scala reference books or online at various sources. This website has a couple of examples: `http://twitter.github.io/effectivescala/`.

(d) Complete the `re.PrettyPrint` class for generating printable forms of expressions represented by a `RENode`, and extend the `main` method to print the parsed expression.

(e) Write a `re.NFABuilder` class 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 builder simply needs to create a new `NFA` object and invoke the appropriate methods on it to create states and edges. See the classes documentation for details on the `NFA` class. This process will be a recursive traversal of the AST. 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 a map from *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 NFA simulator, which can be run as:

```
java -jar tools/nfasim.jar  ex1.nfa
```

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

Running

```
  java -jar tools/nfasim.jar 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 come to your meetings with your final code committed to GitLab. Time permitting, I'd like to do mini-code reviews of your solutions.

4. [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, 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?