Semantic Analysis

The role of semantic analysis is to assign meaning

❑ "It smells fishy."

❑ Lexical analysis
  ➢ Tokenizes "It", "smells", "fishy", "."
  ➢ Determines noun, verb, adjective, punctuation token types

❑ Syntax analysis
  ➢ Parses the grammatical structure of the sentence

❑ Semantic analysis

The role of semantic analysis is to assign meaning

❑ "It smells fishy."

❑ Lexical analysis
  ➢ Tokenizes "It", "smells", "fishy", "."
  ➢ Determines noun, verb, adjective, punctuation token types

❑ Syntax analysis
  ➢ Parses the grammatical structure of the sentence

❑ Semantic analysis
  ➢ Assigns meaning to the words "It", "smells", "fishy"
  ➢ Flags error if the sentence does not make sense

Semantic Analysis = Binding + Type Checking

❑ "I don't wanna eat that sushi."

"It smells fishy."

➢ "It": the sushi
➢ "smells": feels to my nose
➢ "fishy": that the sushi has gone bad

❑ "The professor says that the exam is going to be easy."

"It smells fishy."

➢ "It": the situation
➢ "smells": feels to my sixth sense
➢ "fishy": that it is highly suspicious

Semantic Analysis = Binding + Type Checking

❏ "I don't wanna eat that sushi."

   "It smells fishy."
   - ➤ "It": the sushi
   - ➤ "smells": feels to my nose
   - ➤ "fishy": that the sushi has gone bad

❏ "The professor says that the exam is going to be easy."

   "It smells fishy."
   - ➤ "It": the situation
   - ➤ "smells": feels to my sixth sense
   - ➤ "fishy": that it is highly suspicious

❏ Semantic analysis consists of two tasks
   - ➤ **Binding**: associating a pronoun to an object
   - ➤ **Type checking**: inferring meaning based on type of object

Semantic Analysis = Binding + Type Checking

❏ Semantic analysis performs binding
  ➢ Done by traversing parse tree produced by syntax analysis
  ➢ Declarations are stored in data structure called **symbol table**
  ➢ Uses are bound to entries in the symbol table

❏ Semantic analysis performs type checking
  ➢ Infers what "$a + b$" means:
    ● If a and b are ints, integer add and return int
    ● If a and b are floats, FP add and return float
    ● If a and b are strings, concatenate and return string
  ➢ Infers what "$x.foo()$" means:
    ● If object x is a reference of class A, call to foo() in A
    ● If object x is a reference of class B, call to foo() in B
  ➢ Infers what "$a[i][j]$" means:
    ● Offset from a based on array type and dimensions

Semantic analysis also performs semantic checks

❏ All symbol uses have corresponding declarations

❏ All symbols defined only once
- ➤ Where symbols can be variables, methods, classes
- ➤ Declaration: provides type information for a symbol
- ➤ Definition: allocates a symbol in program memory

❏ All statements do not violate type rules
- ➤ Operators (+, -, *, /, =, >, <, ==, ...) have legal parameters
- ➤ Method calls have correct numbers of legal parameters
- ➤ Private methods are not called by external classes
- ➤ ...

Symbol Binding

What is symbol binding?

"Matching symbol **declarations** with **uses**"

☐ If there are multiple declarations, which one is matched?

What is symbol binding?

"Matching symbol **declarations** with **uses**"

◻ If there are multiple declarations, which one is matched?

```
void foo()
{
  char x;
  ...
  {
     int x;

     ...
  }
  x = x + 1;
}
```

What is symbol binding?

"Matching symbol **declarations** with **uses**"

◻ If there are multiple declarations, which one is matched?

```
void foo()
{
  char x;
  ...
  {
      int x;  ?
  ?
  }
  x = x + 1;

}
```

Scope

❏ Binding: associating a symbol use to its declaration
➢ Which variable (or function) an identifier is referring to

❏ Scope: section of program where a declaration is valid
➢ Uses in the scope of declaration are bound to it

❏ Some implications of scopes
➢ A symbol may have different bindings in different scopes
➢ Scopes for the same symbol never overlap
- there is always exactly one binding per symbol use

❏ Two types: static scope and dynamic scope

Static Scope

❑ Static Scope: scope expressed in program text
  ➢ Also called Lexical Scope
  ➢ C/C++, Java, JavaScript, Python

❑ Rule: bind to the closest enclosing declaration

```
void foo()
{
  char x;

  ...
  {
    int x;

    ...
  }
  x = x + 1;
}
```

Static Scope

❑ Static Scope: scope expressed in program text
  ➢ Also called Lexical Scope
  ➢ C/C++, Java, JavaScript, Python

❑ Rule: bind to the closest enclosing declaration

```
void foo()
{
  char x;
  ...
  {
      int x;

      ...
  }
  x = x + 1;
}
```

Dynamic Scope

❑ Dynamic Scope: bindings formed during code execution
   ➤ LISP, Scheme, Perl

❑ Rule: bind to the most recent declaration during execution

```
void foo()
{
   (1) char x;
   (2) if (...) {
   (3)     int x;
   (4)        ...
          }
   (5) x = x + 1;
}
```

Dynamic Scope

❏ Dynamic Scope: bindings formed during code execution
  ➤ LISP, Scheme, Perl

❏ Rule: bind to the most recent declaration during execution
```
void foo()
{
  (1) char x;
  (2) if (...) {
  (3)    int x;
  (4)      ...
         }
  (5) x = x + 1;
}
```

❏ Which *x*'s declaration is the closest?
  ➤ Execution (a): ...(1)...(2)...(5)
  ➤ Execution (b): ...(1)...(2)...(3)...(4)...(5)

Static vs. Dynamic Scoping

❏ Most languages that started with dynamic scoping (LISP, Scheme, Perl) added static scoping afterwards

❏ Why?
- ➢ It is easier for human beings to understand
  - Bindings are visible in code without tracing execution
- ➢ It is easier for compilers to understand
  - Compiler can determine bindings at compile time
  - Compiler can translate identifier to a single memory location
  - Results in generation of efficient code
- ➢ With dynamic scoping...
  - There may be multiple possible bindings for a variable
  - Impossible to determine bindings at compile time
  - All bindings have to be done at execution time
    (Typically with the help of a hash table)

Symbol Table

Symbol Table

- **Symbol Table**: A compiler data structure that tracks information about all identifiers (symbols) in a program
  - ➤ Maps symbol uses to declarations given a scope
  - ➤ Needs to provide bindings according to the current scope

- Usually discarded after generating the binary code
  - ➤ All symbols are mapped to memory locations already
  - ➤ For debugging, symbols may be included in binary
    - To map memory locations back to symbols for debuggers
    - For GCC or Clang, add "-g" flag to include symbol tables

Maintaining Symbol Table

❑ Basic idea:

int x; ... void foo() { int x; ... x=x+1; } ... x=x+1 ...

