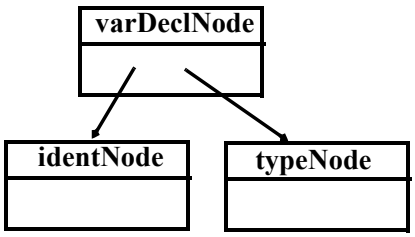
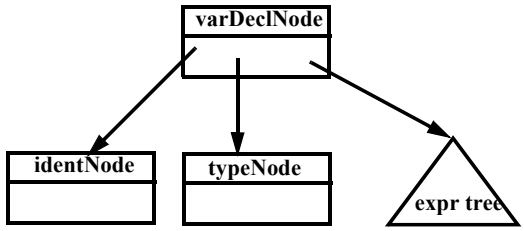
Type Checking Simple Variable Declarations



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

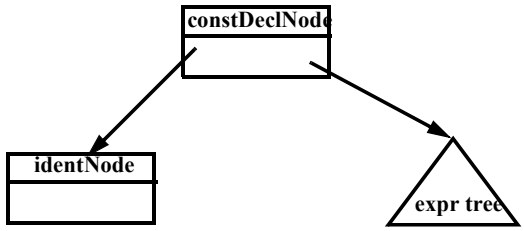


Type checking steps:

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`**.

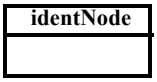
Type Checking Const Decls



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`**.

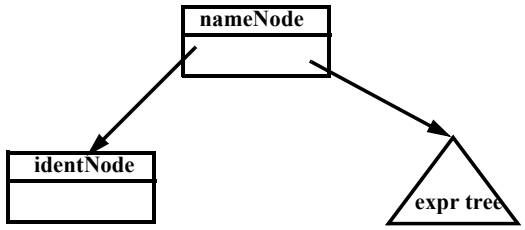
Type Checking IdentNodes



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

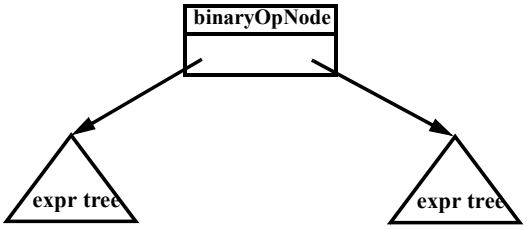


Type checking steps:

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`**.

Type Checking Binary Operators

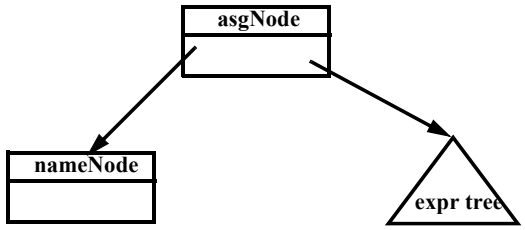


Type checking steps:

1. Type check left and right operands.
2. Check that left and right operands are both scalars.
3. **`binaryOpNode.kind = Value`**.

4. If **binaryOpNode.operator** is a plus, minus, star or slash:
   (a) Check that left and right operands have an arithmetic type (integer or character).
   (b) **binaryOpNode.type = Integer**
5. If **binaryOpNode.operator** is an and or is an or:
   (a) Check that left and right operands have a boolean type.
   (b) **binaryOpNode.type = Boolean**.
6. If **binaryOpNode.operator** is a relational operator:
   (a) Check that both left and right operands have an arithmetic type or both have a boolean type.
   (b) **binaryOpNode.type = Boolean**.

Type Checking Assignments

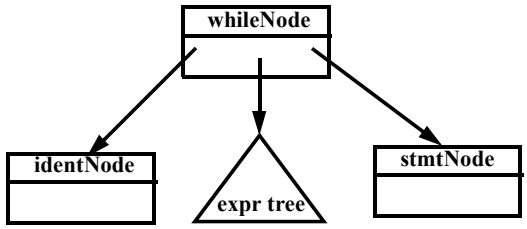


Type checking steps:

1. Type check the **nameNode**.
2. Type check the expression tree.
3. Check that the **nameNode's kind** is assignable (**Variable**, **Array**, **ScalarParm**, or **ArrayParm**).
4. If the **nameNode's kind** is scalar then check the expression tree's **kind** is also scalar and that both have the same type. Then return.

5. If the **nameNode's** and the expression tree's **kinds** are both arrays and both have the same **type**, check that the arrays have the same length. (Lengths of array parms are checked at run-time). Then return.
6. If the **nameNode's kind** is array and its **type** is character and the expression tree's kind is string, check that both have the same length. (Lengths of array parms are checked at run- time). Then return.
7. Otherwise, the expression may not be assigned to the **nameNode**.

