Type Checking

Cosmin E. Oancea
cosmin@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2013 Compiler Lecture Notes

Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree

↓

Type Checking

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

↑

Binary machine code

Interpretation Recap

Compiler	*vs.*	Interpreter
source program ↓ Compiler ↓ *target program*	*input* ↓ Target Program ↓ *output*	*source program* ↓, *input* ↓ Interpreter ↓ *output*

The interpreter directly *executes* one by one the operations specified in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.

Why interpret? Debugging, Prototype-Language Implementation, etc.

Synthesized vs Inherited Attributes

A compiler phase consists of one or several traversals of the AbSyn. We formalize it via *attributes*:

Inherited: info passed downwards on the AbSyn traversal, i.e., from root to leaves. Think: helper structs. Example?

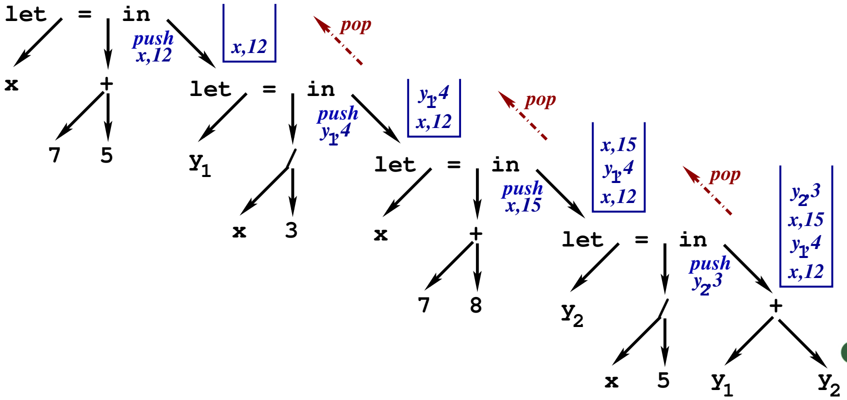
Synthesized: info passed upwards in the AbSyn traversal, i.e., from leaves to the root. Think: the result. Example?

Both: Information may be synthesized from one subtree and may be inherited/used in another subtree (or at a latter parse of the same subtree). Example?

Example of Inherited Attributes

The variable and function symbol tables, i.e., *vtable* and *ftable*, in the interpretation of an expression:

$Eval_{Exp}(Exp, vtable, ftable) = \ldots$



Example of Synthesized Attributes

The interpreted value of an expression / program is synthesized.

Example of both *synthesized* and *inherited* attributes:

$$vtable = Bind_{TypeIds}(TypeIds, args)$$
$$ftable = Build_{ftable}(Funs)$$

and used in the interpretation of a program.

Interpretation vs Compilation Pros and Cons

+ Simple (good for impatient people).

+ Allows easy modification / inspection of the program at run time.

– Typically, it does not discover all type errors. Example?

– Inefficient execution:

  - Inspects the AbSyn repeatedly, e.g., symbol table lookup.
  - Values must record their types.
  - The same types are checked over and over again.
  - No "global" optimizations are performed.

Idea: Type check and optimize as much as you can statically, i.e., before running the program, and generate optimized code.

Type System / Type Checking

Type System: a set of logical rules that a legal program must respect.

Type Checking verifies that the type system's rules are respected.
Example of type rules and type errors:

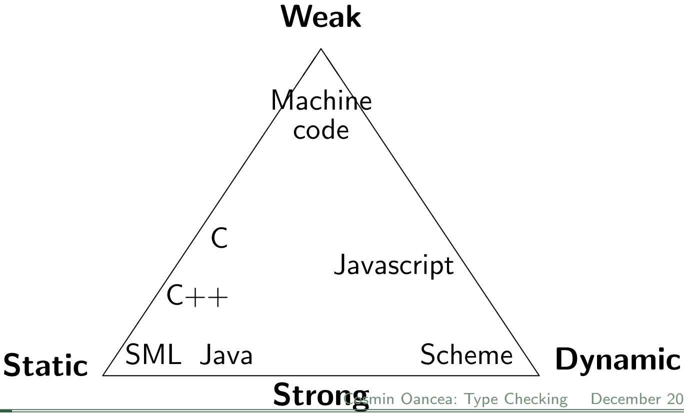
- $+, -$ expect integral arguments: `a + (b=c)`
- `if`-*branch expressions have the same type*:
  `let a = ( if (b = 3) then 'b' else 11 ) in ...`
- *the type and number of formal and actual arguments match:*:
  ```
  fun sum (x :  int list) :  int = foldl (op +) 0 x
  fun main() :  bool list = map sum [0, 1, 2, 3]
  ```
- other rules?

Some language invariants cannot be checked statically: Examples?

Type System

Static: Type checking is performed before running the program.
Dynamic: Type checking is performed while running the program.

Strong: All type errors are caught.
Weak: Operations may be performed on values of wrong types.



What Is The Plan

The type checker builds (statically) unique types for each expression, and reports whenever a type rule is violated.

As before, we logically split the AbSyn representation into different *syntactic categories*: expressions, function decl, etc.,

and implement each syntactic category via one or several functions that use case analysis on the AbSyn-type constructors.

In practice we work on AbSyn, but here we keep implementation generic by using a notation that resembles the language grammar.

For symbols representing variable names, we use *name*(**id**) to get the name as a string. A type error is signaled via function **error()**.

Symbol Tables Used by the Type Checker

vtable binds variable names to their *types*,
e.g., `int`, `char`, `bool` or arrays, e.g., `array of int`.

ftable binds function names to their *types*. The type of a function is written $(t_1, ..., t_n) \rightarrow t_0$, where t_1, ..., t_n are the argument types and t_0 is the result type.

