CSc 420 Compiler Construction
Spring Semester 2021

Professor Vulis

April 29, 2021

Chapter 1

Intro

1.1 Syllabus

Topics:

1,2	History of programming languages; assemblers; macro-preprocessors; pre-theory language (FORTRAN); ALGOL; PL/I; Pascal/Modula; C; ADA.
3,4	Compilers and Interpreters; Anatomy of a compiler; Passes and Phases; Overview of a sample compiler.
5	Overview of optimizations
6,7	Data structures for Symbol tables.
8,9	Scanner
10,11	Context-free grammars; top-down parsing; bringing grammars to LL(1)-suitable form.
12,13	The expression grammar; an expression evaluator; converting expressions to the postfix form

14,15 The target hardware and code-generation overview (P-code machine); a simple model of converting P-code to a register code. Detailed analysis of common statements:

16 Simple control statements (repeat, while, if).

17 goto

18 case

19 arrays, addresses

20-21 procedures.

22 type trees

23 Data-type conversion

24 Separate compilation and units.

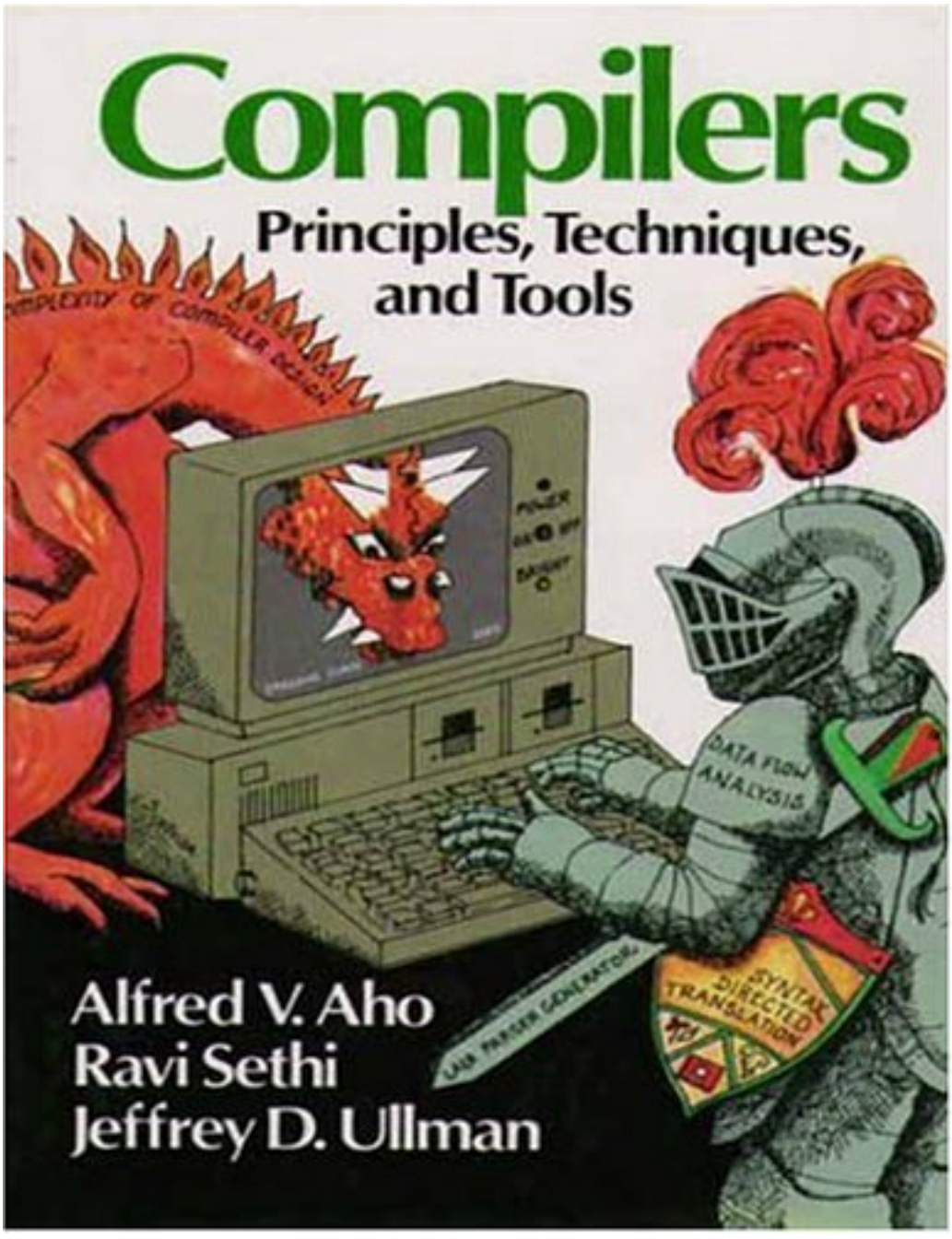
Grading: The grading is based entirely on the success in the implementation of a small Pascal language compiler (100

Office Hours: My hours this term for CSc 420 will be 3:45 - 4:45 on Mondays.

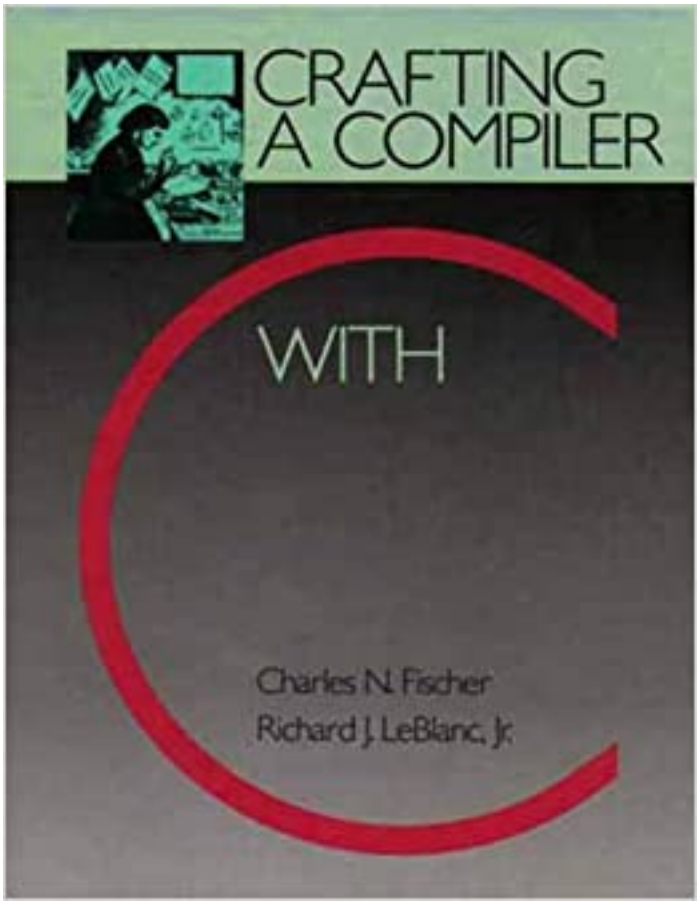
1.2 Bibliography

- Aho, Ullman. Principles of compiler design (Green Dragon)

- Aho, Lam, Sethi, Ullman. Compilers (Red Dragon)

- 

- Tremblay and Sorenson. Theory and Practice of Compiler Writing.

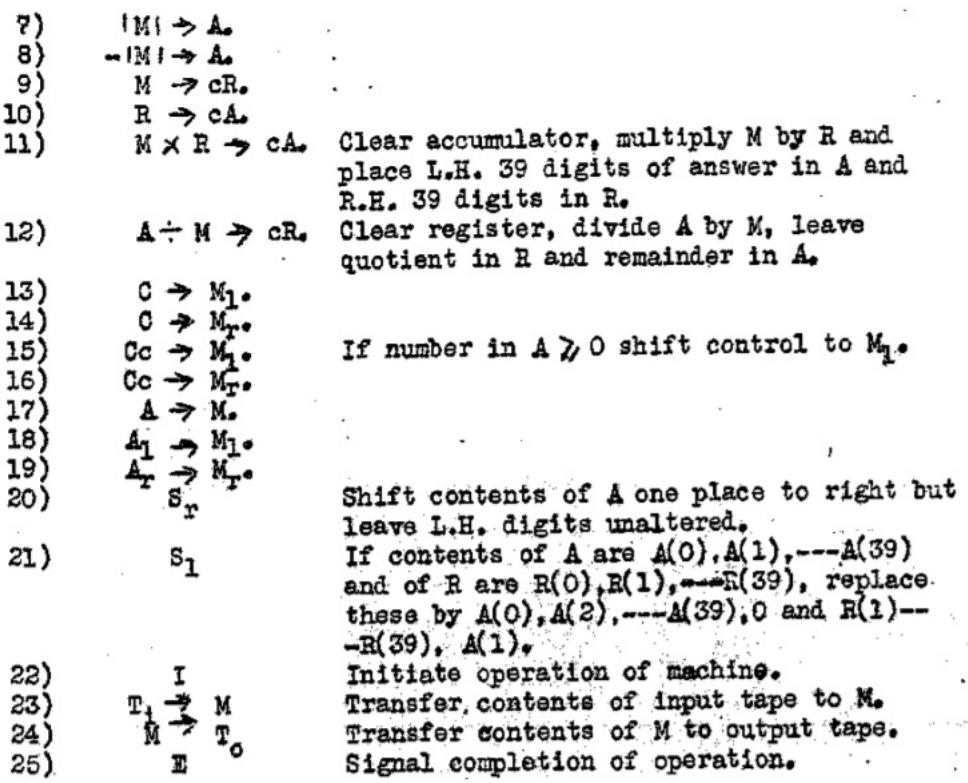
- Tremblay and Sorenson. An Implememtation guide to Compiler Writing.
- Charles N. Fischer, Richard Joseph LeBlanc. Crafting a Compiler with C
- 

- Wirth. The design of a pascal compiler

Chapter 2

Historical Notes

- programming machine language.
- assemblers (circa 1947), Booth 1947



7)	$\|M\| \rightarrow A$.	
8)	$-\|M\| \rightarrow A$.	
9)	$M \rightarrow cR$.	
10)	$R \rightarrow cA$.	
11)	$M \times R \rightarrow cA$.	Clear accumulator, multiply M by R and place L.H. 39 digits of answer in A and R.H. 39 digits in R.
12)	$A \div M \rightarrow cR$.	Clear register, divide A by M, leave quotient in R and remainder in A.
13)	$C \rightarrow M_l$.	
14)	$C \rightarrow M_r$.	
15)	$Cc \rightarrow M_l$.	If number in A $\geq$ 0 shift control to M_l.
16)	$Cc \rightarrow M_r$.	
17)	$A \rightarrow M$.	
18)	$A_l \rightarrow M_l$.	
19)	$A_r \rightarrow M_r$.	
20)	S_r	Shift contents of A one place to right but leave L.H. digits unaltered.
21)	S_l	If contents of A are A(0),A(1),---A(39) and of R are R(0),R(1),---R(39), replace these by A(0),A(2),---A(39),0 and R(1)---R(39), A(1).
22)	I	Initiate operation of machine.
23)	$T_i \rightarrow M$	Transfer contents of input tape to M.
24)	$M \rightarrow T_o$	Transfer contents of M to output tape.
25)	E	Signal completion of operation.

Reference D.Solomon, p.16

- symbolic assembler, Nathaniel Rochester, who developed an assembler for the IBM 701 in 1954.
- FORTRAN (circa 1954, commercially shipped 1957), the original compiler

Imporance of assembler : relocations, not symbols.

```
    JMP  label  ;  [OPCODE][address]
.....
     label:
```

2.1 FORTRAN

Example code - FORTRAN IV or 66

```
C       THE TPK ALGORITHM
C       FORTRAN IV STYLE
        DIMENSION A(11)
        FUN(T) = SQRT(ABS(T)) + 5.)*T**3
        READ (5,1) A
    1   FORMAT(5F10.2)
        DO 10 J = 1, 11
           I = 11 - J
           Y = FUN(A(I+1))
           IF (400.0-Y) 4, 8, 8
    4          WRITE (6,5) I
    5          FORMAT(I10, 10H TOO LARGE)
           GO TO 10
    8          WRITE(6,9) I, Y
               FORMAT(I10, F12.6)
   10   CONTINUE
        STOP
        END
```

B. Fortran 90 (and 95) sample code

```
module module1
  integer:: n
  contains

  recursive subroutine sub1(x)
  integer,intent(inout):: x
  integer:: y
  y = 0
  if (x < n) then
    x = x + 1
    y = x**2
    print *, 'x = ', x,', y = ', y
    call sub1(x)
    print *, 'x = ', x,', y = ', y
  end if
  end subroutine sub1

  end module module1

  program main
  use module1
  integer:: x = 0
  print *, 'Enter number of repeats'
  read (*,*) n
  call sub1(x)
  end program main
```

Was FORTRAN meant to be a high level language?

The big bug:

```
   DO 1 I=1.10
   ....
   ....
   ....
1  CONTINUE
```

the intended code was

```
   DO 1 I=1,10
   ....
   ....
   ....
1  CONTINUE
```

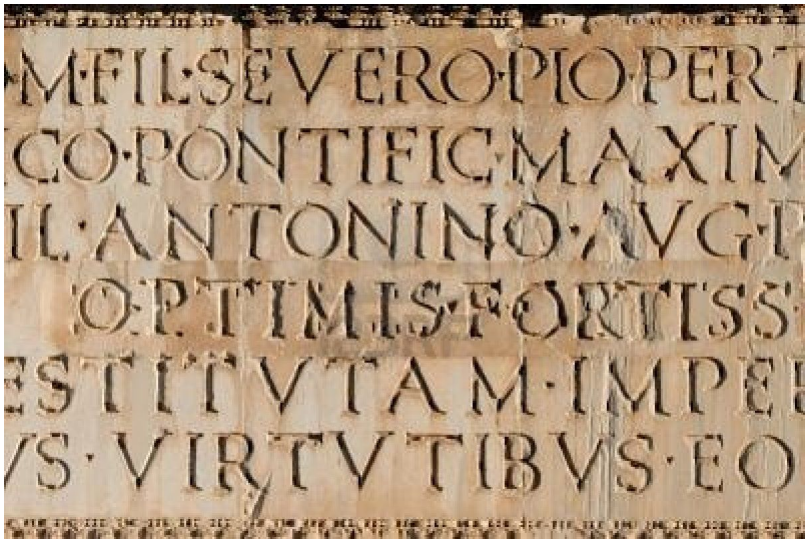
the actual code is equivalent to

```
   DO1I=1.10
   ....
   ....
   ....
1  CONTINUE
```

Why spaces are not made important?

Rationale : economy of space caused by expensive media!

Same applies to human languages, for example, old Latin inscriptions did not use space, but rather a smaller (usually central) dot:



Limit on array dimensions and loops in the original FORTRAN

2.2 IAL/Algol 60 project

```
begin
        real x;
        integer i, j;
        for i := 2 until Z do begin
                x := Y[i];
                for j := i-1 step -1 until 1 do
                        if x >= A[j] then begin
                                A[j+1] := x; goto Found
                        end else
                                A]j+1] := A[j];
                A[1] := x;
        Found:
                end
        end
end Sort
```

IAL == International Algebraic Language == Original Algol 58.

Preliminary report

```
<statement list> -> <statement>
                  | <statement list> ; <statement>

<statement> -> <assignment>
             | if <expression> then <statement>
             | if <expression> then <statement>
               else <statement>
             | begin <statement list> end
```

In the 1963 report on Algol, this was changed to:

```
<statement list> -> <statement>
                  | <statement list> ; <statement>

<statement> -> <unconditional statement>
             | if <expression> then <unconditional statement>
             | if <expression> then <unconditional statement>
               else <statement>

<unconditional statement> -> <assignment>
                           | begin <statement list> end
```

Reasons for failure:

1. No I/O
2. Dialects split the market
3. IBM

2.3 JOVIAL

Jules Own Version of International Algebraic Language.

JOVIAL is a high-level programming language similar to ALGOL, specialized for developing embedded systems

Effectively the official programming language of US Airforce in 1960s/1970s.

2.4 Algol W

W: Wirth or Weldon?

> It represented a relatively conservative modification of ALGOL 60, adding string, bitstring, complex number and reference to record datatypes and call-by-result passing of parameters, introducing the while statement, replacing switch with the case statement, and generally tightening up the language.

Algol W

Charles Weldon

2.5 PL/I

IBM's answer. Concept : 1964, Implementation : 1969.

PL/I

```
Hello2: proc options(main);
     put list ('Hello, world!');
end Hello2;
```

PLIOPT Compiler

Unusual features include label variables, multiple entry points to a procedure.

2.6 Algol 68

Algol 68

ALGOL 68 (short for Algorithmic Language 1968) is an imperative programming language that was conceived as a successor to the ALGOL 60 programming language, designed with the goal of a much wider scope of application and more rigorously defined syntax and semantics.

The complexity of the language's definition, which runs to several hundred pages filled with non-standard terminology, made compiler implementation difficult and it was said it had "no implementations and no users".

2.7 Pascal

Pascal (1968)

p-System

```
program HelloWorld(output);
begin
    Write('Hello, World!')
    {No ";" is required after the last statement of a block -
        adding one adds a "null statement" to the program,
        which is ignored by the compiler.}
end.
```

Unusual features: sets, ranges, `with`, nested procedures.

Weaknesses: no separate compilation, bad file design(initially), numeric labels (initially).

2.8 C

C (1971)

2.9 PL/M (1973)

PL/M

```
FIND: PROCEDURE(PA,PB) BYTE;
    DECLARE (PA,PB) BYTE;
    /* FIND THE STRING IN SCRATCH STARTING AT PA AND ENDING AT PB */
    DECLARE J ADDRESS,
        (K, MATCH) BYTE;
    J = BACK ;
    MATCH = FALSE;
        DO WHILE NOT MATCH AND (MAXM > J);
        LAST,J = J + 1; /* START SCAN AT J */
        K = PA ; /* ATTEMPT STRING MATCH AT K */
            DO WHILE SCRATCH(K) = MEMORY(LAST) AND
                NOT (MATCH := K = PB);
            /* MATCHED ONE MORE CHARACTER */
            K = K + 1; LAST = LAST + 1;
            END;
        END;
    IF MATCH THEN /* MOVE STORAGE */
        DO; LAST = LAST - 1; CALL MOVER;
        END;
    RETURN MATCH;
    END FIND;
```

iRMX

2.10 CLU

CLU (MIT), 1974-1975)

(special mention: iterators)

2.11 Modula-2

Modula 2 (1977-1985)

==> Modula 3

==> Oberon

2.12 Ada

Ada (Standard 1983)

Different numeric base syntax:

```
2#01000111
8#770077
16#12AE    == #12AE
10#2021    == 2021
```

2.13 C vs Pascal

2.14 Object oriented languages

2.15 Other languages

I will also refer to Java Basic (1964) APL (1960s) Forth (1970) Postcript

Languages Timeline

TIOBE index

Chapter 3

Compilers and Interpreters

Interpreted vs Compiled Programming Languages: What's the Difference?

They define

> In a compiled language, the target machine directly translates the program. In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code.

Typical *compiled* languages: C, Pascal, Fortran...

Typical *interpreted* languages: Basic, APL, Lisp et al, Javascript et al, PHP, Perl, dBase, SQL, ...

The breakdown is *iffy*

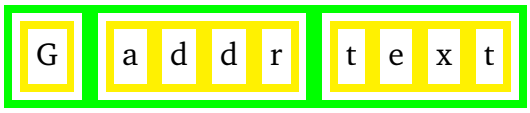
Interpreters are compilers .. a bit

Many interpreters include partial compilation for performance reasons.

Example: tokenization in Basic

```
    GOTO 123

123  LET X=5
```

Actual representation in memory



where *G* is one-byte code for the `goto` keyword, followed by the address of the source line 123 in memory, followed by the text form of the label – not read during the execution, except for the first time.

Other tricks include:

- replacing variables to pointers to symbol table
- converting the program to tree structure (cf. scanner+parser)
- moving bulk of the computations to system routines (APL)

is C compiled?: consider

```
int x,y;

printf("x=%d,y=%d\n",x,y);
```

"compiled" would imply conversion into machine code like

```
push offset str1 ; x=
call print_string
```

```
push x
call print_int
push offset str2 ; ,y=
call print_string
push y
call print_int
call print_nl
```

but this is not possible, the template string may not even exist during the compilation:

```
int x,y;
char *get_template() { ... }

printf(get_template(),x,y);
```

Thus, `C` actually includes a mini-interpreter for `printf`....