➢ In *foo*, add *x* to table, overriding any previous declarations

➢ After *foo*, remove *x* and restore old declaration if any

❑ Operations

enter_scope()	start a new nested scope
exit_scope()	exit current scope
find_symbol(x)	find declaration of *x*
add_symbol(x)	add declaration of *x* to symbol table

Adding Scope Information to the Symbol Table

❏ To handle multiple scopes in a program,
  ➢ (Conceptually) need an individual table for each scope
  ➢ Symbols added to the table may not be deleted just
    because you exited a scope

```
class X { ... void f1() {...} ... }
class Y { ... void f2() {...} ... }
class Z { ... void f3() {
    X v;
    v.f1();
} ... }
```

  ➢ Without deleting symbols, how are scoping rules enforced?
    ☞ Keep a list of all scopes in the entire program
    ☞ Keep a stack of active scopes at a given point

Symbol Table with Multiple Scopes
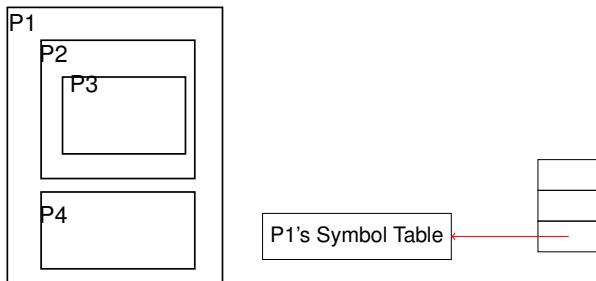
❑ For nested scopes,

➢ Search from top of the active symbol table stack
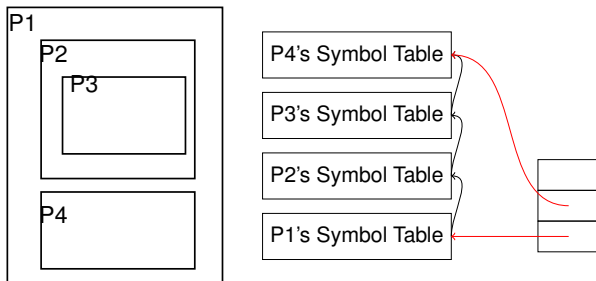
➢ Remove pointer to symbol table when exiting its scope

Symbol Table with Multiple Scopes
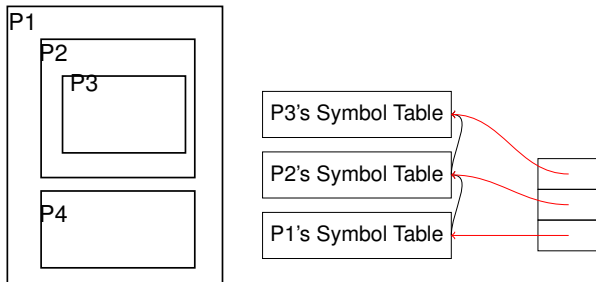
□ For nested scopes,

➢ Search from top of the active symbol table stack

➢ Remove pointer to symbol table when exiting its scope

Symbol Table with Multiple Scopes

☐ For nested scopes,

  ➢ Search from top of the active symbol table stack
  ➢ Remove pointer to symbol table when exiting its scope

What Information is Stored in the Symbol Table

❑ Entry in Symbol Table:

string	kind	attributes

➢ String — the name of identifier
➢ Kind — variable, parameter, function, class, ...

❑ Attributes vary with the kind of symbol

➢ variable → type, address in memory
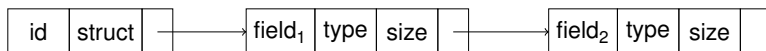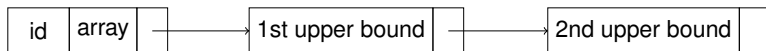➢ function → return type, parameter types, address

❑ Vary with the language

➢ Fortran's array → type, dimension, dimension size
real A(5) /* dimension required for static allocation */
➢ C's array → type, dimension, optional dimension size
char A[5]; /* statically sized array */
char A[]="hello"; /* dynamically sized to fit content */

Symbol Table Attribute List

❏ Type information might be arbitrarily complicated
  ➢ In C: struct {
            int a[10];
            char b;
            float c;
           }
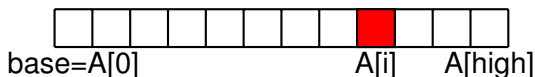
❏ Store all relevant attributes in an attribute list

| id | array | | → | 1st upper bound | | → | 2nd upper bound | |

| id | struct | | → | field$_1$ | type | size | → | field$_2$ | type | size | |

Example application of Type to an operator:
Array index operator

Addressing Array Elements

int A[0..high];

A[i] ++;

base=A[0]   A[i]   A[high]

> width — width of element type
> base — address of the first
> high — upper bound of subscript

☐ Addressing an array element:

address(A[i]) = base + i * width

offset(A[i]) = i * width
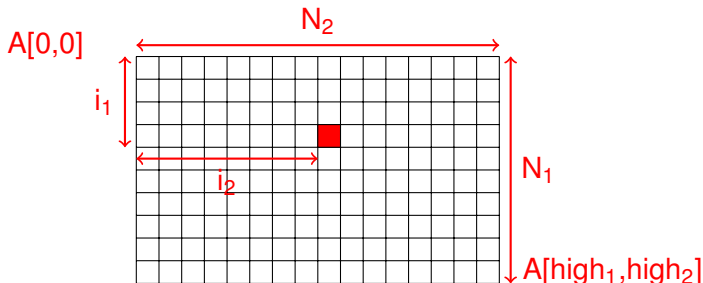
Multi-dimensional Arrays
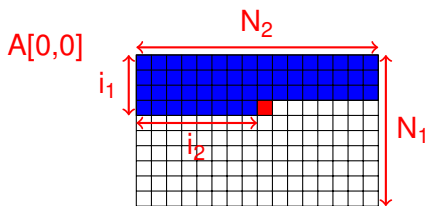
❑ Layout n-dimension items in 1-dimension memory

$\quad$ int $A[N_1][N_2]$; /* int $A[0..high_1][0..high_2]$; */

$\quad$ $A[i_1][i_2]$ ++;

Row Major

Row major — store row by row

A[0,0]

N_2

i_1

i_2

N_1

❑ Offset inclues all the "blue" items before $A[i_1, i_2]$

offset($A[i_1, i_2]$) = ($i_1 * N_2 + i_2$) * width
= $i_1 * N_2 *$ width + $i_2 *$ width
= offset($A[i_1]$) * $N_2 + i_2 *$ width

Column Major

Column major — store column by column
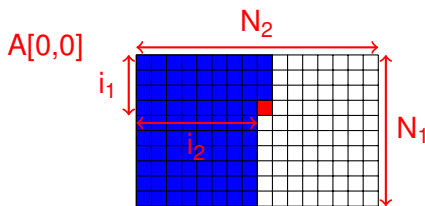
☐ Offset inclues all the "blue" items before $A[i_1, i_2]$

$offset(A[i_1, i_2]) = (i_2 * N_1 + i_1) * width$
$= i_2 * N_1 * width + i_1 * width$
$= i_2 * N_1 * width + offset(A[i_1])$

Generalized Row/Column Major

❑ Let A_k = offset($A[i_1, i_2, ..., i_k]$). Then,

❑ Row major (C/C++, C#, Objective-C)
 1-dimension: $A_1 = i_1*$width
 2-dimension: $A_2 = (i_1*N_2+i_2)*$width $= A_1*N_2+i_2*$width
 3-dimension: $A_3 = (i_1*N_2*N_3+i_2*N_3+i_3)*$width $= A_2*N_3+i_3*$width
 k-dimension: $A_k = A_{k-1}*N_k + i_k*$width
 ☞**Type** needs to provide $N_2...N_k$ and width for offset

❑ Column major (Fortran, Matlab, R)
 1-dimension: $A_1 = i_1*$width
 2-dimension: $A_2 = (i_2*N_1 + i_1)*$width $= i_2*N_1*$width $+ A_1$
 3-dimension: $A_3 = ((i_3*N_2+i_2)*N_1+i_1)*$width $= i_3*N_2*N_1*$width $+ A_2$
 k-dimension: $A_k = i_k*N_{k-1}*N_{k-2}*...*N_1*$width$+A_{k-1}$
 ☞**Type** needs to provide $N_1...N_{k-1}$ and width for offset

C's implementation

☐ C uses row major

```
int fun1(int p[ ][100])
{
...
  int a[100][100];
  a[i₁][i₂] = p[i₁][i₂] + 1;
}
```

Why is p[][100] allowed?

Why is a[][100] not allowed?

C's implementation

❏ C uses row major

```
int fun1(int p[ ][100])
{
...
  int a[100][100];
  a[i₁][i₂] = p[i₁][i₂] + 1;
}
```

Why is p[][100] allowed?

➤ The info is enough to compute $p[i_1][i_2]$'s address

➤ $A_2 = (i_1 * N_2 + i_2) *$ width (N_1 is not required)

Why is a[][100] not allowed?

