A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
    ONE
    NOUGHT
    ONE
END
```

- The formal grammar : Backus-Naur form (BNF) :

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>    ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```

- The formal grammar : Backus-Naur form (BNF) :

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>    ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```

- The formal grammar : Backus-Naur form (BNF) :

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>     ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

- The '|' means OR.

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```

- The formal grammar : Backus-Naur form (BNF) :

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>     ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

- The '|' means OR.
- "BEGIN", "ONE", "NOUGHT" and "END" are string constants.

A Formal Grammer : 0&1s

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```

- The formal grammar : Backus-Naur form (BNF) :

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>     ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

- The '|' means OR.
- "BEGIN", "ONE", "NOUGHT" and "END" are string constants.
- <CODE> is described recursively.

A Formal Grammer : 0&1s

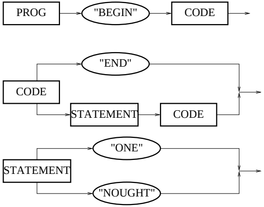
- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```

- The formal grammar : Backus-Naur form (BNF) :

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>    ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

- The '|' means OR.
- "BEGIN", "ONE", "NOUGHT" and "END" are string constants.
- <CODE> is described recursively.
- You could also think of this grammar in terms of a *railroad diagram*:



Coding a 0 & 1s Parser

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#define MAXNUMTOKENS 100
#define MAXTOKENSIZE 20
#define strsame(A,B) (strcmp(A, B)==0)
#define ERROR(PHRASE) { fprintf(stderr, \
          "Fatal Error %s occurred in %s, line %d\n", PHRASE, \
          __FILE__, __LINE__); \
          exit(EXIT_FAILURE); }

struct prog{
   char wds[MAXNUMTOKENS][MAXTOKENSIZE];
   int cw; // Current Word
};
typedef struct prog Program;

void Prog(Program *p);
void Code(Program *p);
void Statement(Program *p);

int main(void)
{
   Program* prog = calloc(1, sizeof(Program));
   int i=0;
   while(scanf("%s", prog->wds[i++])==1 && i<MAXNUMTOKENS);
   assert(i<MAXNUMTOKENS);
   Prog(prog);
   printf("Parsed OK\n");
   return 0;
}
```

Coding a 0 & 1s Parser

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#define MAXNUMTOKENS 100
#define MAXTOKENSIZE 20
#define strsame(A,B) (strcmp(A, B)==0)
#define ERROR(PHRASE) { fprintf(stderr, \
          "Fatal Error %s occurred in %s, line %d\n", PHRASE, \
          __FILE__, __LINE__); \
          exit(EXIT_FAILURE); }

struct prog{
   char wds[MAXNUMTOKENS][MAXTOKENSIZE];
   int cw; // Current Word
};
typedef struct prog Program;

void Prog(Program *p);
void Code(Program *p);
void Statement(Program *p);

int main(void)
{
   Program* prog = calloc(1, sizeof(Program));
   int i=0;
   while(scanf("%s", prog->wds[i++])==1 && i<MAXNUMTOKENS);
   assert(i<MAXNUMTOKENS);
   Prog(prog);
   printf("Parsed OK\n");
   return 0;
}
```

```c
void Prog(Program *p)
{
   if(!strsame(p->wds[p->cw], "BEGIN")){
      ERROR("No BEGIN statement ?");
   }
   p->cw = p->cw + 1;
   Code(p);
}

void Code(Program *p)
{
   if(strsame(p->wds[p->cw], "END")){
      return;
   }
   Statement(p);
   p->cw = p->cw + 1;
   Code(p);
}

void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

Running the Parser

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

Parsed OK

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

```
Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END
```

```
Parsed OK
```

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

```
Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END
```

```
Parsed OK
```

```
BEGIN END
```

```
Parsed OK
```

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

```
Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END
```

```
Parsed OK
```

```
BEGIN END
```

```
Parsed OK
```

```
BEGIN
 ONE
 TWO
END
```

```
Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

```
Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END
```

```
Parsed OK
```

```
BEGIN END
```

```
Parsed OK
```

```
BEGIN
 ONE
 TWO
END
```

```
Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
BEGIN
 ONE
 NOUGHT
```

```
Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

```
Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END
```

```
Parsed OK
```

```
BEGIN END
```

```
Parsed OK
```

```
BEGIN
 ONE
 TWO
END
```

```
Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
BEGIN
 ONE
 NOUGHT
```

```
Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
 ONE
 NOUGHT
END
```

```
Fatal Error No BEGIN statement ? occurred in p01a.c, line 55
```

- Notice that the END statement is actually used as the recursive base-case in the formal grammar in the function Code().

