Context-sensitive Analysis

Attribute Grammar And Type Checking

Context-Sensitive Analysis

- To understand the input computation, a compiler/interpreter need to discover
  - The types of values stored in each variable
  - The types of argument and return values for each function
  - The representation/interpretation of each value
  - The memory space allocated for each variable
  - The scope and live range of each variable
- Static definition of variables: variable declarations
  - Compilers need properties of variables before translation
  - Use symbol tables to keep track of variable information
- Context-sensitive analysis
  - Determine properties of program constructs
    - E.g., CFG cannot enforce all variables are declared before used

Syntax-Directed Translation

- Compilers translate language constructs
  - Need to keep track of relevant information
    - Attributes: relevant information associated with a construct
  - Attribute grammar (syntax-directed definition)
    - Associate a collection of attributes with each grammar symbol
    - Define actions to evaluate attribute values during parsing

e ::= n | e+e | e–e | e * e | e / e

Attributes for expressions:
    type of value: int, float, double, char, string,…
    type of construct: variable, constant, operations, …
Attributes for constants: values
Attributes for variables: name, scope
Attributes for operations: arity, operands, operator,…

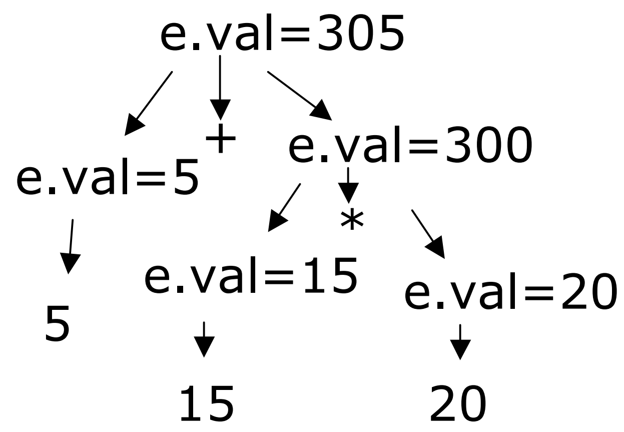
Attribute Grammar

- Associate a set of attributes with each grammar symbol
- Associate a set of semantic rules with each production
  - Specify how to compute attribute values of symbols
- Systematic evaluation of context information through traversal of parse tree (or abstract syntax tree)

e ::= n | e+e | e–e | e * e | e / e

production	Semantic rules
e ::= n	e.val = n.val
e ::= e1 + e2	e.val = e1.val [+] e2.val
e ::= e1 - e2	e.val = e1.val [-] e2.val
e ::= e1 * e2	e.val = e1.val [*] e2.val
e ::= e1 / e2	e.val = e1.val [/] e2.val

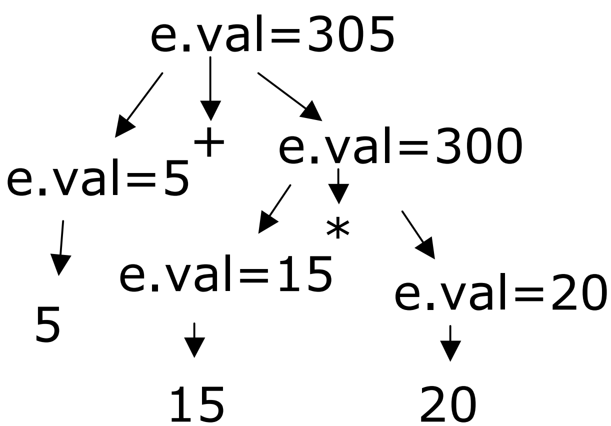
Annotated parse tree for 5+15*20:



Synthesized Attribute Definition

- An attribute is synthesized if in the parse tree,
  - Attributes of parents are determined from those of children
- S-attributed definitions
  - Syntax-directed definitions with only synthesized attributes
  - Can be evaluated through post-order traversal of parse tree

e ::= n | e+e | e–e | e * e | e / e

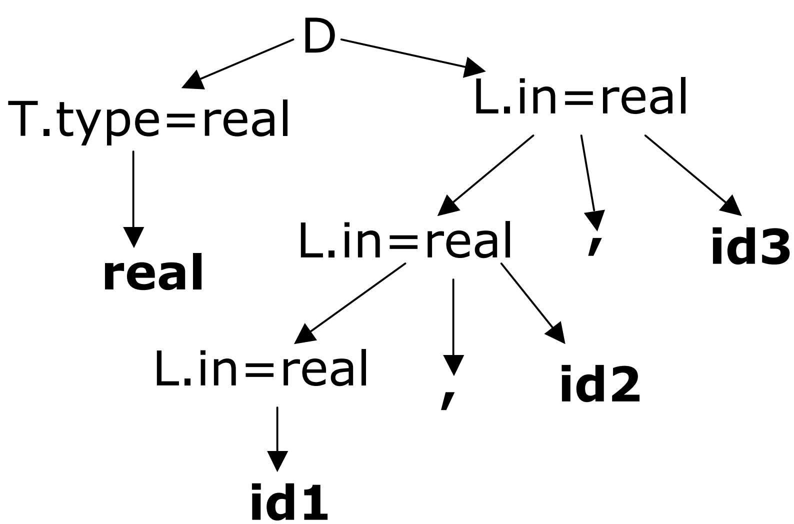


production	Semantic rules
e ::= n	e.val = n.val
e ::= e1 + e2	e.val = e1.val [+] e2.val
e ::= e1 - e2	e.val = e1.val [-] e2.val
e ::= e1 * e2	e.val = e1.val [*] e2.val
e ::= e1 / e2	e.val = e1.val [/] e2.val

Inherited Attribute Definition

- An attribute is inherited if
  - The attribute value of a parse-tree node is determined from attribute values of its parent and siblings