(strictly speaking, one can eliminate `printf` – do not view it as a part of the language. This would also kill `cout` and streams. but this cannot be done with `format` in `FORTRAN` – this is a keyword!)

More interesting case: assume an `exec("...");` construct in a language.

A pseudo-C example:

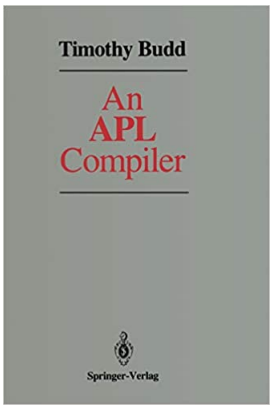
```
int x,y,z;
x=1;
y=2;
exec("z=x+y;");
printf("z=%d",z);
```

This cannot be compiled in principle, thus any language that has exec("..") feature cannot be compiled!

(Naturally, `C` does not have it, but `APL`, `dbaseII`, had!)

With dBase/Clipper the solution was to compile what was possible, but also embed a full scale interpreter into the executable.

With APL, I doubt any sensible solution exists, albeit there is a book with a highly suspect title:



This is how *exec* actually looks in APL.

```
⍎'A←B+C'
```

But it gets worse

```
M←⎕CR FUNC

....

⎕FX M
```

And yet worse: dynamic types

```
A ←1

A ←1 2 3 4 5
```

```
A ←'Hello World'

A ←2 2 2 2⍴0
```

Where does this place Java and p-System?

Chapter 4

Overall structure of a compiler

We can divide a compiler in four ways, into:

Phases : a phase is a logical operation on the information, phases will be described below.

Passes : a pass is a rewrite of the information from one format to (usually) another. One phase may include multiple passes, but multiple phases may be combined into one pass.

Modules : phases + other components of the software (that do not do processing of the program per se)

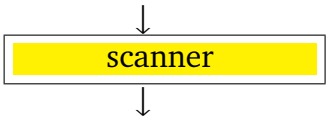
Ends : Front End and Back End. Front End is language dependent, target independent; Back End is language-independent, target dependent.

We notice that it is possible to have a single Front end being accompanied by multiple Back Ends (multi-target compiler); but it is also possible to have several Front Ends sharing the back end.

For example, the original Microsoft's Pascal compiler for IBM PC included two files: `pas1.exe` and `pas2.exe`; their Fortran compiler included files `for1.exe` and `pas2.exe`, and indeed both `pas2.exe` were identical.

MetaWare's Professional Pascal and High C compilers shared the backends too, albeit this was not so openly displayed.

4.1 Scanner



Scanner – also known as tokenizer – is responsible for converting a string of input characters into a stream of words (tokens).

Example:

```
for i:=1 to 10 do write("Hello World!");
```

becomes

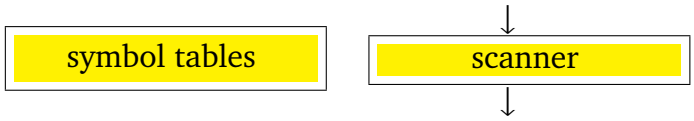
for | i | := | 1 | to | 10 | do | write | (| "Hello World!" |) | ;

we are not discussing the actual representation now.

we are also not discussing how to read the source file and how to store the tokens.

Scanner is a *phase*, but usually not a path (rather an input reading function under control of the parser).

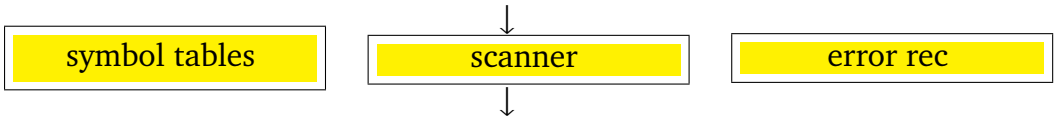
4.2 Symbol table(s?)



Symbol tables are a small database that keeps track of located symbols (keywords, identifiers, etc).

We will discuss this in details later.

4.3 Error recovery



Exercise: *What kind of errors are possible during scanning?*

- `123abc`
- `1e2e3`
- `"hello,world!`
- `/* comment not closed`
- `$` (invalid character)

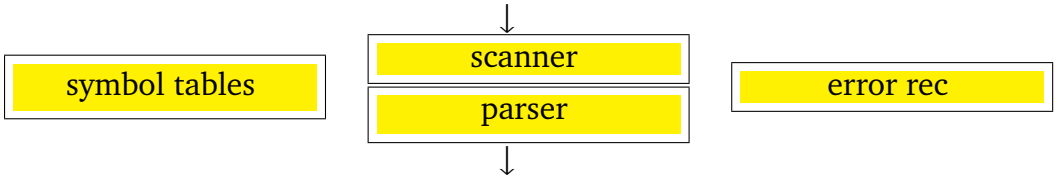
On the other hand, `end begin` is a valid Pascal program (from the point of view of scanner), so is `1 2 3 4 5` (also a valid C program).

`}{` is a valid C program (but not a Pascal one!).

Bad news: proper error recovery is very difficult, especially with scanner errors.

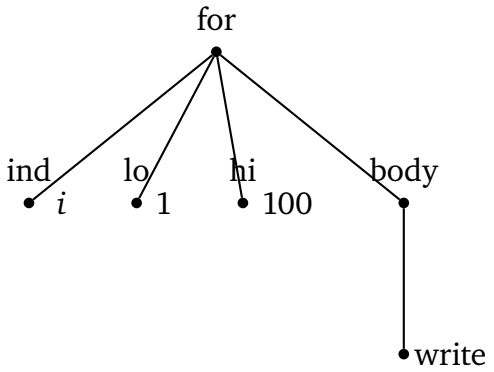
Good news: we really would not have to do this.

4.4 Parser

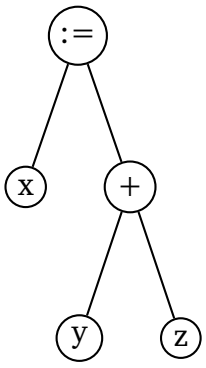


Example:

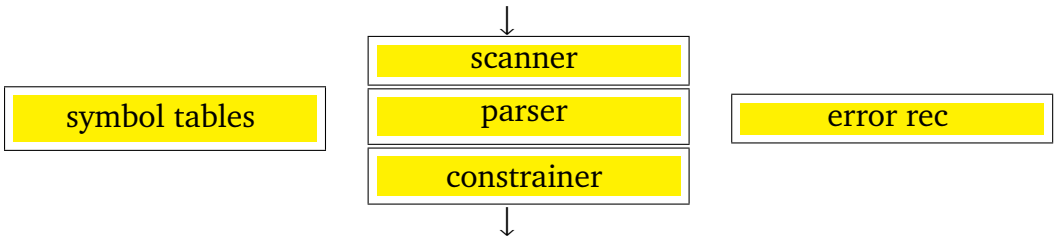
```
for i:=1 to 10 do write("Hello World!");
```




```
x:=y+z;
```



4.5 Constrainer (Semantics)

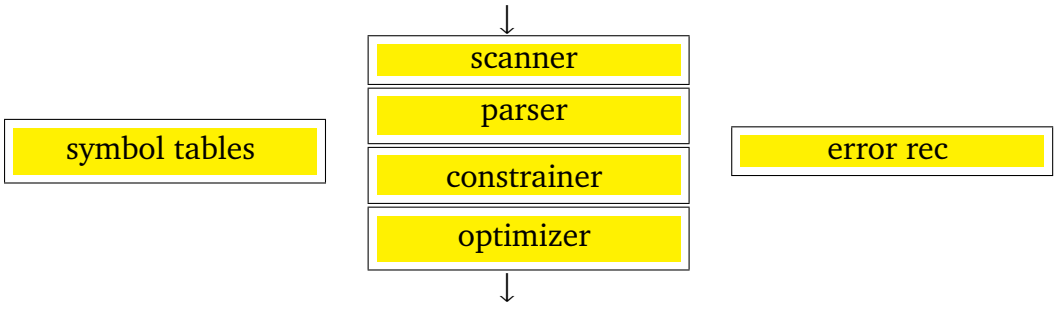


The above tree may or may not represent valid code, depending on the types of the variables.

While type checking is the main purpose of this phase, other checks may be performed, for example, range:

```
x:=sqrt(-1);
```

4.6 Optimizer



Optimizer performs different subtasks that may require different data representation; thus it is may include many passes.

4.6.1 Strength Reduction

example:

```
var x,y:integer;

y:=2*x;
```

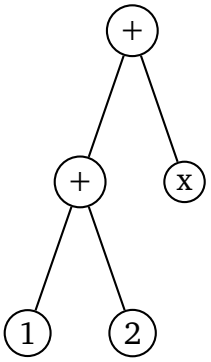
What about `n*x` for other small values of n?

Likewise, division and mod can be optimized if the 2nd argument is a power of 2.

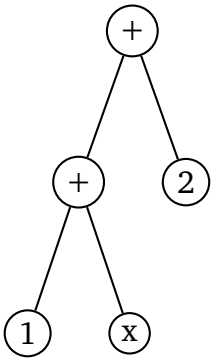
4.6.2 Constant folding

```
var x,y:integer;


y:=1+2;  {TP/D: YES}

y:=1+2+x; {TP/D: YES}

y:=1+x+2; {TP/D: NO}

y:=x+1+2; {TP/D: NO}
```

compare:



and



Optimizations of this type can done relatively easily if a structural tree is indeed constructed – but nearly impossible for a *one-pass compiler* that does not build a tree.

How can we check if a compiler does this type of optimization? Look at the assembler listing – if the compiler creates it – or under a debugger, if it does not.

BCC does create such a listing (compile with -S option) and we see that it indeed folds constants.

```
   ;    x=1+y+2;
   ;

@1:
        mov       eax,dword ptr [_y]
        add       eax,3
        mov       dword ptr [_x],eax
```

Here is another snippet that illustrates what is done and what is not:

```
   ;
   ; x=y*2;
   ;
 ?debug L 13
 mov       ecx,eax
 add       ecx,ecx
 mov       dword ptr [_x],ecx
   ;
   ;
   ;
   ; x=y+y+y+y+y+y+y+y;
   ;
 ?debug L 16
 mov       ecx,eax
 add       ecx,eax
 add       ecx,eax
 add       ecx,eax
```

```
add       ecx,eax
add       ecx,eax
add       ecx,eax
add       ecx,eax
xor       eax,eax
mov       dword ptr [_x],ecx
  ;
  ;
```

Constant folding should be done using the target arithmetic!

What about this?

```
x:= 4*y - 3*y;
```

4.6.3 CSE

```
a := b * c + g;
d := b * c * e;
```

can be replaced by effectively

```
temp := b*c ;
a := temp + g;
d := temp * e;
```

(Notice that compiler optimization will likely accomplish more than human's!)

```
  var a,b: array [0..100] of integer;
      i: integer;

  a[i*3+1]:=b[i*3+1];  {perhaps in a loop}
```

4.7 Dead code elimination-1

```
   a:=b+c;

   goto lab;

   a:=b-c;  {never executes!}

lab:
```

4.8 Dead code elimination-2

```
   a:=b-c;  {result is never used}

   a:=b+c;
```

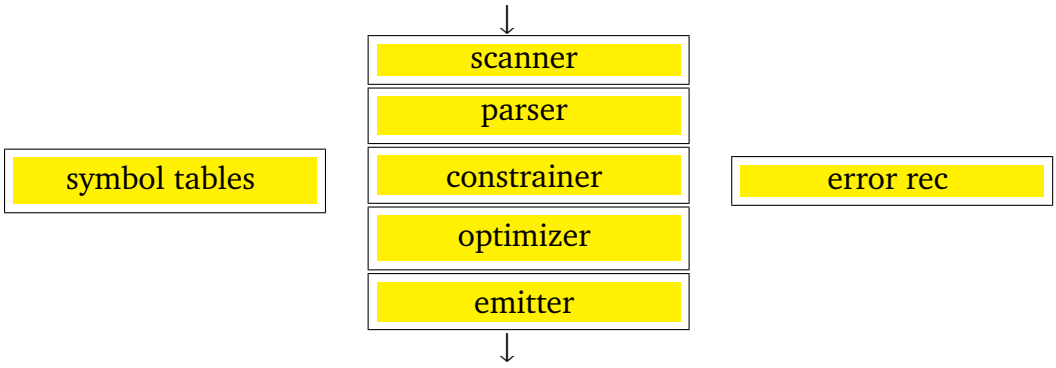
4.9 Dead code elimination-3

```
procedure p; {never called!}
begin


end;
```

4.10 Can optimizer produce errors?

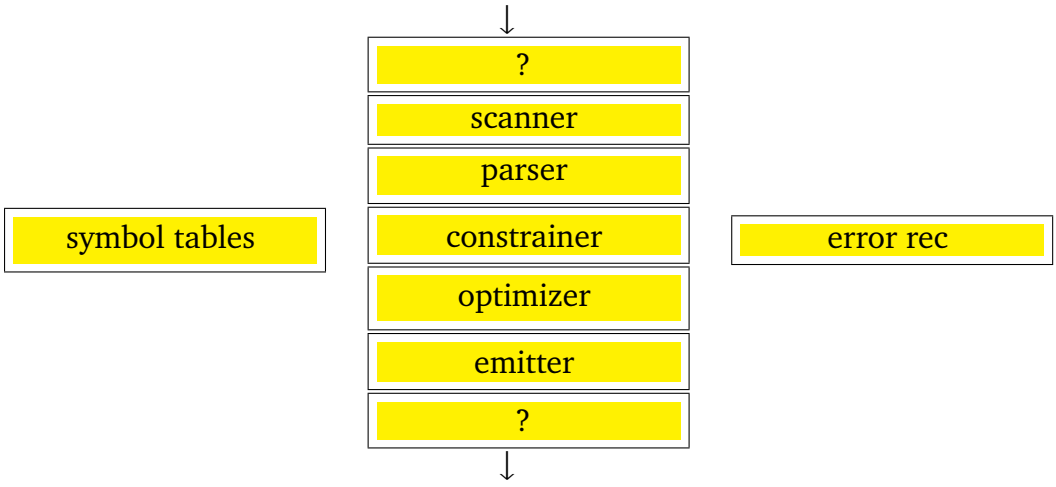
```
x:=y div (3-3);
```

4.11 Emitter (Code Generator)

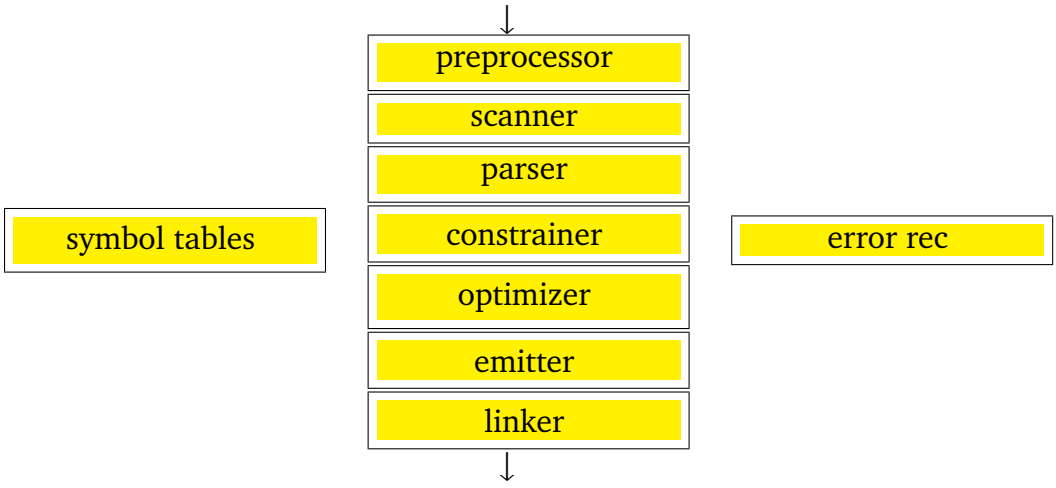


(Emitter may be multi-pass to optimize jumps; it may be logically divided into two phases : Jump Fixup and Emitter proper.

4.12 What is missing?



4.13 Let's add


preprocessor
scanner
parser
symbol tables
constrainer
error rec
optimizer
emitter
linker

Chapter 5

Scanner

5.1 Overall structure

We assume that we need only one token at a time (in fact this is almost always the case), cf. names of the parsing methods: **LL(1)**, **SLR(1)**, **LALR(1)** – in all cases **(1)** means that decisions can be made based on one token!

To avoid passing the only token we are working with, we will store it in a global variable(s), so every parser routine has access to it. The following structure will be capable of describing all tokens we may need:

```
  int curtoken; // main value: token type
  int curvalue; // token value
// for unresolved yet identifiers
  char curname[MAXNAME];
// primarily for diagnostics
  int curfile; // file index
  int curline; // current line number
  int curcol; // current column number
```

(your programming style is your choice – if you prefer `struc` or a `class`,

please yourself!)

We will assume that the scanner is a procedure (void function) `gettoken()` which will be called repeatedly to fill the current token values.

We will also assume that one of possible token types is `TK_EOF`, returned when the input ended.

Finally we assume that we will have a debugging routine `prnttoken()` that can print tokens in symbolic form.

Thus, we can describe a working tokenizer (scanner-only compiler) as

```
initialize();  // open files

do {
  gettoken();
  printtoken();
  }
while (curtoken != TK_EOF);
```

Effectively, this is the first part of building a compiler!

5.2 Token types

Exercise: *what kind of tokens may occur in a typical programming language?*

- keywords (reserved words) (`for`)
- user-defined identifiers (`xyz123`)
- operators and punctuation (`+`, `;`)
- integer constants
- real constants
- character constants (???)
- string constants
- options / pragmas (???)
- end of line (???)
- end of file (`TK_EOF`)

5.3 Representation

For simplicity and efficiency we keep the token type (`curtoken`) as an integer (or if you like, `enum`).

We now will describe the needed token values.

5.3.1 End of file

```
#define TK_EOF 0
```

We don't use the value field.

5.3.2 End of line

```
#define TK_EOLN 1
```

We don't use the value field.

(This token type is not needed for most languages, including C and Pascal)

5.3.3 keywords

We will assign a different token type for each keyword to avoid unneeded comparisons

```
#define TK_BEGIN 2
#define TK_END 3
#define TK_IF 4
...
#define TK_WHILE 30
```

We don't use the value field.

5.3.4 operators

We don't use the value field.

```
#define TK_PLUS 40
#define TK_MINUS 41
...
#define TK_AND 30  /* && in C */
```

```
#define TK_BAND 31  /* & in C */
...
#define TK_COMMA 50
#define TK_COLON 51
...
#define TK_LBRACE /* { in C */
#define TK_RBRACE /* } in C */
```

Notice that `TK_AND` is an operator in C but keyword in Pascal – but beyond the scanning there is no difference.

Pascal's `TK_BEGIN` is of course identical to C's `TK_LBRACE`.

5.3.5 integer constant (literal)

```
#define TK_INTLIT 100
```

Token value should have the actual value (read from the program source).

5.3.6 char constant (literal)

```
#define TK_CHARLIT 101
```

Token value should have the actual value (read from the program source), for example `'A'` should result in 65.

Note: this token type is actually unnneeded for either Pascal or C, albeit for different reasons. In C character constant can be treated as `int`, in Pascal, character constant can be seen as a string of length 1.

5.3.7 real constant (literal)

```
#define TK_REALLIT 102
```

Token value can pack the real value itself (in which case its type needs to be changed), or — better – be an integer index into an array of all different real constants encountered.

Token value is a pointer to a string constant in heap – or better – index into a pool of all strings.

5.3.8 string constant (literal)

```
#define TK_STRLIT 103
```

5.3.9 non-keyword identifiers

This is one part that scanner cannot fully do. We would like to set up token types like

```
#define TK_A_VAR 201
#define TK_A_CONST 202
#define TK_A_TYPE 203
#define TK_A_FUNC 204
...
```

but scanner cannot determine what is what... to do this one needs the parser reading the declarations!

Thus, for now :

```
#define TK_UNKNOWN 200
```

with the `curname` field holding the actual identifier's name.

