Semantic Analysis
Compiler Phases and Errors

- **Lexical analysis**
  - Converts source code into tokens
  - Detects inputs with illegal tokens
  - Does the input program use words defined in the "dictionary" of the language?

- **Syntax analysis**
  - Takes tokens as input
  - Converts tokens into an abstract syntax tree (AST)
  - Detects inputs with incorrect structure
  - Is the input program grammatically correct?

- **Semantic analysis**
  - Takes the AST as input
  - Modifies the AST and creates a symbol table
  - Detects semantic errors (errors in meaning)
  - Does the input program "make sense"?
Compiler Phases and Errors

- Lexical analysis
  - source code $\rightarrow$ tokens
  - detects inputs with illegal tokens
  - does the input program use words defined in the "dictionary" of the language?

- Syntax analysis

- Semantic analysis
Compiler Phases and Errors

- **Lexical analysis**
  - source code → tokens
  - detects inputs with illegal tokens
  - does the input program use words defined in the "dictionary" of the language?

- **Syntax analysis**
  - tokens → abstract syntax tree (AST)
  - detects inputs with incorrect structure
  - is the input program grammatically correct?

- **Semantic analysis**
Compiler Phases and Errors

- **Lexical analysis**
  - source code $\rightarrow$ tokens
  - detects inputs with illegal tokens
  - does the input program use words defined in the "dictionary" of the language?

- **Syntax analysis**
  - tokens $\rightarrow$ abstract syntax tree (AST)
  - detects inputs with incorrect structure
  - is the input program grammatically correct?

- **Semantic analysis**
  - AST $\rightarrow$ (Modified) AST + Symbol Table
  - detects semantic errors (errors in meaning)
  - does the input program "make sense"?
Why is Semantic Analysis Required?

- Because programs use symbols (a.k.a. identifiers)
  - Identifiers require context to figure out the meaning

- Consider the English sentence: "He ate it."
  - This sentence is syntactically correct
  - But it makes sense only in the context of a previous sentence: "Sam bought a fish."

- Semantic analysis
  - Associates identifiers with objects they refer to
    - "He" → "Sam"
    - "it" → "fish"
  - Checks whether identifiers are used correctly
    - "He" and "it" refer to some object: def-use check
    - "it" is a type of object that can be eaten: type check
Why is Semantic Analysis Required?

- Context cannot be analyzed using a CFG parser
- Associating IDs to objects require expressing the pattern:

\[ \{ wcw \mid w \in (a|b)^* \} \]

The 1st \( w \) represents the definition of the ID,
The \( c \) represents arbitrary intervening code,
The 2nd \( w \) represents the use of that ID.
This cannot be expressed using a context free grammar.

- Requires semantic analysis in a separate pass to analyze:
  - Def-use relationships
  - Type consistency when using IDs
  - Inheritance relationships between types
  - ...

...
A Simple Semantic Check

“Matching identifier definitions with uses”
- Important analysis in most languages
- If there are multiple definitions, which one to match?
A Simple Semantic Check

“Matching identifier definitions with uses”

- Important analysis in most languages
- If there are multiple definitions, which one to match?

```c
void foo()
{
    char x;
    ...
    {
        int x;
        ...
    }
    x = x + 1;
}
```
A Simple Semantic Check

“Matching identifier **definitions** with **uses**”

- Important analysis in most languages
- If there are multiple definitions, which one to match?

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

A *binding* is the association of a use of an identifier to the definition of that identifier
  - Which variable (or function) an identifier is referring to

The *scope* of a definition is a portion of a program in which the binding is valid
  - Uses of identifier in the scope is bound to that definition

Some properties of scopes
  - The definition of an identifier is restricted by its scope
  - Scopes for the same identifier can never overlap - there is always exactly one binding per identifier use
  - If an identifier use is not within the scope of any definition, the identifier is said to be undefined