D ::= T L
T ::= int | real
L ::= L , id | id

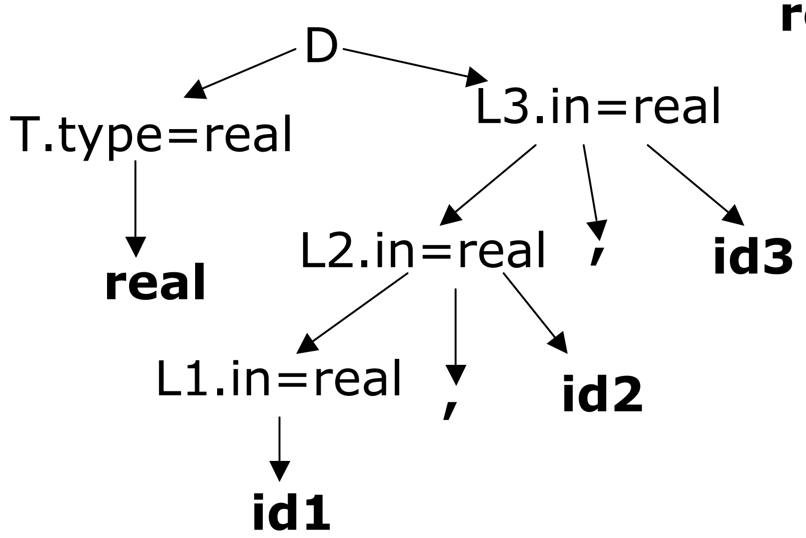


Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

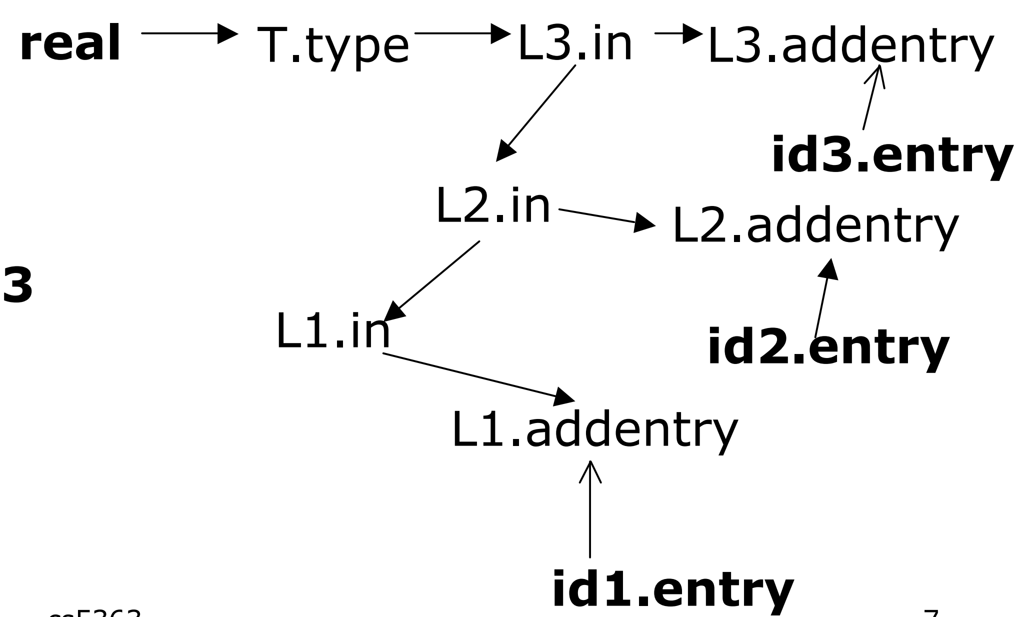
Dependences In Attribute Evaluation

- If value of attribute b depends on attribute c,
  - Value of b must be evaluated after evaluating value of c
  - There is a dependence from c to b

Annotated parse tree:



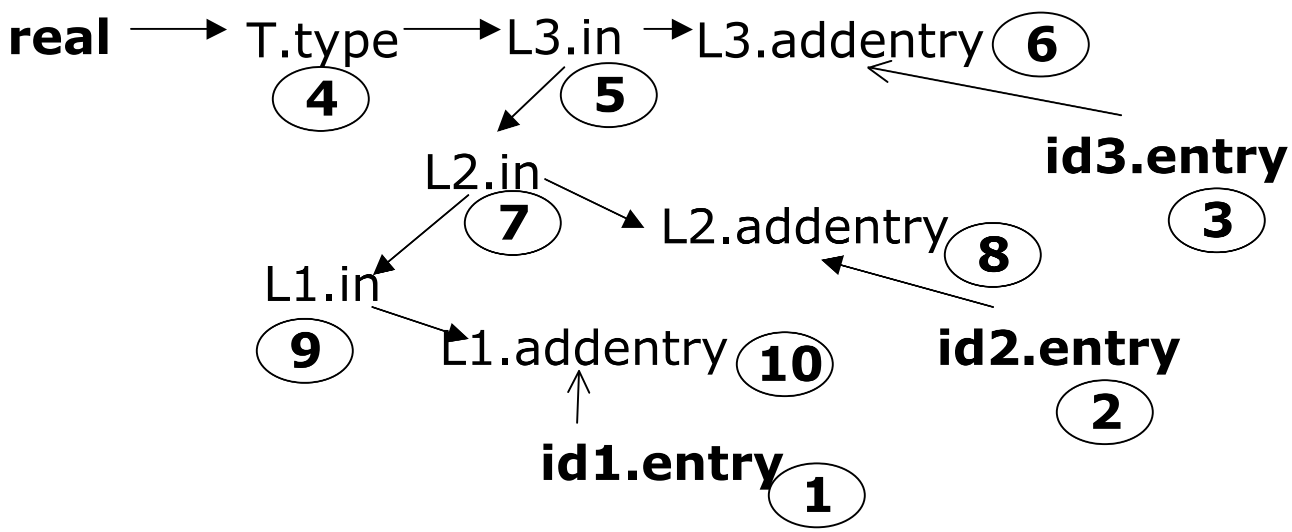
Dependency graph:



Evaluation Order Of Attributes

- Topological order of the dependence graph
  - Edges go from nodes earlier in the ordering to later nodes
  - No cycles are allowed in dependence graph

Input string → **Parse tree** → **Dependence graph** → **Evaluation order for Semantic rules**



Evaluation Of Semantic Rules

- Dynamic methods (compile time)
  - Build a parse tree for each input
  - Build a dependency graph from the parse tree
  - Obtain evaluation order from a topological order of the dependency graph
- Rule-based methods (compiler-construction time)
  - Predetermine the order of attribute evaluation based on grammar structure of each production
  - Example: semantic rules defined in Yacc
- Oblivious methods (compiler-construction time)
  - Evaluation order is independent of semantic rules
  - Evaluation order forced by parsing methods
  - Restrictive in acceptable attribute definitions