5.3.10 pragma

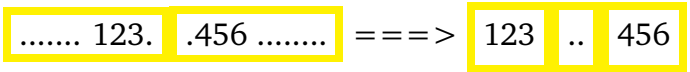
(leaving this to the reader)

5.4 How to read the source file(s)?

Possible techniques are

1. read it character by character, cf. `getc()` .. and `\ungetc()` – inefficient.
2. read it line by line, using standard I/O library (`getline()` et al) – inefficient, problematic with alternative concepts of text files (see below), problems with not knowing the structure of lines. Is having standard text input library a good idea?

- read it as a binary file, one buffer at a time – efficient, but may cause problems with overshoots. Consider (Pascal example)

  We do not know if 123 is a an integer or a real constant without having to load the next buffer, but then we may need to backstep!

3. circular buffer – resolves this issue.
4. read file at once and find the lines yourself. Sample `C` code

```
FILE *f=fopen("mysource.cpp","rb");
if (!f)
  error("Source file not found");
fseek(f,0,SEEK_END);
int sz=ftell(f);
fseek(f,0,SEEK_SET);
unsigned char *buff=new unsigned char[sz+1];
fread(buff,sz,1,f);
fclose(f);
buff[sz]=0;
```

5 same, using `unix.h` functions (`open`,`read`,...)

6 same, using WinAPI functions.

7 use memory mapped files (exist under both Windows and Unix).

Methods 4-7 are equivalent from the point of work : in call cases we end up with the pointer to the beginning of the buffer and – if we need – the size.

We assume the following global scanner variables :

```
unsigned char *scanp;  //(init to buff)
int lin; //current line, init 1
int col; //current col, init 1
```

If multiple source files are to be handled, we will need a stack of these variables.

5.5 How to scan a token?

The basic idea, applicable to most languages[1], is that the first character of the token defines the token.

[1] (not FORTRAN)

We can now write |gettoken()|, the overall scheme is

```
void gettoken()  {
unsigned char c;
restart:
switch (c=*scanp++) {
  case LETTER:  // a..z, A..Z, and possibly other chars
    // return one of the keyword tokens, or TK_UNKNOWN (id, not a keyword)
  case DIGIT:
   //  return TK_INTLIT or TK_REALLIT
  case PUNCT_OR_OPER:
   //  return one of the operator tokens or skip comment and restart
  case QUOTE:
  //   return TK_STRLIT
  case SPACE:
       goto restart;
  case NEWLINE:
  // update line number
       goto restart; // or return TK_EOLN
  case 0:
  //   return TK_EOF
   }
```

The cases are language-specific and need to be worked out. `col` needs to be updated too – if precise error diagnostics is wanted, I leave this to the reader.

5.6 How to scan an identifer token?

We look at `case LETTER` in more details.

- copy characters that may occur in an identifier to `curname`, truncating those beyond `MAXNAME`, while advancing `scanp`; at the end of this step `scanp` should point to the first character beyond the identifier.

- convert `curname` to upper (or lower) case, if the language is case-insensitive.
- (we can insert something else here – but later :P)
- search the keyword symbol table only and return a keyword token if found. *return* here and elsewhere means set `curtoken`, other fields if needed and then return from the function[2].
- return `TK_UNKNOWN`

5.7 How to scan a numeric constant token?

A numeric constant, whether integer or floating point, *in Pascal* always begins with an integer number (in C, an additional case needs to be considered, in addition to what we do here)

- `int value=0; int base=10;`
- char c; //init to the 1st, already read digit.
- `while (c>='0'&&c<='9') { value=value*base+c-'0'; c=*scanp++; }`
- check for `E`, `#`, `.`,... ; if no modifiers noticed, save `value` into `curvalue` and return `TK_INTLIT`.
- otherwise, scan the rest of the number.

Notice that the exact scanning rules are language-dependent! for example, in Pascal, decimal dot (`.`) must always be followed by a digit, in `C` this is optional.

Other quircks may occur, for example leading `0` in C.

[2]this implies that the keyword symbol table is separate from the other symbol tables and may be implemented differently, for example inline!

To scan an ADA-style number (`base#digits`) one can use exactly the same loop as above, with `10` replaced by `base`.

To scan the fractional digits in a floating point constant one can use nearly the same loop, but keep the result as float and divide it by 10 after new digit is added.

5.8 How to scan an operator or punctuation sequence?

Divide the problem into a large number of subscases, based on the first character. The subscases are generally trivial, but language dependent.

Example: Assume the first scanned character is <.

Which tokens in C may begin with it?

(answer: <, <<, <=)

Which tokens in Pascal may begin with it?

(answer: <, <>, <=)

Example: Assume the first scanned character is `.`.

In C this would indicate either a single dot token, or a beginning of a floating point number (if digit follows)

In Pascal this would indicate a single dot (`.`), or range (`..`) or `.)` (square bracket substitute).

In most languages some operating sequences begin a comment. In C such are `//` and `/*`, in Pascal they are `{` and `(*` and sometimes ADA-style end-of-line comment `--`).

Comments usually do not produce tokens, one needs to skip them and restart scanning.

In Pascal, comments are often used to embed options (language extensions), such comments begin with the `$` option.

Example:

```
(* Normal comment -- skip it *)

(*$R+*)  {enable range check}

{$I morecode.pas} {include file}
```

5.9 How to scan an string?

Every language has its quirks.

In C, a string is enclosed in double quotes, escapes (`\..`) are used to embed special characters or the double quote itself into the string.

In Pascal, a string is enclosed in single quotes, if a single quote is needed inside the string, it is repeated twice. (Example:`'Don''t do it'`). Borland had its own extensions to embed arbitrary characters.

In C string may come in several parts, in Pascal this is not allowed.

5.10 How to scan EOLN?

Even if you do not return `TK_EOLN`, you need to recognize end-of-line character to correctly maintain the line and column numbers.

Keep in mind that there is no uniform definition of the end-of-line in text files, this is system-dependent!

Generally:

- Unix : line feed (lf) : (10 dec, `0x0A`, `\n`)
- Classic Mac/OS : carriage return (cr) : (13 dec, `0x0D`, `\r`)
- Dos/Win : cr, followed by lf; cr followed by multiple lf's are acceptable and sometimes occur.

Thus a program that reads all these variants should recognize what it is dealing with.

More information than you want..

5.11 How to add include ability?

(This is a part of the preprocessor, and in fact the easiest to implement: historically it was the first part implemented in C, to quote "The Development of the C Language" by Dennis M. Ritchie):

> Many other changes occurred around 1972-3, but the most important was the introduction of the preprocessor, partly at the urging of Alan Snyder [Snyder 74], but also in recognition of the utility of the the file-inclusion mechanisms available in

BCPL and PL/I. Its original version was exceedingly simple, and provided only included files and simple string replacements: `#include` and `#define` of parameterless macros. Soon thereafter, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation. The preprocessor was originally considered an optional adjunct to the language itself. Indeed, for some years, it was not even invoked unless the source program contained a special signal at its beginning. This attitude persisted, and explains both the incomplete integration of the syntax of the preprocessor with the rest of the language and the imprecision of its description in early reference manuals.

It can be done as a separate program (as in C initially) or integrated into the scanner as we will describe here.

Review our scanner template:

```
void gettoken()  {
unsigned char c;
restart:
switch (c=*scanp++) {
  case LETTER:  // a..z, A..Z, and possibly other chars
    // return one of the keyword tokens, or TK_UNKNOWN (id, not a keyword)
  case DIGIT:
   //  return TK_INTLIT or TK_REALLIT
  case PUNCT_OR_OPER:
   //  return one of the operator tokens or skip comment (1a) and restart
  case QUOTE:
  //   return TK_STRLIT
  case SPACE:
       goto restart;
  case NEWLINE:
  // update line number
       goto restart; // or return TK_EOLN
  case '#':  (1b)
```

```
case 0: (2)
//   return TK_EOF
 }
```

We add:

- stack of input files (buff,scanp,filename,linenum,colnum fields)
- if we want to handle pascal-style includes (f.e. `{$i myinclude.pas}`, at (1a) we check for comment beginning with `$i`, read the file name, scan to the end of the comment, push the current file info on the stack, switch to the new file and return to `restart` – we still need to return a token.
- alternatively, if we prefer C-like syntax `#include "myinclude.h`, we check at (1b) for `include`, read the file name, scan to the end of the line, push the current file info on the stack, switch to the new file and return to `restart` – we still need to return a token.
- we also modify end-of-file handling: if the file stack is not empty, rather than returning `TK_EOF`, we will simply pop off the previous include file of the stack and resume (goto restart); only if the stack is empty `TK_EOF` is returned.

MetaWare compilers supported also a `#c_include` directive. `c` is for *conditional*, with the meaning: include the file only if it were not previously included. This is trivial to add.

(Why is this helpful?)

(How is this managed without this directive?)

What do you think about compiling `myprog.cpp` like this:

```
#include "myprog.cpp"
```

How can we implement prep's conditional compilation?

Chapter 6

Symbol tables

The issues we discuss now:

- searching
- representation
- compiled symbol tables
- one table or many (scopes)

We do not discuss the exact information to be stored, instead we design with sufficient generality to support anything we may possibly need.

This will be done at a very low level – not just because it is more effecient but because we have to learn how to work with low-level structures anyway.

For me, symbol table will be just raw memory – or a character array – and we'll put our own structure on it.

One extra benefit is the easiness of saving/loading of compiled symbol tables.

If you don't like what I'm doing use dicts or even SQL ... it may even work ...

6.1 Compiled symbol tables?

Goal : avoid repeated readings of often huge include files.

C solution: precompiled headers. For a taste:

4 Ways Precompiled Headers Cripple Your Code.

Cause: total lack of enforced structure in header files. A clean solution is, however, possible (Java, or Pascal Units).

6.2 Linear search.

The dictionary entries are arranged as a single link list.

We will show how a symbol table may look like.

Meaning :

- link – link fields contain offsets of the next entry, the list ends with the link field being set to the terminator (perhaps, `-1`).

- key – contains the key part of a dictionary entry as a pascal (length-first) string.
- data – always contains the token type (`TK_BEGIN`, `TK_END`,etc) and may contain additional information after it.

Link field can be 2-byte or 4-byte, 2-byte links offer more compact tables but limit their size to 64kb. (not enough for `win.h`)

Links can run forward (as shown) or backward - the second appoach actually works better (scopes).

Because of its low performance ($O(N^2)$ on a sample that contains N keys) this searching method should not be used and is shown only as an illustration.

6.3 Binary trees

A binary tree is

> In computer science, a binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

Binary tree symbol table can be build by repeating additions of nodes to an empty tree; the first node becomes the root, the subsequent nodes are added by

0 set current node to root

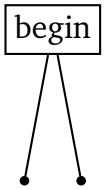
1 compare the new key with the key of the current mode

2 if equal, node is already present, return

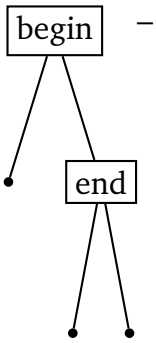
3 if less, if left child is present, set current node to left child and go to step 1; otherwise append new node as the left child and return;

4 otherwise (if greater), if right child is present, set current node to right child and go to step 1; otherwise append new node as the right child and return;

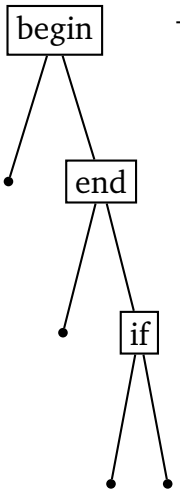
We will show the process of building a tree for a sample input **begin**, **end**, **if**, **repeat**, **while**, **until**, **else**, **do**



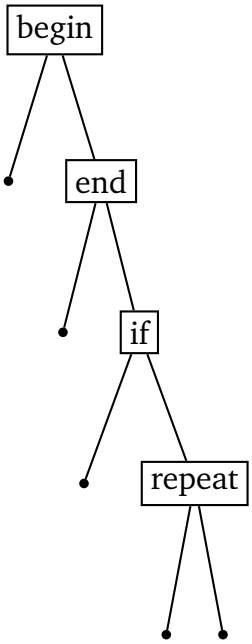
– after adding **begin**.



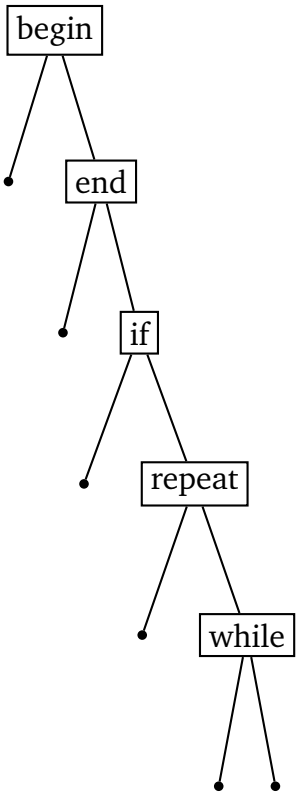
– after adding **end**.



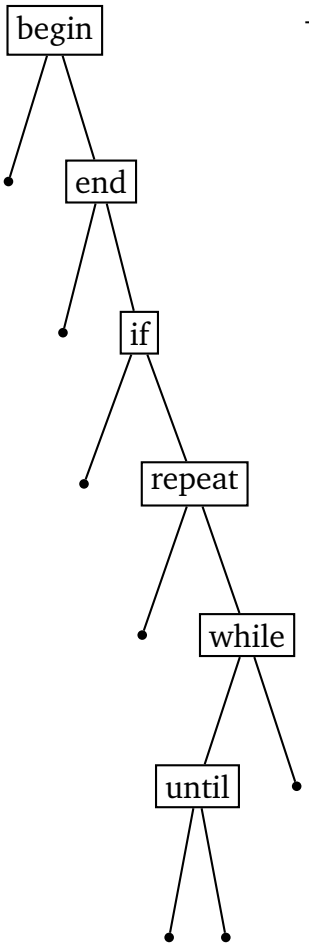
– after adding **if**.



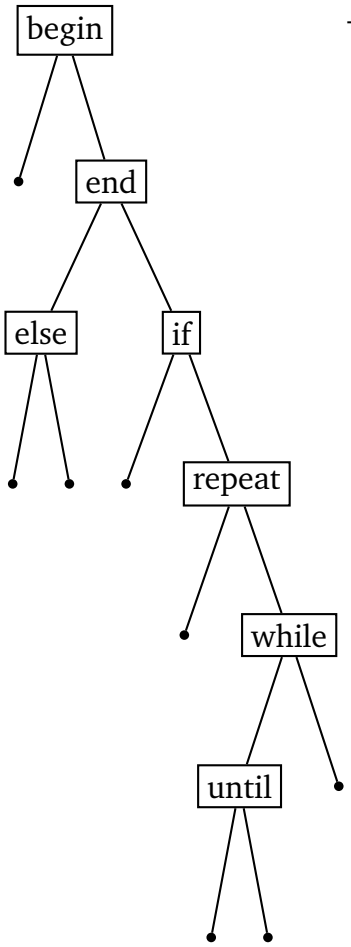
– after adding **repeat**.



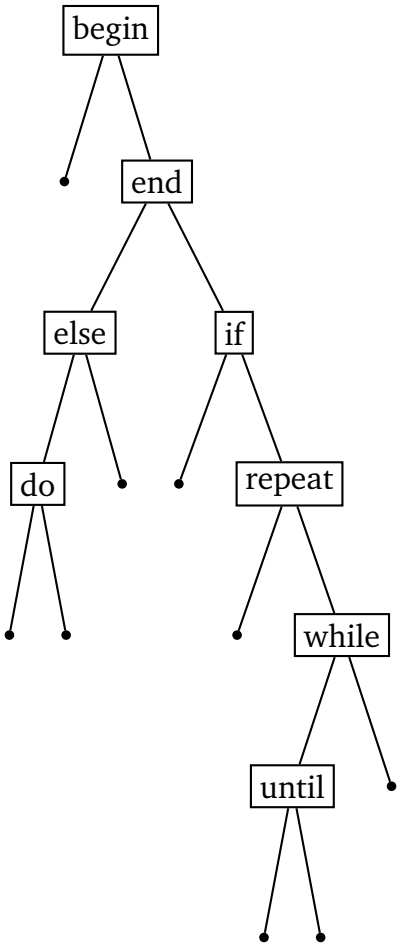
– after adding **while**.



– after adding **until**.



– after adding **else**.



– after adding **do**.

The performance of the resulting tree depends on how well it is balanced, ranging from $O(n)$ for unbalanced tree to $O(\log N)$ for a balanced. This translates to $O(n^2)$ and $O(n \cdot \log N)$ for a full run.

Algorithms exist to balance trees

AVL trees

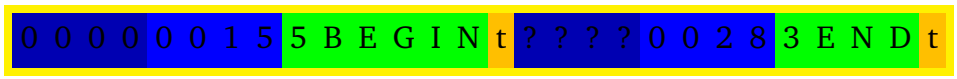
Red Black trees

Fun questions:

- how to produce a tree that is just a linked list (bad case?)
- how to produce a tree that is balanced (assuming you know the keys in advance?)
- how to dump a sorted symbol table?
- do you really need to balance trees?

6.4 Byte packed binary tree symbol table

We can adapt linear list bpst by providing two link fields, for left and right kids:



6.5 Hashing

The idea is to compute the location of the key in the symbol table in constant (or, close to constant) time.

Constant time is achieved in the original basic where the identifiers are limited to single letters or letters followed by digits.

This, obviously, cannot be done when the number of potential identifiers is large.

We define a hash function $h(s)$ that maps the set of keys to the integer range $0..(H-1)$, H is an integer number indicating the number of "buckets". For each possible hashed value we form a separate linked list.

For illustration, we will assume that the language is case-insensitive (uses only upper case letters) and define $h(s) = s[0] -' A'$, $h()$ can produce

numbers in the 0..25 range and thus at most 26 buckets.

Some of the resulting link lists will be

0 **AND** $\Rightarrow \emptyset$

1 **BEGIN** $\Rightarrow \emptyset$

2 $\emptyset$

3 **DO** $\Rightarrow \emptyset$

4 **END** $\Rightarrow$ **ELSE** $\Rightarrow \emptyset$

5 **FOR** $\Rightarrow \emptyset$

...

Good hashing function will provide uniform distribution of data in buckets.

In the worst case scenario (achieved with $h(s) = 0$) hashing deteriorates into linear list.

How to choose a good value of H? Aim for the link lists to be at most 2-3 element-long, with a hust function that uniformly distributes the keys, H therefore should be about max $/ 3$ where max is the largest expected number of keys.

Sample functions (we assume $l = length(s)$ below.

- $h(s) = 0$ – horrible.
- $h(s) = s[0]$ – bad
- $h(s) = \sum_{i=0}^{l-1} s[i] \mathrm{mod} H$ – bad
- $h(s) = \sum_{i=0}^{l-1} s[i] \times 2^i \mathrm{mod} H$ – ok
- $h(s) = s[0] + s[len - 1] + s[(l/2)]$ – MetaWare function[1].

[1] we don't include this part in Notes.

Per Knuth, choosing prime value of H will result in most uniform distibution for a polynomial hashing function; the author prefers powers of 2. Values 256-512-1024 should be considered for most applications.

Polynomial hash function can be computed without exponents or multiplications. Consider:

```
// compute hash of string s;
//
char *s=.... ;
unsigned h=0;
while (*s)
  h=h+h+*s++;
h=h % H;
// use h=h & (H-1) if H is a power of 2.
```

6.6 Byte packed binary hashing table

Binary packed hash table should begin with the array that contains the starts of link lists, the array will consist of H entries (4*H bytes).

It will be immediately followed by packed data in the linear search model, described above.

The offsets in such table may be computed relative to the beginning of the entire array or relative to the beginning of the linear list portion.

6.7 How to write integer types into a byte array?

Problem: write `short x` into character array `char b[]`, beginning with position `i`.

Slow solution:

```
char b[N];
short x;

b[i]=x / 256;      // or x >> 8
b[i+1]=x % 256;    // or x & 255
```

This will always work.

Exercises:

1. Read the data back. 2. Do this for int (4 bytes)

Faster solution:

```
char b[N];
short x;

*(short*)(b+i) = x;
```

(this will fail on most RISC chips)

6.8 How many tables?

In order of being checked, the following tables may occur in a compiler:

- macros – used by scanner or prep
- keywords – used by scanner
- local tables (correspond to nested procedures, scopes (C), `with` (Pascal)
- global tables (correspond to units and globally declared variables) – unlike local tables there may be ambiguities in the search order!

