

Honors Compiler, G22.3130

Fall 2007: Semantic Analysis Assignment

Due Thursday 11.22.07

1 General Description

The next component of your compiler to be implemented is the semantic analyzer. At this point, its function will be the following:

- Ensure the type correctness of the source program and incorporate type information into the abstract syntax tree.
- Incorporate scope information into the abstract syntax tree and to differentiate between occurrences of the same identifier in the program that denote references to different objects.
- Catch errors in the source program due to references to undeclared names, and multiple declarations of the same name in the same scope.

You may choose to implement the semantic analyzer as a separate phase that traverses the abstract syntax tree (AST), or as part of your parser by adding additional semantic actions to your existing Bison/JavaCC specification. The routines provided by the semantic analyzer will be used to associate additional attributes with various nodes of the AST. In order to do this, a symbol table should be created (if you have not already done so) and the data structures used for the AST must be extended to include these attributes. At this point, the following attributes must be associated with each name in the program:

- The *kind* of entity this identifier names. This can be any of the following: *program*, *type*, *procedure*, *function*, *variable*, *formal_parameter*, *field_designator* (within record).
- The type of the entity named by the identifier. This is relevant for the case that the identifier is of kind *type*, *function*, *variable*, *formal_parameter*, or *field_designator*. The value of the type attribute is a *type expression* whose structure is described below.
- The scope of the declaration of the identifier. Since the source language does not have nested procedures, there are only two levels of scoping – global and local. All procedure and type names are global and variable names can be either global or local. A local scope is always associated with a procedure (or function). Under this attribute, please specify the procedural unit which defines the scope.

The type information can be represented by tree representations of type expressions (such as those already in the abstract syntax tree) or the more concise representations (value numbering, etc.) described in the text. Note that nodes representing function and procedure names must also have type information associated with them (i.e. types of parameters, return types).

2 Symbol Table

The semantic analyzer should create a symbol table so that different attributes can be associated with different declarations of variables with the same name. One possible organization of the symbol table entries might be:

Entry No.	Name	Kind	Type	Scope
1.	y	<i>variable</i>	Symb[4]	P_1
2.	x	<i>variable</i>	integer	P_1
3.	x	<i>variable</i>	Ast[1]	Global
4.	foo	<i>type</i>	Ast[2]	Global
5.	z	<i>field_designator</i>	integer	Global

where the abstract syntax tree (AST) contains the following fragment:

Node No.	Node Type	Child-1	Child-2	Child-3
...				
1.	ARRAY	3	5	integer
2.	RECORD	Ast[3]	-	-
3.	<i>FieldList_element</i>	integer	Symb[5]	-
...				

In this tables, the symbol table contains 4 entries, for the identifiers y, x, foo, and z. y is a local variable (in procedure P_1) of type “foo”. We refer to the type “foo” through a pointer to the appropriate entry (entry 4) in the Symbol Table. Identifier x names both a local variable of type integer in procedure P_1 and a global variable. The type of the global variable x is an integer array with declaration “array[3..5] of integer”. To specify this compound type we point to the appropriate entry (entry 1) in the AST table. The identifier “foo” is the name of a record type corresponding to the declaration “record z: integer end”. The description of this type resides at entry 2 of the AST. Identifier z is a field designator in the record comprising type “foo”.

In the representation above we use “Symb[n]” and “Ast[k]” in order to refer to entries n and k in the Symb and AST tables. In the actual C implementation you could use pointers instead.