Type Checking an Expression (Part 1)

Inherited attributes: *vtable* and *ftable*.

Synthesized attribute: the expression's type.

$Check_{Exp}(Exp, vtable, ftable) = \texttt{case } Exp \texttt{ of}$	
num	`int`
id	$t = lookup(vtable, name(\mathbf{id}))$ $if\ (\ t = unbound\ )\ then\ \mathbf{error}();\ \texttt{int}$ $else\ \ t$
$Exp_1 + Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ (\ t_1 = \texttt{int}\ and\ t_2 = \texttt{int}\ )\ then\ \texttt{int}$ $else\ \ \mathbf{error}();\ \texttt{int}$
$Exp_1 = Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ (\ t_1 = t_2\ )\ then\ \texttt{bool}$ $else\ \ \mathbf{error}();\ \texttt{bool}$
...	

Type Checking an Expression (Part 2)

$Check_{Exp}(Exp, vtable, ftable)$ = `case` Exp `of`	
...	
`if` Exp_1 `then` Exp_2 `else` Exp_3	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ *if* (t_1 = `bool` *and* $t_2 = t_3$) *then* t_2 *else* **error**(); t_2
`let` **id** `=` Exp_1 `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, name(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$
id ($Exps$)	$t = lookup(ftable, name(\textbf{id}))$ *if* ($t = unbound$) *then* **error**(); `int` *else* $((t_1, \ldots, t_n) \rightarrow t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ *if* ($m = n$ *and* $t_1 = t'_1, \ldots, t_n = t'_n$) *then* t_0 *else* **error**(); t_0

Type Checking a Function (Declaration)

- creates a *vtable* that binds the formal args to their types,
- computes the type of the function-body expression, named t_1,
- and checks that the function's return type equals t_1.

$Check_{Fun}(Fun, ftable) = \texttt{case}\ Fun\ \texttt{of}$	
$Type\ \textbf{id}\ (\ TypeIds\ )\ =\ Exp$	$vtable = Check_{TypeIds}(TypeIds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ $if\ (\ t_1\ \neq\ Type\ )$ $then\ \ \textbf{error}();\ \texttt{int}$

$Check_{TypeIds}(TypeIds) = \texttt{case}\ TypeIds\ \texttt{of}$	
$Type\ \textbf{id}$	$bind(SymTab.empty(),\ \textbf{id}, Type)$
$Type\ \textbf{id}\ ,\ \ TypeIds$	$vtable = Check_{TypeIds}(TypeIds)$ $if\ (\ lookup(vtable,\ \textbf{id}) = unbound\ )$ $then\ \ bind(vtable,\ \textbf{id},\ \ Type)$ $else\ \ \ \textbf{error}();\ vtable$

Type Checking the Whole Program

- builds the functions' symbol table,
- type-checks all functions,
- checks that a `main` function of no args exists.

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$ $if\ (\ lookup(ftable, \texttt{main}) \neq (\ ) \to \alpha\ )$ $then$ **error**()

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
$Fun\ Funs$	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

Building the Functions' Symbol Table

$Get_{Funs}(Funs)$ = case $Funs$ of	
Fun	$(f, t) = Get_{Fun}(Fun)$ $bind(SymTab.empty(),\ f,\ t)$
$Fun\ Funs$	$ftable = Get_{Funs}(Funs)$ $(f, t) = Get_{Fun}(Fun)$ $if\ (\ lookup(ftable, f) = unbound\ )$ $then\ \ bind(ftable, f, t)$ $else$ **error**(); $ftable$

$Get_{Fun}(Fun)$ = case Fun of	
$Type$ **id** ($TypeIds$) = Exp	$[t_1, \ldots, t_n] = Get_{Types}(TypeIds)$ (**id**, $(t_1, \ldots, t_n) \rightarrow Type$)

$Get_{Types}(TypeIds)$ = case $TypeIds$ of	
$Type$ **id**	$[Type]$
$Type$ **id** , $TypeIds$	$Type :: Get_{Types}(TypeIds)$

Advanced Type Checking

Data-Structures: Represent the data-structure type in the symbol table and check operations on the values of this type.

Overloading: Check all possible types. If multiple matches, select a default typing or report errors.

Type Conversion: if an operator takes arguments of wrong types then, if possible, convert to values of the right type.

Polymorphic/Generic Types: Check whether a polymorphic function is correct for all instances of type parameters. Instantiate the type parameters of a polymorphic function, which gives a monomorphic type.

Type Inference: Refine the type of a variable/function according to how it is used. If not used consistently then report error.

Polymorphic Functions: By Checking All Instances

Assume we are in our simple functional language in which all function declarations are typed, and we extend it with list/array types and with `map`, i.e., second-order function of know semantics:

$\texttt{map} : \forall\, \alpha.\, \forall\, \beta.\, (\alpha \;\rightarrow\; \beta\,) \;*\; [\alpha] \;\rightarrow\; [\beta]$,
$\texttt{map}(f, [x_1, .., x_n]) \;\equiv\; [f(x_1), .., f(x_n)]$
where f must be a function name, and
the 2nd argument is an arbitrary expression.

Type rule for checking individually each `map` call, i.e., `map(f, exp)`:

Polymorphic Functions: By Checking All Instances

Assume we are in our simple functional language in which all function declarations are typed, and we extend it with list/array types and with `map`, i.e., second-order function of know semantics:

$\texttt{map} : \forall\,\alpha.\ \forall\,\beta.\ (\alpha\ \rightarrow\ \beta\ )\ *\ [\alpha]\ \rightarrow\ [\beta]$,
$\texttt{map}(f, [x_1, .., x_n])\ \equiv\ [f(x_1), .., f(x_n)]$
where f must be a function name, and
the 2nd argument is an arbitrary expression.