We do not consider macro and keyword tables here: they are handled differently (scanner!) and contain different types of data too. Thus, they should be separate tables.

Possible approaches to implementing other tables:

- separate tables – arranged in a linked list. (Advantage: simpler implementation. Disadvatage: performance, especially with C scopes : every `{` opens a new table?)
- single table with a save/restore mechanism (Disadvatage: more complicated but faster!)
- combination appoach (some separate, some combined tables) – not examining it here.

Generally, separate tables are more suitable for Pascal (compilation units!), single - for `C` – because of very deep nesting possible.

with multiple tables we maintain a stack of tables and search them all (or just the top one in some cases).

with single table we need a safe/restore mechanism.

Example:

```
int i;

function p() {   // <<==push
  float i;

  cout << i;


  }             //  <<==pop
```

with the multiple table approach,`push` means create new table and push it on the stack of the symbol tables. `pop` means remove it.

with the single table approach,`push` means initialize save/restore mechanism, so that new copy of `i` does not permanently kill the previous one; `pop` means restore all the saved entries and destroy all the entries declared in the most recent scope.

(a very simple and neat algorithm for save/restore exists for binary packed tables that does not require labelling entries with scopes).

6.9 How to check symbol tables?

One does not need a working compiler, only a somewhat working scanner.

Symbol tables are applicable to many other applications, including, for example, spelling checker.

A scanner can be run on a normal text file and made to suppress (ignore) all the non-word tokens. Feeding the tokens into symbol table machinery will produce a dictionary of all the words used!

Chapter 7

Context-free grammars

(For additional information and historical reference)

CFG

Context-free grammar consists of

- set of terminal symbols $\mathcal{T}$
- set of non-terminal symbols $\mathcal{N}$
- set of productions $\mathcal{P}$, each is a string in the format $N \Rightarrow \mathcal{T} \cup \mathcal{N}^*$
- start symbol $\mathbf{S} \in N$

For our purposes $\mathcal{T}$ means tokens, $\mathcal{N}$ syntactical elements of the language, **S** defines a full program (or compilation unit).

7.1 Example: Fully matched strings

$$\mathcal{T} = \{'(', ')'\}$$

$$\mathcal{N} = \{M\}$$

$$\mathcal{P} = \begin{cases} M \Rightarrow' (' \ M \ ')' \\ M \Rightarrow \epsilon \end{cases}$$

$$\mathbf{S} = M$$

ϵ denotes an empty string.

This notation is actually incorrect – we should have written instead

$$\mathcal{T} = \{\texttt{TK\_LPAREN}, \texttt{TK\_RPAREN}\}$$

$$\mathcal{N} = \{M\}$$

$$\mathcal{P} = \begin{cases} M \Rightarrow \texttt{TK\_LPAREN} \ M \ \texttt{TK\_RPAREN} \\ M \Rightarrow \epsilon \end{cases}$$

$$\mathbf{S} = M$$

We will, however, use shorter (and incorrect) notation and even omit quotes for brevity. We note that we can only specify $\mathcal{P}$ when describing grammars and combine Right Hand Sizes for compactness, getting very brief description.

$$M \Rightarrow (\ M \ ) \ \mid \ \epsilon$$

7.2 derivations

Starting with **S** and applying productions repeatedly until we obtain a string that consists only of nonterminals is called a *derivation*

$M \rightarrow (M) \rightarrow ((M)) \rightarrow (((M))) \rightarrow ((()))$

The language defined by a CFG consists of all strings that can be derived from the start symbol, in our example it is

ϵ, `()`, `(())`, `((()))` ...

7.3 Example: Matched strings

Slightly more interesting example:

$M \Rightarrow (\ M\ )M \quad | \quad \epsilon$

will enlarge the set to the set of matched strings to include, for example, `(())()`.

This string can be derived, for example, with

$M \rightarrow$ `(M)M` $\rightarrow$ `(M)(M)M` $\rightarrow$ `(M)(M)` $\rightarrow$ `(M)()` $\rightarrow$ `((M)M)()` $\rightarrow$ `((M))()` $\rightarrow$ `(())()`

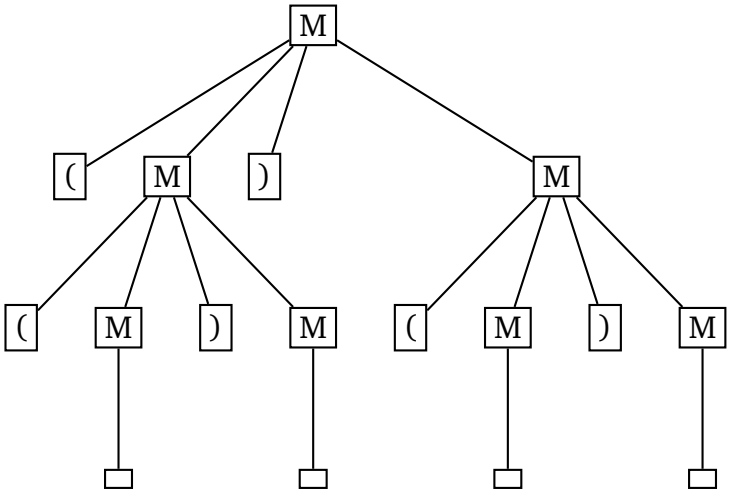
---

or

$M \rightarrow$ `(M)M` $\rightarrow$ `((M)M)M` $\rightarrow$ `(()M)M` $\rightarrow$ `(())M` $\rightarrow$ `(())(M)M` $\rightarrow$ `(())()M` $\rightarrow$ `(())()`

Both derivations work, but for our purposes having multiple derivations may created an abmiguity. We therefore will consider only left-canonical derivations; the 2nd example is such.

One way to remove the worries about possible ambiguity here is to use a parse tree instead:



Exercise: *Unrelated, but nice exercise: how many matched strings of length 2n exist?*

7.4 Expressions

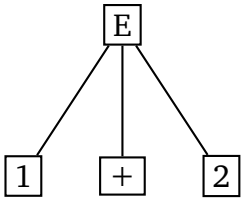
A sample Expression grammar:

Grammar :

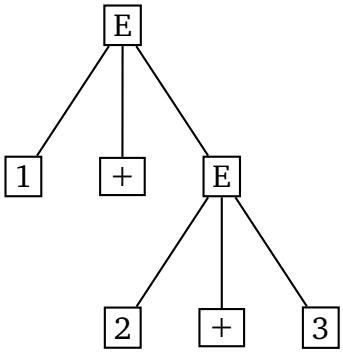
$E \Rightarrow \text{LIT} + \text{LIT} \quad | \quad \text{LIT}$

Important : LIT refers to a token of `TK_INTLIT` type, we disregard everything but the token type during parsing. Thus, `0`, `-1` and `2021` are seen as the same terminal symbol.

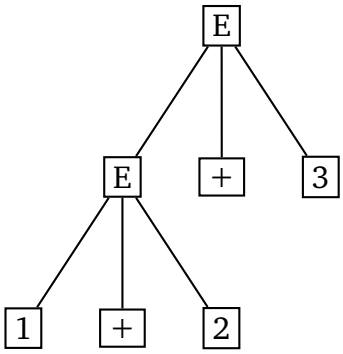
Expression `1+2` can be derived with this parse tree:



Expression `1+2+3` can be derived with this parse tree:



or this one:



A CFG is ambiguous if one or more terminal strings have multiple leftmost derivations from the start symbol.(Equivalently: multiple rightmost derivations, or multiple parse trees.)

How bad this is?

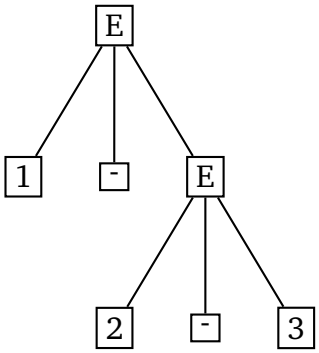
In general :

- ambiguous grammars lead to ambiguous understanding of text (in our case: program).
- in our specific case it creates an ambiguity in the evaluation order (see below)
- some grammars are ambiguous (expressions), some not (matched strings)
- the problem of finding out if a grammar is ambiguous or not is *undecidable*

Does the order of evaluation matter? Consider a variant of the grammar that uses subtractions rather than addition:

$E \Longrightarrow \text{LIT} - \text{LIT} \quad | \quad \text{LIT}$

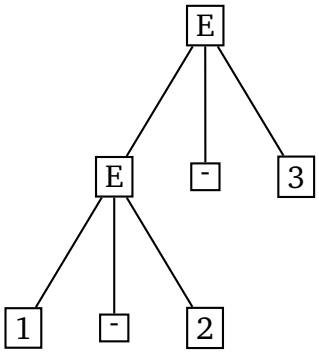
Expression `1+2+3` can be derived with this parse tree:



(Result = 2)

---

or this one:



(Result = -4)

But then for addition we will get 6 either way.

7.5 Is addition associative

.

Because addition (unlike subtraction) is associative?

But is this really true? Is it true that this program (in SOME language) will print `FALSE`?

```
number x,y,z;

x=... ;
y=... ;
z=... ;

number result1 = (x+y)+z;
number result2 = (x+y)+z;

if (result1==result2)
  output ("TRUE");
else
  output ("FALSE");
```

Exercise: *Think about it!*

Basic idea: Let's prove that $1 = 0$:P.

Let $x = -10^{10}, y = 10^{10}, z = 1$.

$x + y$ would result in 0, so $(x + y) + z = 1$.

$y + z$ would result in y (insufficient precision), so $x + (y + z) = 0$

The example that proves that addition is not always associative, in C, is shown below.

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

int main() {

float x,y,z,r1,r2,r3;

x=-10E10;
y=10E10;
z=1;

r1=(x+y); //printf("\n%20.5f",r1);
  r2=r1+z; printf("   (x+y)+z = %20.5f",r2);
r1=(y+z); //printf("\n%20.5f",r1);
  r3=x+r1; printf("   x+(y+z) = %20.5f",r3);

return 0;
}
```

Output:

```
(x+y)+z =  1.00000   x+(y+z) =   0.00000
```

Fine points:

- While variables are declared as `float`, no fractions appear in computations, all values are really integers.
- This is not a function of a particular C compiler, but rather of IEEE arithmetic.
- To see what is really going on, uncomment the commented out output.
- Some compilers (including BCC used here) may do CSE optimization that may result in the same results – if both outputs are computed in the same or adjacent expressions.
- For better feeling, experiment.

This all brings up a question: is the addition commutative? The author does not know the answer.

For practical issues: we do assume that addition is commutative and associative (cf. the CSE logic).

7.6 Cause of the ambiguity

The grammar does not specify the order of the evaluation of addition, it should, but let's first define the turns.

- Operation $\diamond$ on set X is said to be associative if $\forall a, b, c \in X \quad (a \diamond b) \diamond c = a \diamond (b \diamond c)$.
- Operation $\diamond$ on set X is said to be left-associative is $a \diamond b \diamond c$ means $(a \diamond b) \diamond c$
- Operation $\diamond$ on set X is said to be right-associative is $a \diamond b \diamond c$ means $a \diamond (b \diamond c)$

Notice the different nature of the definitions! The first deals with the actual properties of it,the other two with our interpretation of *inherenty ambigous notation a-b-c* !

Most of operations in programming languages are left-associative, but not all.

Exercise: *Which operations are right-associative?*

- `**` in FORTRAN, and likewise `^` in BASIC.
- `=` (assignment) in C and related languages, consider `x=y=z`.
- **All** binary operations in APL.

In APL, `1-2-3` results in `2`, not `¯4`.

Conclusion: We need to know how to specify left- and right- associativity of binary operations in grammars we use.

Overall approach for such problems : we must stick to grammars that are not ambiguous. This can only be done by sticking to subsets of general CFG. Such subsets exist and include LL(1) and SLR(1).

In our Expression example: grammar is badly designed, we must have one that is not ambiguous – and thus must enforce the order of evaluation.

7.7 How to fix the grammar?

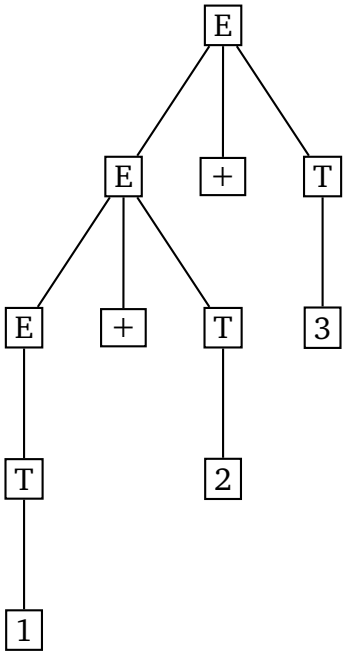
The following grammar will ensure that addition is left-associative:

$E \Rightarrow E + T \quad | \quad T$

$T \Rightarrow LIT$

(T stands for *term*)

Expression `1+2+3` can be derived with this parse tree:



There is no other way to draw a parse tree – this can be shown *for this example* by manual parsing.

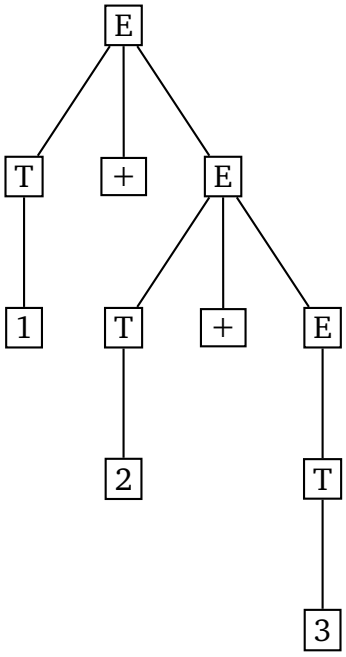
For right-associative operations we will use this template instead:

$E \Rightarrow T + E \quad | \quad T$

$T \Rightarrow LIT$

Then,

Expression `1+2+3` can be derived with this parse tree:



Notice that the complexity of the tree grew and the parse tree is no longer a convenient to work with data structure.

7.8 Extending the grammar

We can supplement our grammar with subtraction and ability to work with variables (`TK_A_VAR` tokens)

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Rightarrow LIT \quad | \quad VAR$

The grammar so far is problem-free and you can try manual parsing with it.

The next operation to include is multiplication, and we can try

---

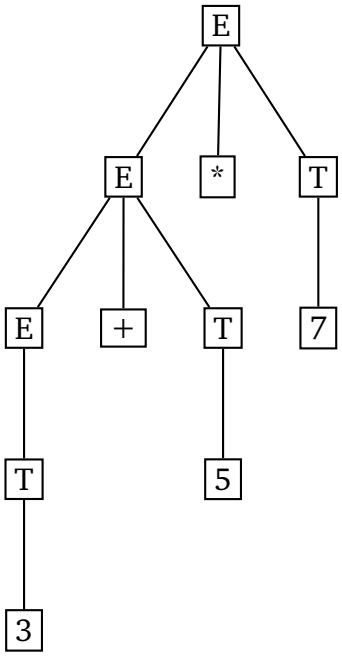
Grammar :

$$E \Longrightarrow E + T \quad | \quad E - T \quad | \quad E * T \quad | \quad T$$

$$T \Longrightarrow LIT \quad | \quad VAR$$

(It should be obvious without any examples that this grammar is incorrect, but we will supply an example to demonstrate this)

Consider `3+5*7`. There is one and only one parse tree possible.



It should not be surprising that evaluation accordingly to this grammar would produce 56 rather than expected 38.

Cause: we neglected to define the precedence of operations; multiplication should be done first (has higher precedence order). The problem is obvious once one notices that the grammar above treats both operations equally.

Before fixing this grammar, let's consider APL: in this language **all** binary operations have the same precedence and **all** are evaluated right to left.

Then we can include all of them at once:

Grammar APL:

$E \Rightarrow T + E \quad | \quad T - E \quad | \quad T \times E \quad | \quad T \div E \quad | \quad T \leftarrow E... \quad | \quad T$

$T \Rightarrow LIT$

And yes, in APL `5×3+2` is indeed 25.

Reason: nobody would be able to remember precedence rules for APL, it has just too many operators!

For "conventional" language precedence is introduced by an additional level in the grammar:

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Rightarrow T * F \quad | \quad T / F \quad | \quad F$

$F \Rightarrow LIT \quad | \quad VAR$

(`F` stands for *Factor*)

We will add a few more features and then attempt manual parsing.

With unary operators (only minus shown):

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Rightarrow T * F \quad | \quad T / F \quad | \quad F$

$F \Rightarrow \quad | \quad -F \quad | \quad LIT \quad | \quad VAR$

With parenthesis we have recursion:

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Rightarrow T * F \quad | \quad T / F \quad | \quad F$

$F \Rightarrow \quad | \quad -F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$

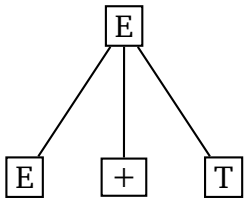
Additional operators can be added on the same lines (if they have the same precedence as those already in) or, otherwise, by adding more levels.

For example, Pascal's `div` and `mod` will not need new levels, they go together with `*` and `/`. Neither will `or` and `and`. On the other hand, C's `&&` and `||` will indeed need new levels (and the total number of levels in `C` is higher! – do you remember all the rules?)

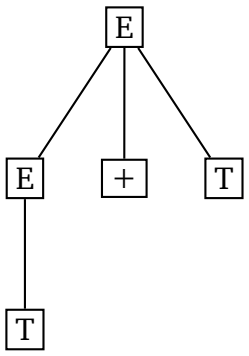
7.9 Manual parsing

We will now attempt to manually parse `3*4+5*7`.

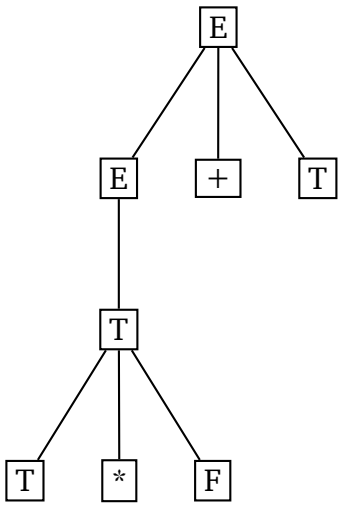
1. We notice that we must use $E \Rightarrow E + T$ for the first step in derivation.



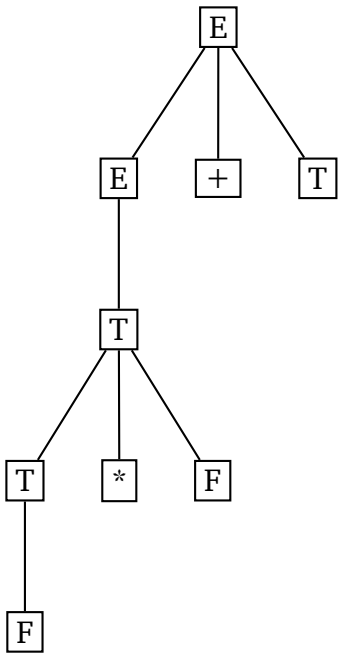
2. We must use $E \Rightarrow T$ for the leftmost unopened nonterminal (E)



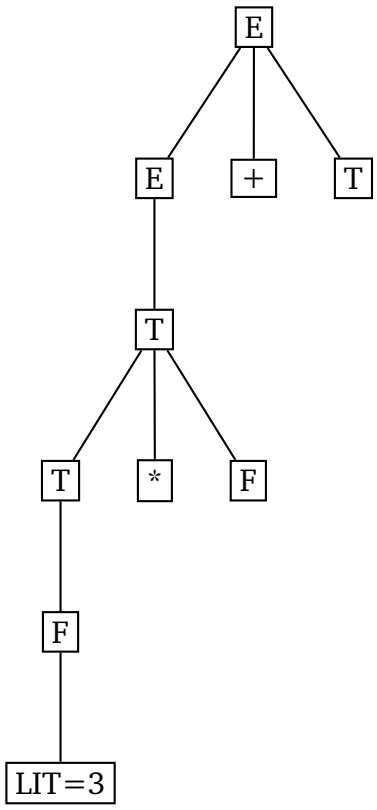
3. We must use $T \Rightarrow T * F$ for the leftmost unopened nonterminal (T)



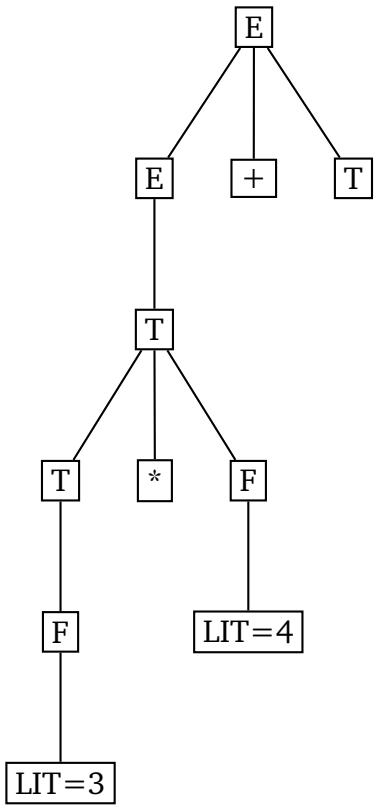
4. We must use $T \Rightarrow F$ for the leftmost unopened nonterminal (T)



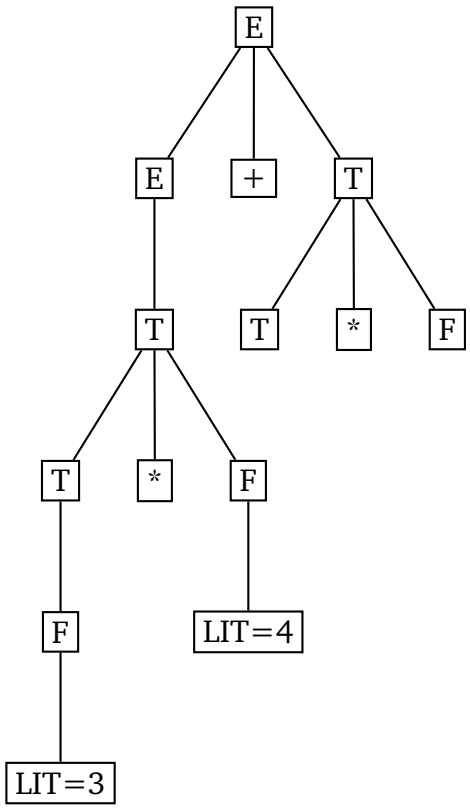
5. We must use $F \Rightarrow LIT$ for the leftmost unopened nonterminal (F)



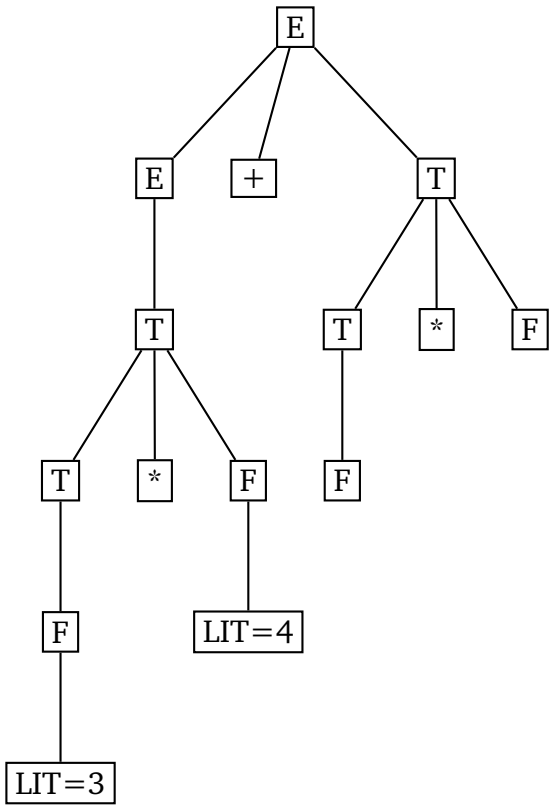
6. We must use $F \Rightarrow LIT$ for the leftmost unopened nonterminal (F)



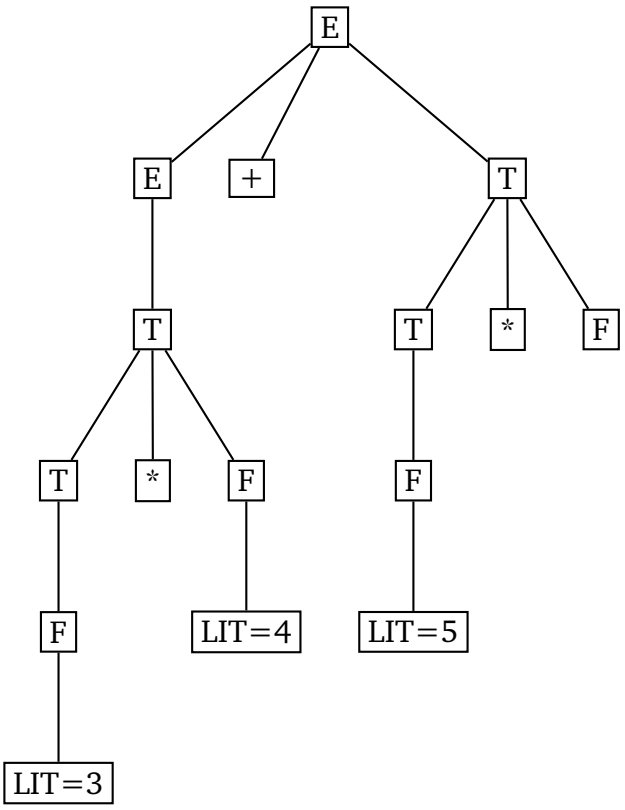
7. We must use $T \Rightarrow T * F$ for the leftmost unopened nonterminal (T)



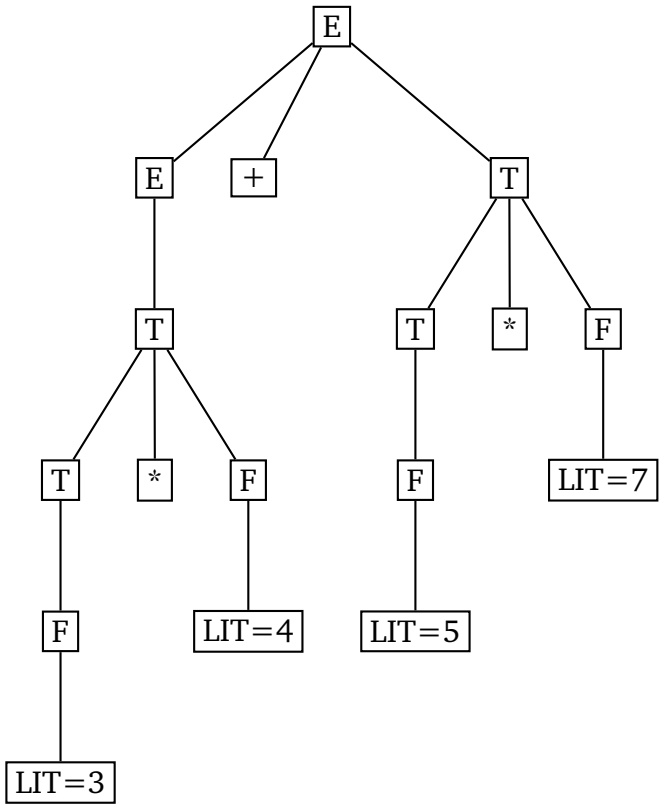
8. We must use $T \Rightarrow F$ for the leftmost unopened nonterminal (T)



9. We must use $F \Rightarrow LIT$ for the leftmost unopened nonterminal (F)



10. We must use $F \Rightarrow LIT$ for the leftmost unopened nonterminal (F)



This is the final tree – no "unopened" nonterminals remain.

7.10 Extending grammar further

(We are doing this only for Pascal : C grammar would be larger and have additional levels)

We have already:

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Rightarrow T * F \quad | \quad T / F \quad | \quad F$

$F \Rightarrow \quad | \quad -F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$

Operators `div`, `mod`, and non-canonical `shl`, `shr` all have the same precedence as multiplication:

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Rightarrow T * F \quad | \quad T / F \quad | \quad T$ **div** $F \quad | \quad T$ **mod** $F \quad | \quad T$ **shl** $F \quad | \quad T$ **shr** $F \quad | \quad F$

$F \Rightarrow \quad | \quad -F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$

Operator `or` (and non-canonical `xor`) has the same precedence as addition, operator `and` – as multiplication, operator `not` – as negation. Pascal also allows unary + (does not do anything). Thus

---

Grammar :

$E \Rightarrow E + T \quad | \quad E - T \quad | \quad E$ **or** $T \quad | \quad E$ **xor** $T \quad | \quad T$

$T \Rightarrow T * F \quad | \quad T / F \quad | \quad T$ **div** $F \quad | \quad T$ **mod** $F \quad | \quad T$ **shl** $F \quad | \quad T$ **shr** $F \quad | \quad T$ **and** $F \quad | \quad F$

$F \Longrightarrow \quad | \quad +F \quad | \quad -F \quad | \quad \textbf{not}\ F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$

(This is not a complete expression grammar – but close. What is missing?)

It is also not the final grammar – we will have to rearrange it for building a parser yet. For such work we will revert to the smaller grammar at the beginning of the section.

Chapter 8

Parsing

8.1 Trivial TDRD parser

The grammar for expressions developed so far is not suitable for building a parser – but to understand why we should attempt this anyway, and to illustrate the process we will return to much simpler matched parenthesis grammar:

Grammar :

$M \Longrightarrow (\ M\ )M \quad | \quad \epsilon$

The basic idea: we transform a grammar into a set of procedures (void functions per C), where

- Each production becomes a procedure.
- The LHS of a production becomes the name of the procedure

- The RHS of a production becomes the body of the procedure
- The nonterminals in the RHS become calls to the procedure for that nonterminal
- The terminals in the RHS become calls to the `match(terminal)` function shown below.

The grammar above thus becomes

