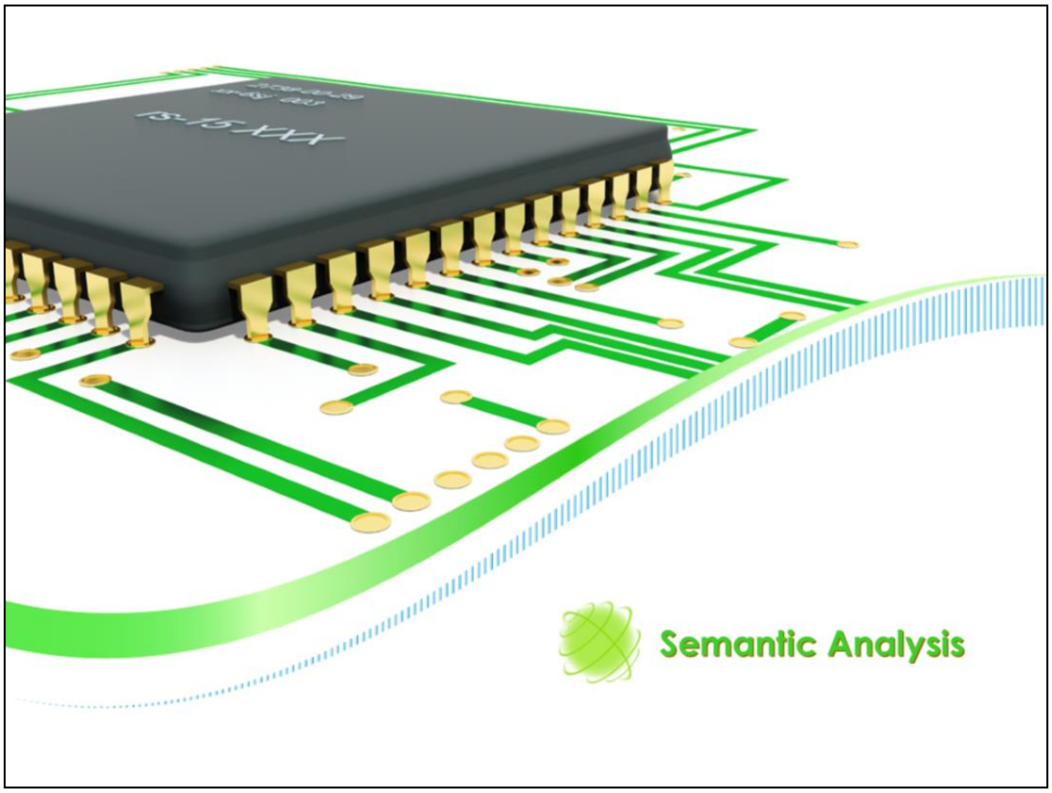
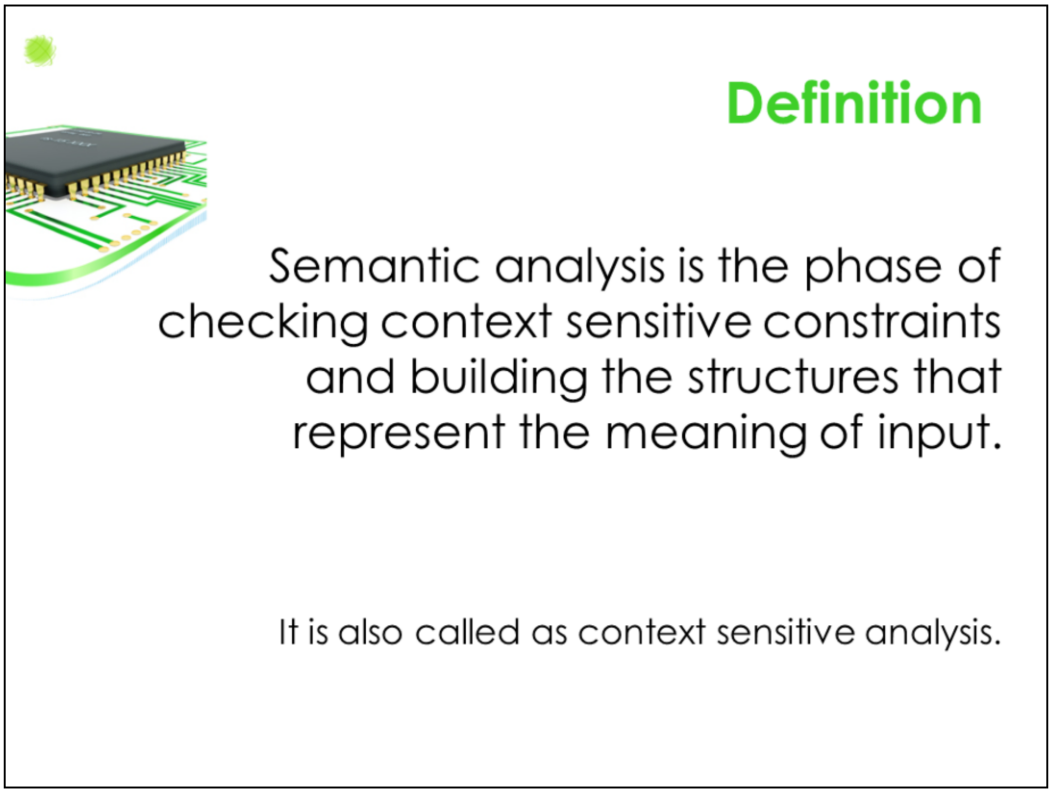


