Semantic Analysis

- The parser returns an abstract syntax tree (AST), a structured representation of the input program. All information present in the input program (except maybe for comments) is also present in the AST.
- Literals (integer/real/... constants) and identifiers are available as AST input attributes.
- During semantic analysis we add new attributes to the AST, and traverse the tree to evaluate these attributes and emit error messages.
- At compiler construction time we have to decide which attributes are needed, how they should be evaluated, and the order in which they should be evaluated.

Why Semantic Analysis?

1. Is the program statically correct? If not, report errors to user:
   - “undeclared variable”
   - “illegal procedure parameter”
   - “type incompatibility”
2. Make preparations for later compiler phases (code generation and optimization):
   - Compute types of variables.
   - Compute addresses of variables.
   - Store transfer modes of procedure parameters.
   - Compute labels for control structures (maybe).

Compiler Phases

- source → Lexer → tokens → Parser → errors → AST
- AST → Semantic Analysier
- IR → Optimize → IR → Machine Code Gen → ass
- IR → Intern. Code Gen → IR → Mach Code Gen
- We are here!
**Static Semantic Rules**

**Static Semantics:** ≈ type checking rules. The rules that are checked by the compiler before execution.

**Dynamic Semantics:** Rules that can only be checked when the program is run. Example: "pointer reference to NIL".

**Context Conditions:** Static semantic rules.
- Obviously, different languages have different static semantic rules. Ada, for example, allows null ranges (e.g. array [9..7] of char), while Modula-2 doesn’t.
- It’s our job as compiler writers to read the language definition and encode the rules in our semantic analyzer.

---

**Kinds of Context Conditions**

**Type Checks** We must check that every operator used in the program takes arguments of the correct type.

**Kind Checks** We must check that the right kind of identifier (procedure, variable, type, label, constant, exception) is used in the right place.

**Flow-of-control Checks** In Modula-2 an EXIT-statement must only occur within a LOOP-statement:

```plaintext
LOOP IF ... THEN [EXIT] ENDIF; END
```

**Uniqueness Checks** Sometimes a name must be defined exactly once. Example: variable declarations, case labels.

**Name Checks** Sometimes a name must occur more than once, e.g. at the beginning and end of a procedure.
Tree-Walk Evaluators...

--- Attributes ---

- Some attributes are given values by the parser. They are called input attributes.
- The attributes can store whatever we like, e.g. the types of expressions.

--- Context Conditions ---

- The context conditions are encoded as tests on the values of attributes (node.type is the type attribute of node, node.pos the line number in the source code):

```plaintext
if node.type ≠ "integer" then
  print "Integer expected at " node.pos
```

--- Tree-Walk Evaluators ---

--- Static Semantic Rules are Confusing! ---

- Check out any C++ manual...
- Ada’s semantic rules are so unwieldy that compiler error messages often contain references to the relevant sub-sub-section of the Ada Reference Manual (ARM): "Type error. See ARM section 13.2.4."
- We must organize the semantic analysis phase in a systematic way.

--- Tree-Walk Evaluation ---

The syntax analyzer produces an Abstract Syntax Tree (AST), a structured representation of the input program.
- Each node in the tree has a number of variables called attributes.
- We write a program that traverses the tree (one of more times) and assigns values to the attributes.
Tree Traversal...

- Each time we visit a node \( n \) we can
  1. Evaluate some of \( n \)'s attributes.
  2. Print a semantic error message.
  3. Visit some of \( n \)'s children.

PROCEDURE Stat(n : Node);
  IF n.Kind = Assign THEN
    Expr(n.Des); Expr(n.Expr);
  ELSIF n.Kind = IfElse THEN
    Expr(n.Expr); Stat(n.Stat1); Stat(n.Stat2);
  ENDIF
END Stat;

Tree-Walk Evaluation...

1. Evaluate some of \( n \)'s attributes.
2. Print a semantic error message.
3. Visit some of \( n \)'s children.

PROCEDURE Expr(n : Node);
  IF n.Kind = BinOp THEN
    Expr(n.LOP);
    Expr(n.ROP);
  ELSIF n.Kind = Name THEN
    (* Process n.Name *)
  ELSIF n.Kind = IntCont THEN
    (* Process n.Value *)
  ENDIF
END Expr;

Tree Traversal

- A tree-walker is a number of procedures that take a node as argument. They start by processing the root of the tree and then work their way down, recursively.
- Often we will have one procedure for each major node-kind, i.e. one for declarations, one for statements, one for expressions. Notation:
  - \( n.\text{Kind} \) is \( n \)'s node type, for example IfStat, Assignment, etc.;
  - \( n.C \) is \( n \)'s child \( C \), for example \( n.\text{expr} \), \( n.\text{left} \), etc.;
  - \( n.A \) is \( n \)'s attribute \( A \), for example \( n.\text{type} \), \( n.\text{value} \), etc.
Constant Expressions