3 The Abstract Syntax Tree

The attribute information (“attribute nodes”) incorporated into the symbol table should also be incorporated in the abstract syntax tree (AST). In particular, all occurrences of a name

(variable references, references to type names, etc.) in the AST should share an attribute node. The attribute node for an identifier will have been created when the identifier was declared, and will have been installed in the symbol table. Thus, any reference to that identifier can point to the attributed node. For example, the AST representation for the program fragment:

```
var x: integer;
begin
    x := x + 1;
end;
```

might be:

	Node No.	Node Type	Child-1	Child-2	Child-3	Child-4
	1.	<i>Block</i>	Ast[2]	Ast[4]	-	-
	2.	<i>VariableDeclarations</i>	Ast[3]	-	-	-
	3.	<i>VariableDeclaration</i>	integer	Symb[1]	-	-
Ast:	4.	<i>StatementSequence</i>	Ast[5]	-	-	-
	5.	<i>AssignmentStatement</i>	Ast[6]	Ast[7]	-	-
	6.	ID	Symb[1]	-	-	-
	7.	<i>SimpleExpression</i>	-	Ast[6]	Ast[8]	Ast[9]
	8.	<i>AddOp</i>	PLUS	-	-	-
	9.	INTEGER	1	-	-	-

assuming that the symbol table contains the entry

Symb:	Entry No.	Name	Kind	Type	Scope
	1.	x	<i>variable</i>	integer	global

A graphical representation of this abstract syntax tree is presented in Fig. 1.

These drawings are just schematic, of course. You are free to use whatever representation you see fit. Just be sure that attribute nodes are shared appropriately. In subsequent phases (e.g. during code generation) these attribute nodes will need to contain other necessary information (locations, offsets, sizes, register info, etc.).

4 The Type System

The source language contains three predefined types, integer, string, and boolean, and two predefined type constructors, array and record. Type equivalence is defined as follows:

- Two types named T1 and T2 are equivalent if T1 = T2 (i.e. they are the same type name) or if T1 was defined as follows

```
type T1 = T3;
```