Semantic Analysis

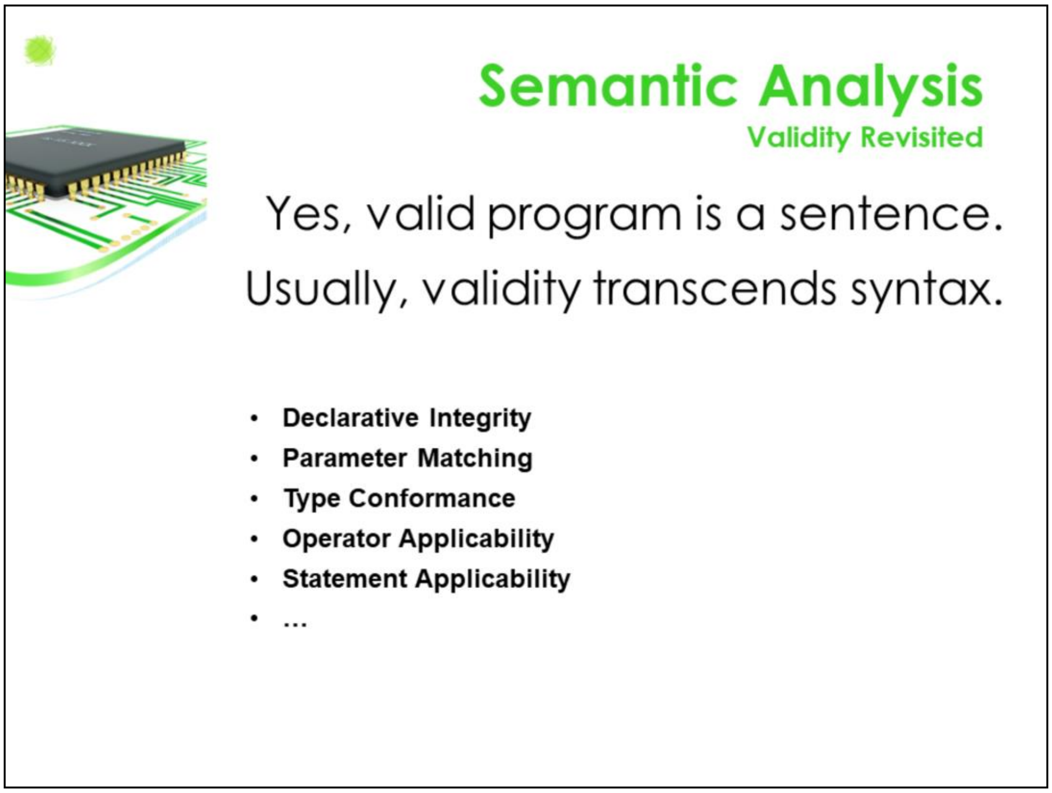


Definition

Semantic analysis is the phase of checking context sensitive constraints and building the structures that represent the meaning of input.

It is also called as context sensitive analysis.

Semantic analysis is sensitive to the rules of the language that transcends the syntactical constraints. Typically, semantic analysis is performed over syntactically validated input. The formal elements found in the syntactical representation are the findings that will be used for semantic checks. In some contexts, the results of the semantic analysis can be the final step for some language processing problems. For most of the applications, it is the “meaning represented” for further processing such as optimization, interpretations, code generation, and similar.



Semantic Analysis

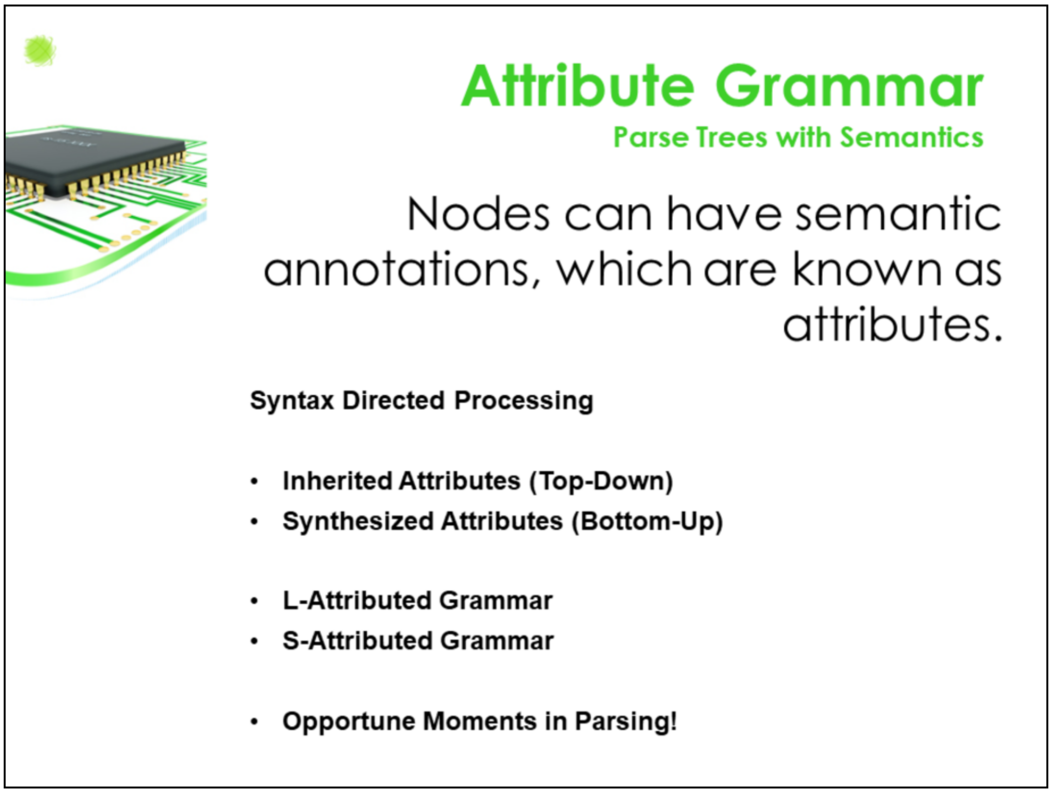
Validity Revisited

Yes, valid program is a sentence.