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 definition

```c
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 definition

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

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

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

Which `x`'s definition is the closest?

- case (a): ...(1)...(2)...(5)
- case (b): ...(1)...(2)...(3)...(4)...(5)
Dynamic Scope

- Dynamic scoping depends on bindings formed during the execution of the program
  - Perl, Bash, LISP, Scheme

- Rule: Refer to the closest binding in the current execution

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

- Which x’s definition is the closest?
  - case (a): ...(1)...(2)...(5)
  - case (b): ...(1)...(2)...(3)...(4)...(5)
Most languages that started with dynamic scoping (LISP, Scheme, Perl) added static scoping afterwards.

Why?

- 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

- Static scoping is easier for human beings to understand
  - Bindings readily apparent from lexical structure of code

- Static scoping 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
Symbol Table
A compiler data structure that tracks information about all identifiers (symbols) in a program

- Each entry represents a definition of that identifier
- Maps use of IDs to definitions at any given point in program
- Contents updated whenever scopes are entered or exited
- Built by either...
  - Traversing the parse tree from top to bottom
  - Using semantic actions as an integral part of parsing process

Usually discarded after generating binary code

- By then, all references to symbols have been mapped to memory locations already
- For debugging, symbols may be included in binary
  - To map memory locations back to symbol names when using debuggers
  - For GCC, using ‘gcc -g ...” includes debug symbol tables
Maintaining Symbol Table

- **Basic idea**
  ```
  int x; ... void foo() { int x; ... x=x+1; } ... x=x+1 ...
  ```
  - Before processing *foo*:
    Add definition of *x*, overriding old definition of *x* if any
  - After processing *foo*:
    Remove definition of *x*, restoring old definition of *x* if any

- **Operations**
  - `enter_scope()` start a new nested scope
  - `exit_scope()` exit current scope
  - `find_symbol(x)` find the information about *x*
  - `add_symbol(x)` add a symbol *x* to the symbol table
  - `check_symbol(x)` true if *x* is defined in current scope
Symbol Table Structure

What data structure to choose?
- List
- Binary tree
- Hash table

Tradeoffs: time vs space
- Let us first consider the organization w/o scope
List

- Linked list or array
  - id\_1 | info\_1 → id\_2 | info\_2
  - id\_1 | info\_1
  - id\_2 | info\_2
  - ...

- Array: no space wasted, insert/delete: \(O(n)\), search: \(O(n)\)
- Linked list: extra pointer space, insert/delete: \(O(1)\), search: \(O(n)\)
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
Discussion:

- Use more space than array/list
- But insert/delete/search is $O(\log n)$ on balanced tree
Discussion:
- Use more space than array/list
- But insert/delete/search is $O(\log n)$ on balanced tree
- In the worst case, tree may reduce to linked list
  - Then insert/delete/search becomes $O(n)$
Hash Table

- Hash function decides mapping from identifier to index
- Conflicts resolved by chaining multiple IDs to same index

Memory consumption from hash table \((N << M)\)
- \(M\): the size of hash table
- \(N\): the number of stored identifiers

But insert/delete/search in \(O(1)\) time
- Can become \(O(n)\) with frequent conflicts and long chains

\[
\text{hash}(\text{id}_\text{name}) \rightarrow \text{index}
\]
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

```c
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
Scopes are defined by nested lexical structures

- Push pointer to symbol table when entering scope
- Pop pointer to symbol table when exiting scope
- Search from top of the active symbol table stack

Above shows example with nested function definitions
Scopes are defined by nested lexical structures
- Push pointer to symbol table when entering scope
- Pop pointer to symbol table when exiting scope
- Search from top of the active symbol table stack

Above shows example with nested function definitions
Scopes are defined by nested lexical structures

- Push pointer to symbol table when entering scope
- Pop pointer to symbol table when exiting scope
- Search from top of the active symbol table stack

Above shows example with nested function definitions
Dealing with Scopes Using Chaining

- Disadvantages of stacking symbol tables
  - Inefficient searching with deeply nested scopes
  - Inefficient use of memory due to multiple hash tables
    (Must size hash tables for avg anticipated size of scope)

- Solution: Single symbol table for all scopes using chaining

- Does not maintain list of all scopes
  - Cannot refer back to old scopes later
  - But still useful for implementing block scopes
Dealing with Scopes Using Chaining

Operation:

- **Insert**: insert (ID, current nesting level) at front of chain
- **Search**: fetch ID at the front of chain
- **Delete**: when exiting level \( k \), remove all symbols with level \( k \) (IDs for each nesting level are maintained in a separate list, not shown in pic)

- Exiting scope discards it (suitable for block scopes)
- Exiting scopes becomes slightly more expensive
- Lookup requires only a single hash table access
- Savings in memory due to single large hash table
What Information is Stored in the Symbol Table

Entry in symbol Table:

<table>
<thead>
<tr>
<th>string</th>
<th>kind</th>
<th>attributes</th>
</tr>
</thead>
</table>

- String — the name of identifier
- Kind — function, variable, type name, parameter

Attributes vary with the kind of symbols

- variable → type, address of variable
- function → prototype, address of function body

Vary with the language

- Fortran’s array → type, dimensions, dimension sizes
  
  `real A(3,5) /* size required for static allocation */`

- C’s array → type, dimensions, optional dimension size
  
  `int A[][5] /* A can be dynamically allocated */`
Type information might be arbitrarily complicated

In C:
```c
struct {
    int a[10];
    char b;
    real c;
}
```

Store all relevant attributes in an attribute list

<table>
<thead>
<tr>
<th>id</th>
<th>array</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1st dimension:

<table>
<thead>
<tr>
<th>id</th>
<th>array</th>
<th>type</th>
<th>lower bound</th>
<th>upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2nd dimension:

<table>
<thead>
<tr>
<th>id</th>
<th>struct</th>
<th>total size</th>
<th>field1</th>
<th>type</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>struct</th>
<th>total size</th>
<th>field2</th>
<th>type</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now we have a Symbol Table with type info for each symbol.
How is this information useful for semantic analysis?
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

Some type checking examples:
- Given `char *str = "Hello";`, `str[2]` is legal, `str/2` is not
- Given `const int pi = 3.14;`, `pi/2` is legal, `pi=2` is not
Static Type Checking vs. Dynamic Type Checking

Static type checking: type checking at compile time

- Infers program is type consistent through code analysis
- E.g. int a, b, c; a = b + c; can be proven type consistent because the addition of two ints is an int

Dynamic type checking: type checking at execution time

- Type consistency by checking types of runtime values (Usually stored in the form of "type tags" alongside values)
- E.g. Java array bounds check: Is int A[10], i; ... A[i] = i; type consistent?
- E.g. C++/Java downcasting to a subclass: Is dynamic_cast<Child*>(parent); type consistent?

Static type checking is always more desirable. Why?