- In many languages there are special constructs where only constant expressions may occur.
- For example, in Modula-2 you can write:
  
  ```
  CONST C = 15;
  TYPE A = ARRAY [5..C*6] OF CHAR;
  ```
  
  but not:
  
  ```
  VAR C : INTEGER;
  TYPE A = ARRAY [5..C] OF CHAR;
  ```
  
  i.e. the upper bound of an array index must be constant (value known at compile time).

Constant Expressions...

Concrete Syntax:

```plaintext
Expr ::= Add | Mul | IntConst
Add ::= Expr + Expr
Mul ::= Expr * Expr
IntConst ::= number
```

Abstract Syntax:

```plaintext
Expr ::= Add | Mul | IntConst
Add ::= LOP:Expr ROP:Expr ↑Val:INTEGER
Mul ::= LOP:Expr ROP:Expr ↑Val:INTEGER
IntConst ::= ⇐Value:INTEGER ↑Val:INTEGER
```

PROCEDURE Expr (n: Node);

```plaintext
IF n.Kind = Add THEN
    Expr(n.LOP); Expr(n.ROP);
    n.Val := n.LOP.Val + n.ROP.Val;
ELSIF n.Kind = Mul THEN
    Expr(n.LOP); Expr(n.ROP);
    n.Val := n.LOP.Val * n.ROP.Val;
ELSIF n.Kind = IntConst THEN
    n.Val := n.Value;
ENDIF
END;
```

- `n.LOP.Val` has been evaluated after `Expr(n.LOP)` has returned.
- `n.LOP.Val` is the value of `n`'s left child’s `Val` attribute.

Constant Expressions

- Constant declarations can depend on other constant declarations:
  
  ```
  CONST C1 = 15;
  CONST C2 = C1 * 6;
  TYPE A = ARRAY [5..C2] OF CHAR;
  ```
  
- Write a tree-walk evaluator that evaluates constant integer expressions.
- `IntConst` has an input attribute `Value`. We mark input attributes with a `⇐` in the abstract syntax.
- Each node is given an attribute `Val`.
- `Val` moves up the tree, so we mark it with a `⇑` in the abstract syntax.
Constant Expressions...

- Let’s extend this exercise to handle Modula-2 style constant declarations:
  
  ```
  CONST C1 = 15;
  CONST C2 = C1 * 6;
  TYPE A = ARRAY [5..C2] OF CHAR;
  ```

- We assume there is a magic function Lookup(ID) that returns TRUE if ID is a constant identifier, and a function GetValue(ID) which returns the value of this constant.
Assign ::= Expr ::= Expr
Expr ::= Expr + Expr | name | integer | real | char

Assign ::= Left:Expr Right:Expr
Expr ::= Add | Name | IntConst | RealConst | CharConst
Add ::= LOP:Expr ROP:Expr ↑Type:String
Name ::= ⇐Name:String ↑Type:String
IntConst ::= ⇐Value:INTEGER ↑Type:String
RealConst ::= ⇐Value:REAL ↑Type:String
CharConst ::= ⇐Value:CHAR ↑Type:String

PROCEDURE Assign (n: Node);
    Expr(n.Left); Expr(n.Right);
    IF NOT(n.Left.Type = n.Right.Type OR
        (n.Left.Type="REAL" AND n.Right.Type="INT"))
    THEN PRINT n.Left.Pos ":Type mismatch" ENDIF

PROCEDURE Expr (n: Node);
    IF n.Kind = Add THEN BinArith(n);
    ELSIF n.Kind = Name THEN n.Type := lookup(n.Name);
    ELSIF n.Kind = IntConst THEN n.Type := "INT";
    ELSIF n.Kind = CharConst THEN n.Type := "CHAR";
    ELSIF n.Kind = RealConst THEN n.Type := "REAL";
    ENDIF

Constant Declarations...

