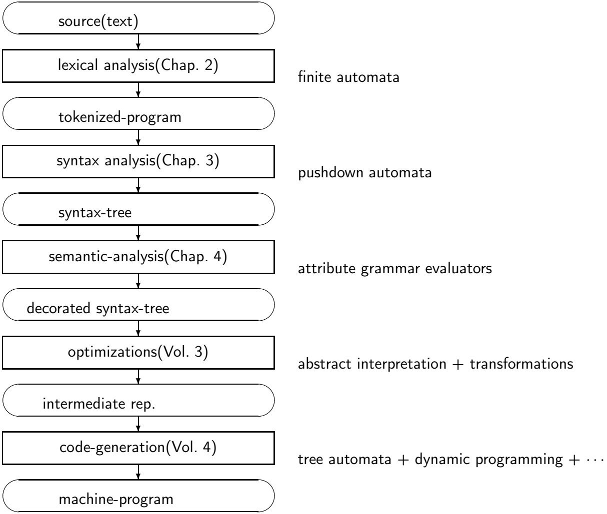
Semantic Analysis
Wilhelm/Seidl/Hack: Compiler Design – Syntactic and Semantic Analysis, Chapter 4

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

"Standard" Structure



When is a Program Incorrect?

A program is incorrect, if it does not adhere to language-specific constraints.

- Scanner: Catches sequence of characters that do not form valid tokens
  Example: ïnτ instead of `int`
  Specification mechanism: regular expressions (cannot describe matching parentheses)
- Parser: Catches sequence of symbols that do not form valid words in a CFG
  Example: `while while int 2 do`
  Specification mechanism: CFGs (cannot describe declaredness requirements)
- This leaves context sensitive constraints
  Example: `int f(){return 4;} void g(){return f(6);}`

Semantic Constraints

Typical semantic constraints, checked by the compiler:

- Each variable declared in an enclosing scope.
- Variables uniquely declared within a scope.
- Types of operands and operators in expressions must match.

Programs violating such semantic constraints are rejected by the compiler.

Note: Dynamic semantic constraints (no division by zero, no dereferencing of null pointers) are not (cannot) be checked by the compiler, the potential can, cf. static program analysis (Vol. 3)!

Types and Variables: Terminology

- Identifiers denote program objects (variables, constants, types, methods,. . .).
- A declaration introduces an identifier, binds it to an element.
- A defining occurence of an identifier is an occurence in a declaration.
- An applied occurence of an identifier is an occurence somewhere else.
- The scope of a defining occurence is that (textual) part of a program, in which an applied occurence may refer to this defining occurrence.
- A defining occurence of an identifier is visible, if it is directly visible (in the scope) or made visible by name extensions (`std::cin`)

Types and Variables: Terminology (continued)

- The type of a constant (variable) constrains which operations can be applied to the constant (variable).
- Overload of an identifier is the legal existence of several defining occurrences of this identifier in the same scope.

Symbol Table

- A data structure used to store information on declared objects
- Supports insertions and deletions of declarations and opening and closing of scopes
- Supports efficient search for the defining occurrence associated with an applied occurrence: identification of identifiers

Symbol Table Functionality

Language with nested scopes, (blocks

create_symb_table	creates an empty symbol table,
enter_block	notes the start of a new scope,
exit_block	resets the symbol table to the state before the last *enter_block*,
enter_id(*id*, *decl_ptr*)	inserts an entry for identifier *id* with a link to its defining occurrence passed in *decl_ptr*,
search_id(id)	searches the def. occ. for *id* returns a pointer to it if exists.

Symbol Table Implementation

- Data structure with constant time for *search_id*,
- all currently valid defining occurrences of an identifier are stored in a (stack like) linear list,
- new entry is inserted at the end of this list,
- the end of this list is pointed to by an array component for this identifier,
- all entries for a block are chained through a linear list.

```
proc create_symb_table;
  begin create empty stack of block entries end ;
proc enter_block;
  begin push entry for the new block end ;
proc exit_block;
  begin
    foreach decl. entry of the curr. block do
      delete entry
    od;
    pop block entry from stack
  end ;

proc enter_id ( id: Idno; decl: ↑node );
  begin
    if exists entry for id in curr. block
    then error("double declaration")
    fi;
    create new entry with decl and no. of curr. block;
    insert entry at tail of linear list for id;
    insert entry at tail of linear list for curr. block
  end ;
```

```
function  search_id ( id: idno ) ↑node;
  begin
    if list for id is empty
    then  error("undeclared identifier")
    else  return (value of decl-field of first elem. in id-list)
    fi
  end
```