- Always desirable to catch errors before runtime
- Dynamic type checking carries runtime overhead
Static Type Checking vs. Dynamic Type Checking

- **Static type checking**: type checking at compile time
  - Inference that program is type consistent through code analysis
  - E.g. `int a, b, c; a = b + c;` can be proven type consistent because the addition of two ints is an int

- **Dynamic type checking**: type checking at execution time
  - Type consistency by checking types of runtime values
    (Usually stored in the form of "type tags" alongside values)
  - E.g. Java array bounds check:
    Is `int A[10], i; ... A[i] = i;` type consistent?
  - E.g. C++/Java downcasting to a subclass:
    Is `dynamic_cast<Child*>(parent);` type consistent?
Static Type Checking vs. Dynamic Type Checking

- **Static type checking**: type checking at compile time
  - Infers program is type consistent through code analysis
  - E.g. `int a, b, c; a = b + c;` can be proven type consistent because the addition of two ints is an int

- **Dynamic type checking**: type checking at execution time
  - Type consistency by checking types of runtime values (Usually stored in the form of "type tags" alongside values)
  - E.g. Java array bounds check:
    - Is `int A[10], i; ... A[i] = i;` type consistent?
  - E.g. C++/Java downcasting to a subclass:
    - Is `dynamic_cast<Child*>(parent);` type consistent?

- Static type checking is always more desirable. Why?
  - Always desirable to catch errors before runtime
  - Dynamic type checking carries runtime overhead
Static vs. Dynamic Typing

- Statically typed: C/C++, Java
  - E.g. int x; /* type of x is int */
  - Types are explicitly defined or can be inferred from code
    + Most type checking happens at compile time → fast code

- Dynamically typed: Python, JavaScript, PHP
  - E.g. var x; /* type of x unknown */
  - Type is a runtime property decided only during execution
    - Same variable may go through different types in its lifetime
    - Most type checking must happen at runtime → slow code
    - Flexibility results in shorter code / less programmer effort
      → Suitable for scripting or prototyping languages

In this lecture, we will focus on static type checking of statically typed languages, but same techniques can be applied to dynamic typing.
Static vs. Dynamic Typing

- **Statically typed: C/C++, Java**
  - E.g. `int x; /* type of x is int */`
  - Types are explicitly defined or can be inferred from code
    - Most type checking happens at compile time → fast code

- **Dynamically typed: Python, JavaScript, PHP**
  - E.g. `var x; /* type of x unknown */`
  - Type is a runtime property decided only during execution
    - Same variable may go through different types in its lifetime
      - Most type checking must happen at runtime → slow code
    - Flexibility results in shorter code / less programmer effort
      → Suitable for scripting or prototyping languages
Static vs. Dynamic Typing

- **Statically typed: C/C++, Java**
  - E.g. `int x; /* type of x is int */`
  - Types are explicitly defined or can be inferred from code
    - Most type checking happens at compile time → fast code

- **Dynamically typed: Python, JavaScript, PHP**
  - E.g. `var x; /* type of x unknown */`
  - Type is a runtime property decided only during execution
  - Same variable may go through different types in its lifetime
    - Most type checking must happen at runtime → slow code
  - Flexibility results in shorter code / less programmer effort
    → Suitable for scripting or prototyping languages

- In this lecture, we will focus on static type checking of statically typed languages, but same techniques can be applied to dynamic typing
Rules of Inference

What are *rules of inference*?
- Inference rules have the form
  
  \[ \text{if Precondition is true, then Conclusion is true} \]

- Below concise notation used to express above statement

\[
\begin{array}{c}
\text{Precondition} \\
\text{Conclusion}
\end{array}
\]

- In the context of type checking:
  
  \[ \text{if expressions E1, E2 have certain types (Precondition), expression E3 is legal and has a certain type (Conclusion)} \]

Type checking via inference
- Start from variable types and constant types
- Repeatedly apply inference rules until entire program is inferred legal
By tradition inference rules are written as

$$\text{Precondition}_1, \ldots, \text{Precondition}_n$$

$\text{Conclusion}$

- The precondition/conclusion has the form “$e:T$”
- Meaning
  - If $\text{Precondition}_1$ and $\ldots$ and $\text{Precondition}_n$ are true, then $\text{Conclusion}$ is true.
  - “$e:T$” indicates “$e$ is of type $T$”
  - Example: rule-of-inference for add operation

$$\begin{align*}
  e_1 &: \text{int} \\
  e_2 &: \text{int} \\
  e_1 + e_2 &: \text{int}
\end{align*}$$

Rule: If $e_1$, $e_2$ are ints then $e_1 + e_2$ is legal and is an int
Two Simple Rules

[Constant]

\[ \text{i is an integer} \]

\[ i: \text{int} \]

[Add operation]

\[ e_1: \text{int} \]

\[ e_2: \text{int} \]

\[ e_1 + e_2: \text{int} \]

Example: given “i is an integer” and “j is an integer”, is the expression “i+j” legal? Then, what is the type?

\[ \text{i is an integer} \]

\[ i: \text{int} \]

\[ \text{j is an integer} \]

\[ j: \text{int} \]

\[ i+j: \text{int} \]

This type of reasoning can be applied to the entire program.
More Rules

[New]

\[
\text{new } T : T
\]

[Not]

\[
\text{e: } \text{Boolean} \\
\text{not e: } \text{Boolean}
\]

However,

[Var?]

\[
\text{x is an identifier} \\
x : ?
\]