Type rule for checking individually each `map` call, i.e., `map(f, exp)`:

- compute t, the type of (arbitrary expression) `x`,
  and check that $t\ \equiv\ [t_{el}]$ for some t_{el}.
- get f's signature from `ftable`. `IF` f does not receive exactly one arg `THEN error() ELSE` $f\ :\ t_{in} \rightarrow t_{out}$, for some t_{in} and t_{out}.
- `IF` $(t_{el} \equiv t_{in})$ `THEN` `map(f, x)` $:\ [t_{out}]$,
- `ELSE error()`

Type Inference for Polymorphic Functions

Key difference: type rules check whether types can be "unified", rather than type equality.

```
if ...  then ([],     [1,2,3], [])
        else (['a','b'], [],   [])
```

When we do not know a type we use a (fresh) type variable:

then: $\forall\alpha.\forall\beta.list(\alpha) * list(int) * list(\beta)$

else: $\forall\gamma.\forall\delta.list(char) * list(\gamma) * list(\delta)$

notation: use Greeks for type vars, omit $\forall$ but use fresh names.

Types t_1 and t_2 can be unified $\Leftrightarrow \exists$ substitution $S \mid S(t_1) \;=\; S(t_2)$.

Most-General Unifier: $list(char) * list(int) * list(\beta)$

$$S = \{\alpha \leftarrow char, \gamma \leftarrow int, \delta \leftarrow \beta\}$$

Example: Inferring the Type of SML's `length`

```
fun length(x) = if null(x) then 0
                else length( tl(x) ) + 1
```

Example: Inferring the Type of SML's `length`

```
fun length(x) = if null(x) then 0
                else length( tl(x) ) + 1
```

EXPRESSION	:	TYPE	UNIFY
length	:	$\beta \rightarrow \gamma$	
x	:	β	
if	:	$bool * \alpha_i * \alpha_i \rightarrow \alpha_i$	
null	:	$list(\alpha_n) \rightarrow bool$	
null(x)	:	$bool$	$list(\alpha_n) \equiv \beta$
0	:	int	$\alpha_i \equiv int$
$+$	:	$int * int \rightarrow int$	
tl	:	$list(\alpha_t) \rightarrow list(\alpha_t)$	
$tl(x)$	:	$list(\alpha_t)$	$list(\alpha_t) \equiv list(\alpha_n)$
$length(tl(x))$	:	γ	$\gamma \equiv int$
$length(tl(x)) + 1$	:	int	
if(..) then .. else ..	:	int	

Most-General Unifier Algorithm (MGU)

- a type expression is represented by a graph (typically acyclic),
- a set of unified nodes has one representative, REP, (initially each node is its own representative),
- `find(n)` returns the representative of node `n`.
- `union(m,n)` merges the equivalence classes of `m` and `n`:
  - if `m` is a type constructor (or basic type) then REP of all nodes in `m`'s equivalence class are set to `find(m)` (and similar for `n`),
  - otherwise pick one, e.g., `n`, and set the REP of all the nodes in `m`'s equivalence class to `find(n)`

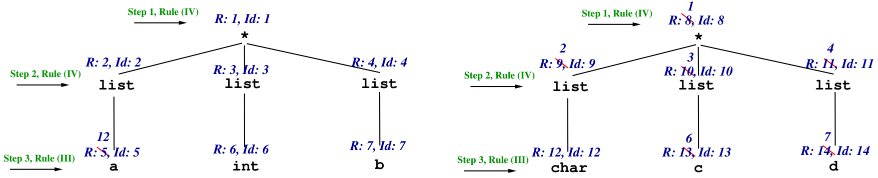
boolean unify(Node m, Node n)

(I) **if** (*find*(*m*) = *find*(*n*)) **then return** *true*;
(II) **else if** (*m and n are the same basic type*) **then return** *true*;
(III) **else if** (*m or n represent a type variable*) **then** `union`(m, n); **return** *true*;
(IV) **else if** (*m and n are the same type − constructor with children* m_1, .., m_k *and* n_1, .., n_k, $\forall\ k$) **then**
`union`(m, n); **return** `unify`(m_1, n_1) *and* .. *and* `unify`(m_k, n_k);
(V) **else** **return** *false*;

Most-General Unifier Example

Each node is annotated with two integer values: – REP (R) – node's identifier (Id)

Initially, Id = R, i.e., every node in its own equiv class.



SUCCESS (after three big horizontal steps), MGU is: list(char) * list(int) * list(b)

To construct the unified type (after MGU succeeds): start with any of the two type expressions, and write down the "representative" nodes, i.e., the ones with Node Id = REP (otherwise jump to the corresponding REP node and write it down).

Structural-Equivalence Example

Intuitively, the names of the structs and fields, e.g., `A`, `a`, do NOT matter, but only the type constructors, e.g., `struct`, `*` and basic types, e.g., `int`.

Under Structural Equivalence: types `A`, `B` and `C` Are Equivalent

```
struct A {              struct B {                        struct C {
    int       a;            int       b;                      int       d;
    struct B* b;            struct A* a;                      struct C* c;
};                      };                                };
```

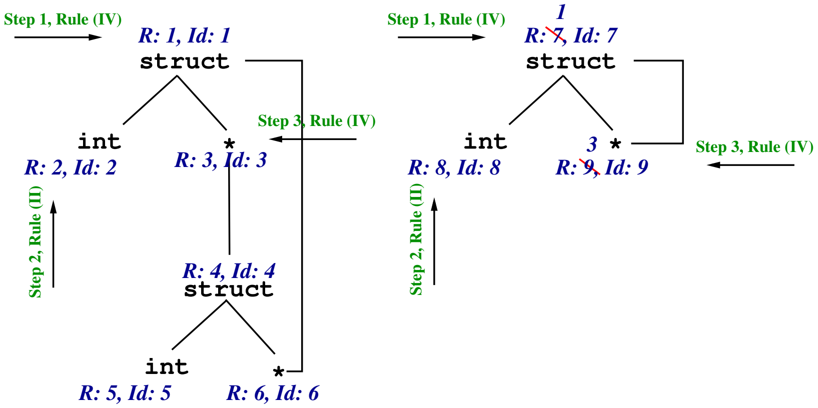
Next slides compute the most-general unifier (MGU) of `A` and `C`, which both have cyclic graph representations of their types.

