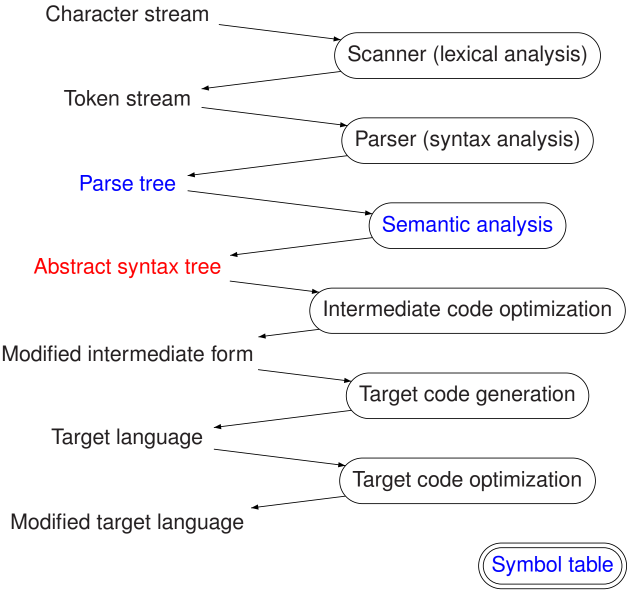
CS 403: Semantic Analysis

Gregory Mierzwinski

Fall 2022

The Compilation Process



Syntax Directed Translation

- Syntax-directed translation $\rightarrow$ the source language translation is completely driven by the parser
    - The parsing process and parse trees/AST used to direct semantic analysis and the translation of the source program
    - Separate phase of a compiler or grammar augmented with information to control the semantic analysis and translation (attribute grammars)
- Attribute grammars $\rightarrow$ associate attributes with each grammar symbol
    - An attribute has a name and an associated value: string, number, type, memory location, register — whatever information we need.
    - Examples
        - Attributes for a variable include type (as declared, useful later in type-checking)
        - An integer constant will have an attribute value (used later to generate code)
- With each grammar rule we also give semantic rules or actions, describing how to compute the attribute values associated with each grammar symbol in the rule
    - An attribute value for a parse node may depend on information from its children nodes, its siblings, and its parent

Grammar			Action(s)
$\langle\text{digit}\rangle$	$::=$	0	$\{\langle\text{digit}\rangle.\mathit{value} = 0;\}$
	$\mid$	1	$\{\langle\text{digit}\rangle.\mathit{value} = 1;\}$
	$\mid$	2	$\{\langle\text{digit}\rangle.\mathit{value} = 2;\}$
	$\ldots$		
	$\mid$	9	$\{\langle\text{digit}\rangle.\mathit{value} = 9;\}$
$\langle\text{int}\rangle$	$::=$	$\langle\text{digit}\rangle$	$\{\langle\text{int}\rangle_0.\mathit{value} = \langle\text{digit}\rangle.\mathit{value};\}$
	$\mid$	$\langle\text{int}\rangle\langle\text{digit}\rangle$	$\{\langle\text{int}\rangle_0.\mathit{value} = \langle\text{int}\rangle_1.\mathit{value} * 10 + \langle\text{digit}\rangle.\mathit{value};\}$

- Attributes are computed during the construction of the parse tree and are typically included in the node objects of that tree
- Two general classes of attributes:
  - Synthesized: passed up in the parse tree
  - Inherited: passed down the parse tree

Attribute Grammars and Actions (cont'd)

- Synthesized attributes: the left-side attribute is computed from the right-side attributes.

$$
\begin{array}{rcl}
X & ::= & Y_1\, Y_2 \ldots Y_n \\
X.a & = & f(Y_1.a, Y_2.a, \ldots, Y_n.a)
\end{array}
$$

  - The lexical analyzer supplies the attributes of terminals
  - The attributes for nonterminals are built up and passed up the tree

$\langle\text{int}\rangle$
$\mathit{value} = 4 * 10 + 2 = 42$

$\langle\text{int}\rangle$ $\mathit{value} = 4$ — $\langle\text{digit}\rangle$ $\mathit{value} = 2$

