b
UNIVERSITÄT
BERN

5. Semantic Analysis

Prof. O. Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

Roadmap

- Context-sensitive analysis
- Strategies for semantic analysis
- Attribute grammars
- Symbol tables and type-checking

See, *Modern compiler implementation in Java* (Second edition), chapter 5.

Roadmap

- **Context-sensitive analysis**
- Strategies for semantic analysis
- Attribute grammars
- Symbol tables and type-checking

Semantic Analysis

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines:

- interpret meaning of the program based on its syntactic structure
- two purposes:
  - finish analysis by deriving context-sensitive information
  - begin synthesis by generating the IR or target code
- associated with individual productions of a context free grammar or sub-trees of a syntax tree

Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is `x` scalar, an array, or a function?
2. Is `x` declared before it is used?
3. Are any names declared but not used?
4. Which declaration of `x` is being referenced?
5. Is an expression type-consistent?
6. Does the dimension of a reference match the declaration?
7. Where can `x` be stored? (heap, stack, ...)
8. Does `*p` reference the result of a `malloc()`?
9. Is `x` defined before it is used?
10. Is an array reference in bounds?
11. Does function `foo` produce a constant value?
12. Can `p` be implemented as a memo-function?

These questions cannot be answered with a context-free grammar

Context-sensitive analysis

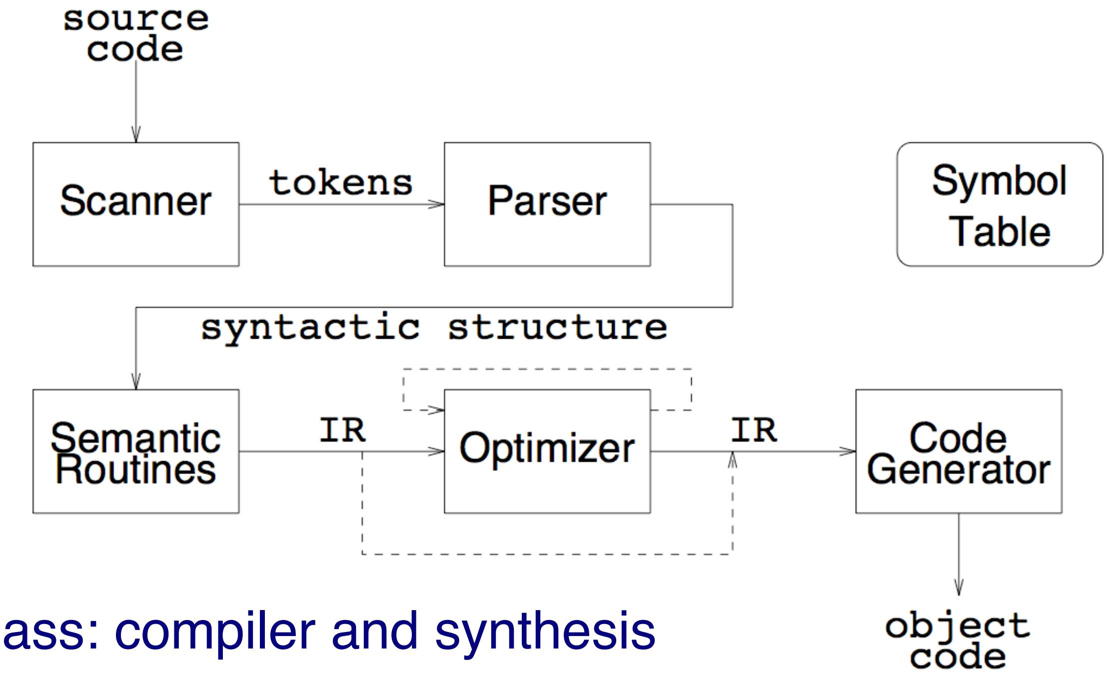
- Why is context-sensitive analysis hard?
  - answers depend on values, not syntax
  - questions and answers involve non-local information
  - answers may involve computation

- Several alternatives:
  - *symbol tables:* central store for facts; express checking code
  - *abstract syntax tree (attribute grammars):* specify non-local computations; automatic evaluators
  - *language design:* simplify language; avoid problems

Roadmap

- Context-sensitive analysis
- **Strategies for semantic analysis**
- Attribute grammars
- Symbol tables and type-checking

Alternatives for semantic processing



- one-pass: compiler and synthesis
- two-pass: compiler + peephole
- two-pass: compiler & IR synthesis + code generation pass
- multi-pass analysis
- multi-pass synthesis
- language-independent and re-targetable compilers

One-pass compilers

- interleave scanning, parsing and translation
  - no explicit IR
  - generate target code directly
    - emit short sequences of instructions on each parser action
    - little or no optimization possible (minimal context)

- can add *peephole optimization pass*
  - extra pass over generated code through small window ("peephole") of instructions
  - smoothes out "rough edges" between code emitted by subsequent calls to code generator

Two-pass: analysis & IR synthesis + code generation

- Generate explicit IR as interface to code generator
  - linear (e.g., tuples)
  - can emit multiple tuples at a time for better code context
- Advantages
  - easier retargeting (IR must be expressive enough for different machines!)
  - can add optimization pass later (multi-pass synthesis)

Multi-pass analysis

- Several passes, read/write intermediate files
  1. scan source file, generate tokens
     - place identifiers and constants in symbol table
  2. parse token file
     - generate semantic actions or linearized parse tree
  3. process declarations to symbol table
  4. semantic checking with IR synthesis

