Final Notes on Grammars

- Not all useful grammars can be parsed using recursive descent.

One typical problem is left recursion, as in left-associative arithmetic grammars:

\[ E \rightarrow E \ ' + ' \ T \mid T \]

Recursive descent parser would go into infinite loop calling E() routine!

Solution: Rewrite grammar to remove left-recursion.

\[ E \rightarrow T \ ' + ' \ E \mid T \]

 Parses same language but builds wrong tree! Typically use ad-hoc solution to build proper tree.

- Not all useful languages have context-free grammars.

Example: \( L = \{ w c w \mid w \in \{a, b\}^+ \} \) is not context-free.

\( L \) abstracts idea of variable declaration before use.

So “semantic” analysis (e.g., type-checking) uses additional (mostly ad-hoc) techniques.

Informal Semantics

- Grammars can be used to define the legal programs of a language, but not what they do! (Actually, most languages place further, non-grammatical restrictions on legal programs, e.g., type-correctness.)

- Language behavior is usually described, documented, and implemented on the basis of natural-language (e.g., English) descriptions.

- Descriptions are usually structured around the language’s grammar, e.g., they describe what each nonterminal does.

- Natural-language descriptions tend to be imprecise, incomplete, and inconsistent.

Example: FORTRAN DO-loops.

```
"DO \( n \ i = m_1, m_2, m_3 \)
Repeat execution through statement \( n \), beginning with \( i = m_1 \), incrementing by \( m_3 \), while \( i \) is less than or equal to \( m_2 \). If \( m_3 \) is omitted, it is assumed to be 1. \( m \)’s and \( i \)’s cannot be subscripted. \( m \)'s can be either integer numbers or integer variables; \( i \) is an integer variable.”

```

Consider:

```
DO 100 I = 10, 9, 1
...
100 CONTINUE
```

How many times is the body executed?
**Experimental Semantics**

Try it and see!

**Implementation** becomes standard of correctness.

This is certainly **precise**: compiler source code becomes specification.

But it is:

- difficult to understand;
- awkward to use;
- subject to accidental change;
- wholly non-portable.

**Operational Semantics**

Define behavior of language on an **abstract machine**.

Abstract machine should be much **simpler** than real machines, since otherwise a compiler for a real machine would be just as good!

Typical mechanisms:

- Characterize the state of the abstract machine (typically as an **environment** mapping variables to values) and give a set of **evaluation rules** describing how each syntactic construct affects the state.
- Define a simple Von Neumann-style **stack machine** and describe how each syntactic construct can be compiled into stack machine instructions.

Some useful things to do with an operational semantics:

- Build an implementation for a real machine by interpreting or compiling the abstract machine code.
- Explicate the meaning of a language feature by proving that it has the same behavior as a combination of simpler features.
- Prove that correctly typed programs cannot "dump core" at runtime.

**Formal Semantics**

**Aims:**

- **Rigorous** and **unambiguous** definition in terms of a well-understood formalism, e.g., logic, naive set theory, etc.
- Independence from **implementation**. Definition should describe how the language behaves as abstractly as possible.

**Uses:**

- Provably-correct implementations.
- Provably-correct programs.
- Basis for language comparison.
- Basis for language design.

(But usually not basis for learning a language.)

**Main varieties:**

- Operational
- Denotational
- Axiomatic

Each has different purposes and strengths. In this course, we’ll mostly focus on operational semantics.

**Semantics from Interpreters**

Starting this week in the homework, we’ll be building **definitional interpreters** for small languages that display key programming language constructs.

Our goal is to study the interpreter code in order to understand **implementation** issues associated with each language.

In addition, the interpreter serves as a form of **semantic** definition for each language construct. In effect, it defines the meaning of the language in terms of the semantics of Java or ML.

(Of course, you’ll also be learning more about the semantics of Java and ML as we go!)
Syntax-directed Language Processing

Use grammatical structure of language to guide checking, interpretation, translation into lower-level form, etc.

One way to use this structure: traverse parse tree or abstract syntax tree, evaluating semantic rules.

