Semantic Analysis

What is The Semantics of A Program?

- Syntax
  - How a program looks like
  - Textual representation or structure
  - A precise mathematical definition is possible
- Semantics
  - What is the meaning of a program
  - Harder to give a mathematical definition

Why Do Semantic Checking?

- Make sure the program confirms to the programming language definition
- Provide meaningful error messages to the user
- Don’t need to do additional work, will discover in the process of intermediate representation generation

Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language's static semantics, which are checked at compile time
- *Runtime checking*: dynamic semantics are checked at run time by special code generated by the compiler

Static Checking

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

Type checking

- We may not do all type checking at compile-time.
- A *type system* is a collection of rules for assigning type expressions to the parts of a program.
- A *type checker* implements a type system.
- A *sound type system* eliminates run-time type checking for type errors.
- A programming language is *strongly-typed*, if every program its compiler accepts will execute without type errors.
  - In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
  - Ex: **`int x[100];`** … **`x[i]`** → most of the compilers cannot guarantee that i will be between 0 and 99

Type Checks, Overloading, Coercion, and Polymorphism

```
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c+d;     // FAIL

*d = a;      // FAIL

a = op(d);      // OK: overloading (C++)

a = f(d);       // OK: coersion

vector<int> v; // OK: template instantiation
```

Flow-of-Control Checks

```
myfunc()
{ …
  break; // ERROR
}
```

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

```
myfunc()
{ …
  switch (a)
  {
    case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

Uniqueness Checks

```
myfunc()
{
  int i, j, i; // ERROR
  …
}
```

```
cnufym(int a, int a) // ERROR
{
   …
}
```

```
struct myrec
{
  int name;
};

struct myrec // ERROR
{
  int id;
};
```

Name-Related Checks

```
LoopA: for (int I = 0; I < n; I++)
       { …
         if (a[I] == 0)
           break LoopB;
         …
       }
```

```
function Minimum (A, B : Integer) return Integer is
begin
   if A <= B then
      return A;
   else
      return B;
   end if;
end Minimum;
```

One-Pass versus Multi-Pass Static Checking

- *One-pass compiler*: static checking for C, Pascal, Fortran, and many other languages can be performed in one pass while at the same time intermediate code is generated
- *Multi-pass compiler*: static checking for Ada, Java, and C# is performed in a separate phase, sometimes requiring traversing the syntax tree multiple times

Type Expressions

- Type expressions are used in declarations and type casts to define or refer to a type
  - Primitive types, such as int and bool
  - Type constructors, such as pointer-to, array-of, records, functions
  - Type names, such as typedefs in C and named types in Pascal, refer to type expressions

Type Expressions

- Basic type: *boolean*, *char*, *integer*, *real, void; type_error* to signal a type error
- Named type (typedef)
- Constructed types:
  - Arrays: *array*(*I* , *T*)

    var A: array[1..10] of integer;

    array(1..10 , *integer*)
  - Cartesian products

    $T_1 \times T_2$
  - Records

