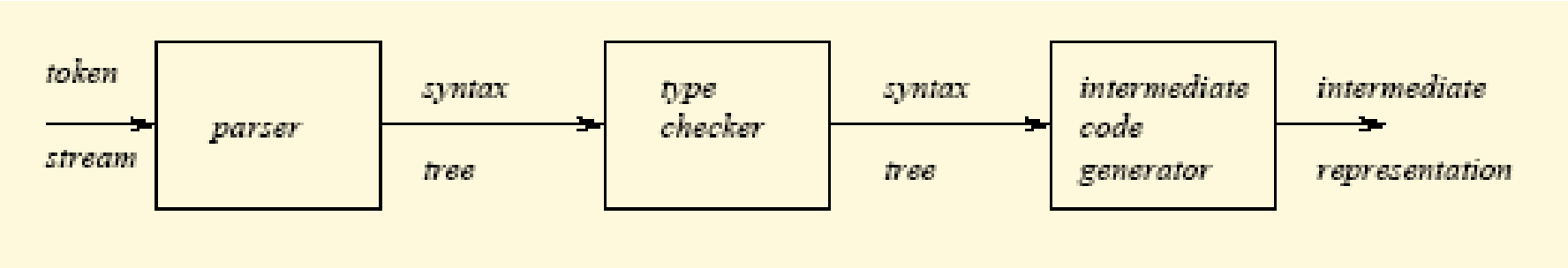
Type Checking

Reading: Section 6.1 – 6.4 of chapter 6

A compiler must check the source program to its semantic conventions in addition to syntactic.

Type Checking



Position of Type Checker

The design of a type checker for a language is based on information about the syntactic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

Type Expression

The type of a language construct is denoted by a *type expression*.

A *type expression* can be:

- **A basic type**
  - a primitive data type such as *integer, real, char, boolean*, …
  - *type-error* signal an error during type checking
  - *void* : no type
- **A type name**
  - a name can be used to denote a type expression.
- **A type constructor applies to other type expressions.**
  - **arrays**: If T is a type expression, then *array(I,T)* is a type expression where I denotes index range. Ex: array(0..99,int)
  - **products**: If T_1 and T_2 are type expressions, then their cartesian product $T_1 \times T_2$ is a type expression. Ex: int x int
  - **pointers**: If T is a type expression, then *pointer(T)* is a type expression. Ex: pointer(int)
  - **functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

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*"
- A type checker implements type system
- A sound type system eliminates run-time type checking for type errors.

Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language's *static semantics*
  - Program properties that can be checked at compile time
  - Typical examples of static checking are
    - Type checks
    - Flow-of-control checks
    - Uniqueness checks
    - Name-related checks
- *Dynamic semantics*: checked at run time
  - Compiler generates verification code to enforce programming language's dynamic semantics

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

A Simple Type Checking System

Example Simple Language

$P \rightarrow D$ **;** S

$D \rightarrow D$ **;** $D \mid$ **id :** T

$T \rightarrow$ **boolean** $\mid$ **char** $\mid$ **integer** $\mid$ **array [num] of** $T \mid \uparrow T$

$S \rightarrow$ **id =** $E \mid$ **if** E **then** $S \mid$ **while** E **do** $S \mid S$ **;** S

$E \rightarrow$ **true** $\mid$ **false** $\mid$ **literal** $\mid$ **num** $\mid$ **id** $\mid E$ **and** $E \mid E$ **+** $E \mid E$ **[** E **]** $\mid E \uparrow$

Pointer to T

Pascal-like pointer dereference operator

This grammar generate the language with declarations followed by statements like

key : integer; key = 199

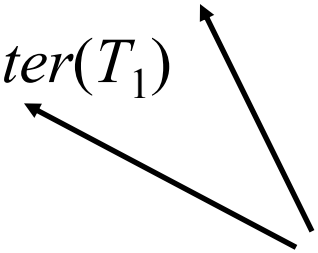
.