```
void M() {
  match(TK_LP);
  M();
  match(TK_RP);
  M();
  }

void M() {
  }

void match(TOKEN t) {
  if (curtoken!=t)
     error();
  else
     gettoken();
  }
```

We will supplement this code with the scanner and the main:

```
  int main() {
    initialize();
    gettoken();
    M();  // this should invoke the goal symbol!
    }
```

In the form given the code is not even compilable: we have multiple procedures with the same name, there should be a way to decide which of the procedures to call. Ideally, the decision can be made by examinining the current token.

An alternative way to write our prototype would be combine the multiple RHS procedures into one:

```
void M() {
  do either {
     match(TK_LP);
     M();
     match(TK_RP);
     M();
     }
  or {
     }
  }
```

And then replace `do either .. or` by some *real* code, and this is what we try now.

```
void M() {
  if (curtoken==TK_LP) {
     match(TK_LP);
     M();
     match(TK_RP);
     M();
     }
  else {
     }
  }
```

We test the approach with the complete program below. It includes a rudementary scanner and extra code for printing out the call tree.

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

#define TOKEN unsigned char
#define TK_LP '('
#define TK_RP ')'
#define TK_EOF 0

unsigned char curtoken;
unsigned char *scanp;

int indent=0;

void printindent(){
 printf("\n  ");
 for (int i=0; i<indent; i++)
   printf("..");
}

void gettoken();

void error() {
  printf("\n\n\n Syntax error!"); exit(-1);
  }


void match(TOKEN t) {
  if (curtoken!=t)
     error();
  else {
   printindent(); printf("%c",curtoken);
   gettoken();
  }
  }
```

```
void M() {
  printindent(); printf("M",curtoken);
  indent++;
  if (curtoken==TK_LP) {
     match(TK_LP);
     M();
     match(TK_RP);
     M();
     }
  else {
     }
  indent--;
  }


void gettoken() {
  if (*scanp=='(') { scanp++; curtoken=TK_LP; }
    else
  if (*scanp==')') { scanp++; curtoken=TK_RP; }
  else
    curtoken=TK_EOF;
  }


void main(int argn, char ** argv) {

  if (argn<1) return;

  scanp=argv[1];

  gettoken();

  M();
}
```

The input string should be supplied on the command line. Assuming that this program is called `mp`, we try it with different strings, first a valid one:

```
C>mp (())()

  M
  ..(
  ..M
  ....(
  ....M
  ....)
  ....M
  ..)
  ..M
  ....(
  ....M
  ....)
  ....M
```

then invalid

```
C>mp (())(

  M
  ..(
  ..M
  ....(
  ....M
  ....)
  ....M
  ..)
  ..M
  ....(
  ....M
```

```
 Syntax error!
```

and another invalid:

```
C>mp ((()))

  M
  ..(
  ..M
  ....(
  ....M
  ....)
  ....M
  ..)
  ..M
```

Notice that the 2nd attempt produced an error message while the third did not. Why?

The issue is best corrected by modifying the grammar (and then adding new production to the code).

Grammar :

$M \Longrightarrow (\ M\ )M \quad | \quad \epsilon$

$G \Longrightarrow M\textbf{EOF}$

G is the new Goal symbol. The resulting program will detect all errors.

This very issue resurfaces in compilers for "real" languages. A minimal Pascal program is

---

```
  begin
  end.
```

Is the following program valid?

```
  begin
  end.
Some junk just for fun of it.
```

Oddly, there is no consistent answer. The author believes it is not valid and will ensure this by checking for `TK_EOF`. (Metaware compilers shared this belief.)

Trivial improvements to the program left to the reader:

- point to the exact place where the error occurred.
- support brackets `[`, `]` in addition to the parenthesis.
- print the call tree in a graphical way, suitable for a book illustration. (well,... this is a little less trivial).

It is very important to notice that call tree *looks* just like the parse tree.

8.2 Adapting expression grammar

We will use a simplified version of the grammar, everything we do can be adapted to a more complete grammar.

Grammar :

$$E \Longrightarrow E + T \quad | \quad E - T \quad | \quad T$$

$$T \Longrightarrow T * F \quad | \quad T / F \quad | \quad F$$

$$F \Longrightarrow +F \quad | \quad -F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$$

In informal way we used before we can try implementing `E()` as

```
void E() {
  do either {
     E();  // ***
     match(TK_PLUS);
     T();
     }
  or {
     E();  // ***
     match(TK_PLUS);
     T();
     }
  or {
     T();
     }
  }
```

No matter how we implement |do either|, entering the lines marked with *** would be suicidal.

This is not the only problem, but this is fatal. In general the problem is known as left recursion and affects all grammars that contain productions in the form

---

Grammar :

$A \Rightarrow A \ldots\ldots\ldots$

The good news is that left recursion can be corrected and easily; and doing this – miracleously – tends to fix other problems that are possible and even occur in our grammar!

8.3 Correcting left recursion

Assume a grammar contains a left-recursive instance

Grammar :

$A \Rightarrow A\alpha$

(We use Greek letters are used to denote strings of terminals and non-terminals).

We can assume that there is also a non-left-recursive option for A; if this is not the case, then A cannot ever generate a string of terminals and we can simply delete it from the grammar! Thus:

Grammar :

$A \Rightarrow A\alpha \quad | \quad \beta$

(The general case is when A has one or more left-recursive options, and one or more non-left-recursive options:

Grammar :

$A \Rightarrow A\alpha_1 \quad | \quad A\alpha_2 \quad | \quad A\alpha_3 \ldots$

$A \Rightarrow \beta_1 \quad | \quad \beta_2 \quad | \quad \beta_3 \ldots$

---

but since the proof of the conversion we will show for the one-and-one case can be easily extended to it, so we will do the simple case only.

Any derivation from A would use left-recursive production zero or more times and then stop by using the non-recursive option:

$A \rightarrow A\alpha \rightarrow A\alpha\alpha \rightarrow A\alpha\alpha\alpha \rightarrow \beta\alpha\alpha\alpha$

Thus, we can generate strings in the format $\beta alpha^n$ ($n \geq 0$) and no others.

We can generate exactly the same strings with another grammar that is not left-recursive:

Grammar :

$A \Rightarrow \beta A'$

$A' \Rightarrow \alpha A' \quad | \quad \epsilon$

---

and this alternative grammar will not suffer from the left-recursion problem.

(In the general case, new grammar should offer options for all possible α's and β's.)

Grammar :

$A \Rightarrow \beta_1 A' \quad | \quad \beta_2 A' \quad | \quad \beta_3 A' \ldots$

$A' \Rightarrow \alpha_1 A' \quad | \quad \alpha_2 A' \quad | \quad \alpha_3 A' \ldots \quad | \quad \epsilon$

A' is usually referred to as $A-$tail.

Other possible problems in the grammar include:

- non-immediate left recursion:

Grammar :

$A \Rightarrow B\beta$

$B \Rightarrow C\gamma$

$C \Rightarrow A\alpha$

There is an algorithm for resolving this issue; we don't discuss it since the author is not aware of this coming up in "real-life" grammars.

- Similar RHS. How do we know which production to choose between

---

Grammar :

$A \Rightarrow B\beta_1$

$A \Rightarrow B\beta_2$

---

this can be fixed in the obvious way; and this does occur in our expression grammar.

Inability to make a decision without looking far ahead: this does occur in our grammar and is related to the previous issue.

The good news is that these problems will dissappear on their own once we correct left recursion.

Rewriting of the expression grammar would result in :

Grammar :

$E \Longrightarrow TE'$

$E' \Longrightarrow +TE' \quad | \quad -TE' \quad | \quad \epsilon$

$T \Longrightarrow FT'$

$T' \Longrightarrow *FT \quad | \quad /TF \quad | \quad \epsilon$

$F \Longrightarrow +F \quad | \quad -F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$

Notice that we had to correct productions for E and T, but not F. Notice that the new grammar obtained a very useful property: when there are multiple options for the production, all of them except for at most one start with different terminals; this suggests we can write a deterministic parser.

Here is a sample of the code that can be used to implement a TDRD[1] parser for this grammar

[1]Top-Down Recursive-Descend

```
  void E() {
    T();
    Etail();
    }

  void Etail() {
    switch(curtoken) {
      case TK_PLUS:
         match(TK_PLUS);T(); Etail(); break;
      case TK_MINUS:
         match(TK_MINUS);T(); Etail(); break;
      default:
         ;
      }
    }

....
```

We leave the task of implementing the other procedures to the reader. Together with the code used in the matched parenthesis parser (shown above) this would result in a functional deterministic expression parser.

A sample call tree is shown below.

```
C:>me /t 2*3+4*86

  G
  ..E
  ....T
  ......F
  ........LIT=2
  ......T'
  ........*
  ........F
  ..........LIT=3
```

```
........T'
....E'
......+
......T
........F
..........LIT=4
........T'
..........*
..........F
............LIT=86
..........T'
......E'
..#EOF#
```

The grammar used was supplemented with the top-level production

---

Grammar :

$G \Rightarrow E$ **EOF**

$E \Rightarrow TE'$

$E' \Rightarrow +TE' \quad | \quad -TE' \quad | \quad \epsilon$

$T \Rightarrow FT'$

$T' \Rightarrow *FT \quad | \quad /TF \quad | \quad \epsilon$

$F \Rightarrow +F \quad | \quad -F \quad | \quad (E) \quad | \quad LIT$

(*VAR* option omitted, EOF checking added)

Another example, here we test unary minus:

```
C:>me  /t ----999

  G
  ..E
  ....T
  ......F
  ........-
  ........F
  ..........-
  ..........F
  ............-
  ............F
  ..............-
  ..............F
  ................LIT=999
  ......T'
  ....E'
  ..#EOF#
```

and parenthesis

```
C:>me  /t (91-18)*3

  G
  ..E
  ....T
  ......F
  ........(
  ........E
  ..........T
  ............F
```

```
.............LIT=91
............T'
..........E'
............-
............T
..............F
................LIT=18
..............T'
............E'
........)
......T'
........*
........F
..........LIT=3
........T'
....E'
..#EOF#
```

Notice that the call tree (which is the same as the parse tree) became yet longer.

8.4 Decorated grammars

Despite the apparent success we are not finished with the grammar writing, a couple more rewrites follow.

The first is to address our goal: it is conversion of the code and not just validation. Thus we are stepping beyond formal languages now.

We extend the definition of CFG by adding new set of symbols, **actions**, that may appear in the right hand side of productions. Actions play no role in parsing; they can be seen as additional calls that can be added to a TDRD parser. The grammar below our expression grammar with actions added.