Usually, validity transcends syntax.

- Declarative Integrity
- Parameter Matching
- Type Conformance
- Operator Applicability
- Statement Applicability
- …

Thinking of our experiences with the programming languages which we are accustomed to, event without consulting to the programming language definitions we can safely state that syntactic compliance is necessary to validate a program but not enough. Validity of a given program is governed by the rules that transcends the grammatical constraints. Can you give examples of context dependent properties of a programming language that you know?



Attribute Grammar

Parse Trees with Semantics

Nodes can have semantic annotations, which are known as attributes.

Syntax Directed Processing

- **Inherited Attributes (Top-Down)**
- **Synthesized Attributes (Bottom-Up)**

- **L-Attributed Grammar**
- **S-Attributed Grammar**

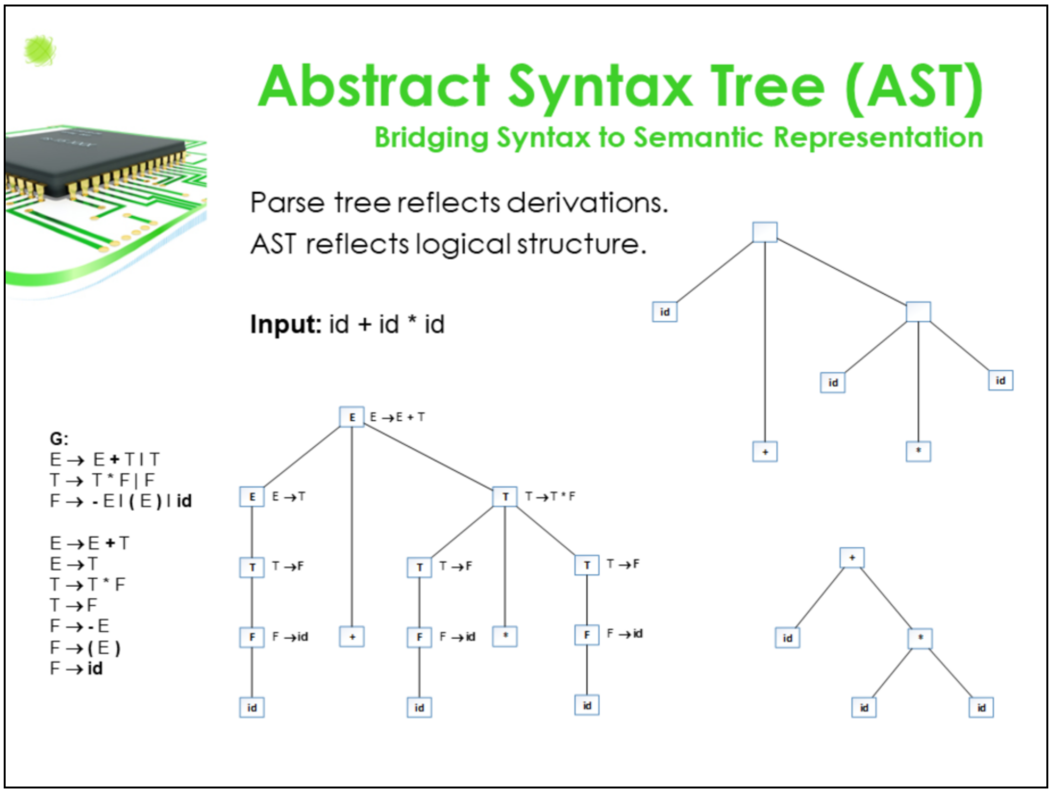
- **Opportune Moments in Parsing!**

Attribute grammars provides with initial collection of the semantics as a function of parsing. An attribute is a piece information associated with a parse tree node. The parsing strategy defines the way of collecting semantics. In a parsing strategy that builds the tree in bottom-up fashion, attributes are “synthesized”, collected from the nodes beneath. In a parsing method generating a parse tree from top to bottom, attributes can be collected from the upper node and the sibling nodes generated before. For this reason, top-down parsers are said to use inherited attributes. Depending on the implementation of a top-down parser, there are critical moments where attribute evaluations are possible. Thinking about the parse stack, the pushing and popping moments of symbols are opportune moments for semantic data collection. So, it is possible to implement top-down parsing strategy that can use both inherited and synthesized attributes. However, the top-down parsing is prone to back-tracking, meaning that tear-down of attributes is also a capability to implement. Shift-reduce strategy has opportune moment at reduction time, which makes synthesized attributes possible only.

In an L attributed grammar may have attribute actions after each symbol on the RHS. There may be one action for each production in S-attributed grammar, which are mainly intended for bottom up parsers.

The project that you will implement on the applied track of the course will use

Bison, which processes S-attributed grammar.



Abstract Syntax Tree (AST)

Bridging Syntax to Semantic Representation

Parse tree reflects derivations.
AST reflects logical structure.