➤ The info is not enough to allocate space for the array

Type Checking

What, When, and Why

❑ What?
  ➢ Type: a set of values + a set of operations on values
  ➢ Type Checking: Verifying and enforcing type consistency
    ● Only legal values are assigned to a type
    ● Only legal operations are performed on a type

❑ When?
  ➢ Static Type Checking: Type checking at compile-time
  ➢ Dynamic Type Checking: Type checking at execution time

❑ Static type checking is more desirable. Why?
  ➢ Better to fail at compile time than during deployment
  ➢ Dynamic type checking can impact runtime performance
  ➢ Check dynamically only when static checking is infeasible
    ● E.g. Java array bounds checks
    ● E.g. Type checks to verify C++/Java downcasting

Static vs. Dynamic Typing

❏ Statically typed: C/C++, Java   ☞Our discussion

➢ Types are explicitly declared or can be inferred from code
  int x; /* type of x is int */
➢ Better compiler error detection due to static type checks
➢ Efficient code since dynamic type checks are not needed

❏ Dynamically typed: Python, JavaScript, PHP

➢ Type is a runtime property decided only during execution
  var x; /* type of x is undecided */
  x = 42; /* type of x is int */
  x = "forty two"; /* type of x is now string */
  /* Type of x changes depending on the value it holds */
➢ Static type checking and error reporting is impossible
➢ More memory since every variable now needs a "type tag"
➢ Inefficient code due to dynamic checks on type tags

Rules of Inference

❏ What are *rules of inference*?
  - ➤ Inference rules have the form
    if **Precondition** is true, then **Conclusion** is true
  - ➤ Below concise notation used to express above statement

    Precondition
    _____
    Conclusion

  - ➤ In the context of type checking:
    if expressions E1, E2 have certain types (Precondition),
    expression E3 is legal and has a certain type (Conclusion)

❏ Type checking via inference
  - ➤ Start from variable types and constant types
  - ➤ Repeatedly apply rules until entire program is inferred legal

Notation for Inference Rules

❏ By tradition inference rules are written as

Precondition$_1$, ..., Precondition$_n$
Conclusion

➤ The precondition/conclusion has the form **"e:T"**

❏ Meaning

➤ If **Precondition$_1$** and ... and **Precondition$_n$** are true, then **Conclusion** is true.

➤ **"e:T" indicates "e is of type T"**

➤ Example: rule-of-inference for add operation

e_1: int
e_2: int

e_1+e_2:int

Rule: If e_1, e_2 are ints then e_1+e_2 is legal and is an int

Two Simple Rules

[Constant]

$$\frac{\text{i is an integer}}{\text{i: int}}$$

[Add operation]

$$\frac{e_1: \text{int}}{e_1+e_2:\text{int}}$$

❏ Example: given "10 is an integer" and "20 is an integer", is the expression "10+20" legal? Then, what is the type?

$$\frac{\text{10 is an integer}}{\text{10: int}} \qquad \frac{\text{20 is an integer}}{\text{20: int}}$$

$$\overline{\hspace{4cm}}$$

$$\text{10+20:int}$$

❏ This type of reasoning can be applied to the entire program

More Rules

[New]

$$\frac{}{\text{new T: T}}$$

[Not]

$$\frac{\text{e: Boolean}}{\text{not e: Boolean}}$$

◻ However,

[Var?]

$$\frac{\text{x is an identifier}}{\text{x: ?}}$$

➤ the expression itself insufficient to determine type
➤ **solution:** provide context for this expression

Type Environment

☐ A *type environment* gives type info for free variables

➤ A variable is *free* if not declared inside the expression
➤ It is a function mapping Symbols to Types

• Set of declarations active at the current scope
• Conceptual representation of a symbol table

Type Environment Notation

Let O be a function from Symbols to Types,
the sentence **O e:T**

is read as "under the assumption of environment O,
expression e has type T"

	O e1: int	
i is an intger	**O e2: int**	**O(x) == T**
O i: int	**O e1+e2: int**	**O x: T**

– "if i is an integer, expression i is an int in any environment"

– "if e1 and e2 are ints in O, expression e1+e2 is int in O"

– "if variable x is mapped to int in O, expression x is int in O"

Declaration Rule

[Declaration w/o initialization]

$$\frac{O[T_0/x] \; e_1 : T_1}{O \text{ let } x : T_0 \text{ in } e_1 : T_1}$$

$O[T_0/x]$: environment O modified so that it return T_0 on argument x and behaves as O on all other arguments:

$O[T_0/x](x) = T_0$

$O[T_0/x](y) = O(y)$ when $x \neq y$

☐ Translation: "If expression e_1 is type T_1 when x is mapped to type T_0 in the current environment, expression e_1 is type T_1 when x is declared to be T_0 in the current environment"

Declaration Rule with Initialization

[Declaration with initialization (initial try)]

$$O\ e_0 \colon T_0$$
$$O[T_0/x]\ e_1 \colon T_1$$
$$\overline{O\ \textbf{let}\ x \colon T_0 \leftarrow e_0\ \textbf{in}\ e_1 \colon T_1}$$

❏ The rule is too strict (i.e. correct but not complete)

Example
class C inherits P ...
let x:P ← new C in ...

☞ the above rule does not allow this code

Subtyping

❏ Subtyping is a relation ≤ on classes
  ➢ $X \leq X$
  ➢ if X inherits from Y, then $X \leq Y$
  ➢ if $X \leq Y$ and $Y \leq Z$, then $X \leq Z$

❏ An improvement of our previous rule

  [Declaration with initialization]

$$\frac{O \; e_0 : T \quad\quad T \leq T_0 \quad\quad O[T_0/x] \; e_1 : T_1}{O \; let \; x : T_0 \leftarrow e_0 \; in \; e_1 : T_1}$$

  ➢ Both versions of declaration rules are correct
  ➢ The improved version checks more programs

Assignment

⬛ A correct but too strict rule

[Assignment]

$$O(id) = T_0$$
$$O \vdash e_1 : T_1$$
$$\frac{T_1 \leq T_0}{O \vdash id \leftarrow e_1 : T_0}$$

➢ The rule does not allow the below code
    class C inherits P { only_in_C() { ... } }
    let x:C in
    let y:P in
    x ← y ← new C
    x.only_in_C()

Assignment

❏ An improved rule

[Assignment]

$$O(id) = T_0$$
$$O \vdash e_1 : T_1$$
$$T_1 \leq T_0$$

$$O \vdash id \leftarrow e_1 : T_1$$

➢ The rule now does allow the below code
   class C inherits P { only_in_C() { ... } }
   let x:C in
   let y:P in
   $x \leftarrow y \leftarrow$ new C
   x.only_in_C()

If-then-else

❏ Consider

     if e_0 then e_1 else e_2

- The result can be either e_1 or e_2
- The type is either e_1's type or e_2's type

➤ The best that we can do (statically) is the super type larger than e_1's type and e_2's type

❏ Least upper bound (LUB)

➤ $Z = lub(X,Y)$ — Z is defined as the least upper bound of X and Y *iff*

- $X \leq Z \wedge Y \leq Z$; Z is an upper bound
- $X \leq W \wedge Y \leq W \Longrightarrow Z \leq W$; Z is least among all upper bounds

If-then-else, case

[If-then-else]

$$\frac{O\ e_0\text{: Bool} \qquad O\ e_1\text{: }T_1 \qquad O\ e_2\text{: }T_2}{O\ \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi: lub}(T_1, T_2)}$$

❑ The rule allows the below code

    let x:float, y:int, z:float in

    x ← if (...) then y else z

    /* Assuming lub(int, float) = float */

Error Recovery

❑ Just like other errors, we should recover from type errors
  ➢ Too many errors?
    let y: int ← x+2 in y+3
    • if x is undefined —- reporting an error "x type undefined"
    • x+2 is undefined —- reporting an error "x+2 type undefined"
    • ...

