Overloading

A number of programming languages, including Java and C++, allow method and subprogram names to be *overloaded*.

This means several methods or subprograms may share the same name, as long as they differ in the number or types of parameters they accept. For example,

```
class C {
  int x;
  public static int sum(int v1,
                        int v2) {
     return v1 + v2;
  }
  public int sum(int v3) {
     return x + v3;
  }
}
```

For overloaded identifiers the symbol table must return a *list* of valid definitions of the identifier. Semantic analysis (type checking) then decides which definition to use.

In the above example, while checking

```
(new C()).sum(10);
```

both definitions of **`sum`** are returned when it is looked up. Since one argument is provided, the definition that uses one parameter is selected and checked.

A few languages (like Ada) allow overloading to be disambiguated on the basis of a method's result type. Algorithms that do this analysis are known, but are fairly complex.

Overloaded Operators

A few languages, like C++, allow operators to be overloaded.

This means users may add new definitions for existing operators, though they may not create new operators or alter existing precedence and associativity rules.

(Such changes would force changes to the scanner or parser.)

For example,

```
class complex{
  float re, im;
  complex operator+(complex d){
    complex ans;
    ans.re = d.re+re;
    ans.im = d.im+im;
    return ans;
} }
complex c,d; c=c+d;
```

During type checking of an operator, all visible definitions of the operator (including predefined definitions) are gathered and examined.

Only one definition should successfully pass type checks.

Thus in the above example, there may be many definitions of **+**, but only one is defined to take **`complex`** operands.

Contextual Resolution

Overloading allows multiple definitions of the same kind of object (method, procedure or operator) to co-exist.

Programming languages also sometimes allow reuse of the same name in defining different kinds of objects. Resolution is by context of use.

For example, in Java, a class name may be used for both the class and its constructor. Hence we see

```
C cvar = new C(10);
```

In Pascal, the name of a function is also used for its return value.

Java allows rather extensive reuse of an identifier, with the same identifier potentially denoting a class (type), a class constructor, a

package name, a method and a field.

For example,

```
class C {
  double v;
 C(double f) {v=f;}
}
class D {
  int C;
  double C() {return 1.0;}
 C cval = new C(C+C());
}
```

At type-checking time we examine all potential definitions and use that definition that is consistent with the context of use. Hence new `C()` must be a constructor, `+C()` must be a function call, etc.

Allowing multiple definitions to co-exist certainly makes type checking more complicated than in other languages.

Whether such reuse benefits programmers is unclear; it certainly violates Java's "keep it simple" philosophy.

Type and Kind Information in CSX

In CSX symbol table entries and in AST nodes for expressions, it is useful to store *type* and *kind* information.

This information is created and tested during type checking. In fact, most of type checking involves deciding whether the type and kind values for the current construct and its components are valid.

Possible values for **`type`** include:

- **`Integer (int)`**
- **`Boolean (bool)`**
- **`Character (char)`**
- **`String`**

- **`Void`**
  `Void` is used to represent objects that have no declared type (e.g., a label or procedure).
- **`Error`**
  `Error` is used to represent objects that should have a type, but don't (because of type errors). **`Error`** types suppress further type checking, preventing cascaded error messages.
- **`Unknown`**
  `Unknown` is used as an initial value, before the type of an object is determined.

Possible values for **`kind`** include:

- **`Var`** (a local variable or field that may be assigned to)
- **`Value`** (a value that may be read but not changed)
- **`Array`**
- **`ScalarParm`** (a by-value scalar parameter)
- **`ArrayParm`** (a by-reference array parameter)
- **`Method`** (a procedure or function)
- **`Label`** (on a **`while`** loop)

Most combinations of **`type`** and **`kind`** represent something in CSX.

Hence **`type==Boolean`** and **`kind==Value`** is a **`bool`** constant or expression.

`type==Void` and **`kind==Method`** is a procedure (a method that returns no value).

Type checking procedure and function declarations and calls requires some care.

When a method is declared, you should build a linked list of (**`type`**,**`kind`**) pairs, one for each declared parameter.

When a call is type checked you should build a second linked list of (**`type`**,**`kind`**) pairs for the actual parameters of the call.

You compare the lengths of the list of formal and actual parameters to check that the correct number of parameters has been passed.

You then compare corresponding formal and actual parameter pairs to check if each individual actual parameter correctly matches its corresponding formal parameter.

