CSc 453 — Compilers and Systems Software

9 : Semantic Analysis I

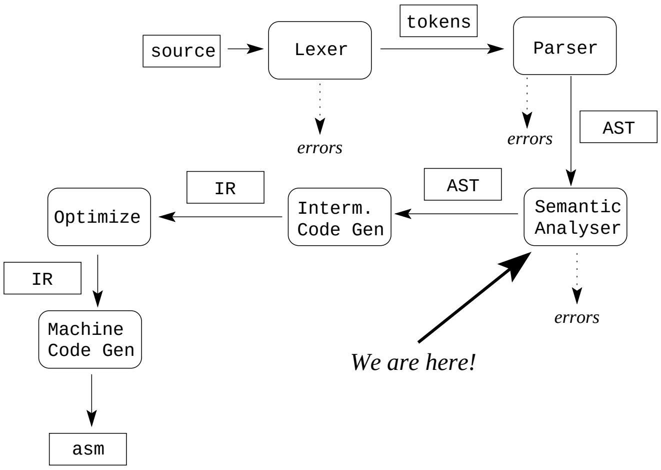
Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

September 27, 2009

1

Introduction

2 Compiler Phases



3 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.

4 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).

5 Typical Semantic Errors

```
program X;
                                "multiple declaration"
   procedure P (
     x,y : integer);
     var z,x : char;            "type mismatch"
   begin
     y := "x"
   end;                         "type name expected"
   var k : P;                   "identifier not declared"
   var z : R;
   type R = array [9..7] of char;
   var x,y,t : integer;
begin                           "empty range"
    ....
end Y.                          "wrong closing identifier"
```

6

```
program X;
   ....
begin       "too few parameters"
  P(1);                   "too many parameters"
  P(1,2,3);               "integer type expected"
  P("x",2);               "variable expected"
  R[5] := "x";            "type mismatch"
  z["x"] := 5;            "constant expected"
  case x of
     y : t := 5 |         "repeated case labels"
     3+2 : t := 9 |
     1+4 : t := 8
  end                     "boolean expression expected"
  if x then t := 4;
end Y.
```

7 Static Semantic Rules

Static Semantics: $\approx$ 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.

8 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:

```
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.

9 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.

10 Tree-Walk Evaluators...

- 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 or more times) and assigns values to the attributes.

11 Tree-Walk Evaluators...

Attributes

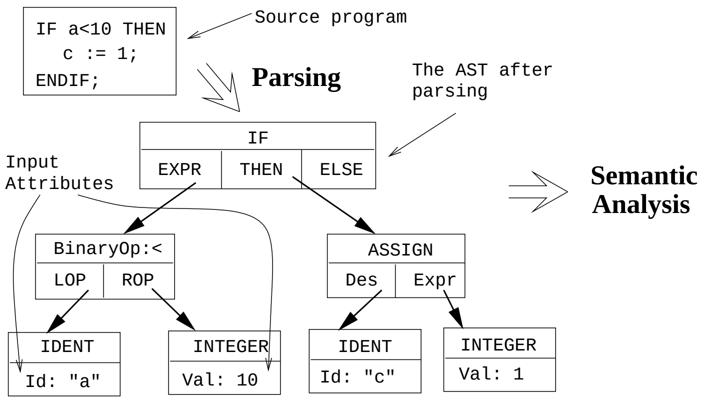
- Some attributes are given values by the parser. They are called **input** attributes.
- The attributes can store whatever we like, e.g. the types of expressions.

Context Conditions

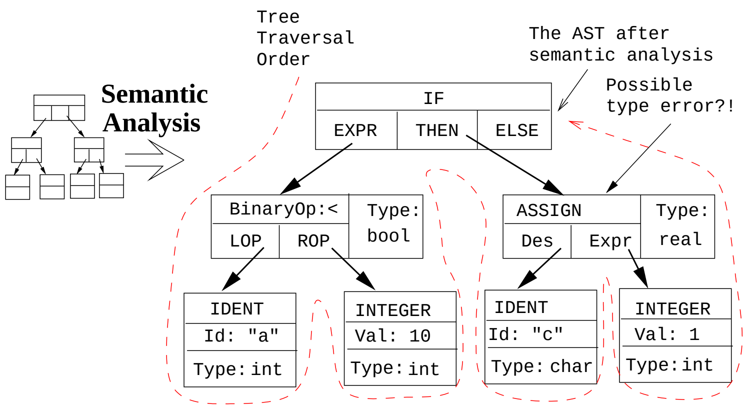
- The context conditions are encoded as tests on the values of attributes (`node.type` is the `type` attribute of `node`, `node.pos` the line number in the source code):

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

12 Tree-Walk Evaluation



13 Tree-Walk Evaluation...



14

Tree Traversal

15 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:

  $\boxed{n.\texttt{Kind}}$ is n's node type, for example `IfStat`, `Assignment`, etc.;

  $\boxed{n.C}$ is n's child C, for example n`.expr`, n`.left`, etc.;

  $\boxed{n.A}$ is n's attribute A, for example n`.type`, n`.value`, etc.

16 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;
```