L-attributed Definitions

- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1<=j<=n$, on the right side of $A::=X_1X_2\ldots X_n$, depends only on
  - the attributes of $X_1, X_2, \ldots, X_{j-1}$ to the left of X_j in the production
  - the inherited attributes of A

L-attributed definition

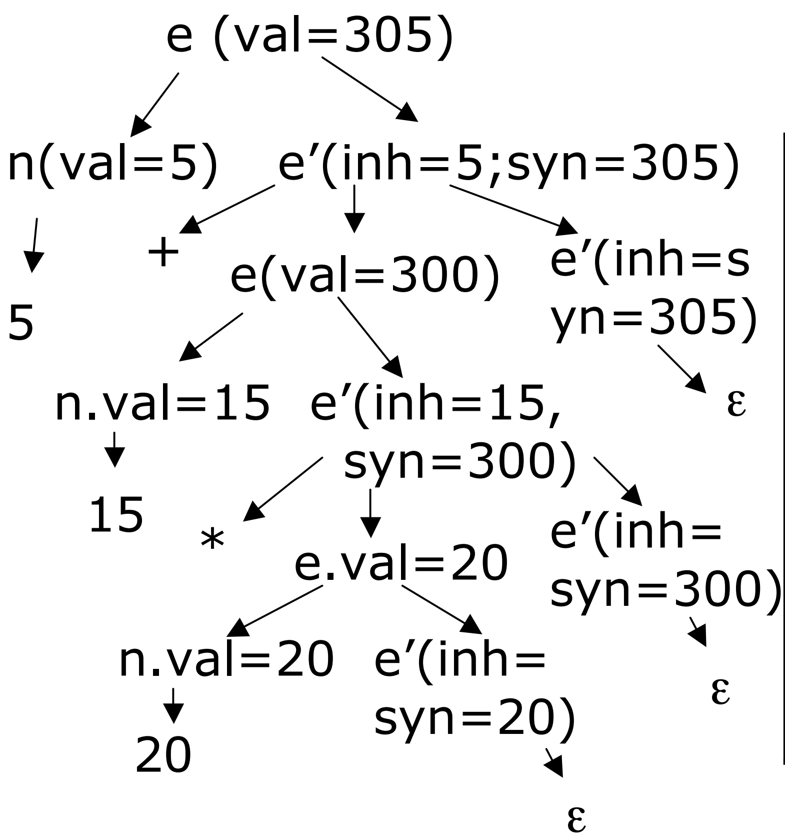
Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

Non L-attributed definition

Production	Semantic rules
A::=L M	L.i = A.i M.i = L.s A.s = M.s
A ::= Q R	R.i = A.i Q.i = R.s A.s = Q.s

Synthesized And Inherited Attributes

- L-attributes may include both synthesized and inherited attributes



e ::= n e’
e’ ::= +ee’ | *ee’ | ε

production	Semantic rules
e ::= n e’	e’.inh=n.val; e.val = e’.syn
e’ ::= + e e’1	e’1.inh = e’.inh [+] e.val e’.syn = e’1.syn
e’ ::= * e e’1	e’1.inh = e’.inh [*] e.val e’.syn = e’1.syn
e’ ::= ε	e’.syn = e’.inh

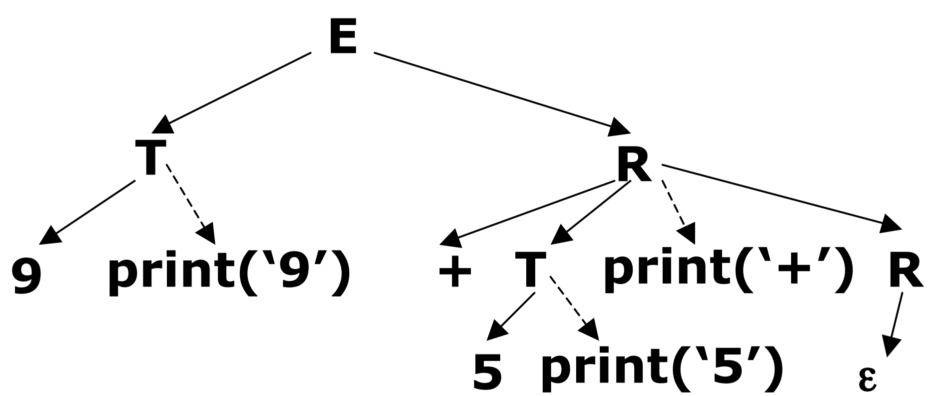
Translation Schemes

- A translation scheme is a BNF where
  - Attributes are associated with grammar symbols and
  - Semantic actions are inserted within right sides of productions
- Notation for specifying translation during parsing