Semantic rules (“attribute equations”):

- Assign values to attributes attached to nodes of tree.
  Examples: type or value of expression; code for statement block.

- Perform side-effects on global state (DEPRECATED!).
  Examples: make entries in symbol table; issue errors; generate code to output file.

Attributes are pieces of information (any kind!) attached to nodes of a grammar-induced tree.

Semantic rules are associated with grammar productions, because each tree node is “built” by a production. (Terminal nodes are assumed to have their attributes “at the beginning.”)

Collectively, semantic rules make up an attribute grammar.

Parse Trees vs. Abstract Syntax Trees

Parse tree reflects details of concrete syntax; for processing a language, we usually want a simpler, more abstract view of the program.

Simple concrete grammar:

\[
E \rightarrow E \cdot + \cdot T \mid E \cdot - \cdot T \mid T \\
T \rightarrow ID \mid NUM \mid '(' E ')' \\
\]

Example concrete parse tree for \(a - (b + 3)\):

```
     E
    / \  \
   E   T
  /   / \\
 T   ID a
 |
 ID b
```

Possible abstract syntax tree for \(a - (b + 3)\):

```
 Sub
/   \
 ID a       Add
         /    \  \
        ID b   NUM 3
```

AST’s obey a tree grammar. Rules have form

\[\text{Label} : \text{Kind} \rightarrow \text{Kind}_1 \ldots \text{Kind}_n\]

where the LHS classifies the possible node labels into kinds, and the RHS describes the kinds of children required.

\[
\text{Add} : E \rightarrow E E \\
\text{Sub} : E \rightarrow E E \\
\text{Id} : E \rightarrow e \\
\text{Num} : E \rightarrow e
\]

There are no firm rules for designing AST grammars: matter of taste, convenience.

Attribute Evaluation

Attribute grammars can be used with parse tree or abstract syntax tree.

Evaluation order of semantic rules may or may not follow reduction order during parser: depends on form of rules.

Computing attribute values is called annotating or decorating the tree.

If used with parse tree, often try to compute attribute values while parsing, instead of building a parse tree.

Example: can use attribute grammar on parse trees to compute AST as an attribute!

More complicated attribute equations may require whole tree to exist first, before attribute evaluation begins.

An attribute is:

- synthesized if its value at a node depends only on values of attributes of descendents of that node; or

- inherited if its value at a node depends only on the values of attributes of ancestors and/or siblings of that node.
Synthesized attributes on Parse Trees

Attribute values at non-terminal node depend only on values at node’s children. Values at terminal nodes are provided by lexical analyzer.

Example: interpreting expression language

```
S → E
E → T
E → E₁ + T
T → F
T → T₁ * F
F → (E)
F → I
I → I₁ digit
I → digit
```

```
S.val := E.val
E.val := T.val
E.val := E₁.val + T.val
T.val := F.val
T.val := T₁.val * F.val
F.val := E.val
F.val := I.val
I.val := 10 * I₁.val + digit.lexval - '0'
I.val := digit.lexval - '0'
```

Here `digit.lexval` is a terminal attribute containing the ASCII code of a digit character.

Attributes can be evaluated bottom-up.

Evaluation could be done on the parse tree, or while parsing (either top-down or bottom-up).

Automated parser generators (like yacc) include support for calculating synthesized attributes easily.

Syntax-directed definitions that use only synthesized attributes are called S-attributed definitions.

Inherited Attributes

Sometimes convenient to make node’s attributes dependent on siblings or ancestors in tree.

Useful for expressing dependence on context, e.g., relating identifier uses to declarations. (Especially important because CFG cannot capture such dependencies.)

Example: Parsing Declarations

```
D → T L
T → int
T → real
L → L₁, id
L₁.type := L.type;
addsymb(id.name, L.type)
L → id
addsymb(id.name, L.type)
```

where `addsymb` adds `id` and its type to symbol table.

Here `L.type` is inherited attribute.

Dependency Graphs

Arrows show dependency relation among attributes. Taken together, arrow describe dependency graph.