- Motivations:
  - Historical: constrained address spaces
  - Language: *e.g.*, declaration after use
  - Multiple analyses over IR tree

Multi-pass synthesis

- Passes operate on linear or tree-structured IR
- Options:
  - code generation and peephole optimization
  - multi-pass IR transformation
    - machine-independent then dependent optimizations
  - high-level to low-level IR transformation before code generation
    - e.g., in gcc high-level trees drive generation of low-level Register Transfer Language for machine-independent optimization
  - language-independent front ends
  - retargetable back ends

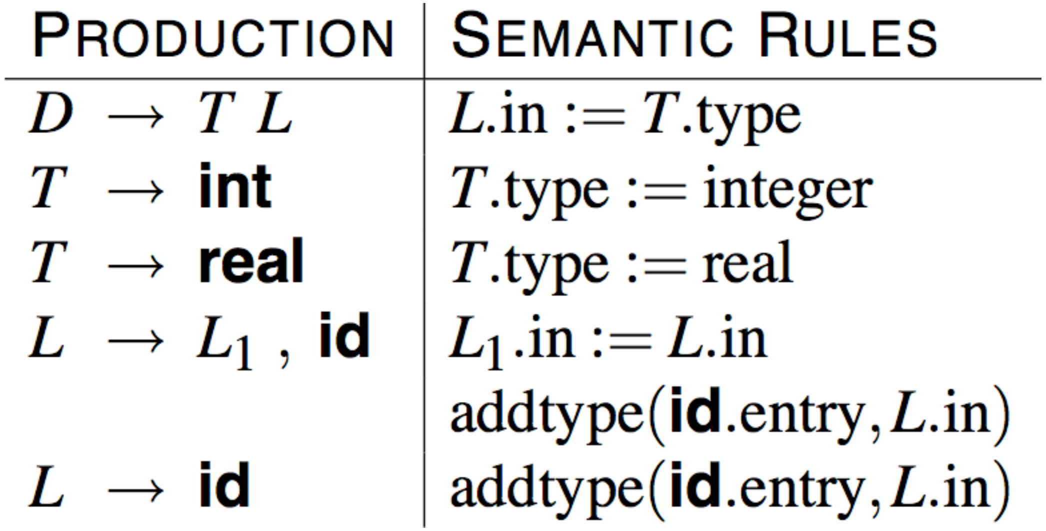
Roadmap

- Context-sensitive analysis
- Strategies for semantic analysis
- **Attribute grammars**
- Symbol tables and type-checking

Attribute grammars

- Add attributes to the syntax tree:
  - can add attributes (fields) to each node
  - specify equations to define values
  - propagate values up (synthesis) or down (inheritance)
- **Example:** ensuring that constants are immutable
  - add *type* and *class* attributes to expression nodes
  - add rules to production for `:=`
    1. check that LHS.*class* is variable (not constant)
    2. check that LHS.*type* and RHS.*type* are compatible

Attribute grammar actions



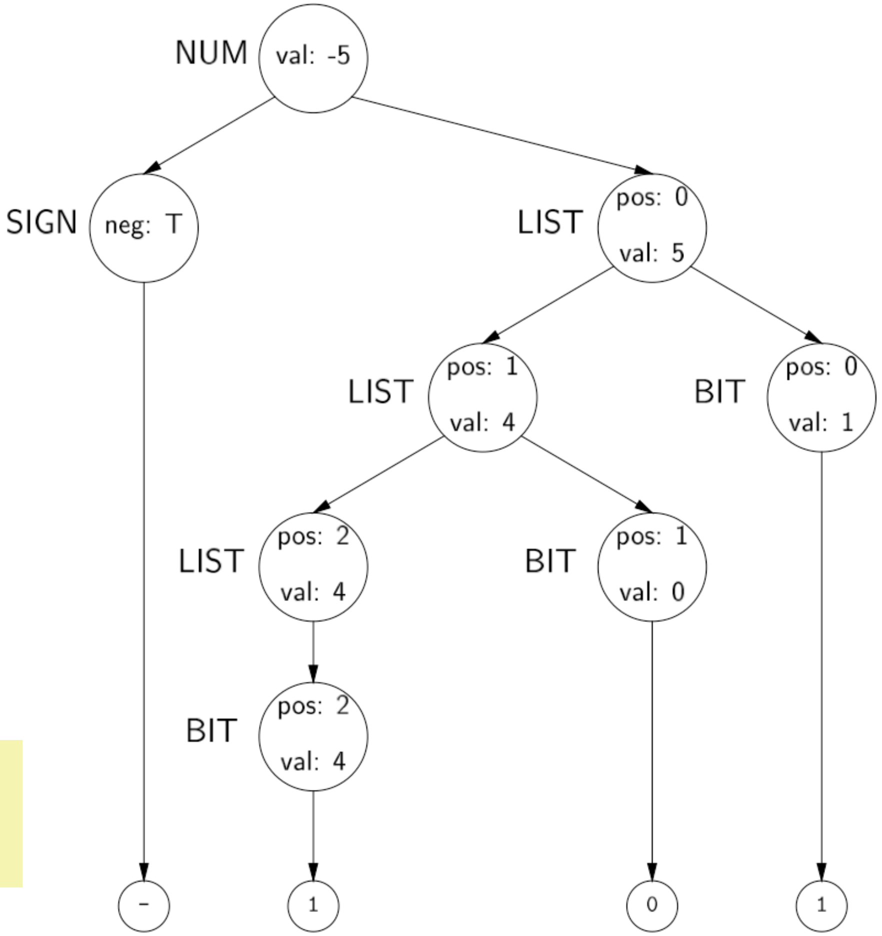
PRODUCTION	SEMANTIC RULES
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := integer
$T \rightarrow$ **real**	T.type := real
$L \rightarrow L_1$, **id**	L_1.in := L.in
	addtype(**id**.entry, L.in)
$L \rightarrow$ **id**	addtype(**id**.entry, L.in)