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END

Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END

Parsed OK
```

```
BEGIN END

Parsed OK
```

```
BEGIN
 ONE
 TWO
END

Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
BEGIN
 ONE
 NOUGHT

Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
 ONE
 NOUGHT
END

Fatal Error No BEGIN statement ? occurred in p01a.c, line 55
```

- Notice that the END statement is actually used as the recursive base-case in the formal grammar in the function Code().
- The parser doesn't actually **do** anything other than check that the input is **valid** or not.

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END

Parsed OK
```

```
BEGIN ONE NOUGHT NOUGHT END

Parsed OK
```

```
BEGIN END

Parsed OK
```

```
BEGIN
 ONE
 TWO
END

Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
BEGIN
 ONE
 NOUGHT

Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79
```

```
 ONE
 NOUGHT
END

Fatal Error No BEGIN statement ? occurred in p01a.c, line 55
```

- Notice that the END statement is actually used as the recursive base-case in the formal grammar in the function Code().
- The parser doesn't actually **do** anything other than check that the input is **valid** or not.
- An interpreter performs the required operations (e.g. printing to the screen in this case) alongside the parser checking the syntax.

Running the Parser

```
BEGIN
  ONE
  NOUGHT
  ONE
END
```

Parsed OK

```
BEGIN ONE NOUGHT NOUGHT END
```

Parsed OK

```
BEGIN END
```

Parsed OK

```
BEGIN
 ONE
 TWO
END
```

Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79

```
BEGIN
 ONE
 NOUGHT
```

Fatal Error Expecting a ONE or NOUGHT ? occurred in p01a.c, line 79

```
 ONE
 NOUGHT
END
```

Fatal Error No BEGIN statement ? occurred in p01a.c, line 55

- Notice that the END statement is actually used as the recursive base-case in the formal grammar in the function Code().
- The parser doesn't actually **do** anything other than check that the input is **valid** or not.
- An interpreter performs the required operations (e.g. printing to the screen in this case) alongside the parser checking the syntax.
- A slight modification to the code is required to produce an interpreter.

Interpreters are Modified Parsers

```
void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      printf("1\n");
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      printf("0\n");
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

Interpreters are Modified Parsers

```
void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      printf("1\n");
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      printf("0\n");
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

Execution :

```
BEGIN
ONE NOUGHT ONE NOUGHT
END
1
0
1
0
```

Interpreters are Modified Parsers

```c
void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      printf("1\n");
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      printf("0\n");
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

Execution :

```
BEGIN
ONE NOUGHT ONE NOUGHT
END
1
0
1
0
```

- I've also taken out the "Parsed OK" message.

Interpreters are Modified Parsers

```c
void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      printf("1\n");
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      printf("0\n");
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

Execution :

```
BEGIN
ONE NOUGHT ONE NOUGHT
END
1
0
1
0
```

- I've also taken out the "Parsed OK" message.
- To extend the parser to be an interpreter you might now need to ‘understand’ what the input means - the context-free requirement is removed somewhat.

Formal Grammar for Parsing Maths Expressions

To parse a string such as:
"A+B*C"
"A*(B+C)" or
"-(B*F)"
we could invent our own grammar :

Formal Grammar for Parsing Maths Expressions

To parse a string such as:
"A+B*C"
"A*(B+C)" or
"-(B*F)"
we could invent our own grammar :

```
<EXPR> ::= <EXPR><OP><EXPR> |
           "(" <EXPR> ")" |
           "-"<EXPR> | Letter
<OP>   ::= "+" | "-" | "*" | "/"
```

Formal Grammar for Parsing Maths Expressions

To parse a string such as:
"A+B*C"
"A*(B+C)" or
"-(B*F)"
we could invent our own grammar :

```
<EXPR> ::= <EXPR><OP><EXPR> |
           "(" <EXPR> ")" |
           "-"<EXPR> | Letter
<OP>   ::= "+" | "-" | "*" | "/"
```

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXEXPR 400
struct prog{
        char str[MAXEXPR];
        int count;
};
typedef struct prog Prog;
void Op(Prog *p);
int isop(char c);
void Expr(Prog *p);
#define ON_ERROR(S) {fprintf(stderr, "%s", S);\
                     exit(EXIT_FAILURE);}
int main(void)
{
   Prog p;
   p.count = 0;
   if(scanf("%[A-Z-+()]s", p.str) != 1){
      ON_ERROR("Couldn't read your expression ?\n");
   }
   Expr(&p);
   printf("Parsed OK !\n");
   return 0;
}

int isop(char c)
{
   if(c=='+' || c=='-' || c=='*' || c=='/'){
      return 1;
   }
   return 0;
}
```

Running the Maths Parser

```
void Op(Prog *p)
{
    if(!isop(p->str[p->count]))
        ON_ERROR("I was expecting a letter ?\n");
}
void Expr(Prog *p)
{
    if(p->str[p->count] == '('){
        p->count = p->count + 1;
        Expr(p);
        p->count = p->count + 1;
        if(p->str[p->count] != ')'){
            ON_ERROR("I was expecting a ) ?\n");
        }
    }
    else if(p->str[p->count] == '-'){
        p->count = p->count + 1;
        Expr(p);
    }
    // Note Look-Ahead
    else if(isop(p->str[p->count+1])){
        if(isupper(p->str[p->count])){
            p->count = p->count + 1;
            Op(p);
            p->count = p->count + 1;
            Expr(p);
        }
    }
    else{
        if(!isupper(p->str[p->count]) ||
            isupper(p->str[p->count+1])){
            ON_ERROR("Expected a single letter ?\n");
        }
    }
```

Running the Maths Parser

