

# C and Compilers 2006

**Dr. James A.D.W. Anderson**  
**Room 134**



# Objectives

- To give an opportunity to learn the C programming language.
- To teach elementary compiler writing.
- To present research challenging the Turing model of computation.

# Examinations

- The course is assessed 30% by course work and 70% by examination.
- The Department's rules on submission of course work and penalties for late work apply.

# Assessment

- Write a compiler in C or C<sup>++</sup> that reads C source code and compiles *main* to Three Address Code (TAC) implemented in C.
- Deadline - Monday of week 10. That is, 11 December, 2005.
- Submit a hard copy of a word processed report to the Student Information Office by the deadline.
- The report should explain the design of your compiler, give the input and output of example runs, and contain the full source code of the compiler.

## Note

There are two different ways to write a compiler.

- Specify the language in a right-recursive grammar and translate this by hand to recursive subroutines that implement the lexical analyser, parser, and code generator.
- Specify the language in a left-recursive grammar and use a compiler generator to generate the lexical analyser and parser from the grammar. The grammar will be augmented with hand-written code that is used to generate a code generator.
- There is an algorithm for converting left-recursive grammars to right-recursive ones and *vice versa*.

# Why Learn C?

- C is a very popular language that generates efficient code.
- There are many programming environments for C.
- There are many compiler writing tools written in C. For example, *lex* and *yacc* may be obtained from:

[www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html](http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html)

- C is a defective language which illustrates many language faults.

# How To Learn C

- Work through on-line courses on the C Programming Language.
- Read books on C.
- Read the C programs in these lecture notes and elsewhere.
- Make notes when we discuss compiling C.
- Write programs in C and test them.

# How To Learn Compiler Writing

- Read books on compiler theory and practice.
- Read books on compiler tools.
- Read web documents on compilers.
- Write some compilers.
- Test your own and pre-existing compilers.

# Recommended Text Book

- *Algorithms for Compiler Design* O.G. Kakde, pub. Charles River Media Inc., Hingham, Massachusetts, (2003).

## Other Books to Read

- *lex & yacc* 2<sup>nd</sup> edn. Levine, J., Mason, T. & Brown, D. pubs. O'Reilly (1992).
- *A Retargetable C Compiler: Design and Implementation* C.W. Fraser & D.R. Hanson, Addison-Wesley, (2003).
- *Introduction to Compiling Techniques* 2nd edn. Bennett, J.P. pubs. McGraw-Hill (1990).
- *Compiler Design in C* Holub, A.I. pubs. Prentice-Hall (1996).

# Quiz

- What is a compiler, linker, and loader?
- What is a language, grammar, sentence, symbol?
- What is a computer?
- Can a computer be programmed without using a language?
- What does a computer do?
- Is there anything that a computer cannot do?
- How do human and computer languages differ?

# Quiz

- How do schools and universities differ?

# Hello World

```
/* This is a very simple program. */  
  
#include <stdio.h>  
  
main() {  
    printf("Hello, world\n");  
}
```

# Printf

The procedure *printf* takes a string as argument and prints it on the standard output stream (*stdout*). Optional arguments indicate the values that are to be printed by conversions embedded in the string. Conversions are introduced by percent ‘%’, special characters are escaped with backslash ‘\’.

```
int printf(char *format, ...);
```

```
printf("My lucky number is %d\n", 7);
```

# Optimising Constant Programs

- If a program has no input it is a constant program. It can be optimised to literal statements that supply the output.
- If a program has no output it does nothing. It can be optimised to a null program.
- I/O can be supplied by devices or volatile memory. Hence C's type qualifier *volatile*.
- Delay loops cannot be optimised! Delays must be provided by kernel timers.

# Conditionals

- Conditional compilation (inclusion) is available via the preprocessor. Commands begin with '#’.
- There is a trinary conditional operator: ‘? :’.
- Conditional (selection) statements: *if, if else, switch*.

# Conditional Operator (?:)

*conditional-expression:*

*logical-OR-expression*

*logical-OR-expression ? expression :*  
*conditional-expression*

The first expression is evaluated, including all side effects, if the result is non-zero then the result is the value of the second expression otherwise that of the third expression. Only one of the second and third expressions is evaluated.

```
x = (x < y) ? x : y;
```

```
x = (x < y) ? x : (y < z) ? y : z;
```

# Selection Statement (if)

**if** ( *expression* ) *statement*

A selection statement selects among a set of statements depending on the value of a controlling expression.

```
if (TRUE)
    ;
if (x >= y) x = y;
if (x >= y) {
    x = y;
    printf("Value: %i\n", x);
}
```

# Selection Statement (if else)

**if** ( *expression* ) *statement* **else** *statement*

A selection statement selects among a set of statements depending on the value of a controlling expression. Only one branch out of the conditional is evaluated.

```
if (x >= y)
    if (y < z)
        x = y;
    else
        x = z;
```

```
x = (x < y)? x : (y < z)? y : z;
```

# Selection Statement (switch)

**switch** ( *expression* ) *statement*

A selection statement selects among a set of statements depending on the value of a controlling expression. Control drops through ‘case’ labels.

```
switch (x) {  
case 0:  printf("Hello ");  
case 1:  printf("World.\n");  
        break;  
default: printf("Default x is %i?\n", x);  
}
```

# Loops

- The loop (iterative) statements are: *for*, *while*, and *do*.

# Iterative Statement (for)

**for** ( *expression\_1* ; *expression\_2* ; *expression\_3* )  
*statement*

Except for the behaviour of a *continue* statement in the loop body, the above is equivalent to the following.

```
expression_1;  
while ( expression_2 ) {  
    statement ;  
    expression_3 ;  
}
```

```
for (i=0; i<10; printf("%i\n", i++));
```

# Jump Statements

- The jump statements are: *goto*, *continue*, *break*, *return*, and *exit*.

# Jump Statement (goto)

**goto** *identifier*

The *identifier* must be used as a label in the current block of code.

```
i = 0;
startloop:
if (i < 10) {
    printf("%i\n", i++);
    goto startloop;
}
```

# Example Program

```
/* Print Farehnheit-Celcius table
   for fahr = 0, 20, ..., 300;
   using floating point numbers.
*/

#include <stdio.h>

#define RATIO (5.0/9.0)
#define OFFSET (-32.0)

main() {

    /* Declare local variables. */
    float fahr, celsius;
    int lower, upper, step;
```

# Example Program

```
/* Initialise local variables. */
lower = 0;
upper = 300;
step  = 20;

/* Print table. */
fahr = lower;
while (fahr <= upper) {
    celsius = RATIO*(fahr + OFFSET);
    printf("%3.0f %6.1f\n",
           fahr,
           celsius
    );
    fahr += step;
}
}
```

# Summary

- You may use CPP or any other pre-processor in the assessment.
- The library *stdio* supplies the output procedure *printf*.
- Conditionals are: *'? :'*, *if*, *if else*, *switch*.
- Loops are: *for*, *while*, *do*.
- Jumps are: *goto*, *continue*, *break*, *return*, *exit*.  
There are also some exotic jumping functions provided by libraries.

# Simple Type Specifiers

The simple type specifiers in C are: *int*, *float*, *char*, *void*, *short*, *long*, *double*, *signed*, *unsigned*.

- The default type is *int*.
- An *int* has the natural size suggested by the architecture, usually one word long.
- A *float* is a floating point number, usually one word long.
- A *char* is large enough to represent the basic execution character set, usually one byte long.
- A *void* is the absence of a type. Void functions cannot contain a *return* statement.

# Sizeof

C provides an operator *sizeof* that returns the size of its argument measured in bytes. This operator is implemented in the compiler and uses type information to compute the size of its argument.

- $\text{sizeof}(\text{char}) == \text{sizeof}(\text{signed char}) == \text{sizeof}(\text{unsigned char})$
- $\text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short int}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long int})$
- $\text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double})$
- How can *sizeof* be used to find the number of elements in an array?

# Function Prototype

Function prototypes:

- Allow the compiler to check the call and return values of a function are consistent.
- Allow mutually recursive functions to be defined.

*return-type function-name ( parameter-list )*

```
int power(int m, int n);
```

# Function Definition

In the following grammar rule the back slash is a meta character that denotes continuation of the line.

*return-type function-name ( parameter-list<sub>opt</sub> ) \*  
*{ declarations statements }*

```
int power(int base, int exp) {  
    int i, p;  
    for(i=p=1; i<=exp; i++)  
        p *= base;  
    return p;  
}
```

# Example Program

```
/* Print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300;
   using floating point numbers.
*/

#include <stdio.h>

#define RATIO (5.0/9.0)
#define OFFSET (-32.0)

main() {

    /* Declare local variables. */
    float fahr, celsius;
    int lower, upper, step;
```

# Main Prototype

- You can declare a prototype for the function *main* with no arguments.

```
int main(void);
```

- You can declare a prototype for the function *main* with two arguments.

```
int main(int argc, char *argv[]);
```

# Declarations and Definitions

- Declarations tell the compiler what type(s) an object has. Declarations do not allocate storage.
- Definitions allocate storage for an object of a declared type.
- It is usually a good policy to put declarations, but no definitions, in a header file. Thus, the declarations can be shared amongst all programs needing access to them and objects can be allocated memory in just one place.

# Structure

- Structure declaration.

```
struct point {  
    int x;  
    int y;  
};
```

- Structure definition.

```
struct point p1;
```

# Structure-pointer

- Structure definition.

```
struct point {  
    int x;  
    int y;  
} p2;
```

- Pointer declaration.

```
struct point *ptrp2;
```

- Pointer definition.

```
ptrp2 = &p2;
```

# Structure-member referencing

- Structures are dereferenced by the “.” operator.

```
p1.x = 20;  
printf("%d,%d", p1.x, p1.y);
```

- Pointers to structures are dereferenced by the “->” operator.

```
ptrp2->x = 20;  
printf("%d,%d", ptrp2->x, ptrp2->y);
```

# Unions

- Unions can hold different types of objects at different times. The compiler reserves enough space for the largest element.

```
union u_tag {  
    int ival;  
    float fval;  
    char *sval;  
} u1;  
union u_tag u2;
```

- What is declared and defined above?

# Unions

- Unions are dereferenced just like structures.
- It is the programmer's responsibility to ensure that reads and writes of union elements are consistent.
- How would you record what type of element a union contains?

# Symbol Table

A compiler uses a symbol table to record information about symbols built in to a language and those declared by an end-user programmer.

- A symbol table can hold all of the information needed to compile named objects.
- A symbol table allows error checking, for example of type compatibility, name clashes, and missing declarations.

# Example Symbol Table

```
#include <stdio.h>
```

```
#define SYMTABSIZE 1000
```

```
#define INT 1
```

```
#define FLOAT 2
```

```
#define STRING 3
```

# Example Symbol Table

```
/* Define a symbol table.*/  
struct {  
    char *lexeme;  
    int  token;  
    int  valtype;  
    union {  
        int    ival;  
        float fval;  
        char *sval;  
    } val;  
} symtab[SYMTABSIZE];
```

# Example Symbol Table

```
switch (symtab[i].valtype) {
case INT:
    printf("%d ", symtab[i].val.ival);
    break;
case FLOAT:
    printf("%f ", symtab[i].val.fval);
    break;
case STRING:
    printf("%s ", symtab[i].val.sval);
    break;
}
```

- A switch is silent if the expression value does not match any of the cases.

# Advice

You might find it helpful to read about the following in a text book, or in the UNIX man pages.

- ctype
- setlocale
- strtok

On the other hand it might make you despair!

# Summary

- C is a weakly typed language with no specific size for types, hence the need for the *sizeof* operator.
- Prototypes can be put in header files as a common declaration.
- Prototypes allow the definition of mutually recursive functions.
- Structures are de-referenced by the “.” operator.
- Pointers are de-referenced by the “->” operator.
- The operator “&” returns the address of its argument.

# Typedef

The declaration *typedef* makes a synonym for a type specifier (name).

```
typedef char *Lexeme;  
typedef int Token;  
typedef struct {  
    Lexeme lex;  
    Token valtype;  
    union {  
        int ival;  
        float fval;  
        char *sval;  
    } val;  
} Value;
```

# Cast

- Preceding an expression by a parenthesised type name converts the value of the expression to the named type.

```
Value *valloc(void) {  
    return (Value *) malloc(sizeof(Value));  
}
```

What does the above code do?

# Quiz

- C policy is to use upper case names for defined types, like “FILE”. Is this sensible?
- How does *typedef* in C compare to *TYPE* in Modula-2 or to similar declarations in strongly typed languages?
- What are the advantages and disadvantages of using *typedef*.

# Bit fields

Bit fields are machine dependent structures whose fields are a specified width in bits. Fields might be allowed to cross word boundaries. A null field-width forces alignment on a word boundary.

```
struct identifier {  
    unsigned int          : 0;  
    unsigned int is_keyword : 1;  
    unsigned int is_external : 1;  
    unsigned int is_static  : 1;  
    unsigned int is_constant : 1;  
    unsigned int is_volatile : 1;  
};
```

# Register

Automatic (non-static local) variables can be allocated to registers. However, the compiler is free to ignore register allocations. Sometimes optimising compilers can do a better job of register allocation than a human programmer.

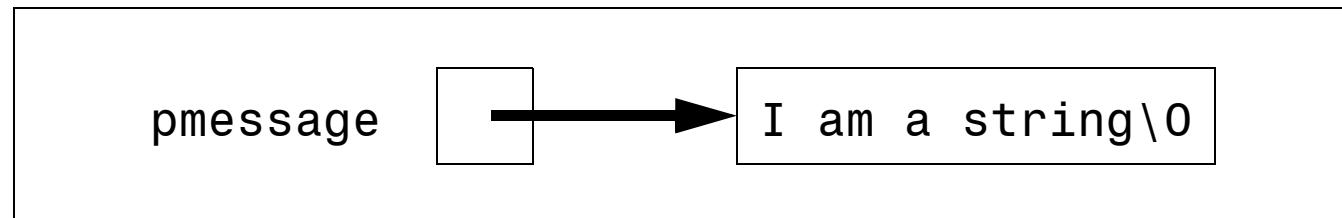
```
void foo(void) {  
    register int    x;  
    register char   y;  
    register Huge  *z;  
}
```

# Arrays

Strings are represented by null-terminated character arrays.

- The string “A” has two elements: A\0.
- Strings are passed by reference (as pointers).

```
char *pmessage;  
pmessage = "I am a string";  
printf(pmessage);
```



# Array Addressing

- Arrays can be addressed by index.

```
/* Copy t into s. */
void strcpy(char *s, char *t) {
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

- Arrays can be addressed by pointer.

```
/* Copy t into s. */
void strcpy(char *s, char *t) {
    while (*s++ = *t++);
}
```

# Strcmp

- Strcmp can be implemented by index or pointer.

```
/* return: <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    int i;
    for (i=0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

```
/* return: <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    for (; *s == *t; s++,t++) if (!*s) return 0;
    return *s - *t;
}
```

# Advice

- There are many standard libraries in C.
- String handling is provided in `<string.h>` and its corresponding library.

# Array Zeroing

- Static memory is guaranteed to be zero on first access, dynamic memory is undefined.

```
#define len 10

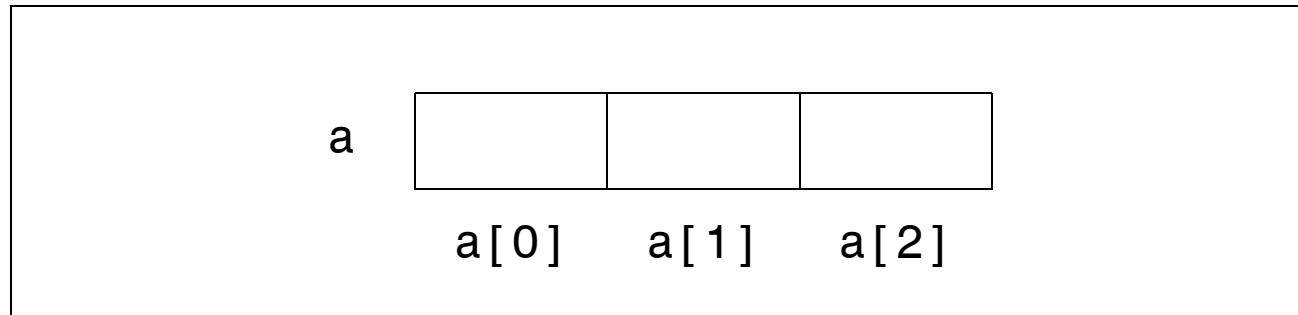
/* Static memory already zeroed. */
int zeroed[len];

/* Dynamic memory arbitrary. */
main () {
    int tozero[len], i;
    for(i=0; i<len; tozero[i++]=0);
}
```

- What are the relative merits of storing arrays in static and dynamic memory?

