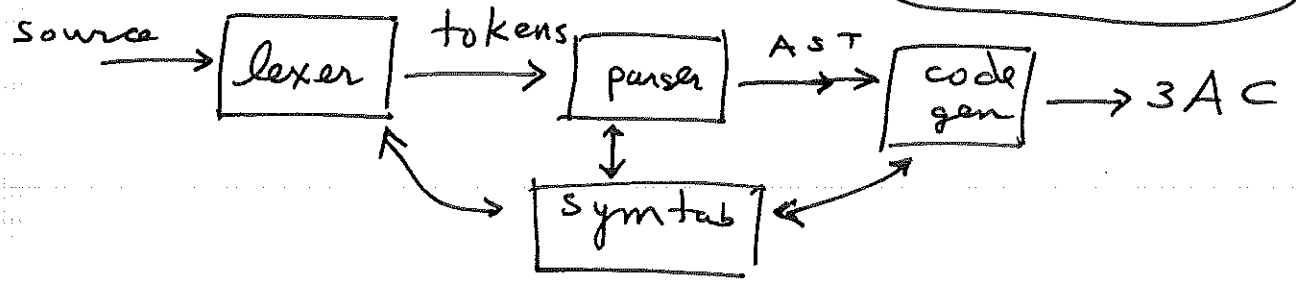
Dragon 2:39
CMPS-104A

Simple Compiler

ch 2 : p1



syntax - form of language
σύντάξις - arrange together

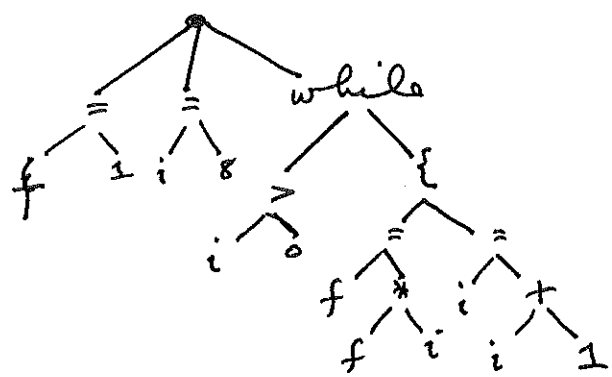
semantics - meaning

syntax - BNF = Backus-Naur Form.
- build AST

lex. syntax - regular exprs
- identify tokens

ex:
```
f = 1; i = 8;
while (i > 0) {
    f = f * i;
    i = i - 1;
}
```

AST



prefix
postfix
infix
matchfix

3AC

```
    f = 1
    i = 8
w1: if i<=0 goto w2
    f = f * i
    i = i + 1
    goto w1
w2:
```

note operator interior
operands children

Dragon 2:42

CMPS-104A

Syntax Defn

ex: <u>if</u> (expr) stmt [else stmt] ch 2: p2

BNF: stmt → <u>if</u> <u>(</u> expr <u>)</u> stmt [<u>else</u> stmt]

CFG = $\langle V_N, V_T, P, S \rangle$

- V_N: nonterminals
- V_T: terminals
- P: set of rules (productions)
- S: start symbol.

token = terminal + semantic attr + lex attr.

example (ETF)

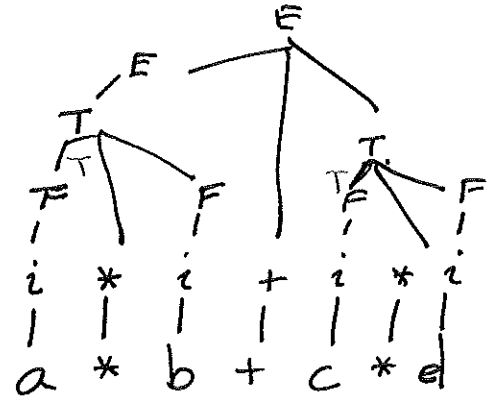
$V_N = \{E\ T\ F\}$

$V_T = \{+\ *\ i\}$

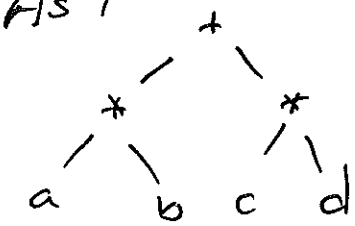
$S = E$

$$P = \left\{ \begin{array}{l} E \to E+T \\ E \to T \\ T \to T*F \\ T \to F \\ F \to (E) \\ F \to i \end{array} \right\}$$