```
void Op(Prog *p)
{
    if(!isop(p->str[p->count]))
       ON_ERROR("I was expecting a letter ?\n");
}
void Expr(Prog *p)
{
    if(p->str[p->count] == '('){
       p->count = p->count + 1;
       Expr(p);
       p->count = p->count + 1;
       if(p->str[p->count] != ')'){
          ON_ERROR("I was expecting a ) ?\n");
       }
    }
    else if(p->str[p->count] == '-'){
       p->count = p->count + 1;
       Expr(p);
    }
    // Note Look-Ahead
    else if(isop(p->str[p->count+1])){
       if(isupper(p->str[p->count])){
          p->count = p->count + 1;
          Op(p);
          p->count = p->count + 1;
          Expr(p);
       }
    }
    else{
       if(!isupper(p->str[p->count]) ||
          isupper(p->str[p->count+1])){
          ON_ERROR("Expected a single letter ?\n");
       }
    }
```

Execution :

```
A+(B*C)
Parsed OK !
```

Running the Maths Parser

```
void Op(Prog *p)
{
   if(!isop(p->str[p->count]))
      ON_ERROR("I was expecting a letter ?\n");
}
void Expr(Prog *p)
{
   if(p->str[p->count] == '('){
      p->count = p->count + 1;
      Expr(p);
      p->count = p->count + 1;
      if(p->str[p->count] != ')'){
         ON_ERROR("I was expecting a ) ?\n");
      }
   }
   else if(p->str[p->count] == '-'){
      p->count = p->count + 1;
      Expr(p);
   }
   // Note Look-Ahead
   else if(isop(p->str[p->count+1])){
      if(isupper(p->str[p->count])){
         p->count = p->count + 1;
         Op(p);
         p->count = p->count + 1;
         Expr(p);
      }
   }
   else{
      if(!isupper(p->str[p->count]) ||
         isupper(p->str[p->count+1])){
         ON_ERROR("Expected a single letter ?\n");
      }
   }
```

Execution :

```
A+(B*C)
Parsed OK !
```

Execution :

```
-(B*C+D)
Parsed OK !
```

Running the Maths Parser

```c
void Op(Prog *p)
{
   if(!isop(p->str[p->count]))
      ON_ERROR("I was expecting a letter ?\n");
}
void Expr(Prog *p)
{
   if(p->str[p->count] == '('){
      p->count = p->count + 1;
      Expr(p);
      p->count = p->count + 1;
      if(p->str[p->count] != ')'){
         ON_ERROR("I was expecting a ) ?\n");
      }
   }
   else if(p->str[p->count] == '-'){
      p->count = p->count + 1;
      Expr(p);
   }
   // Note Look-Ahead
   else if(isop(p->str[p->count+1])){
      if(isupper(p->str[p->count])){
         p->count = p->count + 1;
         Op(p);
         p->count = p->count + 1;
         Expr(p);
      }
   }
   else{
      if(!isupper(p->str[p->count]) ||
         isupper(p->str[p->count+1])){
         ON_ERROR("Expected a single letter ?\n");
      }
   }
```

Execution :

```
A+(B*C)
Parsed OK !
```

Execution :

```
-(B*C+D)
Parsed OK !
```

Execution :

```
A
Parsed OK !
```

Running the Maths Parser

Execution :

```
A+(C*
I was expecting a single letter ?
```

Running the Maths Parser

Execution :

```
A+(C*
I was expecting a single letter ?
```

Execution :

```
a+c
Couldn't read your expression ?
```

Running the Maths Parser

Execution :

```
A+(C*
I was expecting a single letter ?
```

Execution :

```
a+c
Couldn't read your expression ?
```

Execution :

```
A*B+(C*D
I was expecting a ) ?
```

Running the Maths Parser

Execution :

```
A+(C*
I was expecting a single letter ?
```

Execution :

```
a+c
Couldn't read your expression ?
```

Execution :

```
A*B+(C*D
I was expecting a ) ?
```

- The formal grammar doesn’t explain everything that the programmer needs to know.

Running the Maths Parser

Execution :

```
A+(C*
I was expecting a single letter ?
```

Execution :

```
a+c
Couldn't read your expression ?
```

Execution :

```
A*B+(C*D
I was expecting a ) ?
```

- The formal grammar doesn't explain everything that the programmer needs to know.
- It is not clear whether the a+c example is invalid or not.

Running the Maths Parser

Execution :

```
A+(C*
I was expecting a single letter ?
```

Execution :

```
a+c
Couldn't read your expression ?
```

Execution :

```
A*B+(C*D
I was expecting a ) ?
```

- The formal grammar doesn't explain everything that the programmer needs to know.
- It is not clear whether the a+c example is invalid or not.
- It is not clear how spaces should be dealt with.