CEN G444:
Semantic Analysis
Code Generation

Cem Bozşahin

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

- The most crucial property of code generation: Generate code from RULE USE.
- The assumption here is that THE PROGRAMMER expresses her INTENTIONS in the SOURCE language's SYNTAX.
- Generating code compositionally is the surest way to avoid second-guessing the programmer.

- But what is 'RULE USE' ?
  - In top-down parsing, we have not seen the RHS yet when we replace the leftmost variable. So we must EXPECT code to arise as we recognize the RHS.
  - In bottom-up parsing, we have already seen the RHS. We must bundle the info in it to be passed on the variable on the left.
  - If you generate from an AST, they are respectively top-down and bottom-up info passing in the construction of the AST.
- Attributes: compositional information passing during parsing.

Attribute Grammar

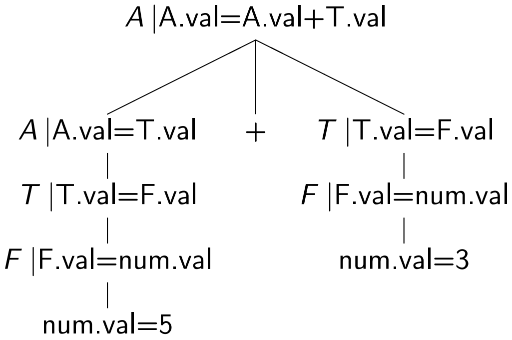
- Attribute/feature: a value associated with a grammar variable at parse time.
- An attribute grammar is a CFG in which the grammar symbols have attributes associated with them.
- This actually *extends* the power beyond context-freeness, but the form of the grammar is similar to CFGs in the sense that 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}
```

- We will also see syntax-directed code generation for this fragment.

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 $Y_i.a, i \neq k$

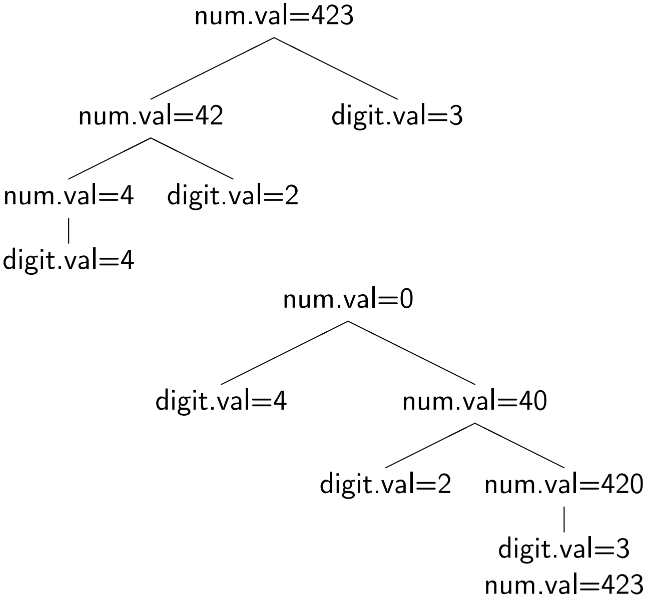
ex: synthesized vs. inherited derivation of numbers

```
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
```

423=


num.val=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)
```

Syntax-directed AST generation.

```
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)
```

Order of evaluation
of semantic actions

Translation scheme | Syntax-directed definition | tree parsing

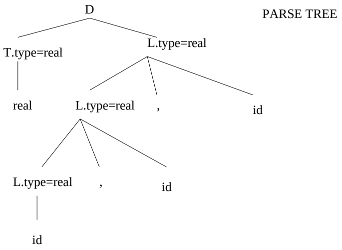
- Translation scheme: Order of semantic actions are explicitly shown within the RHS of a rule; their evaluation depends on the parsing strategy.
- Tree parsing: our rules do less work than they could.

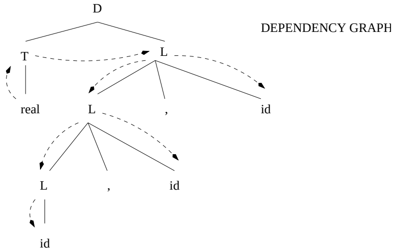
- SDD: Syntax-directed definition (official use), Syntax-directed derivation of semantics (my use).

