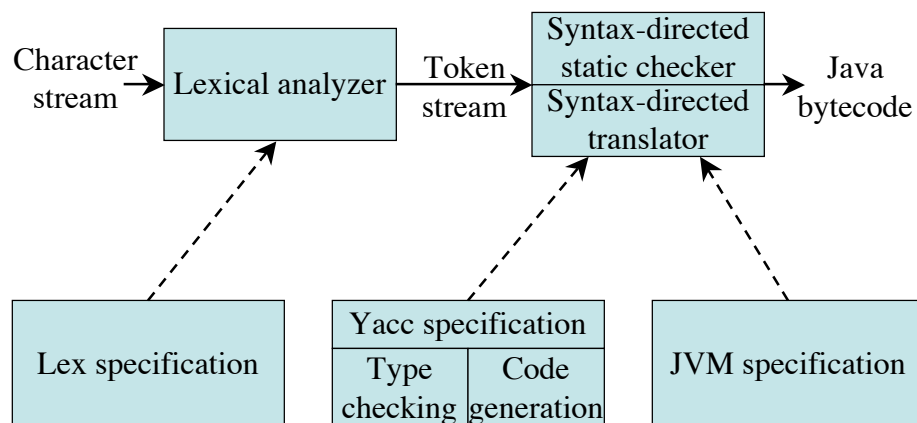


Static Checking and Type Systems

Chapter 6

COP5621 Compiler Construction
Copyright Robert van Engelen, Florida State University, 2005

The Structure of our Compiler Revisited



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

- Typical examples of static checking are
 - Type checks
 - Flow-of-control checks
 - Uniqueness checks
 - Name-related checks

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: coercion

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;
  ...
}
```

One-Pass versus Multi-Pass Static Checking

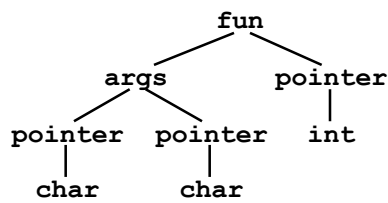
- *One-pass compiler*: static checking for C, Pascal, Fortran, and many other languages can be performed in one pass while at the same time intermediate code is generated
- *Multi-pass compiler*: static checking for Ada, Java, and C# is performed in a separate phase, sometimes requiring traversing the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

Type Expressions

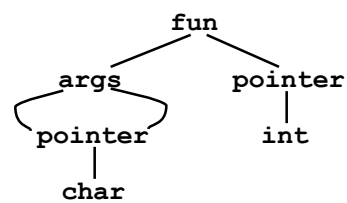
- *Type expressions* are used in declarations and type casts to define or refer to a type
 - *Primitive types*, such as **int** and **bool**
 - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
 - *Type names*, such as typedefs in C and named types in Pascal,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

Graph Representations for Type Expressions

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



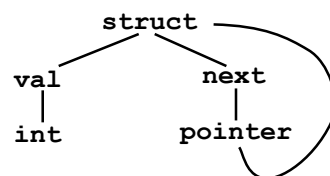
Tree forms



DAGs

Cyclic Graph Representations