where T2 and T3 are equivalent, or if T2 was defined as follows

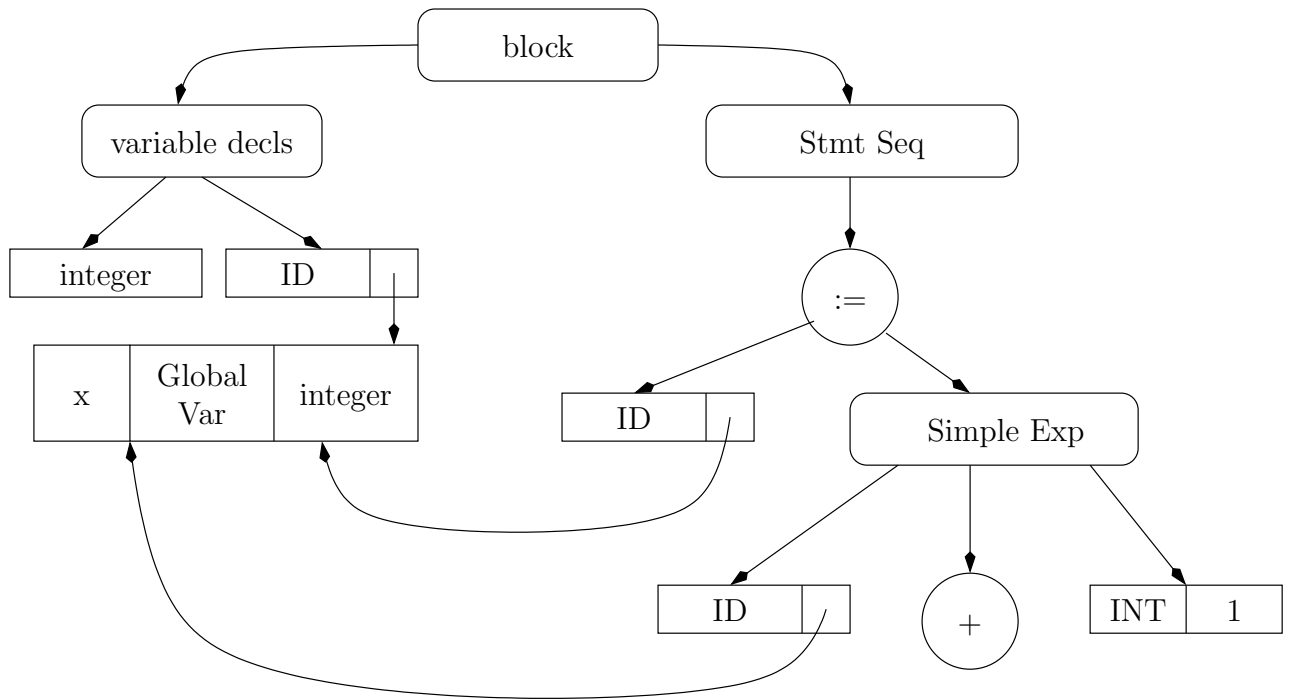


Figure 1: An abstract Syntax Tree

type T2 = T3;

where T1 and T3 are equivalent. For this definition, the basic types (*integer*, *string*, and *boolean*) are considered named types.

- Two anonymous types are equivalent if they are constructed from the same type constructors and their components are equivalent. For records, this means that the names and order of the fields must be identical and their corresponding types must be equivalent. For arrays, the bounds must be identical and the element types must be equivalent.

Type compatibility is defined in the usual way:

- The types of the corresponding formal parameters and actual parameters for procedure and function calls must be equivalent.
- The types of the LHS and RHS of an assignment statement must be equivalent. Notice that there is a special case in which the name of a function can be assigned to (in which case the result type of the function and the type of the RHS must be equivalent).
- The predefined operators (+,*, etc.) and literals (123, "hello") carry their usual types

Notice that some new entries must be entered in the symbol table before compilation begins, namely the entries for *integer*, *string*, *boolean*, *true*, *false*, and other predefined identifiers.

5 Implementation of Type Equivalence

This kind of equivalence is easy to implement. If, in your symbol table, the symbol A simply points to a type descriptor (attribute node), then if your type checker encounters:

```
type B = A;
```

it can install B into the symbol table, pointing to the same type descriptor as A.

6 The Namespace

There is only a single namespace where type, variable, and procedure names reside. This means that within a given scope (either global or within a procedure), there can be at most one declaration of each name. For example, one could not declare a type named *bar* and a procedure named *bar* in the same program.

7 Deliverables

As part of your submission deliver the source program that is supposed to construct the symbol table, the abstract syntax tree enhanced with the semantic attributes discussed in this assignment, and perform type checking.

Run your program on each of the provided test programs. For each test program print and submit the following outputs:

7.1 Symbol Table

The symbol table produced by your compiler. Print it in the format presented in Section 2. For non-basic types, print the type of each typed entry in a format compatible with the following inductive definition:

- The basic types *integer*, *string* and *boolean* are types.
- If t_1, \dots, t_k, t are types, the so are: $t_1 \times \dots \times t_k$ (corresponding to records), $array[n_1..n_2]$ of t , and $t_1 \times \dots \times t_k \rightarrow t$ (corresponding to a function).

For example, if we have the declaration

```
x : array[1..10] of record a : integer; b : string end;
```

then the type of “x” should be printed as

```
array[1..10] of [integer X string]
```

7.2 Abstract Syntax Tree

For each program print the abstract syntax tree in the format of Section 3, where all ID nodes point to entries in the symbol table. Use entry numbers for pointers into both the symbol table and the AST.

7.3 Type Checking

Perform type checking on each of the test programs. Among other things, check that:

- Every accessed identifier is declared in the relevant scope.
- Every identifier is declared at most once in each scope.
- The actual arguments and formal parameters have matching types.
- The types of left-hand side and right-hand side of an assignment match.
- The operands of each operation are of the right types (e.g., boolean operations such as OR and AND are only applied to boolean arguments).
- The conditions in an “if” or a “while” statements are of type *boolean*.
- Subscripts are of type *integer*.
- The expressions appearing in a “for” statement are of type *integer*.

For each identified type-checking error, print an error message, identifying the line number at which the error occurred and providing information about the violating variables or expressions.