# CENG444:
# Semantic Analysis
# Code Generation
# Run-time Organization

Cem Bozşahin

Cognitive Science Department, Informatics Institute,
Middle East Technical Univ. (ODTÜ), Ankara

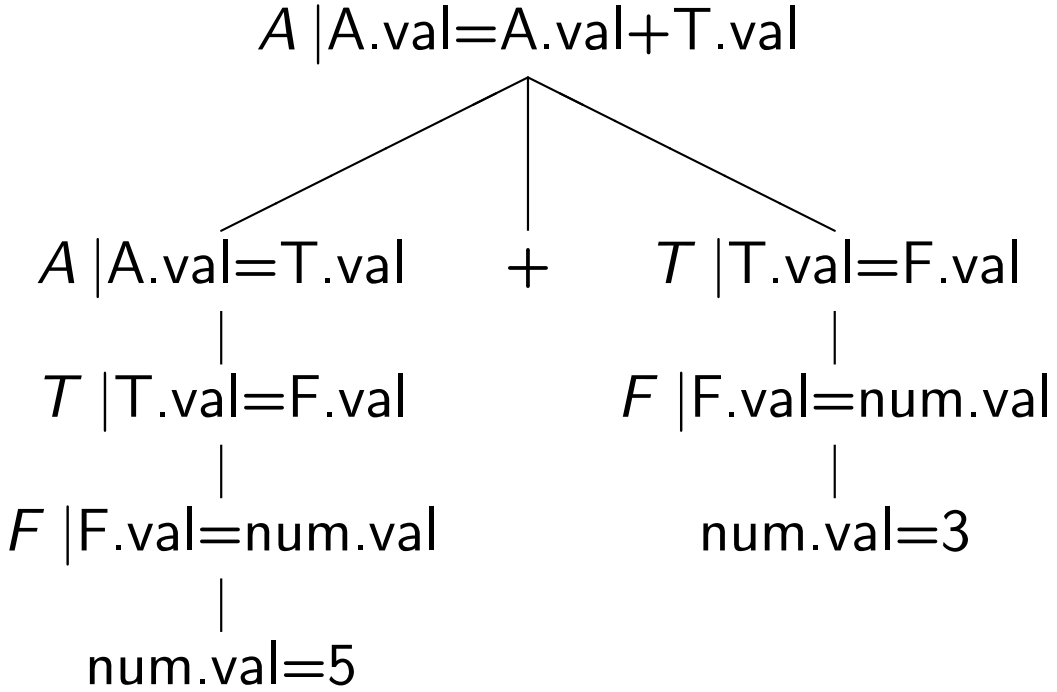
# Attribute Grammar

- Attribute/feature: a value associated with a grammar variable at parse time.
- An attribute grammar is a CFG in which the grammar symbols have attributes associated with them.
- This actually *extends* the power beyond context-freeness, but the form of the grammar is similar to CFGs in the sense that there is still one symbol on the LHS (in general, this is called a phrase structure grammar).
- AGs help define form-meaning correspondences.

- ex: A calculator (this is syntax-directed evaluation)

```
CF rule    semantic action
--------   --------------------

A -> A+T   {A0.val = add(A1.val,T.val)}

F -> num   {F.val = num.val}
```

ex: a decorated (annotated) parse tree for 5+3




- In what order the information is passed?
  From RHS to LHS: synthesized attributes
  From LHS to RHS: inherited attributes

- Synthesized: $X.a \rightarrow Y_1.a \cdots Y_n.a$
  X.a is a function of $Y_i.a$

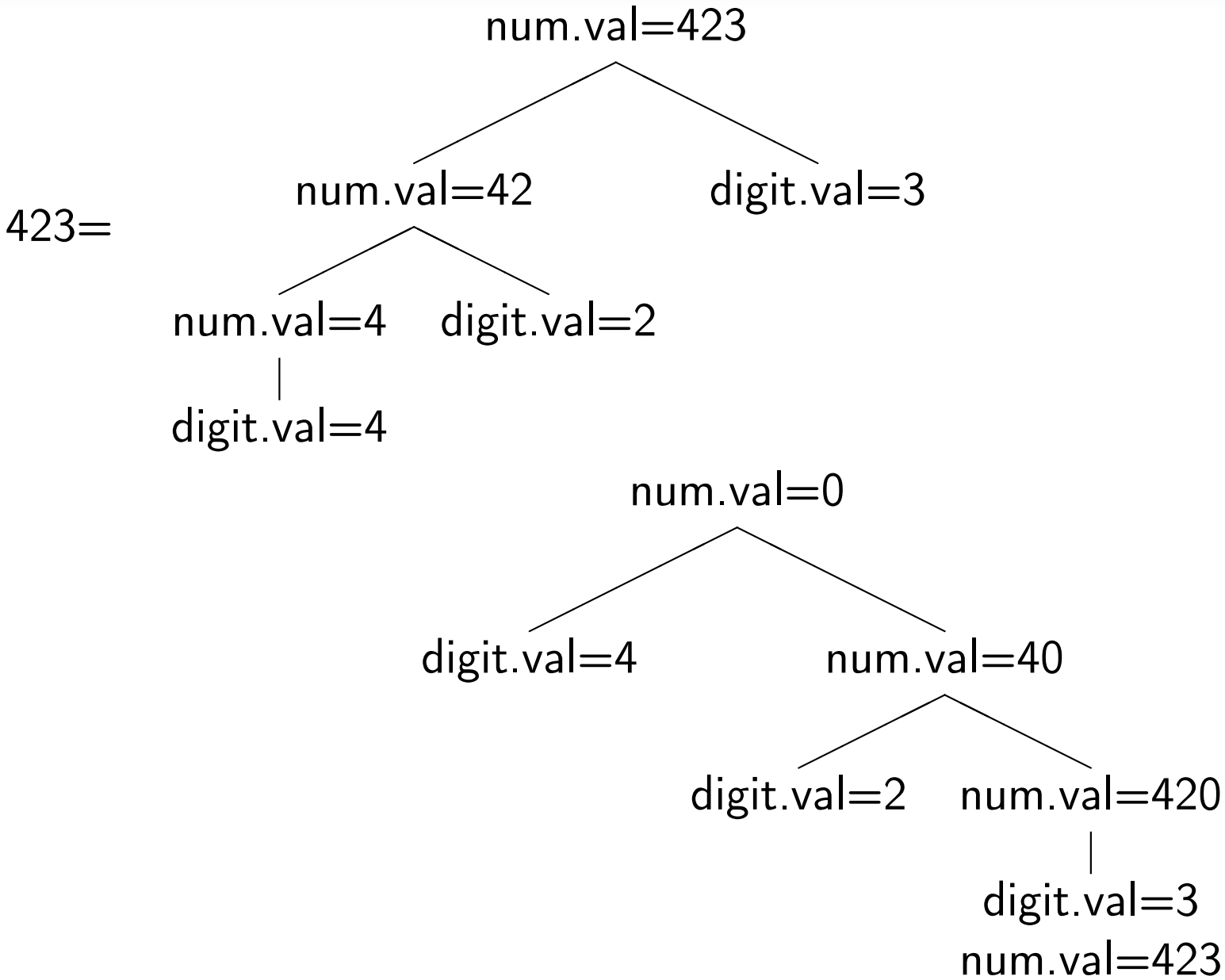
- Inherited: $X.a \rightarrow Y_1.a \cdots Y_n.a$
  $Y_k.a$ is a function of $X$ and $Y_i.a, i \neq k$

ex: synthesized vs. inherited derivation of numbers

```
Num -> Digit      {num.val=digit.val}
Num -> Num Digit  {Num1.val=Num2.val*10
                             +Digit.val}
```

```
Num -> Digit      {Num1.val=Num1.val
                             +Digit.val}
Num -> Digit Num  {Num2.val=(Num1.val
                             +Digit.val)*10}
```

assume initially num.val=0


num.val=423
423=
num.val=42
digit.val=3
num.val=4
digit.val=2
digit.val=4
num.val=0
digit.val=4
num.val=40
digit.val=2
num.val=420
digit.val=3
num.val=423

- Composition of semantics reflects the underlying parsing strategy as well.

  ex: checking the declaration of variables in top-down parse (assume D.dl=nil initially)

```
P -> D S          {S.dl = D.dl}

D -> var V ; D    {D2.dl=addlist(V.name,D1.dl)}

D -> null         {}

S -> V := E ; S   {check(V.name,S1.dl);
                   S2.dl=S1.dl}

V -> id           {V.name=id.val}
```

At what time do we execute the semantic action? In above convention, dependency of one attribute over another tells you when to execute (after D is recognized in 1st rule)

But, the time of semantic action can be made explicit by putting it in a position where it can be evaluated

```
P -> D {S.dl = D.dl} S
```

The latter convention is known as the *translation scheme*. It is a special case of syntax-directed definition in which rule evaluation and attribute evaluation use the same order and strategy.