- tree attributes specified by grammar
- productions associated with attribute assignments
- each attribute defined uniquely and locally
- identical terms are labeled uniquely

Example: evaluate signed binary numbers

PRODUCTION	SEMANTIC RULES
NUM → SIGN LIST	LIST.pos := 0 if SIGN.neg NUM.val := -LIST.val else NUM.val := LIST.val
SIGN → +	SIGN.neg := false
SIGN → -	SIGN.neg := true
LIST → BIT	BIT.pos := LIST.pos LIST.val := BIT.val
LIST → LIST_1 BIT	LIST_1.pos := LIST.pos + 1 BIT.pos := LIST.pos LIST.val := LIST_1.val + BIT.val
BIT → 0	BIT.val := 0
BIT → 1	BIT.val := $2^{\text{BIT}.pos}$

Attributed parse tree for -101



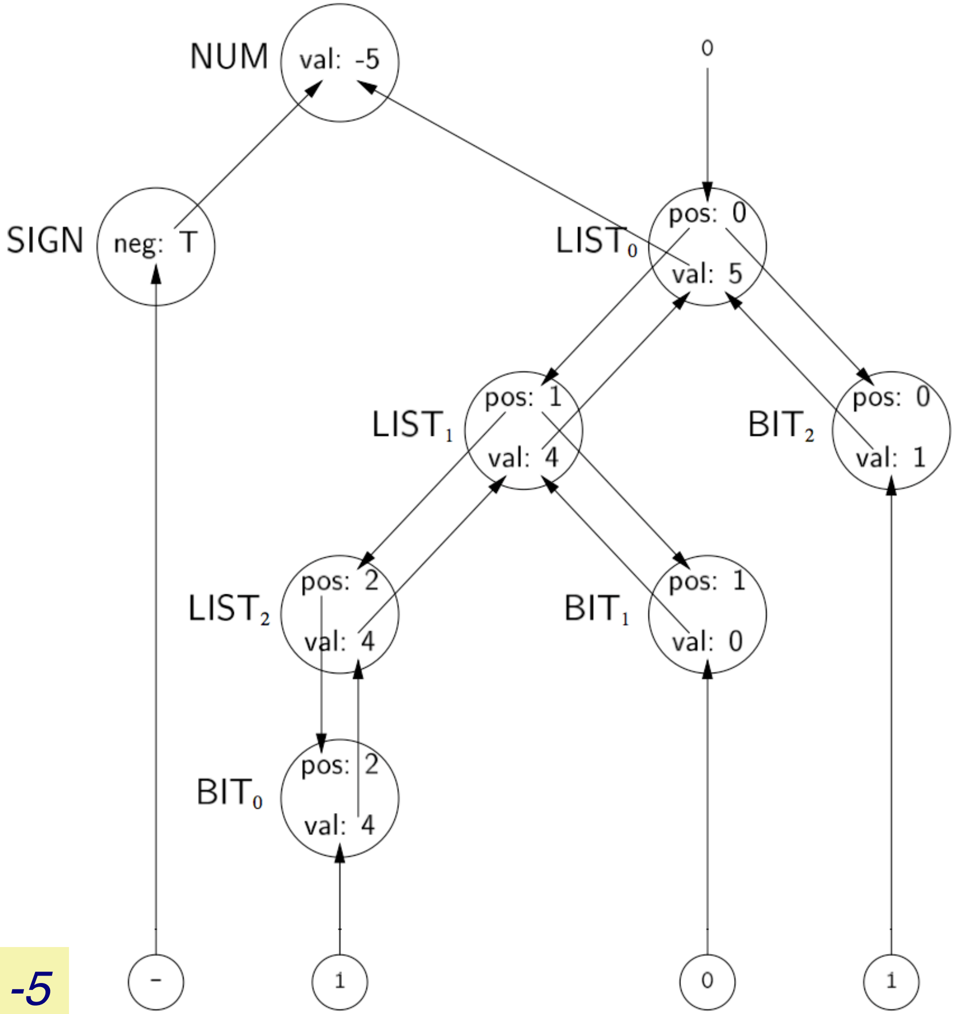
- *val* and *neg* are *synthetic* attributes
- *pos* is an *inherited* attribute

Attribute dependency graph

- *nodes* represent *attributes*
- *edges* represent *flow of values*
- graph must be *acyclic*
- *topologically sort* to order attributes
  - use this order to evaluate rules
  - order depends on both grammar and input string!

1. SIGN.neg
2. LIST_0.pos
3. LIST_1.pos
4. LIST_2.pos
5. BIT_0.pos
6. BIT_1.pos
7. BIT_2.pos
8. BIT_0.val
9. LIST_2.val
10. BIT_1.val
11. LIST_1.val
12. BIT_2.val
13. LIST_0.val
14. NUM.val

Evaluating in this order yields NUM.val = -5



Evaluation strategies

- ***Parse-tree methods***
  1. build the parse tree
  2. build the dependency graph
  3. topologically sort the graph
  4. evaluate it
- ***Rule-based methods***
  1. analyse semantic rules at compiler-construction time
  2. determine static ordering for each production’s attributes
  3. evaluate its attributes in that order at compile time
- ***Oblivious methods***
  1. ignore the parse tree and the grammar
  2. choose a convenient order (e.g., left-to-right traversal) and use it
  3. repeat traversal until no more attribute values can be generated

Attribute grammars in practice

- *Advantages*
  - clean formalism
  - automatic generation of evaluator
  - high-level specification
- *Disadvantages*
  - evaluation strategy determines efficiency
  - increase space requirements
  - parse tree evaluators need dependency graph
  - results distributed over tree
  - circularity testing

Historically, attribute grammars have been judged too large and expensive for industrial-strength compilers.

