Semantic Analysis

- Often called "Contextual analysis"
  - As opposed to our syntax analysis — which was "context free"
- Properties that cannot be formulated via CFG
  - Declare before use
  - Type checking
  - Initialization
- Properties that are clumsy to formulate via CFG
  - "break" only appears inside a loop
  - "..."
Semantic Analysis

- **Identification**
  - Gather information about each named item in the program
  - e.g., what is the declaration for each usage

- **Context checking**
  - Type checking
  - e.g., the condition in an if-statement is a Boolean

Identification

```plaintext
month : integer RANGE 1..12;
month := 1;
while (month <= 12) {
  print(month_name[month]);
  month := month + 1;
}
```

Symbol table

- A table containing information about identifiers in the program
- Single entry for each named item

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>month</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>month_name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Not so fast...

```plaintext
struct one_int {
  int i;
} i;

main() {
  i.i = 42;
  int t = i.i;
  printf("%d",t);
}
```

Not so fast...

```plaintext
struct one_int {
  int i;
} i;

main() {
  i.i = 42;
  int t = i.i;
  printf("%d",t);
  { int i = 73; printf("%d",i); }
```
Scopes

- Typically: stack structured scopes
- Scope entry
  - push new empty scope element
- Scope exit
  - pop scope element and discard its content
- Identifier declaration
  - identifier created inside (current) top scope
- Identifier Lookup
  - Search for identifier top-down in scope stack

Scope-structured Symbol Table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phil</td>
<td>Long</td>
<td>Bob</td>
<td>Sarah</td>
</tr>
</tbody>
</table>

Scope and Symbol Table

- Scope × Identifier → properties
  - Expensive lookup
- A better solution
  - Hash table over identifiers
  - List of scopes for each identifier

Hash Table-based Symbol Table

Scope info

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phil</td>
<td>Long</td>
<td>Bob</td>
<td>Sarah</td>
</tr>
</tbody>
</table>
Remember Lexing+Parsing?
- How did we know to always map an identifier to the same token?
- We didn’t! now it is the first time.

Semantic Checks
- Scope rules
  - Use symbol table to check that
    - No multiple definition of same identifier
    - Identifiers defined before used
- Type checking
  - Check that types in the program are consistent
    - How?

Types
- What is a type?
  - Simplest answer: a set of values
    - Integers, real numbers, booleans, ...
- Why do we care?
  - Safety
    - Guarantee that certain errors cannot occur at runtime
  - Abstraction
    - Hide implementation details
  - Documentation
  - Optimization

Type System
- A type system of a programming language is a way to define how “good” programs behave
  - Good programs = well-typed programs
  - Bad programs = not well typed
- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime
- Type inference
  - Automatically infer types for a program (or show that there is no valid typing)

Static Typing vs. Dynamic Typing
- Static type checking is conservative
  - Any program that is determined to be well-typed is free from certain kinds of errors
  - May reject programs that cannot be statically determined to be safe
  - Why?
- Dynamic type checking
  - May accept more programs as valid (runtime info)
  - Errors not caught at compile time
  - Runtime overhead
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - formal and actual parameters of a method call

- Examples
  
  ```
  string    string
  "drive" + "drink"

  int       string
  42 + "the answer"
  ```
  
  ERROR

Type Checking Rules

- Specify for each operator
  - Types of operands
  - Type of result

- Basic Types
  - Building blocks for the type system (type rules)
  - e.g., int, boolean, [sometimes] string

- Type Expressions
  - Array types
  - Function types
  - Record types / Classes

Typing Rules

If \( E_1 \) has type \( \text{int} \) and \( E_2 \) has type \( \text{int} \),
then \( E_1 + E_2 \) has type \( \text{int} \)

\[
\begin{array}{c@{}c@{}c}
E_1 : \text{int} & E_2 : \text{int} \\
\hline
E_1 + E_2 : \text{int}
\end{array}
\]

More Typing Rules (examples)

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>+</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>op(+, -, /, %)</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>&lt;=, &lt;, &gt;=</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>==, !=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array</th>
<th>Function</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1 : int</td>
<td>E2 : int</td>
<td>E1 : T</td>
</tr>
</tbody>
</table>

And Even More Typing Rules

| boolean | boolean | logical (\&\&, ||) |
|---------|---------|------------------|
| E1 : int | E2 : boolean |
| E1 \&\& E2 : boolean |

<table>
<thead>
<tr>
<th>Array</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1 : T</td>
<td>E2 : int</td>
</tr>
</tbody>
</table>
Type Checking
- Traverse AST and assign types for AST nodes
  - Use typing rules to compute node types
- Alternative: type-check during parsing
  - (Slightly) more complicated
  - But naturally also more efficient

Type Rules

Strongly Typed vs. Weakly Typed
- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - Perl, PHP

