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 Inference

- "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 Inference

- "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 cannot be done during parsing

- Context Free Grammars (CFGs) cannot recognize bindings
  - Every use of a name needs to be bound to the declaration.
  - Name can refer to a variable, function, class, ...
  - Names are called **symbols** in semantic analysis
- To do bindings, a CFG must recognize this language:

$$\{\alpha c \alpha | \alpha \in (a|b)^*\}$$

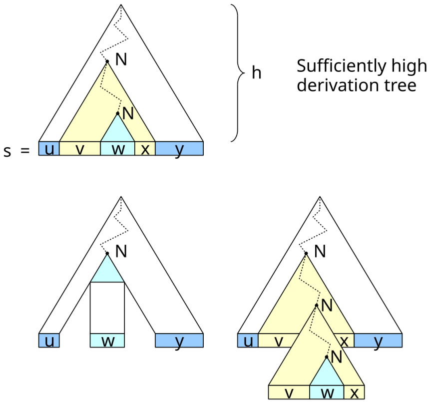
  The 1st α represents the declaration,
  The 2nd α represents a use.
- Above language is a Context Sensitive Language

Why is $\{\alpha c \alpha | \alpha \in (a|b)^*\}$ not a CFG?

- We will base our proof on the **pumping lemma** for CFGs.
- **Pumping lemma**: a theorem about strings in a grammar
  - "lemma": a mathematical term for a theorem
  - "pumping": for a sufficiently long enough string,
    a substring exists within that string that can be "pumped"
    (repeated 0 or more times and still be in the language).
- For example, for the Regular Language 0(0|1)*0:
  - A string longer than 2 will look like 000, 010, 0100, ...
  - Let's take "010". Here, substring "1" can be pumped.
  - ("00", "010", "0110", "01110" are all in the language)
- Pumping Lemma applies to CFGs as well.

Pumping Lemma for CFGs

- For a sufficiently long string s derived from a CFG, s can be written as s = uvwxy (u,v,w,x,y are substrings)
  - Where v and x can be pumped and $|vx| \geq 1$.



Generating uv^0wx^0y Generating uv^2wx^2y

$\{\alpha c\alpha | \alpha \in (a|b)^*\}$ is not a CFG

- Let's say s = uvwxy is a sufficiently long string in language $\{\alpha c\alpha | \alpha \in (a|b)^*\}$,

  where v, x can be pumped and $|vx| \geq 1$.

1. The substring vwx must bisect $\alpha c\alpha$.
   - If vwx is contained in 1st α (or mostly contained), if we pump v and x 0 times, 1st α gets shorter than 2nd α.
   - string is no longer in $\{\alpha c\alpha | \alpha \in (a|b)^*\}$. Contradiction.
   - The same applies to when vwx is contained in 2nd α.
2. Even when vwx bisects $\alpha c\alpha$, pumping fails.
   - Let string s' be the result of pumping v and x 0 times.
   - Let's say s' = $\alpha_1 c\alpha_2$, where α_1 and α_2 are shortened versions of the 1st and 2nd αs.
   - While $|\alpha_1| = |\alpha_2|$, there exist α_1 and α_2 such that $\alpha_1 != \alpha_2$.
   - E.g. s = abcab, and vwx = bca where v = b and x = a. Then, α_1 becomes "a" and α_2 becomes "b". Contradiction.

Semantic analysis does binding and type checking

- Semantic analysis performs binding
  - Since CFGs cannot recognize bindings, as we just proved
  - Done by traversing parse tree produced by syntax analysis
  - Definitions are stored in data structure called **symbol table**
  - Uses are bound to entries in the symbol table
- Semantic analysis performs type checking
  - Infer 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
  - Infer what "$a.foo()$" means:
    - If object *a* is an instance of class *A*, call A.foo()
    - If object *a* is an instance of class *B*, call B.foo()
  - Infer what "$a[i][j]$" means:
    - Offset from *a* calculated based on type and dimensions

Semantic analysis also performs semantic checks

- All symbol uses have a corresponding declaration;
- All operations are type legal;
- Inheritance relationships are correct;
- A class is defined only once;
- A method in a class is defined only once;
- ...

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: the association of a use of a symbol to the declaration of that symbol
  - 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 depends on the program text, not run-time behavior (also known as lexical scoping)
  - C/C++, Java, Objective-C
- Rule: Refer to the closest enclosing declaration

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

Static Scope

- Static scope depends on the program text, not run-time behavior (also known as lexical scoping)
  - C/C++, Java, Objective-C
- Rule: Refer to the closest enclosing declaration

```
void foo()
{
    char x;

  ...
  {
      int x;

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

Dynamic Scope

- Dynamic scoping depends on bindings formed during the execution of the program
  - LISP, Scheme, Perl
- Rule: Refer to the closest binding in the current execution

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

Dynamic Scope

- Dynamic scoping depends on bindings formed during the execution of the program
  - LISP, Scheme, Perl
- Rule: Refer to the closest binding in the current 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

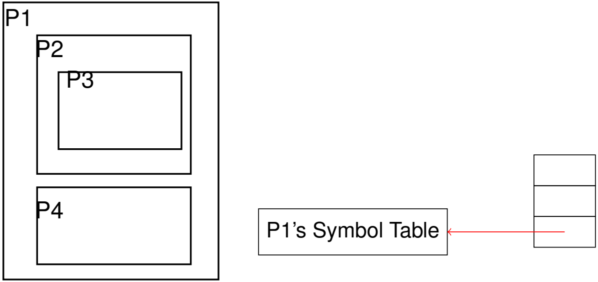
Adding Scope Information to the Symbol Table

- 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() {...} ... }
X v;
call v.f1();
```