❑ Introduce no-type for ill-typed expressions
  ➢ It is compatible with all types
  ➢ Report the place where no-type is generated
    • Reduce the number of error messages

Wrong Declaration Rule (case 1)

❑ Consider a hypothetical let rule

[Wrong Declaration with initialization (case 1)]

$$O \vdash e_0 : T$$
$$T \leq T_0$$
$$O \vdash e_1 : T_1$$
$$\overline{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

➤ How is it different from the the correct rule?
➤ The following good program does not pass check
　　let x: int ← 0 in x+1

Wrong Declaration Rule (case 2)

❏ Consider a hypothetical let rule

[Wrong Declaration with initialization (case 2)]

$$\frac{O\ e_0 : T \quad T_0 \leq T \quad O[T_0/x]\ e_1 : T_1}{O\ \text{let } x: T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

➢ How is it different from the the correct rule?
➢ The following bad program passes the check
    class B inherits A { only_in_B() { ... } }
    let x: B ← new A in x.only_in_B()

Discussion

❑ Type rules have to be carefully constructed, or

➢ The type system becomes unsound
(bad programs are accepted as well typed)

➢ The type system becomes unusable
(good programs are rejected as badly typed)

Discussion

❏ Type rules have to be carefully constructed, or
  ➢ The type system becomes unsound
    (bad programs are accepted as well typed)
  ➢ The type system becomes unusable
    (good programs are rejected as badly typed)

❏ What is a "good" program anyway?
  ➢ Good program: a program where all operations on all
    values are type consistent **at runtime**

Discussion

❏ Type rules have to be carefully constructed, or
  ➢ The type system becomes unsound
    (bad programs are accepted as well typed)
  ➢ The type system becomes unusable
    (good programs are rejected as badly typed)

❏ What is a "good" program anyway?
  ➢ Good program: a program where all operations on all
    values are type consistent **at runtime**

❏ All runtime behavior not expressed in a static type system
  ➢ At below is a good program rejected by the type system
    obj ← if (x > y) then new Child else new Parent
    if (x > y) then obj.only_in_Child()
  ➢ LUB type makes a choice of soundness over usability

Designing a Good Type Checking System

❑ A good type system achieves two opposing goals:
  ➤ Prevents **false negative** type errors, that is,
    runtime errors that are missed by type checking
  ➤ Minimizes **false positive** type errors, that is,
    type errors that do not cause runtime errors

❑ A good type system should allow the following code:

```
class Parent {
  Parent clone() { return new this.getClass(); }
}
class Child inherits Parent { ... }
  void main() {
    // Error! Assignment of parent to child reference.
    Child c ← (new Child).clone();
  }
}
```

What Went Wrong?

❏ What is (new Child).clone()'s type?
  - ➢ Dynamic type — Child
  - ➢ Static type — Parent

  - ➢ Type system is not able to express runtime types precisely
  - ➢ This makes inheriting clone() not very useful
    - clone() needs redefinition to return correct type anyway

❏ A "SELF_TYPE" would be useful in these situations.

SELF_TYPE expresses runtime types precisely

❑ What is SELF_TYPE?
  ➢ clone() returns "self" instead of "Parent" type
  ➢ Self can be Parent or any subclass of Parent

❑ SELF_TYPE is a static type
  ➢ Type reflects precise runtime behavior for each class
 ➢ Type violations can still be detected at compile time

❑ In practice
  ➢ Python, Rust, Scala: language support for self types
  ➢ C++: can emulate using C++ templates
  ➢ Java: can emulate to a lesser degree using Java generics

Can Static Type Checking ever be Perfect?

❏ Many examples where "good" programs are disallowed
  ➢ Reason for elaborate type systems like generics
  ➢ Why programmers must sometimes typecast anyway

❏ Solution? Can't have your cake and eat it too.
  ➢ Dynamic typing: values have types, variables do not
    + Allows all runtime behaviors that are type consistent
    - Type errors occur at runtime rather than compile time
    ☞ Best used for fast prototyping (scripting languages)
  ➢ Static typing: variables have declared (or inferred) types
    + Type errors can be caught at compile time
    - Effort needed to express "good" programs using type system
    ☞ Best used when reliability is important

Implementing Type Checking on AST
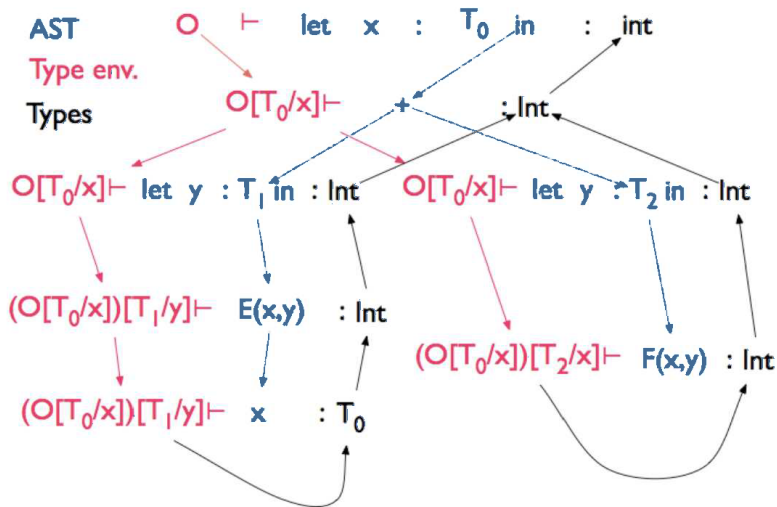
Syntax Directed Definitions (SDDs)

SDD: Definitions of attributes and rules

❏ Syntax Directed Definitions (SDD):
  1. Set of **attributes** attached to each grammar symbol
  2. Set of **semantic rules** attached to each production
  ➤ Semantic rules define values of attributes
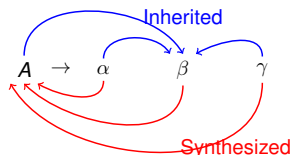
❏ Attribute Grammar:
  ➤ An SDD where rules depend only on other attributes
    (i.e. An SDD that does not rely on any side-effects)
  ➤ Think of it as a "grammar" for semantic analysis

❏ Example: let's say we want to define type checking
  ➤ SDD can have semantic rules to access a symbol table
  ➤ Attribute grammar must transmit type info through attributes

Syntax Directed Definition (SDD)

❑ Semantic rule:

SDD has rule of the form for each CFG production

$$b = f(c_1, c_2, ..., c_n)$$

either

1. If b is a synthesized attribute of A,
   c_1 $(1 \leq i \leq n)$ are attributes of grammar symbols of its Right Hand Side (RHS); or
2. If b is an inherited attribute of one of the symbols of RHS,
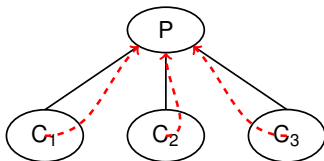 c_i's are attribute of A and/or other symbols on the RHS

Two Types of Attributes

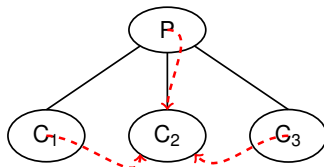❑ **Synthesized attributes:** attributes are computed from attributes of children nodes

➢ P.synthesized_attr = $f(C_1.attr, C_2.attr, C_3.attr)$

❑ **Inherited attributes:** attributes are computed from attributes of sibling and parent nodes

➢ C_3.inherited_attr = $f(P_1.attr, C_1.attr, C_3.attr)$

Synthesized attribute   Inherited attribute

Synthesized Attribute Example

❏ Example

➢ Each non-terminal symbol is associated with **val** attribute
➢ The **val** attribute is computed soley from children attributes

[Grammar Rules]	[Semantic Rules]
$L \rightarrow E$	print(E.val)
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T_1.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ digit	F.val = digit.lexval

Inherited Attribute Example

❑ Example:
- ➢ T.type: synthesized attribute
- ➢ L.in: inherited attribute
- ➢ id.type: inherited attribute