$\langle\text{digit}\rangle$ $\mathit{value} = 4$ — 2

4

- Inherited attributes: the right-side attributes are derived from the left-side attributes or other right-side attributes

$$
\begin{array}{rcl}
X & ::= & Y_1\, Y_2 \ldots Y_n \\
Y_k.a & = & f(X.a, Y_1.a, Y_2.a, \ldots, Y_{k-1}.a, Y_{k+1}.a, \ldots, Y_n.a)
\end{array}
$$

  - Used for passing information about the context to nodes further down the tree

$$
\begin{array}{llll}
\langle \mathrm{P} \rangle & ::= & \langle \mathrm{D} \rangle \langle \mathrm{S} \rangle & \{\langle \mathrm{S} \rangle.\mathit{dl} = \langle \mathrm{D} \rangle.\mathit{dl};\} \\
\langle \mathrm{D} \rangle & ::= & \mathit{var}\ \langle \mathrm{V} \rangle\ ;\ \langle \mathrm{D} \rangle & \{\langle \mathrm{D} \rangle_0.\mathit{dl} = \mathrm{addList}(\langle \mathrm{V} \rangle.\mathit{name}, \langle \mathrm{D} \rangle_1.\mathit{dl});\} \\
 & \mid & \varepsilon & \{\langle \mathrm{D} \rangle_0.\mathit{dl} = \mathrm{NULL};\} \\
\langle \mathrm{S} \rangle & ::= & \langle \mathrm{V} \rangle\ := \langle \mathrm{E} \rangle\ ;\ \langle \mathrm{S} \rangle & \{\mathrm{check}(\langle \mathrm{V} \rangle.\mathit{name}, \langle \mathrm{S} \rangle_0.\mathit{dl}); \langle \mathrm{S} \rangle_1.\mathit{dl} = \langle \mathrm{S} \rangle_0.\mathit{dl};\} \\
 & \mid & \varepsilon & \{\} \\
\langle \mathrm{V} \rangle & ::= & x & \{\langle \mathrm{V} \rangle.\mathit{name} = "x";\} \\
 & \mid & y & \{\langle \mathrm{V} \rangle.\mathit{name} = "y";\} \\
 & \mid & z & \{\langle \mathrm{V} \rangle.\mathit{name} = "z";\}
\end{array}
$$

Inherited Attributes (cont'd)

$$
\begin{array}{llll}
\langle \mathrm{P} \rangle & ::= & \langle \mathrm{D} \rangle \langle \mathrm{S} \rangle & \{ \langle \mathrm{S} \rangle.\mathit{dl} = \langle \mathrm{D} \rangle.\mathit{dl}; \} \\
\langle \mathrm{D} \rangle & ::= & \mathit{var}\ \langle \mathrm{V} \rangle\ ;\ \langle \mathrm{D} \rangle & \{ \langle \mathrm{D} \rangle_0.\mathit{dl} = \mathrm{addList}(\langle \mathrm{V} \rangle.\mathit{name}, \langle \mathrm{D} \rangle_1.\mathit{dl}); \} \\
 & | & \varepsilon & \{ \langle \mathrm{D} \rangle_0.\mathit{dl} = \mathrm{NULL}; \} \\
\langle \mathrm{S} \rangle & ::= & \langle \mathrm{V} \rangle\ := \langle \mathrm{E} \rangle\ ;\ \langle \mathrm{S} \rangle & \{ \mathrm{check}(\langle \mathrm{V} \rangle.\mathit{name}, \langle \mathrm{S} \rangle_0.\mathit{dl}); \langle \mathrm{S} \rangle_1.\mathit{dl} = \langle \mathrm{S} \rangle_0.\mathit{dl}; \} \\
 & | & \varepsilon & \{\} \\
\langle \mathrm{V} \rangle & ::= & x & \{ \langle \mathrm{V} \rangle.\mathit{name} = "x"; \} \\
 & | & y & \{ \langle \mathrm{V} \rangle.\mathit{name} = "y"; \} \\
 & | & z & \{ \langle \mathrm{V} \rangle.\mathit{name} = "z"; \}
\end{array}
$$