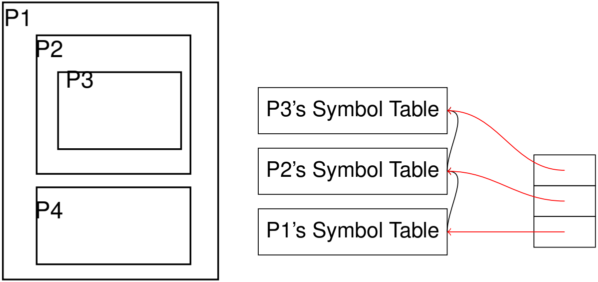
  - Without deleting symbols, how are scoping rules enforced?
    - ☞ Keep a list of all scopes in the entire program
    - ☞ Keep a stack of active scopes at a given point

Symbol Table with Multiple Scopes



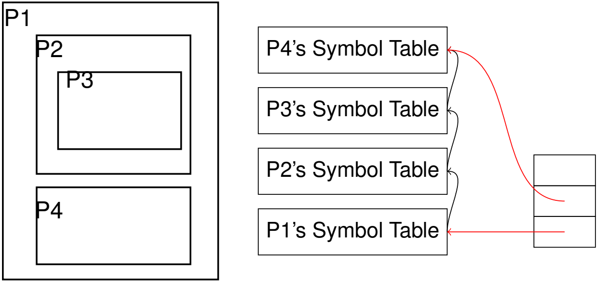
- For nested scopes,
  - Search from top of the active symbol table stack
  - Remove pointer to symbol table when exiting its scope

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

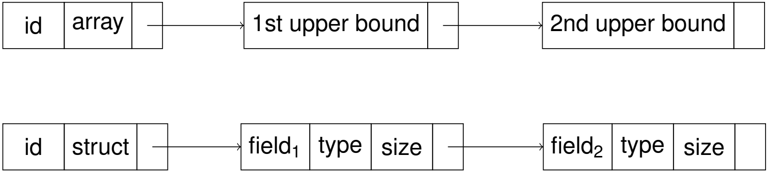
  - String — the name of identifier
  - Kind — variable, parameter, function, class, ...
- Attributes vary with the kind of symbol
  - variable $\rightarrow$ type, address in memory
  - function $\rightarrow$ return type, parameter types, address
- Vary with the language
  - Fortran's array $\rightarrow$ type, dimension, dimension size
    real A(5) /* dimension required for static allocation */
  - C's array $\rightarrow$ 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

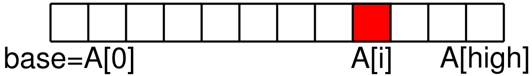


Example application of Type to an operator: Array index operator

Addressing Array Elements

int A[0..high];

A[i] ++;



- 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

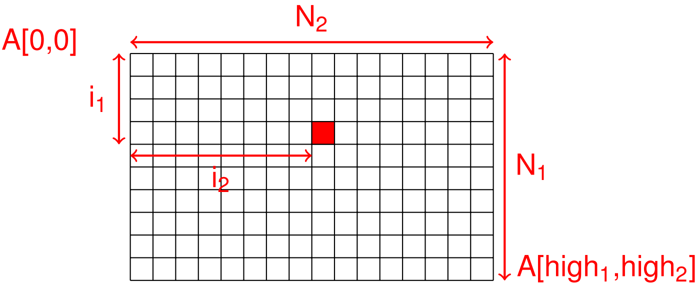
  offset(A[i]) = i * width

Multi-dimensional Arrays

- Layout n-dimension items in 1-dimension memory

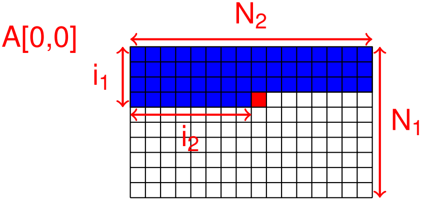
  int A[N_1][N_2]; /* int A[0..$high_1$][0..$high_2$]; */

  A[i_1][i_2] ++;



Row Major

Row major — store row by row

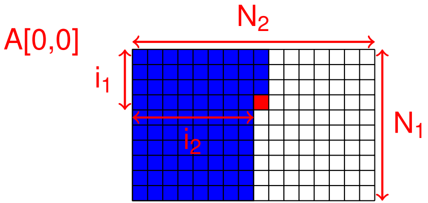


- Offset inclues all the “blue” items before $A[i_1,i_2]$

$\text{offset}(A[i_1,i_2]) = (i_1 * N_2 + i_2) * \text{width}$
$= i_1 * N_2 * \text{width} + i_2 * \text{width}$
$= \text{offset}(A[i_1]) * N_2 + i_2 * \text{width}$

Column Major

Column major — store column by column



- Offset inclues all the “blue” items before $A[i_1,i_2]$

$\text{offset}(A[i_1,i_2]) = (i_2 * N_1 + i_1)*\text{width}$
$= i_2 * N_1 * \text{width} + i_1 * \text{width}$
$= i_2*N_1*\text{width} + \text{offset}(A[i_1])$

Generalized Row/Column Major

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

- Row major
  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
  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
- $A_2 = (i_1 * N_2 + i_2) * \text{width}$ (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, Why and When