# Array Definition

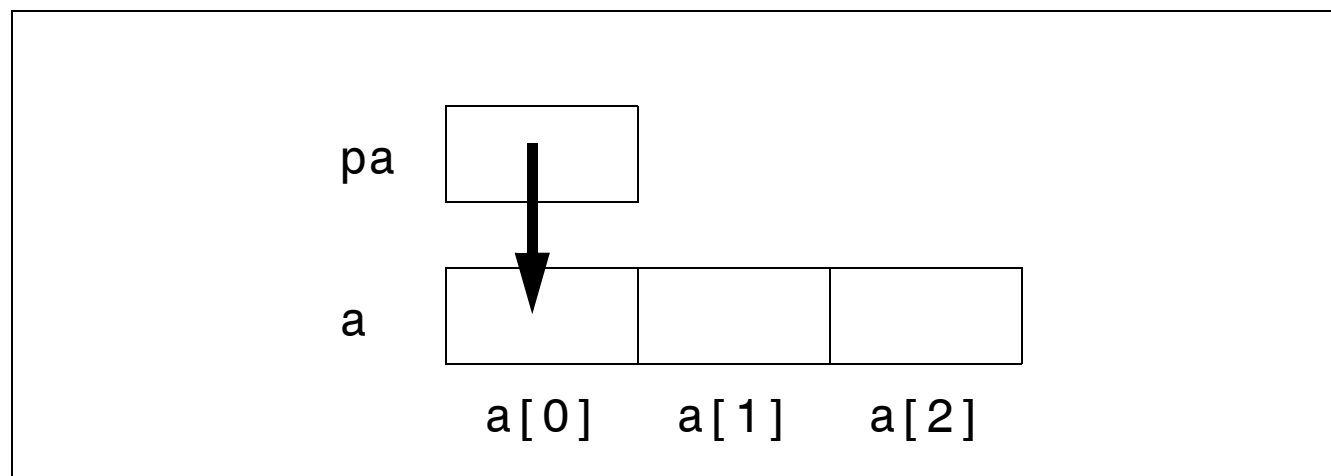
```
/* Define 'a'. */  
int a[3];
```



# Pointer to an Array

```
/* Declare pa. */  
int *pa;
```

```
/* Define pa. */  
pa = &a[0];
```



# Address Arithmetic

```
pa = a;    /* Define pointer. */
```

```
pa++;     /* Increment pa by 1. */
```

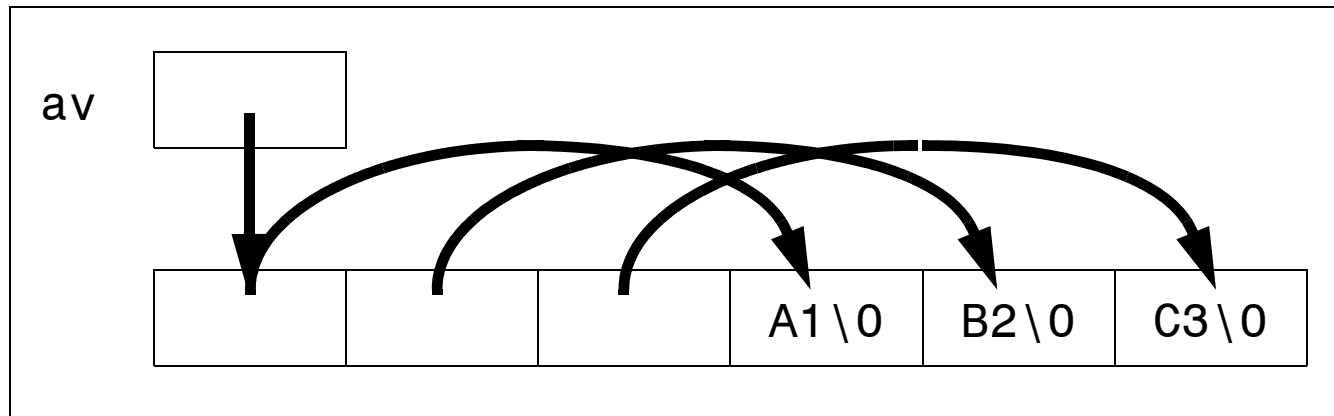
```
pa += i;  /* Increment pa by i. */
```

```
a[i];    /* Access i'th element. */
```

```
*(a+i);  /* Access i'th element. */
```

# Array of Strings

```
/* Efficient definition. */  
char *av[] = {"A1", "B2", "C3"};
```



```
/* Inefficient definition. */  
char *av[99] = {"A1", "B2", "C3"};
```

- Why is the definition immediately above inefficient?

# Echo

```
#include <stdio.h>

main(int ac, char *av[]) {
    int i;
    for(i=1; i<ac; i++)
        printf("%s%s", av[i], (i<ac-1) ? " " : "");
    printf("\n");
    return 0;
}
```

- What does the second string in *printf* do?
- What does the third string in *printf* do?
- How could the conditional in *printf* be made easier to read?

# Echo

```
#include <stdio.h>

main(int ac, char *av[]) {
    while (--ac > 0)
        printf("%s%s", *++av, (ac>1) ? " " : "");
    printf("\n");
    return 0;
}
```

- How does this version of *echo* work?

# Summary

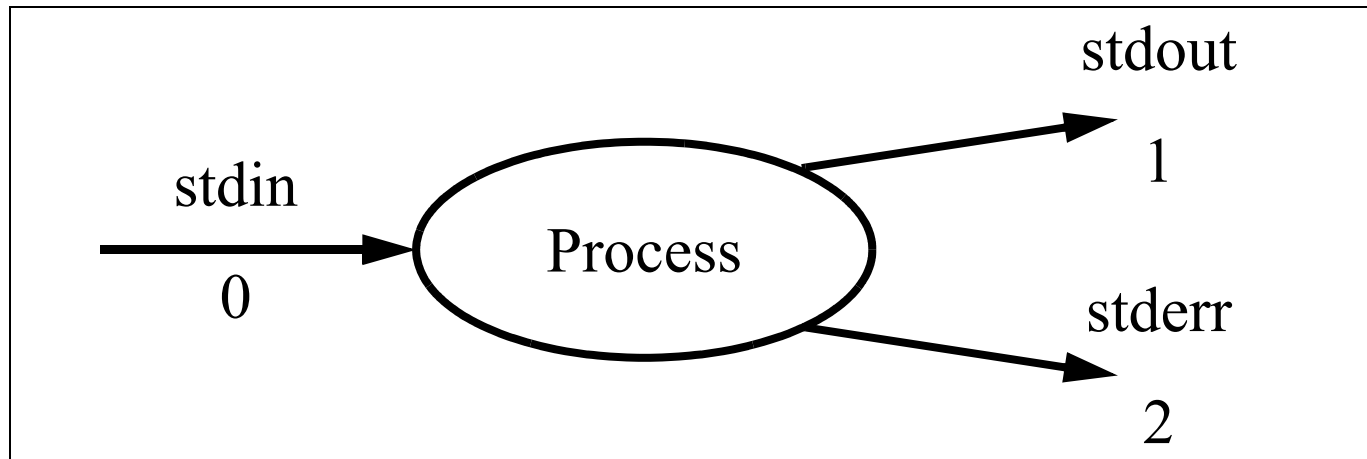
- C provides the structures you would expect of a simple procedural language.
- C is a weakly typed language using structural equivalence.
- C structures and types provide direct access to machine resources such as: bits, words, addresses, registers.
- A String 'is an array of *char* that is null terminated:\0.
- Arrays may be addressed by index or pointer.

# Summary

- The argument vector, known as “av” or “argv”, is an array of strings, i.e. an array of pointers to strips of characters with each strip terminated by null.
- The argument count, known as “ac” or “argc”, runs from 1 ... N, but the corresponding arguments are indexed from 0 ... N-1.

# Files

Everything in UNIX is either a process or a file.



- Files can be unbuffered and are accessed by file descriptors (e.g. 0,1,2).
- Files can be buffered and are accessed by file pointers (e.g. *stdin*, *stdout*, *stderr*).

# Output

- The procedure *printf* prints to the file stream *stdout*.

```
printf("Hello world\n");
```

- The procedure *fprintf* prints to any file stream.

```
fprintf(stdout, "Hello world\n");  
fprintf(stderr, "Oops!\n");
```

- Buffers are flushed automatically when they are full, implicitly on a new line, or explicitly by a call to the operating system flush functions.

# File Stream Opening

```
#include <stdio.h>

/* Declare pointer and function. */
FILE *fp,
    *fopen(char *name, char *mode);

/* Define file pointer. */
fp = fopen("junk.c", "r");
```

- The basic modes are Read (“r”), Write (“w”), and Append (“a”). These can be combined with each other and with other modes.

# File Stream Opening

The function *fopen* returns null on error. This error should be trapped.

```
#define ENREAD -1
#define EMREAD "Cannot read file "

if (!(fp = fopen("junk.c", "r"))) {
    fprintf(stderr, EMREAD);
    exit(ENREAD);
}
```

- It is possible to register functions that are called on `exit`.

# File Stream I/O

- There are functions to get a character from an input stream and put a character on an output stream.

```
int getc(FILE *fp);  
int putc(int c, FILE *fp);
```

- There are specialisations of these for *stdin* and *stdout*.

```
int getchar();  
int putchar(int c);
```

- There is also a function to push a character (back) onto an input stream.

```
int ungetc(int c, FILE *fp);
```

# File Stream I/O

The output procedures *printf* and *fprintf* have corresponding reading procedures *scanf* and *fscanf*.

```
scanf(char *format, ...);  
fscanf(FILE *fp, char *format, ...);
```

- It is possible to use *fscanf* instead of *atoi*, *atol*, *atof*.
- It is possible to use *fprintf* instead of *itoa*.
- All of the reading and writing procedures return either (a) the number of characters read or written or (b) a negative number if an error occurs. Reading *EOF* is sometimes considered an error. You should check this number and exit silently on real errors.

# File Stream Closing

- When files are closed they are also flushed.

```
#include <stdio.h>
int fclose(FILE *stream);
```

- Files can be flushed explicitly.

```
#include <stdio.h>
int fflush(FILE *stream);
```

- Files are closed automatically on *exit* and return from *main*. Abends and calls to *abort* do not close files.

# Example Cat

```
/* cat: concatenate files as in UNIX. */  
  
#include <stdio.h>  
  
void filecopy(FILE *ifp, FILE *ofp);  
  
main(int ac, char *av[]) {  
    FILE *fp;  
    void filecopy(FILE *, FILE *);
```

# Example Cat

```
if (ac == 1)
    filecopy(stdin, stdout);
else
    while(--ac > 0)
        if (!(fp = fopen(*++av, "r"))){
            printf("Can't open %s\n", *av);
            return -1;
        } else {
            filecopy(fp, stdout);
            fclose(fp);
        }
    return 0;
}
```

# Example Cat

```
/* Copy the input file to the output file.  
   Copying stops on completion or error.  
*/  
void filecopy(FILE *in, FILE *out) {  
    int c;  
    while((c = getc(in)) != EOF) putc(c, out);  
}
```

# Unbuffered Files

- If the operating system allows it, files can be opened, read and written, and closed without associating buffers (streams) with them. This can be done in UNIX.

```
#include <fcntl.h>
```

```
int fd;  
int open(char *name, int flags, int perms);  
int creat(char *name, int perms);  
int read (int fd, char *buf, int n);  
int write(int fd, char *buf, int n);  
int close(int fd);
```

- There are many more OS specific functions.

# Programmer Coded Buffer

```
#include <syscalls.h>

/* Copy input to output. */
main {
    char buf[BUFSIZ];
    int n;

    while((n = read(0, buf, BUFSIZ))>0)
        if (write(1, buf, n) != n)
            return -1;
    return 0;
}
```

# Quiz

- What are the relative merits of buffered and unbuffered I/O for the compiler writer?
- How can Unix style file redirection be implemented?
- Which part of the C pre-processor could be implemented with the procedure *filecopy*?
- How can the C pre-processor strip comments from a file?

# Summary

- Three standard streams: *stdin*, *stdout*, *stderr*.
- Formatted read and write: *fscanf*, *fprintf*.
- Character handling: *getc*, *ungetc*, *putc*.
- There are many more facilities provided by C libraries for operating on buffered and unbuffered files, including sophisticated error checking functions.

# External Variables

- External variables, *extern*, are permanent globals that have lexical scope across all linked object files.
- External variables must be defined exactly once outside of any function.
- Definitions outside of a function default to *extern*, but only in the defining file.
- External variables must be declared in every function that wants access to them.
- Local identifiers default to *extern* within the scope of the defining file.

# Default External

```
/* This is FILE 1. */

# include <stdio.h>

/* Default external definition. */
char *message = "Hello World.\n";

/* Default external reference. */
int printm(void) {
    return printf("%s", message) == sizeof(message);
}
```

# Explicit Global External

```
/* This is FILE 2.  
   Must be linked with file 1.  
*/  
  
# include <stdio.h>  
  
/* Explicit external declaration. */  

```

# Explicit Local External

```
/* This is FILE 3.  
   Must be linked with file 1.  
*/  
  
# include <stdio.h>  
  
/* Explicit external declaration. */  

```

# Header Files

- Header files should be used to collect together the external declarations for a program.
- Header files should not define anything because multiple includes will make multiple copies of anything that is defined.

```
/* File: printmessage.h */  
  
#define M1 "Macros are jolly dee.\n"  
  
extern char *M2;
```

# Header File Inclusion

```
/* File: printmessage.c */

#include <stdio.h>
#include "printmessage.h"

main(void) {
    M2 = "So are pointers!\n";
    exit (printf("%s",M1)==sizeof(M1)) &&
        (printf("%s",M2)==sizeof(M2));
}
```

# Static

- Static objects are local to the file or function in which they are declared.
- Static objects are permanent.
- Static objects are zero on first access.

# Static

```
/* File 1. */

#include <stdio.h>

static char mysecret[] = "I love ...";

int printsecret(void) {
    static char *hersecret = "And she loves ...";
    printf("%s\n", mysecret);
    printf("%s\n", hersecret);
}
```

# Quiz

- How can *extern* and *static* best be used to implement abstract data types?
- How can *static* be used to implement a lexical analyser that returns one word read from a file each time it is called and a special marker *EOF* denoting End Of File when there is no more data to be read?
- What can pointers to functions be used for?
- Is C a block structured language?

# Summary

- The declaration *extern* makes a variable permanent and global to all object files that link to the variable.
- An *extern* variable must be defined in exactly one file.
- The declaration *static* makes a variable permanent and local to a file or function.
- On first access a *static* variable is zero.

# TAC - Three Address Code

Three Address Code (TAC) is made up of instructions with at most three addresses.

- “ $a = b \text{ op } c$ ” is used for most arithmetical assignments.
- “ $a = b \text{ op}$ ” is used for  $a = b++$ , and  $a = b--$ .
- “ $a = \text{op } c$ ” is used for  $a = ++c$ ,  $a = --c$ ,  $a = \&c$ , and “ $a = *c$ ”.
- “ $a = b$ ” is used for assignment.

You are free to make up any TAC instructions you like, but, for the assignment, it must be possible to implement them in C.

# TAC - Three Address Code

Here is one way to implement the function call “*foo(x,y,z)*”.

```
stack z
stack y
stack x
call foo
```

- These all use the pattern “*op c*”.
- Two address code, where the accumulator is used as a hidden address, is very popular.
- One and Zero address codes are theoretically possible, but are seldom, if ever, used.