- Two attributes: *name* for the name of the variable and *dl* for the list of declarations
- Each time a new variable is declared a synthesized attribute for its name is attached to it
- That name is added to a list of variables declared so far in the synthesized attribute *dl* created from the declaration block

Inherited Attributes (cont'd)

$$
\begin{array}{llll}
\langle \mathrm{P} \rangle & ::= & \langle \mathrm{D} \rangle \langle \mathrm{S} \rangle & \{\langle \mathrm{S} \rangle.dl = \langle \mathrm{D} \rangle.dl;\} \\
\langle \mathrm{D} \rangle & ::= & var\ \langle \mathrm{V} \rangle\ ;\ \langle \mathrm{D} \rangle & \{\langle \mathrm{D} \rangle_0.dl = \mathrm{addList}(\langle \mathrm{V} \rangle.name, \langle \mathrm{D} \rangle_1.dl);\} \\
 & | & \varepsilon & \{\langle \mathrm{D} \rangle_0.dl = \mathrm{NULL};\} \\
\langle \mathrm{S} \rangle & ::= & \langle \mathrm{V} \rangle\ := \langle \mathrm{E} \rangle\ ;\ \langle \mathrm{S} \rangle & \{\mathrm{check}(\langle \mathrm{V} \rangle.name, \langle \mathrm{S} \rangle_0.dl); \langle \mathrm{S} \rangle_1.dl = \langle \mathrm{S} \rangle_0.dl;\} \\
 & | & \varepsilon & \{\} \\
\langle \mathrm{V} \rangle & ::= & x & \{\langle \mathrm{V} \rangle.name = "x";\} \\
 & | & y & \{\langle \mathrm{V} \rangle.name = "y";\} \\
 & | & z & \{\langle \mathrm{V} \rangle.name = "z";\}
\end{array}
$$

- Two attributes: *name* for the name of the variable and *dl* for the list of declarations
- Each time a new variable is declared a synthesized attribute for its name is attached to it
- That name is added to a list of variables declared so far in the synthesized attribute *dl* created from the declaration block
- The list of variables is then passed as an inherited attribute to the statements following the declarations so that it can be checked that variables are declared before use

- Typically handling of attributes: associate with each symbol some sort of structure (e.g., list) with all the necessary attributes
- Then have such a list as a member variable in each node structure
- Insert code in each nonterminal function to carry on the attribute computations
- Also need some convention for referring to individual symbols in a rule while defining the associated action
  - Typical convention in compiler generators: `$$` to refer to the left hand side and `$i` to refer to the `i`-th component of the right hand side:

```
P -> DS          { $2.list = $1.list; }
D -> var V; D    { $$.list = add_to_list($2.name, $4.list); }
   |             { $$.list = NULL; }
S -> V := E; S   { check($1.name, $$.list); $5.list = $$.list; }
   |
V -> x           { $$.name = "x"; }
   | y           { $$.name = "y"; }
   | z           { $$.name = "z"; }
```

- Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination – now we move to check whether they form a sensible set of instructions in the programming language $\rightarrow$ semantic analysis
  - Any noun phrase followed by some verb phrase makes a syntactically correct English sentence, but a semantically correct one
    - has subject-verb agreement
    - has proper use of gender
    - the components go together to express a sensible idea
- For a program to be semantically valid:
  - all variables, functions, classes, etc. must be properly defined
  - expressions and variables must be used in ways that respect the type system
  - access control must be respected
  - etc.
- Note however that a valid program is not necessariy correct

```
int Fibonacci(int n) {
    if (n <= 1) return 0;
    return Fibonacci(n - 1) + Fibonacci(n - 2); }
int main() { Print(Fibonacci(40)); }
```

SEMANTIC ANALYSIS

- Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination – now we move to check whether they form a sensible set of instructions in the programming language $\rightarrow$ semantic analysis
  - Any noun phrase followed by some verb phrase makes a syntactically correct English sentence, but a semantically correct one
    - has subject-verb agreement
    - has proper use of gender
    - the components go together to express a sensible idea
- For a program to be semantically valid:
  - all variables, functions, classes, etc. must be properly defined
  - expressions and variables must be used in ways that respect the type system
  - access control must be respected
  - etc.
- Note however that a valid program is not necessariy correct

```
int Fibonacci(int n) {
    if (n <= 1) return 0;
    return Fibonacci(n - 1) + Fibonacci(n - 2); }
int main() { Print(Fibonacci(40)); }
```

  - Valid but not correct!