```
type row = record
   address: integer;
   lexeme: array [1..15] of char
 end;
 var table: array [1..101] of row;
```

  $record((\textbf{address} \times integer) \times (\textbf{lexeme} \times array(1..15\ ,\ char)))$

  - Pointers

    var p: ^row

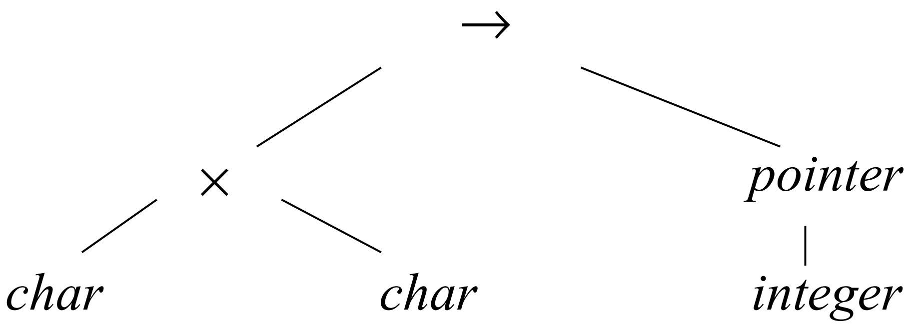
    pointer(row)
- Functions

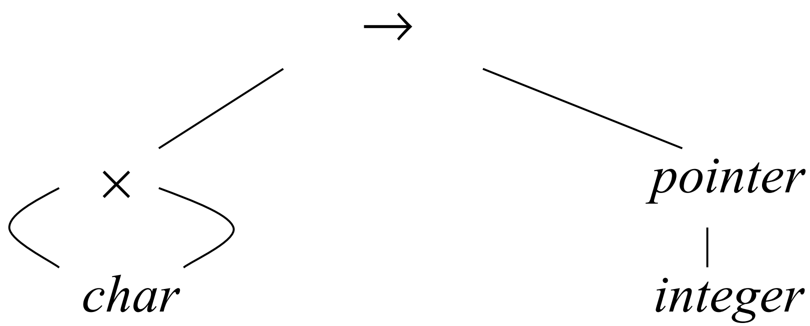
  function f(a, b: char) : ^integer;

  $char \times char \rightarrow pointer(integer)$

Representation of Type by Tree and DAG

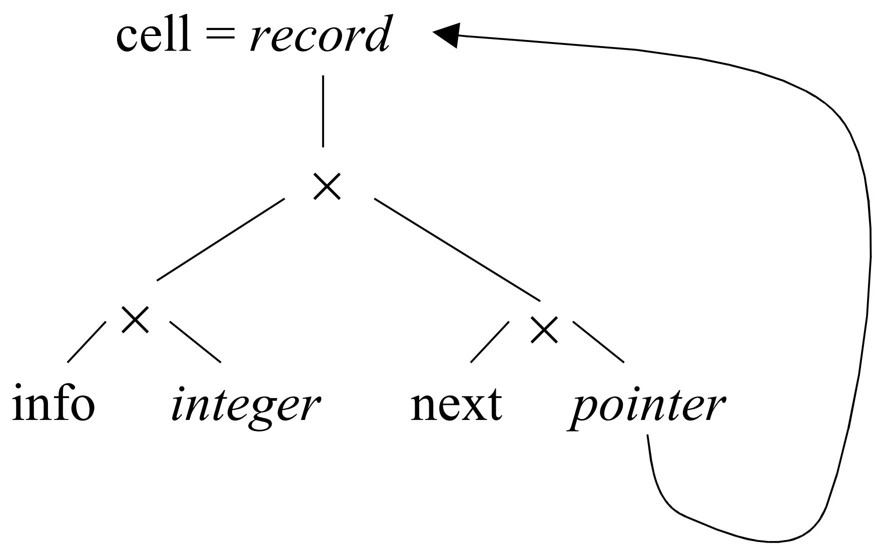
```
int *f(char, char)
```





Cycles in Representation of Types

```
struct cell {
        int info;
        struct cell * next;
};
```



Structural Equivalence and Name Equivalence

- Name equivalence: the same name of type
- Structural equivalence: If the type expression of two types have the same construction, then they are equivalent
- “Same construction”
  - Equivalent base types
  - Same set of type constructors are applied in the same order (i.e. equivalent type tree)

```
type link = ^ cell;
var  next : link;
     last : link;
     p    : ^ cell;
     q, r : ^ cell;

     cell = record
                info : integer;
                next : link
            end;
```

Structural Equivalence

- Definition: by Induction
  - Same basic type (basis)
  - Same constructor applied to SE Type (induction step)
  - Same DAG Representation
- In Practice: modifications are needed
  - Do not include array bounds – when they are passed as parameters
  - Other representations applied (more compact)
- Can be applied to: Tree/ DAG
  - Does not check for cycles
  - Later improve it.

Algorithm Testing Structural Equivalence