# TAC - Three Address Code

- Procedural languages can be converted to TAC by a fairly straight forward parsing strategy.
- Many C operations go straight into TAC.
- TAC can be translated to assembler for most machines.
- TAC can be optimised by peephole optimisation.
- TAC can implement the Perspex machine where an address is a position in 4D space, i.e. each address is a 4D vector. But the vectors can be Turing incomputable.
- TWAC - TWo Address Code can implement a 3D Perspex Machine.

# Peephole Optimisation

- What can you say about this code by examining it one line at a time?

```
void main(void) {
    int i0, i1, i2, i3, i4, i5, i6, i7, i8, i9;
    i0 = 4 - 2;
    i1 = i0 / 2;
    i2 = i7 * i1;
    i3 = i2 * i0;
    i4 = i3 + i8;
    i5 = i2 * i0;
    i6 = i5 + i8;
    i9 = i4 * i6;
    out(i9);
}
```

# Input Output (I/O)

- If a program has no input it is a constant program. It can be optimised to literal statements that supply the output.
- If a program has no output it does nothing. It can be optimised to a null program.
- Delay loops cannot be optimised! Delays must be provided by kernel timers.
- I/O can be supplied by devices or volatile memory. Hence C's type qualifier *volatile*.

# Peephole Optimisation

- What can you say about this code by examining it one line at a time?

```
void main(void) {
    int i0, i1, i2, i3, i4, i5, i6, i7, i8, i9;
    in(i7);
    in(i8);
    i0 = 4 - 2;
    i1 = i0 / 2;
    i2 = i7 * i1;
    i3 = i2 * i0;
    i4 = i3 + i8;
    i5 = i2 * i0;
    i6 = i5 + i8;
    i9 = i4 * i6;
    out(i9);
}
```

# Peephole Optimisation

- Examine a small window of code (3 or 4 lines - more with search) and remove redundant commands or replace with lighter weight commands.
- Optimisation of machine code is not portable, optimisation of TAC is. Probably advisable to do both.
- Global parse trees can be transformed to TAC trees whose blocks of code can undergo local peep hole optimisation.
- Peep hole optimisation is the simplest kind of optimisation and can be combined with all other kinds.

# Common Sub-expression Elimination

- If the right hand sides (rhs) of two TAC instructions are lexically identical and none of the variables appear as intervening left hand sides (lhs) then copy lhs of first into rhs of second.

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i2 * i0;
i6 = i5 + i8;
i9 = i4 * i6;
out(i9);
```

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i3;
i6 = i5 + i8;
i9 = i4 * i6;
out(i9);
```

# Copy Propagation

- If TAC is of the form “ $a := b$ ” then replace all later occurrences of “ $a$ ” by “ $b$ ”. Perform common-sub expression elimination. Recurse.

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i3;
i6 = i5 + i8;
i9 = i4 * i6;
out(i9);
```

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i3;
i6 = i3 + i8;
i9 = i4 * i6;
out(i9);
```

# Copy Propagation

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i3;
i6 = i3 + i8;
i9 = i4 * i6;
out(i9);
```

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i3;
i6 = i4;
i9 = i4 * i6;
out(i9);
```

# Copy Propagation

```
in(i7);  
in(i8);  
i0 = 4 - 2;  
i1 = i0 / 2;  
i2 = i7 * i1;  
i3 = i2 * i0;  
i4 = i3 + i8;  
i5 = i3;  
i6 = i4;  
i9 = i4 * i6;  
out(i9);
```

```
in(i7);  
in(i8);  
i0 = 4 - 2;  
i1 = i0 / 2;  
i2 = i7 * i1;  
i3 = i2 * i0;  
i4 = i3 + i8;  
i5 = i3;  
i6 = i4;  
i9 = i4 * i4;  
out(i9);
```

# Dead Code Elimination

- If TAC is of the form “ $a := b \text{ op } c$ ” and there is no next reference to “ $a$ ” then delete the current TAC.

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i5 = i3;
i6 = i4;
i9 = i4 * i4;
out(i9);
```

```
in(i7);
in(i8);
i0 = 4 - 2;
i1 = i0 / 2;
i2 = i7 * i1;
i3 = i2 * i0;
i4 = i3 + i8;
i9 = i4 * i4;
out(i9);
```

# Constant Folding

- If rhs is a constant expression then evaluate it in the compiler and write result into rhs.
- Recursively perform all previous TAC optimisations.

```
in(i7);  
in(i8);  
i0 = 4 - 2;  
i1 = i0 / 2;  
i2 = i7 * i1;  
i3 = i2 * i0;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i0 = 2;  
i1 = i0 / 2;  
i2 = i7 * i1;  
i3 = i2 * i0;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Constant Folding

```
in(i7);  
in(i8);  
i0 = 2;  
i1 = i0 / 2;  
i2 = i7 * i1;  
i3 = i2 * i0;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i0 = 2;  
i1 = 2 / 2;  
i2 = i7 * i1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Constant Folding

```
in(i7);
in(i8);
i0 = 2;
i1 = 2 / 2;
i2 = i7 * i1;
i3 = i2 * 2;
i4 = i3 + i8;
i9 = i4 * i4;
out(i9);
```

```
in(i7);
in(i8);
i1 = 2 / 2;
i2 = i7 * i1;
i3 = i2 * 2;
i4 = i3 + i8;
i9 = i4 * i4;
out(i9);
```

# Constant Folding

```
in(i7);  
in(i8);  
i1 = 2 / 2;  
i2 = i7 * i1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i1 = 1;  
i2 = i7 * i1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Constant Folding

```
in(i7);  
in(i8);  
i1 = 1;  
i2 = i7 * i1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i1 = 1;  
i2 = i7 * 1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Constant Folding

```
in(i7);  
in(i8);  
i1 = 1;  
i2 = i7 * 1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i2 = i7 * 1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Algebraic Transformation

- If rhs has a simplifying transformation then apply it.

$$x = x + 0 = 0 + x$$

$$x = x - 0$$

$$x = x \times 1 = 1 \times x$$

$$x = x / 1$$

# Algebraic Transformation

```
in(i7);  
in(i8);  
i2 = i7 * 1;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i2 = i7;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Strength Reduction

- If an equivalent, but faster operation is known then substitute it for the slower one.
- Apply all previous TAC optimisations recursively.

$$x \text{ shiftright} = 2 \times x = x \times 2$$

$$x \text{ shiftleft} = x / 2$$

# Strength Reduction

```
in(i7);  
in(i8);  
i2 = i7;  
i3 = i2 * 2;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i2 = i7;  
i3 = i2 << 1;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Strength Reduction

```
in(i7);  
in(i8);  
i2 = i7;  
i3 = i2 << 1;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i2 = i7;  
i3 = i7 << 1;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Strength Reduction

```
in(i7);  
in(i8);  
i2 = i7;  
i3 = i7 << 1;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i3 = i7 << 1;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Packing Temporary Variables

- If a temporary is named in the lhs then replace it everywhere by the first free (dead) temporary.

```
in(i7);  
in(i8);  
i3 = i7 << 1;  
i4 = i3 + i8;  
i9 = i4 * i4;  
out(i9)
```

```
in(i7);  
in(i8);  
i0 = i7 << 1;  
i4 = i0 + i8;  
i9 = i4 * i4;  
out(i9);
```

# Packing Temporary Variables

```
in(i7);  
in(i8);  
i0 = i7 << 1;  
i4 = i0 + i8;  
i9 = i4 * i4;  
out(i9);
```

```
in(i7);  
in(i8);  
i0 = i7 << 1;  
i1 = i0 + i8;  
i9 = i1 * i1;  
out(i9);
```

# Packing Temporary Variables

```
in(i7);  
in(i8);  
i0 = i7 << 1;  
i1 = i0 + i8;  
i9 = i1 * i1;  
out(i9);
```

```
in(i7);  
in(i8);  
i0 = i7 << 1;  
i1 = i0 + i8;  
i2 = i1 * i1;  
out(i2);
```

# Packing User Variables

- Wherever a user variable occurs replace it by the first available (dead) temporary.

```
in(i7);  
in(i8);  
i0 = i7 << 1;  
i1 = i0 + i8;  
i2 = i1 * i1;  
out(i2)
```

```
in(i3);  
in(i8);  
i0 = i3 << 1;  
i1 = i0 + i8;  
i2 = i1 * i1;  
out(i2);
```

# Packing User Variables

```
in(i3);  
in(i8);  
i0 = i3 << 1;  
i1 = i0 + i8;  
i2 = i1 * i1;  
out(i2);
```

```
in(i3);  
in(i4);  
i0 = i3 << 1;  
i1 = i0 + i4;  
i2 = i1 * i1;  
out(i2);
```

# Apply Optimised Code

```
void main(void) {
    int i0, i1, i2, i3,
        i4, i5, i6, i7,
        i8, i9;
    i0 = 4 - 2;
    i1 = i0 / 2;
    i2 = i7 * i1;
    i3 = i2 * i0;
    i4 = i3 + i8;
    i5 = i2 * i0;
    i6 = i5 + i8;
    i9 = i4 * i6;
    out(i9);
}

void main(void) {
    int i0, i1, i2, i3,
        i4;
    in(i3);
    in(i4);
    i0 = i3 << 1;
    i1 = i0 + i4;
    i2 = i1 * i1;
    out(i2);
}
```

- Further peephole optimisations are possible.

# Further Optimisation

```
void main(void) {
    int i0, i1, i2, i3,
        i4;
    in(i3);
    in(i4);
    i0 = i3 << 1;
    i1 = i0 + i4;
    i2 = i1 * i1;
    out(i2);
}

void main(void) {
    int i0, i1;
    in(i0);
    in(i1);
    i0 = i0 << 1;
    i0 = i0 + i1;
    i0 = i0 * i0;
    out(i0);
}
```

# Summary

- Peephole optimisation is the simplest kind of optimisation.
- Algebraic transformations can be arbitrarily sophisticated.
- Strength reduction transformations are limited by hardware.
- Peephole optimisation can be applied to TAC and machine code.
- Peephole optimisation can be mixed with all optimisation strategies.

# Lexical Analysis

A lexical analyser reads the source code and returns:

- A *lexeme*, i.e. the characters that make up a symbol or word in the source language.
- The lexeme's *value*, e.g. the binary representation of a number, a pointer into the symbol table, or an error code.
- A *token*, i.e. a small integer that identifies at least the built in punctuation, symbols, and identifiers.
- A parser can select an action based on a token much faster than on a lexeme.

# Lexical Analysis

- Lexical analysers, or pre-processors, strip out comments. A comment can occur anywhere in a language so handling comments in the parser would double the size of the grammar.
- Lexical analysis is typically 40% of the compilation time.
- Unix supplies a tool called *lex* that compiles a lexical analyser from a specification of lexemes. The specification is written in extended regular expressions.
- Lexical analysers are easy to write by hand, but using *lex* is more productive.

# Tokens

- *yylval* contains the current character or integer value.
- *yytext* contains the current string.
- *ch* contains the next character.
- It is efficient to use the ASCII value of single character symbols as their token.
- Consequently all multi-character symbols have tokens with a value greater than 255.
- Parsing will be faster if each keyword has its own token.

# yylex.h

```
/* yytext.h

    Definitions for yylex.c
*/

/* Tokens.
*/

/* Punctuation. */
#define COMMA          ','
#define SEMICOLON     ';'
#define POINT         '.'
```

# yylex.h

```
/* Brackets. */
#define OPENPARENTHESIS      '('
#define CLOSEPARENTHESIS    ')'
#define OPENLATINBRACE      '{'
#define CLOSELATINBRACE     '}'
#define OPENANGLE           '<'
#define CLOSEANGLE          '>'

/* Operators. */
#define PLUS                 '+'
#define MINUS                '-'
#define TIMES                '*'
#define DIVIDE               '/'
#define ASSIGN               '='
```

# yylex.h

```
/* Key words. */
#define HASH                '#'
#define VOID                256
#define INT                 257
#define MAIN                258

/* Variable objects. */
#define IDENTIFIER          259
#define INTEGER             260

/* Catch all.
   Many people use the value zero for unknown,
   allowing a simple test of whether or not a lexeme
   is known. But, zero could, theoretically, clash
   with ASCII null.
*/
#define UNKNOWN             -2
```

# yylex.c

```
#include <stdio.h>
#include <ctype.h>

#include "yylex.h"
```

# Reserved Words

```
/* Return tokens identifying reserved words.  
   It would be sensible to use a symbol table instead  
   of hard coding the names.  
*/  
int reserved(char name[]) {  
    if (strcmp(name, "int" ) == 0) return INT;  
    if (strcmp(name, "void") == 0) return VOID;  
    if (strcmp(name, "main") == 0) return MAIN;  
    return IDENTIFIER;  
}
```

# yylex

```
int yylex() {
    int i;

    switch (ch) {

        /* Skip white space. */

        /* Skip comment or recognise division symbol
           "/".
        */

        /* Pass EOF.

           WARNING: EOF must not clash with any other
                   token. Usually EOF == -1.
        */
    }
```

# Single Character Symbols

```
/* Return single character tokens as their  
native code, excepting divide '/' which  
is confusable with a comment symbol and  
is handled above.
```

```
NOTE:      The native character set is  
           probably ASCII.
```

```
WARNING:  Single character tokens must not  
          clash with multi-character tokens.
```

```
*/
```

# Single Character Symbols

```
case ',' :  
case ';' :  
case '.' :  
case '(' :  
case ')' :  
case '{' :  
case '}' :  
case '<' :  
case '>' :  
case '+' :  
case '-' :  
case '*' :  
case '=' :  
case '#' :  
    yylval = ch;  
    ch = getchar();  
    return yylval;
```

# Literal Integers

```
/* Return an integer.
   WARNING : Arithmetic overflow will cause
             the lexical analyser to crash.
*/
case '0' :
case '1' :
case '2' :
case '3' :
case '4' :
case '5' :
case '6' :
case '7' :
case '8' :
case '9' :
    yylval = ch - '0';
    while (isdigit(ch = getchar()))
        yylval = yylval*10 + ch - '0';
    return INTEGER;
```

# Identifiers and Keywords

```
/* Return an identifier. Unrealistic. */  
    case 'A' :  
    case 'B' :  
    case 'C' :  
    case 'D' :  
    case 'E' :  
    case 'F' :  
    case 'G' :  
    case 'H' :  
    case 'I' :  
    case 'J' :  
    case 'K' :  
    case 'L' :  
    case 'M' :  
    case 'N' :  
    case 'O' :  
    case 'P' :
```

# Identifiers and Keywords

```
case 'Q' :  
case 'R' :  
case 'S' :  
case 'T' :  
case 'U' :  
case 'V' :  
case 'W' :  
case 'X' :  
case 'Y' :  
case 'Z' :  
case 'a' :  
case 'b' :  
case 'c' :  
case 'd' :  
case 'e' :  
case 'f' :  
case 'g' :  
case 'h' :
```

# Identifiers and Keywords

```
case 'i' :  
case 'j' :  
case 'k' :  
case 'l' :  
case 'm' :  
case 'n' :  
case 'o' :  
case 'p' :  
case 'q' :  
case 'r' :  
case 's' :  
case 't' :  
case 'u' :  
case 'v' :  
case 'w' :  
case 'x' :  
case 'y' :  
case 'z' :
```

# Identifiers and Keywords

```
for (yytext[0] = ch, i=1;
     isalnum(ch = getchar()));
     yytext[i++] = ch
);
yytext[i] = 0;
return reserved(yytext);
```

# Unexpected Character

```
        /* Unexpected character. */  
default  :  
        yylval = ch;  
        ch = getchar();  
        return UNKNOWN;  
    }  
}
```

# testlex.c

```
/* testlex.c

   Test lexical analyser defined in yylex.c
*/

#include <stdio.h>
#include <fcntl.h>

#include "yylex.h"
```

## testlex.c

```
void main(void) {
    int c;
    extern int  yylval;
    extern char yytext[];

    /* Redirect standard input. */

    /* Redirect standard output. */

    /* Initialise the lexical analyser. */
    lex_init();

    /* Test the lexical analyser. */
    printf("\n\nENTERING: testlex.c\n\n");
```

