Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leaves of a syntax tree

Implementing Numbering of Leafs

Idea:
- use helper attributes `pre` and `post`
- in `pre` we pass the value of the last leaf down (inherited attribute)
- in `post` we pass the value of the last leaf up (synthetic attribute)

```
root:
    pre[0] := 0
    pre[1] := pre[0]
    post[0] := post[1]

node:
    pre[1] := pre[0]
    post[0] := post[2]

leaf:
    post[0] := pre[0] + 1
```
The Local Attribute Dependencies

- the attribute system is apparently strongly acyclic
- each node computes
  - the inherited attributes before descending into a child node (corresponding to a pre-order traversal)
  - the synthetic attributes after returning from a child node (corresponding to post-order traversal)
- if all attributes can be computed in a single depth-first traversal that proceeds from left to right (with pre- and post-order evaluation)
- then we call this attribute system $L$-attributed

---

$L$-attributed

**Definition**

An attribute system is $L$-attributed, if for all productions $s ::= s_1 \ldots s_n$
every inherited attribute of $s_j$ where $1 \leq j \leq n$ only depends on
1. the attributes of $s_1, s_2, \ldots, s_{j-1}$ and
2. the inherited attributes of $s$.
L-attributed

Definition
An attribute system is L-attributed, if for all productions $s ::= s_1 \ldots s_n$
every inherited attribute of $s_j$ where $1 \leq j \leq n$ only depends on
1. the attributes of $s_1, s_2, \ldots s_{j-1}$ and
2. the inherited attributes of $s$.

Origin:
- the attributes of an L-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator

L-attributed grammars have a fixed evaluation strategy: a single depth-first traversal
- in general: partition all attributes into $A = A_1 \cup \ldots \cup A_n$ such that for all attributes in $A_i$, the attribute system is L-attributed
- perform a depth-first traversal for each attribute set $A_i$
- craft attribute system in a way that they can be partitioned into few L-attributed sets

Practical Applications
- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars

L-attributed

Definition
An attribute system is L-attributed, if for all productions $s ::= s_1 \ldots s_n$
every inherited attribute of $s_j$ where $1 \leq j \leq n$ only depends on
1. the attributes of $s_1, s_2, \ldots s_{j-1}$ and
2. the inherited attributes of $s$.

Origin:
- the attributes of an L-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator

L-attributed grammars have a fixed evaluation strategy: a single depth-first traversal
- in general: partition all attributes into $A = A_1 \cup \ldots \cup A_n$ such that for all attributes in $A_i$, the attribute system is L-attributed
- perform a depth-first traversal for each attribute set $A_i$
- craft attribute system in a way that they can be partitioned into few L-attributed sets
Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using \(L\)-attributed grammars

Implementation of Attribute Systems

In object-oriented languages, use a visitor pattern:

```java
public abstract class Regex {
    public abstract void accept(Visitor v);
}
```

- by overwriting one of the following methods, we implement an attribute-specific evaluation

```java
public interface Visitor {
    void accept(a);  
    void accept(E e);  
    void accept(Regex this);  
    void accept(Brakre this);  
    ...
    void accept(Token tok) {}  
}
```

- we pre-define a depth-first traversal of the syntax tree

```java
public class OrEx extends Regex {
    Regex this;
    public void accept(Visitor v) { v.visit(this); }
    public void children(Visitor v) {
        l.accept(v); r.accept(v);
    }
}
```

Semantic Analysis

Chapter 2:
Symbol Tables
Scope of Identifiers

```java
void foo() {
    int A;
    void bar() {
        double A;
        A = 0.5;
        write(A);
        :: A
    }
    A = 2;
    bar();
    write(A);
}
```

Visibility Rules in Object-Oriented Languages

```java
public class Foo {
    int x = 17;
    protected int y = 5;
    private int z = 42;
    public int b() { return 1; }
}
```

```java
class Bar extends Foo {
    protected double y = 0.5;
    public int b(int a) {
        return a+x;
    }
}
```

Observations:
- private member z is only visible in methods of class Foo
- protected member y is visible in the same package and in sub-class Bar, but here it is shadowed by double y
- Bar does not compile if it is not in the same package as Foo
- methods b with the same name are different if their arguments differ as static overloading
Visibility Rules in Object-Oriented Languages

