Lecture 12 Outline

• Finish type checking
• Syntax directed translation
• Syntax directed definitions & schemes
• Synthesized & inherited attributes
• S-attributed definition

Reading Assignment:

Section 5.1 - 5.3

Expressiveness of Static Type Systems

• A static type system enables a compiler to detect many common programming errors
• The cost is that some correct programs are disallowed
  - Some argue for dynamic type checking instead
  - Others argue for more expressive static type checking
• But more expressive type systems are also more complex

Dynamic And Static Types

• The dynamic type of an object is the class C that is used in the "new C" expression that created it
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
• The static type of an expression is a notation that captures all possible dynamic types the expression could take
  - A compile-time notion
Dynamic and Static Types. (Cont.)

- In early type systems the set of static types correspond directly with the dynamic types.
- Soundness theorem: for all expressions E
  \[ \text{dynamic_type}(E) = \text{static_type}(E) \]
  (in all executions, E evaluates to values of the type inferred by the compiler)
- This gets more complicated in advanced type systems.

Dynamic and Static Types in COOL

- A variable of static type A can hold values of static type B, if \( B \leq A \)

```
class A { … }
class B inherits A {…}
class Main {
    A x ← new A;
    ...
    x ← new B;
    ...
}
```

- x has static type A
- Here, x’s value has dynamic type A
- Here, x’s value has dynamic type B

Dynamic and Static Types

Soundness theorem for the Cool type system:
\[ \forall E. \text{dynamic_type}(E) \leq \text{static_type}(E) \]

Why is this Ok?
- All operations that can be used on an object of type C can also be used on an object of type \( C' \leq C \)
- Such as fetching the value of an attribute
- Or invoking a method on the object
- Subclasses can only add attributes or methods
- Methods can be redefined but with same type!
An Example

class Count {
    i : int ← 0;
    inc () : Count {
        i ← i + 1;
        self;
    };
};

- Class Count incorporates a counter
- The inc method works for any subclass
- But there is disaster lurking in the type system

An Example (Cont.)

- Consider a subclass Stock of Count
  class Stock inherits Count {
      name : String; -- name of item
  };
- And the following use of Stock:
  class Main {
      Stock a ← (new Stock).inc ();  // Type checking error!
      ... a.name ...
  };

What Went Wrong?

- (new Stock).inc() has dynamic type Stock
- So it is legitimate to write
  Stock a ← (new Stock).inc()
- But this is not well-typed
  - (new Stock).inc() has static type Count
  - The type checker "looses" type information
  - This makes inheriting inc useless
    - So, we must redefine inc for each of the subclasses, with
      a specialized return type
SELF_TYPE to the Rescue

- We will extend the type system
- Insight:
  - inc returns "self"
  - Therefore the return value has same type as "self"
  - Which could be Count or any subtype of Count!
- Introduce the keyword SELF_TYPE to use for the return value of such functions
  - We will also need to modify the typing rules to handle SELF_TYPE.

SELF_TYPE to the Rescue (Cont.)

- SELF_TYPE allows the return type of inc to change when inc is inherited
- Modify the declaration of inc to read inc() : SELF_TYPE {...}
- The type checker can now prove:
  \[ C,M (\text{new Count}).\text{inc}() : \text{Count} \]
  \[ C,M (\text{new Stock}).\text{inc}() : \text{Stock} \]
- The program from before is now well typed.

Notes About SELF_TYPE

- SELF_TYPE is not a dynamic type but it is a static type.
- It helps the type checker to keep better track of types.
- It enables the type checker to accept more correct programs.
- In short, having SELF_TYPE increases the expressive power of the type system.
SELF_TYPE and Dynamic Types (Example)

- What can be the dynamic type of the object returned by inc?
  - Answer: whatever could be the type of "self"
    
```plaintext
class A inherits Count { } ; 
class B inherits Count { } ; 
class C inherits Count { } ; 

(inc could be invoked through any of these classes)
```
  - Answer: Count or any subtype of Count

SELF_TYPE and Dynamic Types (Example)

- In general, if SELF_TYPE appears textually in the class C as the declared type of E then 
  \[ \text{dynamic}_\text{type}(E) \leq C \]
- Note: The meaning of SELF_TYPE depends on where it appears
  - We write SELF_TYPE\textsubscript{C} to refer to an occurrence of SELF_TYPE in the body of C
  - This suggests a typing rule:
    \[ \text{SELF_TYPE\textsubscript{C}} \leq C \]

(*)

Type Checking

- Rule (*) has an important consequence:
  - In type checking it is always safe to replace SELF_TYPE\textsubscript{E} by C
  - This suggests one way to handle SELF_TYPE:
    - Replace all occurrences of SELF_TYPE\textsubscript{E} by C
  - This would be correct but it is like not having SELF_TYPE at all
Operations on SELF_TYPE

- Recall the operations on types
  - \( T_1 \leq T_2 \): \( T_1 \) is a subtype of \( T_2 \)
  - \( \text{lub}(T_1, T_2) \): the least-upper bound of \( T_1 \) and \( T_2 \)

- We must extend these operations to handle SELF_TYPE

Extending \( \leq \)

Let \( T \) and \( T' \) be any types but SELF_TYPE.

