Script generated by TTT

Title: Simon: Compilerbau (02.06.2014)

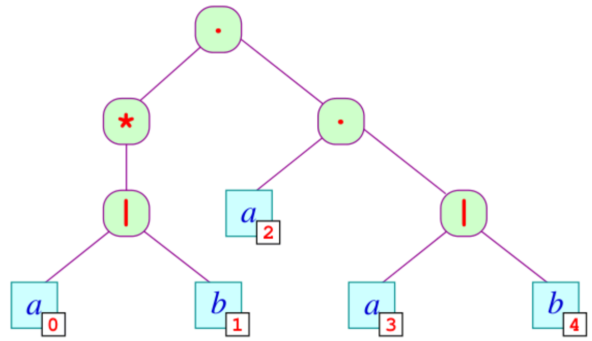
Date: Mon Jun 02 14:15:51 CEST 2014

Duration: 90:47 min

Pages: 63

Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leafs 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: $\text{pre}[0] := 0$
$\text{pre}[1] := \text{pre}[0]$
$\text{post}[0] := \text{post}[1]$

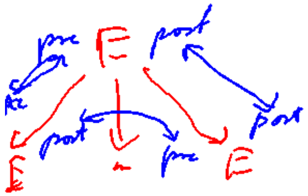
node: $\text{pre}[1] := \text{pre}[0]$
$\text{pre}[2] := \text{post}[1]$
$\text{post}[0] := \text{post}[2]$

leaf: $\text{post}[0] := \text{pre}[0] + 1$

Implementing Numbering of Leafs

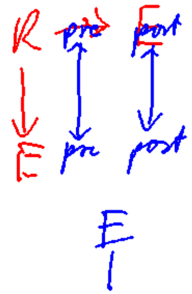
Idea:

- 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)



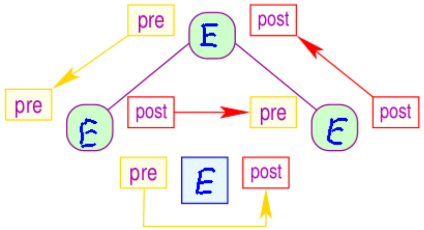
root: $\text{pre}[0] := 0$
$\text{pre}[1] := \text{pre}[0]$
$\text{post}[0] := \text{post}[1]$

node: $\text{pre}[1] := \text{pre}[0]$
$\text{pre}[2] := \text{post}[1]$
$\text{post}[0] := \text{post}[2]$



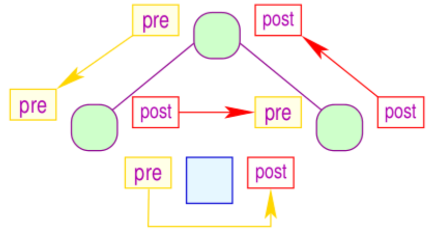
leaf: $\text{post}[0] := \text{pre}[0] + 1$
num := pre[0]

The Local Attribute Dependencies



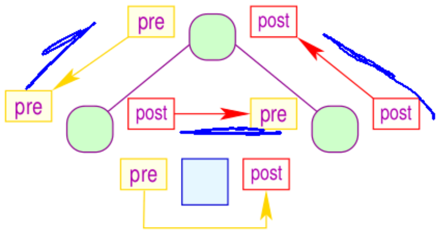
- the attribute system is apparently strongly acyclic

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)

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 $s ::= s_1 \dots s_n$ every inherited attribute of s_j where $1 \leq j \leq n$ only depends on

1. the attributes of $s_1, s_2, \dots s_{j-1}$ and
2. the inherited attributes of s.

