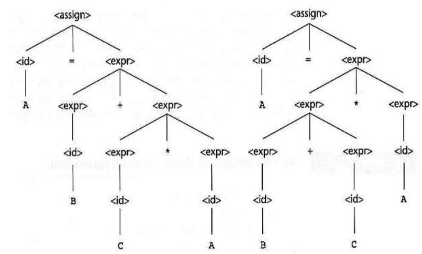
BNF Fundamentals

<LHS> → <RHS>
LHS:abstraction being defined
RHS:definition
"→" means "can have the form"
::= is used for →
<assign> → <var> = <expression>,this is a rule.var and expression be defined.
-These abstractions are called variables or nonterminals
-lexemes and tokens are the terminals
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals.
<LHS> → <RHS>
Grammar: a finite non-empty set of rules.
Example:
Mary greets Alfred
<sentence> ::= <subject><predicate>
<subject> ::= <noun>
<predicate> ::= <verb><object>
<verb> ::= greets
<object> ::= <noun>
<noun> ::= Mary | John | Alfred

Grammars and Derivations

-The sentences of the language are generated through a sequence of applications of the rules, starting from the special nonterminal called start symbol.Such a generation is called a derivation.
If always the leftmost nonterminal is replaced, then it is called leftmost derivation.
A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees.
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>
Parse trees of **A = B + C * A**



<expr> -> <expr> + <expr> | const (ambiguous)
<expr> -> <expr> + const | const (unambiguous)
-In a BNF rule, if the LHS appears at the beginning of the RHS, the rule is said to be left recursive. Left recursion specifies left associativity.
<expr> ::= <expr> + <term>
| <term>
-most of the languages exponential is defined as a right associative operation
<factor> ::= <expr> ** <factor>
| <expr>
<expr> ::= (<expr>)
| <id>
Extended BNF: Optional parts are placed in brackets [ ]. Repetitions (0 or more) are placed inside braces { }. Alternative parts of RHSs are placed inside parentheses and separated via vertical bars.

Designing Patterns(Lex)

[abc] matches **a, b or c**
[a-f] matches **a, b, c, d, e, or f**
[0-9] matches **any digit**
X+ matches **one or more of X**
X* matches **zero or more of X**
[0-9]+ matches **any integer**
(…) grouping an expression into a single unit
(a|b|c)* is equivalent to **[a-c]***
X? X is optional **(0 or 1 occurrence)**
if(def)? matches if or ifdef **(equivalent to if|ifdef)**
[A-Za-z] matches **any alphabetical character**
. matches **any character except newline character**
\. matches **the . character**
\n matches **the newline character**
\t matches **the tab character**
**** matches **the \ character**
[\t] matches **either a space or tab character**
[^a-d] matches **any character other than a,b,c and d**
Real numbers
[0-9]*(\.)?[0-9]+
To include an optional preceding sign:
[+-]?[0-9]*(\.)?[0-9]+
Integer or floating point number
[0-9]+(\.[0-9]+)?
Integer, floating point or scientific notation.
[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?

-Lex builds the **yylex()** function that is called, and will do all of the work for you.
-Lex also provides a count **yyleng** of the number of characters matched.
-yywrap is called whenever lex reaches an end-of-file
-Lex is a tool for writing **lexical analyzers.**
-Yacc is a tool for **constructing parsers.**

Yacc

-Yacc stands for **y**et **a**nother **c**ompiler to **c**ompiler. Reads a specification file that codifies the grammar of a language and generates a parsing routine.
-Yacc specification describes a **Context Free Grammar (CFG)**, that can be used to generate a parser.
Elements of a CFG:
1. Terminals: tokens and literal characters,
2. Variables (nonterminals): syntactical elements,
3. Production rules, and
4. Start rule.
-Format of a **yacc** specification file:
declarations
%%
grammar rules and associated actions
%%
C programs
-Declarations
%token: declare names of tokens
%left: define left-associative operators
%right: define right-associative operators
%nonassoc: define operators that may not associate with themselves
%type: declare the type of variables
%union: declare multiple data types for semantic values
%start: declare the start symbol (default is the first variable in rules)
%prec: assign precedence to a rule
%{ C declarations directly copied to the resulting C program %}
$$: left-hand side
$1: first item in the right-hand side
$n: n[th] item in the right-hand side
-Yacc provides a special symbol for handling errors. The symbol is called error and it should appear within a grammar-rule. **-yylex()** function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yylval.**

```
/* lex specification */
%%
a return A;
b return B;
\n return NL;
. ;
%%
int yywrap() { return 1; }
```