```
struct Node
{ int val;
  struct Node *next;
};
```



Cyclic graph

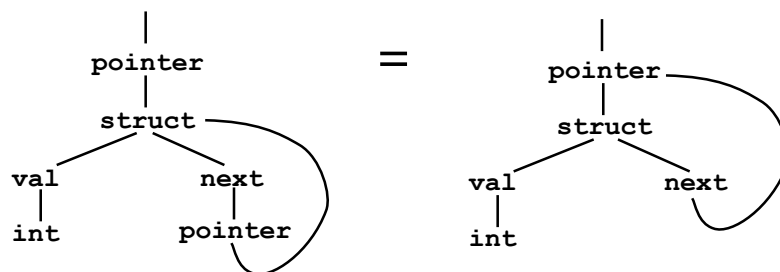
Name Equivalence

- Each type name is a distinct type, even when the type expressions the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

type link = ^node;	With name equivalence in Pascal:
var next : link;	p ≠ next
last : link;	p ≠ last
p : ^node;	p = q = r
q, r : ^node;	next = last

Structural Equivalence of Type Expressions

- Two types are the same if they are structurally identical
- Used in C, Java, C#

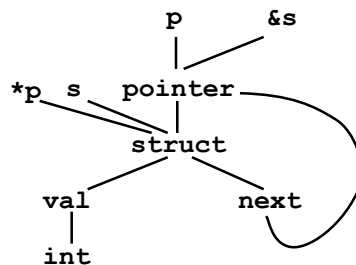


Structural Equivalence of Type Expressions (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{ int val;
  struct Node *next;
};
struct Node s, *p;
```

```
... p = &s; // OK
... *p = s; // OK
```



Constructing Type Graphs in Yacc

Type *mkint()	construct int node if not already constructed
Type *mkarr(Type*,int)	construct array-of-type node if not already constructed
Type *mkptr(Type*)	construct pointer-of-type node if not already constructed

Syntax-Directed Definitions for Constructing Type Graphs in Yacc

```
%union
{ Symbol *sym;
  int num;
  Type *typ;
}
%token INT
%token <sym> ID
%token <int> NUM
%type <typ> type
%%
decl : type ID          { addtype($2, $1); }
    | type ID '[' NUM ']' { addtype($2, mkarr($1, $4)); }
    ;
type : INT              { $$ = mkint(); }
    | type '*'          { $$ = mkptr($1); }
    | /* empty */       { $$ = mkint(); }
    ;
```

Type Systems

- A *type system* defines a set of types and rules to assign types to programming language constructs
- Informal type system rules, for example “*if both operands of addition are of type integer, then the result is of type integer*”
- Formal type system rules: Post system

Type Rules in Post System Notation

$$\frac{E(v) = T}{E \vdash v : T}$$

$$\frac{E(v) = T \quad E \vdash e : T}{E \vdash v := e}$$

Environment E maps
variables v to types T :
 $E(v) = T$

$$\frac{E \vdash e_1 : \text{integer} \quad E \vdash e_2 : \text{integer}}{E \vdash e_1 + e_2 : \text{integer}}$$

A Simple Language Example

$P \rightarrow D ; S$	$E \rightarrow \text{true}$
$D \rightarrow D ; D$	false
id : T	literal
$T \rightarrow \text{boolean}$	num
char	id
integer	$E \text{ and } E$
array [num] of T	$E \text{ mod } E$
$\wedge T$	$E [E]$
$S \rightarrow \text{id} := E$	$E \wedge$
if E then S	
while E do S	
$S ; S$	

Simple Language Example: Declarations

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

Simple Language Example: Statements

$S \rightarrow \text{id} := E$	$\{ S.\text{type} := \text{if } \text{id.type} = E.\text{type} \text{ then } \text{void} \\ \text{else } \text{type_error} \}$
$S \rightarrow \text{if } E \text{ then } S_1$	$\{ S.\text{type} := \text{if } E.\text{type} = \text{boolean} \text{ then } S_1.\text{type} \\ \text{else } \text{type_error} \}$
$S \rightarrow \text{while } E \text{ do } S_1$	$\{ S.\text{type} := \text{if } E.\text{type} = \text{boolean} \text{ then } S_1.\text{type} \\ \text{else } \text{type_error} \}$
$S \rightarrow S_1 ; S_2$	$\{ S.\text{type} := \text{if } S_1.\text{type} = \text{void} \text{ and } S_2.\text{type} = \text{void} \\ \text{then } \text{void} \text{ else } \text{type_error} \}$

Simple Language Example: Expressions

$E \rightarrow \mathbf{true}$	$\{ E.type = \mathit{boolean} \}$
$E \rightarrow \mathbf{false}$	$\{ E.type = \mathit{boolean} \}$
$E \rightarrow \mathbf{literal}$	$\{ E.type = \mathit{char} \}$
$E \rightarrow \mathbf{num}$	$\{ E.type = \mathit{integer} \}$
$E \rightarrow \mathbf{id}$	$\{ E.type = \mathit{lookup}(\mathbf{id}.entry) \}$
...	

