**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
  - ...

---

```
Potato potato;
Tomato tomato;
x = potato + tomato + carrot
```

---

```
… <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF
```

---

```
'carrot' is undefined
'potato' used before initialized
```

---

```
Identifier = potato
Identifier = tomato
Identifier = carrot
```

---

```
Semantic Analysis
```

---

```
What We Want
```

---

```
Executable code
```

---

```
Source text
```

---

```
Lexical analysis
```

---

```
Semantic Analysis
```

---

```
Intermediate representation (IR)
```

---

```
Code generation
```

---

```
Target code
```

---

```
Back end
```

---

```
Compiler
```

---

```
Lexical Analysis
```

---

```
Syntax Analysis
```

---

```
Intermediate code generation
```

---

```
Annotated AST
```

---

```
Semantic Analysis
```

---

```
Process text input
```

---

```
text
```

---

```
characters
```

---

```
tokens
```

---

```
AST
```

---

```
Intermediate code
```

---

```
optimization
```

---

```
IR
```

---

```
Code generation
```

---

```
Target code optimization
```

---

```
Symbolic Instructions
```

---

```
Machine code generation
```

---

```
Write executable output
```

---

```
MI
```

---

```
BI
```

---

```
Back end
```

---

```
What We Want
```
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

### Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>month</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>month_name</td>
<td>string</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

Scope stack

0

1

2

3

Identifier info

Name

Macro

Decl

Scope info

Scope stack

0

1

2

3

Id.info("so")

Id.info("long")

Id.info("and")

Id.info("thanks")

Id.info("x")

Id.info("all")

Id.info("the")

Id.info("fish")

Id.info("thanks")

Id.info("x")

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

Id info

Name

Macro

Decl

Scope info

Scope stack

0

1

2

3

Id.info("so")

Id.info("long")

Id.info("and")

Id.info("thanks")

Id.info("x")

Id.info("all")

Id.info("the")

Id.info("fish")

Id.info("thanks")

Id.info("x")

(now just pointers to the corresponding record in the symbol 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
    - Identifiers defined before used
    - No multiple definition of same identifier
- 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"
  string
  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 int and $E_2$ has type int, then $E_1 + E_2$ has type int

$$
E_1 : \text{int} \quad E_2 : \text{int} \\
E_1 + E_2 : \text{int}
$$

More Typing Rules (examples)

```java
true : boolean
E1 : int
E2 : int
E1 op E2 : int
```

```java
false : boolean
int-literal : int
string-literal : string
```

```java
E1 : int
E2 : int
E1 rop E2 : boolean
```

```java
rop \in \{ \leq, <, >, \geq \}
E1 : int
E2 : int
E1 rop E2 : boolean
```

```java
lop \in \{ ==, \neq \}
E1 : int
E2 : int
E1 rop E2 : boolean
```

And Even More Typing Rules

```java
E1 : boolean
E2 : boolean
E1 op E2 : boolean
```

```java
E1 : int
E2 : int
E1 || E2 : boolean
```

```java
E1 : int
E2 : int
E1 && E2 : boolean
```

```java
E1 : T[*]
E2 : T
E1[E2] : T
```

```java
new T[E1] : T[*]
```

```java
E1.length : int
E1[E2] : T
```

```java
new T[E1][E2] : T[*]
```
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

THEORY OF COMPILATION
LECTURE 05
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

```plaintext
TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;

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

```
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;
```

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
int x = 22;
float y = x;
```

```plaintext
int x = 22;
float y = int_to_float(x);  // ✓
```

l-values and r-values

- What is dst? What is src?
  - dst is a memory location where the value should be stored
  - src is a value
- "location" on the left of the assignment called an l-value
- "value" on the right of the assignment is called an r-value

```plaintext
dst := src
```

l-values and r-values (example)

```plaintext
x := y + 1
```

```plaintext
A[1] := x + 1  // ✓
A[A[1]] := x + 1  // ✓
x + 1 := A[1]  // ✗
```
Subtyping

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

```java
class GeniusMouse extends BluetoothMouse {
  public void initiatePairing();
  public Rk_t bond(BluetoothSocket socket);
}
```

Subtyping

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

\[
E_1 : T < : S \quad \text{if} \quad T : S
\]

• 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

```java
Mouse
  BluetoothMouse
  GeniusMouse
  LogitechMouse

USBDevice
  USBMouse
```

Subtyping

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

```java
Mouse
  BluetoothMouse
  GeniusMouse
  LogitechMouse

USBDevice
  USBMouse
```
Subtyping

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

```java
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

```java
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

\[ \text{Expr} \rightarrow \text{Expr} + \text{Term} \]

\[ \text{Expr} \rightarrow \text{Expr}_1 + \text{Term} \]

\[
\begin{cases}
\text{if (Expr\_type == Term\_type)} \\
\text{Expr\_type = Expr\_type} \\
\text{else error}
\end{cases}
\]

Example

```
float x, y, z
D \rightarrow T L
D.type = float
L.in = T.type
T \rightarrow int
T.type = integer
T \rightarrow float
T.type = float
L \rightarrow L, id
L.in = L.in
addType(id.entry, L.in)
L \rightarrow id
addType(id.entry, L.in)
```

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 \( a = f(b_1, \ldots, b_m) \) requires computation of \( b_1, \ldots, b_m \) to determine the value of \( a \)
- The value of \( a \) depends on \( b_1, \ldots, b_m \)
  - 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

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: \( \langle 1, 4, 3, 2, 5 \rangle \), \( \langle 4, 3, 5, 1, 2 \rangle \)
Example

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 Grammar

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E ;</td>
<td><code>print(E.val)</code></td>
</tr>
<tr>
<td>E → E + T</td>
<td><code>E.val = E1.val + T.val</code></td>
</tr>
<tr>
<td>E → T</td>
<td><code>E.val = T.val</code></td>
</tr>
<tr>
<td>T → T1 * F</td>
<td><code>T.val = T1.val * F.val</code></td>
</tr>
<tr>
<td>T → F</td>
<td><code>T.val = F.val</code></td>
</tr>
<tr>
<td>F → ( E )</td>
<td><code>F.val = E.val</code></td>
</tr>
<tr>
<td>F → digit</td>
<td><code>F.val = digit.lexval</code></td>
</tr>
</tbody>
</table>

L-attributed Grammar

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

L-attributed Grammars

- In recursive-descent parsers:
  - Pass inherited attributes down as arguments

```java
D() {
  Attrs t = T();
  L(in \mapsto t.type);
}
L(Attrs ih) {
  Token id = match(ID);
  addType(id.entry, ih.in);
  A(in \mapsto ih.in);
}
A(Attrs ih) {
  if (current == COMMA) {
    match(COMMA);
    L(in \mapsto ih.in);
  } else if (current == EOF) {
  } else error();
}
T() {
  if (current == INT) {
    match(INT);
    return {type \mapsto int};
  } else if (current == FLOAT) {
    match(float);
    return {type \mapsto float};
  } else error();
}
```
L-attributed Grammars

- In shift-reduce parsers:
  - Use marker variables

Marker Variables

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

  \[ D \rightarrow TL \mid \text{action} \quad \] 

  \[ D \rightarrow T \mid \text{action} \quad \] 

- \text{action}_x \] 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"
- Attribute grammars
  - 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