- What is a type?
  Type = a set of values + a set of operations on these values
- What is type checking?
  Verifying and enforcing type consistency
  - Only legal values are assigned to a type
  - Only legal operations are performed on a type
- Why is compile-time type checking desirable?
  - Type errors easier to debug than malfunctioning programs
  - Dynamic type checking when static checking infeasible
    - E.g. Java null checks and array bounds checks
    - E.g. C++/Java downcasting to a subclass

Static vs. Dynamic Typing

- Statically typed: C/C++, Java ☞Our discussion
  - Types are explicitly declared or can be inferred from code
  - E.g. int x; /* type of x is int */
  - Efficient code since runtime type checks are not needed
- Dynamically typed: Python, JavaScript, PHP
  - Type is a runtime property decided only during execution
  - E.g. var x; /* type of x is undecided */
  - Type of x changes depending on the type of value it holds
  - More memory since every variable now needs a "type tag"
  - Inefficient code due to runtime 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

$$\frac{\textbf{Precondition}}{\textbf{Conclusion}}$$

  - 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

$$\frac{\textbf{Precondition}_1, \ldots, \textbf{Precondition}_n}{\textbf{Conclusion}}$$

  - 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

$$\frac{\begin{array}{c} e_1 : \text{int} \\ e_2 : \text{int} \end{array}}{e_1 + e_2 : \text{int}}$$

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

Two Simple Rules

[Constant]

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

[Add operation]

$$\frac{\begin{array}{c}\mathbf{e_1: int}\\ \mathbf{e_2: int}\end{array}}{\mathbf{e_1+e_2:int}}$$

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

$$\frac{\dfrac{\textbf{10 is an integer}}{\textbf{10: int}} \qquad \dfrac{\textbf{20 is an integer}}{\textbf{20: int}}}{\textbf{10+20:int}}$$

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

More Rules

[New]

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

[Not]

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

- However,

  [Var?]

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

  - 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"

$$\frac{\textbf{i is an intger}}{\textbf{O i: int}} \qquad \frac{\textbf{O e1: int} \quad \textbf{O e2: int}}{\textbf{O e1+e2: int}} \qquad \frac{\textbf{O(x): T}}{\textbf{O x: T}}$$

– "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]

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

$O[T_0/x]$ means, O is modified to 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)]

$$\frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- 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 $\leq$ 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]

$$\frac{\begin{array}{c} \mathbf{O}\ \mathbf{e}_0 : \mathbf{T} \\ \mathbf{T} \leq \mathbf{T}_0 \\ \mathbf{O}[\mathbf{T}_0/\mathbf{x}]\ \mathbf{e}_1 : \mathbf{T}_1 \end{array}}{\mathbf{O}\ \mathbf{let}\ \mathbf{x} : \mathbf{T}_0 \leftarrow \mathbf{e}_0\ \mathbf{in}\ \mathbf{e}_1 : \mathbf{T}_1}$$

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

Assignment

- A correct but too strict rule

[Assignment]

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

- The rule does not allow the below code

  class C inherits P { only_in_C() { ... } }
  x ← y ← new C
  x.only_in_C()

Assignment

- An improved rule

$$\text{[Assignment]}\quad \frac{\begin{array}{c}\mathbf{O(id) = T_0} \\ \mathbf{O \vdash e_1 : T_1} \\ \mathbf{T_1 \leq T_0}\end{array}}{\mathbf{O \vdash id \leftarrow e_1 : T_1}}$$

  - The rule now does allow the below code

```
class C inherits P { only_in_C() { ... } }
x ← y ← 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]

$$\frac{\mathbf{O}\ e_0 : \mathbf{Bool} \quad \mathbf{O}\ e_1 : T_1 \quad \mathbf{O}\ e_2 : T_2}{\mathbf{O}\ \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2\ \textbf{fi} : \mathrm{lub}(T_1, T_2)}$$

- 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)]

$$\frac{\mathbf{O}\ \mathbf{e_0 : T} \qquad \mathbf{T} \leq \mathbf{T_0} \qquad \mathbf{O}\ \mathbf{e_1 : T_1}}{\mathbf{O}\ \mathbf{let\ x{:}\ T_0} \leftarrow \mathbf{e_0\ in\ e_1 : T_1}}$$

  - How is it different from the the correct rule?
  - The following 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)]

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T_0 \leq T \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the the correct rule?
- The following bad program passes the check

```
class B inherits A { only_in_B() { ... } }
let x: B ← new A in x.only_in_B()
```

Wrong Declaration Rule (case 3)

- Consider a hypothetical let rule

[Wrong Declaration with initialization (case 3)]

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ O[T/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the the correct rule?
- The following bad program passes the check

  class B inherits A { only_in_B() { ... } }

  let x: A $\leftarrow$ new B in x.only_in_B()

Discussion

- Type rules have to be very carefully constructed
- Virtually any change in a rule either
  - makes the type system unsound
    (bad programs are accepted as well typed)
  - or, makes the type system less usable
    (good programs are rejected)

- But some good programs will be rejected anyway
  - .... what is a “good” program ?

Discussion

- Type rules have to be very carefully constructed
- Virtually any change in a rule either
  - makes the type system unsound
    (bad programs are accepted as well typed)
  - or, makes the type system less usable
    (good programs are rejected)

- But some good programs will be rejected anyway
  - .... what is a “good” program ?
  - Good program: A program where all operations performed on all values are type consistent **at runtime**

Discussion

- Type rules have to be very carefully constructed
- Virtually any change in a rule either
  - makes the type system unsound
    (bad programs are accepted as well typed)
  - or, makes the type system less usable
    (good programs are rejected)

- But some good programs will be rejected anyway
  - .... what is a “good” program ?
  - Good program: A program where all operations performed on all values are type consistent **at runtime**
  - Impossible to express all runtime behavior in a type system
    - E.g. Type of if-then-else is LUB of two types, a conservative estimate of runtime behavior

Designing a Good Type Checking System

- Type system has two conflicting design goals
  - Give flexibility to the programmer (so that she can write a "good" program within the boundaries of the type system)
  - Prevent type-checked programs from "going wrong"
- Should allow maximum flexibility while guaranteeing safety
  - An example