# testlex.c

```
while ( (c = yylex()) != EOF) {
    switch (c) {

        case INT          :
            printf("INT\n");
            break;

        case VOID         :
            printf("VOID\n");
            break;

        case MAIN         :
            printf("MAIN\n");
            break;
```

## testlex.c

```
case INTEGER    :
    printf("INTEGER %d\n", yylval);
    break;

case IDENTIFIER:
    printf("IDENTIFIER %s\n", yytext);
    break;

/* Single character not identified. */
case UNKNOWN   :
    printf("UNKNOWN %c %d\n", yylval,
          yylval
          );
    break;
```

## testlex.c

```
        /* Single character identified as
           symbol.
        */
        default      :
            printf("%c\n", c);
    }
}
printf("\n\nEXITING: testlex.c\n\n");

exit(NOERR);
}
```

# testlex.output

```
ENTERING: testlex.c
```

```
#  
IDENTIFIER include  
<  
IDENTIFIER stdio  
.  
IDENTIFIER h  
>  
VOID  
IDENTIFIER out  
(  
INT  
IDENTIFIER i  
)
```

# testlex.output

```
{  
IDENTIFIER printf  
(  
UNKNOWN " 34  
UNKNOWN % 37  
IDENTIFIER d  
UNKNOWN \ 92  
IDENTIFIER n  
UNKNOWN " 34  
,  
IDENTIFIER i  
)  
;  
}
```

# testlex.output

```
VOID
MAIN
(
VOID
)
{
INT
IDENTIFIER i0
,
IDENTIFIER i1
,
IDENTIFIER i2
,
IDENTIFIER i3
,
IDENTIFIER i4
,
```

# testlex.output

```
IDENTIFIER i5  
,  
IDENTIFIER i6  
,  
IDENTIFIER i7  
,  
IDENTIFIER i8  
,  
IDENTIFIER i9  
;
```

# testlex.output

```
IDENTIFIER i0
=
INTEGER 4
-
INTEGER 2
;
IDENTIFIER i1
=
IDENTIFIER i0
/
INTEGER 2
;
```

# testlex.output

```
IDENTIFIER i2
=
IDENTIFIER i7
*
IDENTIFIER i1
;
IDENTIFIER i3
=
IDENTIFIER i2
*
IDENTIFIER i0
;
```

# testlex.output

```
IDENTIFIER i4
=
IDENTIFIER i3
+
IDENTIFIER i8
;
IDENTIFIER i5
=
IDENTIFIER i2
*
IDENTIFIER i0
;
```

# testlex.output

```
IDENTIFIER i6  
=  
IDENTIFIER i5  
+  
IDENTIFIER i8  
;  
IDENTIFIER i9  
=  
IDENTIFIER i4  
*  
IDENTIFIER i6  
;
```

# testlex.output

```
IDENTIFIER out
(
IDENTIFIER i9
)
;
}
```

```
EXITING: testlex.c
```

# Summary

- Lexical analysers can be implemented by hand, but the analysers implemented by a lexical analyser generator (such as *lex*) are easier to extend and often execute more quickly.
- Lexical analysers typically return a *token*, *lexeme*, and *value*.
- Lexical analysis typically takes up 40% of compilation time.
- Comments are usually stripped by a pre-processor or else by the lexical analyser, because passing them to the compiler would usually double the size of the language grammar.

# Production Rules

Production rules:

- Replace the left hand side of a rule with the right hand side (lhs  $\rightarrow$  rhs).
- Are Turing equivalent.

# Production Rules

$lhs \rightarrow rhs$

$rhs \rightarrow non\text{-terminals } \mathbf{terminals}$

$rhs \rightarrow this \mid that$

$rhs \rightarrow this$

$that$

$rhs \rightarrow this$

$that$

$\epsilon$

$rhs:$

$this$

$that$

$\epsilon$

# Expression

*expr:*

*term expr'*

*expr':*

**+** *term expr'*

$\epsilon$

*term:*

*factor term'*

*term':*

**\*** *factor term'*

$\epsilon$

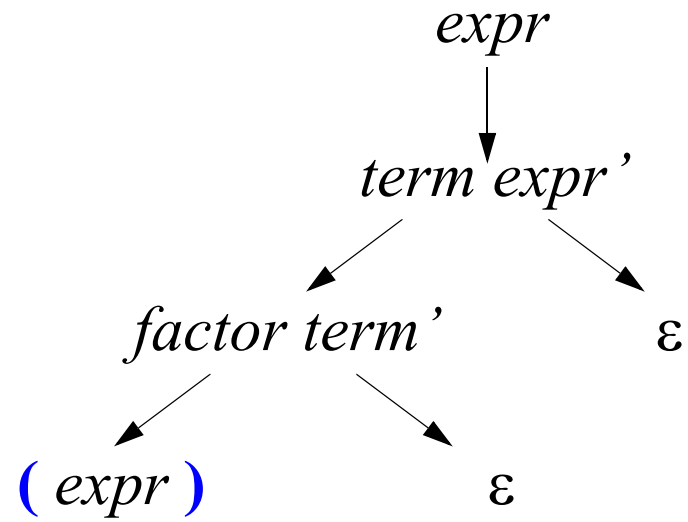
*factor:*

**(** *expr* **)**

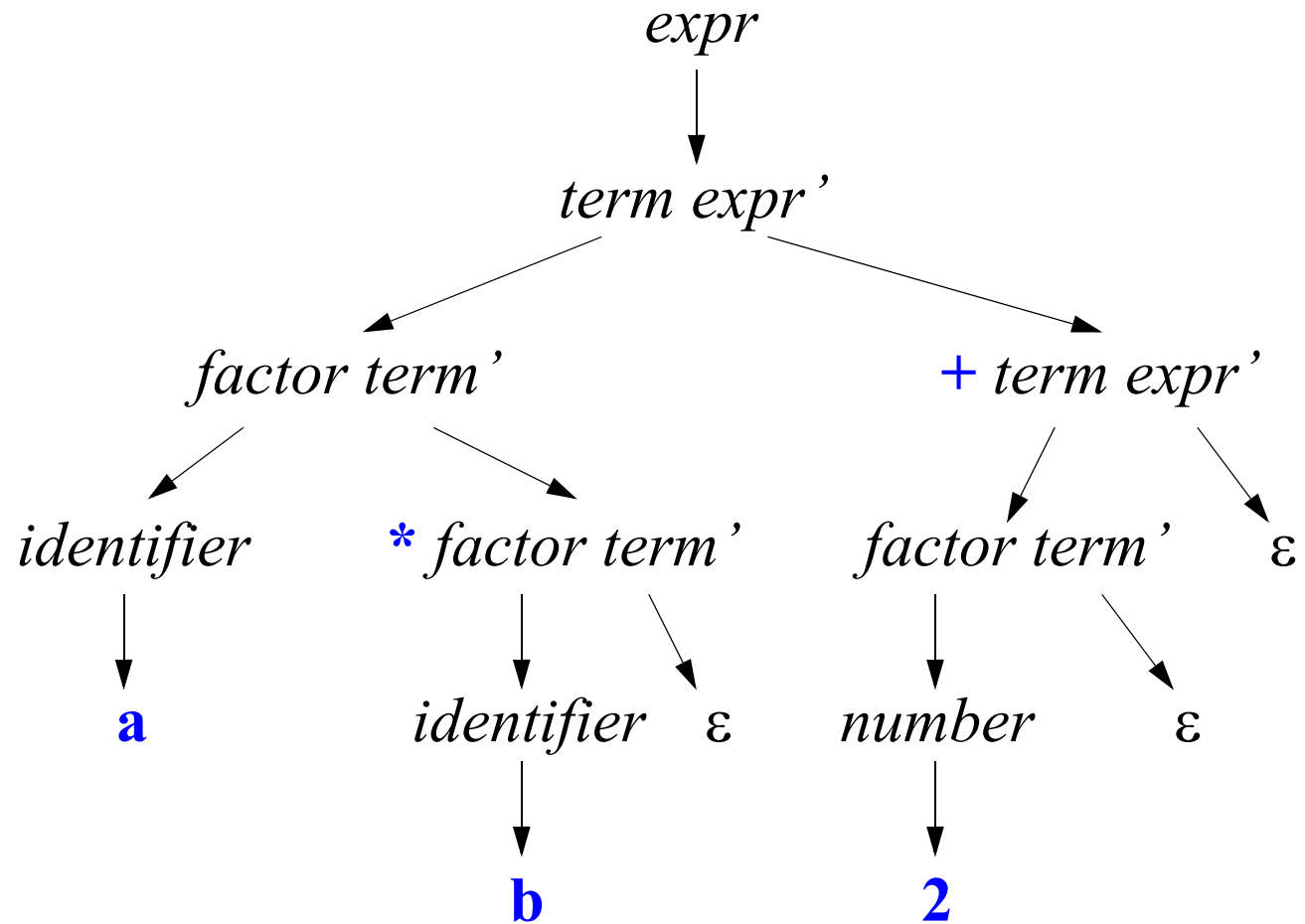
*identifier*

*number*

# Parsing (a\*b + 2)



# Parsing (a\*b + 2)



# Grammar

- Grammars always start from the Sentence symbol ( $S$ ).
- Production rules ( $P$ ) convert the left hand side into the right hand side ( $lhs \rightarrow rhs$ ).
- Non-terminal symbols ( $N$ ) are variables in a grammar. They appear in the branches of a parse tree.
- Terminal symbols ( $T$ ) are constants in the grammar. They appear in the leaves of a parse tree.
- No symbols are common to  $N$  and  $T$  ( $N \cap T = \emptyset$ ) Thus the symbols of the grammar cannot be confused with the symbols of the language.

# Language

- A grammar  $G$  is a 4-tuple  $\langle S, P, N, T \rangle$ .
- The Language  $L(G)$  is the set of sentences derivable from  $S$  in  $G$ .
- There is a formal theory of languages which is very useful to compiler writers. However, in this course we will use the results of language theory without examining any proofs.
- Noam Chomsky was a linguist who introduced a classification scheme for language types and was the first to propose a psychologically plausible theory of human language.

# Chomsky Type 0 Language

TYPE 0:  $a \rightarrow b$

- Free grammars.
- The symbols ' $a$ ' and ' $b$ ' represent arbitrary strings in the vocabulary  $V = N \cup T$ . That is  $a, b \in V$ .
- The string ' $a$ ' is not null ( $\epsilon$ ).

Is English a Type 0 language?

# Chomsky Type 1 Language

TYPE 1:  $abc \rightarrow adc$

- Context sensitive grammars.
- All of the symbols in the strings 'a', 'b', 'c', 'd' are in the vocabulary  $V$ . That is  $a, b, c, d \in V$ .
- The string  $b$  is a single non-terminal symbol. That is  $b \in N$ .
- The string 'd' is not null ( $\epsilon$ ).

Are computer languages context sensitive?

# Chomsky Type 2 Language

TYPE 2:  $a \rightarrow b$

- Context-free grammars.
- All of the symbols in the strings ' $a$ ' and ' $b$ ' are in the vocabulary  $V$ . That is  $a, b \in V$ .
- The string ' $a$ ' is a single non-terminal symbol. That is  $a \in N$ .
- The string ' $b$ ' is not null ( $\epsilon$ ).

All type 2 languages can be parsed deterministically and most computer-language parsers are type 2.

# Chomsky Type 3 Language

TYPE 3:  $a \rightarrow b \mid a \rightarrow bc$

- Finite, or regular grammars.
- The strings ‘ $a$ ’ and ‘ $c$ ’ are single non-terminal symbols. That is  $a, c \in N$ .
- The string ‘ $b$ ’ is a single terminal symbol. That is  $b \in T$

Most lexical analysers for computer languages are Type 3. Type 3 languages can be parsed extremely efficiently.

# Chomskian Languages

The lower numbered Chomskian types contain the higher numbered ones.

- Why are lexical analysers written separately from parsers?
- Why is semantic checking done separately from parsing?
- Is error-checking an example of semantic checking?
- What type of language would a compiler-compiler have to be?

# Summary

- A grammar can be written in production rules.
- A production rule is of the general form lhs  $\rightarrow$  rhs.
- A language is the set of sentences produced by a grammar.
- Lexical analysers are usually Chomsky Type 3.
- Compilers are usually Chomsky Type 2, but with some Chomsky Type 1 error checking.

# Recogniser

- A recogniser parses source code, but does not generate any output code. It recognises whether the source code is syntactically correct or not.
- A compiler is a recogniser to which (output) code generation is added.

# Recursive Descent Parser

- A parser can be implemented by having one recursive procedure for each non-terminal symbol in the grammar. This is called a *recursive descent parser*.
- If a right-recursive grammar is arranged so that at each choice point all of the possible production rules begin with a different symbol on the rhs then a recursive descent parser with one symbol lookahead need never backtrack. That is, it is deterministic.
- Deterministic parsers are generally more efficient than non-deterministic parsers.

# Statements

*statements:*

*statement statements'*

$\epsilon$

*statements':*

*;* *statements*

$\epsilon$

*statement:*

*function\_call*

*declaration*

*assignment*

# Function Call

*function\_call:*  
**in** ( *identifier* )  
**out** ( *identifier* )

# Declaration

*declaration:*

**int** *identifier declaration'*

*declaration':*

**,** *identifier declaration'*

$\epsilon$

# Assignment

*assignment:*  
*identifier = expression*

# Non-Terminal Tokens

- We add some non-terminal tokens so that we can use *checkfor* to generate syntax errors relating to grammatical terms.
- These tokens are:

*STATEMENT*

*EXPRESSION*

*TERM*

*FACTOR*

# Error Messages

- In a recursive descent parser error messages tend to be very specific because they are triggered by terminal symbols.
- Error messages run along  $\epsilon$ -productions so whilst an error message will refer to the correct terminal symbol the diagnosis of the fault might be in a very distant production. This is why recursive descent parsers tend to generate a generic “syntax error” message.

# Error Messages

- A more sophisticated error strategy is to extend the language grammar so that error messages are generated in the same way as code is generated.
- Global error messages can make use of the partial parse tree and can attempt to generate a continuation parse tree by throwing away tokens until some synchronisation symbol, such as a keyword, is encountered.
- Error messages can be optimised, say to propose the minimum discovered change to the code which would make it parsable.