if true then key = 100;

A Simple Type Checking System

Example Simple Translation Scheme

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



Parametric types:
type constructor

A Simple Type Checking System

Example Type Checking of Expressions

$E \rightarrow$ **id** { E.type = l*ookup(id.entry*) }

$E \rightarrow$ **charliteral** { E.type = *char* }

$E \rightarrow$ **intliteral** { E.type = *int* }

$E \rightarrow E_1 + E_2$

{ E.type = (E_1.type == E_2.type) ? E_1.type : *type_error* }

$E \rightarrow E_1 [E_2]$

{ E.type = (E_2.type == *int* and E_1.type == *array(s,t)*) ? t : *type_error*}

$E \rightarrow E_1 \uparrow$ { E.type = (E_1.type == *pointer(t)*) ? t : *type_error* }

A Simple Type Checking System

Example Type Checking of Statements

In some languages statements have a type associated with them, while some other languages don't assign types to statements. In the latter case, statements are given a type void to distinguish a type safe statement with one which has a type error.

$S \rightarrow$ **id** = E {S.type = (**id**.type == E.type) ? *void* : *type_error*}

Note: the type of ***id*** *is determined by :* ***id****.type = lookup(****id****.entry)*

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

$S \rightarrow$ **while** E **do** S_1 {S.type = (E.type == *boolean) ?* S_1.type : *type_error*}

$S \rightarrow S_1$ **;** S_2

{S.type = (S_1.type == *void* **and** S_2.type == *void) ? void : type_error*}

A Simple Type Checking System

Example Type Checking of Functions

A function to an argument can be captured by production:

$T \rightarrow T$ **->** T $\qquad$ $E \rightarrow E$ **(** E **)**

Function type declaration $\qquad$ Function call

Example:

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

$T \rightarrow T_1$ **->** T_2 { T.type = *function*(T_1.type, T_2.type) }

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

Structural Equivalence of Type Expressions

How to know that two type expressions are equal?

As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm:

sequiv (s, t) : boolean;

if (s and t are same basic types) **then return true**
else if (s = array(s_1,s_2) and t = array(t_1,t_2)) **then return** (sequiv(s_1,t_1) and sequiv(s_2,t_2))
else if (s = s_1 x s_2 and t = t_1 x t_2) **then return** (sequiv(s_1,t_1) and sequiv(s_2,t_2))
else if (s = pointer(s_1) and t = pointer(t_1)) **then return** (sequiv(s_1,t_1))
else if (s = $s_1 \rightarrow s_2$ and t = $t_1 \rightarrow t_2$) **then return** (sequiv(s_1,t_1) and sequiv(s_2,t_2))
else return false

Names for Type Expressions

In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

```
type link = ↑ cell;
var p,q : link;
var r,s : ↑ cell
```

? p,q,r,s have same types ?

How do we treat type names?

- Get equivalent type expression for a type name (then use structural equivalence), or
- Treat a type name as a basic type.

Type Conversion and Coercion

Often if different parts of an expression are of different types then type conversion is required.

For example, in the expression: *z = x + y what is the type of z if x is integer and y is real ?*

Compiler have to convert one of the them to ensure that both operand of same type!

In many language *Type conversion* is explicit, for example using type casts i.e. must be specify as *inttoreal(x)*

Type conversions which happen implicitly is called *coercion*. Implicit type conversions are carried out by the compiler recognizing a type incompatibility and running a type conversion routine (for example, something like *inttoreal(int)*) that takes a value of the original type and returns a value of the required type.

Syntax-Directed Definitions for Type Checking in Yacc

```
%{
enum Types {Tint, Tfloat, Tpointer,
Tarray, … };
typedef struct Type
{ enum Types type;
  struct Type *child; // at most one type
parameter
} 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);
                }
```

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();
    }
```

Exercise

Q.1. Questions 6.1, 6.2, 6.3, 6.4, 6.5, 6.10 and 6.13 of text book