To construct the unified type (after MGU succeeds): start with any of the two type expressions, and write down the "representative" nodes, i.e., the ones with Node Id = REP (otherwise go to the corresponding REP node and write it down).

For cyclic graphs, a marking phase is necessary so that you do not visit the same node multiple times, i.e., infinite recursion.

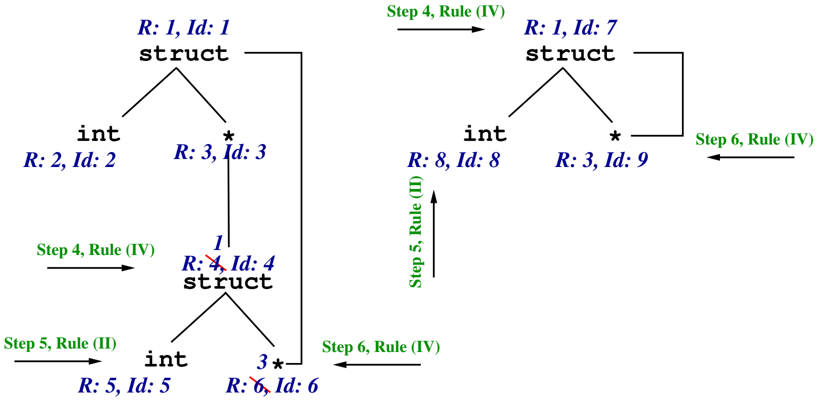
Structural Equivalence Example (2)

Each node is annotated with two integer values:
– REP
– node's identifier

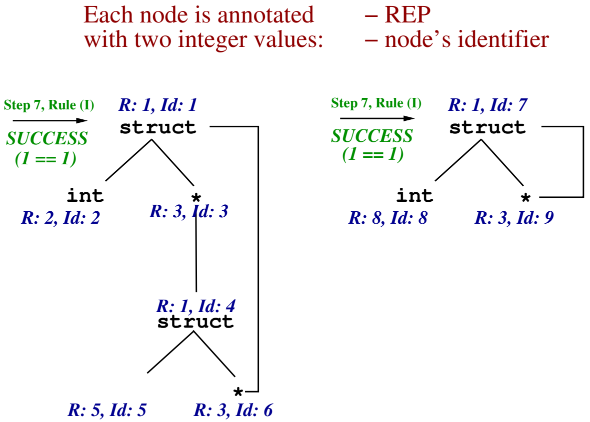


Structural Equivalence Example (3)

Each node is annotated with two integer values: – REP – node's identifier



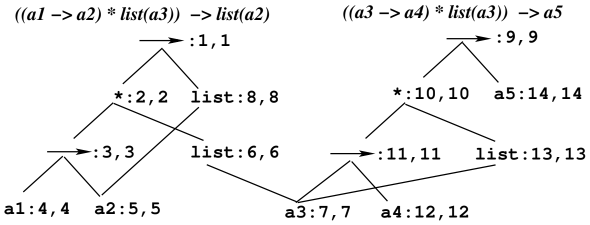
Structural Equivalence Example (3)



After MGU succeeds, to build the unified type when graph may be cyclic, a marking phase is necessary so that you do not visit the same node multiple times, i.e., infinite recursion.

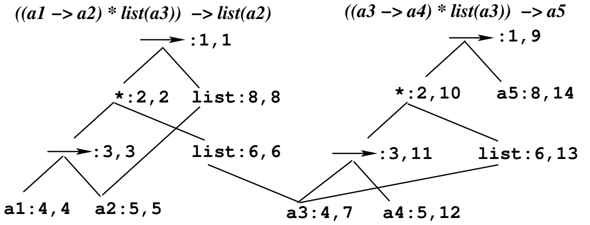
The unified type would be `struct C`.

Another Most-General Unifier Example



Each node is annotated with two integer values:

- REP
- node's identifier

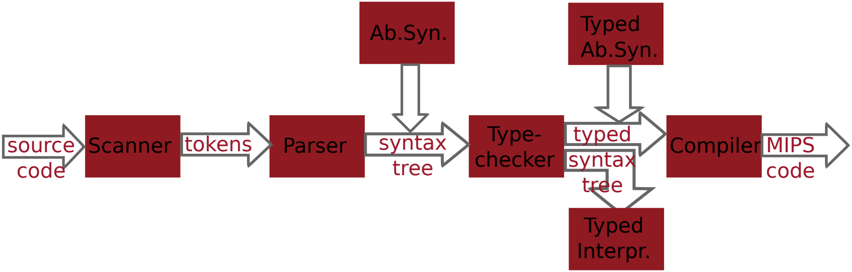


The unifier is constructed by combining nodes' REPs:

((a1 -> a2) * list(a1)) -> list(a2)

To construct the unified type: start with any of the two type expressions; and write down the "representative" nodes, i.e., the ones with node id = REP (otherwise go to the REP node & write it).