- the expression itself insufficient to determine type
- **solution**: provide context for this expression
A type environment gives type info for free variables

- A variable is free if not defined inside the expression
- It is a function mapping Identifiers to Types
  - Set of definitions active at the current point in code
  - i.e. Set of symbols and their types in symbol table
Let $O$ be a function (mapping) from Identifiers to Types, the sentence $O e : T$

is read as “under the assumption of environment $O$, expression $e$ has type $T$”

- “if 5 is an integer constant, expression 5 is an integer in any environment”
- “if $e1$ and $e2$ are int in $O$, then $e1+e2$ is int in $O$”
- “if variable $x$ is mapped to int in $O$, expression $x$ is int in $O$”
Definition Rule

- Let’s say syntax of definitions is: “let x: T in e”  
  (Meaning: expression e when x is defined to be type T)

[Definition w/o initialization]

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

O[T_0/x] means, O is modified to return T_0 on argument x and behave 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, then expression “let x: T_0 in e_1” is type T_1 in the current environment.
Definition Rule with Initialization

[Definition with initialization (initial try)]

\[
\frac{O \ e_0 : T_0}{O[T_0/x] e_1 : T_1}
\]

\[
O \ let \ x : T_0 \leftarrow e_0 \ in \ e_1 : T_1
\]

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

Example

\[
\text{class C inherits P ...}
\]
\[
\text{let x:P \leftarrow new C in ...}
\]

♫ the above rule does not allow this code
Subtyping

Subtyping is a relation $\leq$ on types

- $X \leq X$
- 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

[Definition with initialization]

$$
\begin{align*}
O \ e_0 : T \\
T & \leq T_0 \\
O[T_0/x] \ e_1 : T_1 \\
O \ let \ x : T_0 \leftarrow e_0 \ in \ e_1 : T_1
\end{align*}
$$

- Both versions of definition rules are correct
- The improved version checks more programs
A correct but too strict rule

[Assignment]

\[ O(\text{id}) = T_0 \]
\[ O \ e_1 : T_1 \]
\[ T_1 \leq T_0 \]
\[ \frac{O \ \text{id} \leftarrow e_1 : T_0}{O \ \text{id} \leftarrow e_1 : T_0} \]

The rule does not allow the below code

```latex
\text{class C inherits P \{ ... \}}
\text{let x:C, y: P in}
\text{x \leftarrow y \leftarrow \text{new C}}
```
Assignment

- An improved rule

\[
\begin{align*}
\text{[Assignment]} \\
O(id) &= T_0 \\
O e_1: T_1 \\
T_1 &\leq T_0 \\
O id &\leftarrow e_1: T_1
\end{align*}
\]

- Now the rule allows the below code
  class C inherits P { ... }
  let x: C, y: P in
  x ← y ← new C
If-then-else

Consider

\[ \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \]

- 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)

\[ Z = \text{lub}(X,Y) \] — \( Z \) is defined as the least upper bound of \( X \) and \( Y \) *iff*

- \( X \leq Z \land Y \leq Z \) ; \( Z \) is an upper bound
- \( X \leq W \land Y \leq W \implies Z \leq W \) ; \( Z \) is least among all upper bounds
If-then-else, case

[If-then-else]

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

- The rule allows the below code

```plaintext
let x:float, y:int, z:float in
x ← if (...) then y else z
/* Assuming lub(int, float) = float */
```
Implementing Type Checking on AST
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”
      - ...

- Introducing **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 Definition Rule (case 1)

Consider a hypothetical let rule

\[ \text{Wrong Definition with initialization (case 1)} \]

\[
\begin{align*}
\text{O } e_0 : T \\
T &\leq T_0 \\
\text{O } e_1 : T_1 \\
\text{O let } x : T_0 &\leftarrow e_0 \text{ in } e_1 : T_1
\end{align*}
\]

- How is it different from the correct rule?
  \( \text{O } e_1 : T_1 \) instead of \( \text{O}[T_0/x] e_1 : T_1 \)

- The following good program does not pass check
  
  let x : int \leftarrow 0 \text{ in } x+1

- Reason: Precondition \( \text{O } e_1 : T_1 \), or \( \text{O } x+1 : \text{int} \), is not satisfied without environment \( \text{O}[T_0/x] \), or \( \text{O}[\text{int}/x] \)
Wrong Definition Rule (case 2)

Consider a hypothetical let rule

\[
\text{[Wrong Definition with initialization (case 2)]}
\]

\[
\begin{align*}
O \ e_0 : T \\
T_0 & \leq T \\
O[T_0/x] \ e_1 : T_1 \\
O \ \text{let} \ x : T_0 & \leftarrow e_0 \ \text{in} \ e_1 : T_1
\end{align*}
\]

- How is it different from the correct rule? $T_0 \leq T$ instead of $T \leq T_0$
- The following bad program passes the check

  class C inherits P { only_in_C() { ... } }
  let x: C ← new P in x.only_in_C()

- Reason: Objects of children classes can be assigned to parent class variables, but not vice versa
Consider a hypothetical let rule

\[
\begin{align*}
O e_0 &: T \\
T &\leq T_0 \\
O[T/x] e_1 &: T_1 \\
O &\text{let } x: T_0 \leftarrow e_0 \text{ in } e_1: T_1
\end{align*}
\]

- How is it different from the correct rule?  
  \[O[T/x] e_1 : T_1\] instead of \[O[T_0/x] e_1 : T_1\]