IsConst=FALSE
ID="C"
VAR x:INTEGER;
CONST C = 45+X;

Source
ID="C"
IsConst=TRUE
Ident
ID="X"
Val=45
ConstDecl
Expr
Add
Val=7
IntConst
Value=45
Ident
ID="X"
Val=45
IntConst
Value=45
Ident
ID="X"
Val=7
Add
IntConst
Value=45
IntConst
Value=45

Type Checking Assignments

- Write a tree-walker that type checks assignments in Pascal:
  ```pascal
  var i : integer; var r : real; var c : char;
  begin
      i := 34;
      i := i + 2;
      r := 3.4;
      r := 3.4 + i; (* OK, automatic conversion. *)
      i := r; (* Illegal. *)
      i := c; (* Illegal. *)
  end.
  
  - Assume a function lookup that returns the type of an identifier.
PROCEDURE BinArith (n: Node);
  Expr(n.LOP); Expr(n.ROP);
  IF n.LOP.Type = "INT" AND n.ROP.Type = "INT" THEN
    n.Type := "INT"
  ELSIF (n.LOP.Type = "INT" OR n.LOP.Type = "REAL") AND
    (n.ROP.Type = "INT" OR n.ROP.Type = "REAL") THEN
    n.Type := "REAL"
  ELSIF n.LOP.Type = "ERROR" OR n.ROP.Type = "ERROR" THEN
    n.Type := "ERROR"
  ELSE
    PRINT n.Pos ":Illegal operation"
    n.Type := "ERROR"
  ENDIF

Type Checking Assignments...

- n.LOP.Type & n.ROP.Type are available once we’ve returned
  from Expr(n.LOP);Expr(n.ROP).

- We use the special type value "ERROR" to avoid printing an
  error message more than once for each expression.

- Note the difference between type equivalence and
  assignability:
  1. Type equivalence is used e.g. with binary operators such
     as + and <. In Pascal, integer & real are equivalent.
  2. In Pascal, an integer can be assigned to a real, but not
     vice versa.

- In Modula-2, integers and reals are neither type equivalent
  nor assignable.
Synthesized Attributes

- **Synthesized attributes** move values up the tree (from the leaves towards the root). The value of a synthesized attribute $A$ at a node $n$ is determined from the values of $n$'s children:

  $$ n.A := f(n.Ch_1.A_1, n.Ch_2.A_2) $$

PROCEDURE ConstExpr (n:Node);
ConstExpr(n.LOP);
ConstExpr(n.ROP);
n.Val := n.LOP.Val+
n.ROP.Val;

LOOP–EXIT

- In Modula-2, the EXIT statement can only occur within a LOOP statement:

  ```
  BEGIN
  LOOP
  IF ... THEN
  WHILE ... DO
  EXIT;  
  END
  END
  EXIT  
  END
  ```
  \(\Leftarrow\) OK!

Environments

- In the previous type checking example we assumed there was a function lookup that would find the type of an identifier.

- The problem is that there may be several uses of the same name in a program, and each may have a different type:

  ```
  char x = 'c';
  int main() {
    int x = 10; {
      float x = 10.0;
      printf("%f", x);  // Which x?
    }
  }
  ```

- We'll be using environment attributes to disambiguate identifier references.
Environments...

PROCEDURE Assign (n: Node);
    n.Des.Env := n.Env;
    n.Expr.Env := n.Env;
    Expr(n.Des); Expr(n.Expr);
    IF n.Des.Type ≠ n.Expr.Type THEN
        PRINT n.Expr.Pos ":Type mismatch"
    ENDIF
END;

Environments...

- Write a tree-walk evaluator that type checks Pascal
  assignment statements.
- Let declared variables be stored in an environment
  attribute, a set of tuples of type $\text{EnvT} = \text{Name} \mapsto \text{Type}$.
- Let there be a function $\text{lookup}(E, V)$ that returns the
  type of an variable $V$ in an environment $E$.

Environments...

Assign ::= Des:Expr Expr:Expr $\downarrow$Env:EnvT
Expr ::= Add | Name | IntConst | RealConst
Add ::= LOP:ConstExpr ROP:ConstExpr $\uparrow$Type:String
$\downarrow$Env:EnvT
Name ::= $\leftarrow$Id:String $\uparrow$Type:String $\downarrow$Env:EnvT
IntConst ::= $\leftarrow$Value:INTEGER $\uparrow$Type:String
$\downarrow$Env:EnvT
RealConst ::= $\leftarrow$Value:REAL $\uparrow$Type:String $\downarrow$Env:EnvT
PROCEDURE Expr (n: Node);
  IF n.Kind = Add THEN BinArith(n);
  ELSIF n.Kind = Name THEN
    IF member(n.Env, n.Id) THEN
      n.Type := lookup(n.Env, n.Id);
    ELSE
      PRINT "Ident not declared"
      n.Type := "ERROR"
    ENDIF;
  ELSIF n.Kind = IntConst THEN
    n.Type := "INT"
  ELSIF n.Kind = RealConst THEN
    n.Type := "REAL"
  ENDIF
PROCEDURE BinArith (n: Node);
  n.LOP.Env := n.Env;
  Expr(n.LOP);
  n.ROP.Env := n.Env;
  Expr(n.ROP);
  IF n.LOP.Type = "INT" AND n.ROP.Type = "INT" THEN
    n.Type := "INT"
  ELSIF (n.LOP.Type = "INT" OR n.ROP.Type = "REAL") AND
    (n.ROP.Type = "INT" OR n.LOP.Type = "REAL") THEN
    n.Type := "REAL"
  ELSE
    PRINT n.Pos ";:Illegal operation"
    n.Type := "ERROR"
  ENDIF

Inherited Attributes

- **Inherited attributes** move values down the tree (from the root towards the leaves). They inform the nodes of a subtree of the environment (context) in which they occur.
- The value of an inherited attribute $A$ at a node $n$ is determined from the attributes of $n$'s parent $p$:
  $n.A := f(p.A_1, p.A_2)$
Peter asked me what I thought about formal semantics [...]. [...] my answer was that the best way I knew to define semantics was to use attributes whose values could be defined on a parse tree from bottom to top. [...] We also needed to include some complicated ad hoc methods, in order to get context-dependent information into the tree. So Peter asked, “Why can’t attributes be defined from the top down as well as from the bottom up?”

A shocking idea! Of course I instinctively replied that it was impossible to go both bottom up and top-down. But after some discussion I realized that his suggestion wasn’t so preposterous after all, if circular definitions could somehow be avoided.

Although attribute grammars remained at the back of my mind for several months, my next chance to think seriously about them didn’t come until I was away from home again — this time at a SIAM conference in Santa Barbara, California, at the end of November. Although the conference lists me as one of the participants, the truth is that I spent most of the whole time sitting on the beach outside the conference hotel writing a paper about “semantics of context free languages” (Mathematical Systems Theory, Vol 2 (1968), pp.127–145). [...] I spent the first day working on a test for circularity; after rejecting three obviously false starts, I thought I had found a correct algorithm, and didn’t try to too hard to find fault with it.

“Much of my story takes place in 1967, by which time a great many computer programs had been written all over the world. [...] One of the puzzling questions under extensive investigation at the time was the problem of programming language semantics: How should we define the meaning of statements in algorithmic languages? [...] I was ACM Lecturer that year [...]. My first stop was Cornell, where I spent the first weekend staying at Peter Wegner’s home in Ithaca, New York. I went with Peter to synagogue on Saturday, he went with me to a church on Sunday. We hiked outside the city in a beautiful river valley that contained many frozen water falls. But mostly we talked Computer Science.
[1970] I spent three of four pleasant days sitting under an oak tree near Lake Langunita [Stanford], writing “Examples of formal semantics” (Lecture Notes in Mathematics 188, (1971), pp. 95-96). It is clear from reading [this paper] that I was still unaware of the serious error in the circularity test [...]. I returned the galley proofs [...] to the printer on July 28; then on August 6, I received a letter from Stein Krogdahl in Norway, containing an elegantly presented counterexample to my circularity algorithm. (His letter had come by surface mail, taking six weeks to reach me, otherwise I could have alluded to the problem in [the paper].)

In 1977 I began to work on a language for computer typesetting called \textsc{TeX}, and you might ask why I didn’t use an attribute grammar to define the semantics of \textsc{TeX}. Good question.”

\textbf{The Genesis of Attribute Grammars}, Donald E. Knuth, Stanford University. LNCS 461, \textit{Attribute Grammars and their Applications}.
Summary...

- To perform semantic analysis we
  1. Build an abstract syntax tree during parsing.
  2. Decorate the AST with input attributes (literals and identifiers found in the source).
  3. Add attributes needed during semantic analysis.