Type Checking While Loops

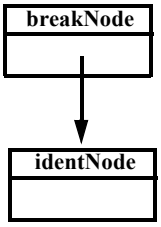


Type checking steps:

1. Type check the **condition** (an expr tree).
2. Check that the **condition's type** is **Boolean** and **kind** is scalar.
3. If the **label** is a null node then type check the **stmtNode** (the loop body) and return.

4. If there is a **`label`** (an **`identNode`**) then:
   (a) Check that the **`label`** is not already present in the symbol table.
   (b) If it isn't, enter **`label`** in the symbol table with **`kind=VisibleLabel`** and **`type= void`**.
   (c) Type check the **`stmtNode`** (the loop body).
   (d) Change the **`label's kind`** (in the symbol table) to **`HiddenLabel`**.

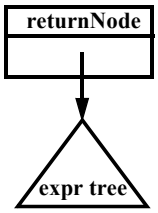
Type Checking Breaks and Continues



Type checking steps:

1. Check that the **`identNode`** is declared in the symbol table.
2. Check that **`identNode's kind`** is **`VisibleLabel`**. If **`identNode's kind`** is **`HiddenLabel`** issue a special error message.

Type Checking Returns

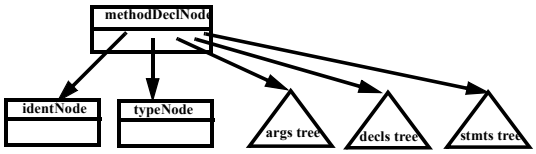


It is useful to arrange that a static field named ***`currentMethod`*** will always point to the **`methodDeclNode`** of the method we are currently checking.

Type checking steps:

1. If **`returnVal`** is a null node, check that **`currentMethod.returnType`** is **`Void`**.
2. If returnVal (an expr) is not null then check that returnVal's kind is scalar and returnVal's type is currentMethod.returnType.

Type Checking Method Declarations (no Overloading)

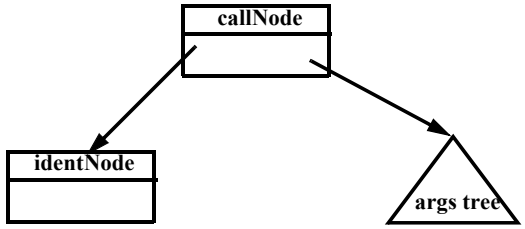


Type checking steps:

1. Create a new symbol table entry **`m`**, with **`type = typeNode.type`** and **`kind = Method`**.
2. Check that **`identNode.idname`** is not already in the symbol table; if it isn't, enter **`m`** using **`identNode.idname`**.
3. Create a new scope in the symbol table.
4. Set **`currentMethod`** = this **`methodDeclNode`**.

5. Type check the **`args`** subtree.
6. Build a list of the symbol table nodes corresponding to the **`args`** subtree; store it in **`m`**.
7. Type check the **`decls`** subtree.
8. Type check the **`stmts`** subtree.
9. Close the current scope at the top of the symbol table.

Type Checking Method Calls (no Overloading)



We consider calls of procedures in a statement. Calls of functions in an expression are very similar.

Type checking steps:

1. Check that **`identNode.idname`** is declared in the symbol table. Its type should be **`Void`** and kind should be **`Method`**.

2. Type check the **`args`** subtree.
3. Build a list of the expression nodes found in the args subtree.
4. Get the list of parameter symbols declared for the method (stored in the method's symbol table entry).
5. Check that the arguments list and the parameter symbols list both have the same length.
6. Compare each argument node with its corresponding parameter symbol:
   (a) Both must have the same type.
   (b) A **`Variable`**, **`Value`**, or **`ScalarParm`** kind in an argument node matches a **`ScalarParm`** parameter. An **`Array`** or **`ArrayParm`** kind in an argument node matches an **`ArrayParm`** parameter.

Reading Assignment

Read Chapters 9 and 12 of **Crafting a Compiler**.

Virtual Memory & Run-Time Memory Organization

The compiler decides how data and instructions are placed in memory.

It uses an *address space* provided by the hardware and operating system.

This address space is usually *virtual*—the hardware and operating system map instruction- level addresses to "actual" memory addresses.

Virtual memory allows:

- Multiple processes to run in private, protected address spaces.
- Paging can be used to extend address ranges beyond actual memory limits.