CHALLENGES IN SEMANTIC ANALYSIS

- Reject the largest number of incorrect programs
- Accept the largest number of correct programs

- Reject the largest number of incorrect programs
- Accept the largest number of correct programs
- Do so quickly!

http://xkcd.com/303/

- Some semantic analysis done during parsing (syntax directed translation)
  - Some languages specifically designed for exclusive syntax directed translation (one-pass compilers)
  - Other languages require repeat traversals of the AST after parsing
- Sample components of semantic analysis: type and scope checking

- A type is a set of values and a set of operations operating on those values
- Three categories of types in most programming languages:
  - Base types (int, float, double, char, bool, etc.) → primitive types provided directly by the underlying hardware
  - Compound types (enums, arrays, structs, classes, etc.) → types are constructed as aggregations of the base types
  - Complex types (lists, stacks, queues, trees, heaps, tables, etc) → abstract data types, may or may not exist in a language
- In many languages the programmer must first establish the name, type, and lifetime of a data object (variable, function, etc.) through declarations

```
double calculate(int a, double b); // function declaration (prototype)
int x = 0;                         // global variables
double y;                          // (throughout the program)
int main() {
    int m[3];                      // local variables
    char *n;                       // (available only in main())
    ...
}
```

Type Checking

- The bulk of semantic analysis = the process of verifying that each operation respects the type system of the language
  - Generally means that all operands in any expression are of appropriate types and number
  - Sometimes the rules are defined by other parts of the code (e.g., function prototypes), and sometimes such rules are a part of the language itself (e.g., "both operands of a binary arithmetic operation must be of the same type")
- Type checking can be done during compilation, execution, or across both
  - A language is considered strongly typed if each and every type error is detected during compilation
  - Static type checking is done at compile time
    - The information needed is obtained (e.g., from declarations) and stored in a symbol table
    - The types involved in each operation are then checked
    - It is very difficult for a language that only does static type checking to meet the full definition of strongly typed (particularly dangerous: casting)
  - Dynamic type checking is implemented by including type information for each data location at runtime
    - For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type"
    - The execution of any operation begins by first checking these type tags and is performed only if everything checks out

Type Checking Variants

- Static type checking done in most programming languages
- Dynamic type checking is done in e.g., LISP, Perl
- Many languages have built-in functionality for correcting the simplest of type errors (implicit type conversion), but others are very strict (Ada, Pascal, Haskell, etc.)
  - Implicit conversions can be handy but may also hide serious errors
  - Classical example in PL/1: declare `A`, `B`, `C` as 3-character arrays, initialize two and add them together

    ```
    DECLARE (A, B, C) CHAR(3);
    B = "123"; C = "456"; A = B + C;
    ```

Type Checking Variants

- Static type checking done in most programming languages
- Dynamic type checking is done in e.g., LISP, Perl
- Many languages have built-in functionality for correcting the simplest of type errors (implicit type conversion), but others are very strict (Ada, Pascal, Haskell, etc.)
  - Implicit conversions can be handy but may also hide serious errors
  - Classical example in PL/1: declare `A`, `B`, `C` as 3-character arrays, initialize two and add them together

    ```
    DECLARE (A, B, C) CHAR(3);
    B = "123"; C = "456"; A = B + C;
    ```

    - The result of `B+C` will be `579` (inplicit conversion to numbers)!