1 public class Foo {
2   int x = 17;
3   protected int y = 5;
4   private int z = 42;
5   public int b() { return 1; }
6 }
7 class Bar extends Foo {
8   protected double y = 0.5;
9   public int b(int a) {
10      return a+x; }
11 }

Modifications:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>no modifier</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Observations:

- private member z is only visible in methods of class Foo
- protected member y is visible in the same package and in sub-class Bar, but here it is shadowed by double y
- Bar does not compile if it is not in the same package as Foo
- methods b with the same name are different if their arguments differ

Dynamic Resolution of Functions

public class Foo {
   protected int foo() { return 1; }
}

class Bar extends Foo {
   protected int foo() { return 2; }
   public int test(boolean b) {
      Foo x = (b) ? new Foo() : new Bar();
      return x.foo();
   }
}

Modifications:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>no modifier</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Observations:

- the type of x is Foo or Bar, depending on the value of b
- x.foo() either calls foo in line 2 or in line 5
- this decision is made at run-time and has nothing to do with name resolution
Resolving Identifiers

**Observation:** each identifier in the AST must be translated into a memory access

**Problem:** for each identifier, find out what memory needs to be accessed by providing *rapid* access to its declaration

**Idea:**

1. *rapid* access: replace every identifier by a *unique* “name”, namely an integer
   - integers as keys: comparisons of integers is faster
   - replacing various identifiers with number saves memory

(1) Replace each Occurrence with a Number

Rather than handling strings, we replace each string with a unique number.

**Idea for Algorithm:**

**Input:** a sequence of strings

**Output:**

- sequence of numbers
- table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier in the scanner.

Example for Applying this Algorithm

**Input:**

<table>
<thead>
<tr>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a peck of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Output:**

<table>
<thead>
<tr>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a peck of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

whereas the peck of pickled peppers Peter Piper picked
Example for Applying this Algorithm

Input:

Peter Piper picked a peck of pickled peppers

If Peter Piper picked a peck of pickled peppers

whereas the peck of pickled peppers Peter Piper picked

Output:

0 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7
9 10 4 5 6 7 0 1 2

and

0 Peter
1 Piper
2 picked
3 a
4 peck
5 of
6 pickled
7 peppers
8 if
9 whereas
10 the

Implementing the Algorithm: Specification

Idea:

- implement a partial map: \( S : \text{String} \to \text{int} \)
- use a counter variable \( \text{count} = 0 \) to track the number of different identifiers found so far

We thus define a function \( \text{int \ getIndex(String w)} \):

\[
\text{int \ getIndex(String w)} \quad \{
    \quad \text{if} \ (S(w) \equiv \text{undefined}) \quad \{
        \quad S = S \oplus \{w \mapsto \text{count}\};
        \quad \text{return} \ \text{count}++;
    \quad \text{else \ return} \ S(w);
    \}
\]

Data Structures for Partial Maps

possible data structures:

- list of pairs \((w, i) \in \text{String} \times \text{int}\):
  - \(O(1)\)
  - \(O(n)\)
  - \(\sim \) too expensive \(X\)

Data Structures for Partial Maps

possible data structures:

- list of pairs \((w, i) \in \text{String} \times \text{int}\):
  - \(O(1)\)
  - \(O(n)\)
  - \(\sim \) too expensive \(X\)
- balanced trees:
  - \(O(\log(n))\)
  - \(O(\log(n))\)
  - \(\sim \) too expensive \(X\)
An Implementation using Hash Tables

- allocated an array $M$ of sufficient size $m$
- choose a hash function $H : \text{String} \rightarrow [0, m - 1]$ with the following properties:
  - $H(w)$ is cheap to compute
  - $H$ distributes the occurring words equally over $[0, m - 1]$
- Possible choices $H_0(x) = (x_0 + x_{m-1}) \mod m$
- $H_1(x) = (\sum_{i=0}^{m-1} x_i \cdot p^i) \mod m$
- $H_2(x) = (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{m-1} \ldots))) \mod m$
  for some prime number $p$ (e.g. 31)
- We store the pair $(w, i)$ in a linked list located at $M[H(w)]$

Resolving Identifiers: (2) Symbol Tables

Check for the correct usage of variables:
- Traverse the syntax tree in a suitable sequence, such that
  - each definition is visited before its use
  - the currently visible definition is the last one visited
- for each identifier, we manage a stack of scopes
- if we visit a declaration of an identifier, we push it onto the stack
- upon leaving the scope, we remove it from the stack
- if we visit a usage of an identifier, we pick the top-most declaration from its stack
- if the stack of the identifier is empty, we have found an error