17 Tree Traversal...

```
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;
```

18

Constant Expression Evaluation

19 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).

20 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 $\Leftarrow$ in the abstract syntax.
- Each node is given an attribute `Val`.
- `Val` moves **up** the tree, so we mark it with a $\Uparrow$ in the abstract syntax.

21 Constant Expressions...

Concrete Syntax:

Expr ::= Add | Mul | IntConst
Add ::= Expr <u>+</u> Expr
Mul ::= Expr <u>*</u> Expr
IntConst ::= <u>`number`</u>

Abstract Syntax:

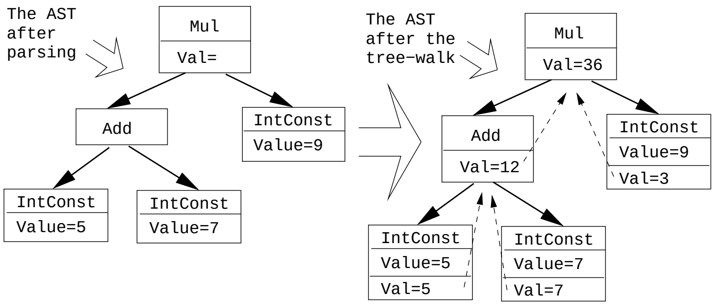
Expr ::= Add | Mul | IntConst
Add ::= LOP:Expr ROP:Expr $\Uparrow$`Val`:**INTEGER**
Mul ::= LOP:Expr ROP:Expr $\Uparrow$`Val`:**INTEGER**
IntConst ::= $\Leftarrow$`Value`:**INTEGER** $\Uparrow$`Val`:**INTEGER**

22

```
PROCEDURE Expr (n:  Node);
   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 of n's left child's Val attribute.

23 Constant Expressions...



24 Constant Declarations

- 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.

25

Concrete Syntax:

ConstDecl ::= CONST Ident = Expr

Expr ::= Expr + Expr | Ident | IntConst

IntConst ::= number

Ident ::= name

Abstract Syntax:

ConstDecl ::= ID:Ident EXPR:Expr

Expr ::= Add | IntConst | Ident

Add ::= LOP:Expr ROP:Expr ⇑Val:**INTEGER** ⇑IsConst:**BOOLEAN**

IntConst ::= ⇐Value:**INTEGER** ⇑Val:**INTEGER** ⇑IsConst:**BOOLEAN**

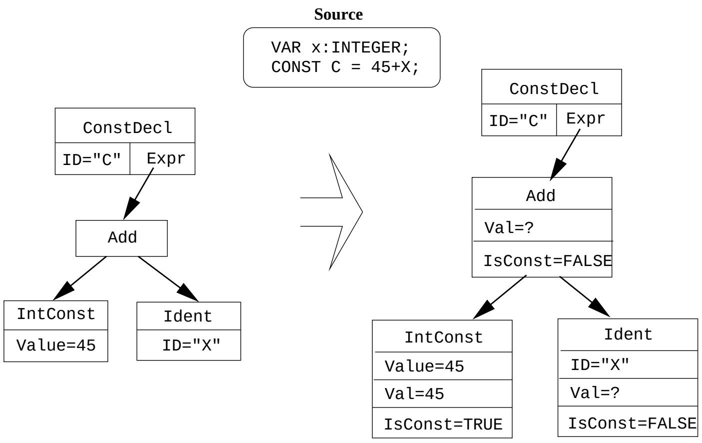
Ident ::= ⇐ID:**String** ⇑IsConst:**BOOLEAN**

26

```
PROCEDURE ConstDecl (n: Node);
   Expr(n.EXPR);
   IF NOT n.EXPR.IsConst THEN
      PRINT "Constant expression expected."
```