Compiler Modules of the Paladim Compiler



- The implementation of the parser is task 1 in G-Assignment (but a hand-implemented parser is provided to you).
- The parser produces an (untyped) AbSyn, i.e., file `AbSyn.sml`.
- The type checker receives an untyped AbSyn and produces a typed one, denoted TpAbSyn, in file `TpAbSyn.sml`, in which the type of any value, exp, stmt, can be querried via `typeOf*`.
- The program interpretation is performed on the TpAbSyn.

Paladim Grammar & Semantics

Prog	→	**program** ID ; *FunDecs*
Type	→	**int** | **char** | **bool**
Type	→	**array of** *Type*
FunDecs	→	*FunDecs FunDec*
FunDecs	→	*FunDec*
FunDec	→	**function** ID(*PDecl*) : *Type Block*;
FunDec	→	**procedure** ID (*PDecl*) *Block* ;
Block	→	*DBlock SBlock*
DBlock	→	**var** *Decs*
DBlock	→	ε
SBlock	→	**begin** *StmtSeq* ; **end**
SBlock	→	*Stmt*
StmtSeq	→	*StmtSeq* ; *Stmt*
StmtSeq	→	*Stmt*
PDecl	→	*Params*
PDecl	→	ε
Params	→	*Params* ; *Dec*
Params	→	*Dec*
Dec	→	ID : *Type*
Decs	→	*Decs Dec* ;
Decs	→	*Dec* ;

- Imperative (destructive updates), with mutually recursive fun/procs.
- Basic types: `int`, `char`, `bool`, and arbitrarily nested array types, e.g., array of array of int.
- Program execution starts with call to procedure `main()` (no args).
- Separate namespaces vars & funs.
- Illegal for two funs/procs to share the same name & Dito for two formal args of the same fun/proc.
- Static scoping. Functions produce a result and use call by value. Procs do not produce a result and use call by value result (Task 5).
- Funs/Procs declare a name (ID), a sequence of (typed) formal args (*PDecl*), and a body (*Block*).
- A *Block* consists of (i) an optional block of (typed) var declarations (*DBlock*), and (ii) a sequence of stmts (*SBlock*), where stmts enclosed in `begin ... end`.

Paladim's Grammar & AbSyn

Prog	→	**program** ID ; *FunDecs*
Type	→	**int** | **char** | **bool**
Type	→	**array of** *Type*
FunDecs	→	*FunDecs FunDec*
FunDecs	→	*FunDec*
FunDec	→	**function** ID(*PDecl*) : *Type Block*;
FunDec	→	**procedure** ID (*PDecl*) *Block* ;
Block	→	*DBlock SBlock*
DBlock	→	**var** *Decs*
DBlock	→	ε
SBlock	→	**begin** *StmtSeq* ; **end**
SBlock	→	*Stmt*
StmtSeq	→	*StmtSeq* ; *Stmt*
StmtSeq	→	*Stmt*
PDecl	→	*Params*
PDecl	→	ε
Params	→	*Params* ; *Dec*
Params	→	*Dec*
Dec	→	ID : *Type*
Decs	→	*Decs Dec* ;
Decs	→	*Dec* ;
LVal	→	ID
LVal	→	ID[*Exps*]

```
structure AbSyn = struct
  type Pos   = int * int (*line,col*)
  type Ident = string
  exception Error of string*(Pos)

  datatype Type =   Int  of Pos
    | Bool of Pos | Char of Pos
    | Array of Type * Pos

  datatype FunDec =
      Func of Type * Ident * Dec list *
                           StmtBlock * Pos
    | Proc of               Ident * Dec list *
                           StmtBlock * Pos
  and Dec = Dec of Ident * Type * Pos
  and StmtBlock =
        Block of Dec list * Stmt list

  and Exp = ... and Stmt = ...
  and LVAL = Var    of Ident (* e.g., x *)
           | Index  of Ident  * Exp list
               (* e.g., arr[1,2,3] *)

  type Prog = FunDec list
```

Paladim Grammar & Semantics (continuation)

Stmt	$\rightarrow$	ID(*CallParams*)
Stmt	$\rightarrow$	**if** *Exp* **then** *Block*
Stmt	$\rightarrow$	**if** *Exp* **then** *Block* **else** *Block*
Stmt	$\rightarrow$	**while** *Exp* **do** *Block*
Stmt	$\rightarrow$	**return** *Ret*
Stmt	$\rightarrow$	*LVal* := *Exp*
CallParams	$\rightarrow$	*Exps*
CallParams	$\rightarrow$	ε
Exps	$\rightarrow$	*Exp* , *Exps*
Exps	$\rightarrow$	*Exp*
Ret	$\rightarrow$	*Exp*
Ret	$\rightarrow$	ε
LVal	$\rightarrow$	ID
LVal	$\rightarrow$	ID[*Exps*]
Exp	$\rightarrow$	NUMLIT | LOGICLIT | CHARLIT
Exp	$\rightarrow$	{ *Exps* } non-empty array literals
Exp	$\rightarrow$	*LVal*
Exp	$\rightarrow$	NOT *Exp*
Exp	$\rightarrow$	*Exp* OP *Exp*
Exp	$\rightarrow$	(*Exp*)
Exp	$\rightarrow$	ID (*CallParams*)
OP	$\rightarrow$	+ | - | * | / | = | < | AND | OR

Statements:

- procedure call, *CallParams* is a comma separated sequence of expressions (actual args),
- `if-then` construct,
- `if-then-else` construct,
- `while` loop,
- return statement, empty return is allowed (for procedures),
- assignment statement updates a left-value (*LVal*), either a var or a basic-type element of an array, i.e., full indexing `a[i]:=x+y;`

Expressions:

- literals: numeric, bools, chars
- array literals, {{1+x,3},{2,4*y}}, multi-dimensional regular arrays, i.e., all rows have the same size.
- (indexed) variable, i.e. *LVal*
- negation & binary operations
- function calls.