Computing a Hash Table for the Example

With $m = 7$ and $H_0$ we obtain:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>

In order to find the index for the word $w$, we need to compare $w$ with all words $x$ for which $H(w) = H(x)$

Resolving Identifiers: (2) Symbol Tables

Check for the correct usage of variables:
- Traverse the syntax tree in a suitable sequence, such that
  - each definition is visited before its use
  - the currently visible definition is the last one visited
- for each identifier, we manage a stack of scopes
- if we visit a declaration of an identifier, we push it onto the stack
- upon leaving the scope, we remove it from the stack
- if we visit a usage of an identifier, we pick the top-most declaration from its stack
- if the stack of the identifier is empty, we have found an error
Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
- equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

\[ a \]
\[ b \]

in front of if-statement

\[ a \]
\[ b \]
\[ a \]
\[ b \]
\[ a \]
\[ b \]

in front of if-statement then-branch else-branch

Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
- equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

\[ a \]
\[ b \]
\[ a \]
\[ b \]
\[ a \]
\[ b \]
\[ a \]
\[ b \]

in front of if-statement then-branch else-branch

Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
- equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

\[ a \]
\[ c \]
\[ a \]
\[ c \]
\[ a \]
\[ c \]
\[ a \]
\[ c \]

in front of if-statement then-branch else-branch

- instead of lists of symbols, it is possible to use a list of hash tables ~ more efficient in large, shallow programs
Alternative Resolution of Visibility

- Resolving identifiers can be done using an L-attributed grammar
- Equation system for basic block must add and remove identifiers
- When using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

![Diagram of a graph with nodes a, b, and c, showing traversal paths.]

- In front of if-statement, then-branch, else-branch
- Instead of lists of symbols, it is possible to use a list of hash tables → more efficient in large, shallow programs
- A more elegant solution is to use a persistent tree in which an update returns a new tree but leaves all old references to the tree unchanged.
  - A persistent tree can be passed down into a basic block where new elements may be added; after examining the basic block, the analysis proceeds with the unchanged.

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- A recursive definition needs to mention a name that is currently being defined or that will be defined later on.
- Old-fashion programming languages require that these cycles are broken by a forward declaration.

Consider the declaration of an alternating linked list in C:

```c
struct list1;
struct list0 { double info;
  int info;
  struct list1* next;
  struct list0* next;
};
```

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- A recursive definition needs to mention a name that is currently being defined or that will be defined later on.
- Old-fashion programming languages require that these cycles are broken by a forward declaration.

Consider the declaration of an alternating linked list in C:

```c
struct list1;
struct list0 { double info;
  int info;
  struct list1* next;
  struct list0* next;
};
```

~ The first declaration `struct list1;` is a forward declaration.
Declarations of Function Names
An analogous mechanism is need for (recursive) functions:

- in case a **recursive function** merely calls itself, it is sufficient to add the name of a function to the symbol table before visiting its body; example:

```c
int fac(int i) {
    return i*fac(i-1);
}
```

- for **mutually recursive functions** all function names at that level have to be entered (or declared as forward declaration). Example ML and C:

```c
fun odd 0 = false
   | odd i = true
   | odd x = even (x-1)
and even 0 = true
   | even i = false
   | even x = odd (x-1)
```

```c
int even(int x);
int odd(int x) {
    return (x==0 ? 0 : (x==1 ? 1 : even(x-1)));
}
```

Overloading of Names

The problem of using names before their declarations are visited is also common in object-oriented languages:

- for object-oriented languages with inheritance, the base class must be visited before the derived class in order to determine if declarations in the derived class are correct
- in addition, the signature of methods needs to be considered
  - qualify a function symbol with its parameters
  - may also require resolution of type names

Once the names are resolved, other semantic analyses can be applied such as **type checking** or **type inference**.

Overloading of Names

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

In some cases a declaration may change the class of an identifier; for example, a **typedef** in C:

- the scanner generates a different token, based on the class into which an identifier falls
- the parser informs the scanner as soon as it sees a declaration that changes the class of an identifier
- the parser generates a syntax tree that depends on the semantic interpretation of the input so far
Multiple Classes of Identifiers

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and constructors

In some cases a declaration may change the class of an identifier; for example, a typedef in C:

- the scanner generates a different token, based on the class into which an identifier falls
- the parser informs the scanner as soon as it sees a declaration that changes the class of an identifier
- the parser generates a syntax tree that depends on the semantic interpretation of the input so far

the interaction between scanner and parser is problematic!