```
class Count {
  int i = 0;
  Count inc() { i=i+1; return this; }
}
class Stock inherits Count { ... }
class Main {
  Stock a ← (new Stock).inc();
}
```

What Went Wrong?

- What is (new Stock).inc()'s type?
  - Dynamic type — Stock
  - Static type — Count

  - The type checker "loooses" the type information
  - This makes inheriting inc() useless
    - Do we really want to redefine inc() for each subclass returning the correct type?
- SELF_TYPE to the rescue

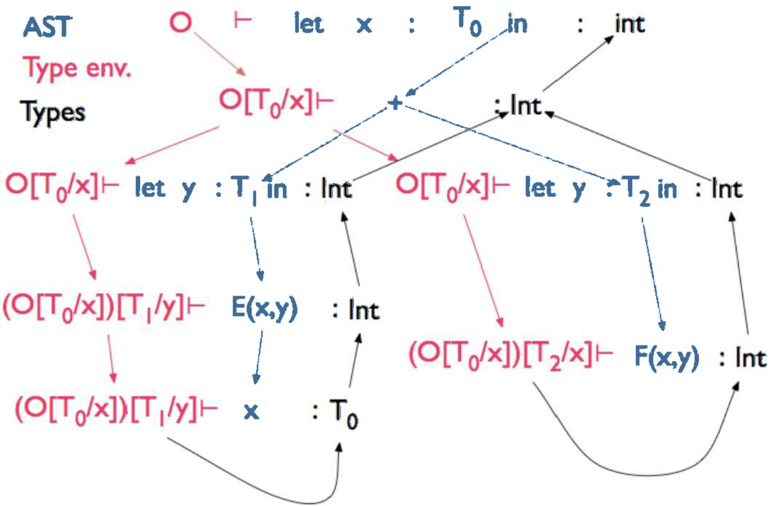
SELF_TYPE to the Rescue

- What is SELF_TYPE?
  - inc() returns “self” instead of “Count” type
  - Self could be Count or any subclass of Count, depending on reference type
- SELF_TYPE is a static type
  - Type violations can still be detected at compile time
  - Expresses runtime behavior accurately w/o undue burden to the programmer
- In practice
  - C++: made possible by language extension using templates
  - Java: not allowed because there are no templates

Can Static Type Checking ever be Perfect?

- Many examples where correct programs are disallowed (besides SELF_TYPE)
  - Why C++ programmers are still forced to downcast (Type of values at runtime are known by programmer but no way to express it in type system)
  - Fundamentally undecidable whether type system is adhered to at runtime
- Solution?
  - Some argue for dynamic type checking instead
    - Philosophy: Maximum expressivity for the programmer
    - Good for scripting languages where expressivity is king
  - Others argue for more expressive static checking rules
    - Philosophy: Too much expressivity is not good for you
    - Good for mission critical code that should never fail
    - Good for performance critical code

Implementing Type Checking on AST



Syntax Directed Translation

What is Syntax Directed Translation?

- To drive semantic analysis tasks based on the language's syntactic structure

- What is meant by semantic analysis tasks?
  - Generate AST (abstract syntax tree)
  - Check type errors
  - Generate intermediate representation (IR)
- What is meant by syntactic structure?
  - Structure of program given by context free grammar (CFG)
  - Structure of the parse tree generated by the parser

How is Syntax Directed Translation Performed?

- How?
  - Attach **attributes** to grammar symbols/parse tree
  - Evaluate attribute values using **semantic actions**

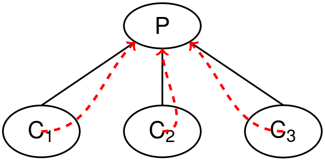
- We already did some of this in Project 2:
  - Attached attributes to grammar symbols
    - **tptr** — “tree pointer” of a non-terminal symbol
      By the time **program.tptr** is evaluated, the parse tree is built
  - Evaluated attributes using semantic rules (actions)
    - { ... $$=makeTree(ProgramOp, leftChild, rightChild); ... }

Attributes?

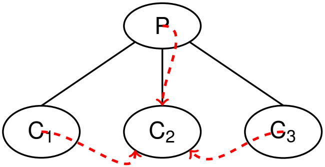
- Attributes can represent anything depending on task
  - A string
  - A type
  - A number
  - A memory location
- An **attribute grammar** is a grammar augmented by associating attributes with each grammar symbol that describes its properties

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
  - Each grammar rule is associated with a **semantic action**

$L \rightarrow E$	{ 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

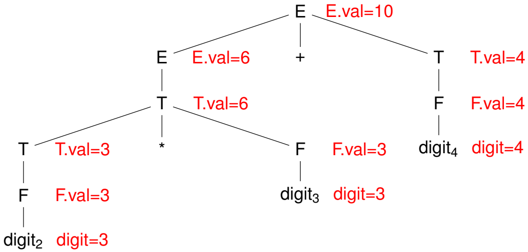
- Example:
  - T — synthesized attribute “type”
  - L — has inherited attribute “in”

$D \rightarrow T\ L$	{ 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, addtype (id.entry, L.in) }
$L \rightarrow$ id	{ addtype(id.entry, L.in) }

- ➢ We can use *inherited attributes* to track **type** information