There are four cases in the definition of \( \leq \):

1. \( \text{SELF}_{\text{TYPE}} \leq \text{SELF}_{\text{TYPE}} \)
   - In Cool we never need to compare SELF_TYPES coming from different classes

2. \( \text{SELF}_{\text{TYPE}} \leq T \) if \( C \leq T \)
   - \( \text{SELF}_{\text{TYPE}} \) can be any subtype of \( C \)
   - This includes \( C \) itself
   - Thus this is the most flexible rule we can allow

3. \( T \leq \text{SELF}_{\text{TYPE}} \) always false
   - Note: \( \text{SELF}_{\text{TYPE}} \) can denote any subtype of \( C \).

4. \( T \leq T' \) (according to the rules from before)

   Based on these rules we can extend \( \text{lub} \) ...
Extending lub(T, T')

Let T and T' be any types but SELF_TYPE
Again there are four cases:
1. lub(SELF_TYPE C, SELF_TYPE C) = SELF_TYPE C
2. lub(SELF_TYPE C, T) = lub(C, T)
   This is the best we can do because SELF_TYPE C ≤ C
3. lub(T, SELF_TYPE C) = lub(C, T)
4. lub(T, T') defined as before

Syntax directed translation
Translation guided by context free grammars
- attributes attached to grammar symbol represent information about the grammar symbol
- values for attributes computed by semantic rules attached to each production

Associate semantic rules with productions
- syntax directed definitions -
  - define high level specifications for translations
  - don’t specify order of evaluation/translation - hides implementation details
- syntax directed translation schemes
  - order in which semantic rules are to be evaluated
  - shows more implementation details
Both notations used for specifying semantic checking
- determining types and then type checking
- generating intermediate code/representation
Conceptual view of syntax-directed translation
- Parse input string
- Build parse tree
- Traverse tree as needed to evaluate the semantic rules
As a result of the evaluation
  a) generate code
  b) modify symbol table
  c) issue error messages as a result of type checking
  d) compute information needed by other rules

<table>
<thead>
<tr>
<th>input</th>
<th>parse tree</th>
<th>dependency graphs</th>
<th>evaluation order for semantic rules</th>
</tr>
</thead>
</table>

Semantic definitions
Sometimes semantic rules can be evaluated during parsing
  - no parse tree needed
  - no graph showing dependencies
  i.e., single pass implementation may be possible
  important for achieving compiler time efficiency
L-attributed definitions is a class of definitions for
which evaluation can be performed without explicit
constructions of parse tree

Syntax directed definitions
Generalization of a context-free grammar in which each
grammar symbol has an associated set of attributes
2 types:
- synthesized attributes
- inherited attributes

Each node in a parse tree has a record with fields
holding attribute values.
Attributes

- a string
- a number
- a type
- a memory location

Value of attributes at parse tree node defined by semantic rule associated with production used at that rule.

Synthesized attributes

Values computed from attributes of the children node

\[ P = f(c_1, c_2, c_3, c_4) \]

\[ S_4 = f(P, s_1, s_2, s_3) \]

Inherited attributes

Value of inherited attributes computed from attributes the siblings and parent of the node

Semantic rules create dependencies between attributes:
- represent the dependencies as a graph
- from graph, derive evaluation order for semantic rules
Annotated parse tree
Parse tree showing values of attributes
Computing of attributes at nodes is called
- annotating or decorating the parse tree

Form of syntax-directed definition
inherited

synthesized

Attributed grammar
Each grammar production \( A \rightarrow \alpha \) has semantic rules of the form

\[ b = f(c_1, c_2, c_3, \ldots, c_k) \]

Either

1. \( b \) is synthesized attribute of \( A \) and \( c_1, c_2, c_3, \ldots, c_k \) are attributes of grammar symbols of rhs
2. \( b \) is an inherited attribute of one of the grammar symbols of rhs of production and \( c_i \)'s are attribute of \( A \) and/or other symbols on the rhs

