Type Checking (1)

- There are 2 classes of 'checking' that are made when during the lifetime of a program, namely, static and dynamic checking. Dynamic checking occurs during the execution (runtime) of a target program. Static checking is made at compile time.
- Examples of static checks include:
  - Type checks: a compiler should produce an error id an operator is applied to an incompatible operand.
  - Flow control checks: statements that effect the flow of a program must have a 'place' were to redirect the flow. For example, the C `break` statement causes the control to leave the enclosing `while`, `for` or `switch` statement. An error occurs if there is no such enclosing statement.
  - Uniqueness checks: there are situations were an object must be defined only once such as a variable declaration.
  - Name-related checks: sometimes, a name must appear two or more times (`for i = a to b … next i`). The compiler must check that the same name is used in both places.

Type Checking (2)

- A type checker verifies that the type of a construct 'fits' into its current context. For example the Pascal `mod` operator requires integer operands, so the compiler must ensure that this is so.
- A symbol that can represent different operations in differing context is said to be 'overloaded'.
- In principal any check can be made dynamically, if the target code contains enough type information.
- A strongly typed language is one that guarantees that if the compiler accepted the input, it will run without type errors.
- In practice there are some check that can be made only dynamically. For example if we declare an array table: `array[0..255] of char;` and try to reference `table[i]`, the compiler cannot guarantee during execution that the value `i` will lien in the `0..255` range.

A Simple Type Checker (1)

- We will specify a small language in which every identifier must be declared before being used.
- The following grammar generates programs starting from the starting symbol P consisting of a sequence of declarations D followed by a single expression E.

```
P → D ; E
D → D ; D | id : T
T → char | integer |
                array[ num ] of T | ^T
E → literal | num | id | E mod E |
                E [ E ] | E^
```


A Simple Type Checker (2)

- A program that can be generated from the grammar is:

```
key: integer;
key mod 1999
```

- Notes:
  - The basic types in the language are char and integer.
  - We assume all array indices start from 1 so `array[256]` is the equivalent of `array[1..256]`.
  - The prefix `^` operator is the pointer type.
  - In the translation scheme we will use, the action associated with the production `D → id : T`, will save the type information for the identifier in the symbol table.
  - Since in the grammar, `D` appears before `E` in `P → D ; E` it is guaranteed that all the types of identifiers will be known before the expression is checked.

Type Checking Expressions (1)

- The following rules, synthesize the the type of an expression :

E → literal	E.Type ::= char
E → num	E.Type ::= integer
E → id	E.Type ::= lookup(id) *Where **lookup** searches for **id** in the symbol table and returns the type of the declared identifier.*
E → E_1 mod E_2	if E_1.Type AND E_2.Type = integer E.Type = integer else E.Type = type_error

Type Checking Expressions (2)

E → E_1[E_2]	if E_2.Type = integer and E_1.Type = array(s,t) E.Type = t else E.Type = type_error *In **array, s** is the size and **t** is the type.*
E → E_1^	if E_1.Type = pointer(t) E.Type = t Else E.Type = type_error *Where **t** in **pointer(t)** is the type of the pointer.*

Type Checking Statements (1)

- Certain language constructs like statements don't have values *per se* so don't have types associated with them. In this case a special basic type *void* can be assigned to them. If an error is detected the *type_error* type is returned.
- We will be considering `assignment`, `while` and `if` statements here.
- Sequences of statements are separated by semicolons.

Type Checking Statements (2)

S → **id** := E	if id.Type = E.Type
	  S.Type = void
	else
	  S.Type = type_error

S → **if** E **then** S_1	if E.Type = boolean
	  S.Type = S_1.Type
	else
	  S.Type = type_error

S → **while** E **do** S_1	*-same as above-*

S → S_1 ; S_2	if S_1.Type = void and S_2.Type = void
	  S.Type = void
	else
	  S.Type = type_error

Runtime Support (1)

- Before discussing code generation, we will examine the relationship between the text of the source program to the actions that have to occur at runtime to implement it.
- The execution of every procedure is referred to as an activation of that procedure.
- If procedures are nested or recursive multiple activations may exist at any one point.
- Let us assume that a program is made up of procedures such as in Pascal.
- In its simplest form a procedure is the relationship between and identifier and a statement, where the identifier is the procedure name and the statement(s) is the procedure body.
- Procedures that return a value are called functions in many programming languages.