- The following good program does not pass the check
  class C inherits P { ... }  
  let x: P \leftarrow new C in x \leftarrow new P

- Reason: Precondition \[O[T/x] e_1 : T_1\], or \[O[C/x] x \leftarrow new P: T_1\], is not satisfied, so rule cannot be applied. With correct rule, precondition would be \[O[P/x] x \leftarrow new P: T_1\]

- Why is it different from the correct rule?  
  \[O[T/x] e_1 : T_1\] instead of \[O[T_0/x] e_1 : T_1\]

- The following good program does not pass the check
  class C inherits P { ... }  
  let x: P \leftarrow new C in x \leftarrow new P
Type rules must walk a tight rope between making the type system unsound (bad programs are accepted as well typed) or, making the type system less usable (good programs are rejected).

What do we mean by a “good” program?

- A program where all operations performed on all values are type consistent at runtime
- Runtime value types may differ from defined variables types
  - Good programs may get rejected by type system. E.g.: Parent type vars may hold to child type values
  - Bad programs may get accepted by type system. E.g.: Child type vars may hold to parent type values, with casting
Type rules must walk a tight rope between

- making the type system unsound
  (bad programs are accepted as well typed)
- or, making the type system less usable
  (good programs are rejected)

What do we mean by a “good” program?

- A program where all operations performed on all values are type consistent \textit{at runtime}
- Runtime value types may differ from defined variables types
  - Good programs may get rejected by type system. E.g.: Parent type vars may hold to child type values
  - Bad programs may get accepted by type system. E.g.: Child type vars may hold to parent type values, with casting

Q: With enough effort can type systems ever hit the sweet spot? Be perfectly sound and perfectly usable?
Designing a Good Type System

- Type system has two conflicting design goals
  - Soundness: prevent “bad” programs from passing
  - Usability: allow programmer to easily express a “good” program within the boundaries of the type system

- Goal: Maximum usability while not compromising soundness
  - An example where the two goals collide

```java
class Count {
    int i = 0;
    Count inc() { i=i+1; return this; }
}
class Stock inherits Count { ... }
class Main {
    Stock a ← (new Stock).inc(); // Type error!
}
```
What Went Wrong?

- What is (new Stock).inc()’s type?
  - Dynamic type — Stock
  - Static type — Count
  - The type checker “looses” the type information
  - This makes inheriting inc() useless
    - Redefining inc() for each subclass to return the correct type defeats the whole purpose

- SELF_TYPE can make type system more usable
SELF_TYPE: More Usability

- What is SELF_TYPE?
  - The type (or class) on which the method is called
  - If we define inc() as `SELF_TYPE inc() { i=i+1; return this; }`, Type of `(new Stock).inc()` would be Stock, not Count
  - SELF_TYPE is a static type (readily apparent from program text) → can be checked at compile time

- Usable: Allows more good programs to pass without burden to programmer

- Sound: Does not compromise type consistency

- In practice,
  - Designed into type systems of some languages (e.g. Scala)
  - Or emulated by others using templates / generics (e.g. C++, Java...)

Can Static Type Checking ever be Perfect?

Fundamentally 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
  
  \[
  x \leftarrow \text{if } (y > 0) \text{ then } 10 \text{ else } "Hello"
  \]
  
  \[
  z \leftarrow \text{if } (y > 0) \text{ then } x/2 \text{ else } x.append("World") \text{ // Type error?}
  \]

- Above is a good program but will not pass type check

Reason why statically typed languages sometimes resort to dynamic type checking

- Why C++ / Java programmers are sometimes forced to downcast with dynamic type checking
  
  Programmer knows object is of child type at runtime in that particular execution path, but type system doesn’t
Can Static Type Checking ever be Perfect?

- Type systems must choose either to be more usable or more sound

- Dynamically typed languages (Python, JavaScript, PHP ...)
  - Usable: No fiddling with type system to make programs run
  - Sound: No type checking done before running

- Statically typed languages (Java, C++, ...)
  - Sound: Can discover many type errors before running
  - Usable: Must sometimes devise elaborate type systems (using inheritance, generic programming, ...), to make programs run

- In between: gradual typing (Cython, TypeScript, ...)
  - Allows a choice between static and dynamic typing
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 (and rules to compute them) with each grammar symbol
Two Aspects to Syntax Directed Translation

- Specification: Syntax Directed Definitions (SDD)
  - Set of **semantic rules** attached to each production
  - Semantic rules compute attributes or have side-effects (e.g. updating the symbol table)
  - In essence, attribute grammar + side-effects

- Implementation
  - Using Parse Trees
    - Apply semantic rules after syntax analysis
    - Rules applied at each node during parse tree traversal
  - Using Syntax Directed Translation Scheme (SDTS)
    - Apply semantic rules in the course of syntax analysis
    - Augment grammar with semantic actions embedded within RHS of productions
    - Used to construct parse tree itself
Two types of attributes:

- **Synthesized attributes**: attributes are computed from attributes of children nodes
  
  \[ P.\text{synthesized\_attr} = f(C_1.\text{attr}, C_2.\text{attr}, C_3.\text{attr}) \]

- **Inherited attributes**: attributes are computed from attributes of sibling and parent nodes
  
  \[ C_3.\text{inherited\_attr} = f(P_1.\text{attr}, C_1.\text{attr}, C_3.\text{attr}) \]
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
Example