Type Checking Variants

- Static type checking done in most programming languages
- Dynamic type checking is done in e.g., LISP, Perl
- Many languages have built-in functionality for correcting the simplest of type errors (implicit type conversion), but others are very strict (Ada, Pascal, Haskell, etc.)
  - Implicit conversions can be handy but may also hide serious errors
  - Classical example in PL/1: declare `A`, `B`, `C` as 3-character arrays, initialize two and add them together

    ```
    DECLARE (A, B, C) CHAR(3);
    B = "123"; C = "456"; A = B + C;
    ```

    - The result of `B+C` will be `579` (inplicit conversion to numbers)!
    - Can we assign numbers to strings? Sure, why not! The default width for such a conversion in PL/1 is 8
    - So the conversion of `579` back to string will result in `"␣␣␣␣␣579"`

Type Checking Variants

- Static type checking done in most programming languages
- Dynamic type checking is done in e.g., LISP, Perl
- Many languages have built-in functionality for correcting the simplest of type errors (implicit type conversion), but others are very strict (Ada, Pascal, Haskell, etc.)
  - Implicit conversions can be handy but may also hide serious errors
  - Classical example in PL/1: declare `A`, `B`, `C` as 3-character arrays, initialize two and add them together

    ```
    DECLARE (A, B, C) CHAR(3);
    B = "123"; C = "456"; A = B + C;
    ```

    - The result of `B+C` will be `579` (inplicit conversion to numbers)!
    - Can we assign numbers to strings? Sure, why not! The default width for such a conversion in PL/1 is 8
    - So the conversion of `579` back to string will result in `"␣␣␣␣␣579"`
    - Still, the size of `A` is only 3, so the string gets truncated implicitly
    - Thus the resulting value stored in `A` is the counterintuitive `"␣␣␣"`

Type Checking Variants

- Static type checking done in most programming languages
- Dynamic type checking is done in e.g., LISP, Perl
- Many languages have built-in functionality for correcting the simplest of type errors (implicit type conversion), but others are very strict (Ada, Pascal, Haskell, etc.)
  - Implicit conversions can be handy but may also hide serious errors
  - Classical example in PL/1: declare `A`, `B`, `C` as 3-character arrays, initialize two and add them together

    ```
    DECLARE (A, B, C) CHAR(3);
    B = "123"; C = "456"; A = B + C;
    ```

    - The result of `B+C` will be `579` (inplicit conversion to numbers)!
    - Can we assign numbers to strings? Sure, why not! The default width for such a conversion in PL/1 is 8
    - So the conversion of `579` back to string will result in `"␣␣␣␣␣579"`
    - Still, the size of `A` is only 3, so the string gets truncated implicitly
    - Thus the resulting value stored in `A` is the counterintuitive `"␣␣␣"`
- Most type systems rely on declarations
  - Notable exceptions: functional languages that do not require declarations but work hard to infer the data types of variables from the code

TYPE CHECKER DESIGN

- Design process defining a type system:
  1. Identify the types that are available in the language
  2. Identify the language constructs that have types associated with them
  3. Identify the semantic rules for the language
- C++-like language example (declarations required = somewhat strongly typed)
  - Base types (int, double, bool, string) + compound types (arrays, classes)
    - Arrays can be made of any type (including other arrays)
    - ADTs can be constructed using classes (no need to handle them separately)
  - Type-related language constructs:
    - Constants: type given by the lexical analysis
    - Variables: all variables must have a declared type (base or compound)
    - Functions: precise type signature (arguments + return)
    - Expressions: each expression has a type based on the type of the composing constant, variable, return type of the function, or type of operands
    - Other constructs (if, while, assignment, etc.) also have associate types (since they have expressions inside)
  - Semantic rules govern what types are allowable in the various language constructs
    - Rules specific to individual constructs: operand to a unary minus must either be double or int, expression used in a loop test must be of bool type, etc.
    - General rules: all variables must be declared, all classes are global, etc.