But, in general, syntax-directed definitions can separate rule and attribute evaluation by dependency graphs.

- S-attributed grammars: only synthesized attributes

  L-attributed grammars: All inherited attributes in a rule are a function only of symbols to their left

- if L-valued, a grammar can be used to parse top-down depth-first.

  If not, leftmost derivations are unable to evaluate $Y_j$ for some $j > k$.

- YACC uses synthesized attributes

- antLR can do both: tree parsing

# Parse trees and Attribute evaluation

- Tree parsing decouples parsing strategy and semantic composition by building Abstract Syntax Trees (AST), which can be traversed in any order to maintain the attribute dependency.

- Parse trees are useful if further (or more detailed) processing and checking needs to be done on structured representation of the source language.

  The most 'natural' grammar may not be the most 'parsable' grammar. A top-down parser might like to save a left-branching parse tree but use a right-branching grammar (due to e.g. easier construction of semantics in the next phase)

A translator might like to know more about internal structure of statements than the rules reveal, e.g., nesting of loops

Order of attribute evaluation can be different than the order of rule use in derivations (e.g., inherited attributes in a bottom-up parser or synthesized attributes in a top-down parser)

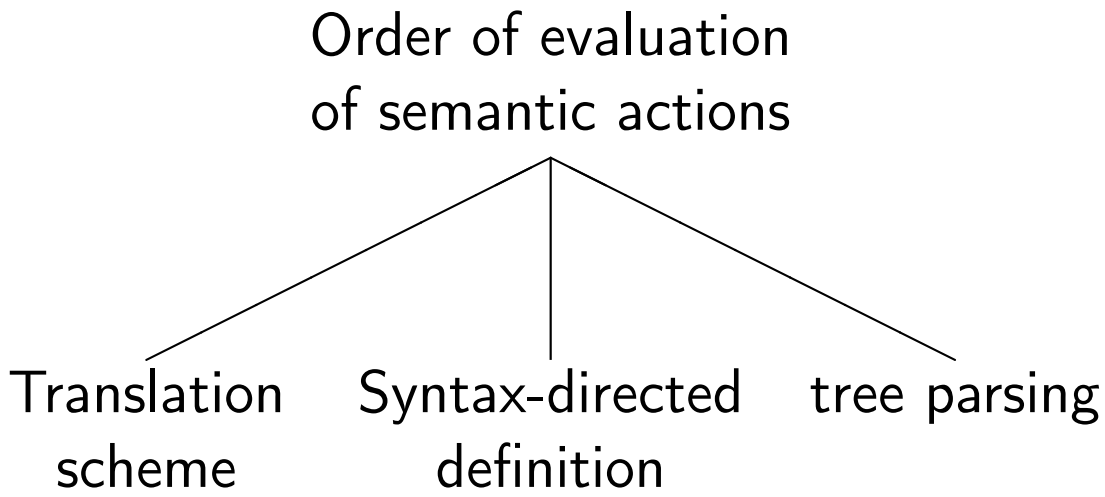
# Decoupling syntax from LL/LR derivations

(Doing manually what antlr does automatically)

building syntax (parse) trees:

```
mknode(op,left,right)
mkleaf(id,entry)
mkleaf(num,value)
```

```
E -> E + T    E.nptr:=mknode('+',E1.nptr,T.nptr)

E -> E - T    E.nptr:=mknode('-',E1.nptr,T.nptr)

E -> T        E.nptr:=T.nptr

T -> (E)      T.nptr:= E.nptr

T -> id       T.nptr:=mkleaf(ID,id.entry)

T -> num      T.nptr:=mkleaf(NUM,num.val)
```




- Translation scheme: Order of semantic actions are explicitly shown within the RHS of a rule; their evaluation depends on the parsing strategy.

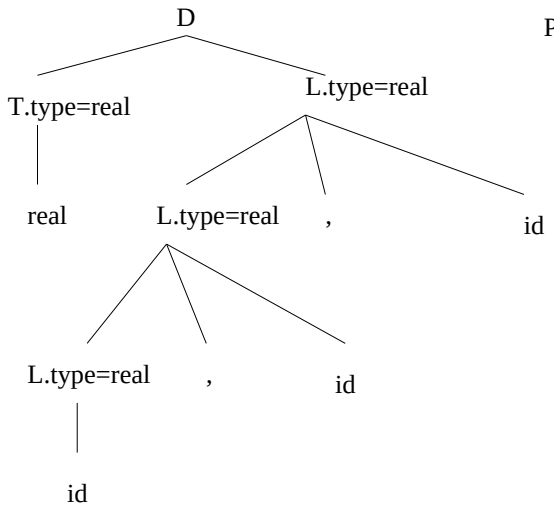
- Tree parsing: our rules do less work than they could.

- SDD: Syntax-directed definition (official use), Syntax-directed derivation of semantics (my use).

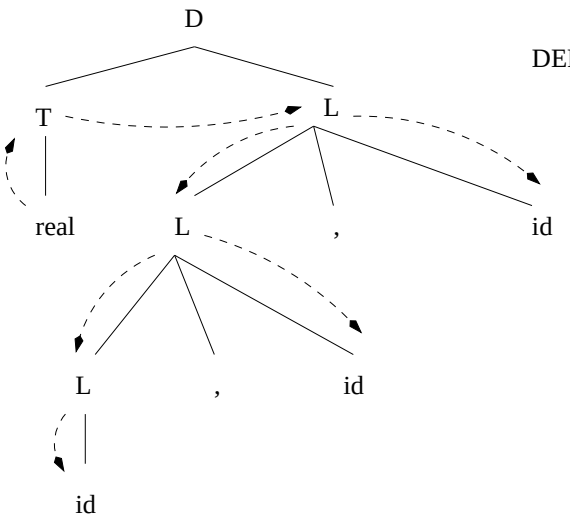
- Syntax-directed definition: The semantic action is associated with the rule. Its order of execution depends on the dependencies among attributes.

  ex: declaring types of several vars

```
D -> T L {L.type=T.type}

T -> int {T.type=integer}

T -> real {T.type=real}

L -> L , id {id.type=L0.type;
             L1.type=L0.type }

L -> id     {id.type=L.type}
```




PARSE TREE




DEPENDENCY GRAPH

- Order of semantic actions:
  1. Topological sort of dependency graph. Earlier nodes are evaluated before dependent nodes.
  2. Set up heuristic rules before compiler construction.
  3. Order is dictated by parsing strategy, not by dependence among attributes (oblivious methods).

- Why `yacc` allows 1-1 pairing as in oblivious methods? If all attributes are synthesized, the dependency is always from right-to-left, hence bottom-up. Order of evalation is in lock step with the order of parsing.

- But, `yacc` can "simulate" a translation scheme by allowing limited kind of actions *within* the RHS. Since the scan is left-to-right, the symbols to the left in the RHS are recognized *before* symbols to the right-edge.

```
x : a {print('a');} b     {print('b')};
```

Stack machines, BUP, and SDD (|| is sequencing op):

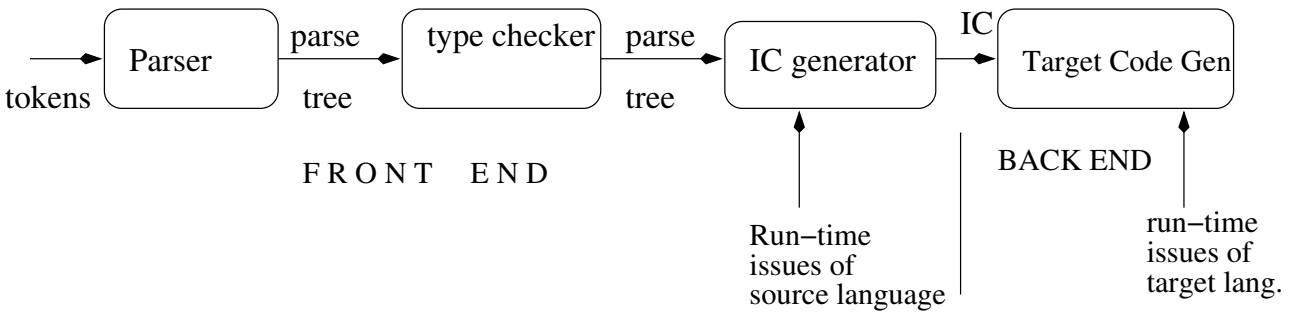
| Form | Meaning |
|---|---|
| ID := E | E.code\|\|lvalue ID\|\|move |
| E -> E + T | E0.code <- E1.code\|\|T.code\|\|add |
| E -> E - T | E0.code <- E1.code\|\|T.code\|\|sub |
| E -> T | E.code <- T.code |
| T -> (E) | T.code <- E.code |
| T -> id | T.code <- rvalue id |
| T -> num | T.code <- push num |

# My advice

- SDD requires a more detailed design in the beginning.
- It pays off in the end:

  The idea of form-meaning correspondence by a rule is re-usable in any computational process.
- It promotes compositionality and transparent computation.
- This is called *grammatical inference*, by De la Higuera (2010); Bozşahin (2018).