Actions between Rule Elements

input: ab
output: 1452673

input: aa
output: 14526 syntax error

input: ba
output: 14 syntax error

```
/* yacc specification */
%{
#include <stdio.h>
%}
%token A B NL
%%
s: {printf("1");}
   a
   {printf("2");}
   b
   {printf("3");}
   NL
   {return 0;}
   ;
a: {printf("4");}
   A
   {printf("5");}
   ;
b: {printf("6");}
   B
   {printf("7");}
   ;
%%
#include "lex.yy.c"
int yyerror(char *s) {
  printf ("%s\n", s);
}

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

Scheme

1.(DEFINE (compare x y)
  (COND
    ((> x y) "x is greater than y")
    ((< x y) "y is greater than x")
    (ELSE "x and y are equal")))
2.(CONS 'A '(B C)) returns (A B C)
3.(LIST 'apple 'orange 'grape) return (apple orange grape)
4.(CAR '(A B C)) yields A
5.(CAR '((A B) C D)) yields (A B)
6.(CDR '(A B C)) yields (B C)
7.(CDR '((A B) C D)) yields (C D)
8.(LIST? '()) yields #T
9.(NULL? '(())) yields #F
10.(DEFINE (member atm lis)
  (COND
    ((NULL? lis) #F)
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR lis)))))
11.(DEFINE (second a_list) (CAR (CDR a_list)))
 (second '(A B C)) = returns B
12.(CADDAR x) = (CAR (CDR (CDR (CAR x))))
 (CADDAR '((A B (C) D) E)) = returns (C)
13.(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
      (equalsimp(CDR lis1)(CDR lis2)))
    (ELSE #F)))
14.(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1))(EQ? lis1 lis2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((equal (CAR lis1) (CAR lis2))
      (equal (CDR lis1) (CDR lis2)))
    (ELSE #F)))
15.(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1)
      (append (CDR lis1) lis2)))))
 *(append '(A B) '(C D R)) returns (A B C D R)
 *(append '((A B) C) '(D (E F))) returns ((A B) C D (E F))
16.(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c)))(* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
  (DISPLAY (+ minus_b_over_2a root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a root_part_over_2a)))
17. Original: (DEFINE (factorial n)
    (IF (= n 0)
      1
      (* n (factorial (- n 1)))))
Tail Recursive: (DEFINE (facthelper n factpartial)
    (IF (= n 0)
      factpartial
        facthelper((- n 1) (* n factpartial)))))
    (DEFINE (factorial n)
      (facthelper n 1))
18.(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
  ((compose CAR CDR) '((a b) c d)) yields c
19.(DEFINE (third a_list)
  ((compose CAR (compose CDR CDR)) a_list)) is equivalent to CADDR
20.(DEFINE (map fun lis)
  (COND
    ((NULL? lis) ())
    (ELSE (CONS (fun (CAR lis))
      (map fun (CDR lis))))))
21.(map (LAMBDA (num) (* num num num)) '(3 4 2 6)) yields (27 64 8 216)
22.((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis)))))
23.(DEFINE sum
  (lambda (l)
    (if (null? l)
      0
      (+ (car l) (sum (cdr l))))))
24.(DEFINE product
  (lambda (l)
    (if (null? l)
      1
      (* (car l) (product (cdr l))))))
25.(DEFINE length
  (lambda (l)
    (if (null? l)
      0
      (+ 1 (length (cdr l))))))
26.(DEFINE reverse
  (lambda (l)
    (if (null? l)
      nil
      (append (reverse (cdr l)) (list (car l))))))
27.(DEFINE (third list)
  (caddr list))
 (third '(1 2 3 4 5 6 7)) returns 3
28. (DEFINE a 1)
  (DEFINE b 2)
  (DEFINE c 3)
  (let ((a 2)
    (b (+ a 7))
    (c b))
  (+ a b c)) returns 12
29.(DEFINE (my-func f)
  (lambda (x y) (f (f x y) (f x y))))
 (my-func *) 2 4) returns 64
30.(DEFINE (smallest x y z)
  (min x y z))

Subprograms

1.Pass-by-value(In Mode): Changes made to the function do not change the actual parameter.
2.Pass-by-result(Out Mode): No value is transmitted to the subprogram.The corresponding formal parameter acts as a local variable.Its value is transmitted to caller's actual parameter when control is returned to the caller.Require extra storage location and copy operation.