```
PROCEDURE Expr (n:  Node);
   IF n.Kind = Add THEN
      Expr(n.LOP); Expr(n.ROP);
      n.Val := n.LOP.Val + n.ROP.Val;
      n.IsConst := n.LOP.IsConst AND n.ROP.IsConst;
   ELSIF n.Kind = IntConst THEN
      n.Val := n.Value; n.IsConst := TRUE;
   ELSIF n.Kind = Ident THEN
      n.IsConst := Lookup(n.ID); n.Val := GetValue(n.ID);
   ENDIF
```

27 Constant Declarations. . .



28

Typechecking

29 Type Checking Assignments

- Write a tree-walker that type checks assignments in 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.

30

Concrete Syntax:

Assign ::= Expr := Expr

Expr ::= Expr + Expr | name | integer | real | char

Abstract Syntax:

Assign ::= Left:Expr Right:Expr

Expr ::= Add | Name | IntConst | RealConst | CharConst

Add ::= LOP:Expr ROP:Expr $\Uparrow$Type:**String**

Name ::= $\Leftarrow$Name:**String** $\Uparrow$Type:**String**

IntConst ::= $\Leftarrow$Value:**INTEGER** $\Uparrow$Type:**String**

RealConst ::= $\Leftarrow$Value:**REAL** $\Uparrow$Type:**String**

CharConst ::= $\Leftarrow$Value:**CHAR** $\Uparrow$Type:**String**

31

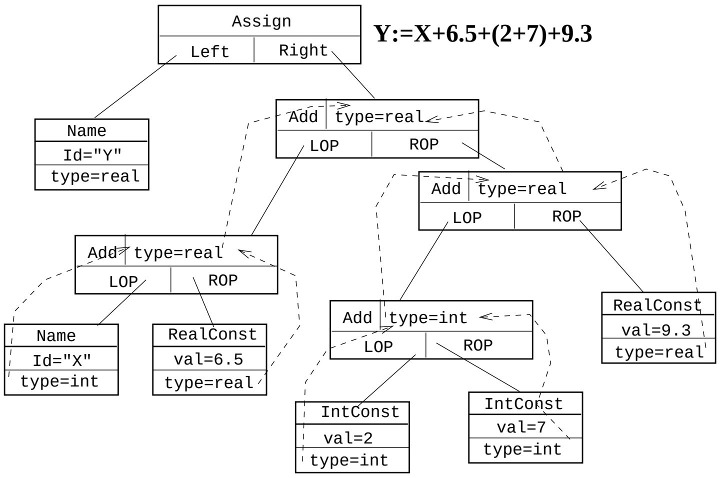
```
PROCEDURE Assign (n:  Node);
   Expr(n.Left); Expr(n.Right);
   IF NOT(n.Left.Type = n.Right.Type OR
            (n.Left.Type="REAL" AND n.Right.Type="INT"))
      THEN PRINT n.Left.Pos ":Type mismatch" ENDIF

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
```

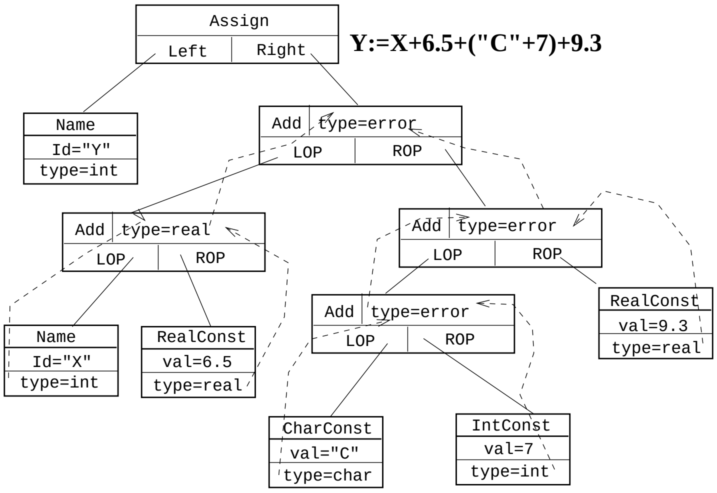
32