# Intermediate Code



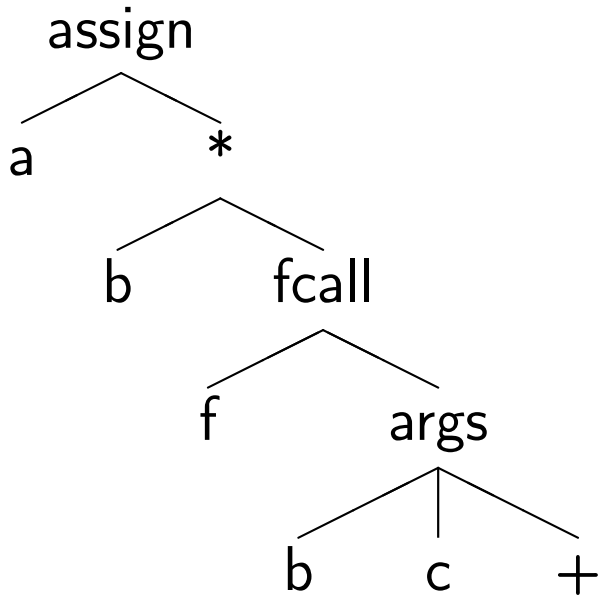

- IC Design Issues:
  What kind of language
  Storage organization for symbols and flow of control
  IC templates for source language constructs

- IC Design
- Closer in spirit to source language execution paradigm but simplified or lower level
- procedural langs: a high-level general purpose assembler
- functional langs: a high-level function description or manipulation language (e.g., $\lambda$-calculus)

- Intermediate representation for Imperative languages
  1. syntax trees (high level rep./no storage concerns)
  2. postfix rep. (linearized syntax tree)
  3. TAC (three address code) : (low level rep./storage for symbols)

syntax tree vs. postfix

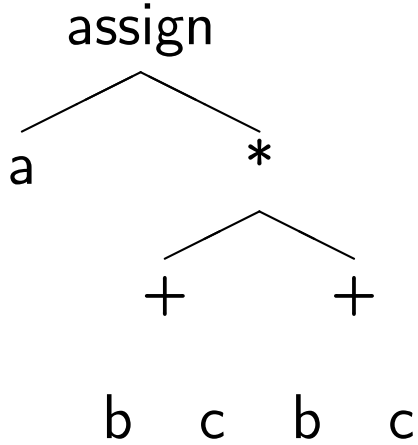
```
a := b * f(B,c,e+d);
```




```
                    e  d
a b f b c e d + 3 args fcall * assign
```

Hard to show common subexpressions in postfix

`a:= (b+c)*(b+c)`



Three-address Code (TAC) for `a:=b*f(b,c,e+d)`

```
param b
param c
t1 := e+d
param t1
fcall f,3
return t2
t3 := b + t2
mv t3 a
```

- TAC instruction set
  Some are quite high level; very few machines have actual counterparts of these instructions directly implemented
  1. assignment `(label:)  x := y op z`
  2. unconditional branch: `goto label`
  3. conditional branch: `bz x L   bnz x L`
  4. `param x`
     `call f,n`
     `return x`
  5. indexed expr.: `x:= y[i]` or `x[i]:=y`
     `x:= y[i]`+b needs 4 addresses (not TAC)
  6. reference: `x:= &y`
  7. dereference: `x:= *y`

- Small IC instruction set simplifies target code generation but produces long sequence of TC instructions
- Large sets may not be easily portable to all architectures (like 4–7)
- TAC generation for
  SL declarations
  IC declarations (temporary storage)
  SL expressions
  SL instructions

- Syntax-directed definition for translating large expressions to TAC

  Conventions: IC function's name indicates arity as well

```
3ac(op,x,y,z)  x:= y op z
2ac(op,x,y)
1ac(op,x)   eg.  param x
2copy(x,y)  x:=y
```

Every grammar symbol has two attributes:
`X.code` (code segment for X);
`X.place` (value holder for X)

`x:= a*b+c` translates to

```
t1 := a*b
t2 := t1 + c
x := t2
```

How to generate new symbols $t_i$?

S -> id := E {S.code= E.code ||
2copy(id.place,E.place)}

E -> E + E { E.place=newtemp();
E.code = E1.code || E2.code ||
3ac(add,E.place,E1.place,E2.place)}

E -> E * E

E -> -E {E.place=newtemp();
E.code=E1.code ||
2ac(uminus,E.place,E1.place)}

E -> id { E.place= id.place; E.code= nil}

e.g., a:=b+e*-c

# IC Templates for SL instructions

- Source Language Instructions to IC instructions: Since IC is lower-level than SL, several IC instructions are needed to translate a single SL instruction.

  The main idea is to preserve meaning in translation. The set of IC instructions *for* a SL instruction must have the same meaning ( must do the same thing computationally).

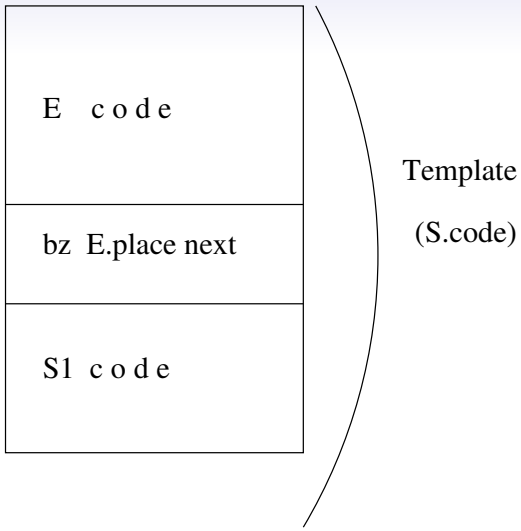
- Since we assume 1-1 correspondence of form and meaning in SL (after all, every syntactic construction is supposed to lend itself to one computation), we can define this correspodence in the form of a *template*.

  A template is a sequence of skeletal IC instructions. Details in the template to be filled in by SL 'contents'

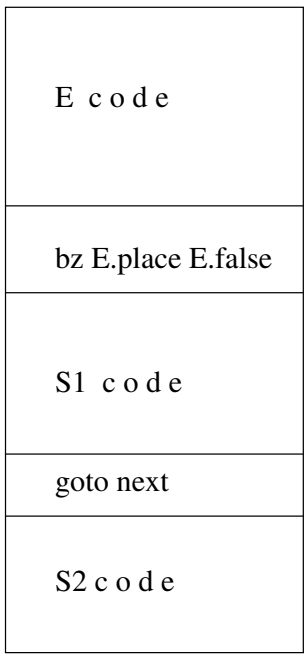
In other words, templates are semantic in nature; they reflect the IC counterparts of SL instructions. But since instructions make reference to data and variables, these are missing from the template.

- IC templates for
  - assignment
  - Explicit (syntactic) constructions for flow of control: if/while/for
  - Implicit flow of control: function call (needs run-time organization)

S –> if E then S;




S –> if E then S
else S;



- The problem with these templates is that a forward label is generated *before* the next instruction which gets the label is generated. Since this rule cannot know the next instruction (if any), but needs the address of it, it can only specify the label.

eg.
```
if E1 then S1
    else S2;
if E2 then S3;
```

- Solutions:

  1. Build a syntax tree and make 2 passes over it. The second pass is top-down to fill in the labels. Uses inherited attributes.

  2. *Backpatching*: Generate branching instructions without labels. Then, when the statement with the label is generated, fill in the labels. This needs to introduce null productions just to mark points of label manipulation.

  3. If multiple labels per statement is allowed, we can generate labels for subsequent statements before they are generated. If the subsequent statement has its own label, there will be multiple labels.