Input: id + id * id

G:
E→ E**+**T|T
T→ T*F|F
F→ -E|(E)|**id**

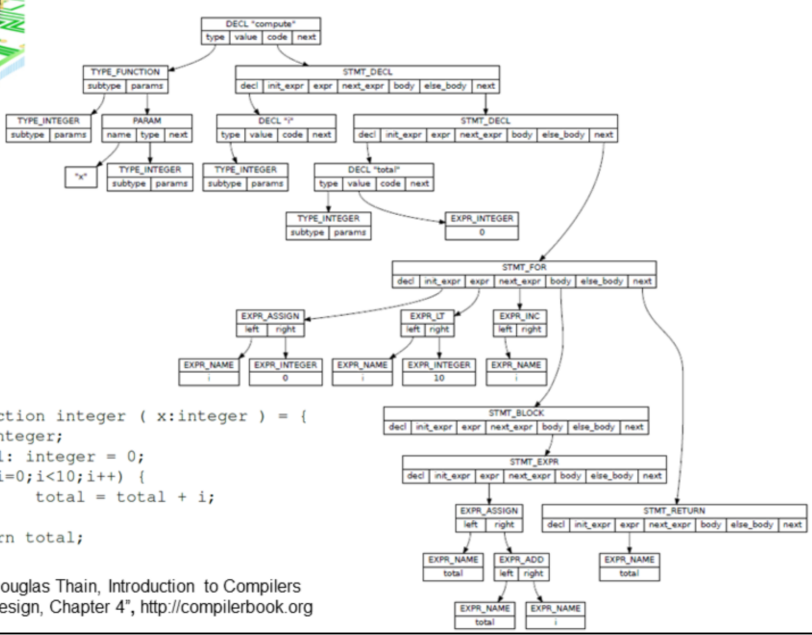
E→E**+**T
E→T
T→T*F
T→F
F→-E
F→(E)
F→**id**

E E→E+T
E E→T
T T→T*F
T T→F
F F→id
+
*
id

An abstract syntax tree a structure that is isolated from the details generated by the syntactic analysis. Remember that grammar of the intended language may be transformed for many reasons like elimination of ambiguity, elimination of left recursion, elimination of conflicts, component modularity, and similar. Even though the intended syntax remains the same, the derivations will generate interim symbols and hence, the parse tree will be populated for applicability reasons. The AST reflects the minimum necessary elements only. AST is less complex, more expressive, tool independent, and easy to process.

Abstract Syntax Tree (AST)

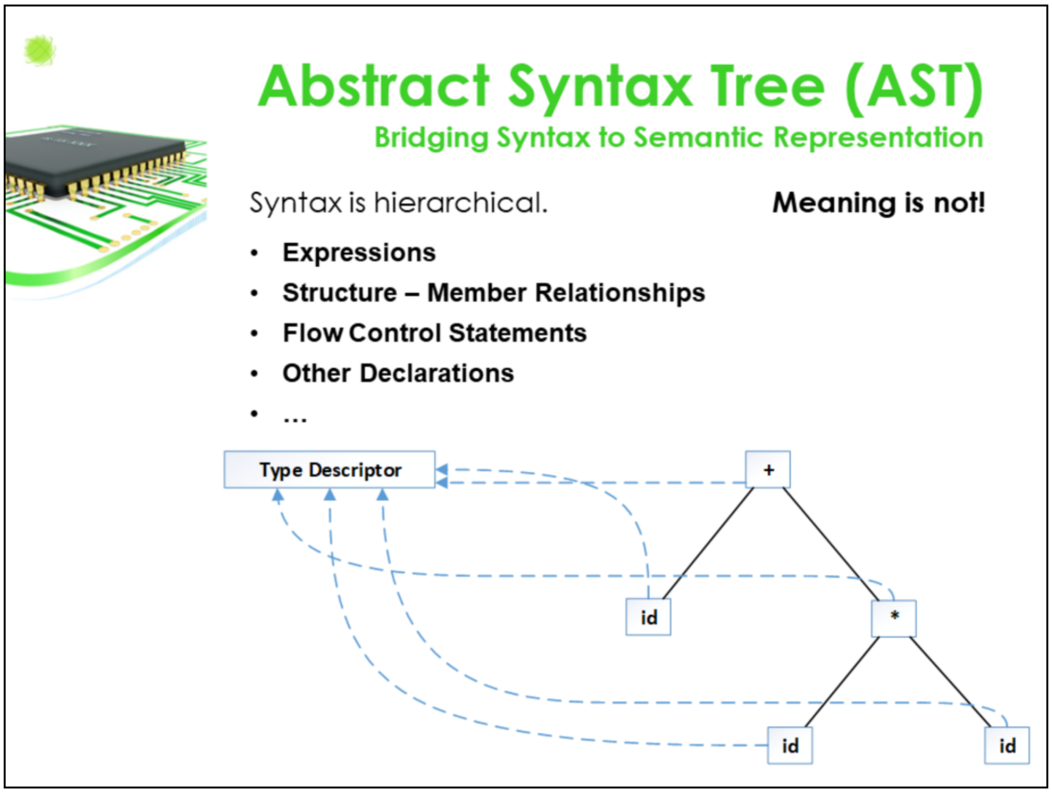
Bridging Syntax to Semantic Representation



```
compute: function integer ( x:integer ) = {
        i: integer;
        total: integer = 0;
        for(i=0;i<10;i++) {
                total = total + i;
        }
        return total;
}
```

Example from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

An abstract syntax tree represents the whole useful syntactic element in a hierarchy!