L-attributed

Definition

An attribute system is L-attributed, if for all productions $s ::= s_1 \dots s_n$ every inherited attribute of s_j where $1 \leq j \leq n$ only depends on

1. the attributes of $s_1, s_2, \dots s_{j-1}$ 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

Definition

An attribute system is L-attributed, if for all productions $s ::= s_1 \dots s_n$ every inherited attribute of s_j where $1 \leq j \leq n$ only depends on

1. the attributes of $s_1, s_2, \dots s_{j-1}$ 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 $\mathcal{A} = A_1 \cup \dots \cup A_n$ 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

$\rightsquigarrow$ 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 $s ::= s_1 \dots s_n$ every inherited attribute of s_j where $1 \leq j \leq n$ only depends on

1. the attributes of $s_1, s_2, \dots s_{j-1}$ 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 $\mathcal{A} = A_1 \cup \dots \cup A_n$ 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

$\rightsquigarrow$ 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:

- class with a method for every non-terminal in the grammar

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

Regex ::= E | E
| E · E
| a

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

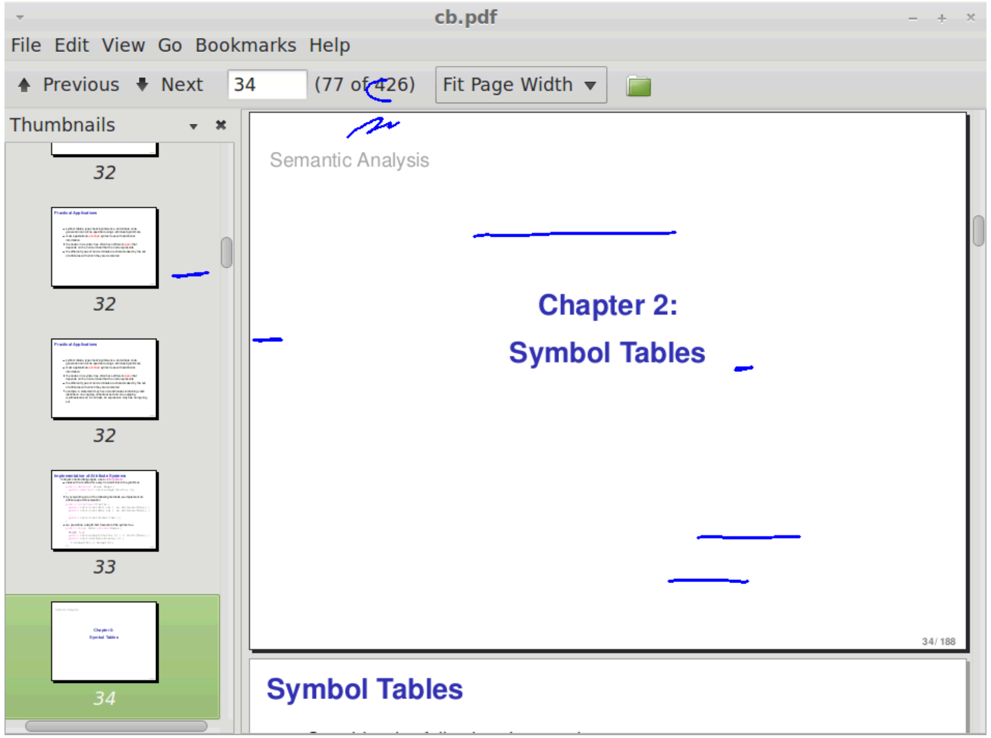
```
public interface Visitor {          re.l.accept(v); re.l.acc
  public void visit(~~Op~~ OrEx re) { ~~re.children(this)~~; }
  public void visit(Bar re) { re.children(this); }
  ...                    Letter
  public void visit(~~Token~~ tok) {}
}
```

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

```
public class OrEx extends Regex {
  Regex l,r;
  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

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

scope of **int** A

Scope of Identifiers

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

scope of **double** A

Visibility Rules in Object-Oriented Languages

```
public class Foo {
  int x = 17;
  protected int y =  5;
  private int z = 42;
  public int b() { return 1; }
}
class Bar extends Foo {
  protected double y = 0.5;
  public int b(int a)
    { return a+x; }
}
```

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

Observations:

Visibility Rules in Object-Oriented Languages

```
public class Foo {
  int x = 17;
  protected int y =  5;
  private int z = 42;
  public int b() { return 1; }
}
class Bar extends Foo {
  protected double y = 0.5;
  public int b(int a)
    { return a+x; }
}
```

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

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 ⇝ static overloading

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

2

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

Observations:

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

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

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 ⇝ ~~static overloading~~

Dynamic Resolution of Functions