Paladim Grammar & AbsYn (continuation)

$$
\begin{array}{lll}
\mathit{Stmt} & \rightarrow & \textsc{Id}(\mathit{CallParams}) \\
\mathit{Stmt} & \rightarrow & \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Block} \\
\mathit{Stmt} & \rightarrow & \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Block} \\
 & & \qquad\quad \textbf{else}\ \ \mathit{Block} \\
\mathit{Stmt} & \rightarrow & \textbf{while}\ \mathit{Exp}\ \textbf{do}\ \mathit{Block} \\
\mathit{Stmt} & \rightarrow & \textbf{return}\ \mathit{Ret} \\
\mathit{Stmt} & \rightarrow & \mathit{LVal}\ :=\ \mathit{Exp} \\
\mathit{CallParams} & \rightarrow & \mathit{Exps} \\
\mathit{CallParams} & \rightarrow & \varepsilon \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ ,\ \mathit{Exps} \\
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Ret} & \rightarrow & \mathit{Exp} \\
\mathit{Ret} & \rightarrow & \varepsilon \\
\mathit{LVal} & \rightarrow & \textsc{Id} \\
\mathit{LVal} & \rightarrow & \textsc{Id}[\mathit{Exps}] \\
\mathit{Exp} & \rightarrow & \textsc{NumLit} \mid \textsc{CharLit} \\
\mathit{Exp} & \rightarrow & \{\ \mathit{Exps}\ \} \\
\mathit{Exp} & \rightarrow & \mathit{LVal} \\
\mathit{Exp} & \rightarrow & \textsc{not}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \mathrm{OP}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & (\ \mathit{Exp}\ ) \\
\mathit{Exp} & \rightarrow & \mathrm{ID}\ (\ \mathit{CallParams}\ ) \\
\mathit{OP} & \rightarrow & + \mid \text{-} \mid * \mid / \mid = \mid < \mid \textsc{and} \mid \textsc{or}
\end{array}
$$

```
structure AbSyn = struct ...
  and Stmt = ProcCall of Ident*Exp list*Pos
    |        Return of Exp option      * Pos
    |        Assign of LVAL * Exp      * Pos
    |        IfThEl of Exp * StmtBlock*
                          StmtBlock * Pos
    |        While  of Exp * StmtBlock* Pos
 (*while a<b do begin x[i]:= a;a:=a+1 end*)

  and LVAL = Var     of Ident (* e.g., x *)
           | Index   of Ident  * Exp list
                        (* e.g., arr[1,2,3] *)
  and Exp
    = Literal of Value                * Pos
    | ArrLit  of Exp list             * Pos
    | LValue  of LVAL                 * Pos
    | Not     of Exp                  * Pos
    | Plus    of Exp * Exp            * Pos
    | Equal   of Exp * Exp            * Pos
    | FunApp  of Ident  * Exp list * Pos
```

Differences: AbSyn vs. TpAbSyn

AbSyn ⇒ Type Checker ⇒ TpAbSyn

AbSyn design targets ease of parsing, TpAbSyn is better suitted for code transformations, e.g., machine code generation.

AbSyn *vs.* **TpAbSyn**

```
structure AbSyn = struct
  datatype Type = Int of Pos
    | Bool of Pos | Char  of Pos
    | Array of Type * Pos
      (*recursive, easy to parse*)
  and Value = BVal  of BasicVal
    | Arr of BasicVal array *
              int list * int list

  type Ident = string
  datatype Dec = Dec of Ident
                    * Type * Pos
  and LVAL = Var  of Ident
    | Index of Ident * Exp list

  and Exp = ArrLit of Exp list*Pos
    |       LValue  of LVAL * Pos
```

```
structure TpAbSyn = struct
  datatype BasicType = Int | Bool | Char
  and Type = BType of BasicType
           | Array of int * BasicType
          (* array's rank * basic type *)
  and Value = BVal  of BasicVal
    | Arr of BasicVal array *
              int list * int list * Type

  type Ident = string * Type
  datatype Dec = Dec of Ident * Pos
              (* Ident contains the Type *)
  and LVAL = Var of Ident     (* x *)
    | Index of Ident * Exp list (* x[a,2]*)
              (* Ident contains the Type *)
  and Exp = ArrLit of Exp list * Type*Pos
    |       LValue  of LVAL * Pos
```

Differences: AbSyn vs. TpAbSyn (cont.)

On the previous slide

- ABSYN's `Type` datatype is recursive (ease of parsing)
- TPABSYN's `Type` is easier to work with (explicit array rank).
- TPABSYN contains all type info and exports function `typeOf*` that querries the type of an expression, value, stmt, etc.

ABSYN *vs.* **TPABSYN**

```
structure AbSyn = struct
  type Ident = string

  and Exp = ...
    | FunApp of Ident*Exp list*Pos
        (* e.g., f(1, 3+x) *) ...

  and Stmt = ProcCall of Ident *
                        Exp list * Pos
```

```
structure TpAbSyn = struct
  type Signature = Type list * Type option
  type FIdent    = string * Signature
  and Exp = ...
    | FunApp of FIdent * Exp list * Pos
        (* e.g., f(1, 3+x) *) ...

  and Stmt = ProcCall of FIdent *
                         Exp list * Pos
```

For example, function/procedure identifiers (appearing in calls), contain the function signature, albeit the return type would suffice.

The Gist of Type.sml: Special/Predefined Functions

Monomorphic: `ord` : $Char \rightarrow Int$, `chr` : $Char \rightarrow Int$.

Notations: $\alpha \in \{Int, Char, Bool\}$
$(Array\ n\ \alpha)$ is the type of a n-dim array of basic type α.