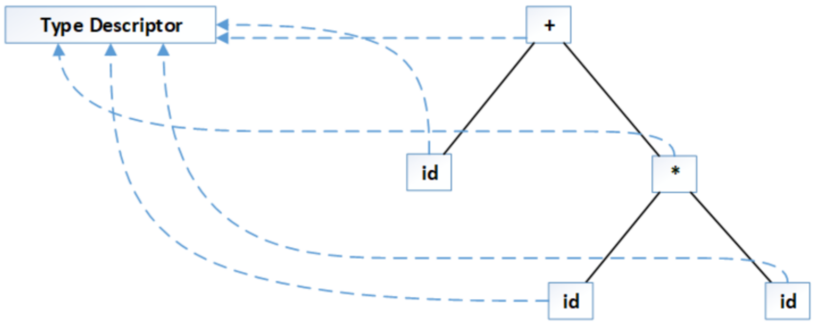


Abstract Syntax Tree (AST)

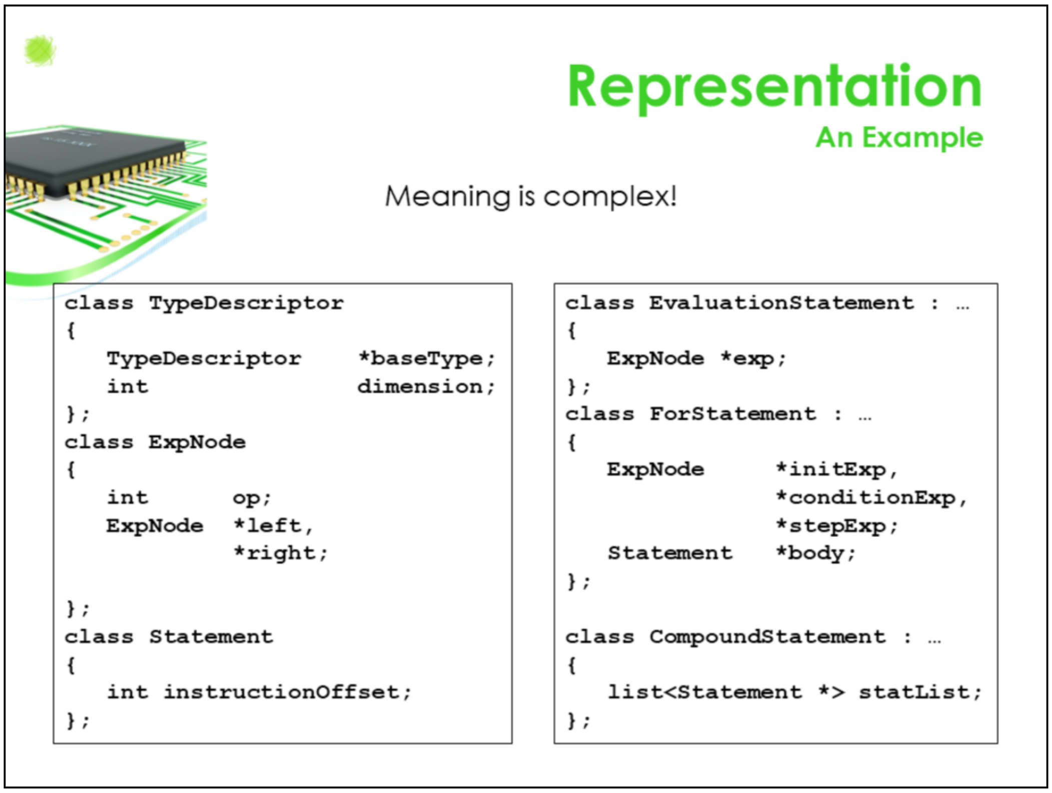
Bridging Syntax to Semantic Representation

Syntax is hierarchical. **Meaning is not!**

- **Expressions**
- **Structure – Member Relationships**
- **Flow Control Statements**
- **Other Declarations**
- **…**



Representation of the meaning transcends the limitations of tree structure. When the input is organized as sequence of symbols, which is a one dimensional composition, the limits of the hierarchical declarative organization cannot be overcome. But, the semantics require more complex representations. This problem is commonly solved by symbolic references.



Representation

An Example

Meaning is complex!