- In a modern VM, 3 is easiest (including MIPS, JVM, LLVM/IR)

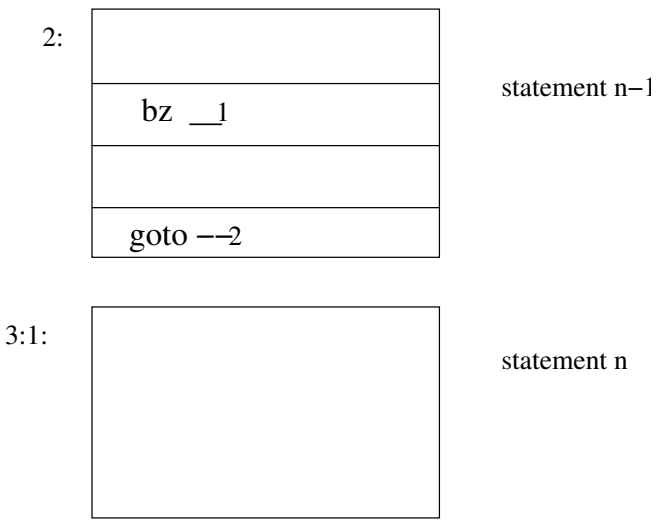
A set of labels for the *same* stament forms an equivalence class, e.g.,

```
L1:L2:L3:   a:= b;
```

We can do a later pass over these equivalence classes and uniquely name them; this will give one label per statement. This is equivalent to backpatching in 2 passes.

Backpatching:

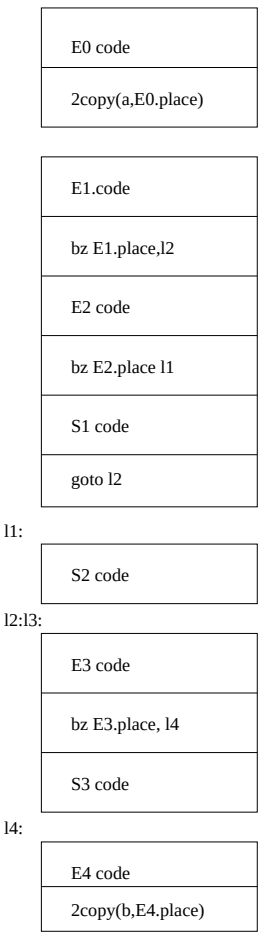
```
S -> if E then X S
     else S;
S -> while X E do X S;
X -> ε  {do something about list of labels}
```



Multiple labels:

```
S -> id := E ;        { S.code=E.code ||
                        2copy(id.place,E.place)}

S -> if E then S;     { t=newlabel();
                        S.code=E.code ||
                        bz E.place, t ||
                        S.code ||
                        label(t)}

S -> if E then S      { t1=newlabel();
  else S;             t2=newlabel();
                        S.code=E.code ||
                        bz E.place, t1 ||
                        S1.code ||
                        goto t2 ||
                        label(t1) ||
                        S2.code ||
                        label(t2)}
```

```
e.g. a := E0;
 if E1 then if E2 then S1 else S2;
 if E3 then S3;
 b:=E4;
```



- Synthesized attributes fit nicely with bottom-up parsing
- and inherited attributes with top-down
- SYNTHESIZED ATTRIBUTES IN A TOP-DOWN PARSER

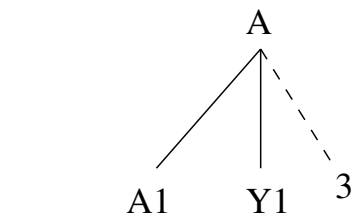
$$A \to AY \quad \{A_0.a = g(A_1.a, Y.y)\}$$
$$A \to X \quad \{A.a = f(X.x)\}$$

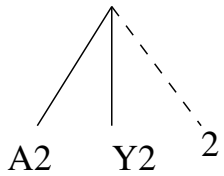
- after removing left recursion:
  $A \to X\{A.a = f(X.x)\}R$
  $X$ must pass on it's attributes to $R$

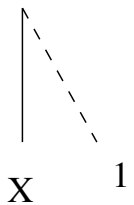
  Two sets of attributes for dummy symbol $R$: $R_i$: inherited.
  $R_S$: synthesized

$$A \rightarrow X \{R.i = f(X.x)\}\ R\ \{A.a = R.s\}$$
$$R \rightarrow Y \{R_1.i = g(R_0.i, Y.y)\}\ R_1\ \{R_0.s = R_1.s\}$$
$$R \rightarrow \varepsilon \{R.s = R.i\}$$

# Attributes are evaluated in the same order

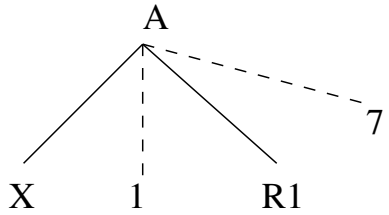


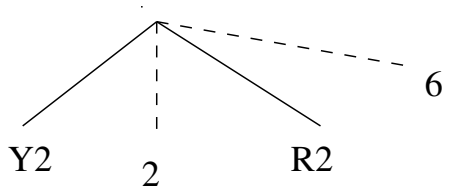



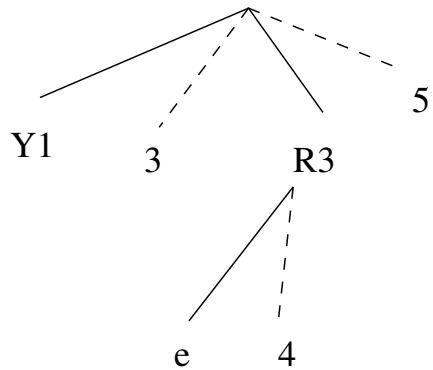




order: 1–2–3

A2.a=f(X.x)
A1.a=g(A2.a,Y2.y)
A.a=g(A1.a,Y1.y)




order: 1–2–3–4–5–6–7

R1.i=f(X.x)
R2.i=g(R1.i,Y2.y)
R3.i=g(R2.i,Y1.y)
R3.s=R3.i
R2.s=R3.s
R1.s=R2.s
A.a=R1.s

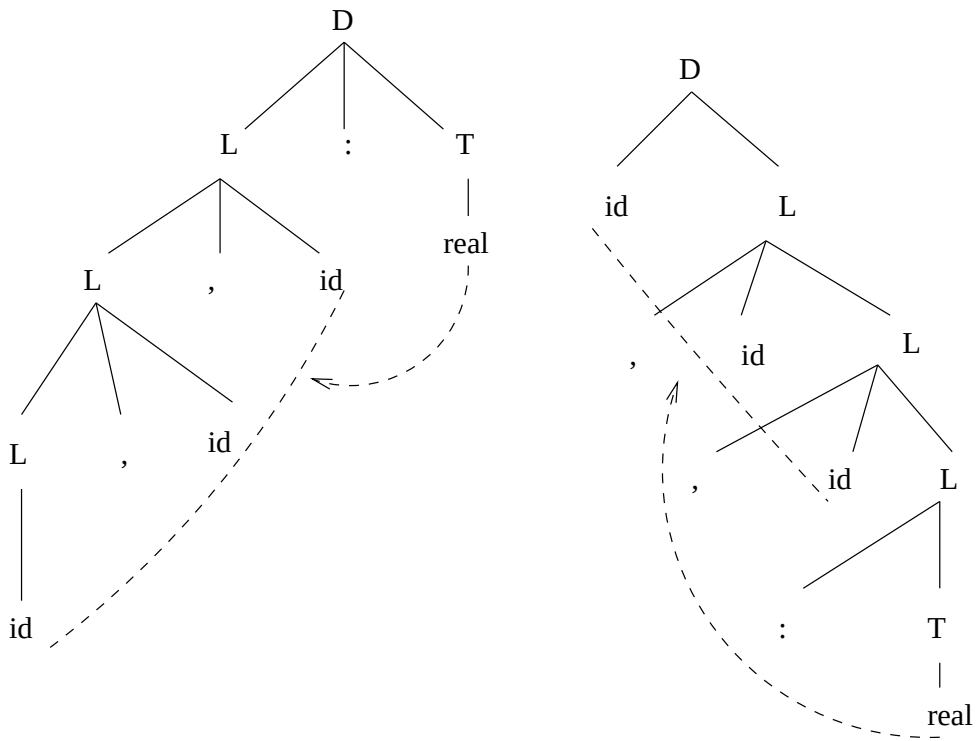
- Inherited attributes in a bottom-up parser

  Bottom-up parsers are natural fits for synthesis; as in left tree.