Polymorphic, i.e., types not expressible in Paladim:

- `len` : $(Int * (Array\ n\ \alpha)) \rightarrow Int, \forall n \in \{1 \ldots\}$.
  `len(i,a)` returns the length of dim `i` of `a`.
  If `i`$\notin \{0, \ldots, n-1\} \Rightarrow$ runtime error.

- `new` : $(Int^1 * \ldots * Int^n) \rightarrow (Array\ n\ \alpha)$,
  α is determined from the context. Creates an n-dim array of specified dimension sizes. If any negative size $\Rightarrow$ runtime error.

- `read` : $() \rightarrow \alpha$, α is determined from the context.

- `write` : $\alpha \rightarrow ()$

The Gist of Type.sml: Special/Predefined Functions

Special functions are added to function's symbol table

```
  open TpAbSyn

  type VTab = (string * Type) list

(*type Signature = Type list * Type option, i.e., t1,...,tn ->  t*)
  type FTab = (string * Signature) list

  val functionTable : FTab ref
    = ref [ ("ord",   ([BType Char], SOME (BType Int ))) (* ord: char->int   *)
          , ("chr",   ([BType Int ], SOME (BType Char))) (* chr: int ->char *)
          , ("len",   ([BType Int ], SOME (BType Int ))) (* polymorphic *)
          , ("new",   ([BType Int ], SOME (BType Int ))) (* polymorphic *)
          , ("write",([BType Int ], NONE))               (* polymorphic *)
          , ("read", ([], SOME (BType Int)))             (* polymorphic *)
       ...]
```

VTab associates a variable name with its type.

FTab associates a function name with its signature.

The polymorphic functions are given phony signatures!

The Gist of Type.sml: Type Checking a Program

Type Checker Entry Point is Function typeCheckPgm

```
open TpAbSyn  (* allows to write  Type instead of  TpAbSyn.Type *)
fun typeCheckPgm( old_fun_decs : AbSyn.FunDec list ) : TpAbSyn.FunDec list =
  let fun getType (f : AbSyn.FunDec) : (string * (Type list * Type option))=
          let val ( fnm, rtp ) = ( AbSyn.getFunName f, AbSyn.getFunRetp f )
              val new_tps= map (fn AbSyn.Dec(id,tp,_) => toTpAbSynType tp)
                               (AbSyn.getFunArgs f)
          in  ( fnm, (new_tps, toTpAbSynOptType rtp) ) end
      val fun_table = map getType old_fun_decs
      val () = functionTable := List.foldr
                                   (fn ((n,ts),tab) => SymTab.insert n ts tab)
                                   (!functionTable) fun_table
      val decorated = map typeCheckFun old_fun_decs
  in  (* check main function presence and type () -> () *)
      case SymTab.lookup "main" (!functionTable) of
        NONE             => raise Error ("No main function defined!", ...)
      | SOME ([], NONE) => decorated (* OK, type correct program *)
      | SOME (args,res) => raise Error ("Illegal Arg to main", ...)   end
```

Function signatures are translated to TpAbSyn types and added to `ftable`. SymTab.insert raises an error if a name duplicate is binded.

Functions are type checked and translated to TpAbSyn!

The Gist of Type.sml: Type Checking a Function

typeCheckFun (f : AbSyn.FunDec) : TpAbSyn.FunDec

```
type VTab = (string * Type) list
fun typeCheckFun ( AbSyn.Func (old_rtp, fnm, old_decs, old_body, pos) ) =
    let val new_rtp = toTpAbSynType old_rtp
        val ()      = current_rettp := SOME new_rtp
        val new_decs= map ( fn (AbSyn.Dec (id,tp,p)) =>
                                  Dec((id, toTpAbSynType tp),p) ) old_decs
        val ()      = checkDuplicateDecs new_decs
        val vtab    = map ( fn (Dec (idtp, _)) => idtp ) new_decs
        val ()      = current_rettp := (SOME new_rtp)
        val new_body= typeCheckBlock(vtab, old_body)
    in  Func ( new_rtp, fnm, new_decs, new_body, pos )
    end
```

Return type & formal-argument declarations translated to TpAbSyn.

If formal parameters have duplicate names, an error is raised.

`vtable` is initialized with the formal-argument bindings and global ref `current_rettp` records the return type (used in RETURN stmt)!.

The function's body is type checked and translated to TpAbSyn!

The Gist of Type.sml: Type Checking a Block

typeCheckBlock(v:VTab, b:AbSyn.StmtBlock) : TpAbSyn.StmtBlock

```
type VTab = (string * Type) list
and typeCheckBlock ( vtab : VTab, AbSyn.Block( decs, stmts ) ) =
      let val new_decs = map ( fn (AbSyn.Dec(id,tp,p)) =>
                                    Dec((id,toTpAbSynType tp),p) ) decs
          val ()       = checkDuplicateDecs new_decs
          val ds = map ( fn (Dec(idtp,_)) => idtp ) new_decs
          val new_vtab = ds @ vtab
          val new_stmts= map (fn stmt=>typeCheckStmt(new_vtab,stmt)) stmts
      in  Block(new_decs, new_stmts)
      end
```

Block's declarations are translated to TpAbSyn.

If declared vars have duplicate names, an error is raised.

`vtable` is extended with the declared var bindings.

The block's statements are type checked and translated to TpAbSyn!