Simple Language Example: Expressions (cont'd)

$E \rightarrow E_1 \mathbf{and} E_2$	$\{ E.type := \mathbf{if} E_1.type = \mathit{boolean} \mathbf{and}$ $E_2.type = \mathit{boolean}$ $\mathbf{then} \mathit{boolean} \mathbf{else} \mathit{type_error} \}$
$E \rightarrow E_1 \mathbf{mod} E_2$	$\{ E.type := \mathbf{if} E_1.type = \mathit{integer} \mathbf{and}$ $E_2.type = \mathit{integer}$ $\mathbf{then} \mathit{integer} \mathbf{else} \mathit{type_error} \}$
$E \rightarrow E_1 [E_2]$	$\{ E.type := \mathbf{if} E_1.type = \mathit{array}(s, t) \mathbf{and}$ $E_2.type = \mathit{integer}$ $\mathbf{then} t \mathbf{else} \mathit{type_error} \}$
$E \rightarrow E_1 \mathbf{\wedge}$	$\{ E.type := \mathbf{if} E_1.type = \mathit{pointer}(t)$ $\mathbf{then} t \mathbf{else} \mathit{type_error} \}$

Simple Language Example: Adding Functions

$$T \rightarrow T_1 \rightarrow T_2 \quad \{ T.type := function(T_1.type, T_2.type) \}$$

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

Example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
  v := 1;
```

Syntax-Directed Definitions for Type Checking in Yacc

```
%{
enum Types {Tint, Tfloat, Tpointer, Tarray, ... };
typedef struct Type
{ enum Types type;
  struct Type *child;
} Type;
%}
%union
{ Type *typ;
}
%type <typ> expr
%%
expr : expr '+' expr { if ($1.type != Tint
                        || $3.type != Tint)
                        semerror("non-int operands in +");
                        $$ = mkint();
                        emit(iadd);
                      }
```

Type Conversion and Coercion

- *Type conversion* is explicit, for example using type casts
- *Type coercion* is implicitly performed by the compiler
- Both require a type system to check and infer types for (sub)expressions

Syntax-Directed Definitions for Type Coercion in Yacc

```
%{ ... %}
%%
expr : expr '+' expr
    { if ($1.type == Tint && $3.type == Tint)
      { $$ = mkint(); emit(iadd);
      }
      else if ($1.type == Tfloat && $3.type == Tfloat)
      { $$ = mkfloat(); emit(fadd);
      }
      else if ($1.type == Tfloat && $3.type == Tint)
      { $$ = mkfloat(); emit(i2f); emit(fadd);
      }
      else if ($1.type == Tint && $3.type == Tfloat)
      { $$ = mkfloat(); emit(swap); emit(i2f); emit(fadd);
      }
      else semerror("type error in +");
      $$ = mkint();
    }
```

Syntax-Directed Definitions for L-Values and R-Values in Yacc

```

%{
typedef struct Node
{ Type *typ;
  int islval;
} Node;
%}
%union
{ Node *rec;
}
%type <rec> expr
%%

expr : expr '+' expr
    { if ($1.typ->type != Tint
      || $3.typ->type != Tint)
      semerror("non-int operands in +");
      $$ .typ = mkint();
      $$ .islval = FALSE;
      emit(...);
    }
| expr '=' expr
    { if (!$1.islval || $1.typ != $3.typ)
      semerror("invalid assignment");
      $$ .typ = $1.typ; $$ .islval = FALSE;
      emit(...);
    }
| ID
    { $$ .typ = lookup($1);
      $$ .islval = TRUE;
      emit(...);
    }
}

```