- Each non-terminal symbol is associated with \texttt{val} attribute
- Each grammar rule is associated with a \texttt{semantic action}

\[
\begin{align*}
L & \rightarrow E \quad \{ \text{print}(E.\text{val}) \} \\
E & \rightarrow E_1 + T \quad \{ E.\text{val} = E_1.\text{val} + T.\text{val} \} \\
E & \rightarrow T \quad \{ E.\text{val} = T.\text{val} \} \\
T & \rightarrow T_1 + F \quad \{ T.\text{val} = T_1.\text{val} + F.\text{val} \} \\
T & \rightarrow F \quad \{ T.\text{val} = F.\text{val} \} \\
F & \rightarrow ( E ) \quad \{ F.\text{val} = E.\text{val} \} \\
F & \rightarrow \text{digit} \quad \{ F.\text{val} = \text{digit.lexval} \}
\end{align*}
\]
Inherited Attribute Example

Example:
We can use *inherited attributes* to track *type* information

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

\[
\begin{align*}
D & \rightarrow T \ L & \{ \ L\.in = T\.type \} \\
T & \rightarrow \text{int} & \{ \ T\.type = \text{integer} \} \\
T & \rightarrow \text{real} & \{ \ T\.type = \text{real} \} \\
L & \rightarrow L_1 , \ id & \{ \ L_1\.in = L\.in, \ \text{addtype} (id\.entry, \ L\.in) \} \\
L & \rightarrow id & \{ \ \text{addtype}(id\.entry, \ L\.in) \}
\end{align*}
\]
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
Implementation 1: Using Parse Trees

- Creates an **attribute parse tree** from the given parse tree
  - Traverse tree in a certain order and evaluate semantic rules at each node
  - Traversal order can be arbitrary as long as it adheres 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

```
D
  ________________
  |               |
  |               |
  |               |
T--------L1-----
 |      |      |
 int    L2     id
 |      |      |
 L3     id     
 |      |      |
 id     id     
```
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
Implementation 2: Using SDTS

- Using parse trees
  - Flexible: works for all SDDs except for dependency cycles
  - Inefficient: needs (one or more) traversals of parse tree
  - Question 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, a 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 symbols to its left i.e. $X_1 ... X_{j-1}$ and/or depends on inherited attributes of $A$ cannot depend on attributes of symbols to its right i.e. $X_{j+1} ... X_n$. 
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
    • Evaluate inherited attributes while going down the tree
    • Evaluate synthesized attributes while going up the tree
  ➢ Order for both top-down and bottom-up parsers

☐ Can be evaluated using SDTS w/o parse tree
Syntax Directed Translation Scheme (SDTS)

Syntax: \( A \rightarrow \alpha \{ \text{action}_1 \} \beta \{ \text{action}_2 \} \ldots \)

Meaning: Actions are executed “at that point” in the RHS
- Top-down: Right after previous symbol consumed
- Bottom-up: Right after previous symbol pushed to stack
  (when the ’dot’ reaches the action)

Actions for synthesized attributes occur at end of RHS
- Children attributes for \( A \) are only available at the end
- What we did when building parse tree for project 2, since all parse tree attributes are synthesized

Actions for inherited attributes occur at middle of RHS
- Inherited attributes for \( \beta \) can be computed at \( \text{action}_1 \)
  - Inherited attribute of \( A \) available from beginning
  - Left sibling attributes for \( \alpha \) available by then
Example: arithmetic expression after left-recursion removal

\[
\begin{align*}
E & \rightarrow T \ R \\
R & \rightarrow + \ T \ R \\
R & \rightarrow - \ T \ R \\
R & \rightarrow \epsilon \\
T & \rightarrow ( \ E ) \\
T & \rightarrow \text{num}
\end{align*}
\]

Both inherited and synthesized attributes are used

- $T$ — synthesized attribute $T.val$
- $R$ — inherited attribute $R.i$
  - synthesized attribute $R.s$
- $E$ — synthesized attribute $E.val$
### Evaluating attributes using SDTS

\[
E \rightarrow T \{R.i=T.val\} \quad R \{E.val=R.s\}
\]

\[
R \rightarrow + \quad T \{R_1.i=R.i+T.val\} \quad R_1 \{R.s=R_1.s\}
\]

\[
R \rightarrow - \quad T \{R_1.i=R.i-T.val\} \quad R_1 \{R.s=R_1.s\}
\]

\[
R \rightarrow \epsilon \quad \{R.s=R.i\}
\]

\[
T \rightarrow (E) \{T.val=E.val\}
\]

\[
T \rightarrow \text{num} \quad \{T.val=\text{num}.val\}
\]
E → T \{R.i=T.val\} R \{E.val=R.s\}
R → + T \{R_1.i=R.i+T.val\} R_1 \{R.s=R_1.s\}
R → - T \{R_1.i=R.i-T.val\} R_1 \{R.s=R_1.s\}
R → \epsilon \{R.s=R.i\}
T → ( E ) \{T.val=E.val\}
T → num \{T.val=num.val\}