Grammar :

$G \Rightarrow E$ **EOF**

$E \Rightarrow TE'$

$E' \Rightarrow +T$ [ADD] $E' \quad | \quad -T$ [SUB] $E' \quad | \quad \epsilon$

$T \Rightarrow FT'$

$T' \Rightarrow *F$ [MUL] $T \quad | \quad /T$ [DIV] $F \quad | \quad \epsilon$

$F \Rightarrow +F$ [NOP] $\quad | \quad -F$ [NEG] $\quad | \quad (E) \quad | \quad LIT$ [LIT]

Actions can be used for different purposes which we will shorly figure

out, but for now, let us just look at a sample expression parsion with added actions that only print their names (and for LIT, the actual value).

```
C:>me  /e /t 3*7+4*5

  G
  ..E
  ....T
  ......F
  ........LIT=3
  ->->->-> 3
  ......T'
  ........*
  ........F
  ..........LIT=7
  ->->->->-> 7
  ->->->-> MUL
  ........T'
  ....E'
  ......+
  ......T
  ........F
  ..........LIT=4
  ->->->->-> 4
  ........T'
  ..........*
  ..........F
  ............LIT=5
  ->->->->->-> 5
  ->->->->-> MUL
  ..........T'
  ->->-> ADD
  ......E'
  ..#EOF#
```

Let us look at the output generated:

```
3 7 MUL 4 5 MUL ADD
```

Any guesses about what this is ?

8.5 Possible interpetations of the output

1 **Postfix (Polish) expression form.**

This is not useful for our course.

2 **Postscript / FORTH translation.**

This is not useful for our course.

3 **Stack machine assembler code**

```
C:>me  /e3 3*7+4*5

      PUSH 3
      PUSH 7
      MUL
      PUSH 4
      PUSH 5
      MUL
      ADD
```

This is not useful for our course.

4 **Stack machine assembler code in binary form**

This is the primary option for our implementation; such code can be run on a p-Code machine. We will discuss how it looks later.

5 **Native assembler code**

This is a possible option for our implementation. Sample:

```
C:>me  /e5 3*7+4*5

      PUSH 3
      PUSH 7
      ; stack SUB (4 lines)
      POP EAX
```

```
POP EBX
IMUL EBX
PUSH EAX
PUSH 4
PUSH 5
; stack SUB (4 lines)
POP EAX
POP EBX
IMUL EBX
PUSH EAX
; stack ADD (4 lines)
POP EAX
POP EBX
ADD EAX,EBX
PUSH EAX
```

The code is of course suboptimal but it can be improved (peephole optimization).

6 **Native machine code**

Same as the previous option except in binary; we do not demonstrate this.

7 **Tree builder**

We maintain a stack of trees, initially empty. Operation `LIT` creates new subtree. Binary operations combine trees.

This can be used in a multi-pass compiler. The example below shows the construction,

subtrees are shown in functional notation, f.e. `ADD(4,5)` denotes a tree with root `ADD` and kids `4` and `5`.

```
C:\>me  /e7 3*(-716)+4*5

 after VAL: 3
 after VAL: 716   3
```

```
 after NEG: NEG(716)   3
 after MUL: MUL(3,NEG(716))
 after VAL: 4   MUL(3,NEG(716))
 after VAL: 5   4   MUL(3,NEG(716))
 after MUL: MUL(4,5)   MUL(3,NEG(716))
 after ADD: ADD(MUL(3,NEG(716)),MUL(4,5))
```

8 Expression evaluator

Perform actions “real-time”.

This is not useful for our course, but perhaps for something else? Sample run, showing the stack after each operation:

```
C:>me  /e8 3*7+4*5

 after VAL:  3
 after VAL:  3  7
 after MUL:  21
 after VAL:  21  4
 after VAL:  21  4  5
 after MUL:  21  20
 after ADD:  41
```

9 Tree drawer

We can emit code needed to draw a tree (cf. PSTREE)

This is not useful for our course... only for the notes :P.

Not a very readable dump of PSTREE code

```
C:\>me  /e9 3*(-716)+4*5

 after VAL: \pstree{\Tbx{3}}{}   \par
 after VAL: \pstree{\Tbx{716}}{}
    \pstree{\Tbx{3}}{}   \par
 after NEG: \pstree{\Tbx{-}}{\pstree{\Tbx{716}}{}}
    \pstree{\Tbx{3}}{}   \par
```

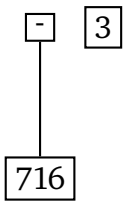
```
after MUL: \pstree{\Tbx{*}}{\pstree{\Tbx{3}}{}
   \pstree{\Tbx{-}}{\pstree{\Tbx{716}}{}}}   \par
after VAL: \pstree{\Tbx{4}}{}   \pstree{\Tbx{*}}{
   \pstree{\Tbx{3}}{}\pstree{\Tbx{-}}{\pstree{
\Tbx{716}}{}}}   \par
after VAL: \pstree{\Tbx{5}}{}   \pstree{\Tbx{4}}{}
   \pstree{\Tbx{*}}{\pstree{\Tbx{3}}{}
\pstree{\Tbx{-}}
{\pstree{\Tbx{716}}{}}}   \par
after MUL: \pstree{\Tbx{*}}{\pstree{\Tbx{4}}{}
   \pstree{\Tbx{5}}{}}   \pstree{\Tbx{*}}{
\pstree{\Tbx{3}}{}
\pstree{\Tbx{-}}{\pstree{\Tbx{716}}{}}}   \par
after ADD: \pstree{\Tbx{+}}{\pstree{\Tbx{*}}{
   \pstree{\Tbx{3}}{}
   \pstree{\Tbx{-}}{\pstree{\Tbx{716}}{}}}
\pstree{\Tbx{*}}{
\pstree{\Tbx{4}}{}\pstree{\Tbx{5}}{}}}   \par
```

after insertion into these notes becomes:

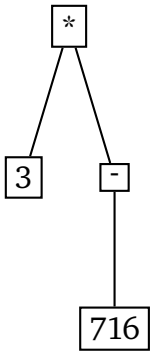
after VAL: 3

after VAL: 716 3

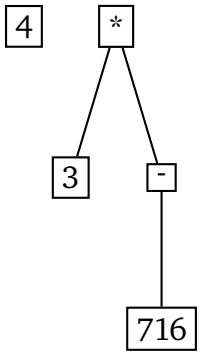
after NEG:



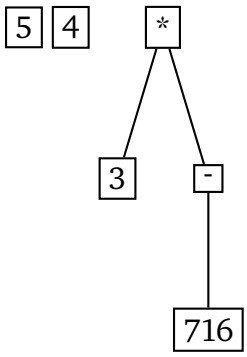
after MUL:



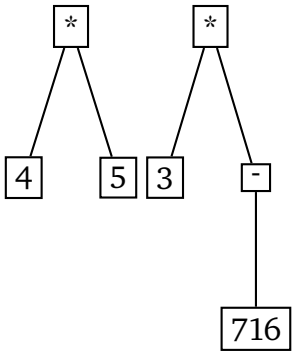
after VAL:



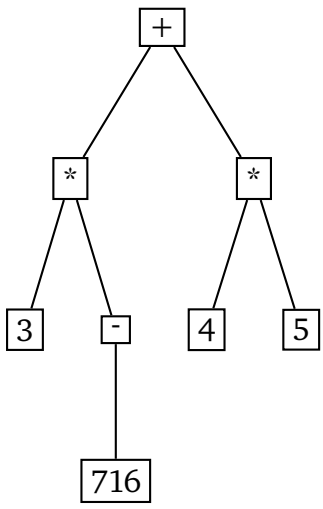
after VAL:



after MUL:



after ADD:



Practical options for the project are 4 and 5.

8.6 And yet one more rewrite: tail recursion

This rewrite steps away from CFG's and is based on convenience. Replacing tail recursion by a `while` loop simplifies the program, reduces the number of procedures, and reduces the depth of the recursion.

We demonstrate this on a simple recursive program that computes the sum of integers, 1 to N.

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

```
int sum, N;

void recurse(int N) {
  if (N>0) {
    sum=sum+N;
    recurse(N-1);
    }
  }

int main() {
  sum=0;
  N=20;
  recurse(20);

  printf("Result=%d",N);
}
```

We can replace `if` by a `while`:

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

int sum, N;

void recurse(int N) {
  while (N>0) {
    sum=sum+N;
    N=N-1;
    }
  }

int main() {
```

```
  sum=0;
  N=20;
  recurse(20);

  printf("Result=%d",N);
}
```

With our grammar, we currently have,with actions added:

```
  void E() {
    T();
    Etail();
    }

  void Etail() {
    switch(curtoken) {
      case TK_PLUS:
         match(TK_PLUS);T();  action(ADD); Etail();  break;
      case TK_MINUS:
         match(TK_MINUS);T(); action(SUB); Etail();  break;
      default:
         ;
      }
    }

....
```

we replace `Etail()` with a while loop:

```
  void Etail() {
    while(curtoken==TK_PLUS||curtoken==TK_MINUS) {
      switch(curtoken) {
      case TK_PLUS:
         match(TK_PLUS);T();  action(ADD); break;
      case TK_MINUS:
         match(TK_MINUS);T(); action(SUB); break;
      default:
```

```
        ;
      }
      }
    }

....
```

We go one step further and realize that we don't need `Etail()` anymore, it can now be merged into `E()`.

```
  void E() {
    T();
    while(curtoken==TK_PLUS||curtoken==TK_MINUS) {
      switch(curtoken) {
      case TK_PLUS:
         match(TK_PLUS);T();  action(ADD); break;
      case TK_MINUS:
         match(TK_MINUS);T(); action(SUB); break;
      default:
         ;
      }
      }
    }

....
```

We can dispose of `Ttail()` similarly.

The final rewriting step will add semantics support – this would have been difficult without the last rewrite.

Why are we doing this? We can talk about type for `E`, `T`, `F`.

$$\underbrace{\underbrace{a * b}_{T} + \underbrace{c * d}_{T}}_{E}$$

But what is the type of `E'`?

$$\underbrace{\underbrace{a * b}_{T} \underbrace{+ c * d}_{E'}}_{E}$$

8.7 TYPES

To correctly generate code for expressions (and not only) we need to introduce types and track them when parsing.

We will define most common types of TYPES now:

`TP_INT` integer data type

`TP_CHAR` character data type

`TP_BOOL` boolean data type

`TP_ENUM` enumerate data type

`TP_REAL` real (floating point) data type

`TP_ARR` array

`TP_STR` string

`TP_PTR` pointer

`TP_REC` record (struct)

`TP_SET` set

`TP_PROC` procedure

`TP_FUNC` function

`TP_FILE` file

`TP_CLASS` class (for OOP)

(This list does not imply that you have to implement all of them!); the list does not attempt to be complete.

The first four types are ordinal, ordinal types, together with the real type, are primitive types. In weak-typed language (`C`) ordinal types are equivalent, in strong typed languages (Pascal) they are different (but still equivalent to ints on the machine level).

Full type descriptors have to add additional information to the basic type. For example, for `TP_INT` this may be the size of the integer, as well as a flag to distinguish between signed and unsigned. For `TP_ARR` this would be the lower bound, the higher bound, the index type (ordinal), the element type (any). For records this would be a symbol table of defined fields.

We do not go into details right now – wait for declaration parsing.

By TYPE we will mean one of the defined types.

8.8 Supporting types in expressions

To support types, we convert all the expression functions (`E()`, `T()`, `F()`, possibly others) from `void` to returning TYPE.

Let's begin with `E()`. We have

```
void E() {
  T();
```

```
    while(curtoken==TK_PLUS||curtoken==TK_MINUS) {
      switch(curtoken) {
      case TK_PLUS:
         match(TK_PLUS);T();  action(ADD); break;
      case TK_MINUS:
         match(TK_MINUS);T(); action(SUB); break;
      default:
         ;
      }
      }
    }

....
```

We will add `OR` to the grammar to have an extra example for later; for `PASCAL` (but not `C`) it does belong here. (One can also add `XOR` here, similar to `OR` handling)

```
  void E() {
    T();
    while(curtoken==TK_PLUS||curtoken==TK_MINUS) {
      switch(curtoken) {
      case TK_PLUS:
         match(TK_PLUS);T();  action(ADD); break;
      case TK_MINUS:
         match(TK_MINUS);T(); action(SUB); break;
     case TK_OR:
         match(TK_OR);T(); action(OR); break;
      default:
         ;
      }
      }
    }

....
```

and add type passing :

```
 TYPE E() {
   TYPE t1; // type of 1st argument
   TYPE t2; // type of 2nd argument
   t1=T();
   while(curtoken==TK_PLUS
  ||curtoken==TK_MINUS
  ||curtoken==TK_OR) {
     switch(curtoken) {
     case TK_PLUS:
        match(TK_PLUS); t2=T();  t1=do_ADD(t1,t2); break;
     case TK_MINUS:
        match(TK_MINUS); t2=T(); t1=do_SUB(t1,t2);  break;
    case TK_OR:
        match(TK_OR); t2=T(); t1=do_OR(t1,t2);  break;
     default:
        ;
     }
     }
return t1;
   }

....
```

where `do_ADD()`, `do_SUB()`, .. are customized action routines to be discussed next.

8.9 Action tables for binary operations

The type of code that should be generated depends on the operation and types of the arguments. Generally, we should consider $NT \times NT$ where NT

is the number of types, but we can reduce our attention to primitive types only for now. Majority of other combinations of `t1` and `t2` are illegal for addition[2]; for example, can we add an array to a function?

We thus look at a 5 by 5 table:

Here letters i,c,b,e,r indicate the five primitive types. There is no addition on three of the primitive types so we can mark most of the entries with right away : these cases can be handled with an `error("Type mismatch");` call.

ADD	i	c	b	e	r
i	.	×	×	×	.
c	×	×	×	×	×
b	×	×	×	×	×
e	×	×	×	×	×
r	.	×	×	×	.

(In `C`, this table would be only 2×2.)

The cases that remain (only four) will be considered for the stack machine assembler case.

We thus have only 4 cases to consider:

i i implement this with the stack machine `ADD` instruction. The return type is `TP_INT`.

r r implement this with the stack machine `FADD` instruction (floating point addition). The return type is `TP_REAL`.

r i we need to convert integer argument (on top) to real; we assume that we have `CVR` instuction (convert-to-real) and emit `CVR` and `FADD`. The return type is `TP_REAL`.

i r we assume that we also have stack machine `XCHG` instruction and generate `XCHG,CVR,FADD`. The return type is `TP_REAL`.

[2]Why did not I write ALL?

Showing resulting types :

ADD	i	c	b	e	r
i	i	×	×	×	r
c	×	×	×	×	×
b	×	×	×	×	×
e	×	×	×	×	×
r	r	×	×	×	r

Exercise: *Can addition work on other types?*

Well, yes

- set addition in Pascal (union)
- string addition in Basic (concat)
- pointer arithmetic

Similar approach is used for subtraction, but since substruction is not commutative, an extra instruction will be needed for the last case.

The other difference is in pointer arithmetic.

The situation with `OR` is different: the operation is not defined for reals, it is defined for booleans and may be defined for integers. The two cases may require different assembler instructions: logical on boolean but bitwise on integers.

OR	i	c	b	e	r
i	i	×	×	×	×
c	×	×	×	×	×
b	×	×	b	×	×
e	×	×	×	×	×
r	×	×	×	×	×

8.10 Type passing from F

We will not discuss `T()` - it is very similar to already discussed |E()| – only note that

- one should pay attention to `/` and `DIV` if one wants to implement Pascal correctly.
- `AND` handling is part of `T()` (similar to `OR`).

We now look at

Grammar :

$F \Longrightarrow +F$ NOP | $-F$ NEG | **not** F NOT | (E) | LIT LIT

A blueprint :

```
TYPE F() {
  switch(curtoken) {
  case TK_PLUS: //unary plus
```

```
    // ensure that recursive call to F() returns TP_INT
 //   and then return TP_INT.
   case TK_MINUS: //unary minus
    // ensure that recursive call to F() returns TP_INT
    //  and then generate NEG and return TP_INT.
   case TK_NOT:
    // handle boolean and integer cases
   case TK_LP:
    // match both parenthesis,
 //   return the type returned by recursive call to E()
   case TK_LIT:
    // push literal on stack;
 //    return appropriate type for the literal.
   case TK_A_VAR:
   // consult the symbol table for the type,
//   push the variable, return the type from
   // symbol table record.
   // we cannot do this YET,
   //    we need to learn how to pass the declarations first.

   // More cases here
   }
```

8.11 Type passing in built-in functions

TBD

Chapter 9

Stack machine description

Generating code for a stack machine is considerably simpler than for a "real" hardware – this is because we do not have to worry about register allocaton (stack machine has no general purpose registers) and because we can chose more convenient instruction set, without too much worry about efficiency.

Below is a sample stack machine description

9.1 Memory and registers

Notice that while we use hardware terminology like "memory" and "registers", we actually discuss a software emulation of a stack machine; thus "registers" are simply variables in our implementation and memory areas are simply arrays.

CODE We consider code to be a byte[1] array. This array is always preinitialized by the compiler.

[1] C:`unsigned char`

DATA We consider data to be a byte array. This array may be preinitialized by the compiler, compiler is always responsible for determining the size of the data array.

STACK We consider stack to be a byte array. This array is not preinitialized by the compiler.

HEAP We consider heap to be a byte array. This array is not preinitialized by the compiler. (We will not discuss HEAP in this course, but note that HEAP can be implemented within DATA or STACK.)

All of these array may contain other type of data (`int`'s, for example).

All of there array should be large enought to allow running sample programs.

While Stack machine does not general purpose (arithmetic) registers, several control registers exist.

IP – Instruction pointer; indicates current code position. During compilation, the same variable will be used to point to the first unfilled byte in the **CODE** array.

DP – Data pointer; is not used during execution, only for allocation during compilation.

SP – Stack pointer; only used during execution.

BP – Base pointer; only used during execution (TBD later).

Overall format: we assume that the opcode part of the instruction takes exactly one byte and may be followed by additional argument bytes. This implies that we cannot have more than 256 opcodes.

We further assume that we have defined symbolic names for the opcodes, for example

```
#define OP_PUSH 1
#define OP_POP 2
....
#define OP_ADD 30
#define OP_SUB 31
....
```

We will omit the `OP_` prefix when referring to the opcodes in descriptions.

We further assume that our machine uses 32-bit addresses (changing to 16-bit or 64-bit is trivial).

In the simple model that follows we also assume that both integer and real values are always 32 bit (corresponds to `int`, `float`).

9.2 The runtime loop

```
void run() {

IP = 0; // or start address
SP = 0; // stack is empty initially

// read code array if needed

while (true) {
  switch(code[IP++]) {
  case OP_PUSH:
     ....
     break;
  case OP_POP:
     ....
     break;
```

```
  case OP_ADD:
     ....
     break;
  default: error("Runtime error: illegal instruction.")
  }
}

}
```

9.3 Instructions

9.3.1 Data Movement

`PUSH` Push data on the stack, argument is a 32-bit integer denoting memory address of a variable, thus this instruction takes 5 bytes total.

`POP` Push data on the stack, argument is a 32-bit integer denoting memory address of a variable, thus this instruction takes 5 bytes total.

`PUSHI` Push value on the stack, argument is a 32-bit numeric quanity that follows the opcode, thus this instruction takes 5 bytes total.

`DUP` Duplicate the top of the stack value.

`XCHG` Exchange the two top stack entries.

`PUT` TBD

`GET` TBD

9.3.2 Arithmetic

Instuction in this group do not have arguments.

BINARY	`ADD`	pop two integer values from the stack, add them, push the result back on stack.
	`SUB`	pop two integer values from the stack, subtract them, push the result back on stack.
	`MUL`	multiplication
	`DIV`	division
	`OR`	or
	`AND`	and
	`EQL`	pop two integer values from the stack, compare them, push the result (0 or 1, 1 if equal) back on stack.
	`NEQ`	pop two integer values from the stack, compare them, push the result (0 or 1, 1 if not equal) back on stack.
	`GTR`	greater
	`LSS`	less
	`GEQ`	greater or equal
	`LEQ`	less or equal
		...
	`FADD`	pop two real values from the stack, add them, push the result back on stack.
	`FSUB`	pop two real values from the stack, subtract them, push the result back on stack.
		...
UNARY	`NEG`	– negate top element of stack, seen as an integer.
	`NOT`	– logical not (curveball: what about bitwise?)
	`FNEG`	– negate top element of stack, seen as a real.
	`CVR`	– convert to real
	`CVI`	– convert to integer (truncate)

This is not a complete list, and additional operations may be added to implement builtin functions. How about `ABS`, `FABS`, `FSIN`, for example?

9.3.3 Control

`HALT` stop execution, no arguments.

`JMP` argument is a 32-bit address (in code)

`JFALSE` argument is a 32-bit address (in code)

`JTRUE` argument is a 32-bit address (in code)

`JTAB` TBD.

`CALL` argument is a 32-bit address (in code), TBD later.

`RETURN` TBD later.

9.3.4 I/O and system

We cheat and make I/O (normally calls to library) into primitive assembler instructions.

Only one some examples for now:

`PRINTINT` print (popped) value from stack as an integer.

`PRINTCHR` print (popped) value from stack as a character.

`PRINTLN` print newline character.

`MALLOC` read (int) argument from stack, allocate that many bytes, put the received address back on stack.

`FREE` read (address) argument from stack, free that memory.

9.4 Compiling a small program

(an ultra-small sample to disect)