Translation scheme:

```
E ::= T R
R ::= '+' T {print('+')} R1
      | ε
T ::= num {print(num.val)}
```

Parse tree for 9+5 with actions



Treat actions as though they are terminal symbols.

Designing Translation Schemes

- Step1: decide how to evaluate attributes at each production

D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in; Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

- Step2: decide where to evaluate each attribute
  - S-attribute of left-hand symbol computed at end of production
  - I-attribute of right-hand symbol computed before the symbol
  - S-attribute of right-hand symbol referenced after the symbol

D::=T **{** L.in:=T.type**}** L
T::= **int** **{**T.Type:=integer**}**
T::=**real** **{** T.type:=real**}**
L::= **{**L1.in := L.in**}** L1**,id** **{**Addtype(id.entry,L.in) **}**
L::=**id** **{**Addtype(id.entry,L.in)**}**

Exercises

- Given the following grammar for a binary number generator

  S ::= L   L ::= L B | B   B ::= 0 | 1
- Compute the value of each resulting number
  - E.g., if s => … => 1101, then the value of s is 13
- Compute the contribution of each digit
  - E.g., if s => … => 1101, the contribution of the four digits are 8,4,0,1 respectively.

Steps for writing translation schemes

(1) Define a set of attributes for each grammar symbol

(2) Categorize each attribute as synthesized or inherited

(3) For each production, define how to evaluate

  (3.1) synthesized attribute of the left-hand symbol

  (3.2) inherited attribute of each right-hand symbol

(4) Insert each attribute evaluation inside the production

  Inherited attribute==> before the symbol;

  synthesized attribute ==> at end of production

Top-Down Translation

- In top-down parsing, a parsing function is associated with each non-terminal
  - To support attribute evaluation, add parameters and return values to each function
- For each non-terminal A, construct a parsing function that
  - Has a formal parameter for each inherited attribute of A
  - Returns the values of the synthesized attributes of A
- The code associated with each production does the following
  - Save the s-attribute of each symbol X into a variable X.s
  - Generate an assignment B.s=parseB(B.i1,B.i2,…,B.ik) for each non-terminal B, where B.i1,…,B.ik are values for the L-attributes of B and B.s is a variable to store s-attributes of B.
  - Copy the code for each action, replacing references to attributes by the corresponding variables

Top-Down Translation Example

```
void parseD()
   { Type t = parseT(); }
      parseL(t);
   }
Type parseT
     {  switch (currentToken()) {
          case INT: return TYPE_INT;
          case REAL:  return TYPE_REAL;
          }
     }
void parseL(Type in)
     {
        SymEntry e = parseID();
        AddType(e, in);
        if (currentToken() == COMMA) {
           parseTerminal(COMMA);
           parseL(in)
       }
}
```

D::=T **{** L.in:=T.type**}** L
T::= **int** **{**T.Type:=integer**}**
| **real** **{** T.type:=real**}**
L::=**id** **{**AddType(id.en,L.in)**}**
| id {AddType(id.en,L.in)} **,** {L1.in=L.in} L1

Bottom-up Evaluation Of Attributes

- Synthesized attributes: consistent with bottom-up reduction
  - Keep attribute values of grammar symbols in stack
  - Evaluate attribute values at each reduction
- Inherited attribute: use attributes already stored in stack
  - Each inherited attribute evaluation is treated as a dummy grammar symbol
  - Evaluation results pushed into stack for later use

Configuration of LR parser:

$$(\underbrace{s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m}_{\text{states}},\ \underbrace{a_i a_{i+1} \ldots a_n \$}_{\text{inputs}},\ \underbrace{v_1 v_2 \ldots v_m}_{\text{values}})$$