Roadmap

- Context-sensitive analysis
- Strategies for semantic analysis
- Attribute grammars
- **Symbol tables and type-checking**

Symbol tables

- For compile-time efficiency, compilers often use a *symbol table*:
  - associates lexical *names* (symbols) with their *attributes*

- What items should be entered?
  - variable names
  - defined constants
  - procedure and function names
  - literal constants and strings
  - source text labels
  - compiler-generated temporaries (we'll get there)

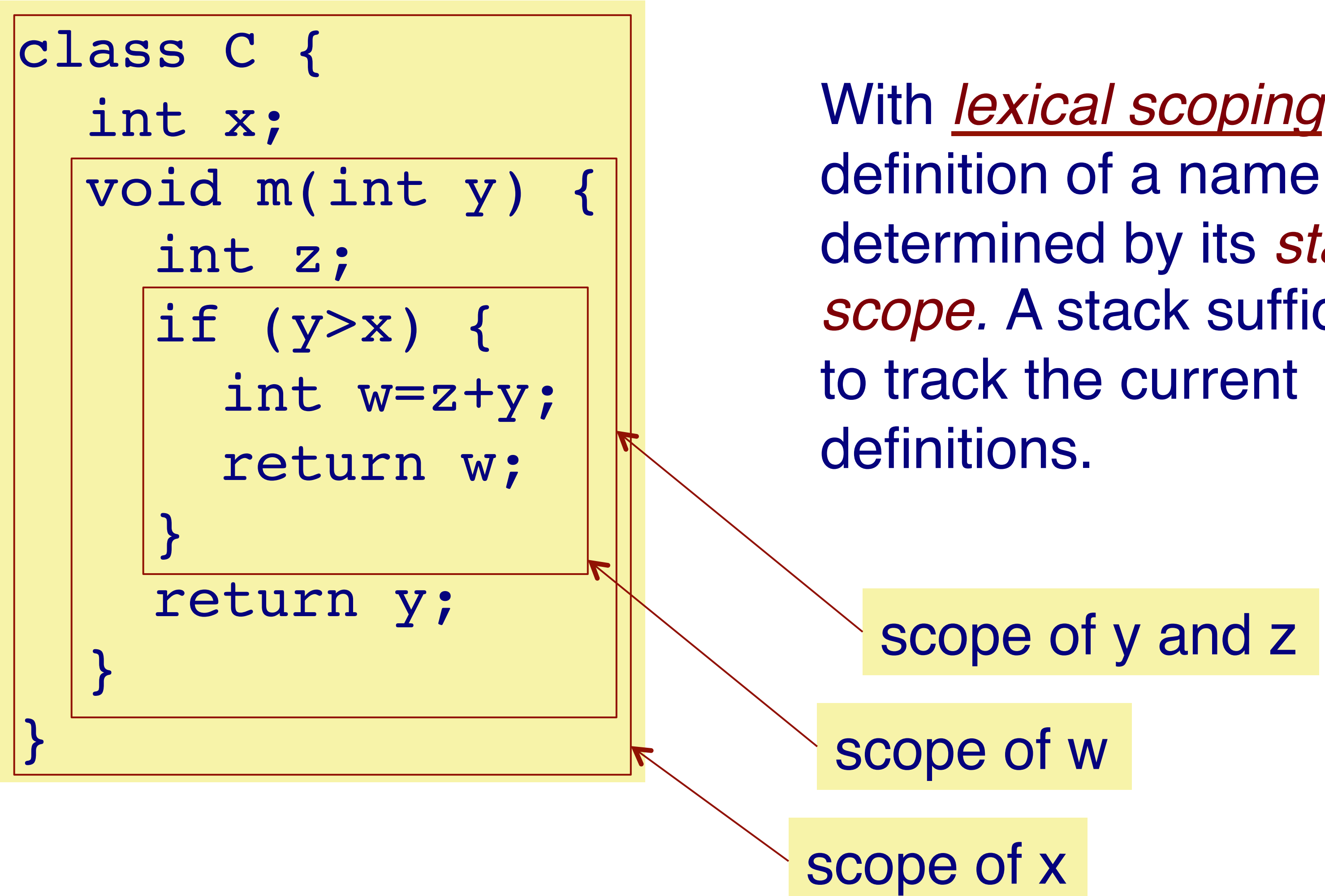
- Separate table of structure layouts for types (field offsets and lengths)

A symbol table is a compile-time structure

Symbol table information

- What kind of information might the compiler need?
  - textual name
  - data type
  - dimension information *(for aggregates)*
  - declaring procedure
  - lexical level of declaration
  - storage class *(base address)*
  - offset in storage
  - if record, pointer to structure table
  - if parameter, by-reference or by-value?
  - can it be aliased? to what other names?
  - number and type of arguments to functions

Lexical Scoping



```
class C {
   int x;
   void m(int y) {
      int z;
      if (y>x) {
         int w=z+y;
         return w;
      }
      return y;
   }
}
```

With *lexical scoping* the definition of a name is determined by its *static scope.* A stack suffices to track the current definitions.