For example, given

```
p(int a, bool b[]){ …
```

and the call

```
p(1,false);
```

you create the parameter list

(**Integer**, **ScalarParm**),
(**Boolean**, **ArrayParm**)

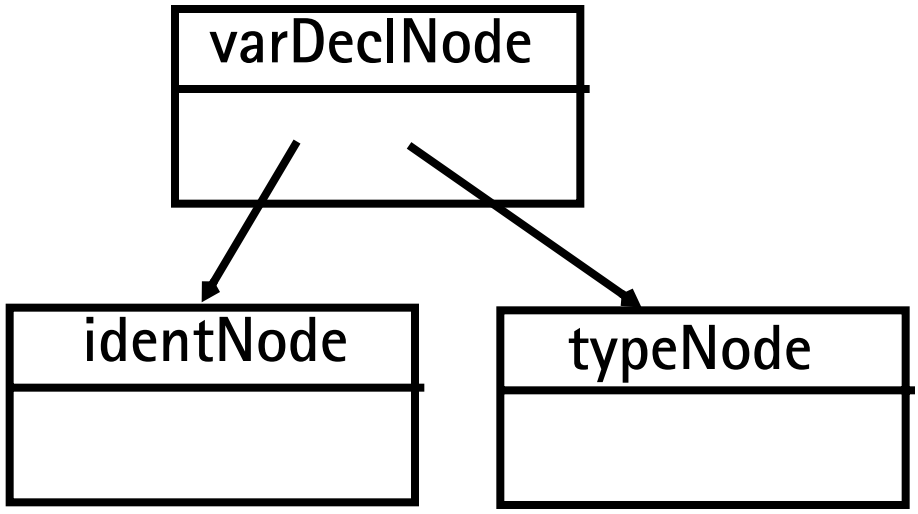
for **p**'s declaration and the parameter list

(**Integer**,**Value**),(**Boolean**, **Value**)

for **p**'s call.

Since a **`Value`** can't match an **`ArrayParm`**, you know that the second parameter in **`p`**'s call is incorrect.

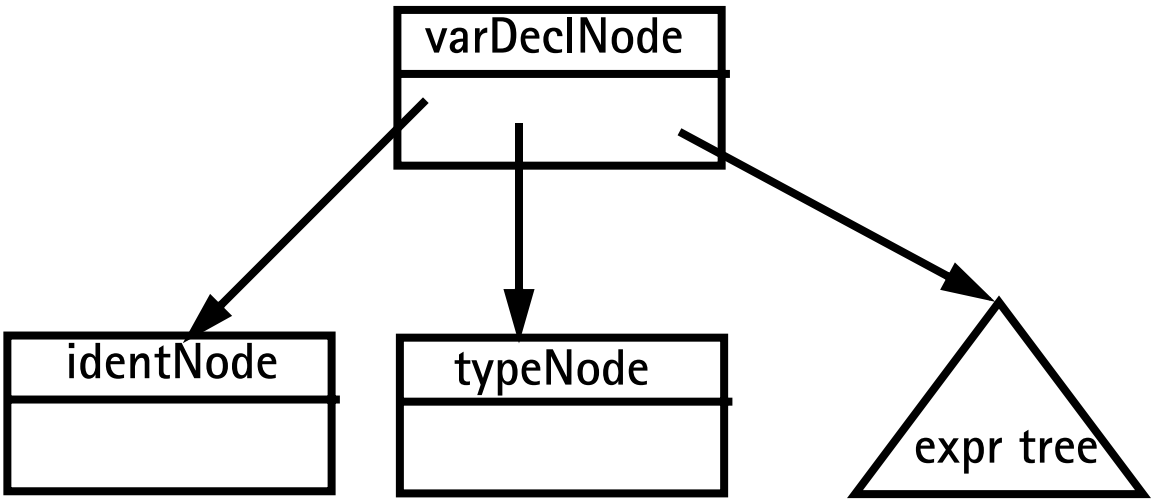
Type Checking Simple Variable Declarations



Type checking steps:

1. Check that **`identNode.idname`** is not already in the symbol table.
2. Enter **`identNode.idname`** into symbol table with
   `type = typeNode.type` and
   `kind = Variable`.