shift/reduce parse



AST



parse trees
- trace

AST
- product

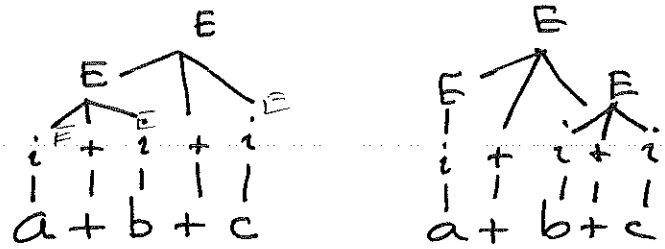
derivation vs parse

Dragon 2:47
CMPS - 104A
ch 2: p3

Ambiguity

>1 parse tree given V_T^*

Ex: $E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to i$



ambiguity ok in English
NOT OK in prog lang.

Precedence & Associativity

make table

~~right~~	=	
left	+	−
left	*	/

$a+b+c \equiv (a+b)+c$

$a=b=c \equiv a=(b=c)$

given ambig: prec diff use higher prec op
prec same use assoc.

show shiftreduce above.

Dragon 2:51
CMPS-104A

ch2: p4

Dangling Else.

$S \rightarrow$ if (E) S else S
$\rightarrow$ if (E) S
$\rightarrow X$

so parse if(E_1) if(E_2) S_1 else S_2

discuss { } req - Perl
elsif Ada

<u>Example</u> Keywords.

```
stmt  -> ID = expr ;
       | if ( expr ) stmt else stmt
       | if ( expr ) stmt
       | while ( expr ) stmt
       | { stmts }
stmts -> stmts stmt
       |
```

~~Tree Traversals~~

- ~~build parse tree or AST.~~
- ~~traverse tree (depth first)~~

Dragon 2:52
CMPS - 104A

ch 2: p5

Syntax Directed Xlation

$E_1 \rightarrow E_2 + T$	print "+"	$E_1.t = tree(+, E_2.t, T.t)$
$E \rightarrow T$		$E.t = T.t$
$T_1 \rightarrow T_2 * F$	print "*"	$T_1.t = tree(*, T_2.t, F.t)$
$T \rightarrow F$		$T.t = F.t$
$F \rightarrow (E)$		$F.t = E.t$
$F \rightarrow i$	print i	$F.t = i$

show post order traversal of tree

infix $\longrightarrow$ postfix

attributes: synthetic
inherited.

Parsing

- how can strings of terminals be generated

Parsing = (Derivation)$^{-1}$ ← inverse

[$\ln = \exp^{-1}$

Bottom-up < shift reduce, op. prec, $L_R(k)$ — large class

Top-down < recursive descent, $LL(k)$ — easier by hand

any CF grammar: $O(n^3)$ — ambiguous

$O(n^2)$ — unambiguous

$LR(k)$ $O(n)$ — actual languages

CMPS-104A

ch2: p6

Dragon 2:61

Top down parsing

- one fn for each $\in V_N$

$$G = \langle V_N, V_T, P, S \rangle$$

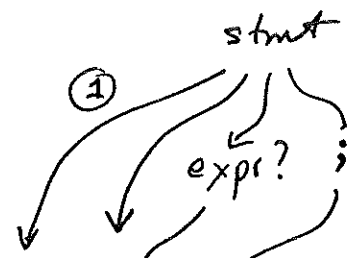
- start with start sym S
- select Rule(P) based on 1st sym.

ex: stmt → expr ';'
| if (expr) stmt
| for (expr? ; expr? ; expr?) stmt
| other

expr? → expr | ε

---

top down



etc.

input: for (ε ; expr ; expr) other

consider node:

V_T → match lookahead & advance

V_N → call parsing function.

general: trial & error

- try rule & backtrack ⇒ $O(n^2)$
- if LL(1) then always know

Predictive Parsing

CMPS-104A
ch 2:7
Dragon 2:64

- recursive descent
- top down.
- lookahead symbol.

```
main () { la = scan() ; stmt () }
stmt () { switch (la) {
            case EXPR : match (EXPR)
                        match (';')
                        break
            case IF:    match (IF)
                        match ('(')
                        match ( EXPR)
                        match (')')
                        stmt ()
                        break
            case FOR :  match (FOR)  match ('(')
                        optexpr ()   match (';')
                        optexpr ()   match (';')
                        optexpr ()   match ('>')
                        stmt () ; break
            case OTHER : match (OTHER)
                         break
            default :  syntax error ()
        }
optExpr () { if (la == EXPR) match (EXPR) }
match (t) {
        if (la == t) la = scan()
        else syntaxerror ()
}
```