Coming Up
Reminder — Semantic Analysis

- Identification
  - Read declarations, build symbols table & scope table
  - Associate declaration and uses

- Context checking
  - Type checking: check that the program is type-safe
  - e.g., the condition in an if-statement is a Boolean

Type Declarations

- So far, we ignored the fact that types can also be user-defined

```
TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;
Var a : ARRAY [Integer 1..42] OF Real;
```

```
TYPE #type01_in_line_73 = ARRAY [Integer 1..42] OF Real;
Var a : #type01_in_line_73;
```
Forward References

- Forward references must be resolved
- A forward reference is added to the symbol table (as unresolved), and later updated when the type declaration is met
- At the end of scope, check that all forward refs have been resolved
- Must add check for circularity

```plaintext
TYPE Ptr_List_Entry = POINTER TO List_Entry;

TYPE List_Entry =
  RECORD
    Element : Integer;
    Next : Ptr_List_Entry;
  END RECORD;
```

Type Table

- All types in a compilation unit are collected in a type table
- For each type, its table entry contains
  - Type constructor: basic, record, array, pointer,…
  - Size and alignment requirements
    - to be used later in code generation
  - Types of components (if applicable)
    - e.g., type of array element, types of record fields

Nominal Type System

Type Equivalence = Name Equality

- Type t1 = ARRAY[Integer] OF Integer;
- Type t2 = ARRAY[Integer] OF Integer;
- t1 not (nominally) equivalent to t2

- Type t3 = ARRAY[Integer] OF Integer;
- Type t4 = t3;
- t3 equivalent to t4

Structural Type System

Type Equivalence = Structure Isomorphism

- Type t5 = RECORD c: Integer; p: POINTER TO t5; END RECORD;
- Type t6 = RECORD c: Integer; p: POINTER TO t6; END RECORD;
- Type t7 =
  RECORD
    c: Integer;
    p: POINTER TO
      RECORD
        c: Integer;
        p: POINTER TO t5;
      END RECORD;
    END RECORD;
- t5, t6, t7 are all (structurally) equivalent

In Practice

- Almost all modern languages use a nominal type system
  - Why?
Coercions

• If we expect a value of type T1 at some point in the program, and find a value of type T2, is that acceptable?

```plaintext
float x = 3.141;  
int y = x;  
✗
```
Subtyping

• A basic concept in Object-oriented Programming
  ▪ Every class has (can have) a superclass

```java
class GeniusMouse {
    public BluetoothMouse {
        public:
        void initiatePairing();
        rk_t bond(BluetoothSocket& socket);
    }
}
```

Subtyping

• Subtyping relation:
  ▪ "T is a sub-type of S"
  ▪ Handled with corresponding typing rule

```
E : T < S
```

• As a consequence:
  ▪ Each term has more than one type
  ▪ Need to find the appropriate type for each context

Subtyping

• Subtyping relation:
  ▪ "T is a sub-type of S"
  ▪ Handled with corresponding typing rule

```
E : T < S
```

• As a consequence:
  ▪ Each term has more than one type
  ▪ Need to find the appropriate type for each context

Subtyping

• Marking some of the types as pure abstract helps alleviate some of the difficulty in multiple inheritance

```
Mouse
```

• Marking some of the types as pure abstract helps alleviate some of the difficulty in multiple inheritance

```
Mouse
```
Subtyping

• Upcast
  • The compiler will insert an implicit conversion from subtype to supertype (similar to coercions)

Mouse m = new GeniusMouse();
registerDevice(m);

Mouse m = GeniusMouse_to_Mouse(
new GeniusMouse());
registerDevice(Mouse_to_Device(m));

So far...

• Static correctness checking
  • Identification
    • Type checking
      • Identification matches applied occurrences of identifier to its defining occurrence
      • The symbol table maintains this information
  • Type checking checks which type combinations are legal
• Each node in the AST of an expression represents either an l-value (location) or an r-value (value)

How does this magic happen?

• We probably need to go over the AST?

• How does this relate to the clean formalism of the parser?

Syntax Directed Translation

• Semantic attributes
  • Attributes attached to grammar symbols
  • Semantic actions
    (already mentioned when we learned recursive descent)
    • Defines how to update the attributes
  • Attribute grammars

Attribute Grammars

• Attributes
  • Every grammar symbol has attached attributes
    • Example: Expr.type
  • Semantic actions
  • Every production rule can define how to assign values to attributes

Expr → Expr + Term
{ if (second Expr).type == Term.type
  (first Expr).type = (second Expr).type;
  else error;
}
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

```
Expr → Expr + Term

Expr → Expr + Term
  { if (Expr.type == Term.type)
    Expr.type = Expr.type;
  else error; }
```

Example

```
float x,y,z
```

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.dt = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L 1, id</td>
<td>L 1.dt = L.dt; addType(id.entry, L.dt)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.dt)</td>
</tr>
</tbody>
</table>

Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before it’s available
    - Each attribute will get a value only once

Dependencies

- A semantic equation \( a = f(b_1, ..., b_m) \)
  requires computation of \( b_1, ..., b_m \) to determine the value of \( a \)

- The value of \( a \) depends on \( b_1, ..., b_m \)
  - We write \( a \leftarrow b_i \)

Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering
- Works as long as there are no cycles
Building Dependency Graph

- All semantic equations take the form
  \( \text{attr}_1 = f_1(\text{attr}_{1,1}, \text{attr}_{1,2}, \ldots) \)
  \( \text{attr}_2 = f_2(\text{attr}_{2,1}, \text{attr}_{2,2}, \ldots) \)

- Actions with side effects use a dummy attribute
- Build a directed dependency graph \( G \)
  - For every attribute \( a \) of a node \( u \) in the AST create a node \( u.a \)
  - For every dependency between attributes \( u.a_1 \leftarrow v.a_2 \) create an edge of the form \( (v.a_2, u.a_1) \)

Example

```
float x, y, z
```

Topological Order

- For a graph \( G=(V, E), |V|=k \)
- Ordering of the nodes as \( \langle v_1, v_2, \ldots, v_k \rangle \) such that for every edge \( (v_i, v_j) \in E, i < j \)

Example topological orderings: \( \langle 1\ 4\ 3\ 2\ 5 \rangle, \langle 4\ 3\ 5\ 1\ 2 \rangle \)
But what about cycles?

- For a given attribute grammar — hard to detect if it has cyclic dependencies
  - Exponential cost

- Special classes of attribute grammars
  - Our “usual trick”: sacrifice generality for predictable performance

Synthesized vs. Inherited Attributes

- **Synthesized attributes**
  - Attributes whose values at a given node depend only on the attributes of its children (and itself)

- **Inherited attributes**
  - Attributes whose values at a given node depend only on the attributes of its parent and siblings

---

S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
  - No use of inherited attributes
- Can be computed by any bottom-up parser during parsing — no need to construct dependency graph
  - Attributes can be stored on the parsing stack
  - Reduce operation computes the (synthesized) attribute from attributes of children
S-attributed Grammars

Production | Semantic Rule
---|---
S → E
E → E + T
E → T
T → T1 * F
T → F
F → ( E )
F → digit
E.val = E1.val + T.val
E.val = T.val
T.val = T1.val * F.val
T.val = F.val
F.val = E.val
F.val = digit.lexval

Arithmetic Calculator

L-attributed Grammars

- L-attributed attribute grammar: when every attribute in a production \( A \rightarrow X_1...X_n \) is either
  - A synthesized attribute, or
  - An inherited attribute of \( X_j \), \( 1 \leq j \leq n \) that only depends on
    - Attributes of \( X_1...X_{j-1} \) to the left of \( X_j \)
    - Inherited attributes of \( A \)

- In recursive-descent parsers:
  - Pass inherited attributes down as arguments
  ```
  D() {
    T();
    L();
  }
  L() {
    match(ID);
    A();
  }
  A() {
    if (current == COMMA) {
      match(COMMA);
      L();
    } else if (current == EOF) {
    } else error();
  }
  T() {
    if (current == INT) {
      match(INT);
      return {type: 'integer'};
    } else if (current == FLOAT) {
      match(float);
      return {type: 'float'};
    } else error();
  }
  ```

L-attributed Grammars

- In recursive-descent parsers:
  - Pass inherited attributes down as arguments
  ```
  D() {
    Attrs t = T();
    L({d t => t.type});
  }
  L(Attrs ih) {
    Token id = match(ID);
    addType(id.entry, ih.dt);
    A({d t => ih.in});
  }
  A(Attrs ih) {
    if (current == COMMA) {
      match(COMMA);
      A({d t => ih.in});
    } else if (current == EOF) {
    } else error();
  }
  ```
L-attributed Grammars

In shift-reduce parsers:

- Use marker variables

Marker Variables

Since a marker only appears in one production rule, it is commonly abbreviated:

\[ D \rightarrow TL \quad \text{action} \quad D \rightarrow T \text{act}\]

Action \( M \) is called a mid-rule action.

It is important to remember that adding mid-rule actions inherently changes the grammar. The marker variable is there even if it is not explicitly visible.

Marker Variables

In particular, marker variables and the associated \( \epsilon \)-productions can violate your grammar’s \( LR(0)/SLR/LALR/LR(1) \)-ness

Summary

✓ Contextual analysis can move information between nodes in the AST
✓ Even when they are not “local”
✓ Attach attributes and semantic actions to grammar
✓ Attribute evaluation
✓ Build dependency graph, topological sort, evaluate
✓ Special classes with pre-determined evaluation order: S-attributed, L-attributed
Coming Up

- Intermediate Representation