```java
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();
  }
}
```

Observations:

Dynamic Resolution of Functions

```java
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();
  }
}
```

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

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
2. link each usage of a variable to the *declaration* of that variable
   - track data structures to distinguish declared variables and visible variables
   - for languages without explicit declarations, create declarations when a variable is first encountered

(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:
1. sequence of numbers
2. 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:

0	1	2	3	4	5	6	7
Peter	Piper	picked	a	peck	of	pickled	peppers

8	0	1	2	3	4	5	6	7
If	Peter	Piper	picked	a	peck	of	pickled	peppers

9	10	4	5	6	7	0	1	2
wheres	the	peck	of	pickled	peppers	Peter	Piper	picked

Output:

Example for Applying this Algorithm

Input:

Peter	Piper	picked	a	peck	of	pickled	peppers	
If	Peter	Piper	picked	a	peck	of	pickled	peppers
wheres	the	peck	of	pickled	peppers	Peter	Piper	picked

Output:

0

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	wheres
10	the

Implementing the Algorithm: Specification

Idea:

- implement a *partial map*: S : **String**$\rightarrow$**int**
- use a counter variable **int** count $= 0$; to track the number of different identifiers found so far

We thus define a function **int** getIndex(**String** w):

```
int getIndex(String w) {
    if (S(w) ≡ undefined) {
        S = S ⊕ {w ↦ count};
        return count++;
    else return S(w);
}
```


Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in$ **String** $\times$ **int** :
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ too expensive ✗

Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in$ **String** $\times$ **int** :
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ too expensive ✗
- balanced trees :
  $\mathcal{O}(\log(n))$
  $\mathcal{O}(\log(n))$ $\rightsquigarrow$ too expensive ✗

An Implementation using Hash Tables

- allocated an array M of sufficient size m
- choose a *hash function* $H : \textbf{String} \to [0, m-1]$ with the following properties:
  - $H(w)$ is cheap to compute
  - H distributes the occurring words equally over $[0, m-1]$

Possible choices ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

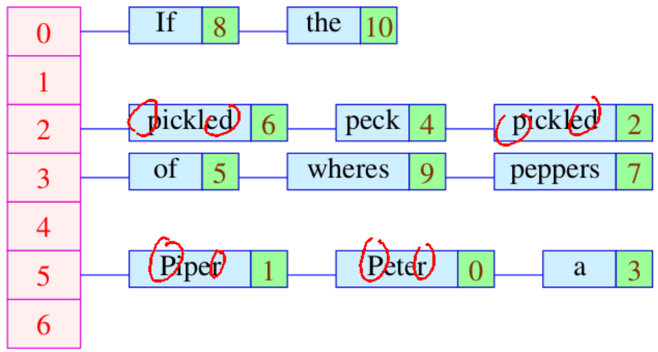
$$\begin{aligned} H_0(\vec{x}) &= (x_0 + x_{r-1}) \,\%\, m \\ H_1(\vec{x}) &= (\sum_{i=0}^{r-1} x_i \cdot p^i) \,\%\, m \\ H_2(\vec{x}) &= (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m \end{aligned}$$

for some prime number p (e.g. 31)

- We store the pair (w, i) in a linked list located at $M[H(w)]$

Computing a Hash Table for the Example

With $m = 7$ and H_0 we obtain:



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

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

Example: A Table of Stacks

V... 2 memory address