- First step: record type information with each identifier
  - The lexical analyzer gives the name
  - The parser needs to connect that name with the type (based on declaration)
  - This information is stored in a symbol table
  - Example declaration: `int a; double b;`
  - When building the node for ⟨var⟩ the parser can associate the type (`int`) with the variable (`a`) and create a suitable entry in the symbol table
  - Typically the symbol table is stored outside the parse tree
  - The `class` or `struct` entry in a symbol table is a table in itself (recording all fields and their types)

⟨decl⟩	::=	⟨var⟩; ⟨decl⟩
⟨var⟩	::=	⟨type⟩ ⟨identifier⟩
⟨type⟩	::=	int
	|	bool
	|	double
	|	string
	|	⟨identifier⟩
	|	⟨type⟩[ ]

Type Checking Implementation

- First step: record type information with each identifier
  - The lexical analyzer gives the name
  - The parser needs to connect that name with the type (based on declaration)
  - This information is stored in a symbol table
  - Example declaration: `int a; double b;`
  - When building the node for ⟨var⟩ the parser can associate the type (`int`) with the variable (`a`) and create a suitable entry in the symbol table
  - Typically the symbol table is stored outside the parse tree
  - The `class` or `struct` entry in a symbol table is a table in itself (recording all fields and their types)

$$
\begin{array}{rcl}
\langle\text{decl}\rangle & ::= & \langle\text{var}\rangle;\ \langle\text{decl}\rangle \\
\langle\text{var}\rangle & ::= & \langle\text{type}\rangle\ \langle\text{identifier}\rangle \\
\langle\text{type}\rangle & ::= & \text{int} \\
 & | & \text{bool} \\
 & | & \text{double} \\
 & | & \text{string} \\
 & | & \langle\text{identifier}\rangle \\
 & | & \langle\text{type}\rangle[\,]
\end{array}
$$

- Second step: verify language constructs for type consistency
  - Can be done while parsing (in such a case declarations must precede use)
  - Can also be done in a subsequent parse tree traversal (more flexible on the placement of declarations)

Type Checking Implementation (cont'd)

- Second step: verify language constructs for type consistency, continued
  1. Verification based on the rules of the grammar
     - While examining an ⟨expr⟩ + ⟨expr⟩ node the types of the two ⟨expr⟩ must agree with each other and be suitable for addition
     - While examining a ⟨id⟩ = ⟨expr⟩ the type of ⟨expr⟩ (determined recursively) must agree with the type of ⟨id⟩ (retrieved from the symbol table)

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{const}\rangle \\
 & | & \langle\text{id}\rangle \\
 & | & \langle\text{expr}\rangle + \langle\text{expr}\rangle \\
 & | & \langle\text{expr}\rangle / \langle\text{expr}\rangle \\
 & \ldots & \\
\langle\text{stmt}\rangle & ::= & \langle\text{id}\rangle = \langle\text{expr}\rangle \\
 & \ldots &
\end{array}
$$

  2. Verification based on the general type rules of the language
     Examples:
     - The index in an array selection must be of integer type
     - The two operands to logical `&&` must both have `bool` type; the result is `bool` type
     - The type of each actual argument in a function call must be compatible with the type of the respective formal argument

Type Checking Implementation (cont'd)

- Second step: verify language constructs for type consistency, continued
  1. Verification based on the rules of the grammar
     - While examining an $\langle\text{expr}\rangle + \langle\text{expr}\rangle$ node the types of the two $\langle\text{expr}\rangle$ must agree with each other and be suitable for addition
     - While examining a $\langle\text{id}\rangle = \langle\text{expr}\rangle$ the type of $\langle\text{expr}\rangle$ (determined recursively) must agree with the type of $\langle\text{id}\rangle$ (retrieved from the symbol table)

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{const}\rangle \\
 & | & \langle\text{id}\rangle \\
 & | & \langle\text{expr}\rangle + \langle\text{expr}\rangle \\
 & | & \langle\text{expr}\rangle / \langle\text{expr}\rangle \\
 & \ldots & \\
\langle\text{stmt}\rangle & ::= & \langle\text{id}\rangle = \langle\text{expr}\rangle \\
 & \ldots &
\end{array}
$$

  2. Verification based on the general type rules of the language
     Examples:
     - The index in an array selection must be of integer type
     - The two operands to logical `&&` must both have `bool` type; the result is `bool` type
     - The type of each actual argument in a function call must be compatible with the type of the respective formal argument
- Most semantic checking deals with types, but generally the semantic analysis must enforce all the rules in the language (type-related or not)
  - Examples: identifiers are not re-used within the same scope, `break` only appears inside a loop, etc.