```
var x,y,z: integer;

begin
x:=y+z;
end.
```

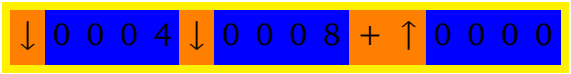
The declaration parser (TBD!) will allocate variables in the data array by assigning them addresses (0, 4, and 8 under our assumptions). These addresses will be stored in the symbol table and placed (instead of variable names!) into the generated code.

The compiler will also calculate that the data array should be at least 12 bytes long.

In symbolic form, the generated code should be

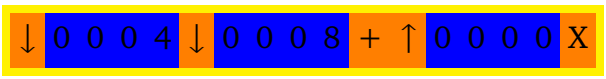
```
; x=y+z
PUSH 4
PUSH 8
ADD
POP 0
```

In machine language, this program will occupy 16 bytes.



Orange bytes are the opcodes, blue bytes are the arguments.

One important missing detail: the compiler must always generate a halt instruction at the end of the program, so our *executable* will actually need 17 bytes:



The output of the compiler can be written to a binary file or passed directly to the runtime execution (`run()`).

9.5 Implementation of some cases

We provide sample implementations of some instructions. Notice that we are not doing any checking : the addresses may be bad, or the stack may overflow.

The most trivial:

```
case OP_HALT:
   exit(0);
```

```
case OP_NEG:
   {
   *(int*)(STACK+SP-sizeof(int)) =
     - *(int*)(STACK+SP-sizeof(int));
   }
   break;
```

```
case OP_ADD:
   {
   int temp;
   SP-=sizeof(int); temp=*(int*)(STACK+SP);
   *(int*)(STACK+SP-sizeof(int)) +=temp;
   }
   break;
```

```
case OP_PUSH:
   {
   int temp;
   temp=*(int *)(CODE+IP); IP+=sizeof(int);  // address
   temp=*(int *)(DATA+temp);    // value of the variable;
   *(int *)(STACK+SP)=temp; SP+=sizeof(int); // push
   }
   break;
```

```
case OP_FSIN:
   { // blatant theft here :P

   *(float *)(STACK+SP-sizeof(float)) =
     sin ( *(float *)(STACK+SP-sizeof(float)));
   }
   break;
```

Chapter 10

Pascal Grammar

BNF for Pascal (original)

```
<program> ::= program <identifier> ; <block> .

<identifier> ::= <letter > {<letter or digit>}

<letter or digit> ::= <letter> | <digit>

<block> ::= <label declaration part> <constant definition part>
   <type definition part> <variable declaration part>

<procedure and function declaration part> <statement part>

<label declaration part> ::=
   <empty> | label <label> {, <label>} ;

<label> ::= <unsigned integer>

<constant definition part> ::= <empty>
  | const <constant definition>
     { ; <constant definition>} ;
```

```
<constant definition> ::= <identifier> = <constant>

<constant> ::= <unsigned number>
  | <sign> <unsigned number>
  | <constant identifier>
  | <sign> <constant identifier>
  | <string>

<unsigned number> ::= <unsigned integer> | <unsigned real>

<unsigned integer> ::= <digit> {<digit>}

<unsigned real> ::= <unsigned integer> . <unsigned integer>
 | <unsigned integer> . <unsigned integer> E <scale factor>|

<unsigned integer> E <scale factor>

<scale factor> ::= <unsigned integer>
  | <sign> <unsigned integer>

<sign> ::= + | -

<constant identifier> ::= <identifier>

<string> ::= '<character> {<character>}'

<type definition part> ::= <empty>
 | type <type definition> {;<type definition>};

<type definition> ::= <identifier> = <type>

<type> ::= <simple type>
 | <structured type>
 | <pointer type>

<simple type> ::= <scalar type>
 | <subrange type>
```

```
 | <type identifier>

<scalar type> ::= (<identifier> {,<identifier>})

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<structured type> ::= <array type>
 | <record type>
 | <set type>
 | <file type>

<array type> ::= array [<index type>{,<index type>}]
  of <component type>

<index type> ::= <simple type>

<component type> ::= <type>

<record type> ::= record <field list> end

<field list> ::= <fixed part>
  | <fixed part> ; <variant part>
  | <variant part>

<fixed part> ::= <record section> {;<record section>}

<record section> ::= <field identifier>
  {, <field identifier>} : <type> | <empty>

<variant type> ::= case <tag field>
  <type identifier> of <variant> { ; <variant>}

<tag field> ::= <field identifier> : | <empty>

<variant> ::= <case label list> : ( <field list> ) | <empty>
```

```
<case label list> ::= <case label> {, <case label>}

<case label> ::= <constant>

<set type> ::=set of <base type>

<base type> ::= <simple type>

<file type> ::= file of <type>

<pointer type> ::= <type identifier>

<variable declaration part> ::= <empty>
  | var <variable declaration> {; <variable declaration>} ;

<variable declaration> ::=
  <identifier> {,<identifier>} : <type>

<procedure and function declaration part> ::=
  {<procedure or function declaration > ;}

<procedure or function declaration > ::=
  <procedure declaration > | <function declaration >

<procedure declaration> ::= <procedure heading> <block>

<procedure heading> ::= procedure <identifier> ; |

procedure <identifier> ( <formal parameter section>
  {;<formal parameter section>} );

<formal parameter section> ::=
  <parameter group> | var <parameter group> |

function <parameter group>
 | procedure <identifier> { , <identifier>}

<parameter group> ::= <identifier>
```

```
  {, <identifier>} : <type identifier>

<function declaration> ::= <function heading> <block>

<function heading> ::=
  function <identifier> : <result type> ; |

function <identifier> ( <formal parameter section>
  {;<formal parameter section>} ) : <result type> ;

<result type> ::= <type identifier>

<statement part> ::= <compund statement>

<statement> ::= <unlabelled statement>
  | <label> : <unlabelled statement>

<unlabelled statement> ::=
  <simple statement> | <structured statement>

<simple statement> ::= <assignment statement>
  | <procedure statement>
  | <go to statement> | <empty statement>

<assignment statement> ::= <variable> := <expression>
  | <function identifier> := <expression>

<variable> ::= <entire variable>
  | <component variable>
  | <referenced variable>

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable> ::= <indexed variable>
  | <field designator> | <file buffer>
```

```
<indexed variable> ::=
  <array variable> [<expression> {, <expression>}]

<array variable> ::= <variable>

<field designator> ::= <record variable> . <field identifier>

<record variable> ::= <variable>

<field identifier> ::= <identifier>

<file buffer> ::= <file variable>

<file variable> ::= <variable>

<referenced variable> ::= <pointer variable>

<pointer variable> ::= <variable>

<expression> ::= <simple expression>
 | <simple expression> <relational op> <simple expression>

<relational op> ::= = | <> | < | <= | >= | > | in

<simple expression> ::= <term> | <sign> <term>
  | <simple expression> <adding operator> <term>

<adding operator> ::= + | - | or

<term> ::= <factor> | <term> <multiplying operator> <factor>

<multiplying operator> ::= * | / | div | mod | and

<factor> ::= <variable>
  | <unsigned constant>
  | ( <expression> )
  | <function designator>
  | <set>
```

```
  | not <factor>

<unsigned constant> ::= <unsigned number> | <string>
  | < constant identifier> < nil>

<function designator> ::= <function identifier>

  | <function identifier

     (<actual parameter> {, <actual parameter>})

<function identifier> ::= <identifier>

<set> ::= [ <element list> ]

<element list> ::= <element> {, <element> } | <empty>

<element> ::= <expression> | <expression> .. <expression>

<procedure statement> ::= <procedure identifier>
  |  <procedure identifier>
     (<actual parameter> {, <actual parameter> })

<procedure identifier> ::= <identifier>

<actual parameter> ::= <expression> | <variable>
  | <procedure identifier> | <function identifier>

<go to statement> ::= goto <label>

<empty statement> ::= <empty>

<empty> ::=

<structured statement> ::= <compound statement>
  | <conditional statement>
  | <repetitive statement>
  | <with statement>
```

```
<compound statement> ::=

   begin <statement> {; <statement> } end;

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= if <expression> then <statement>
   | if <expression> then <statement> else <statement>

<case statement> ::= case <expression> of
   <case list element> {; <case list element> } end

<case list element> ::=

  <case label list> : <statement> | <empty>

<case label list> ::= <case label> {, <case label> }

<repetitive statement> ::=

  <while statement> | <repeat statemant> | <for statement>

<while statement> ::= while <expression> do <statement>

<repeat statement> ::= repeat <statement>

   {; <statement>} until <expression>

<for statement> ::=
   for <control variable> := <for list> do <statement>

<control variable> ::= <identifier>

<for list> ::= <initial value> to <final value> |
    <initial value> downto <final value>

<initial value> ::= <expression>
```

```
<final value> ::= <expression>

<with statement> ::= with <record varlist> do <statement>

<record varlist> ::= <record variable> {, <record variable>}
```

This is the original grammar – not quite the way the language ended up! – but close.

Rather (only declarations for now)

Grammar :

<Pascal program> ⇒ <header> <declarations><begin statement>

<header> ⇒ <program statement> | <unit statement> | ϵ

<declarations> ⇒ <declaration> <declarations> | ϵ

<declaration> ⇒ <label declaration> ;

| <var declaration> ;

| <const declaration> ;

| < type declaration> ;

| <procedure declaration> ;

| <function declaration> ;

| ;

10.1 Declarations parsing

The header can be simply omitted.

The overall compiler now amounts to

```
void compile() {
header();
declarations();
begin_statement();
match(TK_DOT);
emit_opcode(OP_HALT);
```

Notice that ALL declarations begin with keywords, this suggest implementing the entire declaration parser as a single switch.

```
void declaration() {

restart:

    switch (curtoken) {
    case TK_LABEL:
```

```
      label_declaration(); goto restart;
    case TK_VAR:
      var_declaration(); goto restart;
    case TK_CONST:
      const_declaration(); goto restart;

    ....
    default:
      return;
    }

}
```

At this type we mostly care about declaring variables.

10.2 Variable declarations parsing

Sample declaration:

```
  var x,y: integer;
    r1,r2,r3: real;
```

Before going any further : what kind of information should we place into the symbol table?

name – key for searching

token type – `TK_A_VAR`

static – flag (for now, assume YES)

type – TYPE; may include a pointer to detailed type descriptor or additional information (range)

size – optional (should be possible to derive from type)

address – computed during parsing

flags – for example, read only

initial value – if this is at all supported

We notice that we cannot enter information into symbol table until we have the type (this is one place where C is easier.. a bit).

```
void var_declaration() {
  match(TK_VAR);  // never fails

restart:

  if (curtoken!=TK_UNKNOWN)
     error();

  // inialize a temporary token list

  // parse alternating sequence of TK_UNKNOWN and TK_COMMA

  // ensure that each TK_UNKNOWN does
  //   not appear in the top symbol table
  // or in the list we are building

  // append the TK_UNKNOWN token to the token list
```

```
if (curtoken!=TK_COLON) error();

// read type declaration (simplify!)

// go through the list and enter
//   all variables into the symbol table

// free temporary token list

// match(TK_SEMICOLON);

if (curtoken==TK_UNKNOWN) goto restart;

}
```

Why double check? Because we have to worry about both

```
var x,y,z: integer;
var a,b,x: real;
```

and

```
var x,y,a,b,x: integer;
```

How are the addresses computed?

- compute the size (based on type)
- `DP = align(DP,size);` //optional

- `addr = DP;`
- `DP = DP+ size;`

What is "`integer`" exactly ?

Well... it is not a keyword (cf. `int` in C), none of the types are keywords.

It is a predefined (in SYSTEM) type... thus

Thus,

```
type integer = real;
```

is legit (just odd), and so is

```
var integer : boolean;
```

and so is

```
label integer;
```

(SYSTEM is seached last, so this makes it impossible to reach the original definition).

How is it predefines exactly? As range, on a 16-bit system, for example:

```
type integer = -32768..32767;
```

one can declare additional integer types like

```
  type byte = 0..255;
  type shortint = -128..277;
  type word = 0..65535;
```

Notice that these declarations must be somehow read before the compilation starts!

10.3 Statement

Let's look at

```
<compound statement> ::=
  begin <statement> {; <statement> } end;
```

now. We observe that ALL statements are recognizable from `curtoken`, therefore we envision a switch similar to the declarations.

(I will use `begin_statement` instead of `compount statement` of the original.)

```
4
```

```
void statement() {

restart:
```

```
    switch (curtoken) {
    case TK_IF:
      if_statement(); goto restart;
```

```
    case TK_WHILE:
      while_statement(); goto restart;
    case TK_GOTO:
      goto_statement(); goto restart;

    ....
    case TK_A_VAR:
      assignment(); goto restart;
    case TK_WRITE: // notice: TK_WRITE is not a keyword!
      write_statement(); goto restart();
    default:
      return;
    }

}
```

Of interest to us now are the last two, supporting them together with variable declarations would create a working compiler!

How can the `TK_A_VAR` token show up? It is surely not returned by the scanner.

Basically, we need to look it up, and we need to do this with every identifier encountered outside declarations. Roughly, like this:

```
    if (curtoken==TK_UNKNOWN)
      find_token();  // find token in symbol tables
        // and replace TK_UNKNOWN by TK_A_....
```

10.4 Assignment

so... let's handle assignments, here is a simplified version

```
 void assigment() {
```

```
// save information about the current symbol
// including type and address
  TYPE tLHS = <type from symbol table>
  gettoken();
  match(TK_ASSIGN); // := needed
// TYPE tRHS = E();

// generate error if needed
// generate conversion code if needed

// emit OP_POP (with address of LHS)
}
```

And to be able to see the output, we need

10.5 Write Statement

We notice that `TK_WRITE` is not a keyword, again it should be declared in SYSTEM.

For now, we ignore this issue and assume it to be a keyword – the difference only amounts to being able to reuse the symbol.

It is actually correct to call it a statement, even if Pascal literature calls it a built-in "procedure". Like several other built-in procedures and functions, functions like `write` cannot be implemented in Pascal! – they do not even follow language syntax.

Specifally, `write` (as well as `read`,`writeln`, `readln`) have three such violations:

- variable number of parameters
- arbitrary types of parameters

- formatting specs, cf. `write(x:10)` – colon is not a part of syntax in any legitimate procedure!

We also simplify the syntax for now to support only a single argument. Thus

```
var x:integer;
  begin
  x:=10;
  write(x);
  end.
```

would be ok, while (actually, correct!) program

```
var x:integer;
  begin
  x:=10;
  write("x=",x);
  end.
```

is not.

Our simplified `write` now looks like this:

```
 void write_statement() {
   match(TK_WRITE);
   match(TK_LP);
   TYPE t=E();
   match(TK_RP);
   if (t==TP_INT)
     // Emit OP_PRINTINT code.
```

```
else if (t==TP_CHAR)
  // Emit OP_PRINTCHAR code
else
   ....
else error("Unsupported type");
}
```

A better implementation would support different argument types and formatting parameters.

10.6 Relative or absolute jumps?

There are two ways to implement jumps: absolute and relative.

With absolute jumps, the argument of `JUMP`,`JFALSE`, `JTRUE` is a an absolute address that is assigned to IP.

With relative jumps, the argument of `JUMP`,`JFALSE`, `JTRUE` is a signed delta that is added to IP.

The latter approach is a bit more difficult but (1) parallels most of the hardware (2) has several advantages.

10.7 Repeat Statement

We being with it since it is the easiest control statement.

Pascal's `repeat..until` corresponds to C's `do..while`, albeit with the condition reversed.

Example:

```
var x:integer;
begin
x:=1;
repeat write(x); x:=x+1;
until x=10;
end.
```

To support any conditions the expression grammar needs to be extended to include comparison operators (=, <>, >,...) – this is left to the reader!

How to do this? (Hint)

Look at the grammar, and notice that what we called `E` in our grammar corresponds to `simple expression` in the "official" grammar. The complete expression definition is

```
<expression> ::= <simple expression>
| <simple expression> <relational operator> <simple expression>
```

This needs to be added to the grammar (one more level!).

Notice that this comparison operators cannot be used twice on the same level, thus `x<y<z` or `x=y=z`[1] comparison, not assignment!) are illegal.

We can also define a generic `condition()` parser component to be used in `repeat`, `while` and `if`.

```
void condition() {
  if (E() !=  TP_BOOL )
    error();
  }
```

The grammar part corresponding to `repeat` amounts to

[1] (

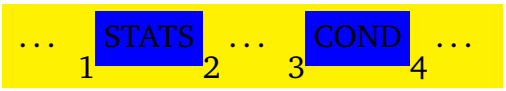
Grammar :

<repeat statement> ⇒ **repeat** <statements> **until** <condition>

Parsing is trivial:

```
void repeat_statement() {
  match(TK_REPEAT);  //never fails
  statements();      //generates some balanced code
  match(TK_UNTIL);   //may fail
  condition();       //generates some unbalanced code
  }
```

We need to generate some connecting code now, to be appended to already emitted:



Here in blue we show already generated copy, we can append more code in the yellow parts now.

A working loop would be accomplished if we simply append a `JFALSE` from point 4 to point 1.

Assuming jumps are absolute, here is how:

```
void repeat_statement() {
  match(TK_REPEAT);  //never fails
  int target=IP;     //position of point 1
  statements();      //generates some balanced code
```

```
match(TK_UNTIL);   //may fail
condition();       //generates some unbalanced code
emit_opcode(OP_JFALSE);  //append 1 byte to code
emit_int(target);      //append address to code
}
```

The last line becomes

```
emit_int(target-IP);
```

if we prefer relative jumps.

NOTE: this routine will be revisited later!

10.8 While Statement

Perhaps counterintuitively, `while` is a bit more complicated than `repeat`, and is more interesting as well.

Grammar :

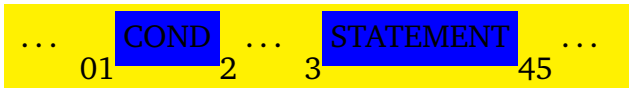
<while statement> ::= while <expression> do <statement>

Parsing is trivial:

```
void while_statement() {
  match(TK_WHILE);  //never fails
  condition();      //generates some unbalanced code
```

```
match(TK_DO);      //may fail
statement();       //generates some balanced code
}
```

The template to fill here looks like this:



and one solution would be to append JFALSE (loop exit) from point 2 to point 5, as well as unconditional JMP from point 4 to point 1 (loop continuation).

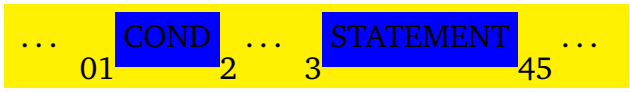
We thus will need to add two jumps, one – backward JMP that can be done immediately, and one – forward JFALSE – that would require patching the code since at the time we generate the instruction we don't yet know the address of the target.