# Function Call

```
/* Consume a call to "in". */  
void call_in( ) {  
    checkfor(OPENPARENTHESIS);  
    checkfor(IDENTIFIER);  
    checkfor(CLOSEPARENTHESIS);  
}
```

```
/* Consume a call to "out". */  
void call_out() {  
    checkfor(OPENPARENTHESIS);  
    checkfor(IDENTIFIER);  
    checkfor(CLOSEPARENTHESIS);  
}
```

# Declaration

```
/* Consume a declaration of integer variables. */  
void declaration_prime() {  
    if (nextsymbol == COMMA) {  
        nextsymbol = yylex();  
        checkfor(IDENTIFIER);  
        declaration_prime();  
    }  
}  
  
void declaration() {  
    checkfor(IDENTIFIER);  
    declaration_prime();  
}
```

# Statements

```
/* Prototypes to avoid mutual use-before
   declaration error. */
void statements(void);
void statement_prime(void);
```

```
/* Consume statements. */
void statements() {
    statement();
    statement_prime();
}
```

```
/* Continuation of statements consumer. */
void statement_prime() {
    if (nextsymbol == SEMICOLON) {
        nextsymbol = yylex();
        statements();
    }
}
```

# } Statement

```
/* Consume one statement. */
void statement() {

    switch(nextsymbol) {

        case INT          :
            nextsymbol = yylex();
            declaration();
            break;
```

# Statement

```
case IDENTIFIER :

    /* Simulate symbol table handling.
    Notice that if statement drops through
    to default.
    */
    if (strcmp("in", yytext) == 0) {
        nextsymbol = yylex();
        call_in();
    }
    else if (strcmp("out", yytext) == 0) {
        nextsymbol = yylex();
        call_out();
    }
    else
        assignment();
    break;
```

# Statement

```
        /* Switch must have empty default to
           handle epsilon.
        */
        default      :
            break;
    }
}
```

# Assignment

```
/* Consume an assignement. */  
void assignment() {  
    checkfor(IDENTIFIER);  
    checkfor(ASSIGN);  
    expression();  
}
```

# Expression

```
/* Prototypes for consumers. */  
void expression(void);  
void expression_prime(void);  
void term(void);  
void term_prime(void);  
void factor(void);
```

# Expression

```
/* Consume an expression. */
void expression() {
    term();
    expression_prime();
}

void expression_prime() {
    if (nextsymbol == PLUS || nextsymbol == MINUS)
    {
        nextsymbol = yylex();
        term();
        expression_prime();
    }
}
```

# Term

```
/* Consume a term. */
void term() {
    factor();
    term_prime();
}

void term_prime() {
    if (nextsymbol == TIMES ||
        nextsymbol == DIVIDE
    ) {
        nextsymbol = yylex();
        factor();
        term_prime();
    }
}
```

# Factor

```
/* Consume a factor. */
void factor() {

    switch(nextsymbol) {

        case OPENPARENTHESIS :
            nextsymbol = yylex();
            expression();
            checkfor(CLOSEPARENTHESIS);
            break;
```

# Factor

```
case IDENTIFIER:
```

```
    /* Simulate separate name spaces for  
       variables and functions.
```

```
    */
```

```
    if (strcmp("in", yytext) == 0 ||  
        strcmp("out", yytext) == 0
```

```
    )
```

```
        /* Notice the use of a non-terminal  
           token in an error message.
```

```
        */
```

```
        checkfor(FACTOR);
```

```
    else
```

```
        nextsymbol = yylex();
```

```
    break;
```

# Factor

```
case INTEGER:
    nextsymbol = yylex();

    /* Notice the defensive "break". */
    break;

default:
    checkfor(FACTOR);
    break;
}
}
```

# Checkfor

```
/* Checks that nextsymbol is the expected symbol and
   consumes it. Otherwise prints a syntax error
   message.
```

```
*/
```

```
void checkfor(int token) {
    if (nextsymbol != token) {
        switch (nextsymbol) {
            case EOF      :
                fprintf(stderr,
                    "Error: expected %s found EOF\n",
                    printableform(token)
                );
                break;
```

# Checkfor

```
case INTEGER :
    fprintf(
        stderr,
        "Error: expected %s found INTEGER %d\n",
            printableform(token),
            yylval
    );
    break;
```

# Checkfor

```
default      :
    if      (nextsymbol < 256 )
        fprintf(
            stderr,
            "Error: expected %s found %s %c\n",
            printableform(token),
            printableform(nextsymbol),
            yylval
        );
```

# Checker

```
        else
            fprintf(
                stderr,
                "Error: expected %s found %s %s\n",
                printableform(token),
                printableform(nextsymbol),
                yytext
            );
        }
        exit(SYNTAXERR);
    }
    nextsymbol = yylex();
}
```

# Source

```
/* Error-free source file to test the compiler. */

void main(void) {

    /* Programmer variables. */
    int a, b, c, tmp1, tmp2;

    /* Input. */
    in(a); in(b);

    /* Arithmetic. */
    tmp1 = 4 + 2;
    tmp2 = a * tmp1 / 2;
    c    = (tmp1*tmp2+b)*(tmp1*tmp2+b);

    /* Output. */
    out(c)
}
```

# Summary

A recursive descent parser:

- Uses a right-recursive grammar;
- Has one recursive subroutine for every non-terminal symbol in the grammar. (It might have additional helper subroutines.)
- Is easy to implement by hand.

# Compiler

- A compiler is a recogniser with code generation.
- The compiler is shown in C Pseudocode, not C code.

# Assignment

*assignment:*  
*identifier = expression*

# Expression

*expr:*

*term expr'*

*expr':*

**+** *term expr'*

$\epsilon$

*term:*

*factor term'*

*term':*

**\*** *factor term'*

$\epsilon$

*factor:*

**(** *expr* **)**

*identifier*

*number*

# Assignment

```
int tempnumber = -1;

/* Consume an assignement. */
void assignment() {

    /* Parse. */
    checkfor(IDENTIFIER);
    checkfor(ASSIGN);
    expression();

    /* Generate code. */
    printf("%s = t%d;\n", identifier, tempnumber);
}
```

# Expression

```
/* Consume an expression. */  
void expression() {  
    term();  
    expression_prime();  
}
```

# Expression'

```
void expression_prime() {  
  
    /* Epsilon on non-existent else branch. */  
    if (nextsymbol == PLUS) {  
        arg1 = tempnumber;  
  
        /* Parse. */  
        term();  
        expression_prime();  
  
        /* Generate code. */  
        arg2 = tempnumber;  
        printf("t%d = t%d + t%d;\n", ++tempnumber,  
            arg1, arg2 );  
    }  
}
```

# Term

```
/* Consume a term. */  
void term() {  
    factor();  
    term_prime();  
}
```

# Term'

```
void term_prime() {  
  
    /* Epsilon on non-existent else branch. */  
    if (nextsymbol == TIMES ) {  
        arg1 = tempnumber;  
  
        /* Parse. */  
        factor();  
        term_prime();  
  
        /* Generate code. */  
        arg2 = tempnumber;  
        printf("t%d = t%d * t%d;\n", ++tempnumber,  
            arg1, arg2 );  
    }  
}
```

# Factor

```
/* Consume a factor. */
void factor() {

    switch(nextsymbol) {

        case OPENPARENTHESIS :
            nextsymbol = yylex();
            expression();
            checkfor(CLOSEPARENTHESIS);
            break;
```

# Factor

```
case IDENTIFIER:
    /* Generate code. */
    printf("t%d = %s;\n", ++tempnumber,
           identifier);
    break;

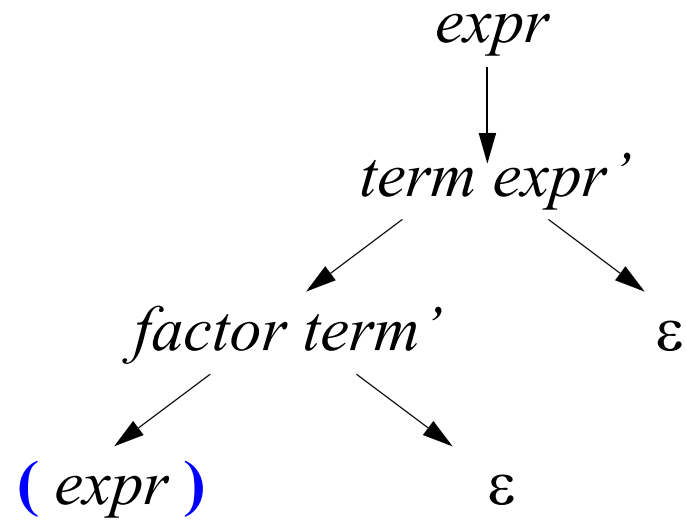
case INTEGER:
    /* Generate code. */
    printf("t%d = %s;\n", ++tempnumber,
           integer);
    break;

default:
    /* Generate error message. */
    checkfor(FACTOR);
    break;
}
}
```

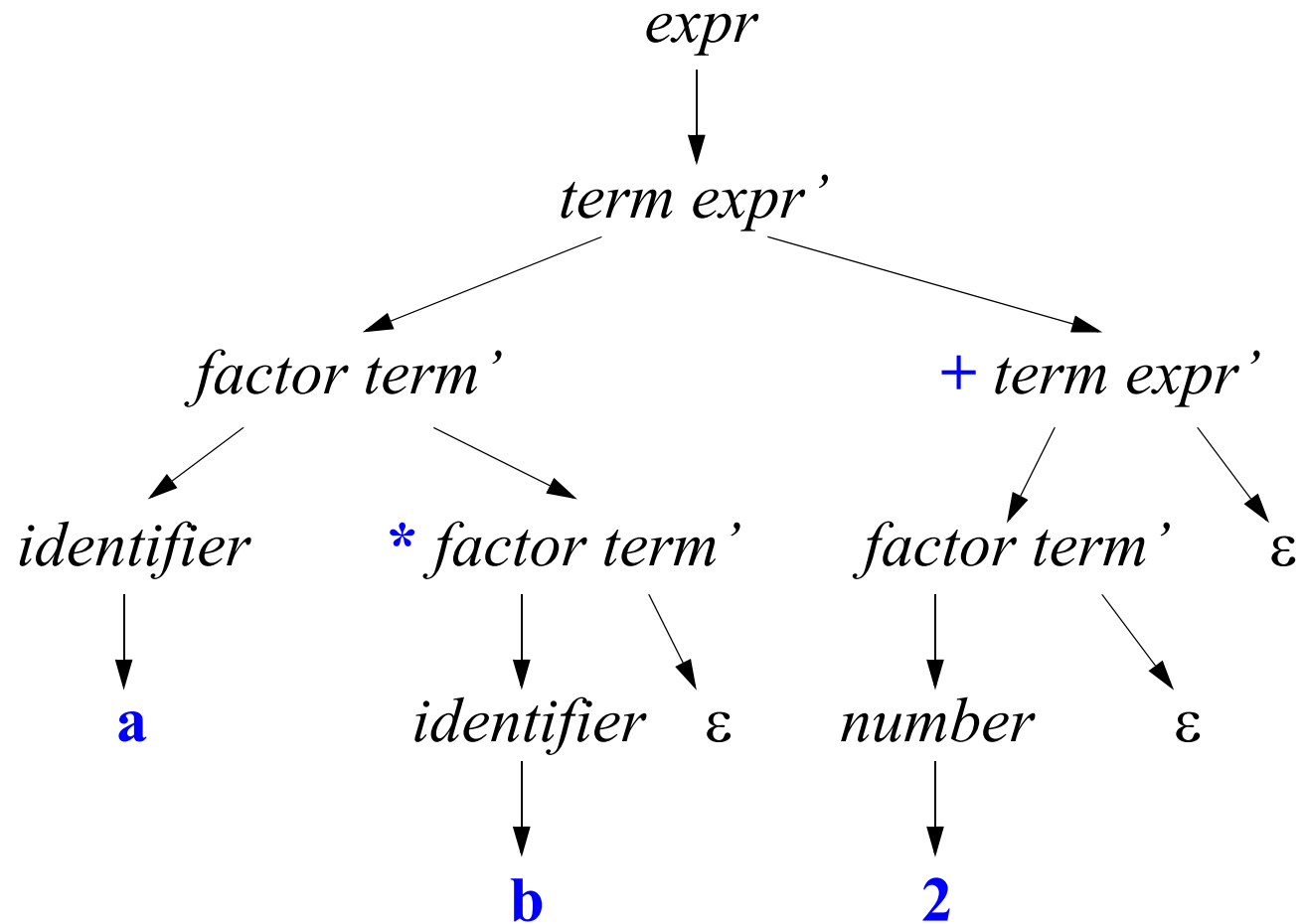
# Compile: $x = (a * b + 2)$

```
t0 = a;  
t1 = b;  
t2 = t0 * t1;  
t3 = 2;  
t4 = t2 + t3;  
x = t4;
```

# Parsing (a\*b + 2)



# Parsing (a\*b + 2)



**Compile:  $x = 2 + 3 + 4 * 5 * 6$**

```
t0 = 2;  
t1 = 3;  
t2 = 4;  
t3 = 5;  
t4 = 6;  
t5 = t3 * t4;  
t6 = t2 * t5;  
t7 = t1 + t6;  
t8 = t0 + t7;  
x = t8;
```

- Notice that this generated code is right associative.

# Quiz

- Do C operators associate left-to-right or right-to-left?
- Does C specify the order of evaluation within an expression?
- Does the direction of associativity make any difference to the evaluation of arithmetic expressions?
- In what C expressions does the direction of associativity make a difference?
- How can source code be written to force the order of evaluation?

# Quiz

- How could the above code generation be changed so that the C operators associate left-to-right?
- The above code is generated on-the-fly. How could the above compiler be changed so that it compiles code into a tree?
- What are the relative merits of on-the-fly code generation and code-tree generation?
- How can parse trees and code trees be unified and why would it be useful to unify them?

# Summary

There are two main strategies for generating code:

- On-the-fly generation of a list of code;
- Generation of a code tree with subsequent flattening of the tree into a list of code.
- The most important thing is to generate correct code. It can be optimised later.
- Tree based methods are more adaptable and support more effective optimisations.
- On-the-fly methods generate code more quickly.

# Code Generation

The simplest way to add code generation to a parser is:

- Define, in the compiler, templates of TAC code to perform basic operations.
- Have the parser build a symbol table of all of the objects in the source code.
- During compilation use data from the symbol table and the currently active parsing subroutine to generate TAC by filling out the templates of TAC code.
- It might be necessary to have more than one pass through the source code to achieve the above.

# Explicit or Implicit Parse Tree

- If multiple passes through the source code are necessary it is probably more efficient to store the parse as an explicit tree and operate on this data structure to do code generation.
- However, explicit parse trees can consume a large amount of memory so doing code generation on-the-fly without a parse tree might be the only practical solution.
- (The assessment require only very simple, on-the-fly code generation.)

# Static Variables

- Static variables are allocated memory in a heap that is present throughout program execution.
- A repeater called by the recogniser can add a static variable to the heap by returning a pointer to the next free block of memory in the heap and recording this pointer in the variable's symbol table entry.
- The initial value of a variable can be stored in the symbol table during parsing. This value is then used to initialise the variables in the heap during code generation.
- (In the assessment you need only declare the variables; the host C compiler will do the memory allocation.)

# Local Variables

- Local variables are allocated on a stack.
- Temporary variables needed by a compiler are almost always local variables.
- A *startblock* TAC instruction is needed to indicate that a new block of code is about to be entered so that all declarations of variables are local to that block.
- An *endblock* TAC instruction is needed to indicate that a block of code has been executed. The top-of-stack pointer can then be decremented to its position at the previous *startblock* so that the stack is reused.

# Local Variables

- The *startblock* and *endblock* instructions can be put on the stack if on-the-fly stack manipulation is wanted.
- (In the examples given in this lecture course it is sufficient to make all variables static, because we have used only one block of code in *main*.)

# Variable Declaration

The parser:

- Builds the symbol table and records the types of variables there.
- Checks that all variables referenced in the source code are declared in the symbol table.
- Checks that assignments involve valid types.
- (In the examples given in this lectures it is sufficient to make all variables integer, because we have used only integer variables. Thus, the only checking to be done is that all variables referenced in the source code are declared in the source.)

# Assignment to a Variable

The result of an arithmetical expression, for example:

$$a = b + c * (d + e)$$

can be assigned to a variable as follows.

- Generate code at the leaves of the parse to perform one arithmetical operation and store the result in a temporary variable.

```
t0 = d + e;  
t1 = c * t0;  
t2 = b + t1;
```

# Assignment to a Variable

- Generate code at the root of the expression tree to assign the result from the last temporary variable into the source variable.

```
a = t3;
```

- One way to do this is to pass an argument back up the recursive parsing subroutines which is the number  $n$  of the temporary variable  $tn$  used in the temporary assignments generated so far.
- It is simplest to leave temporary packing to the optimiser.

# Assignment to a Variable

- If a stack machine is the target then no temporary variable names are needed. Simply push intermediate values onto the stack and pop them off when they are operated on.
- It is often efficient to keep separate stacks of the operators and arguments.

# Backpatching

When creating a jump forward the compiler does not know the address of the jumped-to label until *after* the code has been generated and optimised! How can the compiler handle this?

- Generate a dummy address in a *goto* and over-write it with the true address when it is known.
- The linker also uses back patching to link to *extern* variables and functions defined in binary files different from the binary files in which they are called.
- (In the assessment the host C compiler can handle a *goto label* so no backpatching is needed.)

# Subroutine Call

All of the arguments in a given subroutine can be held in a *struct* known as a “stack frame”.

- Store the current top-of-stack pointer.
- Increment the top-of-stack pointer by the size of the stack frame.
- Initialise the input variables in the stack frame.
- Store the return address.
- Jump to the address of the called function.

# Subroutine Return

- Copy output variables from the stack frame into their locations.
- Restore the top-of-stack pointer.
- Jump to the return address.
- (In the assessment no subroutine call and return is needed because it is sufficient to use just a *main*.)