Right-sentential form: $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n \$$

Automata states: $s_0 s_1 s_2 \ldots s_m$

Grammar symbols in stack: X1X2…Xm

Synthesized attribute values of $X_i \rightarrow v_i$

Bottom-Up Translation In Yacc

D::=T **{** L.in:=T.type**}** L
T::= **int** **{**T.Type:=integer**}**
T::=**real** **{** T.type:=real**}**
L::= **{**L1.in := L.in**}** L1**,id** **{**Addtype(id.entry,L.in) **}**
L::=**id** **{**Addtype(id.entry,L.in)**}**

```
D : T  {$$ = $1; } L
T : INT { $$ = integer; } | REAL { $$ = real; }
L : L COMMA ID { Addtype($3, $0); }
    | ID { Addtype($1,$0); }
```

Types in Programming

- A type is a collection of computable values
  - Represent concepts from problem domain
    - Accounts, banks, employees, students
  - Represent different implementation of values
    - Integers, strings, floating points, lists, records, tuples …
    - Must know the type of a variable before allocating space
- Languages use types to
  - Support organization of concepts (programmability)
  - Support consistent interpretation of values (error checking)
    - Compile-time and run-time type checking
    - Prevent meaningless computation

      3 + true - "Bill"
  - Support efficient translation (by compilers)
    - Short integers require fewer bits
    - Access record component by a known offset
    - Use integer units for integer operations

Values and Types

- Basic types: types of atomic values
  - int, bool, character, real, symbol
  - Values of different types
    - have different layouts
    - have different operations
  - Explicit vs. implicit type conversion of values
- Compound types: types of compound values
  - List, record, array, tuple, struct, ref, pointer
  - Built from type constructors
    - int arr[100] → arr: array(int,100)
    - **(3, 4, “abc”)** : **int * int * string**
    - int *x → x : **pointer(int)**
    - **int f(int x) { return x + 5}** → f : **int→int**

Variables, Scopes, and Binding

- Values and objects (atomic and compound values)
  - Created -> bound to variables -> destructed
  - Their storages could be allocated differently
    - Static allocation: used to initialize global variables
    - Stack allocation: used to initialize local variables of functions/subroutines
    - Heap allocation: dynamically allocated/deleted and used to initialize pointer variables
- Variables: used as placeholders for values
  - Lifetime: from creation to destruction of its value
  - Scope: the block where it is declared (and can be accessed)
  - Binding time: when is a variable bound to its value/storage?
    - Binding variable to value: cannot be modified (functional programming)
    - Binding variable to storage: can be modified (imperative programming)

Managing Storage Using Blocks

- Blocks: regions with local variable declarations
  - Blocks are nested but not partially overlapped
    - What about jumping into the middle of a block?
- Storage management
  - Enter block: allocate space for variables (must know their types)
  - Exits block: some or all space may be deallocated
- Local variables: declared inside the current block
- Global variables: declared in a enclosing block
  - Already allocated before entering current Block
  - Remain allocated after exiting current block
- Function parameters
  - Input parameters
    - Allocated and initialized before entering function body
    - De-allocated after exiting function body
  - Return parameters
    - Address remembered before entering function body
    - Value set after exiting function body

The Type System

- A language supports each type by
  - Providing ways to introduce values of the type
    - Literal integers: 1 23 -3290
    - Literal floating point numbers: 3.5 0.12
    - Arrays, pointers, structs, classes: type constructors
  - Providing ways to operate on values of the type
    - Evaluation rules, equality, introduction and elimination operations
- Every type comes with a set of operations
  - Each operation defined on specific types of operands and return a specific type of value
    - A *type error* occurs if operation applied outside its domain
  - The interfaces of operators are their types (i.e. function types)
- Type declarations
  - Provide ways to declare types of variables
  - Provide ways to introduce new types (user-defined types)

Type Declarations

- Goal: provide ways to introduce new types
  - These types are called user-defined types
- Transparent declarations
  - Introduce a synonym for another type
  - Examples in C
    - typedef struct { int a, b; } mystruct;
    - typedef mystruct yourstruct;
- Opaque declarations
  - Introduce a new type
  - Examples in C
    - struct XYZ { int a, b,c; };
  - Any other examples?

