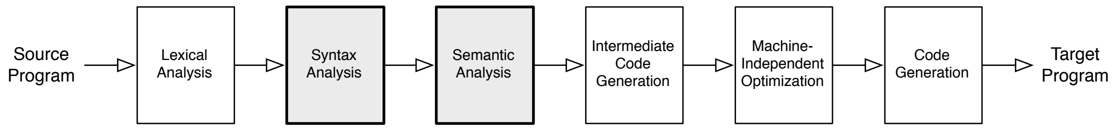
CS 301 Spring 2021 T4
Class Assignment
27 April

Abstract Syntax Trees, Scope

1 Plan



This material explores internal data structures used by a compiler to organize and manipulate a source program. These data structures are the core of a compiler. Developing a good set of abstractions is essential for managing implementation complexity. We focus on construction of *abstract syntax trees* (ASTs) during parsing, preview *symbol tables* for scope and name information, and consider other intermediate representations.

2 Readings

- 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`*
- Abstract Syntax Trees, Intermediate Representations.
  - EC 5.1 – 5.3.
- Scala Case Classes, Pattern Matching, and Traits.
  - `https://docs.scala-lang.org/tour/case-classes.html`
  - `https://docs.scala-lang.org/tour/pattern-matching.html`
  - `https://docs.scala-lang.org/tour/traits.html`
  - More Scala resources as needed: `https://cs.wellesley.edu/~cs301/s21/tools/#scala`
- Roost *Language Specification*, through the *Scope* section
  `https://cs.wellesley.edu/~cs301/s21/project/roost-lang.pdf`
- Scoping and Symbol Tables.
  - EC 5.5

3 Exercises

1. This question requires some programming. We will convert the Tiny compiler to replace the handwritten recursive descent parser with an LALR parser generated by Java CUP from a Tiny grammar.

   The problem explores how CUP and other LR parser generators support semantic actions embedded 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 normal 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 (*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 for simple calculator expressions:

```
// -- Terminals --------------------------------------------------
// The NUM terminal represents a Tiny literal integer value.
// Each NUM token carries an associated value of type Integer.
terminal Integer NUM;

// The PLUS terminal represents a Tiny "+" sign.
// Each PLUS token carries no associated value.
terminal PLUS;

// -- Non-terminals ----------------------------------------------
// The E non-terminal represents a Tiny expression.
// Semantic actions for E productions yield results of type Integer.
nonterminal Integer E;

// -- Precedence Rules -------------------------------------------
// Productions involving PLUS will be made left-associative
// (by resolving shift-reduce conflicts in favor of reduce).
precedence left PLUS;

// -- Grammar Productions and Semantic Actions --------------------
E ::=  E:e1 PLUS E:e2 {: RESULT = e1 + e2; :}
    |  NUM:val {: RESULT = val; :}
    ;
```

In essence, the parser's symbol construction area 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 lexer/scanner must create the attribute values for terminals. The semantic actions in the parser synthesize the attribute values for non-terminals during parsing.

We will use semantic actions to build an abstract syntax tree (AST) data structure. Subsequent analyses inspect and manipulate the AST. Given the following AST node definitions (in Scala):