scope of y and z

scope of w

scope of x

Nested scopes: block-structured symbol tables

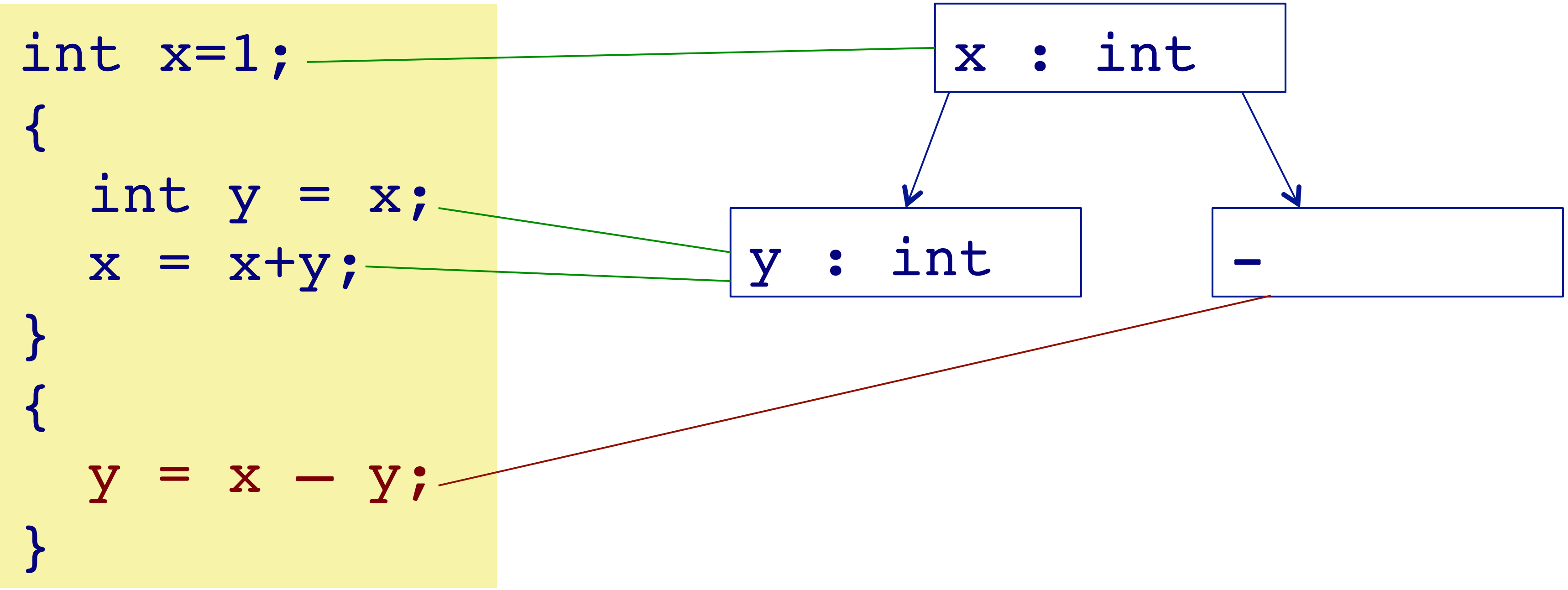
- What information is needed?
  - when we ask about a name, we want the *most recent declaration*
  - the declaration may be from the current scope or some enclosing scope
  - innermost scope overrides declarations from outer scopes

- Key point: *new declarations (usually) occur only in current scope*

- What operations do we need?
  - `void put(Symbol key, Object value)` — bind key to value
  - `Object get(Symbol key)` — return value bound to key
  - `void beginScope()` — remember current state of table
  - `void endScope()` — restore table to state at most recent scope that has not been ended

May need to preserve list of locals for the debugger

Checking variable declarations in a hierarchical symbol table



```
int x=1;
{
   int y = x;
   x = x+y;
}
{
   y = x – y;
}
```

x : int

y : int

–

Attribute information

- Attributes are internal representation of declarations
- Symbol table associates names with attributes

- Names may have different attributes depending on their meaning:
  - *variables:* type, procedure level, frame offset
  - *types:* type descriptor, data size/alignment
  - *constants:* type, value
  - *procedures:* formals (names/types), result type, block information (local decls.), frame size

Static and Dynamic Typing

A language is statically typed if it is always possible to *determine the (static) type* of an expression *based on the program text alone.*

A language is dynamically typed if *only values have fixed type*. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

A language is "strongly typed" if it is impossible to perform an operation on the wrong kind of object.

