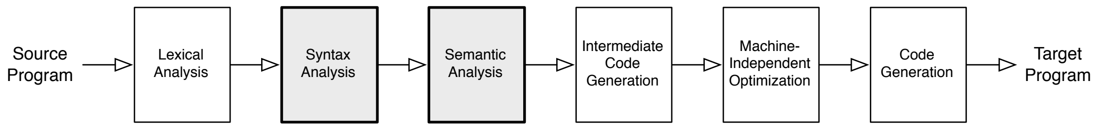
CS 301 Spring 2016
Meetings February 22

Abstract Syntax Trees and Symbol Tables

Plan



This week's goal is to explore 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. We focus on building abstract syntax trees (ASTs) during parsing, developing a general structure for symbols information, and briefly exploring other intermediate representations.

Readings

Copies of *Engineering a Compiler* (EC) are available in the white bookshelf outside my door. 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*, mainly sections 1-2, 6. `http://www2.cs.tum.edu/projects/cup/docs.php`
  - Skim *Java CUP examples*: Calculator and MiniJava with manual user actions. Don't worry about details of the AST representation. `http://www2.cs.tum.edu/projects/cup/examples.php`
  - EC 4.1 – 4.3. *Skim for basic background ideas (recurring themes for us).*
- Abstract Syntax Trees, Intermediate Representations.
  - EC 5.1 – 5.3.
- Scoping and Symbol Tables.
  - EC 5.5
- Scala Case Classes and Pattern Matching.
  - See Scala resources on the tools page: `https://cs.wellesley.edu/~cs301/tools.html`

Exercises

1. This question requires some programming. You may work with a partner if you wish. We will convert the Tiny compiler to use a generated parser using Java CUP instead of the handwritten parser we developed in the first week of classes.

   The problem explores how CUP and other LR parser generators enable embedding semantic actions in a grammar definition. Each production in a grammar can be associated with a semantic action:

```
A  ::=  body₁  {: semantic-action₁ :}
     |  body₂  {: semantic-action₂ :}
        ...
     |  bodyₖ  {: semantic-actionₖ :}
     ;
```

The semantic action i, which is just Java code (if using CUP), 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 (*i.e.*, A), and we can give names to the attributes for the symbols in the body of the production, as seen below with names `e` and `val`. (This attribute grammar implements an interpreter!)

```
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 $\langle \mathit{Symbol}, \mathit{Attribute} \rangle$ 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 when building the IC lexer. The semantic actions in the parser synthesize the attribute values for non-terminals during parsing.

Modern compiler implementations have shifted 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 Plus(e1, e2); :}
    |   NUM:val {: RESULT = new Num(val); :}
    ;
```

where we have the following AST node definitions (given in Scala):

```
abstract class Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Num(value: Int) extends Expr
```

(a) Get code for this exercise, which comes an archive of a single Eclipse/Scala IDE project
   i. Download `https://cs.wellesley.edu/~cs301/labs/cs301-tiny3.tar.gz`.
   ii. Extract it (double-click or `tar xfz cs301-tiny3.tar.gz`) in a Scala IDE workspace.
   iii. `cd` into the project directory and run `make`.
   iv. In Scala IDE, create a new Scala Project, giving `tiny3` as the name so Scala IDE finds and imports the project in your workspace.
   v. After importing, click on the project in the Project Explorer and then select Project → Clean followed by File → Refresh to ensure the project builds correctly.
   vi. `cd` into the project directory and run `make dump` to see the JavaCUP generated state machine. This may be useful later if you have conflicts or JavaCup errors.

(b) Extend the CUP grammar given in `src/tiny/tiny.cup` to implement the grammar for Tiny, with one change: relax the syntax rules to make parentheses optional in expressions (instead of required) and make un-parenthesized addition expressions left-associative.