Example: Subprogram sub(x, y) { x <- 3 ; y <-5;}
call:
sub(p, p)
what is the value of p here ? (3 or 5?)
-The values of x and y will be copied back to p. Which ever is assigned last will determine the value of p.
-The order is important.
3.Pass-by-value-result(Inout Mode): A combination of

pass-by-value and pass-by-result.Formal parameters have local storage.The value of the actual parameter is used to initialize the corresponding formal parameter.
-The formal parameter acts as a local parameter.
-At termination, the value of the formal parameter is copied back.

4.Pass-by-reference(Inout Mode): Changes made to the function do change the actual parameter.

5.Pass-by-name(Inout Mode): The names of the variables sent to the function are matched with the parameter names in the function. Thus, any change in the parameter changes the value of the actual variable corresponding to that parameter.

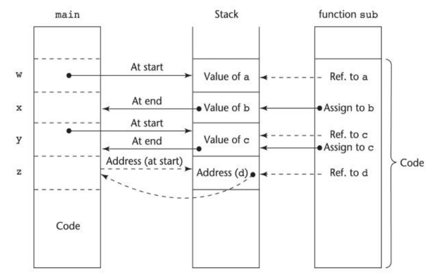
```
function joe(int a, int b, int c)
 begin
   a := b + c;
   b := c + 1;
   print a, b, c;
 end
function main
 begin
   int i := 5;
   int j := 10;
   ink k := 15;
   joe(i, j, j + k);
   print i, j, k;
 end
```

1.All parameters are passed by value.
35 26 25
5 10 15 --parameters are independent variables initialized to the values of the argument expressions. Changes to them do not effect the arguments.

2.Pass a and b by reference, and c by value.
35 26 25
35 26 15 --This is very much same,except the changes to a and b are also made to i and j since these parameters are aliases.

3.Pass a and b by value-result, and c by value.
35 26 25
35 26 15 --This has the same effect as pass-by-reference, since the values of a and b are returned to i and j when the function returns.

4.All parameters are passed by name.
35 26 41
35 26 15 --This is again similar to the last two,since changes to a and b also change i and j. The difference is the value of 41 printed for c, since c is an alias for the expression j+k. The assignment b := c + 1 changes b to 26, which changes j, which changes j+k, now 26+15.



Function header: `void sub(int a, int b, int c, int d)`
Function call in main: `sub(w, x, y, z)`
(pass w by value, x by result, y by value-result, z by reference)

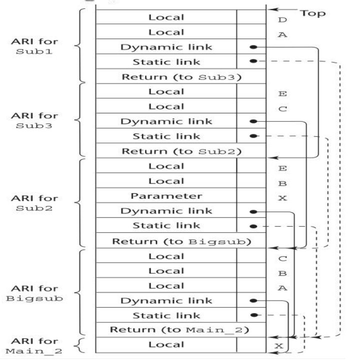
```
Example:
function sub1() {
  var x;              2:declared in        Passed subprogram S2
  function sub2() {                        Output:
    window.status = x;                     is called by S4
  } // sub2                                is declared in S1
  function sub3(){                         is passed in S3
    var x;
    x = 3;
    sub4(sub2);        3: passed in
  } // sub3
  function sub4(subx) {
    var x;                                 1- Shallow binding
    x = 1;
    subx();           1:called by          2- Deep binding
  } // sub4
  x = 2;                                   3- Ad hoc binding
  sub3();
} // sub1
```

-An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment.
-A generic or polymorphic subprogram takes parameters of different types on different activations.
-A coroutine is a special kind of a subprogram that has multiple entries and controls them itself.Coroutines call is a resume.

Implementing Simple Subprograms

-Two separate parts: the actual code (constant) and the non-code part (local variables and data that can change).
-The format, or layout, of the non-code part of an executing subprogram is called an activation record.
-The form of an activation record is static.
-An activation record instance (ARI) is a concrete example of an activation record.

