1

Static Semantics

- So far, our parser correctly parses syntactically correct programs
- This Pascal fragment is syntactically correct, but contains a semantic error.
  - We need to implement the type system

```
var i : integer;
begin
   i := 2.0 / 3.0;
end;
```

- Semantic consistency in a programming language is usually implemented via typing entities and checking for the consistent use of the entity
- Implementing the Static Semantics requires:
  - implementation of an identifier table to store information about identifiers
  - implementation of a type module which checks the usage of the identifiers

High Point University

2

Implementing the Type System

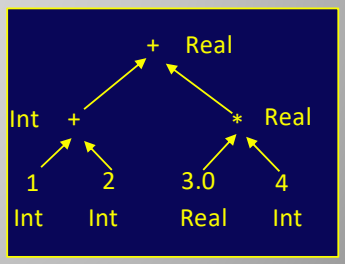
- Before proceeding further, we must note that the type checking we are required to implement will be spread throughout the parser.
  - This is the way we build the recursive descent compiler – by a process of enrichment.
- Implementation of the type system requires two things:
  - A specification of the type rules of the language.
  - A suitable representation of the type of entities in the compiler.
- The specification of the type rules will normally be provided by the programming language designer.
  - Type rules can be expressed in many different ways.
- The representation of types by the compiler may at first seem strange, but it is no different from the representation of any other piece of information.

HIGH POINT UNIVERSITY

3

Arithmetic Expressions

- Consider simple arithmetic expressions:
  - this involves the parsers: Expression, Term & Factor.
- A simple evaluator of arithmetic expressions can be built by using a stack.
  - Operands are pushed on to the stack.
  - Operations are performed on the top of the stack.
- If we note the type of the objects, rather than the value we can deduce the type structure.
  - In real terms, we make the sub–parsers E,T & F return the type of the current sub–expression (rather than void).

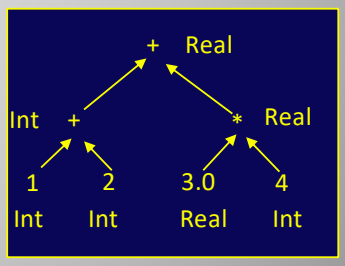


HIGH POINT UNIVERSITY

4

Arithmetic Expressions

- Type information, like the value of the expression is propagated back up the tree.
- We don't generate an explicit parse tree
  – our tree is represented by the run–time stack.
  - Parser methods often return a value such as the type of the exporession being evaluated.



HIGH POINT UNIVERSITY

5

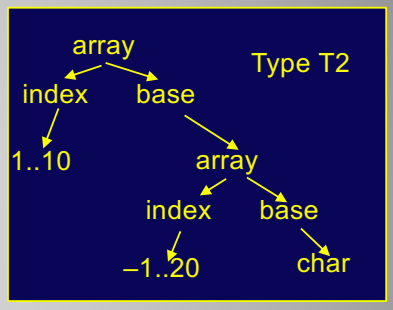
Representing Types

- Types aren't simple – they are structured – and so is their representation
- Example:

```
Type t = array[1..10] of  char;
```

- Our representation would need to record:
  - the fact that it is an array
  - the index type & range
  - the base type of the array
- These descriptions are recursive:

```
Type t2 = array[1..10] of array[-1..20] of char;
```



- Checking types then involves traversing the representation structure
- The representation structure is built out of:
  - the base types
  - the type constructors

HIGH POINT UNIVERSITY

6

Type Nuts & Bolts

- Note the use of the variant record (discriminated union) type here.
  - Programming languages which support higher order functions need to be able to describe function/ procedures in the type system.

```
Type types is (int, real, char, bool, pointer, arry, recrd, set, file);
Type type_rep;
Type type_rep_ptr is access type_rep;
Type type_rep is record (ty : types) is
        ...
        case ty is
                when pointer => pointer_to : type_rep_ptr;
                when arry => base_ty : type_rep_ptr;
                                range_ty : type_rep_ptr;
                when set => set_of : type_rep_ptr;
                ...
        end case;
end record;
```

  - Simple languages (such as that in your assignment) have no structured types and hence are somewhat simpler.

HIGH POINT UNIVERSITY

7

Limitations of C++

- C++ does not support discriminated unions as is found in some other languages such as Ada.
- This means that you have to create a struct or a class with all of the fields even though some fields are invalid for some types.
  - See the token class in the starter kit for your project. Not every token has an integer_value, real_value, string_value and identifier_value.
  - Care is necessary to ensure a field is not illegally referenced.
  - Can lead to some strange errors!