```
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

0	*a*	V
1	*b*	W
2	*c*	

0	*a*	V X
1	*b*	W
2	*c*	Y

0	*a*	V
1	*b*	W
2	*c*	

0	*a*	V
1	*b*	W
2	*c*	Z

Example: A Table of Stacks

```
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

0	*a*	*V*
1	*b*	*W*
2	*c*	

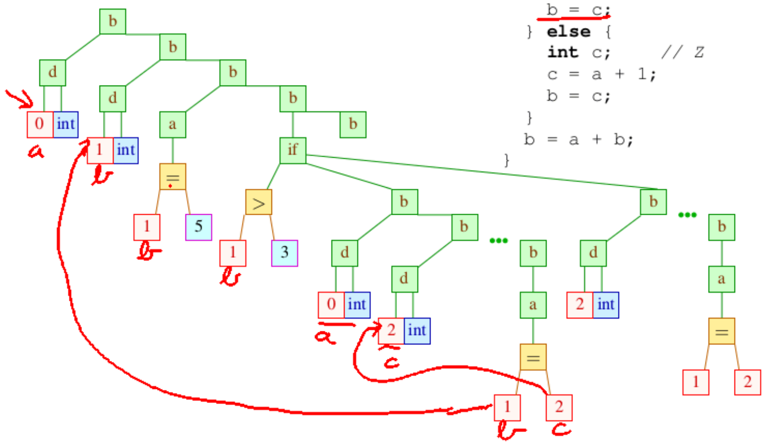
0	*a*	*X*, *V*
1	*b*	*W*
2	*c*	*Y*

0	*a*	*V*
1	*b*	*W*
2	*c*	*Z*

0	*a*	*V*
1	*b*	*W*
2	*c*	

Resolving: Rewriting the Syntax Tree

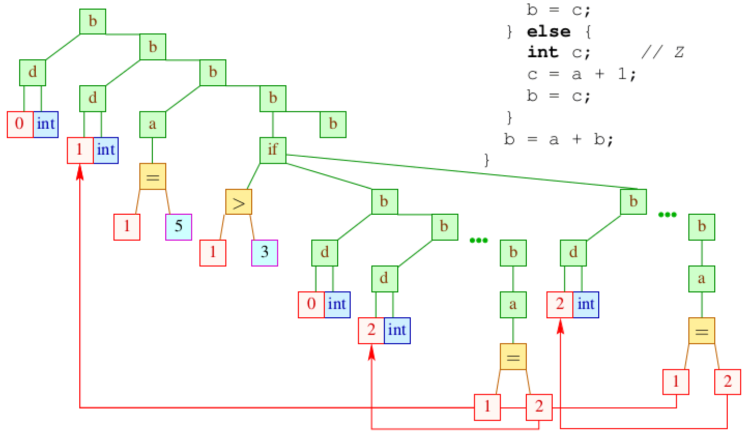
- **d** declaration node
- **b** basic block
- **a** assignment




```
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```


Resolving: Rewriting the Syntax Tree

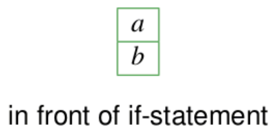
- **d** declaration node
- **b** basic block
- **a** assignment




```
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

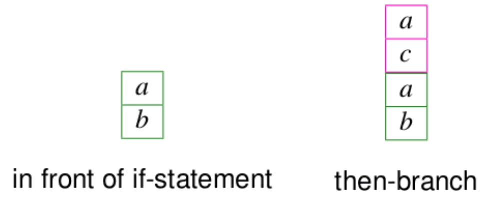
Alternative Resolution of Visibility

- 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



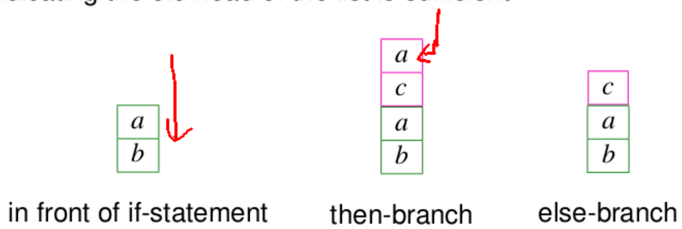
Alternative Resolution of Visibility

- 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



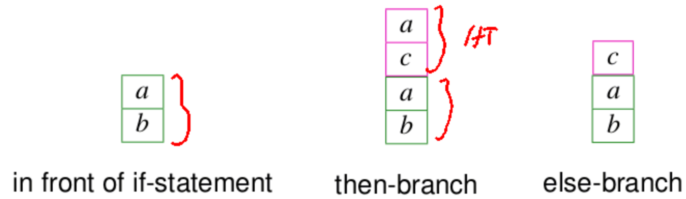
Alternative Resolution of Visibility

- 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



Alternative Resolution of Visibility

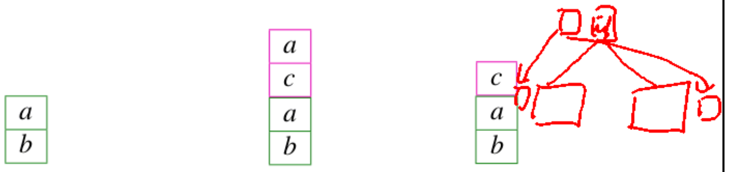
- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



- instead of lists of symbols, it is possible to use a list of hash tables $\rightsquigarrow$ 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



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

- instead of lists of symbols, it is possible to use a list of hash tables $\leadsto$ 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 t 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 t

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

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:

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