Must evaluate attributes in topological order of dependency graph.

If attributes are defined on parse tree, may want to evaluate attributes while (or instead of) building the tree. This is sometimes possible:

- Top-down parsers can easily evaluate L-attributed grammars, in which attributes don’t depend on their right ancestors.
- Bottom-up parsers can easily evaluate S-attributed grammars.
- For some attribute grammars, must build entire tree before evaluating attributes.

Example

“Decorated” parse tree for input `23*5+4`.

```
S.val = 110
E.val = 119
T.val = 4
F.val = 4
I.val = 4
F.val = 5
L.val = 5
digit.lexval = '4'
I.val = 2
digit.lexval = '3'
digit.lexval = '2'
```

Note: Same parse tree and attribute evaluation pattern would hold for static attributes, such as expression type, code sequence, etc.
Attributes on AST’s

Attribute grammar method extends to abstract grammars (not intended for parsing), e.g., AST grammars.

- Same concept, but evaluation always occurs after whole tree is built.
- Can use recursive descent to evaluate (rather than parse).
- Typical applications: typechecking, code generation, interpretation.

Why attribute grammars?

- **Compact**, convenient formalism.
- **Local** rules describe entire computation.
- Separate **traversal** from **computation**.
- (Purely **functional** rules can be evaluated in any order.)

HW 2 Examples

Parser and Interpreter in Java.

1. Build AST as synthesized attribute during parsing.
   - Represent AST using a distinct subclass for each kind of expression.

2. Interpret programs by evaluating attribute grammar over AST:
   - Inherit **environment** mapping variables to integer values.
   - Synthesize integer **value** for each expression.
   - For each subclass of expression, define a function
     ```java
     int eval(Env env)
     ```
     which takes the inherited attribute as argument and returns the synthesized attribute as result.

Names and Binding

One essential part of being a “high-level” language is having convenient **names** for things: operators, variables, constants, types, procedures, classes, etc.

A **binding** is an association between a name and the thing it names.

Bindings can occur at many different **times** in the life-cycle of a language/program.

Key characteristics of any binding:

- **lifetime**: when is the binding created and destroyed?
- **scope**: in what textual region of the program is the binding active?

Examples:

- The C top-level declaration
  ```c
  const int x = 101;
  ```
  binds the identifier `x` to the constant integer 101. This is a **static** binding whose lifetime is the entire program execution and whose scope is (roughly) the entire file in which it appears.

More Examples

- The C top-level declaration
  ```c
  int x = 101;
  ```
  binds the identifier `x` to a constant global memory address, whose initial contents is the integer 101. This is also a **static** binding, although the contents of that address may change during program execution; same lifetime and scope as above.

- The C declaration
  ```c
  foo () {
    int x;
    ...
  }
  ```
  binds `x` to a memory address in the stack activation record for `foo` (or to a machine register). The binding is **dynamic**; it’s lifetime is just while `foo` is actively executing. It’s scope is just the body of `foo`. Different activations of `foo` will produce different versions of the binding.
Binding Lifetime vs. Object Lifetime

Many bindings are to memory-allocated “objects” (variables, records, subroutine activation records, etc.).

The lifetime of a memory object is the period from when it is allocated to when it is deallocated for possible re-use.

The lifetime of a binding to an object is not necessarily the same as the object’s lifetime.

- Objects often remain alive even when there is (temporarily) no live binding to them.

Example: Memory allocated to x remains alive even while bar is executing, but binding is (temporarily) dead.

```c
foo () {
    int x = 0;
    bar ();
    x = x+1;
}
```