HIGH POINT UNIVERSITY

8

Type Nuts & Bolts

- This is an example of a type comparison routine which implements structure equivalence.
  - Some languages use a simpler name equivalent system.
  - Some languages use a mixture of the two!

```
function comp_types(t1,t2 : type_rep_ptr) return boolean is
  begin – –  comp_types
    if t1.ty /= t2.ty then
      return false;
    else
      case t1.ty is
        when pointer => return comp_types(t1.pointer_to, t2.pointer_to);
        when arry => return comp_types(t1.range_ty, t2.range_ty) and then
          comp_types(t1.base_ty, t2.base_ty);
        ...
      end case;
    end if;
  end comp_types;
```


9

Type Checking

- Look at the body of the parser for Term.

```
lhs, rhs : Type_Rep;
begin – – Term
  lhs := Factor;
  while member(symbol, mult) loop
    op := symbol;
    rhs := Factor;
    case op is
      when div => if comp_types(lhs, rhs)
                    and comp_types(lhs, TYPE_INTEGER) then
                    ...
                  else
                    syntax("Type error");
                  end if;
      when and => if comp_types(lhs, rhs)
                    and comp_types(lhs, TYPE_BOOLEAN) then
                    ...
                  else
                    syntax("Type error");
                  end if;
    ...
```


10

Storing Type Information

- Information about user defined types must be stored somewhere.
- Two choices:
  - a type table which maps type names to type structures
  - a type table merged into the identifier table
- Different languages require different approaches, e.g.:
- Ada:

```
type foo is (a,b,c);
foo : foo;
```

- Pascal would disallow this declaration : it stores the type identifiers in the identifier table.
- SUMMARY:
  - Managing the type information is easy.
  - Formal type rules describe where you need to put type checking.
  - Types are represented internally using a tree structure.

HIGH POINT UNIVERSITY

11

The identifier table

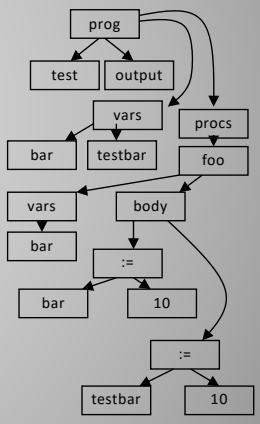
- Management of identifiers is one of the most pivotal aspects of the compiler.
- The identifier table stores:
  - Each identifier.
  - Information about that identifier:
    - Kind of object: variable, constant, parameter, ...
    - Type of object.
    - Address information etc.
- The organization of the identifier table depends on the language being used
  - in particular, we need to consider the scope rules of the language.
  - In a block structured language (Ada, Pascal, etc) when we enter a new scope level we can declare entities which are homomorphs to entities at outer scope levels.
  - In a language like FORTRAN, redeclaration would be an error (there is only one scope level).

HIGH POINT UNIVERSITY

12

Identifiers

- We can now implement type information, but we need to determine which identifiers are valid (and their meaning) in our program.
  - We can use an abstract syntax tree:
- ```
  Program test(output);
  var bar, testbar : integer;
  procedure foo;
  var bar : integer;
  begin (* foo *)
    bar := 10;
    testbar := 10;
  end; (* foo *)
  begin (* test *)
    foo;
  end.
  ```



HIGH POINT UNIVERSITY

13

The Identifier table

- To access the information needed, we need an auxiliary data structure
  - We need to know what things are in scope
- Aside: Scope rules
- The scope rules of the language define the *referencing environment* the module sees.
- Pascal has a scope environment that chains back through the nested modules
  - In the example, the local identifier `bar` and the main program's `testbar` identifiers are in scope inside procedure `foo`.
  - The identifier table also needs to implement the scope rules of the language.
- Observation: number the scope regions, the scopes *nest*.
  - We have only *one* thing contributing to the referencing environment at any given level
  - Our scope regions are therefore a simple linear list of blocks in scope.