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

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); :}
    ;
```

In the rest of this exercise, you will reimplement the Tiny parser using a CUP grammar like this one.

(a) Get code for this exercise:

  i. In IntelliJ, choose *New > Project from Version Control*.

  ii. Enter the URL `https://github.com/wellesleycs301s21t4/tiny-cup.git`

  iii. Click *Clone*.

  iv. When prompted about whether to trust and open the BSP project, click *Trust Project*.

  v. IntelliJ should now clone and automatically build the project. Wait for this to finish: watch progress bars at the bottom of the IntelliJ window.

  vi. When IntelliJ has finished importing and building (or immediately if you are not using IntelliJ builds), open a terminal, `cd` to the `tiny-cup` project directory if needed, and run the command `source env.sh`. This produces wrapper scripts for the Tiny compiler and interpreter and put them on the PATH.

(b) Extend the CUP grammar given in `tinyc/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 Seq<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) Build the parser by clicking the green hammer, choosing *Build > Build Project* from the menus, or running `./mill tinyc.build` at the command line. Test your grammar by running `tinyc` on the example Tiny programs in the `test` directory or your own Tiny programs. (If `tinyc` is not found, first run `source env.sh` in the `tiny-cup` project directory.)

(d) Describe the sequence of actions performed by the parser when parsing this Tiny program:

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

The parse should conclude with symbol *Program* and attribute (AST) of the form `Program([...])`. 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: `tinyc some-program.tiny`

You need not build the parsing automaton, table, etc. Simply describe the actions at a high level using **one** of these formats for demonstrating the parse:

i. Show the steps of the parser in this form:

(Symbol, Attribute) stack	Input	Action
	`x=1+(input+7);print x;$`	reduce *StmtList* → (empty)
(*StmtList*, `[]`)	`x=1+(input+7);print x;$`	shift Id `x`
(*StmtList*, `[]`) (Id, `"x"`)	...	...
...	...	...
...	...	reduce *Program* → *StmtList*
(*Program*, `Program([...])`)	$	accept

ii. A sequence of English action descriptions such as:
   - Reduce: pop the empty sequence (pop nothing) and replace with symbol *StmtList* and attribute `[]`.
   - Shift: push the symbol Id with attribute `"x"`.
   - Reduce: pop three symbols/attributes (E, `Num(3)`), (`+`,), (E, `Plus(Num(7),Num(2))`) and replace with symbol/attribute (E, `Plus(Num(3),Plus(Num(7),Num(2)))`).

If you *are* curious about the parsing table (or to help debug issues with your grammar), run `./mill tinyc.dump` and examine the output to see the parsing table CUP has generated.

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

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

It is often considered bad form to use right recursion in grammars for LR parser generators like CUP, if it can be avoided. Why is left recursion preferable to right recursion? (Hint: think about parsing a not-so-tiny 100000-statement Tiny program.)

2. [Adapted from Cooper and Torczon]

   (a) Show how the following Java code fragment might be represented in an abstract syntax tree, in a control flow graph, and in linear three-address code (TAC). Make detailed diagrams, but feel free to invent diagram conventions and notation as needed. A brief overview of an example TAC format follows at the end of this document.

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

   (b) Discuss the advantages of each representation. For what tasks within a compiler would one representation be preferable to the others?

3. Consider the general design of an AST data structure for the Roost compiler. Do not write code yet, but do think about how to organize your ASTs.

   (a) What kind of nodes will you have? Will any similar nodes share common parts?

   (b) How closely will your AST organization mirror the grammar? How will it differ? What grammar details are not important to preserve in the AST?

   (c) We will do many separate recursive analyses on ASTs, implementing different behavior for each type of AST node. How will Scala case classes help with this?

   (d) How might Scala traits be useful?

4. For the following Roost code, which uses generic types (similar to Java, Scala, Rust, Swift) from the Roost standard extensions:

   (a) Mark which syntactic occurrences of program-defined *identifiers* (names) are declarations (definitions) that introduce new names and which are references (uses) that refer to declarations/definitions.

   (b) For each use/reference, indicate the declaration/definition to which it refers. (Draw an arrow to the declaration/definition or give its line number.)

```
 1  extern fn string_length(s: String) -> i64;
 2
 3  struct A<T> {
 4    x: i64,
 5    y: T
 6  }
 7
 8  enum B<U,V> {
 9    C,
10    D(A<U>),
11    E(V)
12  }
13
14  fn f<W>(s: A<W>, mut y: i64, x: W) -> W {
15    {
16      y = y + s.x;
17      let x = 1 + y;
18      y = y + x;
19    }
20    if (y < 0) {
21      x
22    } else {
23      s.y
24    }
25  }
26
27  fn g(s: A<i64>, r: B<A<bool>, String>, q: i64) -> i64 {
28    if (q == 0) {
29      let q = A { x: s.y, y: "Roost" };
30      f(q, s.x, "Compiler")
31    } else {
32      match (r) {
33        C => g(s, E("Roost"), s.x),
34        D(x) => x.x + s.y,
35        E(s) => string_length(s),
36      }
37    }
38  }
```

4 Example TAC Definition

This page summarizes a simple three-address code (TAC) intermediate language. There are many choices as to the exact instructions to include in such a language; we will use something a little different for the Roost compiler.

- **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`	
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.)