Storage Classes

<table>
<thead>
<tr>
<th>Static Data : Permanent Lifetimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Global variables and constants.</td>
</tr>
<tr>
<td>- Allows fixed address to be compiled into code.</td>
</tr>
<tr>
<td>- No runtime management costs.</td>
</tr>
<tr>
<td>- Original FORTRAN (no recursion) used static activation records.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Data : Nested Lifetimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Allocation/deallocation is cheap (just adjust stack pointer).</td>
</tr>
<tr>
<td>- Most architectures support cheap sp-based addressing.</td>
</tr>
<tr>
<td>- Good locality for VM systems, caches.</td>
</tr>
<tr>
<td>- C, Algol/Pascal family, Java use stack for activation records.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap Data : Arbitrary Lifetimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Requires explicit allocation and (dangerous) explicit deallocation or garbage collection.</td>
</tr>
<tr>
<td>- Lisp, ML, many interpreted languages need heap for activation records, which have non-nested lifetimes.</td>
</tr>
</tbody>
</table>

Lifetimes (continued)

- Objects sometimes die even when there is still a live binding for them. This is usually a bug (a dangling pointer) and many languages try to prevent it:

Example: p still points to the allocated storage, even though it may now be reallocated for something else!

```c
char *foo() {
    char *p = malloc(100);
    free(p);
    return p;
}
```

Storage Organization

- Subdivide machine address space by function, access, allocation.
- Typical organization (Unix)

```
∞ -------------------------
    STACK                “top” of stack (but stacks nearly always grow DOWN!)
    ↓                    (in VM systems, these pages don’t actually exist)
    ↑                    (controlled by sbrk system call)
    ↓                    Managed by allocator/collector.
    ↓                    Uninitialized (“b[lock] s[tatic] s[torage]”) Initialized (in object file)
    ↓                    CODE Read-only (in object-file)
      0
```
Lexical Scope

VAR a : INT = 0
PROCEDURE f(b:INT) IS
   RETURN a+b
END
PROCEDURE g(c:INT) IS
   VAR a : INT = 100
   a := a + 100;
   RETURN f(c)
END
a := -100;
PRINT(g(10))

When \( f \) is executed, it needs to look up a value for \( a \), which is a free variable of \( f \), i.e., not a local variable or parameter. Which \( a \) is visible?

Under lexical scope rules, variables are identified by looking backwards through the program text to find the nearest enclosing declaration.

- In this case, \( f \) would use the global declaration of \( a \).
- At the time \( f \) executes, this has the value -100, so -90 is printed.
- Nearly all languages use lexical scope.

Dynamic Scope

VAR a : INT = 0
PROCEDURE f(b:INT) IS
   RETURN a+b
END
PROCEDURE g(c:INT) IS
   VAR a : INT = 100
   a := a + 100;
   RETURN f(c)
END
a := -100;
PRINT(g(10))

Under dynamic scope rules, variables are identified by looking backwards through the program execution to find the most recently executed declaration.

- In this case, \( f \) would use the local declaration of \( a \) within \( g \).
- At the time \( f \) executes, this has the value 200, so 210 is printed.
- Early versions of LISP and some scripting languages use dynamic scope.
- Dynamic scope makes procedures act like macros.

Block-structured languages

fun sum_list(a:int list) : int =
   let val sum = ref 0
   fun traverse(l:int list) : unit =
      case l of
      nil => ()
      | (h::t) =>
         (sum := !sum + h;
         traverse t)
   in traverse(a);
   !sum
   end
val c : int list = ...;

- In Pascal, Ada, ML, etc., we can nest procedure declarations inside other procedure declarations. (Cannot do this in C,C++,Java!)
- Parameters and local variables of outer procedures are visible within inner procedures (using lexical scoping rules).
- More precisely, the variables associated with the most recent still-live activation of the outer procedure are visible within inner procedures.

Block structure (continued)

- In most languages, if procedure \( f \) is declared inside \( g \), then \( f \) can only appear as descendent of \( g \) in the activation tree. This allows us to stack-allocate activation records, and still guarantee that non-local variables will still exist when they are needed.
- In most languages, procedures themselves are potentially recursive, so their names are visible within their own bodies.
Environment Chaining

To find non-local variables on the stack while executing procedure \( f \), it suffices to know the environment that was in place when the declaration of \( f \) was encountered.