Type Equivalence

- When are two types considered equal?

  struct s {int a,b; }=struct t {int a,b; } ?

  - Structural equivalence: yes
    - s and t are the same basic type or
    - s and t are built using the same compound type constructor with the same components
  - Name equivalence: no
    - S and t are different names
    - Names uniquely define compound type expressions
  - In C, name equivalence for records/structs, structural equivalence for all other types

Polymorphism

- A function is polymorphic if it can operate on different types
  - Interpreted languages
    - Support arbitrary polymorphic functions
    - Type information stored together with each value
  - Compiled languages
    - Need to know storage size for each input value
    - Each expression (including functions) can only only a single type
- Subtype polymorphism: subset relations between types
  - Example in C: a union type includes all of its base types
  - Example in C++/Java, Truck is a subclass of Car
- Parametric polymorphism:
  - Operate on types parameterized with type variables
    - e.g., C++ templates, Java generics
- Ad hoc polymorphism: operator overloading
  - A single function name given different types and implementations
    - + : int->int;   + : real->real

Type Error

- When a value is misinterpreted or misused with unintended semantics, a type error occurs
  - May cause hardware error

    function call x() where x is not a function
    - may cause jump to instruction that does not contain a legal op code
  - May simply return incorrect value

    int_add(3, 4.5)
    - not a hardware error
    - bit pattern of 4.5 can be interpreted as an integer
    - just as much an error as x() above

Type-Safety Of Languages

- Type-safe: report error instead of segmentation faults
- BCPL family, including C and C++
  - Not safe: casts, pointer arithmetic, …
- Algol family, Pascal, Ada
  - Almost safe
  - Dangling pointers:
    - Pointers to locations that have been deallocated
    - No language with explicit de-allocation of memory is fully type-safe
- Type-safe languages with garbage collection
  - Lisp, ML, Smalltalk, Java
  - Dynamically typed: Lisp, Smalltalk
  - Statically typed: ML, JAVA

Type Checking

- Goal: discover and report type errors
  - Type system specify the proper usage of each operator
    - Reject expressions that cannot be typed according to rules
    - Explicit vs. implicit type conversion
  - Can be done at compile-time or run-time, or both
- Run-time (dynamic) type checking
  - Check type safety before evaluating each operation
    - Store type information together with each value in memory
  - In POET, before evaluating (car x), interpreter checks x is a non-empty list
- Compile-time (static) type checking
  - Each variable/expression must have a single type
  - E.g., int f(float x) declares that f can be invoked only with float-type expressions

Static vs Dynamic Type Checking

- Both prevent type errors
- Run-time checking: check before each operation
  - Pros: flexibility and safety
    - Variables/expressions could have arbitrary types
    - Can detect all type errors (language is type safe)
  - Cons: slow down execution, and error detection may be too late
- Compile-time checking
  - Pros: efficiency (no runtime overhead) and early error detection
  - Cons: flexibility and safety
    - Every variable/function can have only a single type
    - Cannot detect some type errors, e.g., accessing arrays out-of-bound, dangling pointers
- Combination of compile and runtime checking
  - Example: Java (array bound check at runtime)

Type Inference

- Static type checking in C/C++/Java

  int f(int x) { return x+1; };
  int g(int y) { return f(y+1)*2;};

  - Programmer has to declare the types of all variables
  - Compilers evaluate the types of expressions and check agreement

- Type inference: extension to static type checking

  ~~int~~ f(~~int~~ x) { return x+1; };
  ~~int~~ g(~~int~~ y) { return f(y+1)*2;};

  - Programmers are not required to declare types for variables
  - Compilers figure out agreeable types of all expressions
    - Solving constraints based on how expressions are used

Compile Time Type Checking

- Types of variables
  - Each variable must have a single type
    - It can hold only values of this type
- Types of expressions
  - Every expression must have a single type
    - It maps input values to a return value
    - It can return only values of this type
- Type system
  - Rules for deciding types of expressions
    - These rules specify the proper usage of each operator
  - Accept only expressions that can be typed according to rules
  - Explicit vs. implicit type conversion