- ➢ We can use *inherited attributes* to track whether an identifier appear on the left or right side of an assignment operator “:=” (e.g. a := a +1)

Attribute Parse Tree

- Parse tree showing values of attributes
  - Parse tree annotated or decorated with attributes
  - Attributes computed at each node
- Properties of attribute parse tree:
  - Terminal symbols — have synthesized attributes only, which are usually provided by the lexical analyzer
  - Start symbol — does not have any inherited attributes



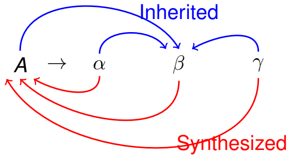
Two Aspects to Syntax Directed Translation

- Syntax Directed Definitions (SDD)
  - Set of **semantic rules** attached to each production
  - Semantic rules define values for attributes
  - Specification rather than implementation

- Syntax Directed Ttranslation Scheme (SDTS)
  - Semantic actions are **implementations of semantic rules**
  - Grammar with **semantic actions** embedded within RHS of productions (can be anywhere)
  - Semantic actions are fragments of code executed “at that point” in the RHS
    - Top-down: Right after previous symbol has been consumed
    - Bottom-up: Right after previous symbol has been pushed to the stack (when the ’dot’ reaches the action)

Syntax Directed Definition (SDD)

- Attribute grammar



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 attributed 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

Syntax Directed Translation Scheme (SDTS)

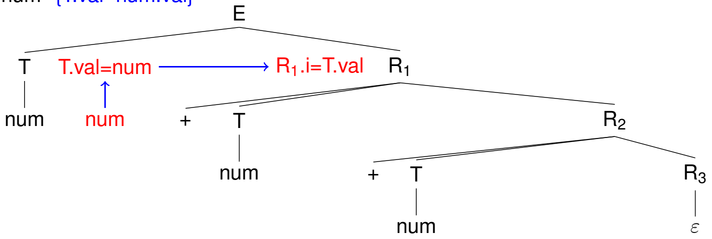
$E \rightarrow T \quad R$
$R \rightarrow + \quad T \quad R$
$R \rightarrow - \quad T \quad R$
$R \rightarrow \varepsilon$
$T \rightarrow (E)$
$T \rightarrow$ num

- 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

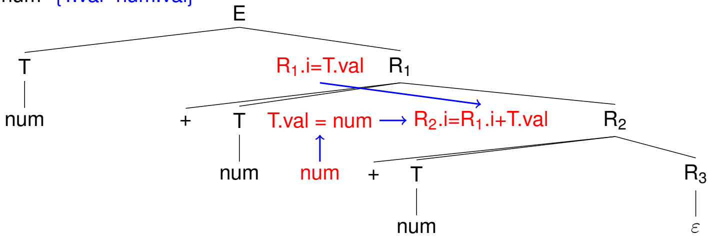
$E \rightarrow T$ {R.i=T.val} R {E.val=R.s}
$R \rightarrow +$ T {R_1.i=R.i+T.val} R_1 {R.s=R_1.s}
$R \rightarrow -$ T {R_1.i=R.i-T.val} R_1 {R.s=R_1.s}
$R \rightarrow \varepsilon$ {R.s=R.i}
$T \rightarrow$ (E) {T.val=E.val}
$T \rightarrow$ num {T.val=num.val}



Syntax Directed Translation Scheme (SDTS)

- Evaluating attributes using SDTS

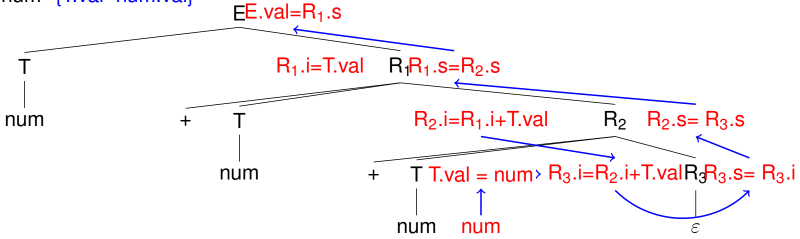
$E \rightarrow T$ {R.i=T.val} R {E.val=R.s}
$R \rightarrow +\ T$ {R_1.i=R.i+T.val} R_1 {R.s=R_1.s}
$R \rightarrow -\ T$ {R_1.i=R.i-T.val} R_1 {R.s=R_1.s}
$R \rightarrow \varepsilon$ {R.s=R.i}
$T \rightarrow (\ E\ )$ {T.val=E.val}
$T \rightarrow$ num {T.val=num.val}



Syntax Directed Translation Scheme (SDTS)

- Evaluating attributes using SDTS

E $\rightarrow$ 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 $\rightarrow$ ε {R.s=R.i}
T $\rightarrow$ (E) {T.val=E.val}
T $\rightarrow$ num {T.val=num.val}

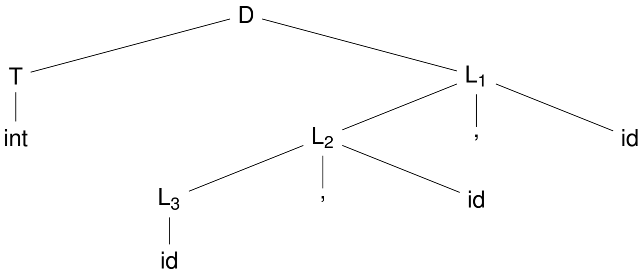


SDD Implementation using Parse Trees

- Alternative to using Syntax Directed Translation Scheme
- Goal: create an **annotated parse tree** from the given parse tree
  - Annotated parse tree: tree annotated with attribute values
  - Traverse in a certain order and evaluate semantic rules at each node
  - Traversal order can be arbitrary as long as it adhers to dependency relationships

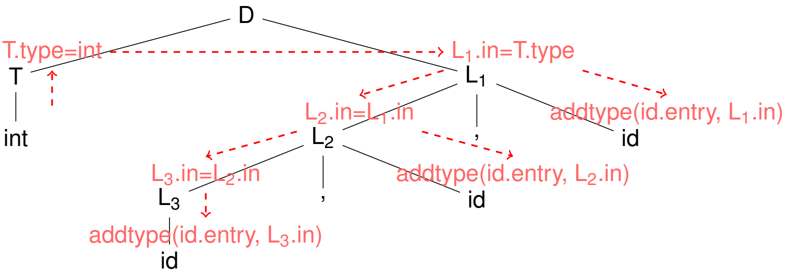
Dependency Graph

- Directed graph where edges are dependency relationships between attributes
  - Needs to be acyclic such that there exists a traversal order for evaluation
    - i.e. all necessary information must be ready when evaluating an attribute at a node



Dependency Graph

- Directed graph where edges are dependency relationships between attributes
  - Needs to be acyclic such that there exists a traversal order for evaluation
    - i.e. all necessary information must be ready when evaluating an attribute at a node