```
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
```

33



34



35 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.

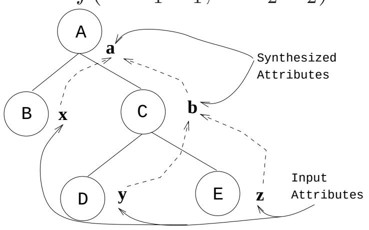
36

Synthesized Attributes

37 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:

$n.A := f(n.\texttt{Ch}_1.A_1, n.\texttt{Ch}_2.A_2)$




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

38

LOOP - EXIT

39 LOOP–EXIT

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

```
BEGIN
   LOOP
      IF ...THEN
         WHILE ...DO
            EXIT;             ⇐ OK!
         END
      END
   END
   EXIT                 ⇐ Illegal!
END
```

40

Stat ::= If | Loop | Exit
If ::= expr:Expr body:Stat ⇓InLoop:**BOOLEAN**
Loop ::= body:Stat ⇓InLoop:**BOOLEAN**
Exit ::= ⇓InLoop:**BOOLEAN**

```
PROCEDURE Stat (N:Node)
  IF n.Kind = If THEN
    Expr(n.expr); n.body.InLoop:=n.InLoop;Stat(n.body);
  ELSIF n.Kind = Loop THEN
    n.body.InLoop := TRUE; Stat(n.body);
  ELSIF n.Kind = Exit THEN
    IF NOT n.InLoop THEN
      PRINT "ERROR: EXIT not in LOOP"; ENDIF
  ENDIF
```

41 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.

42 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 `EnvT=Name` $\mapsto$ `Type`.
- Let there be a function `lookup(E,V)` that returns the type of an variable `V` in an environment `E`.

43 Environments...

Assign ::= Des:Expr Expr:Expr ⇓Env:**EnvT**
Expr ::= Add | Name | IntConst | RealConst
Add ::= LOP:ConstExpr ROP:ConstExpr ⇑Type:**String** ⇓Env:**EnvT**
Name ::= ⇐Id:**String** ⇑Type:**String** ⇓Env:**EnvT**
IntConst ::= ⇐Value:**INTEGER** ⇑Type:**String** ⇓Env:**EnvT**
RealConst ::= ⇐Value:**REAL** ⇑Type:**String** ⇓Env:**EnvT**

44 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;
```

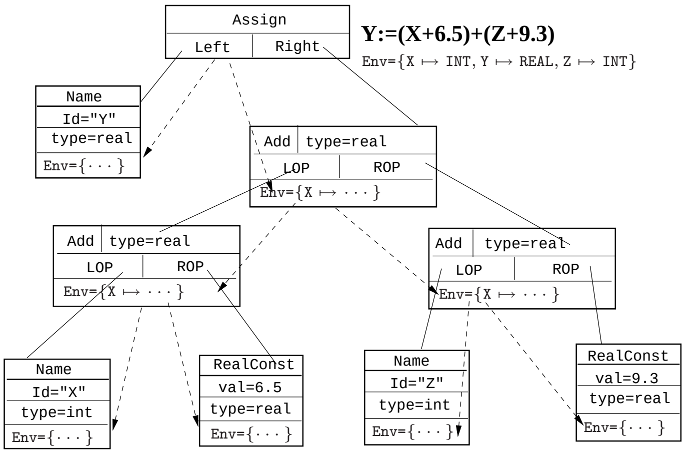
45 Environments...

```
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
```

46

```
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" ORn.LOP.Type = "REAL") AND
         (n.ROP.Type = "INT" OR n.ROP.Type = "REAL") THEN
        n.Type := "REAL"
  ELSE
        PRINT n.Pos ":Illegal operation";
        n.Type := "ERROR"
  ENDIF
```

47

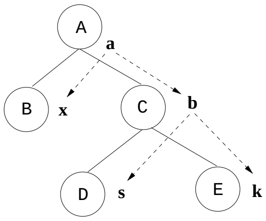


48

Inherited Attributes

49 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: $n.A := f(p.A_1, p.A_2)$




```
PROCEDURE BinArith (n:  Node);
   n.LOP.Env := n.Env;
   Expr(n.LOP);
   n.ROP.Env := n.Env;
   Expr(n.ROP);
```

50

The History of Attribute Grammars

51 History of Attribute Grammars

- What you have seen so far of attribute evaluation was know to the programming language community already in the early 1960's. It was also clear at the time that synthesized attributes alone were not enough to specify the semantics of the languages that were of concern at the time (Algol 60).
- Something more powerful was needed, and it was not clear to anyone exactly what that was.