  `D` → `L : T`

  `T` → `integer | real | char`

  `L` → `L , id | id`

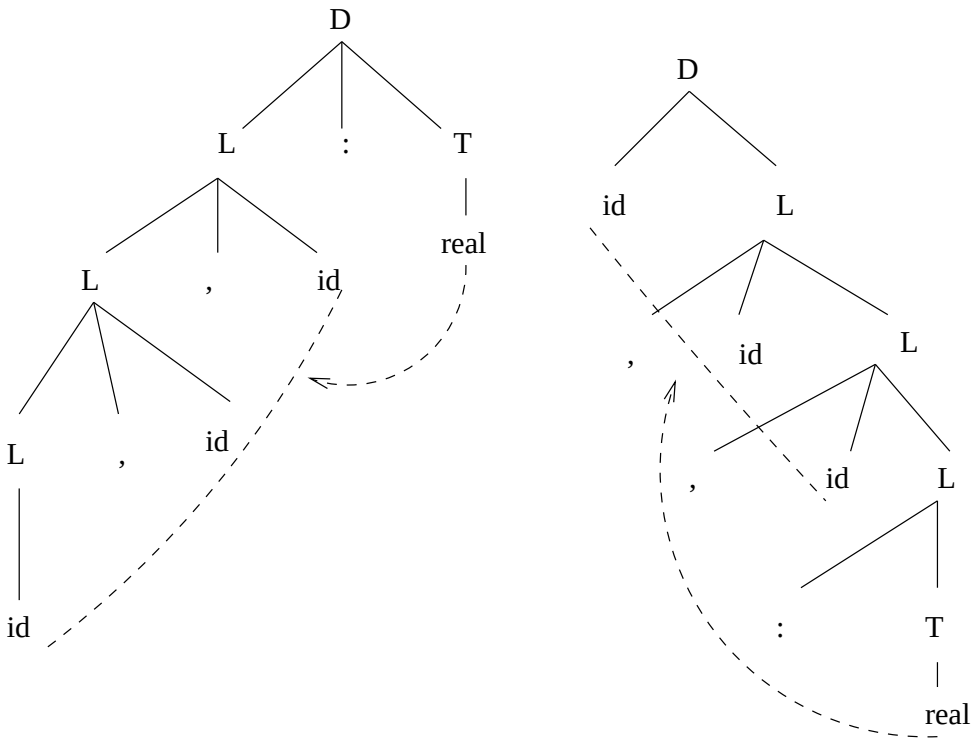


If we need inheritance, the right tree can be realized with:

D → id L

T → integer | real | char

L → , id L | : T

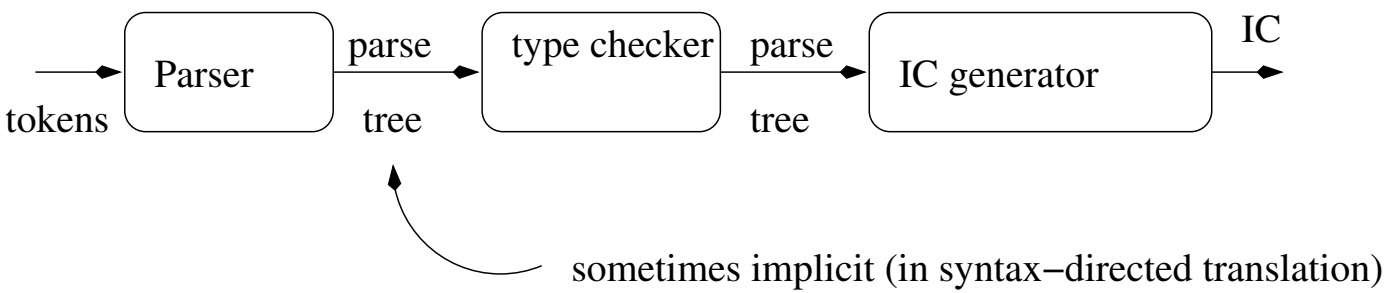


- Semantics checks:
  Type consistency and/or equivalence
  Flow of control (e.g., break must be in a loop)
  Uniqueness checks (e.g., declare once)
  Declaration checks (e.g., declare before use)

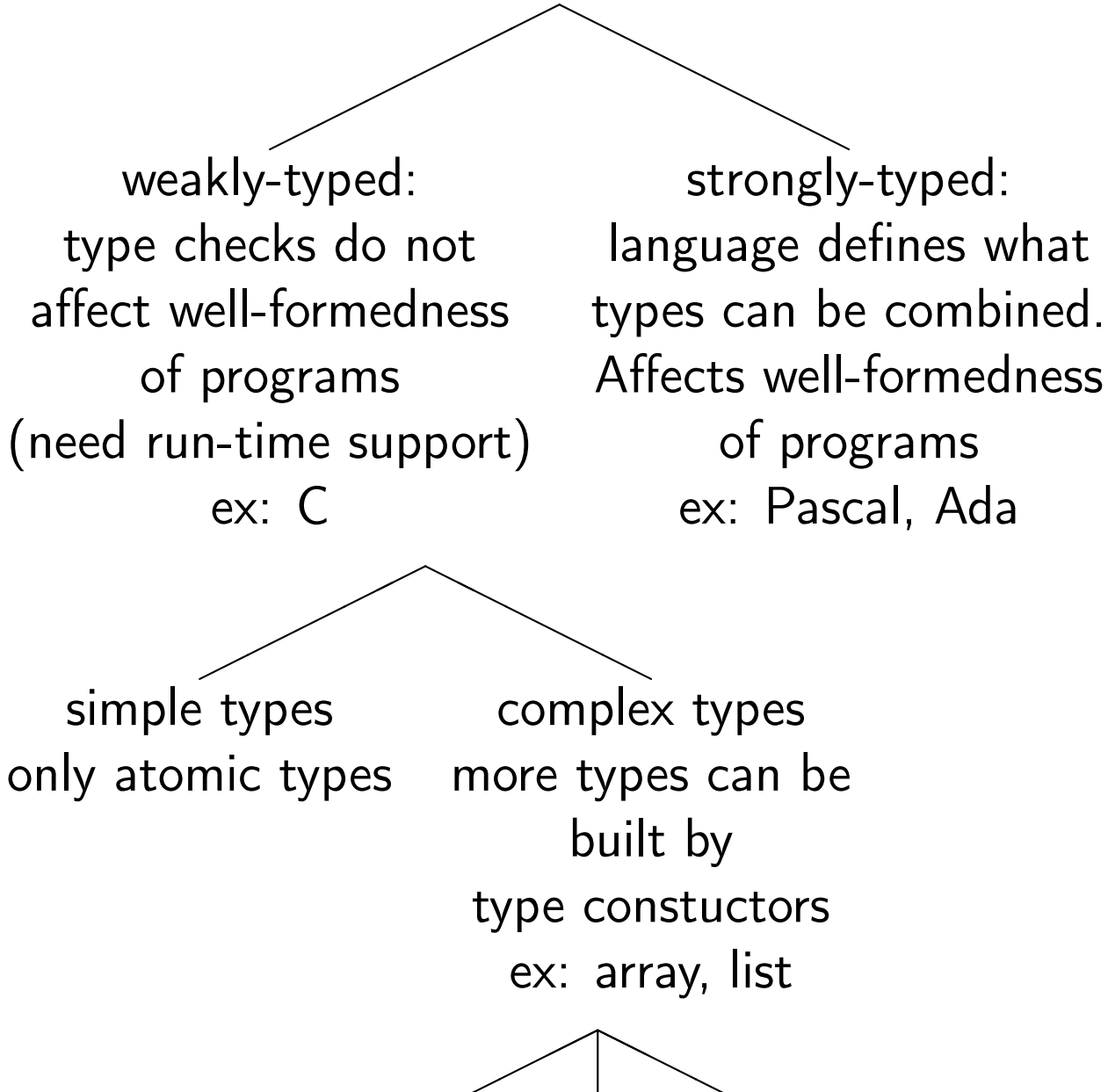
type checking

static (compile-time) check types in source language

dynamic (run-time) check types in target language



## Type systems for programming languages



TYPE EXPRESSIONS:

*basic*: char, boolean..