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

- Parsing and dependency passing may diverge. 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}
```


D
PARSE TREE
T.type=real
L.type=real
real
L.type=real
,
id
L.type=real
,
id
id


D
DEPENDENCY GRAPH
T
L
real
L
,
id
L
,
id
id

- 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')};

Yacc converts this to:

x: a $ACT b    {print('b')};

$ACT:   {print('a');};
```

But this will cause shift/red conflicts.
what about

```
x : a {$3.f=$1.f} b     {print('b')};
```

Here is same problem with the oblivious method (i.e.parsing and semantic action are one to one):

```
D -> T L {assign(T.type, L.nl)}

T -> int {T.type=integer}

T -> real {T.type=real}

L -> L , id {L0.nl=L1.nl+id.name}

L -> id     {L.nl=id.name}
```

- Simple code generation by attributes.
- Assuming: SAM as target arch, BUP, and SDD (|| is the concatenation op in the Dragon book)

```
Form                Meaning

S -> ID := E    S.code  <- 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'
```

Recall SAM from the first weeks of the course:

fetch values of IDs from memory to stack (`rvalue x`)

put values on stack (`push v`)

put address of ID on stack (`lvalue x`)

pop one address and one value, store value in address (`move`)

operators (`add, sub` etc.). Pop enough operands and do op.

- It is a zero-address machine.

'code' attribute accumulating code for a := b-3

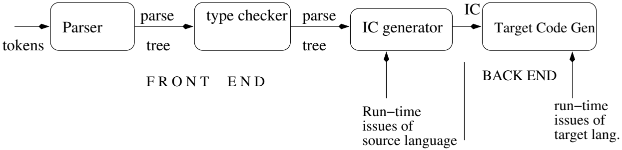
(shift op is skipped to save slide space)

```
stack       input    rule used     code attr
-------     ------   ---------     ----------
$           a:=b-3
$ID:=
$ID:=ID              T <- id       T.code='rvalue b'

$ID:=T               E <- T        E.code=T.code='rvalue b'
$ID:=E
$ID:=E-num           T <- num      T.code='push 3'

$ID:=E-T             E <- E-T      E0.code=E1.code||T.code||'sub'=
                                      'rvalue b; push 3; sub'

$ID:=E               S <- ID:=E    S.code=E.code||'lvalue a'||'move'=
                                      'rvalue b; push 3; sub; lvalue a; move'
$S
```

My advice

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



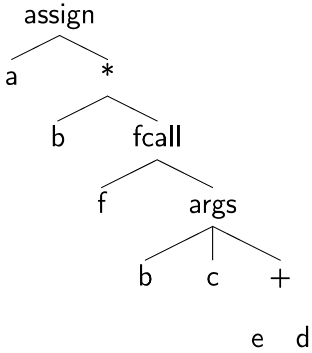
- IC Design Issues:
  What kind of language
  Storage organization for symbols and flow of control
  IC templates for source language constructs

- 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., λ-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

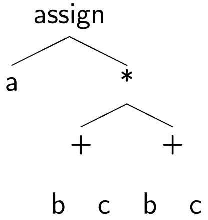
`a := b * f(B,c,e+d);`



`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).

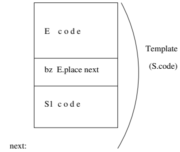
- Since we assume 1-1 correspondence of form and meaning in SL (after all, every syntactic construction is supposed to lend itself to one computation), we can define this correspodence in the form of a *template*.

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

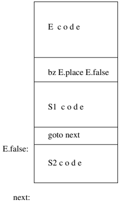
In other words, templates are semantic in nature; they reflect the IC counterparts of SL instructions. But since instructions make reference to data and variables, these are missing from the template.

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

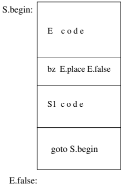
S –> if E then S;



S –> if E then S
else S;



S –> while E do; S



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;
```

```
    E0 code
    2copy(a,E0.place)