~~LR(0)~~

flex
tok code < 256 = char
≥ 256 = tok code

Design Pred. Pars

CMPS-104A
ch 2: 8
Dragon 2: 65

- unia First sets.
- ambig ⇒ choose non-ε

• Left Recursion won't work.

$$B \to B + C$$
$$B \to C$$

} to B or not to B?

$$B \to C\ B'$$
$$B' \to +\ C\ B'$$
$$B' \to$$

} stop when no +

or use a loop: $B \to C \{+ c\}^*$

2.5 Transl for simple exprs

CMPS-104A
ch 2:9

Dragon 2:68

input : ETF plus.
output : RPN

concrete syntax : what you type
abstract syntax : elim punctuation

~~E → E+T | E-T | T~~
~~T → T*F | T/F | F~~
~~F → (E) | num | id~~

$E \to E + T$
$E \to E - T$
$E \to T$
$T \to T * F$
$T \to T / F$
$T \to F$
$F \to (E)$
$F \to i$
$F \to n$

left recursive

elim

$E \to T E'$
$E' \to + T E'$
$E' \to - T E'$
$E' \to$
$T \to F T'$
$T' \to * F T'$
$T' \to / F T'$
$F \to (E)$
$F \to i$
$F \to n.$

right recursive
||
tail recursive

loop

$E \to T \{ \pm T \}^*$

$T \to F \{ {}^{*}_{/} F \}^*$

$F \to (E) \mid i \mid n$

loop

Right (Tail) Recursive ⇒ loop.

CMPS-104A
ch 2:10
Dragon 2:73

```
main(){ la = scan();
        expr();
      }
expr() { term();
         for(;;) {
                switch (la) {
                      case '+' : match('+');
                                 term()
                                 ~~print~~ ("ADD")
                                 break
                      case '-' : match('-');
                                 term()
                                 print ("SUB")
                                 break.
                      default:
                                return
                }
         }
      }
term() { factor();
```

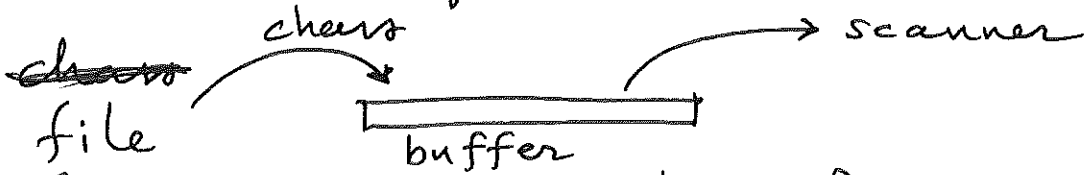
same except + → *
- → /

```
}
factor() {
      switch (la) {
            case NUM:  print("push" + la.string
                       match(NUM);
                       break;
            case ID :  print(la)
                       match("push" + la.string
                       break;
            case '(' : match('(')
                       expr();
                       match (')')
                       break
            default:
                       error();
}
```

```
match (t) {
    if (la == t) la = scan()
    else error()
}
```

CMPS-104A
ch 2:11
Dragon 2:75

2.6 Lexical Analysis



lexeme = seq of chars } token.
$\in V_T$ = terminal

```
char buffer[MAX];
int scan() {
    char *bp = buffer;
    for (;;) {
        t = getchar();
        if (isspace(t)) continue;   //white space
        switch (t) {
            case '+': case '-':
            case '*': case '/':
            case '(': case ')':
                *bp++ = t;
                *bp = '\0';
                return t;
            default:
        }
        if (isdigit(t)) {
            do { *bp++ = t;
                 t = getchar();
            while (isdigit(t));
            ungetc(t); //unpeek
            *bp = 0;
            return NUM
        }
```

if (t == EOF) return EOF;

CMPS-104A

ch 2:12

Dragon 2:76

~ cont ~

```
~~if (isalnum(t))~~
if (isalpha(t)) {
        do { *bp++ = t;
             t = getchar();
        } while (isalnum(t));
        ungetc(t);
        *bp = 0;
        return ID
}
error("bad char")
}
```

---

what about < <= = ==, ...

```
case '<':
        *bp++ = t;
        t = getchar();
        switch (t) {
            case '=': *bp++ = t; *bp = ∅;
                      return LE;
            case '<': *bp++ = t; *bp = 0;
                      return SHL;
            default: ungetc(t);
                     *bp = 0;
                     return LT
        }
}
```

EOF = -1

NUM = 256

ID = 257

CMPS-104A
ch 2:13

Dragon 2:79

Recognizing Keywords

- put in grammar
  → V. large machine
- second table lookup
- pre-insert into string table

Symbol Table

- 2nd hash table.
- block structure stack
- sym table per scope