Example

- Terminal symbols - have synthesized attributes only which are usually provided by the lexical analyzer - no rules for synthesized attributes of terminals
- Start symbol - is assumed not to have any inherited attributes
**S-attributed**

Synthesized attributes used extensively in practice

A syntax directed definition that only used synthesized attributes

- S-attributed definition

Can always evaluate easily in bottom-up parser

- LR parser generator can be adapted to mechanically implement an S-attribute definition based on an LR grammar

---

**Example S-attributed**

- Example:

  
  ```
  L -> E n  print(E.val)
  E -> E1 + T  E.val = E1.val + T.val
  E -> T  E.val = T.val
  T -> T1 * F  T.val = T1.val * F.val
  T -> F  T.val = F.val
  F -> (E)  F.val = E.val
  F -> digit  F.val = digit.lexval
  
  Note: n is end of line
  ```

---

```latex
\begin{aligned}
E.val &= 6 \\
T.val &= 6 \\
F.val &= \text{digit} = 3 \\
\text{digit} &= 2 \\
\end{aligned}
```

---

```latex
\begin{aligned}
E.val &= 10 \\
T.val &= 4 \\
F.val &= 4 \\
\text{digit} &= 4 \\
\end{aligned}
```
Inherited attributes

- convenient for expressing the dependence of a programming language construct in the context in which it appears
  e.g., inherited attribute can keep track of whether an identifier appears on the left or right side of an assignment
  e.g., inherited attribute can keep track of type information

Can write a grammar that has only synthesized attributes but more natural to use inherited attributes.

Example: inherited attributes

Example:

T - synthesized attribute "type"
L - has inherited attribute "in"

\[
\begin{align*}
D & \rightarrow TL & \text{L.in} = \text{T.type} \\
T & \rightarrow \text{int} & \text{T.type} = \text{integer} \\
T & \rightarrow \text{real} & \text{T.type} = \text{real} \\
L & \rightarrow \text{L1, id} & \text{L1.in} = \text{L.in}, \text{addtype(id.entry, L.in)} \\
L & \rightarrow \text{id} & \text{addtype(id.entry, L.in)}
\end{align*}
\]

int id1, id2, id3

\[
\begin{align*}
\text{T.type} & = \text{int} \\
\text{L.in} & = \text{T.type} \\
\text{L.in} & = \text{L.in} \\
\text{id} & = \text{addtype(id.entry, L.in)} \\
\text{id} & = \text{addtype(id.entry, L.in)} \\
\text{id} & = \text{addtype(id.entry, L.in)} \\
\text{id} & = \text{addtype(id.entry, L.in)}
\end{align*}
\]

edges show dependencies
Dependency graph

From dependencies can construct dependency graph shows order in which evaluation must be performed—assuming acyclic graph

do topological ordering to evaluate nodes

\[
\begin{align*}
& n_i \quad \text{evaluate } n_i \text{ before } n_j \\
& n_j \quad n_j \text{ is dependent on } n_i
\end{align*}
\]

Construct abstract syntax tree

Can use syntax directed definitions to construct abstract syntax tree -

- a syntax tree is essentially a restructured/condensed parse tree which is used as an intermediate representation
- syntax trees allow decoupling of translation from parsing

Example

\[
S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2
\]

\[
\begin{array}{c}
S \\
\text{if } B \text{ then } S_1 \text{ else } S_2 \\
B \quad S_1 \quad S_2
\end{array}
\]

The operators and keywords/reserve words do not appear as leaves - instead associated with the parent node in the parse tree

\[
\begin{array}{c}
E \\
T \quad F \\
* \quad * \\
3 \quad 4 \\
2
\end{array}
\]
Semantic rules

- \text{mkleaf}(id, entry) \& \text{mkleaf}(num, val)
  \quad \text{entry is pointer to symbol table entry}
- \text{mknode}(op, left, right)
- Abstract syntax trees can be constructed in bottom-up fashion by calling the above functions
- Consider synthesized attribute evaluation using bottom-up parser

Productions

<table>
<thead>
<tr>
<th>E</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E_1 + T )</td>
<td>( E.nptr = \text{mknode}(\text{'+'}, E_1.nptr, T.nptr) )</td>
</tr>
<tr>
<td>( E \rightarrow E_1 - T )</td>
<td>( E.nptr = \text{mknode}(-,E_1.nptr, T.nptr) )</td>
</tr>
<tr>
<td>( E \rightarrow T )</td>
<td>( E.nptr = T.nptr )</td>
</tr>
<tr>
<td>( T \rightarrow (E) )</td>
<td>( T.nptr = E.nptr )</td>
</tr>
<tr>
<td>( T \rightarrow \text{id} )</td>
<td>( T.nptr = \text{mkleaf}(\text{id}, \text{id}.entry) )</td>
</tr>
<tr>
<td>( T \rightarrow \text{num} )</td>
<td>( T.nptr = \text{mkleaf}(\text{num}, \text{num}.val) )</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow Z Y X )</td>
<td>$s_1$</td>
</tr>
</tbody>
</table>