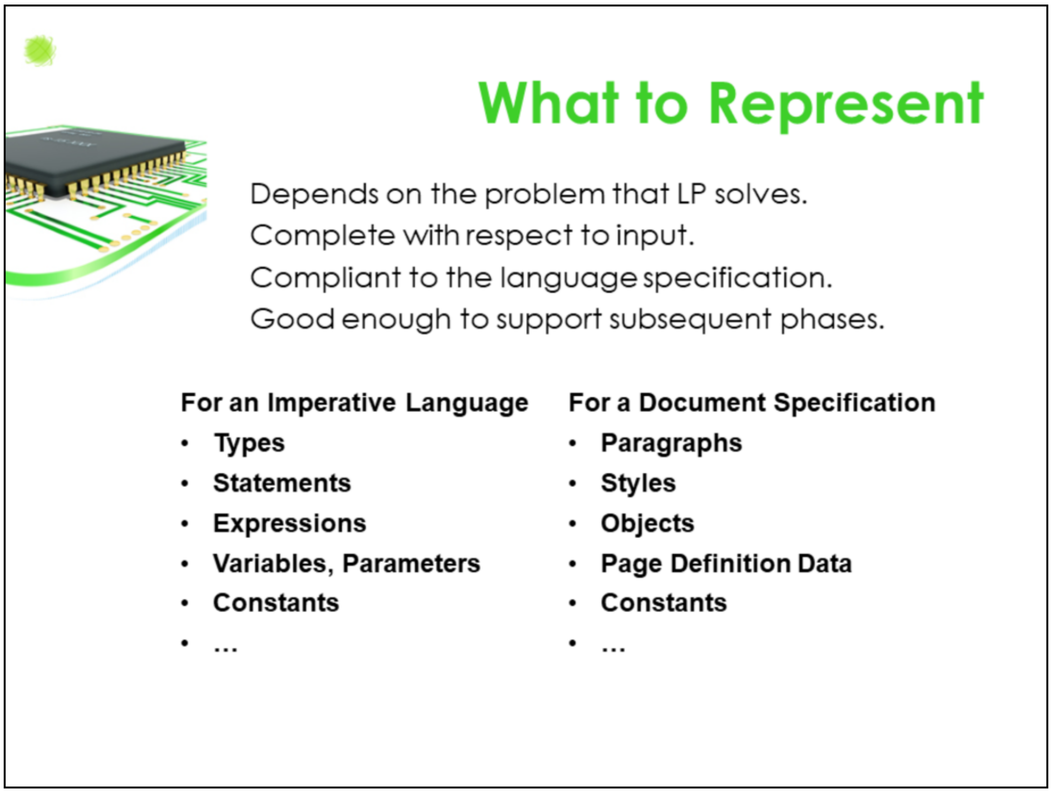
```
class TypeDescriptor
{
   TypeDescriptor    *baseType;
   int               dimension;
};
class ExpNode
{
   int      op;
   ExpNode  *left,
            *right;

};
class Statement
{
   int instructionOffset;
};
```

```
class EvaluationStatement : …
{
   ExpNode *exp;
};
class ForStatement : …
{
   ExpNode     *initExp,
               *conditionExp,
               *stepExp;
   Statement   *body;
};

class CompoundStatement : …
{
   list<Statement *> statList;
};
```

The classes EvaluationStatement, ForStatement, and CompoundStatement in the example on this slide are subclasses of the Statement class. The attributes collected evolves a complex structure that represents the whole in a more processable way.



What to Represent

Depends on the problem that LP solves.
Complete with respect to input.
Compliant to the language specification.
Good enough to support subsequent phases.

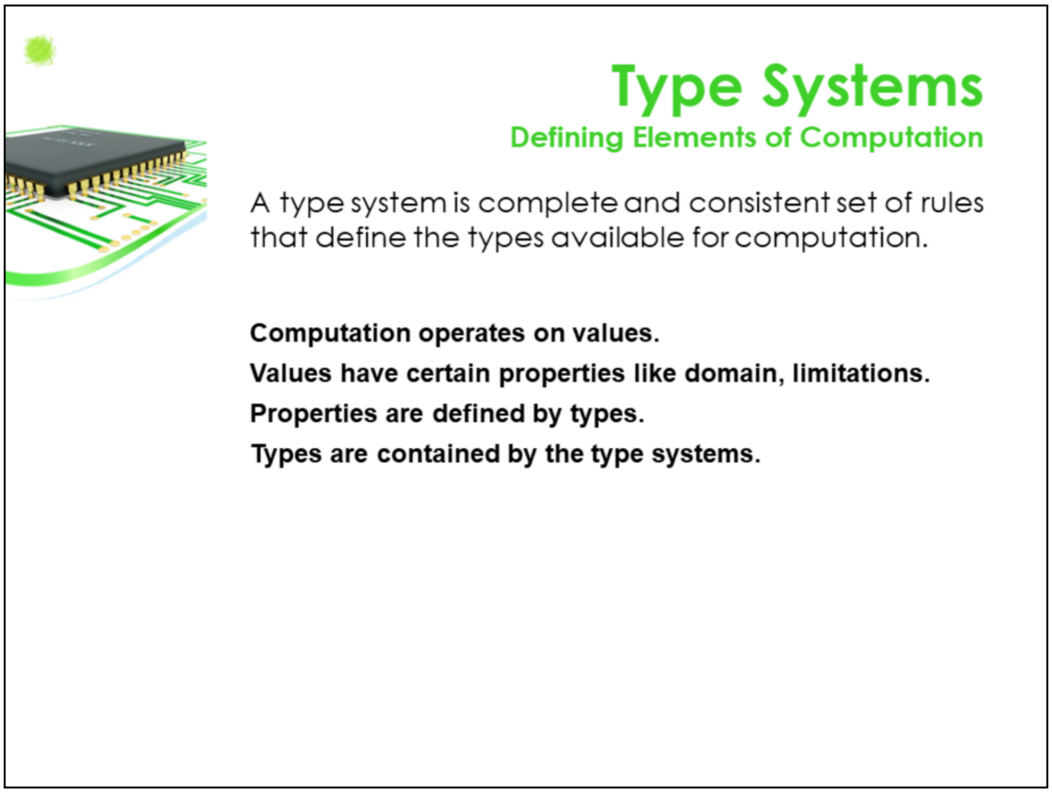
For an Imperative Language

- **Types**
- **Statements**
- **Expressions**
- **Variables, Parameters**
- **Constants**
- **…**

For a Document Specification

- **Paragraphs**
- **Styles**
- **Objects**
- **Page Definition Data**
- **Constants**
- **…**

The meaning to represent is dependent on the problem domain. This is valid for all cases independent of presence of language processor as a part of the solution. For the language processor, the representation must capture all required aspects of the input. These aspects are clearly defined by the language specification. The way to represent these aspects must support subsequent phases of processing sufficiently and efficiently.