Runtime Support (2)

- A complete program will also be treated as a procedure (think Pascal).
- When a procedure appears in an action statement, we say that the procedure has been *called*.
- A procedure may be also called within an expression.
- Some identifiers within a procedure definition are treated special and are called the *formal parameters* of the procedure (also called arguments).
- When a procedure is called, *actual parameters* are substituted for the formal ones.

Activation Trees (1)

- Assumptions on flow control:
  - Control flows sequentially.
  - The execution of a procedure starts at the beginning of the procedure body and ends at the point following where the procedure was called.
- Each execution of a procedure is referred to as an *activation* of the procedure. The *lifetime* of an activation is the sequence of steps between the first and last steps in the execution of the procedure body (including any other procedures called internally).
- In languages like Pascal, each time control enters a procedure Q from another P, control will eventually return to P in the absence of an error.
- So, if P and Q are procedure activations, their lifetimes are either nested or non-overlapping. That is if Q enters before P is left, then Q must terminate before P does.
- A procedure is recursive if a new activation can begin before an earlier activation of the same procedure finished (note: recursion may be indirect (P calls Q which calls P).

Activation Trees (2)

- We use a tree structure called an activation tree to depict this control flow. In this tree:
  - Each node represents an activation of a procedure.
  - The root node represents the activation of the main program procedure.
  - The node for A is the parent of another node B **iff** control flows from activation A to B.
  - The node for A is to the left of the node for B **iff** the lifetime of A occurs before that of B.

Example (1)

```
program sort
  var a: array[0..10] of integer;

  procedure readarray;
  var i:integer;
  begin
    for i := 1 to 9 do read(a[i]);
  end;

  function partition(y,z:integer):integer;
  var ...
  begin
    ...
  end;
```


Example (2)

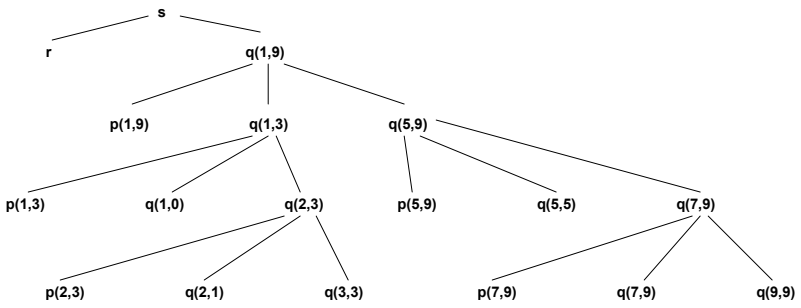
```
  procedure quicksort(m,n:integer);
  var i:integer;
  begin
    if (n > m) then begin
      i := partition(m,n);
      quicksort(m,i-1);
      quicksort(i+1,n);
    end;
  end;

  begin
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9);
  end.
```

Example (3)

- Activation Trace:

```
Execution Begins
Enter readarray
Leave readarray
Enter quicksort(1,9)
Enter partition(1,9)
Leave partition(1,9)
Enter quicksort(1,3)
...
Leave quicksort(1,3)
Enter quicksort(5,9)
...
Leave quicksort(5,9)
Leave quicksort(1,9)
Execution Finishes
```

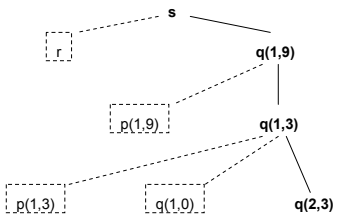

Example (4)



Control Stacks (1)

- The flow of control of a program corresponds to a depth first traversal of the activation tree.
  - (starts at the root, visits nodes before children and visits children in a left-to-right order)
- The trace we have seen before can be reconstructed by traversing the previous tree as illustrated above.
- We can use a stack called the **control stack** to keep track of live procedure activations. The idea is to push a node onto the stack when activation begins and popping it off when activation ends.
- When a node n is on top of the control stack, the stack contains the nodes along the path from n to the root (start).