  4. Traverse the tree (one or more times) to evaluate the attributes and emit error messages.
- **Designators** are the kinds of expressions that denote writable locations (i.e. **L-values**). They are common on the left hand sides of assignment statements but also occur as actual reference parameters in procedure calls.

Summary...

- The **Concrete Syntax** describes the physical layout of the language, the **Abstract Syntax** describes the logical structure of the language.
- A language’s **Static Semantics** gives the rules that a “correct” program has to obey. Static semantic rules are most often (but not always) enforced at compile-time. The **Dynamic Semantics** describes the “meaning” of a program, how it will behave at run-time.
- **Synthesized attributes** get their values from their children only. They move up the tree. **Inherited attributes** get their values from their parent only. They move down the tree.

Summary

- We use the description of the abstract syntax as a description of the structure of abstract syntax trees.
- In other words, we use context free grammars for parsing, and to describe the data structure (the AST) produced by the parser.
- There exist tools that take an abstract grammar as input and produce a AST-manipulation module (with routines for construction, traversal, and input/output of trees) as output.

Readings and References

- Read Louden:
  - *Abstract Syntax*: 109–114
  - *Attribute grammars*: 257–270
- or read the Dragon book:
  - *Abstract Syntax*: 49
  - *Type Checking*: 343–345
  - *AST Construction*: 287–290
  - *Syntax-Directed Definitions*: 280–283
  - *Recursive Evaluators*: 316–319
Summary...

- The rôle of the parser (in a multi-pass analysis compiler) is to construct an abstract syntax tree.
- We can’t always determine a visit sequence (the order in which the AST nodes are visited) that will evaluate all attributes in one pass. Then several traversals will be necessary.
- We always have to convince ourselves that we have devised a non-circular attribute evaluation scheme. We cannot have two attributes \( A_1 \) & \( A_2 \) such that \( A_1 \) must be evaluated before \( A_2 \) and vice versa.

Confused Student Email...

Don’t let the word ”abstract” in ”Abstract Syntax Tree” confuse you. There isn’t anything abstract about it at all; in fact, it is about as concrete as you can get. The idea is that performing semantic analysis on or generating code from an input program in source form (a text file) is much too hard. Therefore we build an internal representation (a data structure) of the input program during parsing, and then work on this structure. The structure happens to be a tree, because programs are naturally tree-shaped.

Confused Student Email...

Don’t let the word ”abstract” in ”Abstract Syntax Tree” confuse you. There isn’t anything abstract about it at all; in fact, it is about as concrete as you can get. The idea is that performing semantic analysis on or generating code from an input program in source form (a text file) is much too hard. Therefore we build an internal representation (a data structure) of the input program during parsing, and then work on this structure. The structure happens to be a tree, because programs are naturally tree-shaped.

Homework I

- Give an abstract syntax specification of Pascal and Modula-2 for-loops.

Pascal’s concrete syntax:

\[
\text{ForStat} ::= \text{for \ ident} := \text{expr} \to \text{expr} \text{ do Stat}
\]

Modula-2’s concrete syntax:

\[
\text{ForStat} ::= \text{FOR \ ident} := \text{expr} \text{ TO expr [ByPart]} \text{ DO StatSeq END}
\]

- The optional ByPart is an integer constant expression which gives the amount to add to the iteration variable each time we go around the loop. If omitted, the increment defaults to 1.

Confused Student Email

Should we know how to convert from concrete to abstract syntax for the exam. If so, can you indicate where I might be able to find more information on how to do this.

Don’t really know what you’re asking. Converting from concrete to abstract syntax is what the parser does. As it is parsing the input it builds the abstract syntax tree; with a bottom-up parser this is almost trivial.

I have read some of the text book, but I didn’t find what I was looking for (I think I’m looking for some sort of algorithm, or set of rules that I can use to make the conversion, like for removing left recursion, and common left factors).

There is no need to do anything like that to the abstract grammar since it is not used for parsing. The abstract grammar will often be ambiguous, left-recursive, etc, and that’s quite all right. The abstract grammar just describes the structure of the AST nodes, that’s all.
Homework II

- Give an abstract syntax for Modula-2’s `CASE`-statement, and construct the AST for the example below.

```plaintext
CASE i OF
  4 .. 7 : j := 77; |
  2, 6 .. 12 : j := 99; |
ELSE
  j := 0;
END;
```

Concrete Syntax:

```
CaseStat ::= CASE Expr OF CaseList [ELSE StatSeq] END
CaseList ::= CaseLabelList : StatSeq | CaseList | ε
CaseLabelList ::= CaseLabel | CaseLabelList | CaseLabel
CaseLabel ::= ConstExpr [.. ConstExpr]
```

Homework III

- Write a Modula-2 type checker. M2 has two mutually
assignable but inequivalent integer types: INTEGER and
CARDINAL (unsigned). Integer literals \( \geq 0 \) are either
INTEGERs or CARDINALs. Integers and reals are neither
assignable nor equivalent. TRUNC and FLOAT convert between the two.

Assign ::= Left:Expr Right:Expr
Expr ::= Add | Name | Trunc | Float | IntConst | RealConst
Add ::= LOP:Expr ROP:Expr
Trunc ::= LOP:Expr
Float ::= LOP:Expr
Name ::= <=Id:String
IntConst ::= <=Value:INTEGER
RealConst ::= <=Value:REAL