## Subroutine TAC

- Ideally one would supply TAC instructions for a subroutine call. For example:

```
Printf ("%d %d %d %d\n", 1, 2, 3, 4);
```

- Becomes:

```
stack 4
```

```
stack 3
```

```
stack 2
```

```
stack 1
```

```
stack pointer to "%d %d %d %d\n"
```

```
call printf
```

# Subroutine TAC

- And then implement a back end that converts the TAC into assembler for the host machine.
- In the assessment a function call could either be a call to the function as supported by the host C compiler or,
  - More instructively, a call to inline assembler.
- If you generate inline assembler code then the assignment should say which assembler and chip set you are targeting.

# Summary

The course work assessment can be done by:

- Having the recursive parsing subroutines generate static temporary variables for the intermediate results in an arithmetical expression.
- Recording the names of declared variables in a symbol table – an array of strings similar to *argv* will suffice – and reporting the use of any undeclared variables.

# Summary

In addition we have discussed:

- Explicit parse trees.
- Local variables using a stack.
- Subroutine call and return.
- Backpatching in the compiler and linker.
- This is sufficient to implement code generation for C and similar languages on standard target machines.

# Grammar Practice

- Grammars always start from the Sentence symbol ( $S$ ).
- Production rules ( $P$ ) convert the left hand side into the right hand side ( $lhs \rightarrow rhs$ ).
- Non-terminal symbols ( $N$ ) are variables in a grammar. They appear in the branches of a parse tree.
- Terminal symbols ( $T$ ) are constants in the grammar. They appear in the leaves of a parse tree.
- No symbols are common to  $N$  and  $T$  ( $N \cap T = \emptyset$ ) Thus the symbols of the grammar cannot be confused with the symbols of the language.

# Language

- A grammar  $G$  is a 4-tuple  $\langle S, P, N, T \rangle$ .
- The Language  $L(G)$  is the set of sentences derivable from  $S$  in  $G$ .
- There is a formal theory of languages which is very useful to compiler writers. However, in this course we will use the results of language theory without examining any proofs.
- Noam Chomsky was a linguist who introduced a classification scheme for language types and was the first to propose a psychologically plausible theory of human language.

# Quiz

Develop a right-recursive grammar of each Chomskian type that describes C's literal numbers.

# Chomsky Type 0 Language

TYPE 0:  $a \rightarrow b$

- Free grammars.
- The symbols ' $a$ ' and ' $b$ ' represent arbitrary strings in the vocabulary  $V = N \cup T$ . That is  $a, b \in V$ .
- The string ' $a$ ' is not null ( $\epsilon$ ).

Is English a Type 0 language?

# Chomsky Type 1 Language

TYPE 1:  $abc \rightarrow adc$

- Context sensitive grammars.
- All of the symbols in the strings 'a', 'b', 'c', 'd' are in the vocabulary  $V$ . That is  $a, b, c, d \in V$ .
- The string  $b$  is a single non-terminal symbol. That is  $b \in N$ .
- The string 'd' is not null ( $\epsilon$ ).

Are computer languages context sensitive?

# Chomsky Type 2 Language

TYPE 2:  $a \rightarrow b$

- Context-free grammars.
- All of the symbols in the strings ' $a$ ' and ' $b$ ' are in the vocabulary  $V$ . That is  $a, b \in V$ .
- The string ' $a$ ' is a single non-terminal symbol. That is  $a \in N$ .
- The string ' $b$ ' is not null ( $\epsilon$ ).

All type 2 languages can be parsed deterministically and most computer-language parsers are type 2.

# Chomsky Type 3 Language

TYPE 3:  $a \rightarrow b \mid a \rightarrow bc$

- Finite, or regular grammars.
- The strings ' $a$ ' and ' $c$ ' are single non-terminal symbols. That is  $a, c \in N$ .
- The string ' $b$ ' is a single terminal symbol. That is  $b \in T$

Most lexical analysers for computer languages are Type 3. Type 3 languages can be parsed extremely efficiently.

# Summary

- Type 2 languages can be parsed deterministically.
- Type 3 languages can be parsed very quickly.
- Modularising the lexical analyser as a type 3 language allows lexical analysis to be performed very quickly.

# Lex

- Lex is a lexical analyser generator that reads a specification of lexemes expressed in extended regular expressions and writes a C program that implements a lexical analyser for the specified lexemes.
- Lex is part of Unix. It is highly compatible with C.
- Flex is a variant of Lex supplied by Gnu. It is highly compatible with C<sup>++</sup>.
- Flex is available without charge for use on Unix and Microsoft operating systems. See, for example:

[www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html](http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html)

# Regular Expressions

- Chomsky Type 3 grammars are finite, or regular grammars.
- Regular grammars can be specified by regular expressions.
- The regular expressions are: union, concatenation, and Kleene closure.
- The regular expressions are usually extended by the addition of variants that make it easier for a human to specify regular grammars.

# Union ( $\cup$ )

Union “ $\cup$ ” is set union and is used to combine sets of elements.

*digit*: **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

*alpha*: **a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q  
| r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G  
| H | I | J | K | L | M | N | O | P | Q | R | S | T | U |  
V | W | X | Y | Z | \_**

*alphanumeric*: *digit*  $\cup$  *alpha*

# Concatenation (.)

Concatenation “.” is the adding together of strings.

**fire . water -> firewater**

# Kleene Closure (\*)

Kleene closure “\*” is the concatenation of a language element zero or more times.

*g1*: **Tarry**

*g2*: **Bum**

*g1*\* . *g2*:

**Bum**

**TarryBum**

**TarryTarryBum**

**TarryTarryTarryBum**

**TarryTarryTarryTarryBum**

# Regular Expression

- Which tune will this regular expression match?

*g1*: **Tarry**

*g2*: **Bum**

*g3*: **Tarra**

$( (g1.g2)^* . g2^* )^* . g3 . g1 . g2$

# Regular Expressions

- Practical implementations of regular expression, such as the search commands in Unix's "vi" or "grep", and specifications in "lex" and "flex", have additional operators that perform variants of the regular expressions.

# Lex Expressions

- “\” escapes a meta character, e.g. “\n” is newline and “\\*” is a literal “\*”.
- “.” any one character except newline “\n”.
- “?” zero or one repetitions of the preceding expression.
- “\*” Kleene closure, i.e. zero or more repetitions of the preceding expression.
- “+” positive closure, i.e. one or more repetitions of the preceding expression.
- “|” alternative expressions e.g. **a|b** is **a** or else **b**.

# Lex Expressions

- “[ ]” bracket a character class, e.g. the digits are specified as [0123456789].
- Within a character class “-” denotes a range, e.g. [0-9] is [0123456789].
- Within a character class “^” denotes all of the characters except those in the character class, e.g. [^0-9] is every character that is not a digit.
- If “-” or “[ ]” is the first character of a character class it is interpreted literally so the class can contain “-” and “[ ]”.

# Lex Expressions

- Within a character class meta characters are treated as literals, except for “\” so the class can contain special characters.
- “^” beginning of line when occurring in an expression.
- “\$” end of line when occurring as the last character of an expression.
- “{n1,n2}”, where n1 and n2 are numbers such that  $n1 \geq 0$  and  $n1 \leq n2$ , indicates a range of repetitions, e.g.  $A\{1,3\}$  is A or AA or AAA.
- “”” bracket characters treated as literals, except “\”, e.g. “\*+\n” is a literal \*, a literal +, and newline.

# Lex Expressions

- "()" brackets a sequence of expressions, e.g. (AB)? is the null sequence or the sequence AB.

# Specifying Numbers

- A digit:  $[0-9]$
- A non-negative integer with at least one digit:  $[0-9]^+$
- An integer with an optional, unary minus:  $-?[0-9]^+$
- A non-negative decimal number with at least one digit before and after the decimal point:  $[0-9]^+\.[0-9]^+$
- A non-negative integer or a non-negative decimal number:  $([0-9]^+)|([0-9]^+\.[0-9]^+)$
- A real number, i.e. an integer or decimal:  
 $-?((([0-9]^+)|([0-9]^+\.[0-9]^+)))$

# Specifying Numbers

- An exponent written as lower or upper case "e" followed by an optional sign followed by a non-null digit string:  $[eE][-+]?[0-9]^+$
- A real number with at most one exponent:  
 $-?([0-9]^+)|([0-9]^+\.[0-9]^+)([eE][-+]?[0-9]^+)?$

# Lex Programs

```
%{ /* Definition section copied as is to the program
      generated by lex. */
```

```
}%
```

```
%%
```

Rules section. Treats comments psychopathically.  
Rules are a pattern (specified as an extended regular expression) followed by white space and an action. An action is C code bracketed by "{}".  
Best to put comments in the action like this

```
{ /* Comment */ }
```

```
%%
```

User subroutine section copied as is to the end of the program generated by lex. If the actions do not have an explicit "return" then yylex processes the whole of the input.

```
main () {
    yylex();
}
```

# Number Recogniser

- A lex program that reads the entire input, ignores white space, prints “number” on a new line when it recognises a number, and prints any unrecognised character as is without inserting a newline.

```
%%  
[\\n\\t]      ;  
-?(([0-9]+)|([0-9]+\\. [0-9]+)([eE] [-+]? [0-9]+)? ) \\  
{printf("\\nnumber\\n");}  
•      ECHO;  
%%  
main() {  
    yylex();  
}
```

# Summary

- A regular expression is any well formed formula over union, concatenation, and Kleene closure.
- All Chomsky Type 3 languages can be described by regular expressions.
- Lex generates a lexical analyser in C given a specification of lexemes expressed in extended regular expressions.

# Transformations

The productions in a grammar can be systematically transformed to:

- remove ambiguity,
- change between left-recursive and right-recursive forms.

In general, transformations of grammars can be used to obtain very many effects.

# Ambiguity

A grammar is ambiguous if it allows more than one parse tree to be generated for any sentence.

For example,

$S: AA$

$A: \mathbf{x}$   
 $\mathbf{xx}$

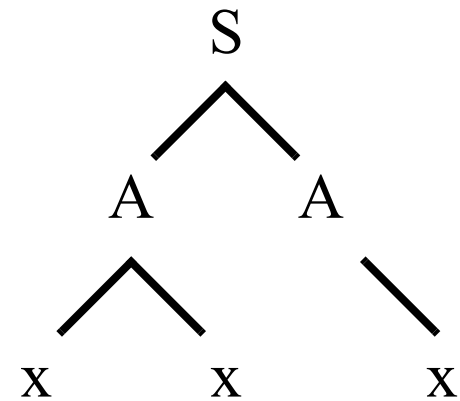
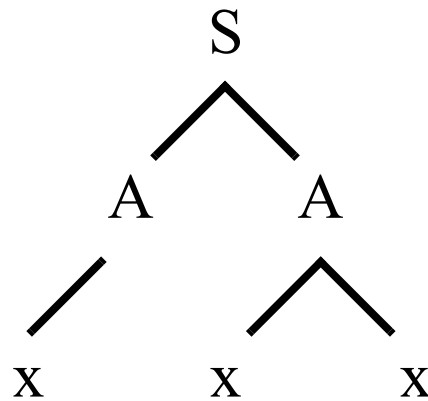
is ambiguous because it has two parses of:

$\mathbf{xxx}$

# Parse xxx

*S*: *AA*

*A*: **x**  
**xx**



# Left-Factoring

A production ( $Z$ ) with identical first parts ( $w$ ) on the rhs, as in:

$$\begin{array}{l} Z: \quad wx \\ \quad \quad wy \end{array}$$

can be re-written as an equivalent grammar with distinct first parts, as in:

$$\begin{array}{l} A: \quad wB \\ B: \quad x \\ \quad \quad y \end{array}$$

Here  $A$  contains the left-factor ( $w$ ) of  $Z$ .

# Removing Ambiguity

Left-factoring can be used to remove ambiguity. For example:

$$A: \quad \mathbf{x}$$
$$\quad \quad \mathbf{xx}$$

is re-written as:

$$B: \quad \mathbf{x}C$$
$$C: \quad \mathbf{x}$$
$$\quad \quad \varepsilon$$

Now  $B$  is an unambiguous version of  $A$ , but  $S \rightarrow AA$  is still ambiguous.

# Removing Ambiguity

The production  $S \rightarrow AA$  generates the language:

**xx**

**xxx**

**xxxx**

Isolating in  $S$  the left factor (xx) of the language yields:

$S:$     **xxD**

$D:$     **x**

**xx**

Isolating the left factor (x) in  $D$  yields:

# Removing Ambiguity

$S: \quad \mathbf{xxD}$

$D: \quad \mathbf{xE}$

$\varepsilon$

$E: \quad \mathbf{x}$

$\varepsilon$

These productions unambiguously generate the language:

**xx**

**xxx**

**xxxx**

# Quiz

The following fragment of grammar is ambiguous.

*statement:    **if test then statement else statement**  
                  **if test then statement***

- What causes the ambiguity?
- Use left-factoring to disambiguate the fragment.
- What practical difference does it make if a language has an explicit “else” symbol and either an implicit “then,” like C, or else an explicit “then” symbol?
- Disambiguate the fragment without using left-factoring!

# Converting Left/Right Recursion

The left-recursive production:

$$A: \quad Ay \\ \quad \quad z$$

is equivalent to the two right-recursive productions:

$$A: \quad zB \\ B: \quad yB \\ \quad \quad \varepsilon$$

- This equivalence can be used to convert a grammar from left- to right-recursive or *vice versa*.

# Quiz

- Convert the following left-recursive grammar into an equivalent right-recursive grammar.

*expr:*

*expr + term*

*term*

- Show your working.

# Left-Recursive Grammar

*expr:*

*expr + term*

*term*

*term:*

*term \* factor*

*factor*

*factor:*

*( expr )*

*identifier*

*number*

# Right-Recursive Grammar

*expr:*

*term expr'*

*expr':*

*+ term expr'*

$\epsilon$

*term:*

*factor term'*

*term':*

*\* factor term'*

$\epsilon$

*factor:*

*( expr )*

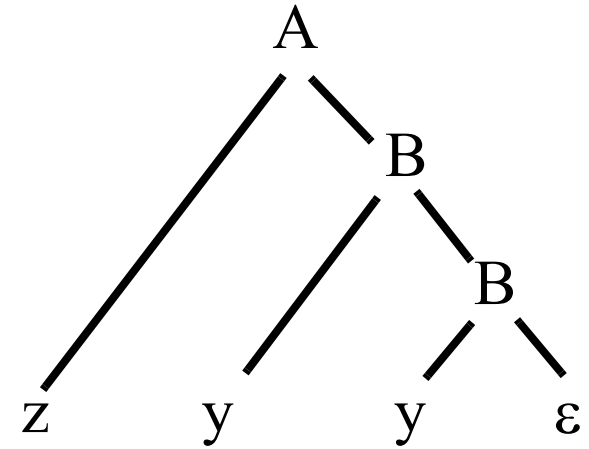
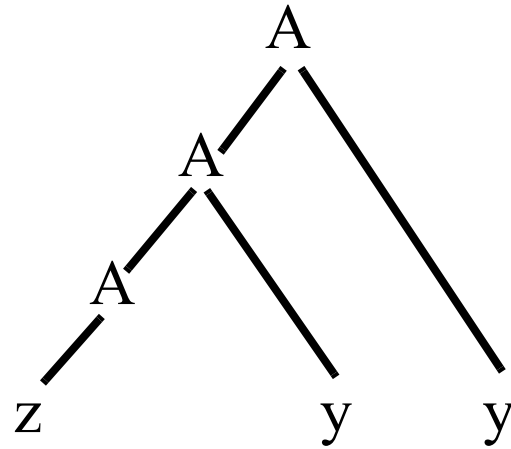
*identifier*

*number*

# Equivalence

- Two grammars,  $G_1$  and  $G_2$ , are equivalent if and only if  $L(G_1) = L(G_2)$ .
- Equivalent grammars can have different parse trees.
- For example:  $A \rightarrow A y \mid z$  is equivalent to  $A \rightarrow z B$  and  $B \rightarrow y B \mid \varepsilon$  but their parse trees are very different as shown next.

# Parse zyy



# Summary

- Left-factoring removes ambiguity.
- Explicit bracketing removes ambiguity.
- Grammars can be transformed automatically from left-to right-recursive and *vice versa*.