We provide a JFlex-based lexer (which also allows arbitrary or no whitespace between tokens), a definition of all terminals and non-terminals, and the grammar rules for high-level program structure. Here is the core of the provided CUP grammar:

```
/* Terminals */
terminal PRINT, INPUT, PLUS, EQ, SEMI, LPAREN, RPAREN;
terminal String ID;
terminal Integer NUM;

/* Nonterminals */
nonterminal Program Program;
nonterminal Stmt Stmt;
nonterminal List<Stmt> StmtList;
nonterminal Expr Expr;

/* The grammar (first listed production determines start symbol) */
Program  ::= StmtList:list          {: RESULT = new Program(list); :}
           ;
StmtList ::=                        {: RESULT = ParserUtil.empty(); :}
           | StmtList:l Stmt:s SEMI {: RESULT = ParserUtil.append(l, s); :}
           ;
```

You must implement grammar rules for statements and expressions, along with appropriate semantic actions to build a valid Tiny AST using the same AST structures we developed earlier. Implement left-associativity for addition expressions without rewriting the grammar in any way. Instead, use CUP's `precedence` directive.

(c) Describe the sequence of actions performed by the parser when parsing:

```
x = 1 + (input + 7);
print x;
```

Be sure to describe the attributes for each symbol on the parsing stack each time a production is reduced, and draw the final attribute created for the `Program`. Running your working version can help you with the latter:

```
scala -cp bin:tools/java-cup-11a.jar tiny.Compiler some-program.tiny
```

You need not build the parsing table, etc. Simply describe the actions at a high level (*i.e.*, "shift `NUM` onto stack, with attribute value ..."; "reduce ... to ..., popping off attribute values ... and pushing attribute value ..."; and so on). If you *are* curious about the parsing table, run `make dump` and examine the output to see the parsing table CUP has generated.

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

```
StmtList ::=  | Stmt SEMI StmtList ;
```

Change the `StmtList` rule in `tiny.cup` to be right-recursive. Update the semantic action to produce the same result as before. (Inspect `ParserUtil.scala` for some methods you can call in CUP to manipulate Scala lists, which we use to represent statement lists in program AST nodes.)

(e) It is often considered bad form to use right recursion in grammars for LR parser generators like CUP, if it can be avoided. Why do you think left recursion is preferable to right recursion? (Hint: think about how a 100000-line program would be parsed.)

Bring a paper copy of your `tiny.cup` grammar to our meetings so we can discuss your grammar. You do not need to submit anything electronically.

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 — a brief overview of TAC follows at the end of this document).
   - Discuss the advantages of each representation.
   - For what applications would one representation be preferable to the others?

3. [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, Scheme, or Racket, 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?

4. Begin designing ASTs for IC. You do not need to write any code (that will be part of the project – feel free to start as well, of course), but do think about how to organize your ASTs. What kind of nodes will you have? Will any similar nodes share common parts? How closely will your AST organization mirror the grammar? How will it differ? We will do a lot of separate recursive analyses on ASTs, implementing different behavior for each type of AST node. How will Scala case classes help with this?

TAC

This page summarizes a simple TAC intermediate language. There are many choices as to the exact instructions to include in such a language, and you will probably want to modify and extend this variant when we translate IC programs into TAC.

Instruction Forms

- **Arithmetic and Logic Instructions.**

  The basic instruction forms are:

  ```
  a = b OP c          a = OP b
  ```

  where `OP` can be

an arithmetic operator:	`ADD`, `SUB`, `DIV`, `MUL`
a logic operator:	`AND`, `OR`, `XOR`
a comparison operator:	`EQ`, `NEQ`, `LE`, `LEQ`, `GE`, `GEQ`
a unary operator:	`MINUS`, `NEG`

- **Data Movement Instructions.**

Copy:	`a = b`	
Load/store:	`a = *b`	`*a = b`
Array load/store:	`a = b[i]`	`a[i] = b`
Field load/store:	`a = b.f`	`a.f = b`

- **Branch Instructions.**

Label:	`label L`	
Unconditional jump:	`jump L`	
Conditional jump:	`cjump a L`	(jump to `L` if `a` is true)

- **Function Call Instructions.**

Call with no result:	call f(a_1, ..., a_n)
Call with result:	a = call f(a_1, ..., a_n)

  (Note: there is no explicit TAC representation for parameter passing, stack frame setup, etc.)