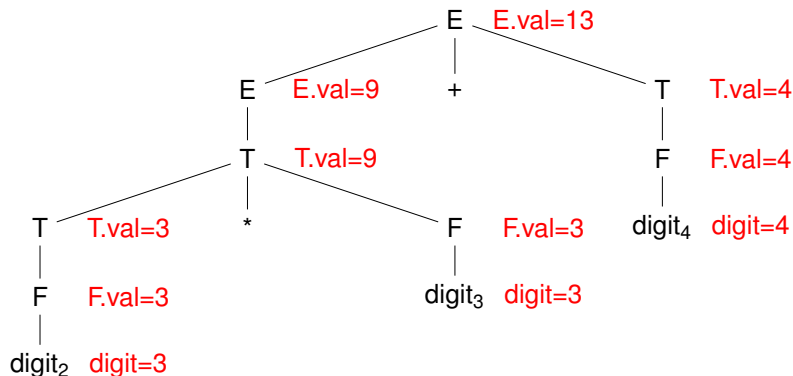| [Grammar Rules]	[Semantic Rules]
$D \rightarrow T L$	L.in = T.type
$T \rightarrow int$	T.type = integer
$T \rightarrow real$	T.type = real
$L \rightarrow L_1$, id	L_1.in = L.in, id.type = L.in
$L \rightarrow id$	id.type = L.in

❑ Why is L.in an inherited attribute?
- ➢ L.in is computed from a sibling T.type
- ➢ L_1.in is computed from a parent L.in

Attribute Parse Tree

▣ SDDs produce an attribute parse tree

  ➤ **Attribute parse tree**: Parse tree decorated with attributes
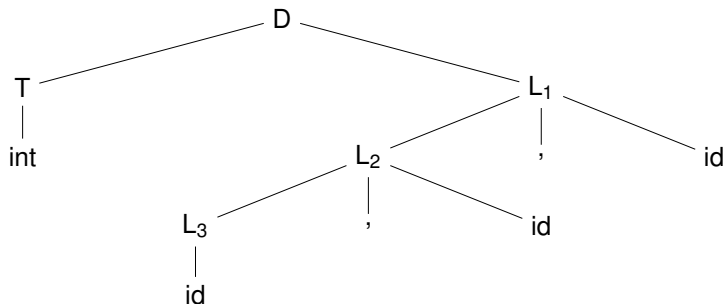
SDD Implementation

SDD Implementation using Parse Trees

❏ Assumes a previous parse stage
- ➢ Input: a parse tree with no attribute annotations
- ➢ Output: an attribute parse tree

❏ Goal: compute attribute values from leaf token values
- ➢ Traverse in some order, apply semantic rules at each node
- ➢ Traversal order must consider attribute dependencies

Dependency Graph

❏ Directed graph where edges are attribute dependencies
  ➢ "To" attribute is computed base on "from" attribute
  ➢ Must be **acyclic** such that there exists "a" traversal order

Dependency Graph

❏ Directed graph where edges are attribute dependencies
  ➢ "To" attribute is computed base on "from" attribute
  ➢ Must be **acyclic** such that there exists "a" traversal order
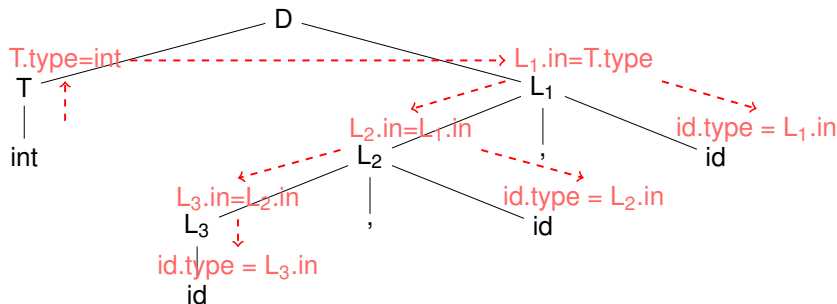
SDD Implementation using SDT

❏ Syntax Directed Translation (SDT)
  ➢ Applying semantic rules as part of syntax analysis (parsing)
  ➢ Does NOT assume a pre-existing parse tree
  ➢ Done through **semantic actions** embedded in grammar

❏ Semantic action:
  ➢ Code between curly braces embedded into RHS
  ➢ Executed "at that point" in the RHS
    ● Top-down: Right after previous symbol has been consumed
    ● Bottom-up: After previous symbol has been pushed to stack
                 (when the 'dot' reaches the semantic action)
  ➢ Example of building a parse tree:
    ● Program : Program IDNum ; Classes
      { $$=makeTree(ProgramOp, $2, $4); }
  ➢ $2 and $4 are indices into the parse stack
    ● RHS is currently at top of stack waiting to be reduced
    ● $2 is attribute value for IDNum and $4 is Classes

Syntax Directed Translation Scheme (SDTS)

❑ Syntax Directed Translation Scheme (SDTS)
  ➢ A "scheme" or plan to perform SDT
  ➢ A grammar specification embedded with semantic actions
  ➢ Depends on choice of top-down or bottom-up parser

❑ Example:

  $E \to T \quad R$
  $R \to + \quad T \quad R$
  $R \to - \quad T \quad R$
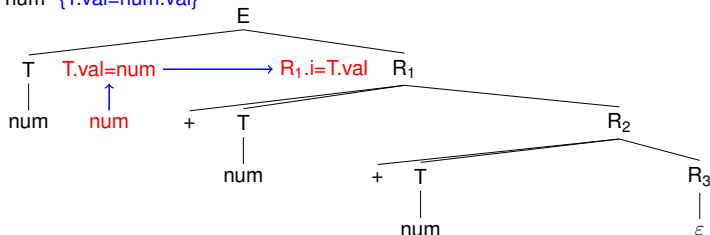 $R \to \varepsilon$
  $T \to (E)$
  $T \to num$

❑ Both inherited and synthesized attributes are used
  ➢ T — synthesized attribute T.val
  ➢ R — inherited attribute R.i
      synthesized attribute R.s
  ➢ E — synthesized attribute E.val

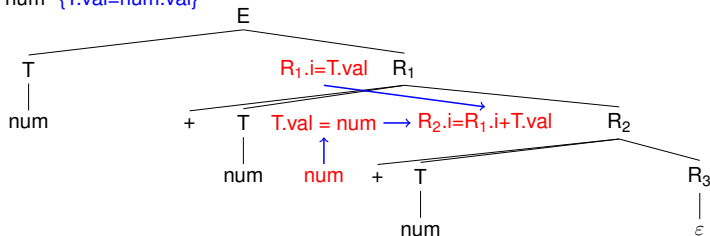# Syntax Directed Translation Scheme (SDTS)

❏ Evaluating attributes using SDTS

$E \rightarrow T$ {R.i=T.val}   R {E.val=R.s}
$R \rightarrow +$ T {R_1.i=R.i+T.val} R_1 {R.s=R_1.s}
$R \rightarrow -$ T {R_1.i=R.i-T.val} R_1 {R.s=R_1.s}
$R \rightarrow \varepsilon$ {R.s=R.i}
$T \rightarrow (E)$ {T.val=E.val}
$T \rightarrow num$ {T.val=num.val}

Syntax Directed Translation Scheme (SDTS)

❑ Evaluating attributes using SDTS

$E \rightarrow T$ {R.i=T.val}   R {E.val=R.s}
$R \rightarrow +$ T {R_1.i=R.i+T.val} R_1 {R.s=R_1.s}
$R \rightarrow -$ T {R_1.i=R.i-T.val} R_1 {R.s=R_1.s}
$R \rightarrow \varepsilon$ {R.s=R.i}
$T \rightarrow$ (E) {T.val=E.val}
$T \rightarrow$ num {T.val=num.val}

Syntax Directed Translation Scheme (SDTS)

❑ Evaluating attributes using SDTS

$E \to T$ {R.i=T.val}   R   {E.val=R.s}
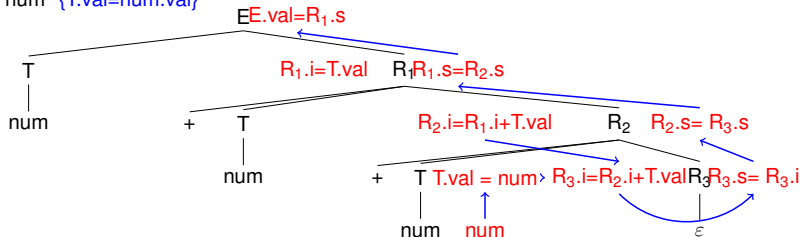$R \to +$ T {R_1.i=R.i+T.val} R_1 {R.s=R_1.s}
$R \to -$ T {R_1.i=R.i-T.val} R_1 {R.s=R_1.s}
$R \to \varepsilon$ {R.s=R.i}
$T \to$ (E)   {T.val=E.val}
$T \to$ num   {T.val=num.val}

What are the dependencies allowed in SDTS?