Type Systems

Defining Elements of Computation

A type system is complete and consistent set of rules that define the types available for computation.

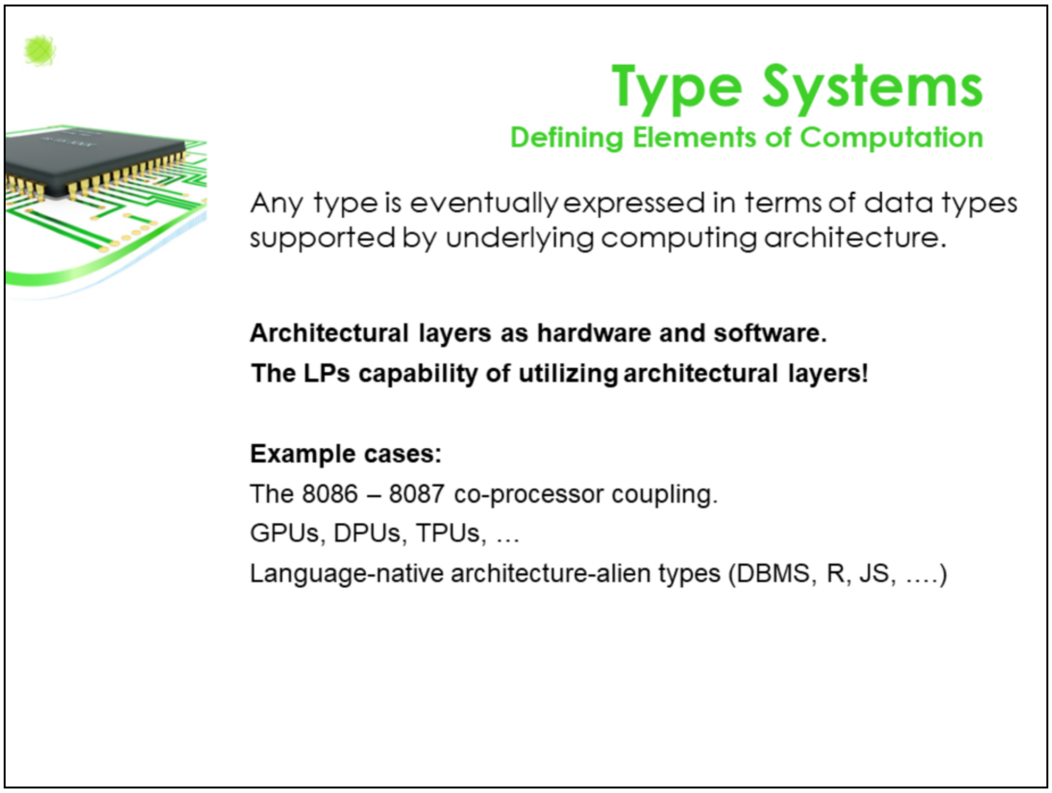
Computation operates on values.

Values have certain properties like domain, limitations.

Properties are defined by types.

Types are contained by the type systems.

A type system is complete and consistent set of rules that define the types available for computation. Typically, these rules address the issues related to type availability, type interactions, type specific limitations, type equivalence, and similar. A type is distinct set of properties that describe the domains of values. Computation requires values having certain properties to operate on.



Type Systems

Defining Elements of Computation

Any type is eventually expressed in terms of data types supported by underlying computing architecture.

Architectural layers as hardware and software.

The LPs capability of utilizing architectural layers!

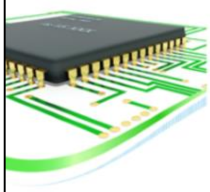
Example cases:

The 8086 – 8087 co-processor coupling.

GPUs, DPUs, TPUs, …

Language-native architecture-alien types (DBMS, R, JS, ….)

Any type is eventually expressed in terms of data types supported by underlying computing architecture. This requires robust transformation of values that belong to the language defined domain into some composition of underlying type which the architecture operate on. Sometimes, this transformation can be as simple as mapping the language type to the architecture defined type. Sometimes, it can be very complex, requiring critical software components at primitive levels. The transformations performed by the language processor must ensure correct computation as defined by the language specification.