Syntax Directed Translation Scheme (SDTS)
E → T {R.i=T.val} R {E.val=R.s}
R → + T {R₁.i=R.i+T.val} R₁ {R.s=R₁.s}
R → - T {R₁.i=R.i-T.val} R₁ {R.s=R₁.s}
R → ε {R.s=R.i}
T → ( E ) {T.val=E.val}
T → num {T.val=num.val}
Syntax Directed Translation Scheme Implementation
When using LR parsing (bottom-up parsing), it is natural and easy to evaluate synthesized attributes.
Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

it is natural and easy to evaluate synthesized attributes

parsing stack:

\[
\begin{array}{ccc}
\text{(state)} & \text{(symbol)} & \text{(attribute)} \\
S? & E & \text{E.val}=6 \\
S? & $ & - \\
\end{array}
\]
When using LR parsing (bottom-up parsing),
\[ \text{it is natural and easy to evaluate synthesized attributes} \]
Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

it is natural and easy to evaluate synthesized attributes

```
parsing stack:

<table>
<thead>
<tr>
<th>(state)</th>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>T</td>
<td>T.val=4</td>
</tr>
<tr>
<td>S?</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>S?</td>
<td>E</td>
<td>E.val=6</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>
```

```
Tree:

E → E.val=6 + T
  |     |
  T → T.val=6 *
      |     |
      T → T.val=3 *
          |     |
          F → F.val=3 digit_3 digit_3
```

```
T → T.val=6 *
  |     |
  F → F.val=4 digit_4 digit_4
```

```
F → F.val=3 digit_2 digit_2
```

```
digit_2 digit_2
```

```
4
3
2
```
Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

\[ \text{it is natural and easy to evaluate synthesized attributes} \]
When using LR parsing (bottom-up parsing),

it is **not natural** to evaluate inherited attributes
When using LR parsing (bottom-up parsing), it is **not natural** to evaluate inherited attributes.

**parsing stack:**

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>
Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

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

```
parsing stack:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>
```

```
T T.type=int
```

```
int int , id
```

```
, id
```

```
id
```
When using LR parsing (bottom-up parsing), it is **not natural** to evaluate inherited attributes.

```
    S?  id  id.type=L3.in
    S?  T   T.type=int
    S?  $   -

    parsing stack:
```

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>id</td>
<td>id.type=L3.in</td>
</tr>
<tr>
<td>S?</td>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>

```
    T  T.type=int
    |  |
    int  int  ,  id
       ,  id
 id  addtype(id,L3.in)
```
When using LR parsing (bottom-up parsing), it is **not natural** to evaluate inherited attributes.

Translation stack:

- **T**: `T.type=int`
- **int**: `int`
- **id**: `id.type=L3.in`
- **id**: `id`
- **id**: `id`
- **addtype(id,L3.in)**

Parsing stack:

<table>
<thead>
<tr>
<th>(state)</th>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>id</td>
<td>id.type=L3.in</td>
</tr>
<tr>
<td>S?</td>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>

(State) (Symbol) (Attribute)
When using LR parsing (bottom-up parsing), it is **not natural** to evaluate inherited attributes

parsing stack:

<table>
<thead>
<tr>
<th>(state)</th>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>id</td>
<td>id.type=L3.in</td>
</tr>
<tr>
<td>S?</td>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>

```
T T.type=int
  /
int int
  |    |
L2 , id addtype(id,L1.in)
  |    |
L3 , id addtype(id,L2.in)
  |    |
  id addtype(id,L3.in)
```
When using LR parsing (bottom-up parsing), it is **not natural** to evaluate inherited attributes.
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 → T  L
T → int  \{\text{stack[top]=integer}\}
T → real  \{\text{stack[top]=real}\}
L → L , id  \{\text{addtype(stack[top],stack[top-3])}\}
L → id  \{\text{addtype(stack[top],stack[top-1])}\}
\[ D \rightarrow T \quad L \]
\[ T \rightarrow \text{int} \quad \{\text{stack[top]=integer}\} \]
\[ T \rightarrow \text{real} \quad \{\text{stack[top]=real}\} \]
\[ L \rightarrow L \quad \text{id} \quad \{\text{addtype(stack[top],stack[top-3])}\} \]
\[ L \rightarrow \text{id} \quad \{\text{addtype(stack[top],stack[top-1])}\} \]

**parsing stack:**

\[
\begin{array}{ccc}
\text{int} & , & \text{id} \\
, & , & \text{id} \\
\text{id} & & \\
\hline
\text{S?} & \$ & - \\
\end{array}
\]

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

parsing stack:
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])\}

parsing stack:

\( T \) T.type=int
\| \rightarrow^{\uparrow}
\text{int} \quad \text{int} \quad , \quad \text{id} \\
\text{id} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{id} \\
\text{id} \quad \text{addtype(id,L_3.in)}
D \rightarrow T \ L
T \rightarrow \text{int} \ \{\text{stack}[\text{top}]=\text{integer}\}
T \rightarrow \text{real} \ \{\text{stack}[\text{top}]=\text{real}\}
L \rightarrow L , \ id \ \{\text{addtype}(\text{stack}[\text{top}], \text{stack}[\text{top}-3])\}
L \rightarrow id \ \{\text{addtype}(\text{stack}[\text{top}], \text{stack}[\text{top}-1])\}

\begin{array}{|c|c|c|}
\hline
\text{(state)} & \text{(symbol)} & \text{(attribute)} \\
\hline
S? & id & \text{id.type=stack[top-1]} \\
S? & T & \text{T.type=int} \\
S? & $ & - \\
\hline
\end{array}
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])}

parsing stack:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>id</td>
<td>id.type=stack[top-3]</td>
</tr>
<tr>
<td>S?</td>
<td>,</td>
<td></td>
</tr>
<tr>
<td>S?</td>
<td>L3</td>
<td>L3.in=int</td>
</tr>
<tr>
<td>S?</td>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>