Run-Time Data Structures

Static Structures

For static structures, a fixed address is used throughout execution.

This is the oldest and simplest memory organization.

In current compilers, it is used for:

- Program code (often read- only & sharable).
- Data literals (often read- only & sharable).
- Global variables.
- Static variables.

Stack Allocation

Modern programming languages allow recursion, which requires *dynamic allocation*.

Each recursive call allocates a *new copy* of a routine's local variables.

The number of local data allocations required during program execution is not known at compile- time.

To implement recursion, all the data space required for a method is treated as a distinct data area that is called a *frame* or *activation record*.

Local data, within a frame, is accessible only while a subprogram is active.

In mainstream languages like C, C++ and Java, subprograms must return in a stack- like manner—the most recently called subprogram will be the first to return.

A frame is pushed onto a *run-time stack* when a method is called (activated).

When it returns, the frame is popped from the stack, freeing the routine's local data.

As an example, consider the following C subprogram:

```
p(int a) {
  double b;
  double c[10];
  b = c[a] * 2.51;
}
```

Procedure **p** requires space for the parameter **a** as well as the local variables **b** and **c**.

It also needs space for control information, such as the return address.

The compiler records the space requirements of a method.

The *offset* of each data item relative to the start of the frame is stored in the symbol table.

The total amount of space needed, and thus the size of the frame, is also recorded.

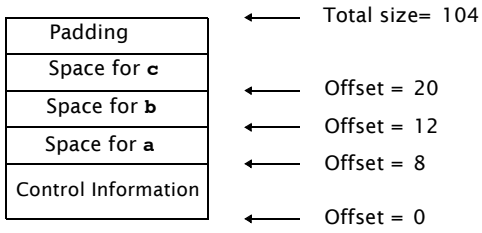
Assume **p**'s control information requires 8 bytes (this size is usually the same for all methods).

Assume parameter a requires 4 bytes, local variable **b** requires 8 bytes, and local array **c** requires 80 bytes.

Many machines require that word and doubleword data be *aligned*, so it is common to pad a frame so that its size is a multiple of 4 or 8 bytes.

This guarantees that at all times the top of the stack is properly aligned.

Here is **p**'s frame:



Within **p**, each local data object is addressed by its offset relative to the start of the frame.

This offset is a fixed constant, determined at compile- time.

We normally store the start of the frame in a register, so each piece of data can be addressed as a (Register, Offset) pair, which is a standard addressing mode in almost all computer architectures.

For example, if register R points to the beginning of **p**'s frame, variable **b** can be addressed as (R,12), with 12 actually being added to the contents of R at run-time, as memory addresses are evaluated.

Normally, the literal **2.51** of procedure **p** is *not* stored in **p**'s frame because the values of local data that are stored in a frame disappear with it at the end of a call.

It is easier and more efficient to allocate literals in a *static area*, often called a *literal pool* or *constant pool*. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.

Accessing Frames at Run-Time

During execution there can be many frames on the stack. When a procedure A calls a procedure B, a frame for B's local variables is pushed on the stack, covering A's frame. A's frame can't be popped off because A will resume execution after B returns.

For recursive routines there can be hundreds or even thousands of frames on the stack. All frames but the topmost represent suspended subroutines, waiting for a call to return.

The topmost frame is *active*; it is important to access it directly.

The active frame is at the top of the stack, so the *stack top*

register could be used to access it.

The run- time stack may also be used to hold data other than frames.

It is unwise to require that the currently active frame always be at *exactly* the top of the stack.

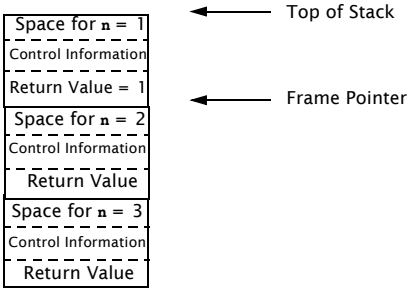
Instead a distinct register, often called the *frame pointer*, is used to access the current frame.

This allows local variables to be accessed directly as offset + frame pointer, using the indexed addressing mode found on all modern machines.

Consider the following recursive function that computes factorials.

```
int fact(int n) {
  if (n > 1)
    return n * fact(n-1);
  else
    return 1;
}
```


The run- time stack corresponding to the call **`fact(3)`** (when the call of **`fact(1)`** is about to return) is:



We place a slot for the function's return value at the very beginning of the frame.

Upon return, the return value is conveniently placed on the stack, just beyond the end of the caller's frame. Often compilers return scalar values in specially