```
procedure Main_2 is
  X : Integer;                                Main_2 calls Bigsub
  procedure Bigsub is                             Bigsub calls Sub2
    A, B, C : Integer;                            Sub2 calls Sub3
    procedure Sub1 is                             Sub3 calls Sub1
      A, D : Integer;
      begin -- of Sub1
      A := B + C;  <-------------------------1
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        Sub1;
        E := B + A:   <---------------------2
        end; -- of Sub3
      begin -- of Sub2
      Sub3;
      A := D + E;  <-------------------------3
      end; -- of Sub2 }
    begin -- of Bigsub
    Sub2(7);
    end; -- of Bigsub
  begin
  Bigsub;
end; of Main_2 }
```



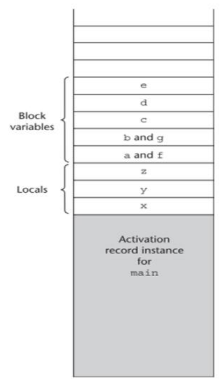
static link parent fonksiyona gider.dynamic link çağrıldığı yere gider.
(CHAIN_OFFSET, LOCAL_OFFSET)
At position 1 in sub1:
A - (0,3)
B - (1,4)
C - (1,5)
At position 2 in sub3:
E - (0,4)
B - (1,4)
A - (2,3)
At position 3 in sub2:
A - (1,3)
D - error
E - (0,5)

Dynamic chain (call chain): The collection of dynamic links in the stack at a given time.
-Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the local_offset.

Implementing Static Scoping

-A static chain is a chain of static links that connects certain activation record instances.
-The static link in an activation record instance for subprogram A points to the bottom of one of the activation record instances of A's static parent.
-The static chain from an activation record instance connects it to all of its static ancestors.

```
void main() {
  int x, y, z;
  while ( ... ) {
    int a, b, c;
    ...
    while ( ... ) {
      int d, e;
      ...
    }
  }
  while ( ... ) {
    int f, g;
    ...
  }
  ...
}
```

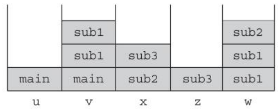


-Static_depth is an integer associated with a static scope whose value is the depth of nesting => indicates how deeply it is nested in the outermost scope.
STATIC_DEPTH-> scope iç içe geme sırası. ağaçtaki derinliği.
-The chain_offset or nesting_depth of a nonlocal reference is the difference between the static_depth of the reference and static_depth of the procedure containing its declaration.
CHAIN_OFFSET-> variablenın declare edildiği yerin ilk tanımlandığı parent scope a olan uzaklığı.
LOCAL_OFFSET-> variablenın ilk olarak tanımladığı fonksiyonun ARI'sindeki alttan yukarıya sırası.0'dan baslayıp.
-A reference to a variable can be represented by the pair: (chain_offset, local_offset), where local_offset is the offset in the activation record of the variable being referenced.

Implementing Dynamic Scoping

-Deep Access: non-local references are found by searching the activation record instances on the dynamic chain.
Length of the chain cannot be statically determined.
Ength of the chain cannot be statically determined.
-Shallow Access: put locals in a central place.
One stack for each variable name.
Central table with an entry for each variable name.

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3

Logical Programming

Name	Symbol	Example	Meaning
negation	¬	¬ a	not a
conjunction	∩	a ∩ b	a and b
disjunction	∪	a ∪ b	a or b
equivalence	≡	a ≡ b	a is equivalent to b
implication	⊃	a ⊃ b	a implies b
	⊂	a ⊂ b	b implies a

Quantifiers

Name	Example	Meaning
universal	∀X.P	For all X, P is true
existential	∃X.P	There exists a value of X such that P is true

Booleans

No	false value
fail	false value
not(4)	logical not
; / .	logical or / and (short circuit)
true	true value
Yes	true value

-parent(X,Y) :- mother(X,Y). >>> "X, Y'nin annesi ise X, Y'nin ebeveynidir."
-grandparent(X,Z):- parent(X,Y), parent(Y,Z). >>> "Y, Z'nin ebeveyni ve X'de Y'nin ebeveyni ise X, Z'nin atasıdır."

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-   speed(X,Speed),
                   time(X,Time),
                   Y is Speed * Time.
```

A query: `distance(chevy, Chevy_Distance).`

Example

```
likes(jake,chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).
((1)) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
((1)) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)
X = apricots
```