(state) (symbol) (attribute)
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])\}

Parsing stack:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>S?</td>
<td>id</td>
<td>id.type=stack[top-3]</td>
</tr>
<tr>
<td>S?</td>
<td>,</td>
<td></td>
</tr>
<tr>
<td>S?</td>
<td>L3</td>
<td>L3.in=int</td>
</tr>
<tr>
<td>S?</td>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>S?</td>
<td>$</td>
<td>-</td>
</tr>
</tbody>
</table>

(state) (symbol) (attribute)
Given the following SDD, where $|\alpha| \neq |\beta|$

A $\rightarrow$ X $\alpha$ Y $|$ X $\beta$ 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 $\alpha$ $M_1$ Y $|$ X $\beta$ $M_2$ 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

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:

\[
\begin{align*}
S & \rightarrow a \ A \ B \ C \ E \ D \\
S & \rightarrow b \ A \ F \ B \ C \ F \ D \\
C & \rightarrow c \ 
\{ */ \ C.s = f(A.s) */ \} \\
D & \rightarrow d \ 
\{ */ \ D.s = f(B.s) */ \}
\end{align*}
\]
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-3]
  ... 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
When using LL parsing (top-down parsing), it is natural to evaluate inherited attributes.

```
parsing stack:
```

```
D

T.type=int        L1 -> L1.in=T.type
    ↑                  ↓
  int  int  id        L2        L2.in=L1.in
        addtype(id,L1.in)
    ↑                  ↓
  id        id        L3        L3.in=L2.in
        addtype(id,L2.in)
                        ↓
id        addtype(id,L3.in)
```

(parsing stack):

<table>
<thead>
<tr>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Translation Scheme for LL Parsing

When using LL parsing (top-down parsing),

it is natural to evaluate inherited attributes

D

parsing stack:

<table>
<thead>
<tr>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
Translation Scheme for LL Parsing

When using LL parsing (top-down parsing), it is natural to evaluate inherited attributes.

D

T T.type=int ----------- L1 → L1.in=T.type

parsing stack:

<table>
<thead>
<tr>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T.type=int</td>
</tr>
<tr>
<td>L1</td>
<td>L1.in=()</td>
</tr>
</tbody>
</table>
Translation Scheme for LL Parsing

When using LL parsing (top-down parsing), it is natural to evaluate inherited attributes.
When using LL parsing (top-down parsing), it is natural to evaluate inherited attributes.

\[
\begin{array}{c}
D \\
T: \text{type} = \text{int} \\
L_1 \rightarrow L_1.\text{in} = T.\text{type} \\
\text{id} \quad \text{addtype}(\text{id}, L_1.\text{in}) \\
L_2 \quad L_2.\text{in} = L_1.\text{in}
\end{array}
\]

Parsing stack:

<table>
<thead>
<tr>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id.\text{type} = L_1.\text{in}</td>
</tr>
<tr>
<td>,</td>
<td></td>
</tr>
<tr>
<td>L_2</td>
<td>L_2.\text{in} = L_1.\text{in}</td>
</tr>
</tbody>
</table>
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

```
E_1 E.val=10
  E_2 E.val=6 + T_1 T.val=4
    T_2 T.val=6
      T_3 T.val=3 *
        F_2 F.val=3 digit_4 digit_4
          digit_3 digit_3
    F_1 F.val=3
      digit_2 digit_2
```
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

1. Always push a 'dummy' stack item below a non-terminal to hold RHS with placeholders for children attributes.
2. Update dummy item whenever a child attribute is computed and popped from stack.
3. When all children attributes have been computed, compute synthesized attribute and pop from stack.

**parsing stack:**

<table>
<thead>
<tr>
<th>(symbol)</th>
<th>(attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁</td>
<td></td>
</tr>
</tbody>
</table>
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

**parsing stack:**

```
E₁  E.val=?
```

1. Always push a 'dummy' stack item below a non-terminal to hold RHS with placeholders for children attributes.
2. Update dummy item whenever a child attribute is computed and popped from stack.
3. When all children attributes have been computed, compute synthesized attribute and pop from stack.
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

\[ E_1 \ E.val=? \]
\[ E_2 \ E.val=? + T_1 \ T.val=? \]

**parsing stack:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_1.val=?</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>E_2</td>
<td>E_2.val=?</td>
</tr>
</tbody>
</table>

(symbol) (attribute)
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

**Solution**

1. Always push a ’dummy’ stack item below a non-terminal to hold RHS with place holders for children attributes.
2. Update dummy item whenever a child attribute is computed and popped from stack.
3. When all children attributes have been computed, compute synthesized attribute and pop from stack.
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

**Solution**

1. Always push a 'dummy' stack item below a non-terminal to hold RHS with place holders for children attributes.
2. Update dummy item whenever a child attribute is computed and popped from stack.
3. When all children attributes have been computed, compute synthesized attribute and pop from stack.
When using LL parsing (top-down parsing), it is **not natural** to evaluate synthesized attributes.

- Solution
  1. Always push a 'dummy' stack item below a non-terminal to hold RHS with placeholders for children attributes.
  2. Update dummy item whenever a child attribute is computed and popped from stack.
  3. When all children attributes have been computed, compute synthesized attribute and pop from stack.