Type consistency may be assured by

I. compile-time type-checking,
II. type inference, or
III. dynamic type-checking.

See: Programming Languages course

Type expressions

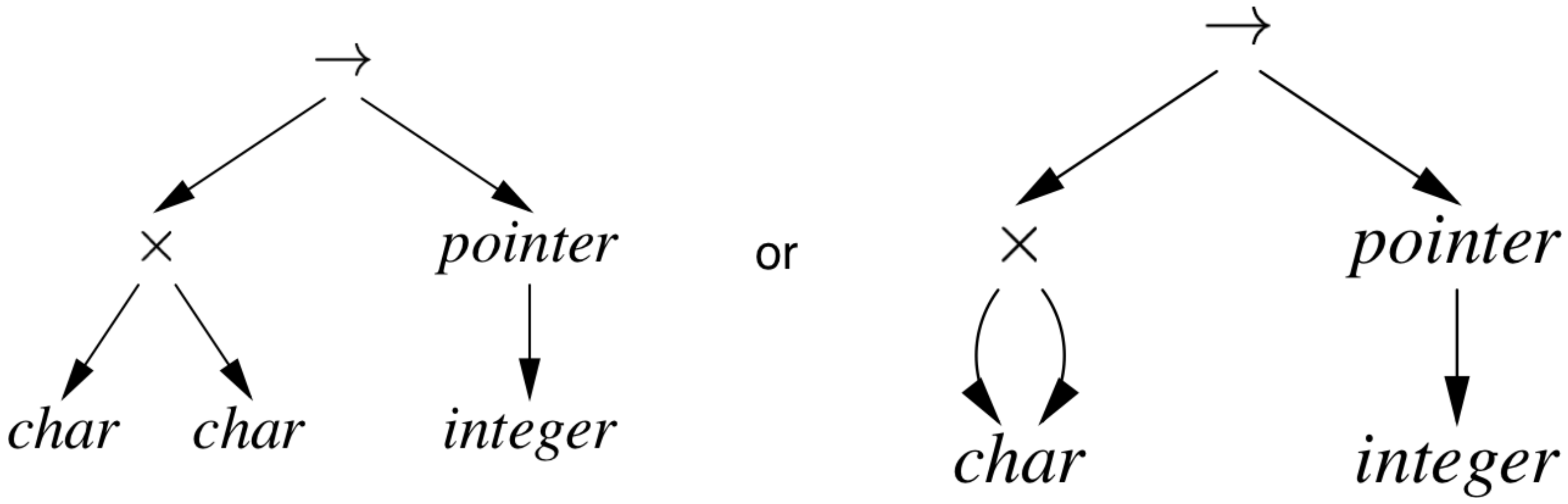
Type expressions are a textual representation for types:

1. basic types: *boolean*, *char*, *integer*, *real*, etc.
2. type names
3. constructed types (constructors applied to type expressions):
    a) array(I,T) denotes *array* of elements type T, index type I
    e.g., array (1...10,integer)
    b) $T_1 \times T_2$ denotes *Cartesian product* of type expressions T_1 and T_2
    c) record(…) denotes *record* with named fields
    e.g., record((a × integer), (b × real))
    d) pointer(T) denotes the type "*pointer* to object of type T"
    e) D → R denotes type of *function* mapping domain D to range R
    e.g., integer × integer → integer

Type descriptors

Type descriptors are compile-time structures representing type expressions

e.g., *char × char → pointer(integer)*



Type compatibility

Type checking needs to determine *type equivalence*

Two approaches:

- *Name equivalence:* each type name is a distinct type
- *Structural equivalence:* two types are equivalent iff they have the same structure (after substituting type expressions for type names)
  - $s \equiv t$ iff s and t are the same basic types
  - $\text{array}(s_1,s_2) \equiv \text{array}(t_1,t_2)$ iff $s_1 \equiv t_1$ and $s_2 \equiv t_2$
  - $s_1 \times s_2 \equiv t_1 \times t_2$ iff $s_1 \equiv t_1$ and $s_2 \equiv t_2$
  - $\text{pointer}(s) \equiv \text{pointer}(t)$ iff $s \equiv t$
  - $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff $s_1 \equiv t_1$ and $s_2 \equiv t_2$

Type compatibility: example

Consider:

```
type link = ^cell
var next : link;
var last : link;
var p : ^cell;
var q, r : ^cell;
```

Under name equivalence:

- `next` and `last` have the same type
- `p`, `q` and `r` have the same type
- `p` and `next` have different type

Under structural equivalence all variables have the same type

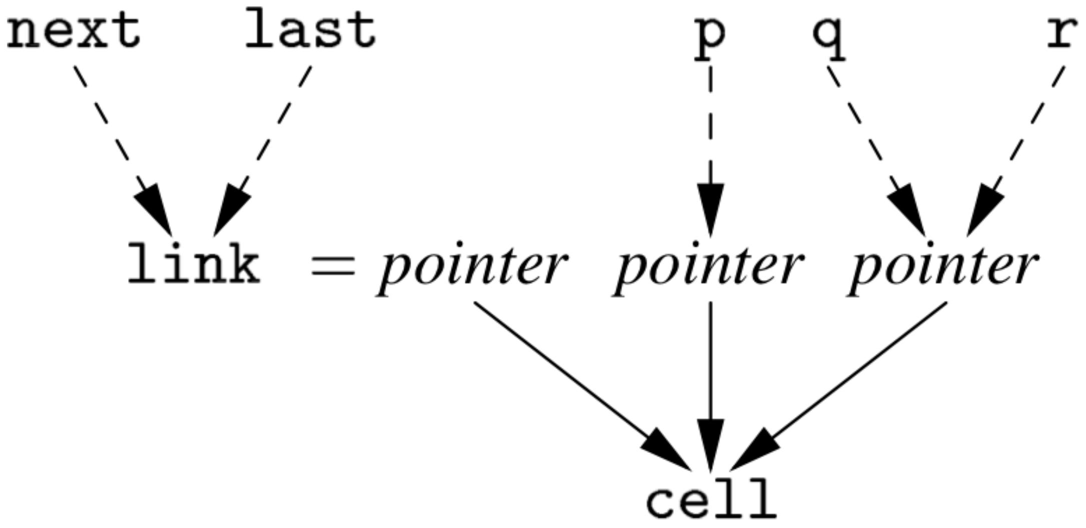
Ada/Pascal/Modula-2 are somewhat confusing: they treat distinct type definitions as distinct types, so

- `p` has different type from `q` and `r` (!)