SDD Implementation using SDTS

- Tree-based evaluation works for all SDDs unless there is a dependency cycle
  - But involves more work since parse tree must be built initially
  - And the question still remains, how do you build the parse tree itself?
- Is it possible to perform evaluation while parsing?
  - Embed semantic actions in grammar using SDTS
  - What are some potential problems?
    - Parser may not have even "seen" some nodes yet
    - Some dependencies may not exist at time of evaluation
  - Different parsing schemes see nodes in different orders
    - Top-down parsing — LL(k) parsing
    - Bottom-up parsing — LR(k) parsing
- For certain classes of SDDs, using SDTS is feasible
  - if dependencies of SDD are amenable to parse order
  - In other words, an L-Attributed Grammar

Left-Attributed Grammar

- A syntax directed translation 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 symbols to its left i.e. $X_1 ... X_{j-1}$
    - and/or depends on inherited attributes of A

Left-Attributed Grammar

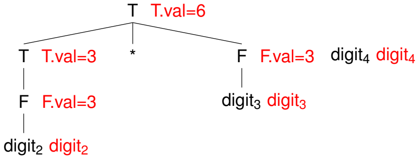
- An L-Attributed grammar
  - may have synthesized attributes
  - may have inherited attributes but only from left sibling attributes or inherited attributes of the parent
- Evaluation order
  - Left-to-right depth-first traversal of the parse tree
    - Order for both top-down and bottom-up parsers
  - 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 Implementation

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

- it is natural and easy to evaluate synthesized attributes



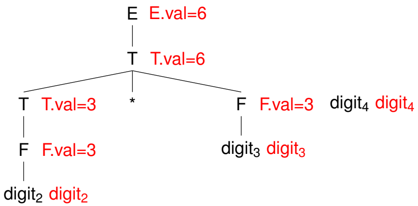
parsing stack:

$S_?$	T	T.val=6
$S_?$	$	-
(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



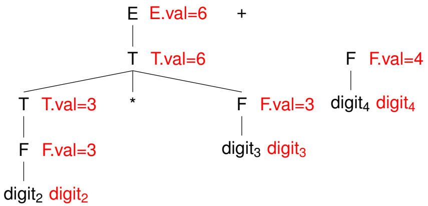
parsing stack:

$S_?$	E	E.val=6
$S_?$	$	-
(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



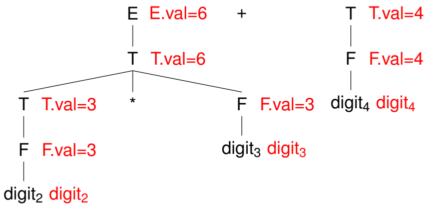
parsing stack:

(state)	(symbol)	(attribute)
$S_?$	F	F.val=4
$S_?$	+	-
$S_?$	E	E.val=6
$S_?$	$	-

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

- it is natural and easy to evaluate synthesized attributes



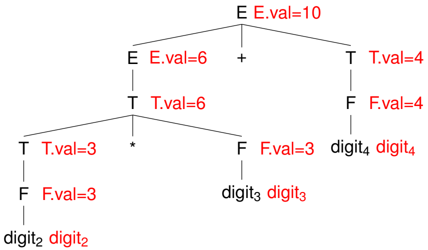
parsing stack:

$S_?$	T	T.val=4
$S_?$	+	-
$S_?$	E	E.val=6
$S_?$	$	-
(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



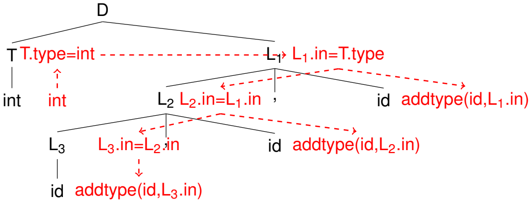
parsing stack:

(state)	(symbol)	(attribute)
$S_?$	E	E.val=10
$S_?$	\$	-

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)

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

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

int , id

, id

id

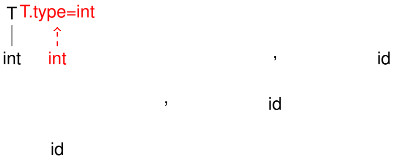
parsing stack:

$S_?$	$	-
(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?	T	T.type=int
S?	$	-

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

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

T T.type=int

int int , id

, id

id addtype(id,L_3.in)

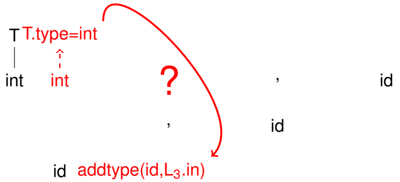
parsing stack:

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

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

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



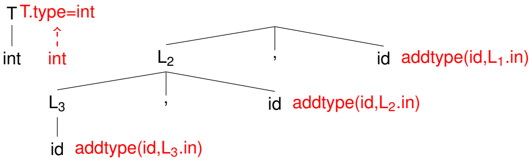
parsing stack:

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

Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

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



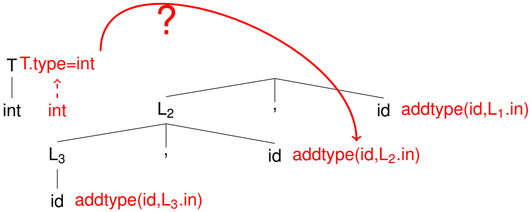
parsing stack:

$S_?$	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_3.in**
$S_?$	T	T.type=int
$S_?$	$	-

(state) (symbol) (attribute)

Evaluating Inherited Attributes using LR

- Recall
- Only applies to L-Attributed grammars