Each reduction put something new on stack

- Since synthesized attribute depends upon the children, it can be computed from values on top of semantic stack - TOP - 1, TOP - 2, TOP - 3, etc.
Execute semantic rules which evaluate attributes before performing the reduction -
- information is available on the stack
- semantic rules associated with reduction

Another class of grammers

Left ~ L-attributed

have both inherited and synthesized attributes but restrictions are put on inherited so that they can be evaluated using depth - first order of the parse tree - left to right

Order: go down tree evaluating inherited & up evaluating synthesized

Order: go down tree evaluating inherited & up evaluating synthesized

Information flows from left to right
Syntax directed definitions based upon LL(1) grammars are L-attributed

Rules for L-attributed definition
A syntax directed definition is L-attributed if an attributed is synthesized or each inherited attribute of \( X_j \) in \( A \rightarrow X_1X_2...X_n \) depends only on
1. attributes of symbols \( X_1, ...X_{j-1} \) i.e. symbols to the left of \( X_j \)
2. inherited attributes of \( A \)

Every S-attributed definition is L-attributed because restrictions apply to inherited attributes and S-attributed definition has no inherited attributes

Depth first traversal - traversal of many top down and bottom up parser
Translation schemes -

- Specify the order in a context-free grammar in which attributes are associated with the grammar symbols
- Semantic actions encoded in {} are inserted within (or at the end) of the rhs of the productions
- Both synthesized and inherited attributes are allowed.
- Must take into account when attributes are known.

Convert Infix to postfix grammar

```
E -> TR
R -> + T {print(addop.lexime)} R | ε
R -> - T {print(minusop.lexeme)} R | ε
T -> num {print(num.val)}
```

Depth-first search

```
9 - 5 + 2 = 95 - 2 +
```

Syntax directed Schemes: make sure attributes are available

- Synthesized attributes - easiest case place semantic rule/action at the end of the rhs
  
  \[ T \rightarrow T_1 \cdot f \ (T_{val} = T_1_{val} \cdot F_{val}) \]

If you have both, how do you convert a definition to a translation scheme - L-attributed special case

The following rules are needed

1. An inherited attribute for a symbol on the rhs must be computed in an action before the symbol

2. Action must not refer to synthesized attributes of a symbol to the right of the action

3. A synthesized attribute for the non-terminal on lhs can only be computed after all attributes it references have been computed. The action can be placed at the end of the rhs production.
• It is always possible to start with L-attributed definition and construct translation scheme which satisfies the above rules (in fact above restrictions are motivated by L-attributed defn)
• Translation schemes satisfying above requirements can be implemented by generalization of top-down and bottom up parsing

Top-down translation of L-attributed
• L-attributed definition can be implemented during predictive parsing
• translation scheme to show evaluation order
• natural order of evaluation for inherited is top-down parser so must consider synthesized attributes

E -> E1 + T                    E.val = E1.val + T.Val
E -> E1 - T                    E.val = E1.val - T.Val
E -> T                           E.val = T.val
T -> num                        T.val = num.val

All attributes are synthesized but
• left associative operator => left recursion used in grammar
• elimination of left recursion algorithm is used by top-down process - show how translation scheme is affected/transformed

E -> TR
R -> +TR
R -> - TR
R -> ε
T -> num
T -> (E)

String 3+4+5

inherited & synthesized attributes will be used
T = synthesized attribute ~ val
R = inherited attribute ~ i
synthesized attribute ~ s
E = synthesized attribute ~ val
Transformed Translation Scheme - Rules apply after expansion

\begin{align*}
E & \rightarrow T (R.i = T.val) \\
R & \rightarrow T (R.i = R.i + T.val) \\
R & \rightarrow - T (R.i = R.i - T.val) \\
R & \rightarrow \varepsilon (R.i = R.i) \\
T & \rightarrow (E) (T.val = E.val) \\
T & \rightarrow \text{num} (T.val = \text{num}) \\
\end{align*}