```
function sequiv(s, t): boolean
{
   if (s and t are of the same basic type) return true;

   if (s = array(s1, s2) and t = array(t1, t2))
      return (s1 = t1) and sequiv(s2, t2);

   if ( s = s1 × s2 and t = t1 × t2 )
      return sequiv(s1, t1) and sequiv(s2, t2);

   if (s = s1 → s2 and t = t1 → t2)
      return sequiv(s1, t1) and sequiv(s2, t2);

   if (s = pointer(s1) and t = pointer(t1))
      return sequiv(s1, t1);

   return false;
}
```

Dealing with Recursive Types in C

- C Policy: avoid cycles in type graphs by:
  - Using structural equivalence for all types
  - Except for records -> name equivalence
- Example:
  - `struct cell {int info; struct cell * next;}`
- Name use: name cell becomes part of the type of the record.
  - Use the acyclic representation
  - Names declared before use – except for pointers to records.
  - Cycles – potential due to pointers in records
  - Testing for structural equivalence stops when a record constructor is reached ~ same named record type?

Simple Type System

$P \rightarrow$ D ; E
$D \rightarrow$ D ; D | **id** : T
$T \rightarrow$ **char** | **integer** | **array [num] of** T | ^T
$E \rightarrow$ **literal** | **num** | **id** | E **mod** E | E [E] | E^

$P \rightarrow$	D ; E	
$D \rightarrow$	D ; D	
$D \rightarrow$	**id** : T	{ *addtype*(**id**.*entry*, *T*.*type*) }
$T \rightarrow$	**char**	{ *T*.*type* := *char* }
$T \rightarrow$	**integer**	{ *T*.*type* := *integer* }
$T \rightarrow$	^ T_1	{ *T*.*type* := *pointer*(T_1.*type*) }
$T \rightarrow$	**array [num] of** T_1	{ *T*.*type* := *array*(1..**num**.*val*, T_1.*type*) }

Type Checking

$E \rightarrow$ **literal**	$\{ E.type := char \}$
$E \rightarrow$ **num**	$\{ E.type := integer \}$
$E \rightarrow$ **id**	$\{ E.type := lookup($**id**$.entry) \}$
$E \rightarrow E_1$ **mod** E_2	$\{ E.type :=$ **if** $E_1.type = integer$ **and** $E_2.type = integer$ **then** $integer$ **else** $type\_error \}$
$E \rightarrow E_1$ **[** E_2 **]**	$\{ E.type :=$ **if** $E_2.type = integer$ **and** $E_1.type = array(s,t)$ **then** t **else** $type\_error \}$
$E \rightarrow E_1\hat{}$	$\{ E.type :=$ **if** $E_1.type = pointer(t)$ **then** t **else** $type\_error \}$

Type Checking for Instructions

$P \rightarrow D \,;\, S$

$S \rightarrow$ **id** $:= E$ { $S.type :=$ **if id**$.type = E.t$ **then** *void* **else** *type_error* }

$S \rightarrow$ **if** E **then** S_1 {$S.type :=$ **if** $E.type = boolean$ **then** $S_1.type$ **else** *type_error* }

$S \rightarrow$ ***while*** E **do** S_1 { $S.type :=$ **if** $E.type = boolean$ **then** $S_1.type$ **else** *type_error* }

$S \rightarrow S_1 \,;\, S_2$ { $S.type :=$ **if** $S_1.type = void$ **and** $S_2.type = void$ **then** *void* **else** *type_error* }

Type Checking for Functions

$E \rightarrow E\,(\,E\,)$

$T \rightarrow T_1$ '$\rightarrow$ ' T_2 { $T.type := T_1.type \rightarrow T_2.type$ }

$E \rightarrow E_1\,(E_2)$ { $E.type :=$ **if** $E_2.type = s$ **and** $E_1.type = s \rightarrow t$ **then** t **else** *type_error* }

Type Conversions

```
x + i
```

Postfix notation:

```
x i inttoreal real+
```

$E \rightarrow$	**num**	$\{ E.type := integer \}$
$E \rightarrow$	**num.num**	$\{ E.type := real \}$
$E \rightarrow$	**id**	$\{ E.type := lookup($**id**$.entry) \}$
$E \rightarrow$	E_1 **op** E_2	$\{ E.type :=$

if $E_1.type = integer$ **and** $E_2.type = integer$ **then** $integer$ **else**
if $E_1.type = integer$ **and** $E_2.type = real$ **then** $real$ **else**
if $E_1.type = real$ **and** $E_2.type = integer$ **then** $real$ **else**
if $E_1.type = real$ **and** $E_2.type = real$ **then** $real$
else $type\_error \}$