- An interpreter can simply attach the current variable environment to its description of \( f \) when it encounters \( f \)'s declaration and records it in the function environment.
- When the interpreter applies \( f \), it evaluates its body in an initial environment taken from the recorded description, which is then extended with \( f \)'s parameters and locals.
- When the interpreter looks up a variable while executing \( f \), it looks first among \( f \)'s locals and parameters, and then in the lexically-enclosing environment.

A compiler can precompute the positions of variables relative to the start of each procedure activation record, supporting fast runtime access to variables. It is still necessary to pass to each nested function (at runtime) a pointer to its lexically enclosing function’s activation record, or equivalent information. Thus, nested functions carry some (modest) runtime cost.

Procedures as Parameters

It can be handy to pass procedures as parameters to other procedures. This feature is supported by Pascal, etc., by ML, and by C/C++ (but not directly by Java).

Example (ML)

```ml
type int_action = integer -> unit

fun do_intlist (a: int list, f: int_action) : unit = case a of
  nil => ()
| h::t => (f h; do_list (t, f))

fun print_int (i: int) : unit = print (Int.toString i)

val sum = ref 0
fun sum_int (i: int) : unit = sum := !sum + i

val a : int list = ...;
do_intlist (a, print_int);
do_intlist (a, sum_int);
print_int (!sum);
```

Using Local (Nested) Procedures

- Sometimes want to pass local functions as parameters.

Example: Improved version of sum:

```ml
fun sum_list (a: int list) : int = let
  val sum = ref 0
  fun sum_int (i: int) = sum := !sum + i
  in do_intlist (a, sum_int);
  !sum
end

val c : int list = ...;
print_int (sum_list (c));
```

- Here `sum_int` operates on the value of variable `sum`, which is neither local nor global.
- Implementation: pass pair of (code-pointer, env-pointer) as “value” of procedure.
- Must guarantee that env pointer is still valid when procedure is called!
- Cannot express this in C.
More Nested Procedures

Example: Use iterator to count how many times specified integer occurs.

```plaintext
fun count(i:int, a:int list) =  
let val sum = ref 0  
fun check_int(j:int) =  
  if j = i then  
    sum := !sum + 1  
  else ()  
  in do_intlist(a,check_int);  
  !sum  
in sum  
end
val c : int list = ...;
...count(17,c);...
```

- Here check_int depends on the value of variable i, which is neither local nor global.
- Going one step further, can be handy to treat **procedure values** just like other values, e.g., to return them as function results or store them into variables.

---

**“First-class” Procedures Example**

```plaintext

type counter = int list -> int

fun make_counter(i:int) : counter =  
  let fun count(a: int list) =  
    let val sum = ref 0  
    fun check_int(j:int) =  
      if j = i then  
        sum := !sum + 1  
      else ()  
      in do_intlist(a,check_int);  
        !sum  
in sum  
    end
    in count  
  end
val g : counter = make_counter(17);  
...  
val c : int list = ...;
val d : int list = ...;
val c17 : int = g(c);  
val d17 : int = g(d);
```
Functional Languages

The distinctive flavor of functional languages derives from being able to return functions that are locally defined.

For convenience, they also support anonymous function values, i.e., a function can be denoted without declaring a name for it. E.g., in ML: fn x => x + 1.

Functional languages allow more sophisticated abstraction mechanisms. Consider the following problems:

Sum a list of integers:

```ml
fun sum l = 
  case l of
    nil => 0
  | h::t => h + (sum t)
```

Multiply a list of integers:

```ml
fun prod l = 
  case l of
    nil => 1
  | h::t => h * (prod t)
```

Folds

We can abstract over the common inductive pattern displayed by these examples:

```ml
fun foldr (f,n) = 
  let fun r l =
    case l of
      nil => n
    | h::t => f(h,r t)
  in r
  end

fun sum l = foldr(fn (x,y) => x+y, 0) l
fun prod l = foldr(op*, 1) l
fun copy l = foldr(incr, nil) l
fun length l = foldr(fn (_,y) => 1+y, 0) l
```

Folds (continued)

Function foldr computes a value working from the tail of the list to the head (from right to left). Argument n is the value to return for the nil list. Argument f is the function to apply to each element and the previously computed result.