  - ☞ What is L-attributed grammar?

- **Claim:** the information is in the stack, we just do not know the exact location
- **Solution:** let us hack the stack to find the location

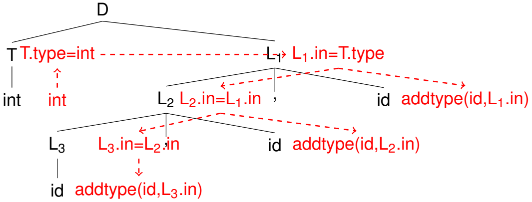
D $\rightarrow$ T L

T $\rightarrow$ int {stack[top]=integer}

T $\rightarrow$ real {stack[top]=real}

L $\rightarrow$ L , id {addtype(stack[top],stack[top-3])}

L $\rightarrow$ id {addtype(stack[top],stack[top-1])}



parsing stack:

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

$D \rightarrow T \quad L$
$T \rightarrow$ int {stack[top]=integer}
$T \rightarrow$ real {stack[top]=real}
$L \rightarrow L$, id {addtype(stack[top],stack[top-3])}
$L \rightarrow$ id {addtype(stack[top],stack[top-1])}

int , id

, id

id

parsing stack:

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

$D \rightarrow T \quad L$
$T \rightarrow$ int {stack[top]=integer}
$T \rightarrow$ real {stack[top]=real}
$L \rightarrow L \;,\; id$ {addtype(stack[top],stack[top-3])}
$L \rightarrow id$ {addtype(stack[top],stack[top-1])}

T T.type=int

int int , id

, id

id

parsing stack:

(state)	(symbol)	(attribute)
$S_?$	T	T.type=int
$S_?$	$	-

D → T L
T → int {stack[top]=integer}
T → real {stack[top]=real}
L → L , id {addtype(stack[top],stack[top-3])}
L → id {addtype(stack[top],stack[top-1])}

T T.type=int
int int , id

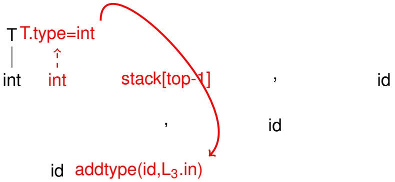
, id

id addtype(id,L_3.in)

parsing stack:

$S_?$	id	id.type=stack[top-1]
$S_?$	T	T.type=int
$S_?$	\$	-
(state)	(symbol)	(attribute)

D $\rightarrow$ T L
T $\rightarrow$ int {stack[top]=integer}
T $\rightarrow$ real {stack[top]=real}
L $\rightarrow$ L , id {addtype(stack[top],stack[top-3])}
L $\rightarrow$ id {addtype(stack[top],stack[top-1])}



parsing stack:

$S_?$	id	**id.type=stack[top-1]**
$S_?$	T	T.type=int
$S_?$	\$	-
(state)	(symbol)	(attribute)

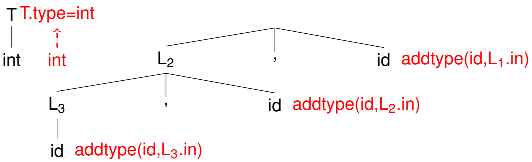
$D \rightarrow T \quad L$

$T \rightarrow$ int {stack[top]=integer}

$T \rightarrow$ real {stack[top]=real}

$L \rightarrow L \;,\; $ id {addtype(stack[top],stack[top-3])}

$L \rightarrow$ id {addtype(stack[top],stack[top-1])}



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)

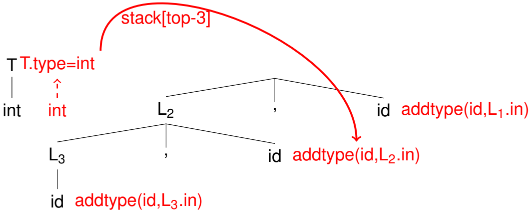
D $\rightarrow$ T L

T $\rightarrow$ int {stack[top]=integer}

T $\rightarrow$ real {stack[top]=real}

L $\rightarrow$ L , id {addtype(stack[top],stack[top-3])}

L $\rightarrow$ id {addtype(stack[top],stack[top-1])}



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
$S_?$	$	-

Marker

- Given the following SDD, where $|\alpha|$!= $|\beta|$

  A$\rightarrow$ X α Y | X β Y

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

  $M_1$$\rightarrow$ ε {M_1.s = X.s}

  $M_2$$\rightarrow$ ε {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

- How to add the marker ?

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] }

How to Add the Marker?

1. Identify the stack location(s) to find the desired attribute
2. Is there a conflict of location?
   - Yes, add a marker;
   - No, no need to add.
3. Add the marker in the place to remove location inconsistency

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

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

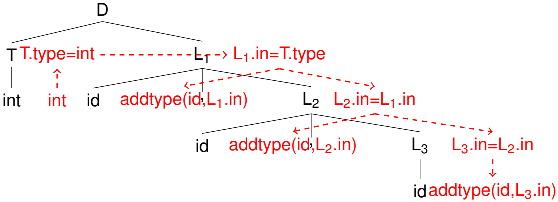
Translation Scheme for Top-Down Parsing

- Predictive Recursive Descent Parsers: Straightforward
  - Synthesized Attribute: Return value of function call for non-terminal is synthesized attribute
    - All function calls for children nodes would have completed by the time this function call returns
    - All dependent values would have been computed