Type compatibility: Pascal-style name equivalence

Build compile-time structure called a *type graph:*

- each constructor or basic type creates a node
- each name creates a leaf (associated with the type's descriptor)



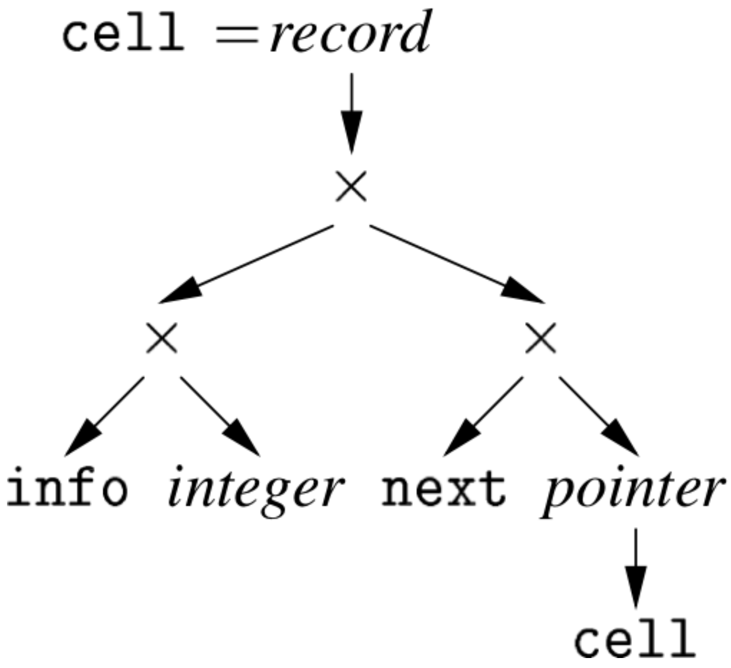
Type expressions are equivalent if they are represented by the same node in the graph

Type compatibility: recursive types

Consider:

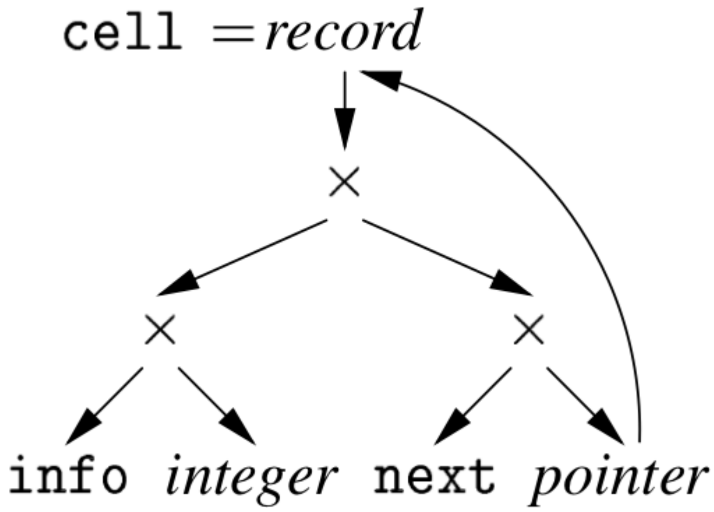
```
type link = ^cell
var cell = record
              info : integer;
              next : link;
           end
```

Expanding `link` in the type graph yields:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:



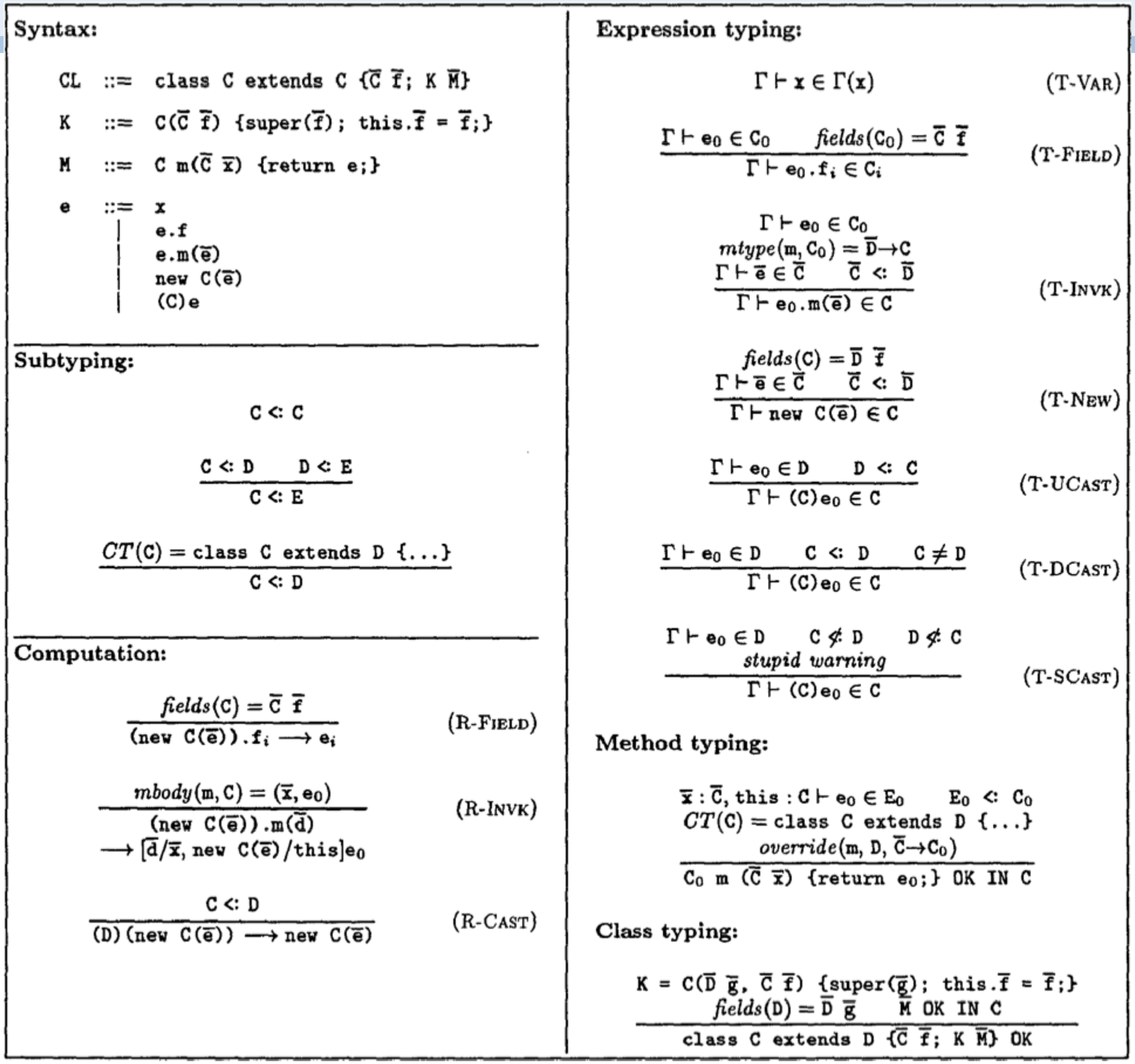
Type rules

Type-checking rules can be formalized to prove soundness and correctness.

$$\frac{f : A \rightarrow B,\ x : A}{f(x) : B}$$

If f is a function from A to B, and x is of type A, then f(x) is a value of type B.

Example: Featherweight Java



Syntax:

$\mathtt{CL} ::= \mathtt{class\ C\ extends\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}}$

$\mathtt{K} ::= \mathtt{C(\overline{C}\ \overline{f})\ \{super(\overline{f});\ this.\overline{f} = \overline{f};\}}$

$\mathtt{M} ::= \mathtt{C\ m(\overline{C}\ \overline{x})\ \{return\ e;\}}$

$$\begin{array}{lll} \mathtt{e} & ::= & \mathtt{x} \\ & | & \mathtt{e.f} \\ & | & \mathtt{e.m(\overline{e})} \\ & | & \mathtt{new\ C(\overline{e})} \\ & | & \mathtt{(C)e} \end{array}$$

Subtyping:

$$\mathtt{C} <: \mathtt{C}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}}}{\mathtt{C} <: \mathtt{D}}$$

Computation:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{(new\ C(\overline{e})).f}_i \longrightarrow \mathtt{e}_i} \qquad (\textsc{R-Field})$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\begin{array}{c}\mathtt{(new\ C(\overline{e})).m(\overline{d})} \\ \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C(\overline{e})/this}]\mathtt{e}_0\end{array}} \qquad (\textsc{R-Invk})$$

$$\frac{\mathtt{C} <: \mathtt{D}}{\mathtt{(D)(new\ C(\overline{e}))} \longrightarrow \mathtt{new\ C(\overline{e})}} \qquad (\textsc{R-Cast})$$

Expression typing:

$$\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad (\textsc{T-Var})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{C}_i} \qquad (\textsc{T-Field})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{e}_0\mathtt{.m(\overline{e})} \in \mathtt{C}} \qquad (\textsc{T-Invk})$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new\ C(\overline{e})} \in \mathtt{C}} \qquad (\textsc{T-New})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash \mathtt{(C)e}_0 \in \mathtt{C}} \qquad (\textsc{T-UCast})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash \mathtt{(C)e}_0 \in \mathtt{C}} \qquad (\textsc{T-DCast})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash \mathtt{(C)e}_0 \in \mathtt{C}} \qquad (\textsc{T-SCast})$$

Method typing:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}} \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)\end{array}}{\mathtt{C}_0\ \mathtt{m\ (\overline{C}\ \overline{x})\ \{return\ e}_0\mathtt{;\}\ OK\ IN\ C}}$$

Class typing:

$$\frac{\begin{array}{c}\mathtt{K = C(\overline{D}\ \overline{g},\ \overline{C}\ \overline{f})\ \{super(\overline{g});\ this.\overline{f} = \overline{f};\}} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\end{array}}{\mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}\ OK}}$$

Used to prove that generics could be added to Java without breaking the type system.

Igarashi, Pierce and Wadler, *"Featherweight Java: a minimal core calculus for Java and GJ", OOPSLA '99*
doi.acm.org/10.1145/320384.320395

What you should know!

- ✎ *Why is semantic analysis mostly context-sensitive?*
- ✎ *What is “peephole optimization”?*
- ✎ *Why was multi-pass semantic analysis introduced?*
- ✎ *What is an attribute grammar? How can it be used to support semantic analysis?*
- ✎ *What kind of information is stored in a symbol table?*
- ✎ *How is type-checking performed?*

Can you answer these questions?

- ✎ *Why can semantic analysis be performed by the parser?*
- ✎ *What are the pros and cons of introducing an IR?*
- ✎ *Why must an attribute dependency graph be acyclic?*
- ✎ *Why would be the use of a symbol table at run-time?*
- ✎ *Why does Java adopt nominal (name-based) rather than structural type rules?*

License

http://creativecommons.org/licenses/by-sa/3.0/