Type Environment

- Symbol table
  - Record information about names defined in programs
    - Types of variables and functions
    - Additional properties (eg., scope of variable)
  - Contain information about context of program fragment
- Name conflicts
  - The same name may represent different things in different places
    - Separate symbol tables for names in different scopes
    - Multiple layers of symbol definitions for nested scopes
- Implementation of symbol tables
  - Hash table from strings (names) to properties (types)

Evaluating Types Of Expressions

```
P ::= D ;  E
D ::= D ; D | id : T
T ::= char | integer
E ::= literal | num | id | E mod E
```

```
P ::= D ; E
D ::= D ; D  |    id : T  { addtype(id.entry, T.type); }
T ::= char { T.type = char; } | integer { T.type = integer ;}
E ::= literal      { E.type = char;} |  num  { E.type = num;}
      | id  { E.type = lookupType(id.entry); }
      | E1 mod E2  {if (E1.type == integer && E2.type==integer)
                             E.type = integer; else E.type = type_error;}
```

Type Checking With Coercion

- Implicit type conversion
  - When type mismatch happens, compilers can automatically convert inconsistent types into required types
    - 2 + 3.5: convert 2 to 2.0 before adding 2.0 with 3.5

```
E ::= ICONST   { E.type = integer;}
E ::= FCONST { E.type = real; }
E ::= id          { E.type = lookup(id.entry); }
E ::= E1 op E2   { if (E1.type==integer and E2.type==integer)
                         E.type = integer;
                  else if (E1.type==integer and E2.type==real)
                         E.type=real;
                  else if (E1.type==real and E2.type==integer)
                          E.type=real;
                  else if (E1.type==real and E2.type==real)
                         E.type=real;
                  }
```

Example: Types For Arrays

```
P ::= D ;  E
D ::= D ; D | id : T
T ::= char | integer | T [ num ]
E ::= literal | num | id | E mod E | E[E]
```

```
P ::= D ; E
D ::= D ; D  |    id : T  { addtype(id.entry, T.type); }
T ::= char { T.type = char; } | integer { T.type = integer ;}
      | T1[num]  { T.type = array(num.val, T1.type);}
E ::= literal     { E.type = char;} |  num  { E.type = num;}
      | id  { E.type = lookupType(id.entry); }
      | E1 mod E2  {if (E1.type == integer && E2.type==integer)
                           E.type = integer; else E.type = type_error;}
      | E1[E2]  { if (E2.type == integer && E1.type==array(s,t))
                           E.type = t;   else   E.type = type_error; }
```

Exercise:Type Checking For Arrays

```
P ::= P S | S
S ::= T D ";" | E ";"
T ::= float | integer
D ::= id | D[inum]
E ::= fnum | inum | id | E+E | E[E]
```

Example: Type checking For Statements

```
P ::= D ;  S
D ::= D ; D | id : T
T ::= char | integer
S ::= id `=' E ; | {S S} | if (E) S | while (E) S
E ::= literal | num | id | E mod E
```

```
S ::= id '=' E ;   { if (E.type!=type_error &&
                         Equiv(lookup_type(id.entry),E.type))
                           S.type = void;
                     else S.type = type_error; }
    | '{' S1 S2 '}' {  if (S1.type == void)  S.type = S2.type;
                         else S.type = type_error; }
    | if '(' E ')' S1 { if (E.type == integer) S.type=S1.type;
                        else S.type=type_error; }
    | while '(' E ')' S1 { if (E.type == integer) S.type=S1.type;
                        else S.type=type_error; }
```

Example:
Type Checking With Function Calls

```
P ::= D ;  E
D ::= D ; D | id : T | T id (Tlist)
Tlist ::= T, Tlist | T
T ::= char | integer | T [ num ]
E ::= literal | num | id | E mod E | E[E]  | E(Elist)
Elist ::= E, Elist | E
```

```
……
D ::= T1 id (Tlist)  { addtype(id.entry, fun(T1.type,Tlist.type)); }
Tlist ::= T, Tlist1  { Tlist.type = tuple(T1.type, Tlist1.type); }
        | T  { Tlist.type = T.type }
E ::= E1 ( Elist )   { if (E1.type == fun(r, p) && p ==Elist.type)
                        E.type = r ;  else E.type = type_error; }
Elist ::= E, Elist1   { Elist.type = tuple(E1.type, Elist1.type); }
        | E           { Elist.type = E.type; }
```