  - Inherited Attribute: Pass as argument to function call for non-terminal inheriting attribute
    - L-Attributed grammar guarantees that dependent attributes come from left sibling attributes or parent inherited attributes
    - Left sibling function calls would have completed and parent inherited attribute would have been passed in as argument
    - All dependent values would have been computed
- Now let's focus on table-driven LL Parsers

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

- it is natural to evaluate inherited attributes



parsing stack:

(symbol)	(attribute)

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

- it is natural to evaluate inherited attributes

D

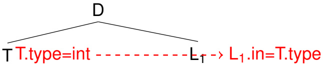
parsing stack:

(symbol)	(attribute)
D	

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

- it is natural to evaluate inherited attributes



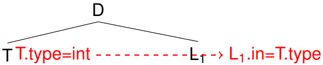
parsing stack:

T	T.type=int
L_1	L_1.in=()
(symbol)	(attribute)

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

- it is natural to evaluate inherited attributes



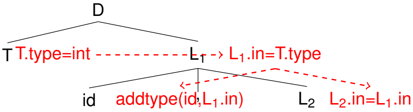
parsing stack:

(symbol)	(attribute)
L_1	L_1.in=int

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

- it is natural to evaluate inherited attributes



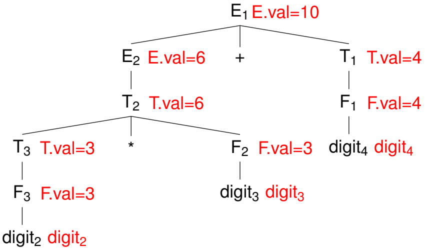
parsing stack:

(symbol)	(attribute)
id	id.type=L_1.in
,	
L_2	L_2.in=intL_1.in

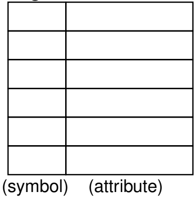
Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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



parsing stack:



Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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

E_1

parsing stack:

E_1	
(symbol)	(attribute)

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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

E_1 E.val=?

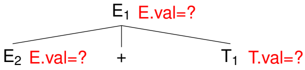
parsing stack:

E_1	E_1.val=?
(symbol)	(attribute)

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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



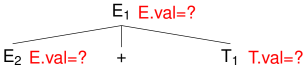
parsing stack:

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

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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



parsing stack:

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

- Solution
  - Always push a 'dummy' stack item below a non-terminal to hold intermediate values for attribute calculation
  - Update dummy item whenever a child node is popped with intermediate value
  - When all children nodes have been popped, compute synthesized attribute from stored values

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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

E_1

parsing stack:

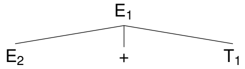
(symbol)	(attribute)
E_1	
E_1.val	???

- Solution
  - Always push a ’dummy’ stack item below a non-terminal to hold intermediate values for attribute calculation
  - Update dummy item whenever a child node is popped with intermediate value
  - When all children nodes have been popped, compute synthesized attribute from stored values

Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

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



parsing stack:

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

- Solution
  - Always push a 'dummy' stack item below a non-terminal to hold intermediate values for attribute calculation
  - Update dummy item whenever a child node is popped with intermediate value
  - When all children nodes have been popped, compute synthesized attribute from stored values