```
void while_statement() {
  match(TK_WHILE);  //never fails
  int target=IP;    //for JMP back
  condition();      //generates some unbalanced code
  emit_opcode(OP_JFALSE);
  int hole=IP;
  emit_int(0);
  match(TK_DO);     //may fail
  statement();      //generates some balanced code
  emit_opcode(OP_JMP);
  emit_int(target)
  int save_IP=IP;
  IP=hole;
  emit_int(save_IP); //patch the code-generation
  IP=save_IP;
  }
```

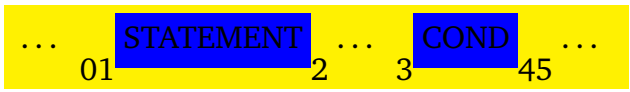
This implementation will work correctly but will have overhead of two jumps per loop iteration vs one we saw in the repeat statement.

Can anything be done about this?

Assuming we can simply swap the bytes in `COND` and `STATEMENT` parts, replacing



with



we can now compose jumps differently: first, an unconditional jump `JMP` from point 0 to point 3, plus a `JTRUE` from point 4 to point 1.

The overhead now will be only one jump per iteration since `JMP` will be executed only once.

But can this be really done?

10.9 If Statement

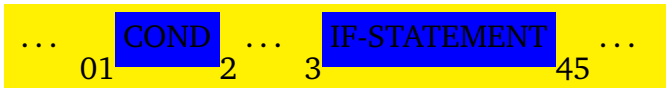
`if` statement presents a different problem: it has two forms!

Grammar :

<if statement> ::= if <expression> then <statement>

| if <expression> then <statement> else <statement>

The templates for it are



What kinds of jumps are needed in each case?

In the compiler implementations both cases can be handled by one function that would begin by generating common code and check for the presence of `else` after the return from the `statement();` call.

Notice that this binds `else` with the nearest `if`, as it should.

10.10 For Statement

Is left entirely to the reader

10.11 Goto Statement

10.11.1 Do we need goto's?

(Can we get by with `break`, `continue` ?)

Two answers: firstly, remember Modula-2 story.

secondly, there are – often simple – programs that beg for having a goto. One typical case is breaking out of a double loop.

Sample problem: find an integer solution of equation $x^2 + y^2 == N$.

This can be only down by brute force

```
  int x,y;
  for (x=0;x<=N;x++)
    for (y=0;y<=N;y++)
      if (x*x+y*y==N)
        goto found;
not_found:
          ....
found:
```

`break` is useless, and use of flags will lead to slower code.

Incidentally, the code above can be sped up, and in more than one way, but double loop is not avoidable.

10.11.2 Difficulties

The difficulties with the `goto` include:

- `goto` processing occurs in multiple places – unlike other statements.
- some `goto`'s are illegal.
- some `goto`'s require stack adjustment (and this applies not to just stack machine implementations).
- `goto's` are optimization-unfriendly.

The processing of `goto` occurs in these places:

- in declarations (`label` in Pascal, not present in `C`).

- `goto` statement itself.
- label processing (also, within statements).
- at the end of a procedure, function, or program block.
- upon entry or exit of any block (to discuss later)

In the first part of the discussion we will ignore the issues of legality and stack.

This sample code shows where the processing occurs:

```
  program test;

  label lab;  (*1*)

  begin

  goto lab;   (*2*)

  ...

lab: (*3*)

  ...
   goto lab; (*2*)
   ...

   end(*4*) .
```

10.11.3 label declaration

In Pascal `label` (and ALL) declarations are mandatory.

`label` declarations are part of the general declaration machinery, already seen:

```
void declaration() {

restart:

    switch (curtoken) {
    case TK_LABEL:
      label_declaration(); goto restart;
    case TK_VAR:
      var_declaration(); goto restart;
    case TK_CONST:
      const_declaration(); goto restart;

    ....
    default:
      return;
    }

}
```

Sample label declaration is

```
  label lab1,lab2,restart;
```

(In the original Pascal definition labels were integer constants, treated as idenfiers. Such form of labels remains part of the standard and should be supported by any implementation.)

Parsing a list of identifiers is similar – and simpler – than in the case of variables.

Upon parsing we should enter all the labels into the symbol table, we mark them with the `TK_A_LABEL` token type.

Checks for possible double declarations should be performed, just like with variables.

Exercise: *What fields should be provided in the symbol table*

- `addr` == the address of the label (to be learned later).
- `seen`: (flag) has the label been seen? (that is, do we know the address?)
- `gotolist` of `goto's` that target this label (with hole, source and scope information)
- scope information.
- (optional) `nextlabel` == link to the next label record in the symbol table

We initialize `seen=false; gotolist=NULL;`.

10.11.4 label processing

This and next sections may appear in any order!

Relavant part of the original grammar.

Grammar :

<statement> ::= <unlabelled statement> | <label> : <unlabelled statement>

(generally changed to

Grammar :

<statement> ::= <unlabelled statement> | <label> : <statement>

---

to allow multiple labels with the same statement).

We can include it into the main statements switch

```
void statement() {

restart:



    switch (curtoken) {
    case TK_IF:
      if_statement(); goto restart;
    case TK_WHILE:
      while_statement(); goto restart;
    case TK_GOTO:
      goto_statement(); goto restart;

    ....
    case TK_A_VAR:
      assignment(); goto restart;
 case TK_A_LABEL:
   process_a_label(); goto restart;
    case TK_WRITE: // notice: TK_WRITE is not a keyword!
      write_statement(); goto restart();
    default:
      return;
    }

}
```

The processing logic is simply

```
void process_a_label() {
  if (seen) error();  else seen=true;
  addr=IP;
  // save scope information
  }
```

We do not attempt to patch goto list – no point, there may be more coming.

C difference: C does not have label declarations, thus label processing is invoked by either seening TK_A_LABEL or a TK_UNKNOWN, followed by a colon; in the latter case we enter label into the symbol table – as TK_A_LABEL.

10.11.5 goto processing

This is again a part of the statements switch.

```
  void goto_statement() {
  match(TK_GOTO);
  if (curtoken != TK_A_LABEL)
    error();
  // generate dummy jump
  // append location of a dummy jump,
  // source ref,  and
  // scope information to the gotolist
  // in the symbol table.
  gettoken();  // to continue;
  }
```

C difference: token that follows TK_GOTO may be TK_UNKNOWN; if so, we "declare" it as TK_A_LABEL.

10.11.6 end of procedure/function processing

At the end every function or procedure we need to resolve all the pending `goto's`.

```
void resolve_gotos() {
for (all labels in local symbol table) {
   if (gotolist!=NULL %% !seen)
  error();
for (all entries in gotolist)
  patch;

   }

}
```

To efficiently implement "`for all labels`" one can run a link list through a symbol table, connecting all label records.

10.11.7 Jumps inside a block

Illegal under Pascal, legal but *weird* in C.

By block one understands `begin...end`, as well as the `repeat..until` and `case..end`.

Jumps into blocks require collection scope information.

The following scheme shows a block structure of a sample procedure:

0						
1	◁					
2		◁				
3			◁			
4				◁		
5						
6				▷		
7			▷			
8						
9		▷				
10						
11		◁				
12			◁			
13						
14			▷			
15		▷				
16	▷					
17						

Numbers in the left column are for references (`13.5` refers to the point between `13` and `14`); ◁ and ▷ indicate opening and closing of a scope. We notice that `goto`'s from `5` to `10` or from `11.5` to `14.5` are legal; `goto`'s from `1` to `5`, or from `10` to `13` are not.

Part of the scope information is the `level`; we initialize it to 0 at the beginning of a procedure (function, main); we increment it upon entering a block and decrement upon exiting. In the above example, `level(1)=0`, `level(5)=3)`.

The necessary condition for `goto` from `S`(ource) to `T`(arget) is `level(S)>=(level(T)`.

The condition, however, is not sufficient: `goto` from `5` to `13` is illegal, even if the label decreases.

One algorithm that detects all bad goto's uses a string representation for different areas in code, each is assigned an `lid` (level id) string, constructed as follows:

We initialize sufficiently long[2] string `LID` to `0000000...`

On entering a block, `level++];`; on exiting a block we `lid[level--]++;`

Each location in code is assigned a (head) substring of `LID` of length `level`.

0	0	0	0	0	0	0
1	0 ◁	0	0	0	0	0
2	0	0 ◁	0	0	0	0
3	0	0	0 ◁	0	0	0
4	0	0	0	0◁	0	0
5	0	0	0	0	0	0
6	0	0	0	0▷	0	0
7	0	0	0 ▷	1	0	0
8	0	0	1	1	0	0
9	0	0 ▷	1	1	0	0
10	0	2	1	1	0	0
11	0	2 ◁	1	1	0	0
12	0	2	1 ◁	1	0	0
13	0	2	1	1	0	0
14	0	2	1 ▷	1	0	0
15	0	2 ▷	2	1	0	0
16	0 ▷	3	2	1	0	0
17	1	3	2	1	0	0

Here `lid(0)=""`, `lid(5) = "0000"`, `lid(13) = "021"`, `lid(17)=""`.

The necessary and sufficient condition for `goto` legality is `lid(T)` being a head of `lid(S)`.

[2] longer than maximum level of block nesting

10.12 The great jump into bug

Wirth did not correctly define a block, resulting in bizarre goto's that are allowed:

```
var i:integer;

label L1,L2;

begin

for i:=1 to 10 do goto L1;

for i:=10 downto 1 do L1:write(i);

writeln('now crash');

for i:=1 to 10 do goto L2;

for i:=10 to 1 do L2:write(i);

end.
```

Similarly, one can jump into a `while`, `if`, or `else`, but not into `case` or `repeat`.

10.13 The numeric label bug

Consider:

```
label 10;
```

```
begin
  goto 010;

$A:
  writeln('Hello,World!');

end.
```

10.14 non-local goto's

Pascal allows goto's to jump between procedures.

for example:

```
label leave;


  procedure error;
  begin
    writeln("error");
    goto leave;
  end;

  begin


  leave:
  end.
```

(This made sense, especially because the original Pascal did not include a `halt` (`exit`).

Consider also our compiler : `compile()` may be called from an IDE, but what would an `exit()` inside `error()` do?

This type of jump is supported in `C` albeit the ideology is different! (Pascal: only up/out, C: if you get away with it).

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

static jmp_buf buf;

void second() {
    printf("second\n");         // prints
    longjmp(buf,1);             // jumps back to where setjmp was called - making
}

void first() {
    second();
    printf("first\n");          // does not print
}

int main() {
    if (!setjmp(buf))
        first();                // when executed, setjmp returned 0
    else                        // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints

    return 0;
}
```

Stolen from Wikipedia

See also:

Practical use of setjmp/longjmp

Technical issues:

non-local goto's require stack adjustment, both `SP` and `BP` must be restored to the appropriate values.

Implication: while `label`'s normally do not generate code, a `label` used as a non-local goto target, must.

(C's `setjmp` saves this registers to a global variable, while `longjump` restores them.)

10.15 Case Statement

From the original grammar:

```
<conditional statement> ::= <if statement> | <case statement>

<case statement> ::= case <expression> of
   <case list element> {; <case list element> } end

<case list element> ::=

  <case label list> : <statement> | <empty>

<case label list> ::= <case label> {, <case label> }
```

Sample `case` statement:

```
 var c:char;
```

```
begin
c:=getchar;

case c of
'0'..'9': write('DIGIT'); (*2*)
'a'..'z','A'..Z': write('LETTER');
'+','-','*','/': write('OPERATOR');
',',';','.': write('PUNCT'); (*3*)
'''': write('QUOTE');     (*1*)
end;
```

Things to notice:

- unneeded clause `<conditional statement>`
- more general syntax than `C` and much more convenient, compare `'A'..'Z'` with C's

  ```
  case 'A':case 'B':.. case 'Z':
  ```

- absence of default (`else` in TP, `otherwise` in MetaWare).
- absence of `break` – rarely needed.

`case` can be implemented as a repeated `if` – in which case it will be inefficient – or better. For now we will consider an `if`-style implementation.

`case_statement()` discussed below is a part of a big statement switch.

```
void case_statement() {
  match(TK_CASE);
```

```
TYPE tE=E();
match(TK_OF);
// process cases
match(TK_END);
// finish code writing.
}
```

tE specifies the type of the constants that may appear among case labels (to avoid confusion with labels used by goto, we will call them case tags.

The allowed labels in the process cases loop are the constants of this type, end (terminates case), and usually else or otherwise.

We also note that the call to E() would generate code that leaves the result on the stack, we need to use this value repeatedly but eventually pop it off – this implies need to duplicate this value repeatedly (thus DUP in the stack machine architecture) and eventually remove it. The latter part can be done with POP junk (pop into a junk variable) or with a special REMOVE instruction).

Calls to statement() within case processing will generate balanced code.

We will consider the code to be generated in a series of examples.

10.15.1 Tag is a single constant

This is the simplest case, shown in the example above:

```
'''': write('QUOTE');     (*1*)
```

Assume that val1 below contains the value of the tag constant (in our example, '\'').

The following stack machine code needs to be generated:

```
        DUP
        PUSHI val1
        EQL
        JFALSE <next case>
        REMOVE
        <statement>
        JMP <end of case>

; <next case>
```

Both jumps are forward and need to be patched; `<next case>` should be the address of the next subcase (its `DUP` instruction). If there are no further subcases, `<next case>` points to the "default" case (if such is supported and present) or out of the `case`.

```
    '0'..'9': write('DIGIT'); (*2*)
```

10.15.2 Tag is a range

In our example:

```
    '0'..'9': write('DIGIT'); (*2*)
```

Assume that `val1`, `val2` below contain the values of the tag constants (in our example, `'0'` and `'1'`).

The following stack machine code needs to be generated:

```
        DUP
```

```
        PUSHI val1
        LSS
        JTRUE <next case>
        DUP
        PUSHI val2
        GTR
        JTRUE <next case>
        REMOVE
        <statement>
        JMP <end of case>

; <next case>
```

10.15.3 Tag is a list

In our example:

```
 '+','-','*','/': write('OPERATOR');
 ',',';','.': write('PUNCT'); (*3*)
```

Assume that `val1`, `val2`,... below contain the values of the tag constants (in our example, `'+'`, `'-'`, `'*'`, `'/'`).

The following stack machine code needs to be generated:

```
        DUP
        PUSHI val1
        EQL
        JTRUE <this case>
        DUP
        PUSHI val2
        EQL
        JTRUE <this case>
```

```
        DUP
        PUSHI val3
        EQL
        JTRUE <this case>
        ....
        JMP <next case>
; <this case:>
        REMOVE
        <statement>
        JMP <end of case>

; <next case>
```

10.15.4 Tag is a list of ranges

This case is left to the reader.

Exercise: *Does the code we generated looks familiar?*

Exercise: *Why cannot we wait with `REMOVE`?*

10.15.5 Better case implementation

10.16 With Statement

Nice stuff, but does not fit into the schedule. :(

10.17 Arrays

10.17.1 Declarations

Implementation of arrays begins with declarations. For simplicity we will not consider `type` declaration and limit ourselves to explicit `array` in the variable declaration. Thus

```
var a,b:array [1..10] of char;
var c,d:array ['a'..'z'] of integer;
```

are to be covered while

```
type t=array[1..10] of char;

var a,b:t;
```

will not be. Further, we will limit ourselves to one-dimensional arrays.

Relevant part of the grammar is

```
<structured type> ::= <array type>
 | <record type>
 | <set type>
 | <file type>

<array type> ::= array [<index type>{,<index type>}]
  of <component type>

<index type> ::= <simple type>

<component type> ::= <type>
```

```
<simple type> ::= <scalar type>
 | <subrange type>
 | <type identifier>
```

with everything in `<structured type>`, other than `<array type>` ignored and `<simple type> reduced to \verb<subrange type>|`.

Additional information would be required in symbol tables to represent array variables. To

name – key for searching

token type – `TK_A_VAR`

static – flag (for now, assume YES)

type – TYPE; may include a pointer to detailed type descriptor or additional information (range)

size – optional (should be possible to derive from type)

address – computed during parsing

flags – for example, read only

initial value – if this is at all supported

we add (for TYPE being `TP_ARRAY`)

index type – ordinal type (int, char, bool, enum)

low limit of range

high limit of range

component type – limiting to ordinal type (int, char, bool, enum) or real.

NOTE: correct implementation would put these entries in a type entry in symbol table and point variable entry to it! - we are simplifying.

NOTE: while we leave the implementation to the reader, we notice that the program must verify that the constants defining the range are of the same type and that the low constant is not greater than the high.

10.17.2 LHS use

We need to reexamine our handling of assignments. The previously considered case of

```
a:= <RHS>  (*a is a variable*)
```

relies on the address of a being static, whereas

```
a[i]:= <RHS>  (*a is an array*)
```

needs to dynamically compute the address of the LHS; this is what we will begin with.

Upon finding a in the symbol table we will know that a is an array, and will have access to lo, hi and the two types.

```
int lo= // from symbol table
int hi= // from symbol table
int addr= // addr of the array, from symbol table
TYPE index_type = // from symbol table
TYPE component_type = // from symbol table

match(TK_LB);   // careful here (1)
TYPE t=E();
```

```
if (t!=index_type)  // careful here (2)
  error();


// generate code to generate address of LHS

if (lo!=0) {
  emit_opcode(OP_PUSHI);
  emit_int(lo);
  emit_opcode(OP_SUB);
  }

if (size(component_type)!=1) {
  emit_opcode(OP_PUSHI);
  emit_int(size(component_type));
  emit_opcode(OP_MUL);
  }

  emit_opcode(OP_PUSHI);
  emit_int(addr);
  emit_opcode(OP_ADD);

match(TK_RB);
match(TK_ASSIGN);   // careful here (3)

t=E();

if (t!=component_type)
  error();  // careful here (4)
```

At this point the generating code would result in two entries on the stack: first the address where we should stores the (RHS) value, second the value. We use previously mentioned `PUT` instruction to perform the operation.

In essence, `PUT` does pointer dereferencing, `*addr=val;`.

```
emit_opcode(OP_PUT);
```

10.17.3 RHS use

Arrays may also appear in the RHS. To support this, we need to update handling of factors (`F()`). Recall:

Grammar :

$E \Longrightarrow E + T \quad | \quad E - T \quad | \quad T$

$T \Longrightarrow T * F \quad | \quad T/F \quad | \quad F$

$F \Longrightarrow \quad | \quad -F \quad | \quad (E) \quad | \quad LIT \quad | \quad VAR$

The VAR case includes both single variable as well as an array element lookup. We proceed with very similar address computations

```
// in F(), once we know that the
//   variable is actually an array

int lo= // from symbol table
int hi= // from symbol table
int addr= // addr of the array, from symbol table
TYPE index_type = // from symbol table
TYPE component_type = // from symbol table

match(TK_LB);   // careful here (1)
TYPE t=E();
```

```
if (t!=index_type)  // careful here (2)
  error();


// generate code to generate address of LHS

if (lo!=0) {
  emit_opcode(OP_PUSHI);
  emit_int(lo);
  emit_opcode(OP_SUB);
  }

if (size(component_type)!=1) {
  emit_opcode(OP_PUSHI);
  emit_int(size(component_type));
  emit_opcode(OP_MUL);
  }

  emit_opcode(OP_PUSHI);
  emit_int(addr);
  emit_opcode(OP_ADD);
```

We now have the address of the data on the stack, we need to replace it by the actual value. The dereferencing `val=*addr` is assumed to be done by the `GET` instruction.

```
  emit_opcode(OP_GET);

  return component_type;
```

Exercise: *Explain "careful here" above!*

NOTE: To correctly support different sizes of data one should provide instructions `GET1`,`GET2`, `GET4`, `PUT1`,`PUT2`, `PUT4`.

10.17.4 RANGE check

One of the required features of Pascal is the generation of the range check code; such code should trap all *index out of range* errors. Effectively such code will generate an equivalent of

```
if (index<lo || index>hi)
  runtime_error();
```

before any use of array lookup `a[index]`.

Typical Pascal compiler makes this optional (overhead!), and usually supports embedded options `{$R+}`, `{$R-}` to allow range checking code generated only in part of the code.

The usual way is to write multiple assembler instructions to implement the code above, something like

```
DUP   % assume index is on stack
PUSHI lo
LSS
JTRUE runerror
DUP   % assume index is on stack
PUSHI hi
```

```
GTR
JTRUE runerror
```

with the cost of 8 instructions and extra work of resolving `error`. It would be more efficient to combine all of this into a single instruction

```
BOUND lo,hi
```

or perhaps

```
BOUND lo,hi,src-ref
```

Not in notes: discussion on Intel BOUND instruction and source ref.

10.18 Pointers

The arrays implementation above can be adapted to pointers, by removing unneeded work.

Pointers in Pascal

Sample pointer declaration and use in Pascal:

```
var a:integer;  p:^ integer;

begin
p=@a;

p^=100;  (*LHS use*)

writeln(p^); (*RHS use*)
end.
```

Declaration handling is simple.

The LHS and RHS use of pointers reduce to use of `PUT` and `GET` without the address computation. (The address computation may be present in implementation of `@`, for example, if we are computing an address of an array element!)

Pointer arithmetic is not difficult to implement.