*named type*: zoint= boolean

*type constructor*: a function of *type* $\rightarrow$ *type*

*arrays*: $I \times T \rightarrow arr(I, T)$

*products*: if $T_1$ and $T_2$ are types, so is $T_1 \times T_2$

ML ex: `val(a,b)=(0,2.5);`

C ex: `struct {char c1; int c2}`

*records*: named products

*pointers*: if $T_1$ is a type, so is $p(T_1)$

*functions*: if $T_1$ and $T_2$ are types,

$f(T_1) : T_2$ is of type $T_1 \rightarrow T_2$

*type variables*: if $x$ is a type,

$type\_expr(x)$ is also a type

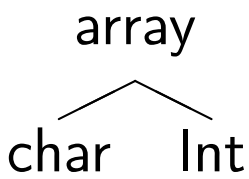
ex: type-checking of simple expressions

```
E -> E + E  {E.type= if E1.type=E2.type then E1.type
             else error}

E -> E mod E {E.type= if E2.type=int and E1.type=int
                 then E1.type else error}

E -> NUM     {E.type=NUM.type}

E -> (E)     {E.type=E1.type}

E -> F(E)    {E.type=if E1.type=t1 and F.type=t1->t2
                 then t2 else error}
```

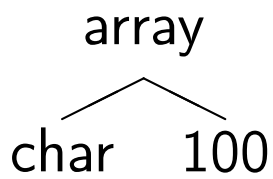
If the language allows complex types, a simple type attribute in the symbol table is not enough

Type trees.
root: type constructor
children: argument types

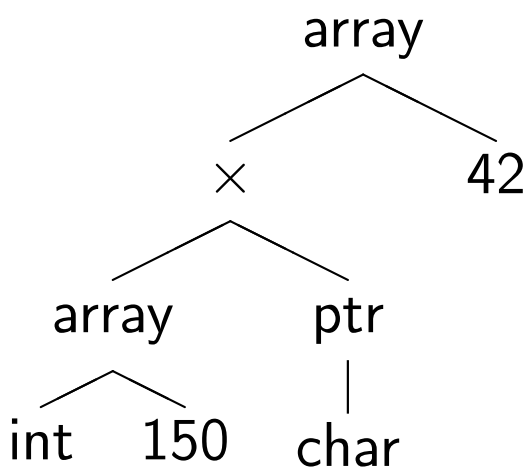
```
char x[100]
```




or

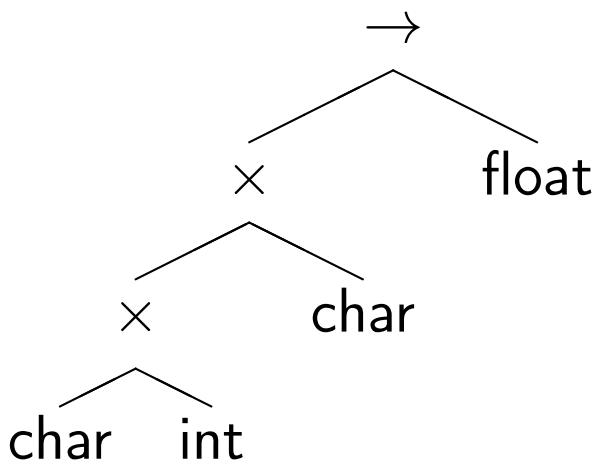



```
struct {int i[150];
        char *c;
       } y[42];
```



For a function

```
float f(x,y,z)
  char x,z;
  int y;
  {....}
```




- The symbol table must reflect the structure of the type

- Complex types constructed with type constructors require some notion of *type equivalence*:

  *named equiv.*: treat named types as simple types; just check that the types have the same name

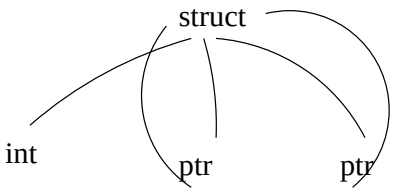
```
var a,b:  array[1..10] of char;
c:  array[1..10] of char;
```

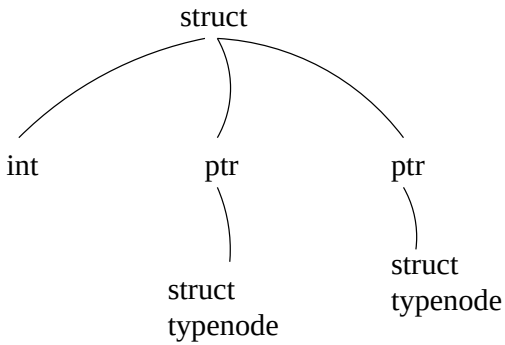
  *structural equiv.*: replace named types with their definitions and recursively check the type trees

```
typedef int[100] list;
typedef int[100] vector;
..
list a;   vector b;
```

What about recursive types?

```
struct typenode
  { int comp0;
   struct typenode *comp1;
   struct typenode *comp2;
  } ;
```




either mark the traversal of type tree or use structural equiv. everywhere except recursive types (C's solution)

# Some type check problems

- Declare variables before use (what about type info?)
  lex analyzer inserts IDs into symbol table with type `T-UNDEF`
  In *use* of a var, if type is not known, issue error

example in Yacc

```
e :  e '+' e { ...  };
  | t { ...  };
t :  ID { struct symtab *s=search-sym($1);
    if (s->type != T-VAR || s->blockno !=
currblock) error(..)};
```

- Uniqueness check (unique types)

```
decl :  vars ':'  type { ...  }
vars :  vars ',' id { check if same ID with
current block no is in ST};
```

- What if procedure nesting is allowed (finite or indefinite length)?

- explicit type conversion : language provides type constructors for type casting

```
float a; int i;
a = (float) i;
```

- implicit type conversion: type checker must perform type coercion

- overloading: syntactically same operator denotes different operations semantically

```
e :  e '+' e { either integer/real addition or set
union in Pascal};
```

- Type polymorphism: use type variables and type inference

- Dynamic aspects of language processing need to be dealt with during execution: local names, implicit flow of control, storage organization, etc.

  1. storage organization (run-time stack management for flow of control)

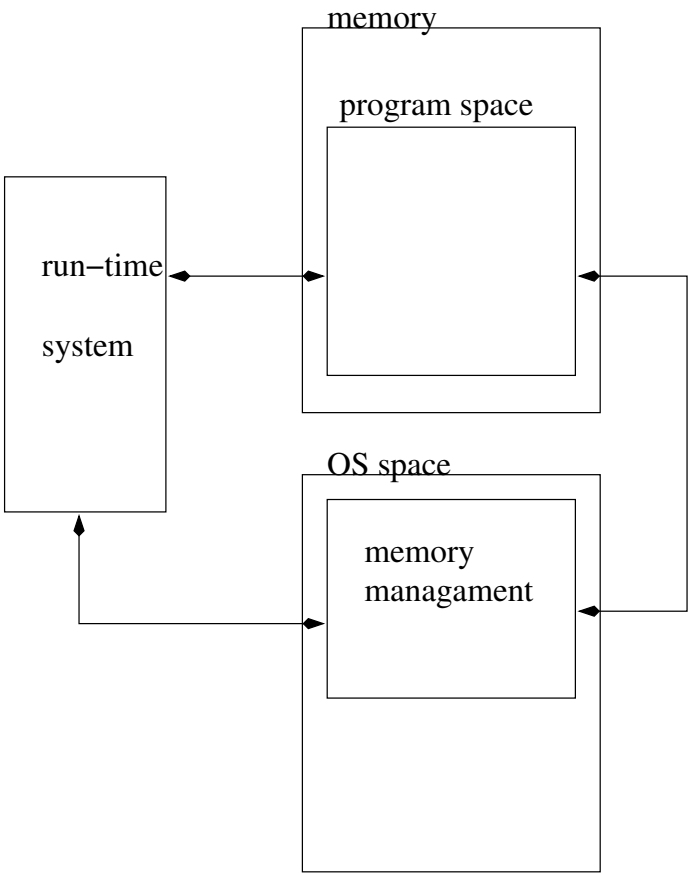
  2. storage allocation (static or dynamic layout of program/data space)

  3. Run-time counterparts of SL constructs (nesting, block structure)

4. Access to names (local/nonlocal, lexical/dynamic scoping)

5. Parameter passing

6. symbol-table access and management

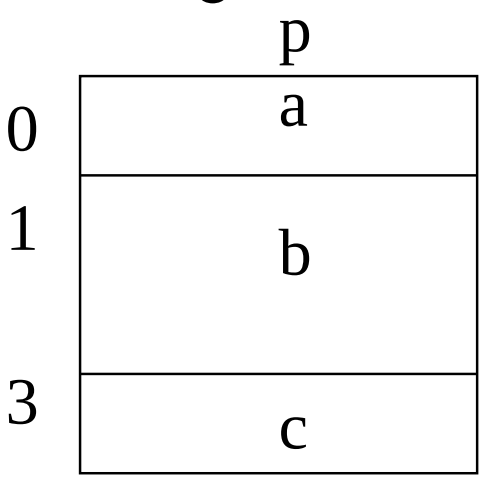


- Translating IC declarations to storage layout within a block:

  Relative addressing within a block
  - base address of the block
  - offset from base
  - amount of space (can be deduced from type of the symbol)

```
ex: struct symtab {
      char *name;
      struct typenode *type;
      int blockno;
      int addr;};

func p ();
var a:  int; b:real; c:int
```

assuming one word for integers and two words for reals:



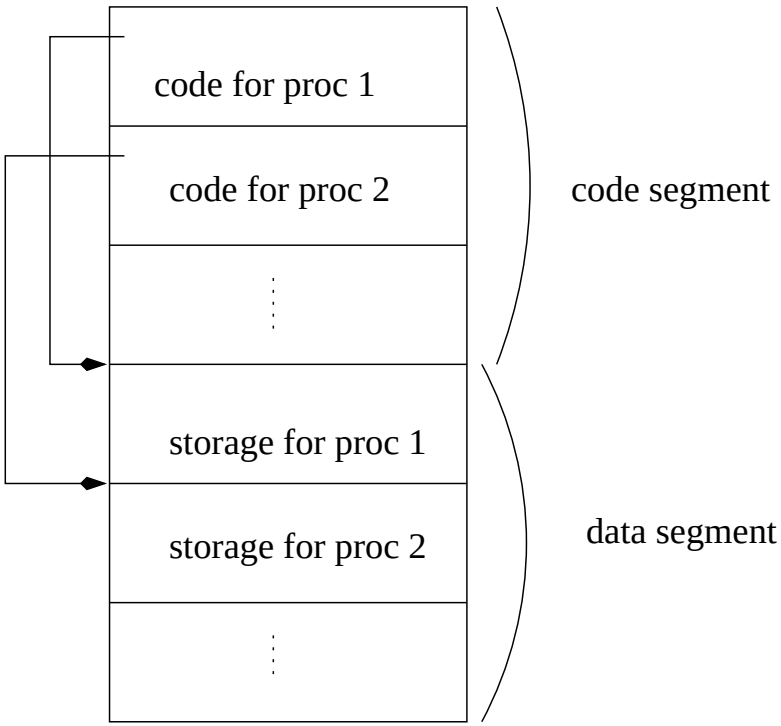
syntax-directed way of calculating storage:

```
P -> proc ID { blockno++; offset=0} (Params) Decl Body
{ ...}
Decl -> Decl ; D | D { ...  }
D -> ID : T { enter(ID.val,T.type,blockno,offset)
              offset=offset+T.width}
T -> int { T.type=INT; T.width=2}
T -> array [num] of T { T.type=array(num.val,T1.type)
              T.width=num.val * T1.width}
```

Records can be treated like procedures as far as storage layout is concerned.

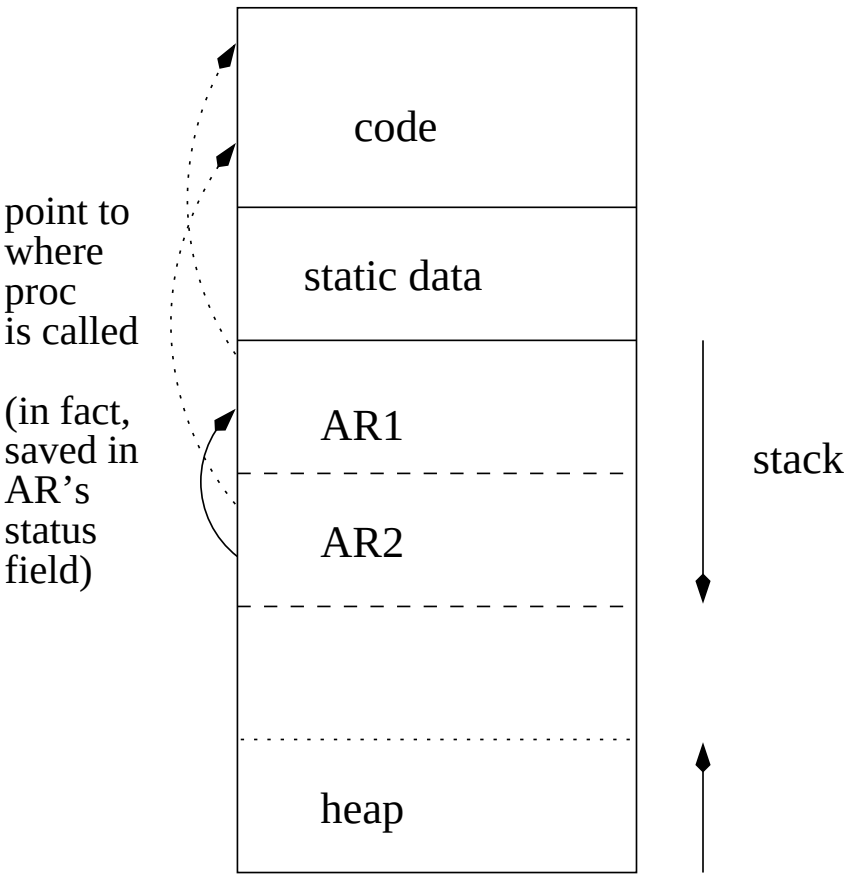
- Run-time management of storage
  - lifetime of variables
  - parameter passing
  - function call/return
  - dynamic storage

# Program space when there is no recursion




FORTRAN's storage allocation
(predefined storage for every proc)

Recursion: keep a stack of Activation Records (AR)
Program space is 1) static, and 2) dynamic in two directions.

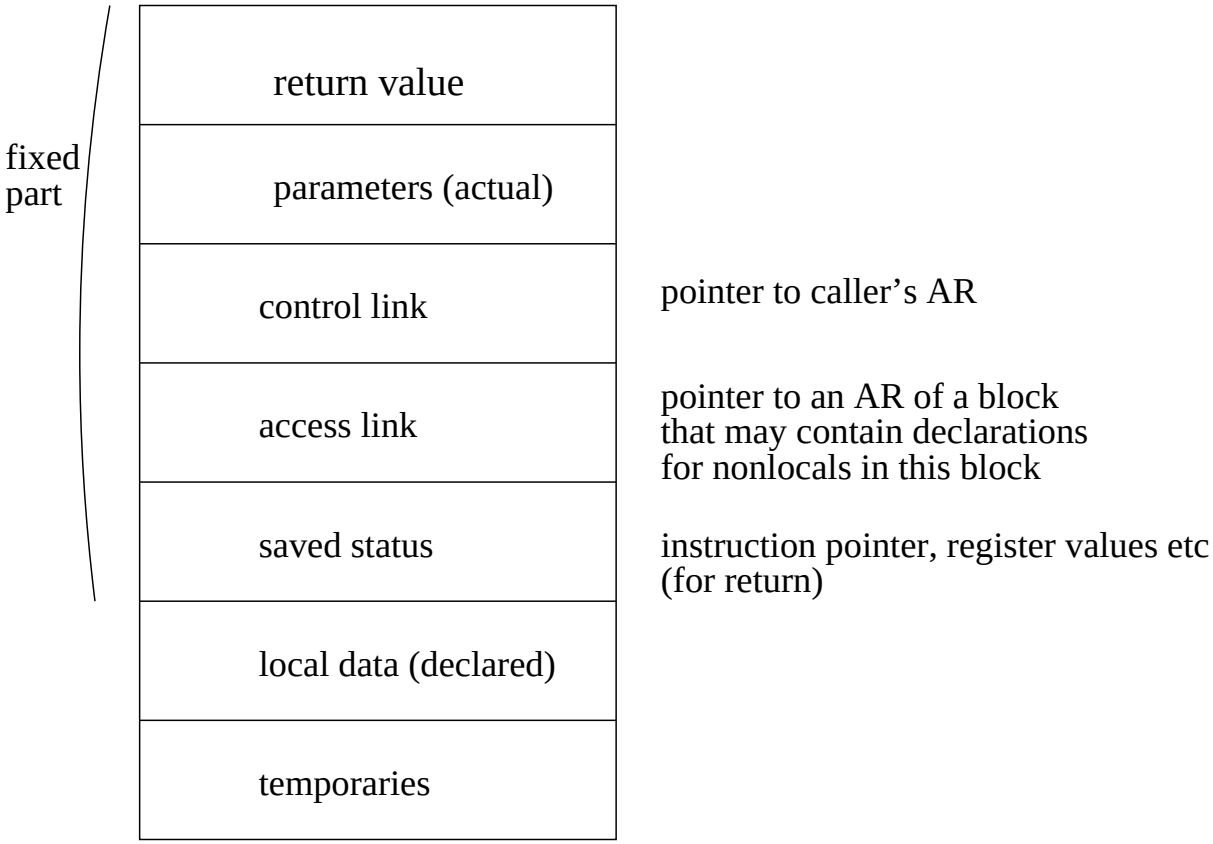


- The structure of AR

  fixed part: things that can be located by the caller without having to know the internal structure of the callee.

  block-dependent part: procedural-specific stuff (local vars etc.)