Control Stacks (2)



The state of the stack when q(2,3) is on top.

Activation Records

- Information (memory space) needed for the execution of a single procedure is managed by a block of storage called an **activation record**.
- Not all languages or compilers use the same structure for this record.
- Common fields in this record are:
  - Temporaries: temporary values such as those intermediate values when evaluating an expression.
  - Local data: local values to the procedures.
  - Saved machine status: the state just before the procedure was called.
  - Access link: link to non-local data.
  - Control link: link to the activation record of the calling procedure.
  - Actual parameters: values of the actual parameters passed the procedure.
  - Returned value: the returned value if the procedure is a 'function'.
- 'Out of Stack Space' issue in infinitely recurring calls.
- The sizes of most fields are usually determined at compile time with exceptions if there is a local array whose size depends on an actual argument or the procedure can take a variable number of parameters.

Intermediate Code

- It is common practice for the front end of a compiler to produce an intermediate form of code before passing that on to the backend to generate the target code itself.
- This is desirable since:
  - Retargeting is facilitated (a compiler for the same language but different machine).
  - A machine-independent code optimiser may be developed (optimisation applied to the intermediate code).
- We will assume that at this point the language has been parsed and statically checked.

Intermediate Languages (1)

- Syntax trees and postfix are two types of intermediate representations.
- In this section we will discuss a new one called the **three address code**.
- The three address code (3AC – *my abbreviation!*) is a sequence of statements of the general form:

  `x := y` *`op`* `z`
- Where x, y and z are names, constants or compiler-generated temporaries.
- Op, stands for an operator such as integer or floating-point arithmetic operators or a logical operator on boolean data.

Intermediate Languages (2)

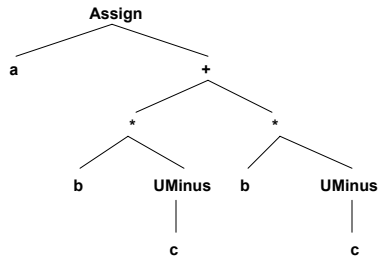
- Note that no 'built-up' expressions are allowed since there is only one operator in the RHS. So, something like `p + q * r`, would look like:

  t_1 `:= q * r`

  t_2 `:= p +` t_1
- Where, t_1 and t_2 are compiler-generated temporaries.
- The use of names for intermediate values allows 3AC to be easily rearrange unlike postfix notation.
- 3AC is a linear representation of the syntax tree (like postfix).
- The reason for the term 'Three Address Code' is that each statement **usually** contains 3 addresses, 2 for the operands and 1 for the result.

Intermediate Languages (3)

Syntax Tree



3AC

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

a := b * -c + b * -c

Types of 3AC (1)

Type	Form	Notes
Assignment	x := y op z	Op is a binary arithmetic or logical operator.
Assignment	x := op y	Op is a unary operator (-, NOT,…)
Copy	x := y	Copy y into x.
Unconditional Jump	goto L	Where L is a label to the next statement to run.
Conditional Jump	if x relop y goto L	Apply a relational operator (>, <, <=,…) to x and y and jump to L if true otherwise continues with the next code.
Parameters, Calls and Returns	param x call p, *n* return y	y is an optional return value. Typically used as a sequence: param x1 … param x*n* call p,*n*

Types of 3AC (2)

Type	Form	Notes
Indexed Assignments	x := y[i] x[i] = y	The first sets x to the value in the location i memory units beyond y. The second sets the value at the location i memory units beyond x to y.
Address/Pointer Assignments	x := &y x := *y *x = y	The first sets the value of x to the memory location of y. In the second, presumably y is a pointer. In the third, presumably x is a pointer.