Type Checking Initialized Variable Declarations

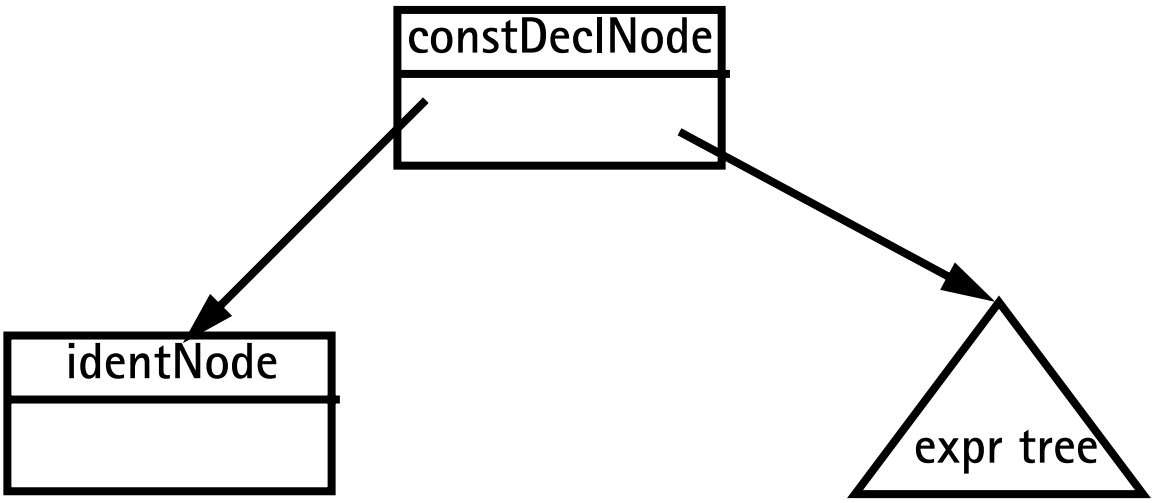


Type checking steps:

1. Check that **`identNode.idname`** is not already in the symbol table.
2. Type check initial value expression.
3. Check that the initial value's type is **`typeNode.type`**

4. Check that the initial value's kind is scalar (**`Variable`**, **`Value`** or **`ScalarParm`**).
5. Enter **`identNode.idname`** into symbol table with
   `type = typeNode.type` and
   `kind = Variable`.

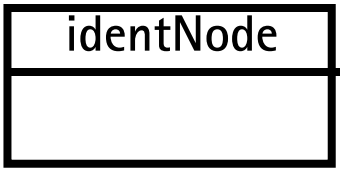
Type Checking Const Decls



Type checking steps:

1. Check that **`identNode.idname`** is not already in the symbol table.
2. Type check the const value expr.
3. Check that the const value's kind is scalar (**`Variable`**, **`Value`** or **`ScalarParm`**).
4. Enter **`identNode.idname`** into symbol table with **`type`** = **`constValue.type`** and **`kind`** = **`Value`**.

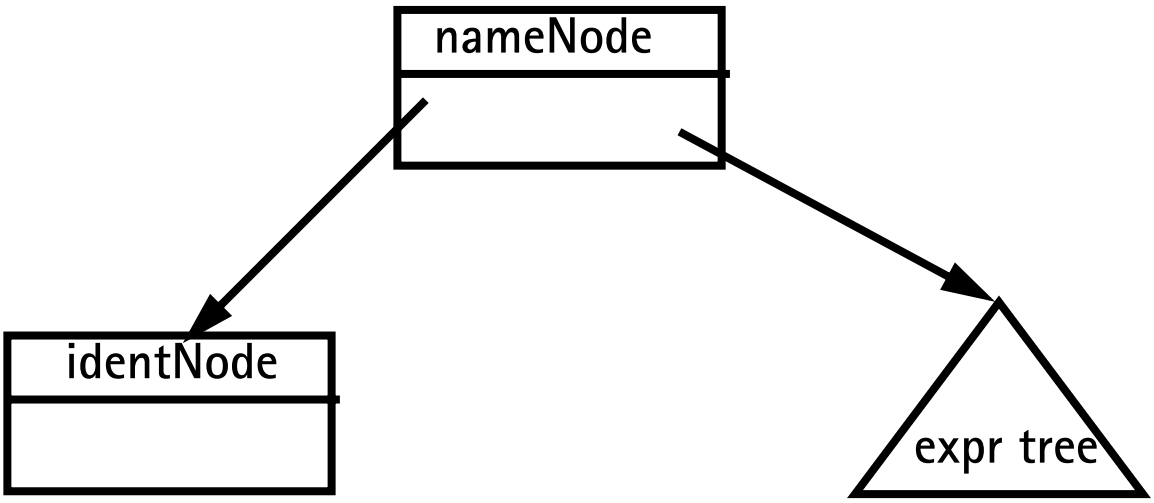
Type Checking IdentNodes



Type checking steps:

1. Lookup **`identNode.idname`** in the symbol table; error if absent.
2. Copy symbol table entry's **`type`** and **`kind`** information into the **`identNode`**.
3. Store a link to the symbol table entry in the **`identNode`** (in case we later need to access symbol table information).

Type Checking NameNodes



Type checking steps:

1. Type check the **`identNode`**.
2. If the **`subscriptVal`** is a null node, copy the **`identNode's type`** and **`kind`** values into the **`nameNode`** and return.
3. Type check the **`subscriptVal`**.
4. Check that **`identNode's kind`** is an array.

5. Check that **`subscriptVal's kind`** is scalar and **`type`** is integer or character.
6. Set the **`nameNode's type`** to the **`identNode's type`** and the **`nameNode's kind`** to **`Variable`**.