- Why does the caller need the fixed part? Some things must be put in there *before* control is transferred to the callee, and some things must be accessible *after* the callee returns.




the structure of a single activation record

- Parameter passing mechanisms:
- Formal arguments and actual arguments must be associated during run-time to do parameter passing.
- Declaration of formal arguments allow space allocation of definite size in the callee's AR

1. call-by-value: Caller evaluates and puts values of arguments in callee's AR

2. call-by-reference: caller puts addresses of actual arguments in callee's AR (r-value of formal parameter is l-value of actual parameter)
what if actual argument is an expression with no l-value?

3. copy-restore (value result, mixture of 1 & 2): caller evaluates actual parameters and passes r-values to callee. Caller saves their l-values. Upon return, it restores the r-value of formal parameters to l-values of actuals.

4. call-by-name: literally substitute actuals for the formals. Local names in callee that clash with names in actuals are renamed.

l-value of actuals are passed in 2 and 3; r-value in 1, and the text itself in 4.

ex: a hypothetical language whose parameter passing is left unspecified

```
proc swap(x,y)
    temp:=x;
    x:= y;
    y:= temp;
endproc;
```

assume we make the call `swap(i,a[i])`

in call-by-value, no change in actuals

in call-by-reference, l-values of `i, a[i]` are passed on. `x,y` indirectly refers to the same l-value.

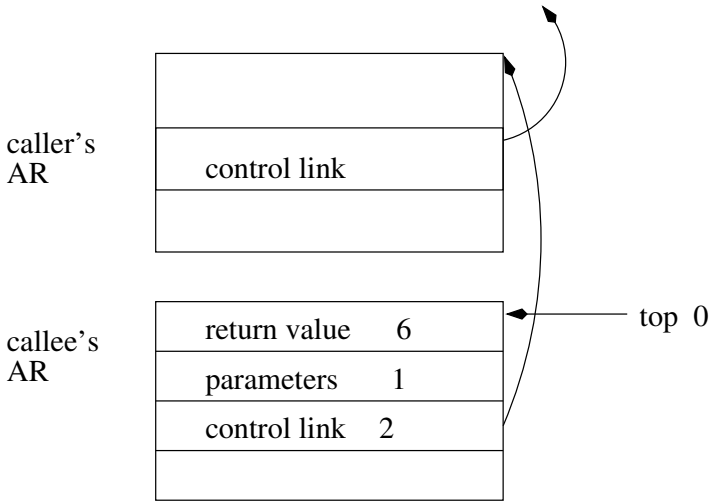
in copy-restore, r-values of `i, a[i]` are passed on, but on return, `x,y`'s r-values are put in l-values of `i, a[i]`

in call-by-name, i is substituted for x, and a[i] for y:

```
temp := i;
i:= a[i];
a[i]:= temp
```

if $I_0$ is initial value of `i`, `a[a[`$I_0$`]]` becomes $I_0$, not `a[`$I_0$`]`.

- Argument passing shows that there are some things that the caller must pass on to the callee before the call,
- and return values of the callee must be accessible to the caller after the call is over.
- How is this done?




CALLER
−−−−−−−−

0. push control stack (enw entry for callee's AR)
1. evaluate actual parameters and put them in callee's AR (they are a fixed distance away from caller's AR)
2. set the control link of the CALLEE to point to the CALLER (it is also fixed distance away from caller's AR)
3. save next intsruction in CALLEE's machine status (also fixed distance away)
9. caller retrieves return value (although stack is popped, it is still fixed distance away from the caller's AR)

CALLEE
−−−−−−−−

4. save machine status
5. initialize and execute
6.. put return value in place
7. restore machine status and and reset instruction pointer to caller's next stmt (was saved as part of machine status in 3)
8. pop control stack

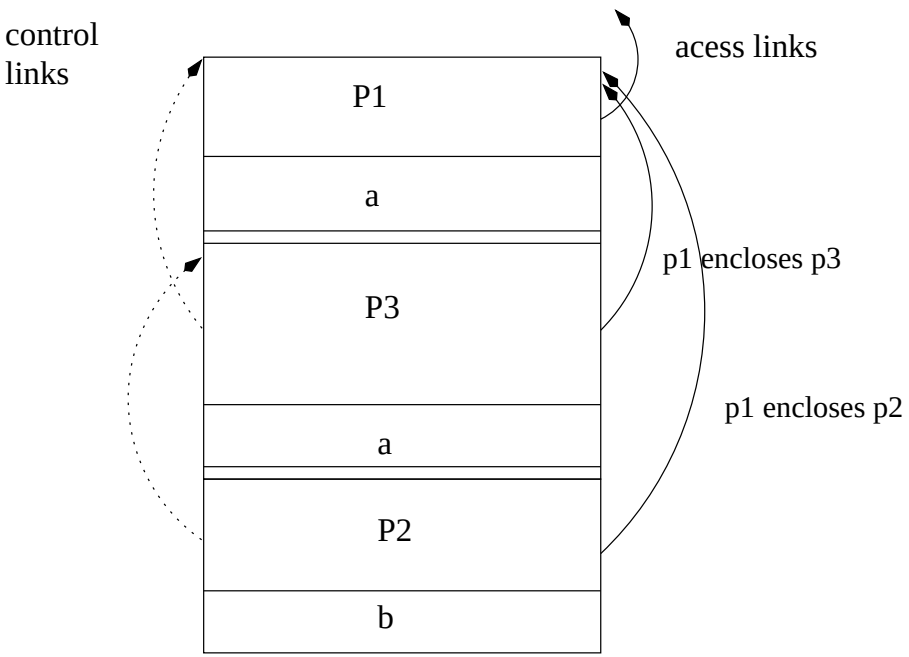
- Why do we need Access links in addition to control links?

  In *dynamic scoping*, the calling sequence determines the environment. Since calling sequence is shown by control links in ARs, there is no need for extra links.

  In *lexical scoping*, textual enclosures in the program determine the environment. But this enclosure is not necessarily the same as calling sequence (hence we need extra information)

```
proc p1 ()
        var a;
        ...
        proc p2 ()
                var b;
                begin
                        x := a +b ;
                end
        proc p3()
                var a;
                begin
                        p2();
                end
```

- Access link: if $q$ is nested immediately in $p$ (textual enclosure), $q$'s access link points to the *most recent activation* of $p$ (and there must be one, why?)




Looking for globals in lexical scope: if not in current AR, follow the access link

- Accessing the access links: if a proc at nesting depth $n_p$ refers to a nonlocal a at $n_a$:

- $n_a \leq n_p$ due to lexical scoping. Follow $n_p - n_a$ number of access links to reach the AR of a's definition block. This value can be computed at compile time (for each nonlocal in every block)
  The tuple ($n_p - n_a$,a's offset) defines the address of any nonlocal

- Setting up the access links: assume p calls x:

  if $n_p < n_x$, x is nested in p. Access link of x points to the access link in AR of the caller (second from top in stack)

  if $n_p \geq n_x$, x is not declared within p (due to property of lexical scope). the procedures at nesting depths 1,2, $n_x - 1$ must be the same (again due to lexical scope). $n_p - n_x$ number of links reach the level of x again, hence $n_p - n_x + 1$ links reach the block that most closely encloses p and x. This can be computed at compile time as well.

- If nesting is deep and calling sequences are not hierarchical, this method causes a lot of hopping along access links.
- One alternative is to use *display* mechanism.
- Intel chips have hardware implementation of *displays*.

- displays make use of the fact that, in lexical scope, two procedures at same nesting depth cannot refer to each other's bindings as nonlocal reference. So, it is possible to use only one storage for any nesting depth $n$.

  `d[n]` is an array of AR pointers, it's value is set during calls and returns. `d[i]` points to the AR of the most current activation of active procedure at level i.

  When a new AR for a proc at level i is set up
  1. save the value of `d[i]` in the new AR (later to be restored to this value upon return)
  2. set `d[i]` to point to the new AR

- Why does this scheme work? let's say p calls q:

  - if $n_p < n_q$ then $n_q = n_p + 1$ because otherwise q will not be visible to p. This means that there is no need to change the first $n_p$ entries in the display.

  - if $n_p \geq n_q$ then `d[i]` entries for $i = 1, 2, \ldots, n_q - 1$ are the same. Save `d[`$n_q$`]` and reset.

- Lexical scoping makes most of the procedure interactions predictable at compile time.
- Access to nonlocals can be fixed at compile time too.
- Dynamic scoping puts the burden on run-time (though nonlocal management is trivial thru ARs).
- Dynamic scoping seems better suited to domains of computations in which a procedure is expected to behave differently depending on who calls it (as in some AI applications).

Bozşahin, C. (2018). Computers aren't syntax all the way down or content all the way up. *Minds and Machines 28*(3), 543–567.

De la Higuera, C. (2010). *Grammatical inference: learning automata and grammars*. Cambridge University Press.