**Compiled code vs. interpreted**

Translation into direct language... Interpreter parses and executes directly.

**Passes**

Physical transformation of the source.

2nd division of the program -> **Phases** - logical stages of compiling.

**Modules**

there are parts of the code which are not translated into working code.

**Scanner Phase**

Sample input / output:

Pascal line:

For I := 1 to 100

words are:

* for
* I
* :=
* I
* to
* 100

We probably don’t tokenize the entire source at once.

we need to key a symbol table for variables and other names when scanning. So as I collect info I can use it.

Scanner errors generally are if tokens can not be formed from source.

11232xzzcxv

string which is not closed.

invalid character (most languages don't accept 2nd 1/2 of ASCII table.

**Error Recovery**

This is difficult in scanner stage.

**Parser**

Convert from words into sentences. Most errors result from parser. Implementation of grammer occurs here.

**Constrainer**

**--------------------------------------**

**1 Pass Compiler**

Almost impossible to complete.

**Optimizer**

Most time is spent here on a commercial compiler. Everything else can be relatively straight forward.

**Optimization**

Reduction in strength. Example 2\*y ---> y + y.

Constant folding. Example x := 3 + 5 ----> x:=8

Common Sub Expression. Example a[I+j\*3] \* b[j+j\*3]

Dead code elimination

Optimizer can find errors. Example x := a / (5-5);

2013-09-10 --> Elementary Programming is going to be next

## 2013-09-17

**Scanner**

Input (File of chars) -> Sequence of Tokens

Tokens:

Keywords

Ids

Operators/Punctuation

Literals (Int, Real, Char, String)

Eolu (end of line)

EOF (End of file)

Pragmas

Every token is a pair:

Token Token-type Token Value Example

Keywords One for each Keyword Not Used #define TK.BEGIN 1

Identifiers TK\_ID Not Used ***Store current name as string***

Operators One token for each operator Not Used #define TK\_PLUS 40

Literals:

int TK\_INTLIT Value

Real TK\_REALLIT Index into table of reals

Char TK\_CHARLIT? Value

String TK\_STRLIT Index into a string table

EOLn TK\_EOLN

EOF TK\_EOF

Real Example:

3.0 1

0.0 2

1.0 3

1.0 3

0.0 2

Ids later becomes TK\_A\_VAR, TK\_A\_LABEL, ...... with token name pointing to symtab.

**Reading the source file:** *(Recommended in* ***BOLD****)*

1. Read text file line by line.
2. Buffers
3. Circular Buffers
4. **Just read the entire file.**
5. **Memory-mapped file. Operating system will manage the file as virtual memory.**

**Overall layout of scanner**

Assume that we are looking at one token at a time. LL(1) only one time to look at a token.

Globals:

int curtoken;//token type

curvalue,

curname //just for id's

char \*scanp //points to the current pos

int curline //for reporting

int cured

Entire program looks like this:

//open input file

scanp <- beginning of the file

curline <- 1

curCol <- 1

do {

gettoken(), //scanner

printtoken()

}while(curtoken != TK\_EOF);

**ACTUALLY WRITING THE SCANNER**

Possible:

{scanp}for i := 1 to 100 do [eof]

void gettoken(){

restart:

switch(catchcode[\*scanp]){

case LETTER:

case DIGIT:

}

}

catchcode is a predefined array.

case LETTER:

//copy all letters and digits to curname, advancing scaanp truncate OS per specs of the language.

Truncate as per specs of the language

Convert to upper/lower case if needed.

return keyword token if this is a keyword, otherwise return TK\_ID

case DIGIT:

Collect digits into integer value.

while(\*scanp >= '0' && \*scanp <= 'g'){

val = val\*10 + \*scanp - '0';

scanp++;

}

//check for .

1.1 3. .5

In pascal digits are required before and after decimal point while in C all of the above will be accepted.

case SPACE:

scanp++; //update line and col numbers

goto restart;

case OPERATOR\_CHAR:

C: '+' **can be** ++ += +

Pascal: '<' **can be** < <> <=

This also checks for comments.

C -> /\*.....\*/ or //

Pascal //............ or {......}

IF comment sequence are found, delete until end of line or end of comment character.

2013-09-24

Pascal comments { .... } (\* ..... \*)

a[i] a(.i.)

Append '0' to the end of the file when it is read in to determine EOF.

**Pragmas**

{$ R +} --> Range check on

{$I include.pas }

Stack of records like this

filename

buff

scanp

line

col

Locate the filename

Open the file, if not found report error

Push on stack.

------------------------------------------------------------------------------------------------------------------------------------------

Symbol table (S)

//Provide a search mechanism

//Pack the data

Symbol table = 1 dimensional char array

link key value

2 ways to store string = "BEGIN",0 or 5,"BEGIN"

|9|5|B|E|G|I|N| |16|3|E|N|D|0|

char symtab[10000];

short s

put s into symtab[i]; symtab[i+1]

symtab[i] = s/256;

symtab[i+1] = s%256;

\*(short\*)(symtab + i) = 2;

Hashing

26 List, one for each letter

A

B begin

C

D

E end else elseif

.

.

.

Z

Better hasing function

H linked list

Start array -> H links

After the initial link, we have the same thing as the previous lists.

Has function would match our string into the range 0 - H-1. This takes us the the start array. That then takes us to a single sequential search.

Compute a normal

for (int i=0; i<strlen(s); i++)

H <- h + s[i];

h <- h % 2\*;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2013-10-01

There is no specification for include in pascal

$i is recommended.