❑ Parse trees: dependencies only required to be acyclic

❑ What is required of dependencies for SDTS?
- ➤ Different parsing schemes see nodes in different orders
  - Top-down parsing — LL(k) parsing
  - Bottom-up parsing — LR(k) parsing
- ➤ What if dependency node has not been seen yet?

❑ For certain classes of SDDs, using SDTS is feasible
- ➤ If dependencies of SDD are amenable to parse order
- ➤ This class of SDDs are called L-Attributed Grammars

Left-Attributed Grammar

☐ An SDD is L-attributed if each of its attributes is either:

➤ a synthesized attribute of A in $A \rightarrow X_1...X_n$,

or

➤ an inherited attribute of X_j in $A \rightarrow X_1...X_n$ that
  - depends on attributes of siblings to its left i.e. $X_1...X_{j-1}$
  - and/or depends on parent A

Left-Attributed Grammar

❏ An L-Attributed grammar
  - ➢ may have synthesized attributes
  - ➢ may have inherited attributes but only from left sibling attributes or inherited attributes of the parent

❏ Evaluation order
  - ➢ Left-to-right depth-first traversal of the parse tree
    - Order for both top-down and bottom-up parsers
  - ➢ Evaluate inherited attributes while going down the tree
  - ➢ Evaluate synthesized attributes while going up the tree

❏ Can be evaluated using SDTS w/o parse tree

Syntax Directed Translation Scheme (SDTS)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

- [ ] it is natural and easy to evaluate synthesized attributes

parsing stack:

$S_?$	T	T.val=9
$S_?$	\$	-

(state)   (symbol)   (attribute)


```
                T  T.val=9
        ┌───────┼───────────┐
   T  T.val=3   *      F  F.val=3   digit_4  digit_4
   │                   │
   F  F.val=3          digit_3  digit_3
   │
   digit_2  digit_2
```

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

◼ it is natural and easy to evaluate synthesized attributes

parsing stack:

$S_?$	E	E.val=9
$S_?$	\$	-

(state)   (symbol)   (attribute)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

■ it is natural and easy to evaluate synthesized attributes
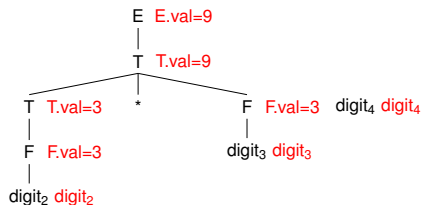

parsing stack:

$S_?$	F	F.val=4
$S_?$	+	-
$S_?$	E	E.val=9
$S_?$	\$	-

(state)   (symbol)   (attribute)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

■ it is natural and easy to evaluate synthesized attributes
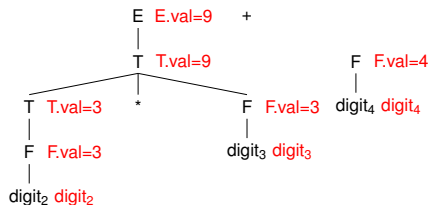
parsing stack:

S?	T	T.val=4
S?	+	-
S?	E	E.val=9
S?	$	-

(state)   (symbol)   (attribute)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

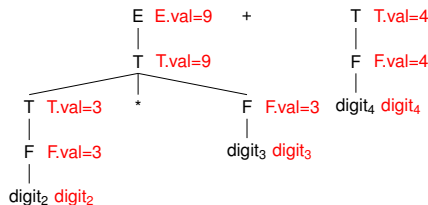☐ it is natural and easy to evaluate synthesized attributes

parsing stack:

$S_?$	E	E.val=13
$S_?$	\$	-

(state)   (symbol)   (attribute)

Tree (left side):

- E   E.val=13
  - E   E.val=9   +   T   T.val=4
    - T   T.val=9
      - T   T.val=3   *   F   F.val=3
        - F   F.val=3
          - $digit_2$ $digit_2$
        - $digit_3$ $digit_3$
    - F   F.val=4
      - $digit_4$ $digit_4$

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

⬜ it is natural and easy to evaluate synthesized attributes
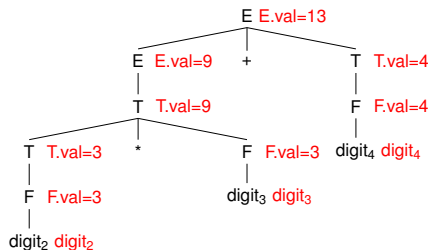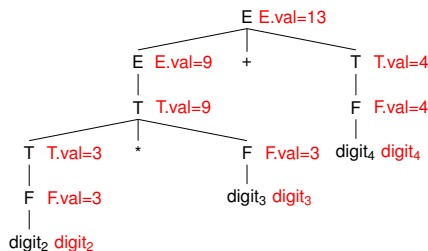

parsing stack:

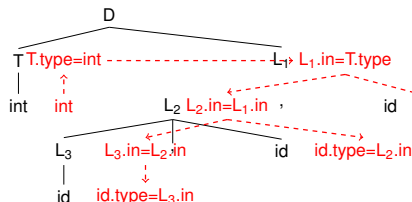|		
$S_?$	E	E.val=13
$S_?$	\$	-

(state) (symbol) (attribute)

⬜ Grammars with only synthesized attributes are called
S-Attributed Grammars

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

◻ it is **not natural** to evaluate inherited attributes

parsing stack:

(state)	(symbol)	(attribute)
$S_?$	\$	-

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

☐ it is **not natural** to evaluate inherited attributes

parsing stack:

$S_?$	\$	-

(state) (symbol) (attribute)

int   ,   id

                 ,   id

     id

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

☐ it is **not natural** to evaluate inherited attributes

parsing stack:

$S_?$	T	T.type=int
$S_?$	\$	-

(state) (symbol) (attribute)

T T.type=int
| ↑
int   int   ,   id

              ,   id

      id

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

■ it is **not natural** to evaluate inherited attributes

parsing stack:

$S_?$	id	**id.type=L_3.in**
$S_?$	T	T.type=int
$S_?$	\$	-

(state) (symbol) (attribute)

T T.type=int
| ↑
int   int   ,   id

                    ,   id

        id   id.type=L_3.in

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

◻ it is **not natural** to evaluate inherited attributes

parsing stack:

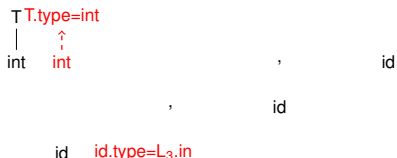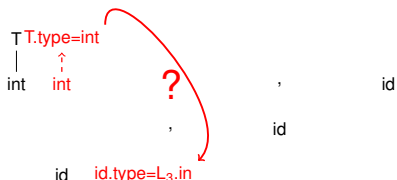|		
$S_?$	id	**id.type=L_3.in**
$S_?$	T	T.type=int
$S_?$	$	-

(state) (symbol) (attribute)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

◻ it is **not natural** to evaluate inherited attributes
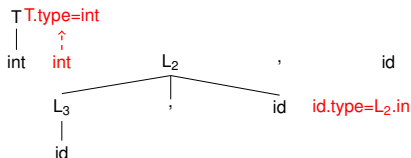
parsing stack:

$S_?$	id	**id.type=L_2.in**
$S_?$	,	
$S_?$	L_3	L_3.in=L_2.in
$S_?$	T	T.type=int
$S_?$	\$	-

(state)   (symbol)   (attribute)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

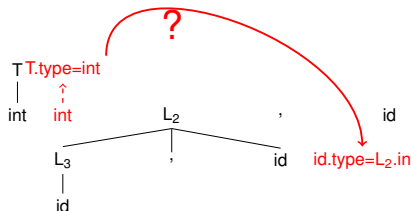◻ it is **not natural** to evaluate inherited attributes

parsing stack:

(state)	(symbol)	(attribute)
$S_?$	id	**id.type=L_2.in**
$S_?$	,	
$S_?$	L_3	L_3.in=L_2.in
$S_?$	T	T.type=int
$S_?$	\$	-

Evaluating Inherited Attributes using LR

❏ **Claim:** Given an L-Attributed grammar, inherited attributes needed for the computation are already on the stack

☞ Recall: What is an L-Attributed grammar?
  ➢ May have synthesized attributes
  ➢ May have inherited attributes but only from:
    ● **Left** sibling attributes
    ● Parent attribute
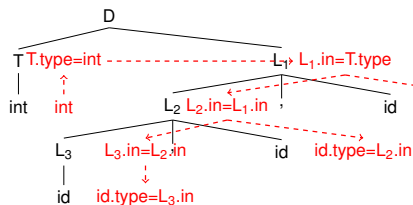
❏ Finding inherited attributes on the stack
  ➢ Left siblings: previously reduced, so already on the stack
  ➢ Parent: not yet reduced, but left siblings of the parent used to compute the parent attribute are on the stack

D → T   L
T → int   {T.type=int}
T → real   {T.type=real}
L → L ,   id   {id.type=stack[top-3].type}
L → id   {id.type=stack[top-1].type}

parsing stack:

$S_?$	$	-

(state) (symbol) (attribute)

D → T   L
T → int   {T.type=int}
T → real   {T.type=real}
L → L ,   id   {id.type=stack[top-3].type}
L → id   {id.type=stack[top-1].type}

int   ,   id

                ,   id

      id

parsing stack:

S$_?$	$	-

(state) (symbol) (attribute)

$D \rightarrow T \quad L$

$T \rightarrow int \quad \{T.type=int\}$

$T \rightarrow real \quad \{T.type=real\}$

$L \rightarrow L \quad , \quad id \quad \{id.type=stack[top-3].type\}$

$L \rightarrow id \quad \{id.type=stack[top-1].type\}$

parsing stack:

T T.type=int
| ↑
int   int   ,   id

                    ,   id

           id

$S_?$	T	T.type=int
$S_?$	\$	-

(state) (symbol) (attribute)

$D \rightarrow T \quad L$

$T \rightarrow$ int $\quad$ {T.type=int}

$T \rightarrow$ real $\quad$ {T.type=real}

$L \rightarrow L$, id $\quad$ {id.type=stack[top-3].type}

$L \rightarrow$ id $\quad$ {id.type=stack[top-1].type}

parsing stack:

T T.type=int

|
int

int

,

id

,

id

id id.type=L_3.in

$S_?$	id	**id.type=stack[top-1]**
$S_?$	T	T.type=int
$S_?$	\$	-

(state) (symbol) (attribute)

D → T   L
T → int   {T.type=int}
T → real   {T.type=real}
L → L ,   id   {id.type=stack[top-3].type}
L → id   {id.type=stack[top-1].type}

T T.type=int
| ↑
int   int   stack[top-1]   ,   id

                          ,   id

            id   id.type=L₃.in

parsing stack:

S₇	id	**id.type=stack[top-1]**
S₇	T	T.type=int
S₇	$	-

(state) (symbol)   (attribute)

$D \rightarrow T \quad L$
$T \rightarrow int \quad \{T.type=int\}$
$T \rightarrow real \quad \{T.type=real\}$
$L \rightarrow L \quad , \quad id \quad \{id.type=stack[top-3].type\}$
$L \rightarrow id \quad \{id.type=stack[top-1].type\}$

parsing stack:

(state)	(symbol)	(attribute)
$S_?$	id	**id.type=stack[top-3]**
$S_?$	,	
$S_?$	L_3	L_3.in=int
$S_?$	T	T.type=int
$S_?$	\$	-

T T.type=int
| int   int   ,   id

L_3 ,   id   id.type=L_2.in

| id   id.type=L_3.in

D → T   L
T → int   {T.type=int}
T → real   {T.type=real}
L → L   ,   id   {id.type=stack[top-3].type}
L → id   {id.type=stack[top-1].type}
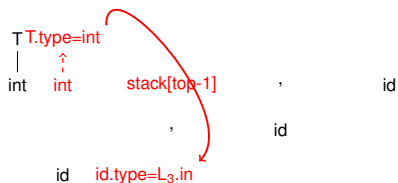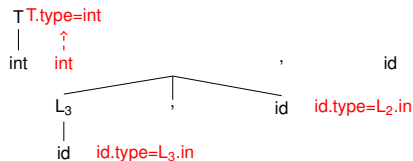
parsing stack:

S?	id	**id.type=stack[top-3]**
S?	,	
S?	L_3	L_3.in=int
S?	T	T.type=int
S?	$	-

(state)   (symbol)   (attribute)

Marker

❏ Given the following SDD, where $|\alpha| \mathrel{!}= |\beta|$

A$\rightarrow$ X α Y | X β Y

Y$\rightarrow \gamma$ {... = f(X.s)}

❏ Problem: cannot generate stack location for X.s since X is at different relative stack locations from Y

❏ Solution: introduce *markers* M_1 and M_2 that are at the same relative stack locations from Y

A$\rightarrow$ X α M_1 Y | X β M_2 Y

Y$\rightarrow \gamma$ {... = f(M_{12}.s)}

$M_1 \rightarrow \varepsilon$ {M_1.s = X.s}

$M_2 \rightarrow \varepsilon$ {M_2.s = X.s}

(M_{12} = the stack location of M_1 or M_2, which are identical)

❏ A marker intuitively marks a stack location that is equidistant from the reduced non-terminal

Example

☐ When is a marker necessary and how is it added?

Example 1:

$S \rightarrow a\ A\ \{\ C.i = A.s\ \}\ C$
$S \rightarrow b\ A\ B\ \{\ C.i = A.s\ \}\ C$
$C \rightarrow c\ \{\ C.s = f(C.i)\ \}$

Solution:

$S \rightarrow a\ A\ \{\ C.i = A.s\ \}\ C$
$S \rightarrow b\ A\ B\ \{\ M.i=A.s\ \}\ M\ \{\ C.i = M.s\ \}\ C$
$C \rightarrow c\ \{\ C.s = f(C.i)\ \}$
$M \rightarrow \varepsilon\ \{\ M.s = M.i\ \}$

That is:

$S \rightarrow a\ A\ C$
$S \rightarrow b\ A\ B\ M\ C$
$C \rightarrow c\ \{\ C.s = f(stack[top-1])\ \}$
$M \rightarrow \varepsilon\ \{\ M.s = stack[top-2]\ \}$

When and how to add a marker

1. Identify the stack offset(s) to find the desired attribute
2. If stack offsets are different, add a marker
3. Add marker where it would result in uniform stack offsets

Example:

$S \rightarrow a \ A \ B \ C \ E \ D$
$S \rightarrow b \ A \ F \ B \ C \ F \ D$
$C \rightarrow c \ \{/* \ C.s = f(A.s) \ */\}$
$D \rightarrow d \ \{/* \ D.s = f(B.s) \ */\}$

Answer

S → a A B C E D
S → b A D M B C F D
C → c {/* C.s = f(stack[top-2]) */}
D → d {/* D.s = f(stack[top-3]) */}
M → ε {/* M.s = f(stack[top-2]) */}

❏ Regarding C.s, from stack[top-2], and stack[top-3]
    .... add a Marker

❏ Regarding D.s, always from stack[top-2]
    ... no need to add

How about Top-Down Parsing?