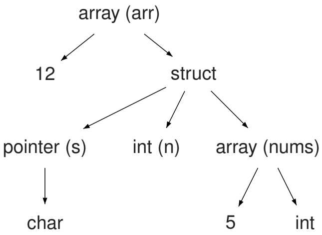
- The major attributes of an identifier are:
  - Name – identify language entities
  - Type – determines range of values and set of operations
  - Value – for storable quantities (r-values)
  - Location (address) – places where values are stored (l-values)
- The meaning of names is determined by its attributes
  - `const n = 5;` $\rightarrow$ associates to name `n` the attributes const and value 5
  - `var x:integer;` $\rightarrow$ associates attributes var and type integer to name `x`
  - The declaration

    ```
    function square_root(the_integer:  integer) :real;
        begin ...  end
    ```

    associates to the name `square_root`:
    - the attribute function
    - the names and types of its parameters
    - the type of the return value
    - the body of code to be executed when the function is called

- The equivalence of base types is easy to establish (`int` is only equivalent to `int`, `bool` is only compatible with `bool`, etc.)
- Common technique for compound types: store compound types as a tree structure

```
struct {
    char *s;
    int n;
    int nums[5];
} arr [12];
```



- Then the comparison will be done recursively based on the tree structure (very much like Prolog's unification)

Equivalence of Compound Types (cont'd)

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
    if (tree1 == tree2) // if same type pointer, must be equivalent!
        return true;
    if (tree1->type != tree2->type) // check types first
        return false;
    switch (tree1->type) {
        case T_INT: case T_DOUBLE: ...  // same base type
            return true;
        case T_PTR:
            return AreEquivalent(tree1->child[0], tree2->child[0]);
        case T_ARRAY:
            return AreEquivalent(tree1->child[0], tree2->child[0]) &&
                   AreEquivalent(tree1->child[1], tree2->child[1]);
        ...
    }
}
```

Equivalence of Compound Types (cont'd)

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
    if (tree1 == tree2) // if same type pointer, must be equivalent!
        return true;
    if (tree1->type != tree2->type) // check types first
        return false;
    switch (tree1->type) {
        case T_INT: case T_DOUBLE: ...  // same base type
            return true;
        case T_PTR:
            return AreEquivalent(tree1->child[0], tree2->child[0]);
        case T_ARRAY:
            return AreEquivalent(tree1->child[0], tree2->child[0]) &&
                   AreEquivalent(tree1->child[1], tree2->child[1]);
        ...
    }
}
```

- Also needs some way to deal with circular types, such as marking the visited nodes so that we do not compare them ever again

USER-DEFINED TYPES

- When are two custom types equivalent?
  - Named equivalence: when the two names are identical
    - Equivalence assessed by name only (just like base types)
  - Structural equivalence: when the types hold the same kind of data (possibly recursively)
    - Equivalence assessed by equivalence of the type trees (as above)
    - Structural equivalence is not always easy to do, especially on infinite (graph) types
- Named of structural equivalence is a feature of the language
  - Most (but not all) languages only support named equivalence
    - Modula-3 and Algol have structural equivalence.
    - C, Java, C++, and Ada have name equivalence.
    - Pascal leaves it undefined: up to the implementation

Type Compatibility and Subtyping

- Some languages require equivalent types in their constructs (expressions, assignment, etc.), but most allow for substitutions of compatible types (implicit coercion)
  - An `int` and a `double` are not equivalent, but a function that takes a `double` may take an `int` instead, since `int` can be converted into a `double` without loss of precision
  - This coercion affect both the type checker (which must take the possibility into account) and the code generator (which must generate appropriate code)
- Subtypes are a way of designating compatible types
  - If a type has all of the behaviour of another type so that it can be freely substituted to that other type then it is called a subtype of that type
  - The type checker must be aware of this so that it allows such a substitution
  - Example: C's `enum` is a subtype of `int`
  - Example: Inheritance in OO languages allows the definition of subtypes (a subclass becomes a subtype of the parent class)