```
struct list1 {
  double info;
  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:

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

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

$\leadsto$ 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:

```
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:

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

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

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*.

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

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

((4==5)==7 true)

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:

```
infixr 8 ^
infixl 7 *,/
infixl 6 +,-
infix  4 ==,/=
```

The grammar is *generic*:

$$
\begin{array}{lcl}
Exp_0 & ::= & Exp_0\ LOp_0\ Exp_1 \\
 & | & Exp_1\ ROp_0\ Exp_0 \\
 & | & Exp_1\ Op_0\ Exp_1 \\
 & | & Exp_1 \\
 & & \vdots \\
Exp_9 & ::= & Exp_9\ LOp_9\ Exp \\
 & | & Exp\ ROp_9\ Exp_9 \\
 & | & Exp\ Op_9\ Exp \\
 & | & Exp \\
Exp & ::= & \textbf{ident} \mid \textbf{num} \\
 & | & \textbf{(}\ Exp_0\ \textbf{)}
\end{array}
$$

- parser enters an `infix` declaration into a table
- scanner checks table and produces:
  - operator `-` turns into token LOp_6.
  - operator `*` turns into token LOp_7.
  - operator `==` turns into token Op_4.
  - etc.
- ⇝ parser recognizes `3-4*5-6` as `(3-(4*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
  $\leadsto$ 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

Fixity-Declarations in Haskell: Observations

Troublesome changes:

- the scanner has a *state* which the parser determines
  $\leadsto$ 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 LOp_0 and transforms the AST afterwards.

Type Synonyms and Variables in C

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

∅

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

identifier

{point_t}

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	$\rightarrow$	`identifier` | ⋯

Type Synonyms and Variables in C

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

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

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	$\rightarrow$	`identifier` | ⋯

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:

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

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
	|	`void` | `char` | `char` $\cdots$ `typedef-name`
declarator	$\rightarrow$	`identifier` | $\cdots$

Problem:

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

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
	|	`void` | `char` | `char` $\cdots$ `typedef-name`
declarator	$\rightarrow$	`identifier` | $\cdots$

Solution is difficult:

- try to fix the look-ahead inside the parser
- add the following rule to the grammar:
  typedef-name $\rightarrow$ `identifier` $\rightsquigarrow \notin LR(1)$
- register type name earlier

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
	|	`void` | `char` | `char` $\cdots$ `typedef-name`
declarator	$\rightarrow$	`identifier` | $\cdots$

Solution is difficult:

- try to fix the look-ahead inside the parser
- add the following rule to the grammar:
  typedef-name $\rightarrow$ `identifier`
- register type name earlier
  - separate rule for `typedef` production

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
	|	`void` | `char` | `char` $\cdots$ `typedef-name`
declarator	$\rightarrow$	`identifier` | $\cdots$

Solution is difficult:

- try to fix ~~the look-ahead~~ inside the parser
- add the following rule to the grammar:
  typedef-name $\rightarrow$ `identifier`
- register type name earlier
  - separate rule for `typedef` production
  - call alternative declarator production that registers `identifier` as type name