# YACC

- YACC is a compiler generator that reads a left-recursive grammar, and actions implemented in C, then writes a C program that implements a compiler for that language. It is usually used in combination with lex.
- YACC is part of Unix. It is highly compatible with C.
- Bison is a variant of YACC supplied by Gnu. It is highly compatible with C<sup>++</sup>.
- Bison is available without charge for use on Unix and Microsoft operating systems. See, for example:

[www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html](http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html)

# YACC

- YACC programs are laid out like this:

Declarations

%%

Rules

%%

Subroutines

- When YACC is run the generated compiler makes calls to `yyparse` which calls `yylex`.
- The lexical analyser `yylex` can be generated by `lex`, or by a similar lexical analyser generator, or it can be implemented by hand.

# Declarations

- The characters in the native character set are treated as tokens.
- C meta characters, such as “\n” and “\t” are treated as tokens.
- Tokens for multi-character lexemes must be declared explicitly.
- C code can be declared. It is copied as is into the source code for the generated compiler.

# Declarations

```
%token NUMBER
```

```
%token IDENTIFIER
```

```
%token IN OUT
```

```
%{
```

```
    /* Arbitrary C code. */
```

```
%}
```

- Can group token declarations on one line to comment their similarity.
- Can have YACC generate a header file of #defines specifying the token values. This will be needed by Lex.

# Rules

- Rules have this format:

```
non_terminal : right_hand_side { actions };
```

- By default the first rule is the start symbol.
- Here is a simple recogniser using the previous tokens.

```
%%  
statement: IDENTIFIER '=' expression  
         | expression  
         ;  
  
expression: NUMBER '+' NUMBER  
          | NUMBER '-' NUMBER  
          ;
```

# Symbol Values

- Every symbol in YACC has a value.
- The value of the left-hand symbol of a rule is referenced by "\$\$".
- The value of the right hand symbols, read left to right, is referenced by "\$n" with  $n = 1, 2, 3, \dots$

expression: NUMBER '+' NUMBER

- \$\$ is the value of “expression”.
- \$1 and \$3 are the values of the successive instances of “NUMBER”.
- \$2 is the value of “+”.

# Calculator

Here is a compiler for a calculator. The actions perform the arithmetic and output.

```
%token IDENTIFIER NUMBER
%%
statement: IDENTIFIER '=' expression
          | expression {printf( "= %d\n", $1);}
          ;

expression: NUMBER '+' NUMBER { $$ = $1 + $3; }
          | NUMBER '-' NUMBER { $$ = $1 - $3; }
          | NUMBER             { $$ = $1; }
          ;

%%
main () {
    while (!feof(yyin)) yyparse();
}
```

# Calculator

Here is the lexical analyser for the calculator.

```
%{
#include "definitions_from_yacc.h"
}%

%%

[\\n\\t]      ;
-? (( [0-9]+ ) | ( [0-9]+\\. [0-9]+ ) ( [eE] [-+]? [0-9]+ ) ? ) \\
{return (NUMBER) ;}
[a-Z]+ ( [a-Z] | [0-9] ) * {return (IDENTIFIER) ;}
•      ECHO;
%%
```

# Advice

Learn how to use lex and YACC. They will help you in your work as professional programmers.

- Every computing task involves the transformation of input text to output text so every task can be performed by a compiler.
- Simple data formatting operations for databases, printers, communications protocols, and the like, can be implemented quickly using lex and YACC.
- It is sometimes easier to extend lex and YACC programs than to extend arbitrary source code.

# Associativity

Associativity controls the grouping of operators of the same precedence. This generally changes the value of the expression.

- If "/" is left associative then  $16/4/2 = (16/4)/2 = 2$ .  
YACC uses the declaration: `%left '/'`
- If "/" is right associative then  $16/4/2 = 16/(4/2) = 8$ .  
YACC uses the declaration: `%right '/'`
- Operators can be non-associative, for example unary minus is non-associative. YACC uses the declaration: `"%nonassoc UMINUS"`

# Precedence

Precedence determines which operators are executed first in an expression. This generally changes the value of the expression.

- If the precedence of "\*" is higher than "+" then  $2+3*5 = 2 + (3*5) = 17$ .
- If the precedence of "\*" is lower than "+" then  $2+3*5 = (2+3)*5 = 25$ .

# Precedence

In YACC successive declarations of associativity declare increasing precedence. Thus:

```
%left '-' '+'  
%left '*' '/'  
%nonassoc UMINUS
```

- Declares left associative operators [-+\*/] with [-+] of lowest precedence, [\*/] of intermediate precedence, and UMINUS of highest precedence.
- A rule can be given the precedence of a specific token by using "%prec TOKEN".

# Advanced Calculator

```
%token IDENTIFIER NUMBER
```

```
%left '-' '+'
```

```
%left '*' '/'
```

```
%nonassoc UMINUS
```

```
%%
```

```
statement: IDENTIFIER '=' expn
```

```
          | expn {printf("= %d\n", $1);}
```

```
          ;
```

# Advanced Calculator

```

expn: expn '+' expn { $$ = $1 + $3; }
      | expn '-' expn { $$ = $1 - $3; }
      | expn '*' expn { $$ = $1 * $3; }
      | expn '/' expn {
          if($3 == 0)
              yyerror("division by zero");
          else $$ = $1 / $3;
        }
      | '-' expn %prec UMINUS { $$ = -$2; }
      | '(' expn ')'          { $$ = $2; }
      | NUMBER                { $$ = $1; }
;

%%

```

# Subroutines

An elementary compiler just calls yyparse:

```
main() {  
    yyparse();  
}
```

A sophisticated compiler might:

- Manipulate files or file streams.
- Call a pre-processor, back-end optimiser, linker, and loader.
- Log compilation details.

# Errors

- Rules can also be used to handle errors.

# Summary

- YACC reads a grammar and C code. It generates C source code for a compiler that compiles the given grammar and executes C code as semantic actions.
- The compiler should call `yyparse` which in turn calls `yylex`.
- The lexical analyser `yylex` can be supplied from any source. Most commonly it is generated by `lex` or implemented by hand.
- YACC rules can be used to handle errors.

# Finite State Automata

- All Chomsky Type 3 Languages can be described by regular grammars or regular expressions.
- All regular expressions can be described by Finite State Automata (FSA).
- FSA are also known as Finite State Machines.

# Finite State Machine

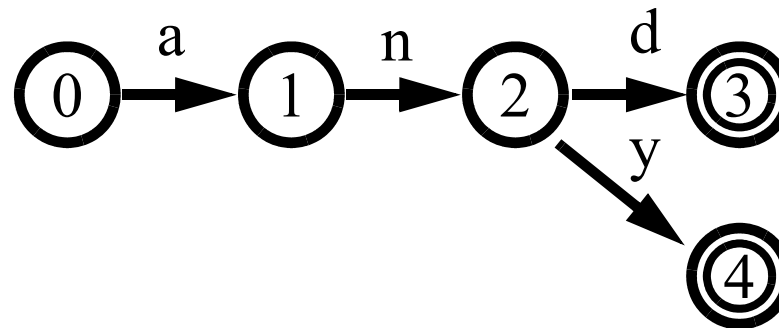
A finite state machine (finite automaton) consists of:

- A finite set of states;
- A set of transitions from one state to another;
- Each transition is labelled by a terminal character;
- There is exactly one start state;
- There is a non-empty set of accepting (end) states.

# DFA

A Deterministic Finite Automaton (DFA) has the following properties.

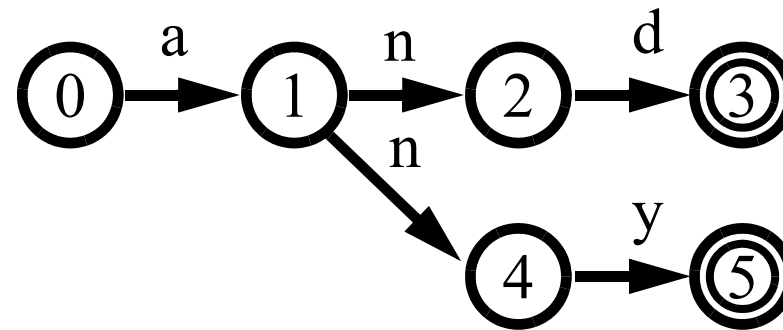
- No transition is labelled  $\epsilon$ .
- Every outgoing transition from a state is labelled with a distinct character.



# N DFA

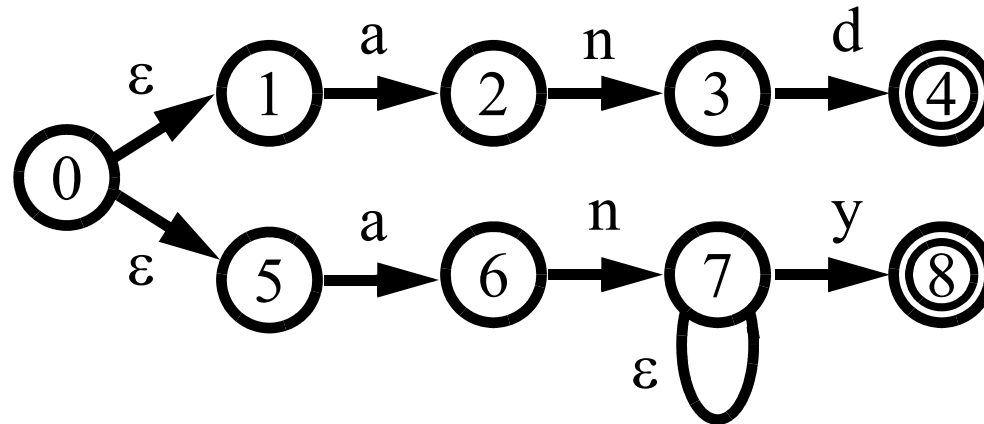
Non-Deterministic Finite Automaton (N DFA).

- Why is the following finite automaton non-deterministic?



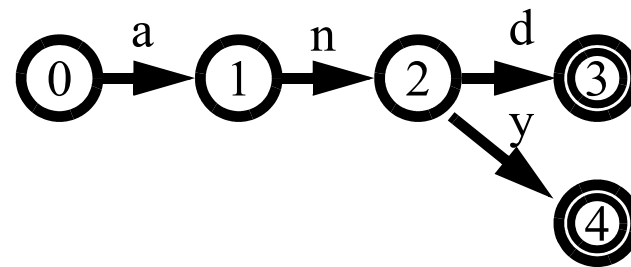
# NDFA

- Why is the following FSA non-deterministic?



# Transition Table

State	Lookahead				Accept
	a	d	n	y	
0	1				
1			2		
2		3		4	
3					T
4					T



# Lex

The Unix utility ‘lex’ constructs a FSA from regular expressions describing lexemes.

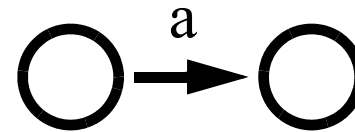
- A table is constructed.
- A driving program is included.
- The driving program uses the greedy algorithm - the longest possible matching sequence is used.
- If two input sequences match then the earlier occurring production is used.
- User-defined C code is included.

# Thompson's Construction

Lex uses Thompson's construction to convert regular expressions to NFA.

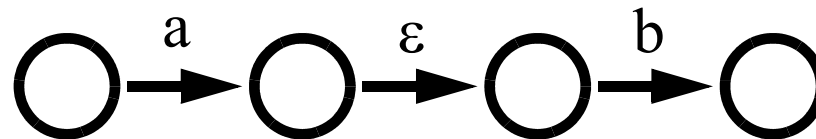
# Thompson: Single Character

The simplest regular expression is a single character, e.g. “a”, and is represented by a very simple FSA:

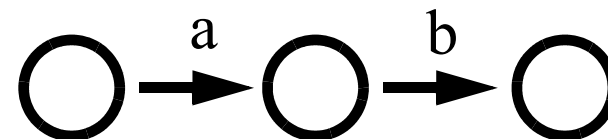


# Thompson: Concatenation

The concatenation of two regular expressions is obtained by concatenating their FSA. For example, the regular expression “ab” is recognised by:



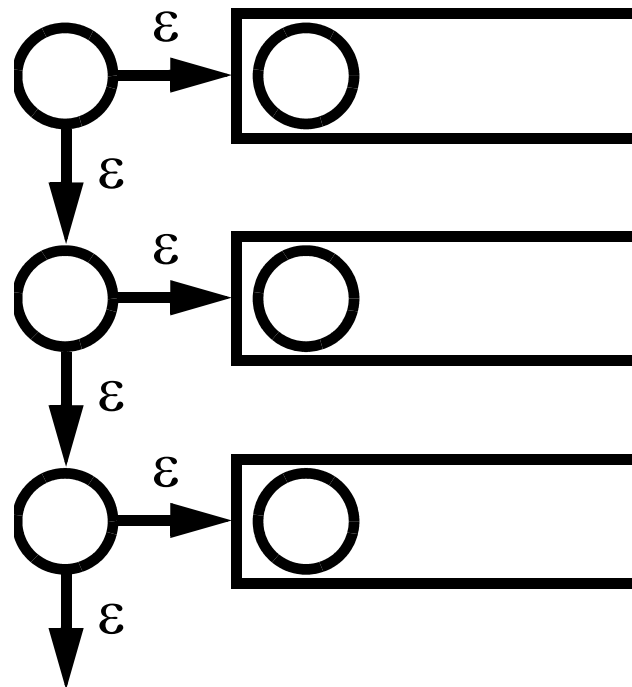
This can be implemented more efficiently as:



This optimisation can be applied in the later examples.

# Thompson: OR of Expressions

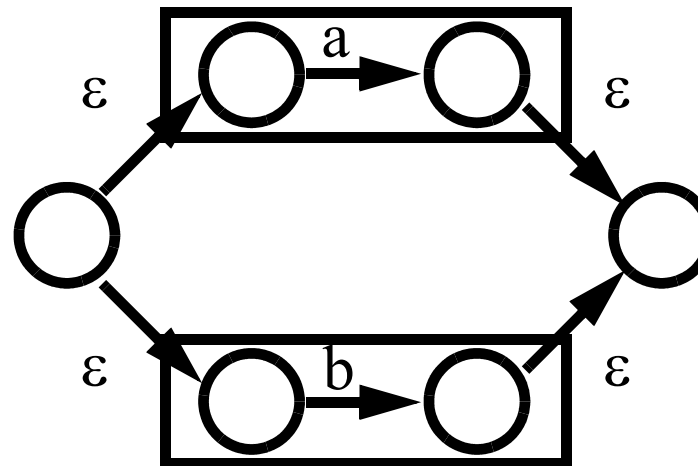
All of the expressions in a lex specification are ORed together. The boxes show NDFA.