The Gist of Type.sml: Type Checking `return e`

typeCheckStmt (v: (string * Type), s: AbSyn.Stmt) : TpAbSyn.Stmt

```
datatype ExpectType = SomeArray of TpAbSyn.BasicType (* not used *)
                    | KnownType of TpAbSyn.Type
                    | UnknownType

fun typeCheckStmt ( vtab, AbSyn.Return( eopt, pos ) ) =
    ( case (eopt, !current_rettp) of  (* the return type of fun/proc *)
        (NONE,   NONE  ) => Return (NONE, pos)
      | (SOME e, SOME t) =>
          let val new_e = typeCheckExp(vtab, e, KnownType t)
          in  if typesEqual( t, typeOfExp new_e ) )
              then Return(SOME new_e, pos)
              else raise Error("Fun Ret Type != returned exp type", pos) end
      | (_, _) => raise Error("Illegal!", pos)  )
```

Returned expression is type checked and translated to TpAbSyn.

Its expected type is the function's return type (inherited attribute).

Any TpAbSyn expression can be queried for its type via `typeOfExp`.

Type Rule: the function's return type matches the returned-exp type.

The Gist of Type.sml: Type Checking `id := e`

typeCheckStmt (v: (string * Type), s: AbSyn.Stmt) : TpAbSyn.Stmt

```
| typeCheckStmt( vtab, AbSyn.Assign(AbSyn.Var(id), e, pos) ) =
    let val id_tp = (case SymTab.lookup id vtab of
                        SOME tp => tp
                      | NONE    => raise Error("Var not in VTab", pos) )
        val new_e = typeCheckExp(vtab, e, KnownType id_tp)
        val e_tp  = typeOfExp new_e
    in  if typesEqual(id_tp, e_tp)
        then Assign( Var(id,id_tp) , new_e, pos )
        else raise Error( "Declared var type != assigned-exp type", pos ) end
```

Assigned expression is type checked and translated to TpAbSyn.

Its expected type is the declared type of `id` variable (from `vtable`).

Type Rule: variable's declared type matched the assigned-exp type!

Type checking While-loop and If statements is straightforward:

- The condition expression (expected type `Bool`) and the then/else/body blocks of statements are type checked.
- The type rule is that the condition's type is `Bool`.

The Gist of Type.sml: Procedure Call

typeCheckStmt (v: (string * Type), s: AbSyn.Stmt) : TpAbSyn.Stmt

```
| typeCheckStmt ( vtab, AbSyn.ProcCall( fid, args, pos ) ) =
    let val proc_arg_tps = case SymTab.lookup fid (!functionTable) of
                NONE => raise Error("Procedure not in function table!", pos)
              | SOME (tps, _) => if ((length tps) = (length args)) then tps
                                 else raise Error("Number of formal/actual args differs!",pos)
            (* building the expected types for the arguments *)
            val expect_arg_tps = map (fn t => KnownType t) proc_arg_tps
            val new_args = ListPair.map
                                ( fn (e,etp) => typeCheckExp(vtab, e, etp) )
                                (args, expect_arg_tps)
            val args_tps = map ( fn e => typeOfExp e ) new_args
            val tpok = ListPair.foldl
                            (fn (t1, t2, b) => b andalso typesEqual(t1,t2))
                            true (proc_arg_tps, args_tps)
        in  if tpok then ProcCall((fid, (proc_arg_tps,NONE)), new_args,pos)
            else raise Error("Actual and formal arg types differ",pos) end
```

Procedure's argument types are taken from `ftable` and serve as expected types for the actual-argument expressions.

Actual-arg expressions are type checked and translated to TpAbSyn.

Type Rule: formal-arg types match the actual-arg number and (computed) types (queried via `typeOfExp`)!

A function call is similar (you need to explain it for W-Assignment 3).

The Gist of Type.sml: Type Checking Expressions

typeCheckExp(v:VTab, e:AbSyn.Exp, expected_tp:ExpectType): Exp

```
| typeCheckExp ( vtab, AbSyn.Equal(e1, e2, pos), _ ) =
    let val e1_new = typeCheckExp(vtab, e1, UnknownType)
        val e2_new = typeCheckExp(vtab, e2, UnknownType)  (* modify *)
        val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)

        val () = case tp1 of (* check that tp1 is not an array type *)
                     Array _ => raise Error("Array type in equal exp!", pos)
                   | _ => ()
    in  if  typesEqual(tp1, tp2) then Equal(e1_new, e2_new, pos)
        else raise Error("Argument types do not match ", pos)
    end
```

What should be the expected type for e1 and e2? (G-Assignment)

Subexpressions are type checked and translated to TpAbSyn.

Type Rule: subexpressions types must match and be a basic type!

A `Plus` expression is even easier.
(what are the subexpressions' expected types?)

The Gist of Type.sml: `len` & `read` Calls

typeCheckExp(v:VTab, e:AbSyn.Exp, expected_tp:ExpectType): Exp

```
| typeCheckExp ( vtab, AbSyn.FunApp ("len", [d,arr], pos), _ ) =
    let val new_d     = typeCheckExp( vtab, d,   KnownType (BType Int) )
        val new_arr   = typeCheckExp( vtab, arr, UnknownType           )
        val (d_tp, arr_tp) = (typeOfExp new_d, typeOfExp new_arr)
        val () = case d_tp of BType Int => ()
                    | tp         => raise Error("First arg not an int", pos)
        val () = case arr_tp of Array _ => ()
                    | tp       => raise Error("Second arg not an array", pos)
    in FunApp( ( "len", ([d_tp,arr_tp], SOME (BType Int)) ), [new_d, new_arr], pc
    end
| typeCheckExp (vtab, AbSyn.FunApp ("read",[],pos), KnownType (BType btp) ) =
    FunApp( ( "read", ([], SOME (BType btp)) ), [], pos )
```

Expected type for the first actual arg of `len` is `Int`, i.e., represents the number of the queried dimension

Type Rule: `len` *is called on exactly two arguments: 1st-arg type is* `Int`, *2nd-arg type is some* `Array`*!*

The result type of a call to `read` is the expected type (if known, otherwise fails!)