HIGH POINT UNIVERSITY

14
  ```

Scope

```
Program Scope(output);
var bar, testbar : integer;

  procedure one;
    var bar : integer;

    procedure two;
    begin (* two *)
         bar := 1;
    end; (* two *)

  begin (* one *)
    bar := 10;
    two;
    testbar := 10;
  end; (* one *)

    procedure three;
    begin (* three *)
        one;
    end; (* three *)

begin (* scope *)
  three;
end.
```

HIGH POINT UNIVERSITY

15

Scope

```
Program Scope(output);
var bar, testbar : integer;

  procedure one;
    var bar : integer;

    procedure two;
    begin (* two *)
         bar := 1;
    end; (* two *)

  begin (* one *)
    bar := 10;
    two;
    testbar := 10;
  end; (* one *)

    procedure three;
    begin (* three *)
        one;
    end; (* three *)

begin (* scope *)
  foo;
end.
```

RE: bar(one), testbar(Scope), one, two

RE: bar(one), testbar(Scope), one, two

RE: bar(Scope), testbar(Scope), one, three

RE: bar(Scope), testbar(Scope), one, three

HIGH POINT UNIVERSITY

16

Scope

- Note carefully how the local declaration of bar in procedure one hides the more global version in the main program.
  - But the other non-hidden entities are still visible, of course.
- Implementing this is not very hard!
- Look at the compilation process:
  - We see, in order, the following blocks:

    Scope, one , two, one, three, scope
  - Including the bodies!

HIGH POINT UNIVERSITY

17

Scope

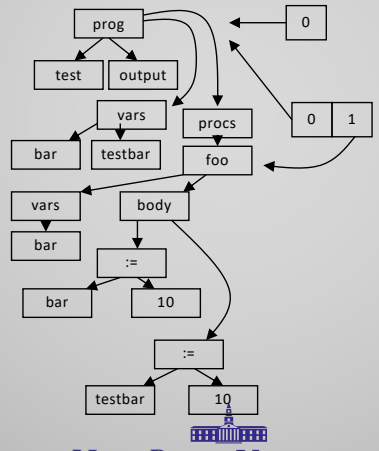
- While compiling one, we need Scope and one.
  - While doing two, we need Scope, one and two.
  - While doing three we need Scope…
- We can implement this by using an array structure where the n^{th} element has the module at scope level n
  - We simply search this from end to beginning!

HIGH POINT UNIVERSITY

18

The Identifier Table

- As we traverse the abstract syntax tree, we need to resolve each identifier – build up the idtable array as we enter each new scope region.




```
Program test(output);
var bar, testbar : integer;
procedure foo;
var bar : integer;
begin (* foo *)
  bar := 10;
  testbar := 10;
end; (* foo *)
begin (* test *)
  foo;
end.
```

HIGH POINT UNIVERSITY

19

Using the IDtable

- We discover an entity is undeclared by falling off the start of the idtable array
  - At this point, we need to enter the id into the idtable with type set to unknown
    - Prevent spurious errors from happening on every future reference
- Once we have determined which identifier it is, the lookup() method should return a reference to the identifier.

HIGH POINT UNIVERSITY

20

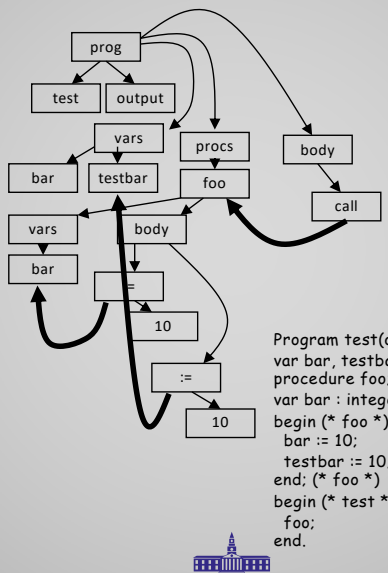
Using the IDtable

- This reference should be written into the tree so that when we perform further work on the tree, we can directly locate what entity is being referenced.
  - You don't strictly need to do this, of course.
    - We could just use the lookup() method again, but this is somewhat wasteful
- IMPORTANT – our AST is now a DAG

HIGH POINT UNIVERSITY

21

Tree with resolved IDs



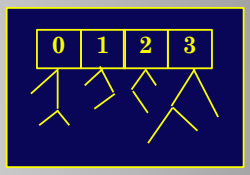