- The person who finally came up with the answer was Donald Knuth (of Stanford University), one of the best known researchers in computer science. The following excerpts are taken from a talk he gave to a conference on attribute grammars.

52 In the Beginning... I

"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.

53 In the Beginning... II

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.

54 In the Beginning... III

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.

55 In the Beginning... IV

[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 TEX, and you might ask why I didn't use an attribute grammar to define the semantics of TEX. Good question."

The Genesis of Attribute Grammars, Donald E. Knuth, Stanford University. LNCS 461, *Attribute Grammars and their Applications*.

56 Donald Knuth

From: `http://www-cs-faculty.stanford.edu/~knuth/vita.html`

Donald E. Knuth was born on January 10, 1938 in Milwaukee, Wisconsin. He studied mathematics as an undergraduate at Case Institute of Technology, where he also wrote software at the Computing Center. The Case faculty took the unprecendented step of awarding him a Master's degree together with the B.S. he received in 1960. After graduate studies at California Institute of Technology, he received a Ph.D. in Mathematics in 1963 and then remained on the mathematics faculty. Throughout this period he continued to be involved with software development, serving as consultant to Burroughs Corporation from 1960–1968 and as editor of Programming Languages for ACM publications from 1964–1967.

57 Donald Knuth...

He joined Stanford University as Professor of Computer Science in 1968, and was appointed to Stanford's first endowed chair in computer science nine years later. As a university professor he introduced a variety of new courses into the curriculum, notably Data Structures and Concrete Mathematics. In 1993 he became Professor Emeritus of The Art of Computer Programming. He has supervised the dissertations of 28 students.

58 Donald Knuth...

Knuth began in 1962 to prepare textbooks about programming techniques, and this work evolved into a projected seven-volume series entitled *The Art of Computer Programming*. Volumes 1–3 appeared in 1968, 1969, and 1973, and he is now working full time on the remaining volumes. Approximately one million copies have already been printed, including translations into six languages. He took ten years off from this project to work on digital typography, developing the TeX system for document preparation and the METAFONT system for alphabet design. Noteworthy byproducts of those activities were the `WEB` and `CWEB` languages for structured documentation, and the accompanying methodology of Literate Programming. TeX is now used to produce most of the world's scientific literature in physics and mathematics.

59 Donald Knuth...

His research papers have been instrumental in establishing several subareas of computer science and software engineering: LR(k) parsing; attribute grammars; the Knuth–Bendix algorithm for axiomatic reasoning; empirical studies of user programs and profiles; analysis of algorithms. In general, his works have been directed towards the search for a proper balance between theory and practice.

60 Donald Knuth...

Professor Knuth received the ACM Turing Award in 1974 [...]

Professor Knuth lives on the Stanford campus with his wife, Jill. They have two children, John and Jennifer. Music is his main avocation.

Professor Knuth has an asteroid named after him:
`http://neo.jpl.nasa.gov/cgi-bin/db?name=21656`
`http://sunkl.asu.cas.cz/~asteroid/planetky/21656/eng.htm`

Professor Knuth's home page: `http://www-cs-faculty.stanford.edu/~knuth`

61

Summary

62 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

63 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.

64 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.

65 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.

66 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.

67 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.

68 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.

69

Homework

70 Homework I

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

Pascal's concrete syntax:

ForStat ::= **for** ident `:=` expr `to` expr `do` Stat
ForStat ::= **for** ident `:=` expr `downto` expr `do` Stat

Modula-2's concrete syntax:

ForStat ::= FOR ident := expr TO expr [ByPart] DO StatSeq END

ByPart ::= BY ConstExpr

- The optional **BY**-part 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.

71 Homework II

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

```
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]

72 Homework III

- Write a Modula-2 type checker. M2 has two mutually assignable but inequivalent integer types: `INTEGER` and `CARDINAL` (unsigned). Integer literals ≥ 0 are either `INTEGER`s or `CARDINAL`s. 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**

73 Homework IV

- Write a concrete grammar that describes the syntax we have been using to describe our abstract grammars.
- The concrete grammar should describe

  Rules LHS ::= RHS

  Choice LHS ::= CH1 | CH2 | $\cdots$

  Children LHS ::= Name:Child

Input Attributes LHS ::= $\Leftarrow$`Attr`:**Type**

Synthesized Attributes LHS ::= $\Uparrow$`Attr`:**Type**

Inherited Attributes LHS ::= $\Downarrow$`Attr`:**Type**