Type Systems

Defining Elements of Computation

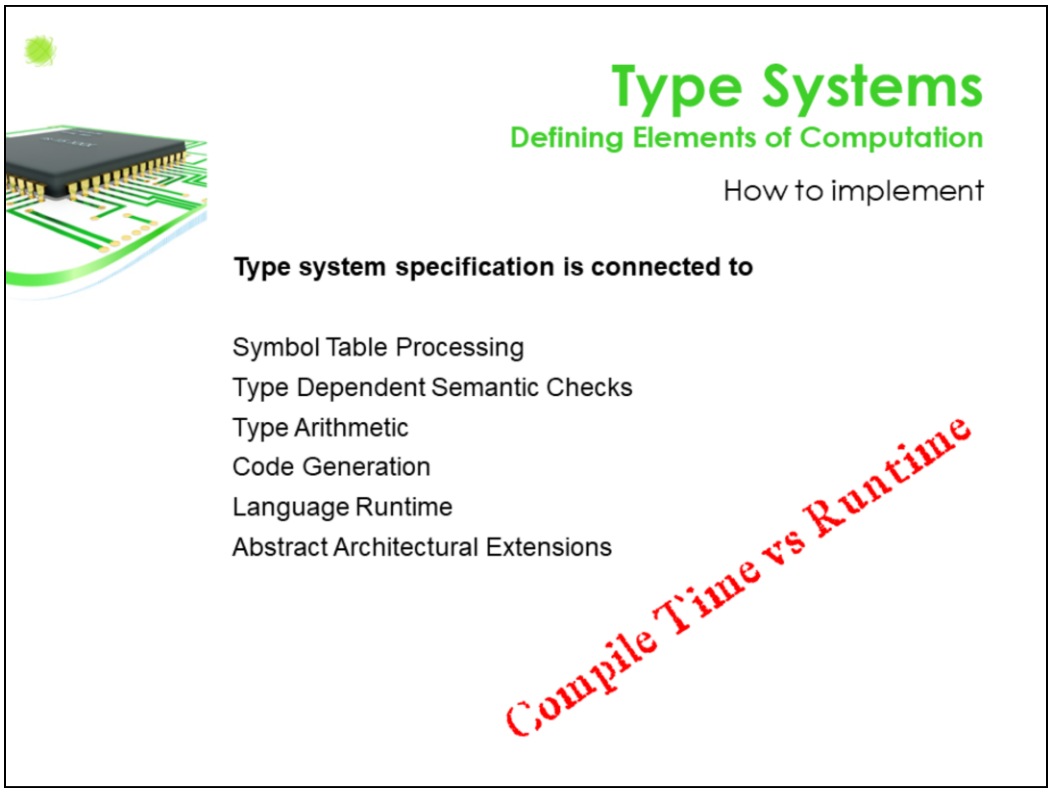
What to know about a type system

Type Manipulation

What are the basic types?
Is type derivation possible? If yes how?
Rules for type equivalence?
What are the implicit / explicit type conversion rules.
Is type inference applicable?

Basic distinctions and challenges

Static / Dynamic Type Checking
Strong / Weak Type Checking
Safe Programming
Extensible / Fixed



Type Systems

Defining Elements of Computation

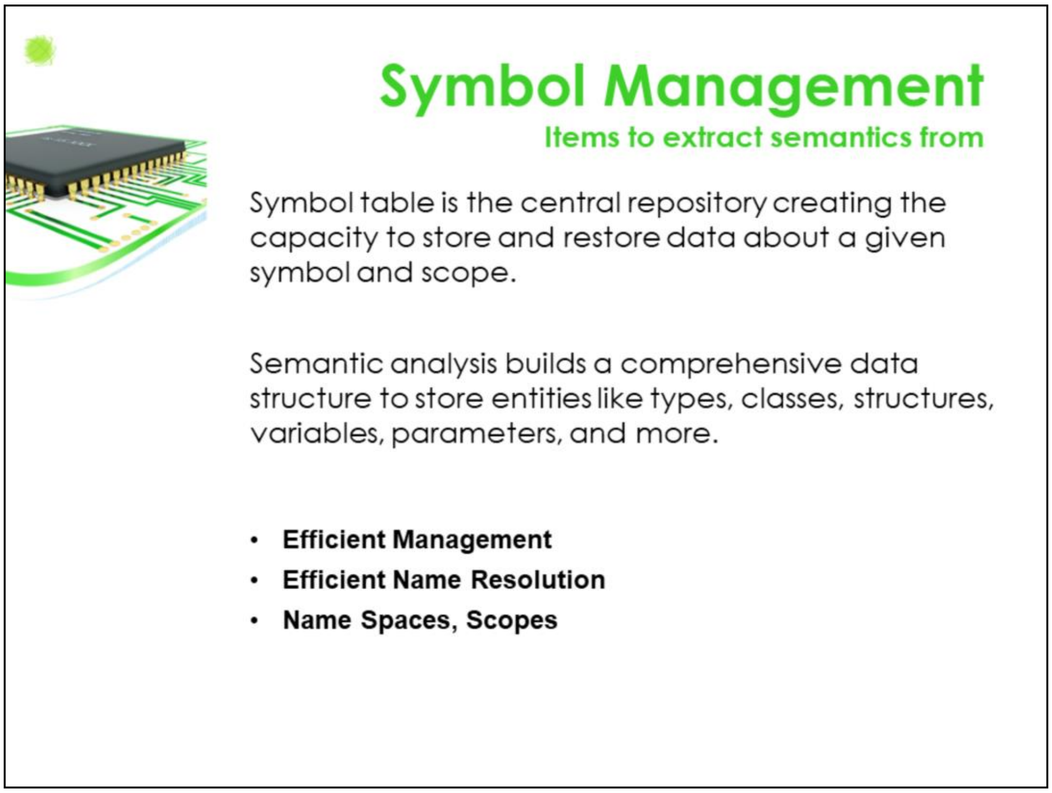
How to implement

Type system specification is connected to

Symbol Table Processing
Type Dependent Semantic Checks
Type Arithmetic
Code Generation
Language Runtime
Abstract Architectural Extensions

Compile Time vs Runtime

The properties of the type system define the way of representing it for language processing purposes. It has a deep effect on language processor implementation.



Symbol Management

Items to extract semantics from

Symbol table is the central repository creating the capacity to store and restore data about a given symbol and scope.

Semantic analysis builds a comprehensive data structure to store entities like types, classes, structures, variables, parameters, and more.

- **Efficient Management**
- **Efficient Name Resolution**
- **Name Spaces, Scopes**