```
Program test(output);
var bar, testbar : integer;
procedure foo;
var bar : integer;
begin (* foo *)
  bar := 10;
  testbar := 10;
end; (* foo *)
begin (* test *)
  foo;
end.
```


CCP 2002 HIGH POINT UNIVERSITY

22

Id–table Organization

- Since we need to search the idtable frequently, we need an efficient storage regime.
  - We will use a sorted binary tree.
- Such a structure allows us to search in *O(logn)* time.
- We need a means of coping with multiple instances of an identifier at different scope levels, however.
  - Use an array of binary trees, indexed by scope level.



HIGH POINT UNIVERSITY

23

Id–table Organization

- Implementing the scope rules is now easy:
  - We search for the identifier at the current level, and if not found we search the array backwards.
  - If still not found, it hasn't been declared!
- Question: how big an array do we need?
  - Not big, 10 to 12 is ample.
  - We can use an unconstrained array to avoid imposing a fixed limit on the number of scope levels!

HIGH POINT UNIVERSITY

24

Id–table Contents

- What information is stored in the identifier table?
  - the name of the identifier (this is the search key!)
  - the kind of identifier: constant, type, variable, proc, func, etc
  - depending on the kind of identifier we will have other things:
    - value (if a constant)
    - list of parameters (if a function or a procedure)
    - result type (if a function)
    - the type of the identifier
    - the address of the identifier (if appropriate)
    - the scope level of the identifier
    - etc etc

HIGH POINT UNIVERSITY

25

I d–Table details

- To manage our ID table, we will implement an Abstract Data Type/class.
  - The ADT will export some operations on the table entries.
  - The ADT will make use of a private variable which will refer to an idtable entry.
- Our idtable abstraction will maintain the scope level information as well.

```
with types, scanner;
use types, scanner;
package id_table is
    type idtable is private;
    idtable_error : exception;

    procedure enter_new_scope;
    procedure exit_scope;
    function lookup(s : str_ptr) return idtable;
```

HIGH POINT UNIVERSITY

26

I d–Table details

- All of our compiler code should be self–checking: when the idtable module detects an inconsistency, it raises an exception.
- In addition to the 3 operations implemented above, there is an operation for entering a new identifier into the table and a collection of operations which return pieces of information about a particular identifier.

HIGH POINT UNIVERSITY

27

Using the Id–table

- By using the idtable & the type system, we are able to choose the right alternative in statement
- Note the use of the selector operations kind and tipe.
  - Remember, the idtable is an abstract data type (class in C++)!

```
...
if symbol = identifier then
    id := lookup(token.ident_value);
    if (kind(id) = variable) or (kind(id) = func) then
        mustbe(becomes);
        if not type_equal(tipe(id), expression) then
            syntax("Type mismatch");
        end if;
    elsif kind(id) = proc then
        ...
    end if;
elsif have(begin_sym) then
```


HIGH POINT UNIVERSITY

28

Using the Id–table

- QUESTION: What if the identifier wasn't yet declared?
  - Won't "lookup" fail??
  - NO! Our lookup routine will insert the identifier into the table & issue an error message when this happens!
  - The inserted ID is of type "unknown" - used to help error recovery.

HIGH POINT UNIVERSITY

29

Using the Id–table

- Extracting information from the id–table is easier than putting it in!
- Consider the block parser in Pascal.
  - When we process a variable declaration:

```
var i, j, k : integer;
```

  - We don't know the type of the identifier `i` until we process the entire declaration!
  - There are a number of solutions to the problem, but the best is to enter the identifiers into the table (as they are encountered) and to then "fix–up" the type information.
  - Doing it this way, will cause errors about duplicate declarations to appear in the correct place.
  - Implementation of this scheme requires us to maintain a list of idtable type objects in the parser and the provision of a "fix–up" operation in the identifier table ADT.
    - Different languages require different sorts of fix–ups, Pascal and *lille* needs them for variable declarations and procedure/function declarations.

HIGH POINT UNIVERSITY

30

Using the Id–table

```
– function enter_id(id : str_ptr; kind: id_kind := unknown;
                    ty : lang_type := type_unknown) return idtable;
...
    list := null;
    loop
        if symbol = identifier then
            add_to_list(list, enter_id(id => token.ident_value,
                                       kind => variable); get_token;
        else
            syntax("identifier expected");
        end if;
        ...
    end loop;
    if symbol = identifier then
        ty := lookup(token.ident_value); ...
    end if;
    while list /= null loop
        this_id := next_from_list(list);
        fix_up(this_id, ty);
        next(list);
    end loop;
```

Note how the use of default values makes this operation simpler

31

Using the Id–table

- C++ has a limited form of default parameters. It can be used to some advantage, but it is not as versatile as the mechanism permitted in some other languages.

HIGH POINT UNIVERSITY

32

Simple Coding tips for the id–table

- There are many coding tips for making life easier…
  - By choosing sensible default values for parameters, you can avoid having to make several similar procedures.
    - enter_id for example would also have several parameters for the value of a constant – all with defaults, and it is only when the object being entered is of kind = constant that these are used. You choose the appropriate one based on the type of the constant being entered!
  - Make all your idtable code self checking!
  - Implement a dump procedure which will print the identifier table
    - Make this callable from the parser.
    - You might like to make it be able to dump just the current scope level as an option. VERY useful wen debugging!

HIGH POINT UNIVERSITY

33

34