    E1.code
    bz E1.place,l2
    E2 code
    bz E2.place l1
    S1 code
    goto l2
l1:
    S2 code
l2:l3:
    E3 code
    bz E3.place, l4
    S3 code
l4:
    E4 code
    2copy(b,E4.place)
```

- The potential 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.,

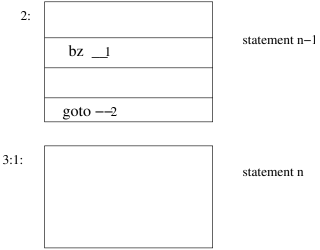
```
L1:L2:L3:  a:= b;
```

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}



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

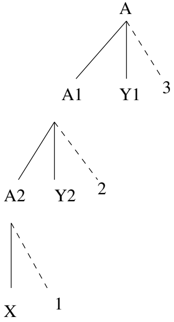
$$
\begin{array}{ll}
A \rightarrow AY & \{A_0.a = g(A_1.a, Y.y)\} \\
A \rightarrow X & \{A.a = f(X.x)\}
\end{array}
$$

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

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

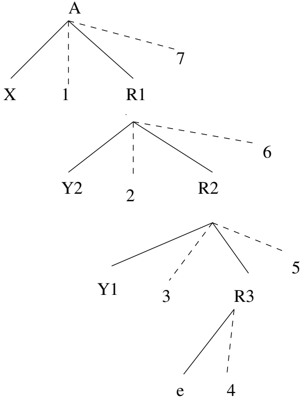
$$A \rightarrow X\{R.i = f(X.x)\}\ R\ \{A.a = R.s\}$$
$$R \rightarrow Y\{R_1.i = g(R_0.i, Y.y)\}\ R_1\ \{R_0.s = R_1.s\}$$
$$R \rightarrow \varepsilon\{R.s = R.i\}$$

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)



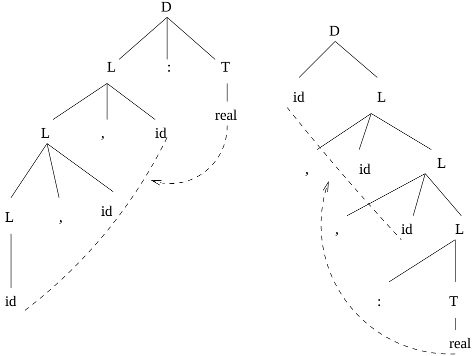
order: 1–2–3–4–5–6–7

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`

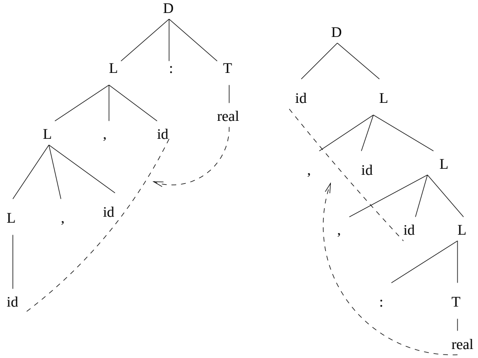


If we need inheritance, the right tree can be realized with:

`D` $\rightarrow$ `id L`

`T` $\rightarrow$ `integer | real | char`

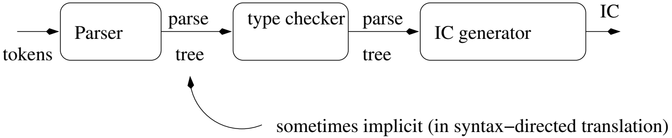
`L` $\rightarrow$ `, id L | : T`



- 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

weakly-typed:
type checks do not
affect well-formedness
of programs
(need run-time support)
ex: C

strongly-typed:
language defines what
types can be combined.
Affects well-formedness
of programs
ex: Pascal, Ada

simple types
only atomic types

complex types
more types can be
built by
type constuctors
ex: array, list

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

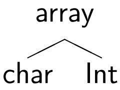
ex: type-checking of simple expressions

```
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}
```

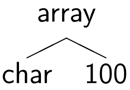
If the language allows complex types, a simple type attribute in the symbol table is not enough

Type trees.
root: type constructor
children: argument types

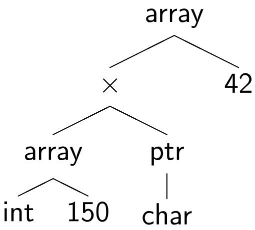
```
char x[100]
```



or

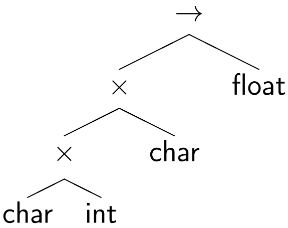



```
struct {int i[150];
        char *c;
       } y[42];
```



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

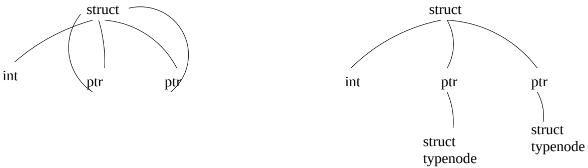
  ```
  var a,b:  array[1..10] of char;
  c:  array[1..10] of char;
  ```

  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

Bozșahin, 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.