Translation Scheme for Top-Down Parsing

❏ Recursive Descent Parsers: Straightforward
  ➢ Synthesized Attribute
    ● Say function for non-terminal returns synthesized attribute
    ● Compute attribute from children function call return values
  ➢ Inherited Attribute
    ● Pass as argument to function call for inheriting non-terminal
    ● Left sibling attributes: left sibling calls already complete
    ● Parent attributes: passed in as arguments to parent function

❏ How about table-driven LL parsers?

Translation Scheme for LL Parsing

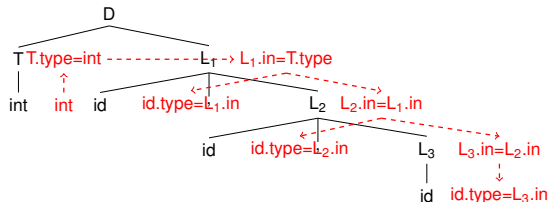■ it is natural to evaluate inherited attributes

$D \rightarrow T$ {L.in=T.type} L
$T \rightarrow$ int {T.type=int}
$T \rightarrow$ real {T.type=real}
$L \rightarrow$ {id.type=L.in} id , {L_1.in=L.in} L_1
$L \rightarrow$ {id.type=L.in} id

parsing stack:

(symbol)   (attribute)

Translation Scheme for LL Parsing

◻ it is natural to evaluate inherited attributes

$D \rightarrow T$ {L.in=T.type} L
$T \rightarrow int$ {T.type=int}
$T \rightarrow real$ {T.type=real}
$L \rightarrow$ {id.type=L.in} id , {L_1.in=L.in} L_1
$L \rightarrow$ {id.type=L.in} id

D

parsing stack:

D	

(symbol)   (attribute)

Translation Scheme for LL Parsing

⬛ it is natural to evaluate inherited attributes

D → T {L.in=T.type} L
T → int {T.type=int}
T → real {T.type=real}
L → {id.type=L.in} id , {L_1.in=L.in} L_1
L → {id.type=L.in} id

parsing stack:

T	T.type=int
	{L_1.in=T.type}
L	L_1.in=???

(symbol)   (attribute)

D
T.type=int L_1.in=T.type

Translation Scheme for LL Parsing

◻ it is natural to evaluate inherited attributes

$D \rightarrow T$ {L.in=T.type} L
$T \rightarrow int$ {T.type=int}
$T \rightarrow real$ {T.type=real}
$L \rightarrow$ {id.type=L.in} id , {L_1.in=L.in} L_1
$L \rightarrow$ {id.type=L.in} id

D

T T.type=int ---------- L_T -> L_1.in=T.type

parsing stack:

	{L_1.in=int}
L	L_1.in=???

(symbol)   (attribute)

Translation Scheme for LL Parsing

■ it is natural to evaluate inherited attributes

$D \rightarrow T$ {L.in=T.type} L
$T \rightarrow$ int {T.type=int}
$T \rightarrow$ real {T.type=real}
$L \rightarrow$ {id.type=L.in} id , {L_1.in=L.in} L_1
$L \rightarrow$ {id.type=L.in} id

D
T T.type=int - - - - - - - - - - L_T -› L_1.in=T.type

parsing stack:

	L_1.in=int

(symbol)   (attribute)

Translation Scheme for LL Parsing

■ it is natural to evaluate inherited attributes

$D \rightarrow T \ \{L.in=T.type\} \ L$
$T \rightarrow int \ \{T.type=int\}$
$T \rightarrow real \ \{T.type=real\}$
$L \rightarrow \{id.type=L.in\} \ id \ , \ \{L_1.in=L.in\} \ L_1$
$L \rightarrow \{id.type=L.in\} \ id$
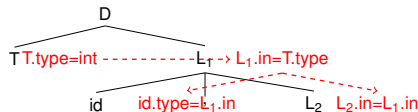
parsing stack:

	$\{id.type=L_1.in\}$
id	$id.type=???$
,	
	$\{L_2.in=L_1.in\}$
L_2	$L_2.in=???$

(symbol)   (attribute)

Translation Scheme for LL Parsing

◼ it is natural to evaluate inherited attributes

$D \rightarrow T$ {L.in=T.type} L
$T \rightarrow int$ {T.type=int}
$T \rightarrow real$ {T.type=real}
$L \rightarrow$ {id.type=L.in} id , {L_1.in=L.in} L_1
$L \rightarrow$ {id.type=L.in} id

parsing stack:

	{id.type=int}
id	id.type=???
,	
	{L_2.in=int}
L_2	L_2.in=???

(symbol)   (attribute)

D
T T.type=int - - - - - - - - L$_T$ -> L_1.in=T.type
id   id.type=L$_1$.in L_2 L_2.in=L_1.in

Translation Scheme for LL Parsing
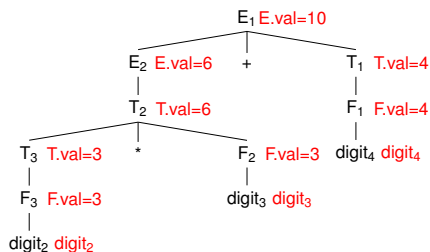
- it is natural to evaluate inherited attributes

$D \to T$ {L.in=T.type} L
$T \to$ int {T.type=int}
$T \to$ real {T.type=real}
$L \to$ {id.type=L.in} id , {L_1.in=L.in} L_1
$L \to$ {id.type=L.in} id

parsing stack:

	{id.type=int}
id	id.type=???
,	
	{L_2.in=int}
L_2	L_2.in=???

(symbol)   (attribute)

- Semantic actions on the stack are called **action-records**.

Translation Scheme for LL Parsing

■ it is **not natural** to evaluate synthesized attributes

parsing stack:

(symbol)	(attribute)

Translation Scheme for LL Parsing

□ it is **not natural** to evaluate synthesized attributes

E_1

parsing stack:

E_1	

(symbol)   (attribute)

Translation Scheme for LL Parsing

☐ it is **not natural** to evaluate synthesized attributes

E_1 E.val=?

parsing stack:

E_1	E_1.val=?

(symbol) (attribute)

Translation Scheme for LL Parsing

■ it is **not natural** to evaluate synthesized attributes

E_1 E.val=?

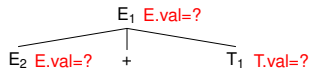E_2 E.val=?   + T_1 T.val=?

parsing stack:

E_2	E_2.val=?
+	
T_1	T_1.val=?

(symbol)   (attribute)

Translation Scheme for LL Parsing
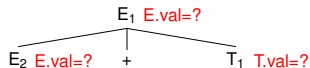
■ it is **not natural** to evaluate synthesized attributes

parsing stack:

E_2	$E_2.val=?$
+	
T_1	$T_1.val=?$

(symbol)   (attribute)

Translation Scheme for LL Parsing

□ it is **not natural** to evaluate synthesized attributes

E_1

parsing stack:
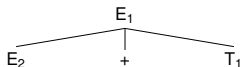
E_1	
E_1.val	???
(symbol)	(attribute)

Translation Scheme for LL Parsing

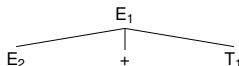◻ it is **not natural** to evaluate synthesized attributes

parsing stack:

(symbol)	(attribute)
E_2	
E_2.val	???
+	
T_1	
T_1.val	???
E_1.val	E_2.val + T_1.val

Translation Scheme for LL Parsing

❑ it is **not natural** to evaluate synthesized attributes

E_1

E_2 + T_1

parsing stack:

E_2	
E_2.val	???
+	
T_1	
T_1.val	???
E_1.val	E_2.val + T_1.val

(symbol)   (attribute)

❑ Synthesized attributes on the stack: **synthesize-records**.
(Inserted below non-terminal with synthesized attribute)

❑ In synthesize-record E_1.val = E_2.val + T_1.val,
E_2.val and T_1.val are place holders for pending values.
(Updated when records E_2.val and T_1.val are popped.)