Example Program with Symboltable

1

Decl. of *a*, *b*

proc *p* (* forward decl. *)

proc *q*

proc *p* **is**

2

Decl. of *a*, *c*

Decl. of *c*, *d*

proc *q* **is**

3

Decl. of *a*, *d*

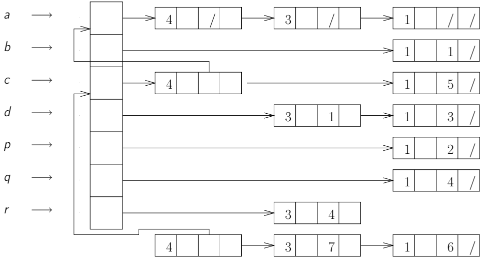
proc *r*

proc *r* **is**

4

Decl. of *a*, *c*

*



Declaration Analysis

```
proc analyze_decl (k : node);
  proc analyze_subtrees (root: node);
  begin
     for  i := 1 to #descs(root) do   (* # children *)
          analyze_decl(root.i)        (* i-th child of root *)
     od
  end ;
begin
  case symb(k) of              (* label of k *)
  block:  begin
              enter_block;
              analyze_subtrees(k);
              exit_block
          end ;
  decl:   begin
              analyze_subtrees(k);
              foreach identifier declared here id do
                  enter_id(id, ↑ k)
              od
          end ;
  appl_id:(* appl. occ. of identifier id *)
          store search_id(id) at k;
  otherwise:    if k no leaf then analyze_subtrees(k) fi
  od
end
```

Overloading of Operators

- An operator symbol (function, procedure identifier) is overloaded, if it may denote several operations at some point in the program.
- The different operators need to have different parameter profiles, i.e., tuples of argument and result types.
- The identification of identifiers may have legally associated several possible parameter profiles with an applied occurrence.
- Overload Resolution needs to identify exactly one defining occurrence depending on its parameter profile.

Overload Resolution I

Overload resolution for Ada:

- Conceptually 4 passes over trees for assignments.
- Passes 1 (initialization) and 2 (bottom-up elimination) and passes 3 (top-down elimination) and 4 (check) can be merged.

```
begin
    init_ops;
    bottom_up_elim(root);
    top_down_elim(root);
    check whether now all ops sets have exactly one element;
        otherwise report an error
end
```

Overload Resolution II

Functions applied at nodes of assignment trees:

$\#descs(k)$	number of child nodes of k,
$symb(k)$	symbol labeling k,
$vis(k)$	set of definitions of $symb$(k) visible at k
$ops(k)$	set of actual candidates for overloaded symbol $symb(k)$,
$k.i$	ith child of k.

For def. occ. of overloaded symbol op with type $t_1 \times \cdots \times t_m \to t$

$$\begin{array}{lcl} rank(op) & = & m \\ res\_typ(op) & = & t \\ par\_typ(op, i) & = & t_i \quad (1 \leq i \leq m). \end{array}$$

Overload Resolution III

```
proc  resolve_overloading (root: node, a_priori_type: type);

func pot_res_types (k: node): set of type;
               (* potential types of the result *)
    return {res_typ(op) | op ∈ ops(k)}

func act_par_types (k: node, i: integer): set of type;
    return {par_typ(op, i) | op ∈ ops(k)}

proc  init_ops
begin
    foreach   k
        ops(k) := {op | op ∈ vis(k) and  rank(op) = #descs(k)}
    od;
    ops(root) := {op ∈ ops(root) |  res_typ(op) = a_priori_typ}
end ;
```

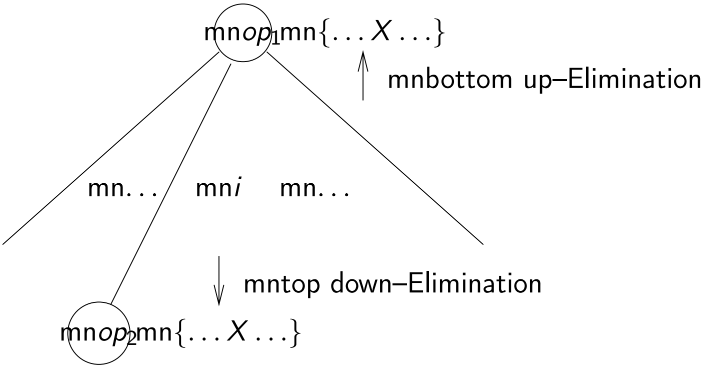
Overload Resolution IV

proc bottom_up_elim (k: node);
begin
  for $i := 1$ **to** #descs(k) **do**
    bottom_up_elim ($k.i$);
    $ops(k) := ops(k) - \{op \in ops(k) \mid$ par_typ$(op, i) \notin$ pot_res_ty
    (* remove the operators, whose ith parameter type does not
    match the potential result types of the ith operand *)
  od;
end ;

Overload Resolution V

```
proc top_down_elim (k: node);
begin
    for i := 1 to #descs(k) do
        ops(k.i) := ops(k.i) − {op ∈ ops(k.i) | res_typ(op) ∉ act_par_t
        (∗ remove the operators, whose result type does not match
        any type of the corresponding parameter ∗)
        top_down_elim(k.i)
    od;
end ;
```

Overload Resolution VI



Quite typical information flow, up and down the parse tree!