Fixity-Declarations in Haskell

Haskell allows for arbitrary binary operators over \((? ! ^ \& | = + \_ \* /)^+\). In Standard Library of Haskell:

\[
\begin{align*}
\text{infixr} & \quad 8 \quad ^ \land \\
\text{infixl} & \quad 7 \quad *, / \\
\text{infix} & \quad 6 \quad +, - \\
\text{infix} & \quad 4 \quad ==, /=
\end{align*}
\]

The grammar is generic:

\[
\begin{align*}
\text{Exp}_0 & ::= \text{Exp}_0 \text{LOp}_0 \text{Exp}_1 \\
& \mid \text{Exp}_1 \text{ROp}_0 \text{Exp}_0 \\
& \mid \text{Exp}_1 \text{Op}_0 \text{Exp}_1 \\
& \mid \text{Exp}_1 \\
& \vdots \\
\text{Exp}_0 & ::= \text{Exp}_0 \text{LOp}_0 \text{Exp} \\
& \mid \text{Exp}_0 \text{ROp}_0 \text{Exp}_0 \\
& \mid \text{Exp} \\
\text{Exp} & ::= \text{ident} \mid \text{num} \\
& \mid (\text{Exp}_0)
\end{align*}
\]

- parser enters an infix declaration into a table
- scanner checks table and produces:
  - operator \(-\) turns into token \(-\)
  - operator \(*\) turns into token \(*\)
  - operator \(==\) turns into token \(==\)
  - etc.

\[3 - 4 \times 5 - 6\] as
\[3 - (4 \times 5) - 6\]

Fixity-Declarations in Haskell: Observations

Troublesome changes:

- the scanner has a state which the parser determines
  - grammar no longer context-free, needs global data structure

Fixity-Declarations in Haskell: Observations

Troublesome changes:

- the scanner has a state which the parser determines
  - grammar no longer context-free, needs global data structure
- a code fragment may have several semantics
- syntactic correctness may depend on imported modules
Fixity-Declarations in Haskell: Observations

Troublesome changes:
- the scanner has a state which the parser determines
  ~ grammar no longer context-free, needs global data structure
- a code fragment may have several semantics
- syntactic correctness may depend on imported modules
- error messages difficult to understand

The GHC Haskell Compiler parses all operators as \texttt{LOp}_n and transforms the AST afterwards.

Type Synonyms and Variables in C

The C grammar distinguishes typedef-name and identifier. Consider the following declarations:

\[
\texttt{typedef struct \{ int x,y \} point_t;}
\]

Relevant C grammar:
- declaration \[ \rightarrow (\text{declaration-specifier})^+ \texttt{declarator} \]
- declaration-specifier \[ \rightarrow \texttt{static|volatile} \ldots \texttt{typedef} \]
- \[ \texttt{void|char|char} \ldots \texttt{typedef-name} \]
- declarator \[ \rightarrow \texttt{identifier} \ldots \]

\textit{Problem:}
- parser adds point\_t to the table of types when the declaration is reduced
Type Synonyms and Variables in C

The C grammar distinguishes `typedef-name` and `identifier`. Consider the following declarations:

```c
typedef struct { int x, y } point_t;
point_t origin;
```

Relevant C grammar:
- `declaration` → `(declaration-specifier)^+ declarator`
- `declaration-specifier` → `static | volatile ... typedef`
- `typedef-name` → `identifier`
- `declarator` → `identifier`

**Problem:**
- The parser adds `point_t` to the table of types when the `declaration` is reduced.
- The parser state has at least one look-ahead token.

**Solution:**
- Try to fix the look-ahead inside the parser.
- Add the following rule to the grammar:
  - `typedef-name` → `identifier`
- Register type name earlier.

---

Type Synonyms and Variables in C: Solutions

Relevant C grammar:
- `declaration` → `(declaration-specifier)^+ declarator`
- `declaration-specifier` → `static | volatile ... typedef`
- `typedef-name` → `identifier`
- `declarator` → `identifier`

Solution is difficult:
- Try to fix the look-ahead inside the parser.
- Add the following rule to the grammar:
  - `typedef-name` → `identifier`
- Register type name earlier.
  - Separate rule for `typedef` production.
  - Call alternative declarator production that registers `identifier` as type name.