Bottom-up evaluation of inherited attributes:
Action taken at time of reduction => at end of production
- synthesized is natural at the end of production i.e., during reduction
- inherited in the middle - how to trigger the action?
Removal of the embedded actions from translation scheme must be carried out
Rules should be put at the end of the production
Introduce markers - non-terminals - where the rule was
For the marker introduce grammar rule

\[ M \rightarrow \varepsilon \]

\begin{align*}
e.g.,
D & \rightarrow T \ L \\
\text{want semantic action: } & D \rightarrow T (L.in = T.type) \ L \\
\text{change to:}
D & \rightarrow T \ M \ L \\
M & \rightarrow \varepsilon (L.in = T.type) \\
\end{align*}

Generally:
\[ A \rightarrow X \ (\text{rule}) \ Y = \]
\[ A \rightarrow X \ M \ Y \]
\[ M \rightarrow \varepsilon \ (\text{rule}) \]
Natural to compute synthesized attribute \( A \rightarrow XY \) at reduction time - synthesized attribute of \( A \) can be computed from attributes of \( X \) & \( Y \)

**Inherited attributes**

If \( Y \) has an inherited attribute which it inherits from \( X \) through copy rule \( Y.i = X.s \) it can do so because the attributes of \( X \) are on the stack before reductions below \( Y \) (in the stack) take place

Example: Translation scheme we want to implement

\[
\begin{align*}
D & \rightarrow T & (L.in = T.type) \\
T & \rightarrow \text{int} & (T.type = \text{integer}) \\
T & \rightarrow \text{real} & (T.type = \text{real}) \\
L & \rightarrow & (L.in = \text{in}) \\
L & \rightarrow \text{id} & (\text{addtype(} \text{id.entry, } L.in) \\
L & \rightarrow & (\text{addtype(} \text{id.entry, } L.in)
\end{align*}
\]

**How to achieve this in a bottom-up parser?**

\[
\begin{align*}
\text{real} & \rightarrow \text{real} \\
\text{p,q,r} & \rightarrow \text{real} \\
\text{p,q,r} & \rightarrow T & T \rightarrow \text{real} \\
\text{q,r} & \rightarrow TL & L \rightarrow \text{id} \\
\text{q,r} & \rightarrow TL & \text{addtype(} \text{val[top], } \text{val[top-3])} \\
\text{q,r} & \rightarrow TL & \text{addtype(} \text{val[top], } \text{val[top-1])} \\
\text{r} & \rightarrow D & D \rightarrow TL
\end{align*}
\]

\( T \) is always just below on the stack every time an \( \text{id} \) is reduced to \( L \) - production \( L \rightarrow L, \text{id} \) or \( L \rightarrow \text{id} \)

We can access this attribute using \( T.type \) i.e., replace \( L.in \) by \( T.type \) since both are the same anyway

\( L.in \) is computed using copy rule \( L.in = T.type \)

When \( L \rightarrow \text{id} \) is applied \( id \) is on top of stack & \( T.type \) is at top-1 portion of the stack

When \( L \rightarrow L, \text{id} \) is applied, \( id \) is on top of the stack & \( T.type \) is at top-3 of the stack

By examining the grammar we can predict the position were the type is

\[
\begin{align*}
D & \rightarrow T L : \\
T & \rightarrow \text{int} & \text{val[top] = integer} \\
T & \rightarrow \text{real} & \text{val[top] = real} \\
L & \rightarrow L, \text{id} & \text{addtype(} \text{val[top], } \text{val[top-3])} \\
L & \rightarrow \text{id} & \text{addtype(} \text{val[top], } \text{val[top-1])}
\end{align*}
\]
Cannot always predict the position where the attribute is added so help to determine where it is

S -> aAC  \( C.i = A.s \)
S -> bABC  \( C.i = A.s \)
C -> c  \( C.s = g(C.i) \)

C inherits attribute from A by copy rule
There may or may not be a B between A & C

When reduction \( C \rightarrow c \) the required value is either at Top-1 or Top-2

\[
\begin{align*}
S & \rightarrow aAC  \quad C.i = A.s \\
S & \rightarrow bABC  \quad M.i = A.s; C.i = M.s \\
C & \rightarrow c  \quad C.s = g(C.i) \\
M & \rightarrow \epsilon  \quad M.s = M.i
\end{align*}
\]


Sometimes change grammar to avoid inherited attributes - replacing inherited attributes by synthesized attributes so, another solution can be used in bottom-up parsers

\[
\begin{align*}
D & \rightarrow L : T \\
T & \rightarrow \text{integer} | \text{char} \\
L & \rightarrow L, \text{id} | \text{id}
\end{align*}
\]

L's attributes inherited from T
- not L attributed

\[
\begin{align*}
D & \rightarrow \text{id} L \\
L & \rightarrow \text{id} L ; T \\
T & \rightarrow \text{integer} | \text{char}
\end{align*}
\]

L's attributes synthesized from T's