Scoping for variables.

Store variables in a stack. New var table for each scope.

var i: integer; Table Global

procedure p; Table for P

var i:real;

begin

end; Remove Table for P

----------------------------------------

Types of token tables:

Keywords - 1

Locals - 2 (may have nested procedures as well as with keyword)

Global - 3 (can have multiple compilation units)

=====================================================================================

PARSER

Context free Grammer

Set of terminal symbols T. (TOKENS)

Set of nonterminal symbols ||V|| Variables

Set of productions IP, each in the form:

n € N => Shrink over IN U TT

Goal Symbol G € IN

M -> (M)

M->

Terminal Symbol = {'(',')'}

IN = {M}

M -> (M) | e

Derivation of (()) :

M -> (M) -> ((M)) -> (())

Should never result into (()

Another language:

M -> (M)M

M -> e

Second language can result in ()() but the first one can't.

Multiple derivation of the same string is possible, but we always derive the left most term at all times.

"Left canonical derivation"

---------------------------------------------------------------------------------------------------------------------------------------

Expression Grammar:

E -> E + E

E -> id | lit

a + b

E

E + E

id(a) id(b)

a + b + c

E

E + E

E + E c

a b

a + b + c

E

E + E

a E + E

b c

The above is an ambiguous grammar. Same grammar, multiple parse trees.

Math op is associative if:

V a,b,c (a op b) op c = a op (b op c)

Computer Science:

[op] is left associative if:

a [op] b [op] c means (a [op] b) [op] c

Same thing for right associative.

Operations that are right associative:

assignment operator

a = b = c

\*\* in Fortran

^ in BASIC

2 \*\* 3 \*\* 2, first do the second operation then the first.

APL has lots of operators.

Every operator is right associative.

A <- 1-2-3 = 2

2 - 3 = -1

1 - (-1) = 2

E -> F + T | T

T -> id | lit

E

E + T

E + T

T b

a

Grammar is left associative.

E -> T + E | T

T -> id | lit

E

T + E

a T + E

b T

c

Grammar is right associative.

E -> E + T | E - T | T | E \* T

T -> id | lit

a + b \* c

E

E \* T

E + T c

T b

a

This results in addition before multiplication.

Change in Grammar:

E -> E + T | E - T | T

T -> T \* F | F

F -> id | lit

E

E + T

T T \* F

F F c

a b

----------------------------------------------------------------------------------------------------

E -> E + T | E - T | T | E or T

T -> T \* F | F | T/F | T div F | T mod F | T and F | T shl F | T shr F

F -> id | lit | -F | +F | (E) | not F

**2013-10-08**

Any production is converted to a recursive production.

M -> (M) M => void M() {

match('(');

M();

RHS

match(')');

M();

}

void match (token t){

if (t != curtoken){

error(); //process dies here...

}

gettoken();

}

void parse(){

gettoken();

go();

}

void M(){

if ( curtoken == '(' ){

match ( '(' );

M();

match( ')' );

M();

}

}

---------------------------------------------------------------------------------------------------------------

( ) ( )

M

( M ) M

| ( M ) M

| |

( ) )

this example will only match the first two braces.

Add to the grammar:

G => MX

in this version you match the string then the end of file marker.

So in the second example with the new grammar, we can output an error message saying that the second ')' is unexpected.

G

M [X]

( M ) M

| ( M ) M

| |

---------------------------------------------------------------------------------------------------------

E -> EX

E -> E + T | E - T | T

E -> T \* F | T / F | F

E -> id | lit | (E) | F

Non-working function for E:

void E(){

try either {

E(); match('+'); T();

} or {

E(); match('-'); T();

} or {

T();

}

}

A -> Ax (Left recursion)

Non recursive production must be present:

A -> Ax | B

----------------------------------------------------------------------------

A -> Ax1 | Ax2 | ............B1 | B2.................

A

A x

A x

Bx^k k >= 0

A -> BA'

A' -> xA' | E

A -> B1A' | B2A' | .......................

A' -> x1A' | x2A' | ................. | E

=====================================================================

G -> Ex

E -> TE'

E' -> + T E' | -T E' | Empty

T -> FT'

T' -> xFT' | FT' | Empty

F -> id | lit | (E) | -F

-------------------------------------------------------------------------------------------------------------------

*Not likely to happen*

A -> B{alpha}

B -> C{alpha}

C -> A{alpha}

---------------------------------------------------------------------------------------------------------------------

**Likely to happen:**

A -> B{alpha1} ===> A -> BA'

A -> B{alpha2} ===> A -> d1 | d2

=========================================================================

Examples:

void E' ( ) {

if ( curtoken == '+'){

match ( '+' ); T(); E' ();}

else{

match('-'); T(); E'();

}

else{

}

}

a \* b + c \* d

G ------> X

E

T E'

F T' + T E' ------> {blank}

a \* F T' F T' ---------> {}

b {} d

==================================================================================

G -> Ex

E -> TE'

E' -> + T E' | -T E' | Empty

T -> FT'

T' -> xFT' | FT' | Empty

F -> id˅ | lit˅ | (E) | -F ®

After applying this new grammar we have postfix notation.

----------------------------------------------------------------------------------------------------------------------------------------

1. Converter to postfix
2. Code generator for stack machine.
3. Native code gen?
4. Generate structure tree.
5. Take out expression and deal with variables

push a

push b

mul

push c

push d

mul

add

------------------------------------------------------------------------------

+

\* \*

a b c d