Note:
The operator set in the design of the 3AC must be rich enough to describe the operations in the source language. A small set is easier to implement on the target machine, but the resulting code would be very long, making the life of the optimiser harder if it is to produce good code.

Code Generation (1)

- The final phase in compiler is the code generator which takes an intermediate representation of a source program and generates equivalent target code.
- In between the intermediate code stage and code generation stage there could be a code optimisation stage. Code optimisation may be implemented on the final target code too.
- The requirements generally imposed on a code generator are that the target code should be correct of high quality and effectively use resources on the target computer. Also the code generator itself must be efficient.
- Mathematically, the problem of generating optimal code is undecidable. In practice, heuristics that generate good code (not necessarily optimal) are typically used.
- The choice of such heuristics is important. Carefully designed code generators may produce code that is several times faster than that produced by a bad one.

Code Generation (2)

- In order to design an efficient code generator the designer must have intimate knowledge of the target hardware and operating system.
- Issues such as memory management, instruction selection, register allocation and evaluation order are inherent to almost all code generation problems.
- Due to highly specialised, platform-dependent issues, in this section will examine generic design issues only.

Input to the Code Generator

- The input is typically,
  - The intermediate code produced by the front end.
  - The symbol table that is used to determine the runtime addresses of the data objects denoted by the names in the intermediate representation.
- We assume that,
  - The source code has been properly scanned and parsed correctly.
  - All relevant information is available to the code generator.
  - Type checking has occurred.
  - In general the input is error-free.

Target Programs

- The output of a code generator is the target language. This may take on different forms such as,
  - Absolute machine code,
  - Relocatable machine language,
  - Or assembly language.
- Producing absolute machine code has the advantage that it can be placed in a fixed memory location and execute immediately.
- Producing relocatable (**object**) code has the advantage that separate sub-programs may be compiled separately and then linked and loaded to execute. Whilst the code has the overhead of linking and loading we gain a lot of flexibility (think DLLs – though not exactly).
- Producing assembly language makes the code generation task simpler but involves the extra step of assembling the output.

Memory Management

- Mapping names in the source program to addresses of data objects is done cooperatively by the front-end and back-end of the compiler.
- A name in a 3AC statement refers to a symbol table entry for the name.
- The type of a declaration determines the amount of storage allocated in memory (e.g. a long integer would take up 4 bytes).

Instruction Selection (1)

- The nature of the instruction set of the target machines determines the instruction selection when generating code.
- Also, if the target machine does not support each data type natively, special arrangements have to be made.
- Instruction speeds are an important factors when generating code. If the quality of the target code is not an issue, then each 3AC statement could be associated with a 'template'. For example, every 3AC statement of the form `x := y + z` could be translated into:

```
MOV y, R0
ADD z, R0
MOV R0, x
```


Instruction Selection (2)

- Unfortunately, this technique can (and most likely will) produce inefficient code. For example:

```
a := b + c
d := a + e
```

- Will produce:

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

- There the third and fourth statements are redundant if a is not subsequently used.

Instruction Selection (3)

- The quality of code is usually determined by its execution speed and its size.
- A target machine with a rich instruction set may provide several ways to perform any given operation. In this case a ‘narrow-minded’ code generator may produce correct, though unacceptably inefficient code.
- For example, say, the machine supports an increment (INC) operation. Then the 3AC instruction a := a + 1 may be implemented more efficiently by the single instruction rather than using the ‘template’ we have seen before.

Register Allocation

- Instructions involving register operands are usually shorter and much faster than those involving memory operands. For this reason, efficient utilization of registers is important in generating fast code.
- The use of registers is often subdivided into two problems:
  - During **register allocation**, we select the set of variables that will reside in registers at a point in the program.
  - During subsequent **register assignment**, we pick the specific register that the variable will ‘live’ in.
- Finding the optimal assignment of registers is mathematically NP complete and further restrictions may be enforced by the hardware and/or operating system