SCOPE CHECKING

- Scope constrains the visibility of an identifier to some subsection of the program
  - Local variables are only visible in the block in this they are defined
  - Global variables are visible in the whole program
- A scope is a section of the program enclosed by basic program delimiters such as { } in C
  - Many languages allow nested scopes
  - The scope defined by the innermost current such a unit is called the current scope
  - The scopes defined by the current scope and any enclosing program units are open scopes
  - All other scopes are closed
- Scope checking: given a point in the program and an identifier, determine whether that identifier is accessible at that point
  - In essence, the program can only access identifiers that are in the currently open scopes
  - In addition, in the event of name clashes the innermost scope wins

```
int a;                   // (1)
void bubble(int a) {     // (2)
    int a;               // (3)
    a = 2;               // (3) wins!
}
```

Implementation of Scope Checking

- Scope checking is implemented at the symbol table level, with two approaches
  1. One symbol table per scope organized into a scope stack
     - When a new scope is opened, a new symbol table is created and pushed on the stack
     - When a scope is closed, the top table is popped
     - All declared identifiers are put in the top table
     - To find a name we start at the top table and continue our way down until found; if we do not find it, then the variable is not accessible
  2. Single symbol table
     - Each scope is assigned a number
     - Each entry in the symbol table contains the number of the enclosing scope
     - A name is searched in the table in decreasing scope number (higher number has priority) $\rightarrow$ need efficient data organization for the symbol table (hash table)
     - A name may appear in the table more than once as long as the scope numbers are different
     - When a new scope is created, the scope number is incremented
     - When a scope is closed, all entries with that scope number are deleted from the table and then the current scope number is decremented

Implementation of Scope Checking (cont'd)

1. Stack of symbol tables
   - Disadvantages
     - Overhead in maintaining the stack structure (and creating symbol tables)
     - Global variables at the bottom of the stack $\rightarrow$ heavy penalty for accessing globals
   - Advantages
     - Once the symbol table is populated it remains unchanged throughout the compilation process $\rightarrow$ more robust code
2. Single symbol table
   - Disadvantages
     - Closing a scope can be an expensive operation
   - Advantages
     - Efficient access to all scopes (including global variables)

Scoping Rules

1. Static (lexical) scoping → each function is called in the environment of its definition (lexical placement in the source code)
2. Dynamic scoping → a function is called in the environment of its caller (using the run time stack of function calls)

- Static vs dynamic scoping – Food for thought
  - Scenario: function `bubble()` accesses variable `x`
  - What if there is no `x` in the enclosing context—can this be determined at compile time for static scoping? How about dynamic scoping?
  - What kind of data structures are necessary at compile time and run time to support static or dynamic scoping?
  - What can be done with static scoping but not with dynamic scoping and vice versa?
  - Over time static scoping has largely won over dynamic scoping; what might be the reason?

```
program static_scope_example;
var x:  integer;
var y:  boolean;

procedure p;
    var x:  boolean;
    procedure q;
        var y:  integer;
        begin
            y := x;
        end;
    begin
    end

begin (* main *)
end
```

```
program dynamic_scope_example;
var x:  integer;

procedure p;
begin
    writeln(x);
end;
procedure q;
var x:  integer;
begin
    x:= 2;
    p;
end
begin (*main*)
    x:= 1;
    q;
end
```

Static vs. Dynamic Scoping

- Static scoping
  - Method of non local access that works
  - Getting around restrictions can result in too many globals
  - C++, Java, Ada, Eiffel, Haskell all use static scoping
- Dynamic scoping
  - Program must be traced to read
  - Clashes with static typing
    - Any type error becomes a run-time error!
  - Access to non local variables takes longer
  - Used by APL, SNOBOL, LISP (older)
    - But new LISP (and variants) use static scoping

Static vs. Dynamic Scoping

- Static scoping
  - Method of non local access that works
  - Getting around restrictions can result in too many globals
  - C++, Java, Ada, Eiffel, Haskell all use static scoping
- Dynamic scoping
  - Program must be traced to read
  - Clashes with static typing
    - Any type error becomes a run-time error!
  - Access to non local variables takes longer
  - Used by APL, SNOBOL, LISP (older)
    - But new LISP (and variants) use static scoping
- Overall static scoping is easier to read, is more reliable, and executes faster