To other machines

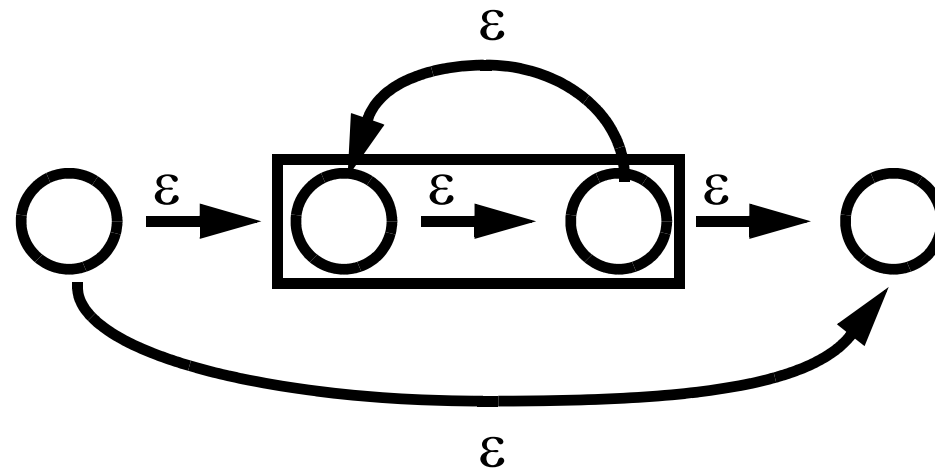
# Thompson: OR within Expressions

Alternatives within an expression are ORed together.  
For example, “a | b” is recognised by:

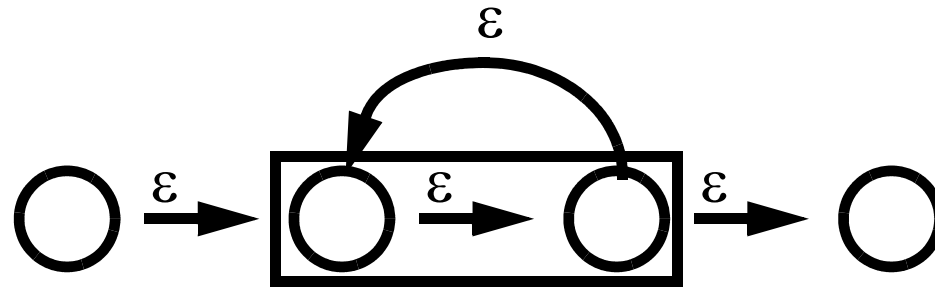


# Thompson: Kleene Closure (\*)

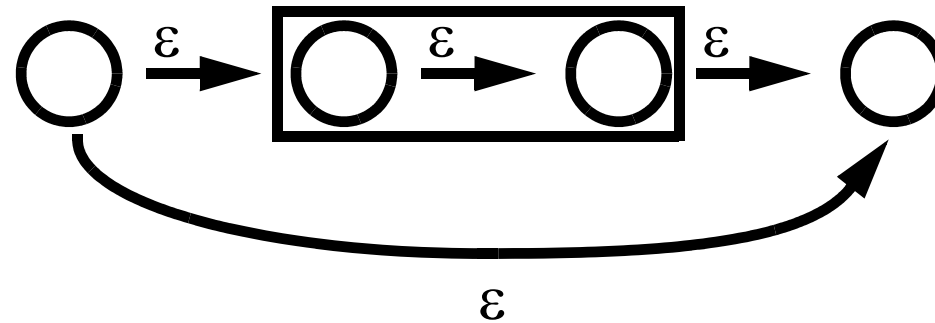
Kleene closure is the most general form of closure.



# Thompson: Positive Closure (+)



# Thompson: Zero-One Closure (?)



# Thompson's Construction

- The NFA produced by Thompson's construction can be converted automatically (deterministically) into DFA.

# Summary

- All regular expressions can be described by FSA.
- Lex reads a specification of lexemes in terms of regular expressions and converts them to a NFA using Thompson's construction.
- Lex converts the NFA to a DFA.
- Lex specifications can contain user-supplied C code.
- Lex generates a lexical analyser comprised of the DFA and a driving program that evaluates the DFA and calls the user-supplied C code.

# Stack Machine

A Finite State Automaton can be augmented with a push-down stack giving rise to a Push Down Stack Automaton (PDSA).

- PDSA can count whereas FSA cannot.
- YACC is implemented in terms of a PDSA.
- It is arguable that the human mind uses a PDSA with three stacks to parse natural languages.

# Recursive Grammars

- Right-recursive grammars are used in top-down compilers.
- Left-recursive grammars are used in bottom-up compilers.

# Right-Recursive Grammar

*expr:*

*term expr'*

*expr':*

*+ term expr'*

$\epsilon$

*term:*

*factor term'*

*term':*

*\* factor term'*

$\epsilon$

*factor:*

*( expr )*

*identifier*

*number*

# Left-Recursive Grammar

*expr:*

*expr + term*

*term*

*term:*

*term \* factor*

*factor*

*factor:*

*( expr )*

*identifier*

*number*

# Bottom-up Parsing

Bottom-up parsing is most easily implemented using a left-recursive grammar and a state table to indicate when to shift, reduce, accept, or error.

lhs  $\rightarrow$  rhs

- REDUCE - if symbols on stack match rhs of production rule then pop symbols and push lhs of production rule.
- SHIFT - push current input symbol and advance read head.

# YACC

The Unix utility YACC constructs a finite state machine from a Chomsky Type 2, left-recursive grammar.

- A table is constructed.
- A driving program is included.
- A lexical analyser, perhaps produced by Lex, is included.
- User defined C code is included.

# Quiz

- What are left- and right-recursive grammars?
- What are top-down and bottom-up parsers?
- Why do most top-down parsers use right-recursive grammars?
- Why do most bottom-up parsers use left-recursive grammars?
- In practice, why are left-recursive grammars shorter than right-recursive grammars.

# Table-Driven Parsers

- Table-driven parsers use a driving program with a state table to direct the parse.
- It is possible to have a top-down, table-driven parser.
- YACC is a bottom-up, table-driven parser.
- Bottom-up, table-driven parsers use the state table to decide whether to shift, reduce, accept, or generate an error.

# Left-Recursive Grammar

- One production has been deleted so that the following examples will fit on the slides!

*expr:*

*expr + term*

*term*

*term:*

*term \* factor*

*factor*

*factor:*

*( expr )*

*number*

# Left-Recursive State Grammar

0: *start*  $\rightarrow$  *expr*

1: *expr*  $\rightarrow$  *expr* + *term*

2: *expr*  $\rightarrow$  *term*

3: *term*  $\rightarrow$  *term* \* *factor*

4: *term*  $\rightarrow$  *factor*

5: *factor*  $\rightarrow$  (*expr*)

6: *factor*  $\rightarrow$  *number*

# State Transition Table

	Input						Goto			
	<b>n</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>⊥</b>	<i>s</i>	<i>e</i>	<i>t</i>	<i>f</i>
0	s1			s2				3	4	5
1		r6	r6		r6	r6				
2	s1			s2				6	4	5
3		s7				a				
4		r2	s8		r2	r2				
5		r4	r4		r4	r4				
6		s7			s9					

# State Transition Table

	Input						Goto			
	<b>n</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>⊥</b>	<i>s</i>	<i>e</i>	<i>t</i>	<i>f</i>
7	s1			s2					10	5
8	s1			s2						11
9		r5	r5		r5	r5				
10		r1	s8		r1	r1				
11		r3	r3		r3	r3				

## Table Driver

```
push(0);
while table[TOP, input] != accept {
    if (table[TOP, input] = "")
        error();
    else if (table[TOP, input] = sX) {
        push(X);
        advance();
    }
    else if (table[TOP, input] = rX) {
        generate_code(X);
        popn(count(RHS));
        push(goto[TOP, LHS]);
    }
} accept();
```

# Parse: $2 * 3 + 4$

State	Input	Reduction
0	$2 * 3 + 4 \perp$	
0 1	$* 3 + 4 \perp$	
0 5	$* 3 + 4 \perp$	6: factor $\rightarrow$ number
0 4	$* 3 + 4 \perp$	4: term $\rightarrow$ factor
0 4 8	$3 + 4 \perp$	
0 4 8 1	$+ 4 \perp$	
0 4 8 11	$+ 4 \perp$	6: factor $\rightarrow$ number

# Parse: $2*3 + 4$

State	Input	Reduction
0 4	+ 4 $\perp$	3: term $\rightarrow$ term * factor
0 3	+ 4 $\perp$	2: exp $\rightarrow$ term
0 3 7	4 $\perp$	
0 3 7 1	$\perp$	6: factor $\rightarrow$ number
0 3 7 5	$\perp$	4: term $\rightarrow$ factor
0 3 7 10	$\perp$	1: exp $\rightarrow$ exp + term
0 3	$\perp$	accept

# Summary

- FSA cannot count, but PDSA can.
- YACC is implemented as a PDSA.
- Bottom-up, table-driven parsers use a driving program with a state table to direct the parse.
- Bottom-up parsers: shift, reduce, accept, or error.
- YACC is a bottom-up, table-driven parser.

# Turing Machine

- The Turing Machine is a *definition* of a theoretical computer.
- The Church-Turing Thesis is that any effective computation can be done by a Turing Machine.
- The Church-Turing Thesis is a *hypothesis*. It might be right, wrong, undecidable, or irrelevant.
- So far as is known, the human mind is effectively computable so, in theory, it could be emulated by a Turing Machine.

# Turing Machine

- A Turing Machine is made up of a FSA and a, potentially, infinitely long data tape.
- There are four instructions in a Turing Machine:

Read symbol on tape (possibly  $\epsilon$  i.e. blank) and change state;

Move tape one symbol left;

Move tape one symbol right;

Write symbol on tape (possibly  $\epsilon$  i.e. blank).

# Turing Machine

- The Turing Machine is given a finitely long (possibly zero) program of symbols on its data tape.
- Execution starts at the left-most, non-blank symbol on the tape (or else at an arbitrary blank symbol).

# Turing Machine

A Turing Machine can be specified by a set of quintuples  $\langle s, r, w, s', d \rangle$  where:

- $s$  is the current state;
- $r$  is the current symbol to be read;
- $w$  is the symbol to be written (possibly  $\epsilon$ );
- $s'$  is the new state;
- $d$  is the direction of tape movement: left or right.

# Turing Machine

- A Turing Machine is deterministic if every quintuple has a discrete initial pair  $\langle s, r \rangle$ .
- If there are any duplicate initial pairs then the machine must make a choice of which quintuple to execute. This choice makes the machine non-deterministic.

# Quiz

- Do you think the human mind is deterministic or non-deterministic?

# Turing Machine

- The Turing symbols are discrete.
- There are finitely many Turing symbols.

# Quiz

- Do you think the human mind operates in discrete states or in a continuum?

# Halting Problem

- A Turing Machine halts when it has computed a result.
- The Universal Turing Machine can emulate any other deterministic Turing Machine.
- A Turing Machine cannot decide whether an arbitrary Turing program halts or not. This is called the halting problem.

# Proof of Undecideability

- A Turing machine is said to be “undecidable” if there is any case in which it does not halt.
- Hypothesise that  $D(X)$  is a deterministic Turing-machine (a Decision machine) that halts and prints TRUE if  $X$  halts or else halts and prints FALSE if  $X$  does not halt.
- Then  $D(D)$  can halt and print TRUE if  $(D)$  halts, but if  $(D)$  does not halt then  $D(D)$  cannot halt and print FALSE.
- The “but” clause contradicts the hypothesis.
- Therefore no universal decision-machine  $D(X)$  exists.

# Unlimited Register Machine

It can be proved that an Unlimited Register Machine (URM) is equivalent to a Turing Machine.

A URM is made up of:

- A finitely long (possibly zero) program of instructions.
- An unlimited strip of registers.

# Unlimited Register Machine

A URM has four instructions:

- G - ground the current register (set it to zero).
- C - count (increment the current register by one).
- A - advance (move to the next instruction if a given register is zero, otherwise go to a given instruction).
- T - transcribe (swap the contents of two given registers).

# Perspex Machine

It can be proved that a Perspex Machine can do anything that a URM can do. So it can do anything that a Turing Machine can do.

A Perspex Machine is made up of:

- A 4D space of  $4 \times 4$  matrices.

# Perspex Matrix

A Perspex Matrix is made up of four columns.

$$\begin{bmatrix} x_1 & y_1 & z_1 & t_1 \\ x_2 & y_2 & z_2 & t_2 \\ x_3 & y_3 & z_3 & t_3 \\ x_4 & y_4 & z_4 & t_4 \end{bmatrix}$$

# Perspex Instruction

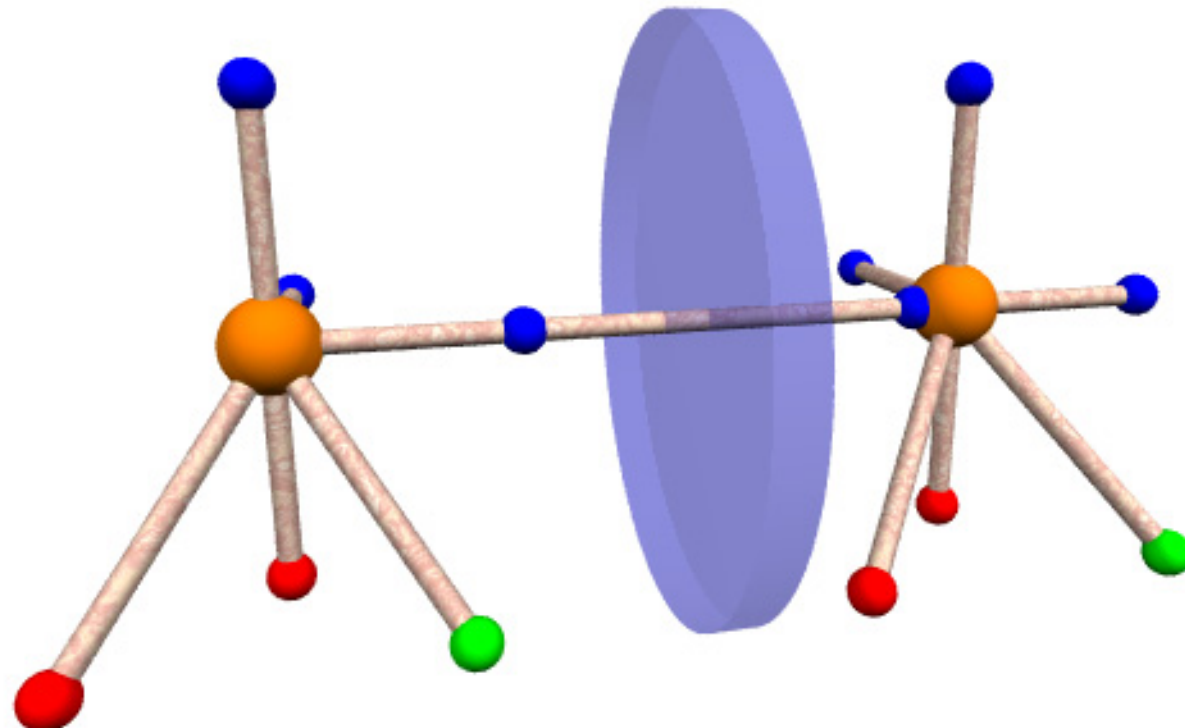
A perspex matrix currently parameterises the instruction:

$$\vec{x}\vec{y} + \text{continuum}(\vec{z}) \rightarrow \vec{z}$$

- $\text{jump}(z_{11}, t)$

# Perspex Neuron

The Perspex Neuron can be a 3D object that communicates across a 1D time.



# Perspex Matrix

A Perspex Matrix can describe, amongst other things:

- An artificial neuron.
- A 3D shape.
- A 3D motion.
- An instruction for a Turing Machine.

# Perspex Machine

- The Perspex Machine is deterministic, but it can be used to emulate non-deterministic Turing Machines.
- The Perspex Machine operates in a continuum so it can solve the halting problem.
- The Perspex Machine operates on a continuum, but it can emulate the symbols in a Turing Machine as special cases of the continuum.

# Perspex Machine

- Is it possible to implement a compiler for a Perspex Machine?
- Can a Perspex Machine acquire information that was not programmed into it?
- What is the basis of language and logic in a Perspex Machine?
- Can a Perspex Machine do anything that cannot be described in language? If so, how does it do them?

# Quiz

- Do you think that the human mind is more like a Turing Machine or a Perspex Machine?

# Summary

- The Turing Thesis is that any computable function of symbols can be computed by a Turing machine.
- In particular, compilers can be implemented on a Turing machine.
- There is a mathematical proof that there is a Universal, deterministic Turing-machine that can emulate any deterministic Turing-machine, including itself.

# Summary

- The Perspex Thesis is that a Perspex Machine can simulate any physical thing, including mind, to arbitrary accuracy and, conversely, any physical thing, including mind instantiates a Perspex Machine.
- In particular, human beings and robots are Perspex Machines.
- There is a mathematical proof that any Perspex Machine can be transformed into any other Perspex Machine.

# Revision

Some of the stages of compilation are:

- Pre-processing;
- Lexical analysis;
- Parsing;
- Code generation;
- Optimisation;
- Linking;
- Loading.

# Revision

Compilers rely on a theoretical basis of:

- Chomskian language types;
- Finite State Automata;
- Turing Machines.

But Perspex Machines:

- Cannot be described completely in any language;
- Have infinite numbers of states in a continuum;
- Are super-Turing machines.

# Revision

If you have studied well you will be able to:

- Write a top-down compiler by hand;
- Specify a compiler and generate it with a compiler compiler;
- Appreciate the theory of compilation;
- Extend the theory and/or practice of compilation.