

Compilerconstructie

najaar 2014

<http://www.liacs.nl/home/rvvliet/coco/>

Rudy van Vliet

kamer 124 Snellius, tel. 071-527 5777

rvvliet(at)liacs(dot)nl

college 5, dinsdag 7 oktober 2014

Static Type Checking

The Phases of a Compiler

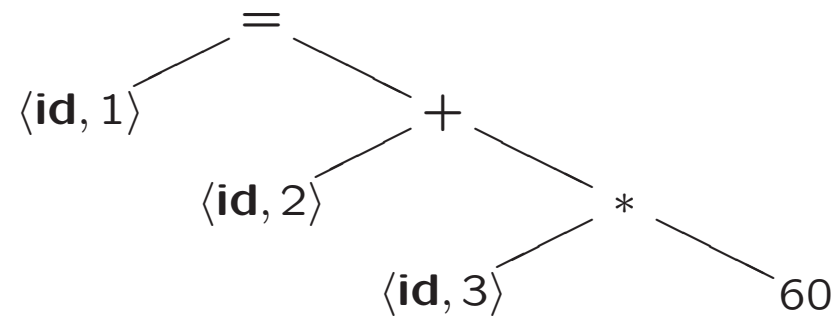
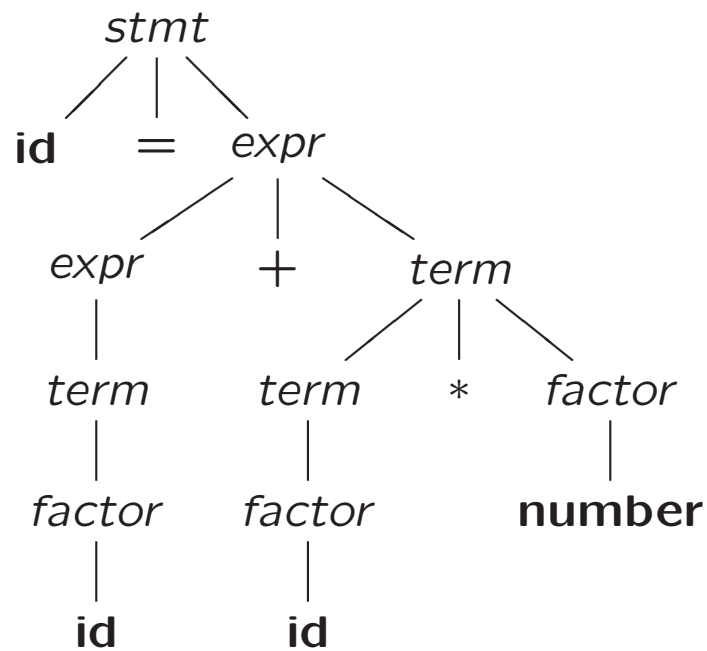
(from lecture 1)

Token stream:

$\langle \mathbf{id}, 1 \rangle \langle = \rangle \langle \mathbf{id}, 2 \rangle \langle + \rangle \langle \mathbf{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Syntax Analyser (parser)

Parse tree / syntax tree:



6.1 Variants of Syntax Trees

Directed Acyclic Graphs for Expressions

$$a + a * (b - c) + (b - c) * d$$

Syntax tree vs DAG...

Pros DAG...

Producing Syntax Trees or DAG's

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.node = getNode('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = getNode('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow T_1 * F$	$T.node = getNode('*', T_1.node, F.node)$
5) $T \rightarrow T_1 / F$	$T.node = getNode('/', T_1.node, F.node)$
6) $T \rightarrow F$	$T.node = F.node$
7) $F \rightarrow (E)$	$F.node = E.node$
8) $F \rightarrow \mathbf{id}$	$F.node = getLeaf(\mathbf{id}, \mathbf{id}.entry)$
9) $F \rightarrow \mathbf{num}$	$F.node = getLeaf(\mathbf{num}, \mathbf{num}.val)$

Parse tree $a + a * (b - c) + (b - c) * d \dots$

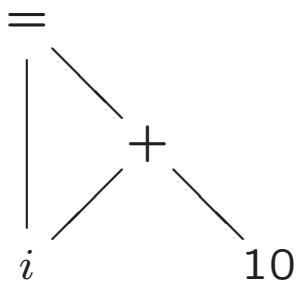
1) $p_1 = getLeaf(\mathbf{id}, entry-a)$

...

6.1.2 The Value-Number Method

An implementation of DAG

DAG for $i = i + 10$



1	id	→ to entry for i	
2	num	10	
3	+	1	2
4	=	1	3
5	...		

- Search array for (existing) node
- Use hash table

Static Checking

- **Type checks:**
Verify that type of a construct matches the expected one
- **Flow-of-control checks:**
Example: break-statement must be enclosed in while-, for- or switch-statement
-

6.3 Types and Declarations

Types can be used for

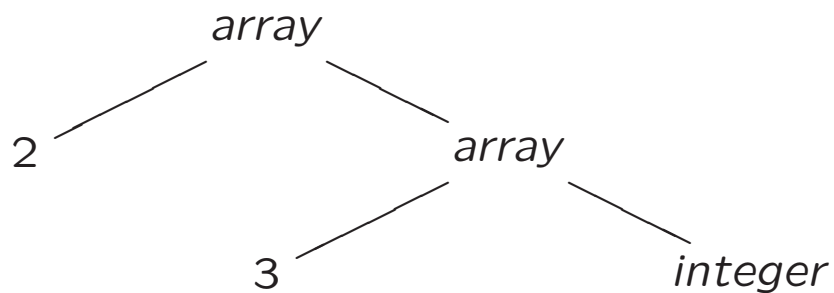
- Type checking
- Translation
 - Type information useful
 - to determine storage needed
 - to calculate address of array element
 - to insert explicit type conversions
 - to choose right version of operator
 - ...

6.3.1 Type Expressions

Types have structure

Example: array type `int[2][3]`

array(2, array(3, integer))



Type Expressions

- **Basic types:** boolean, char, integer, float, void
- **Type names:** typedefs in C, class names in C++
- **Type constructors:**
 - array
 - record: data structure with named fields
 - \rightarrow for function types: $s \rightarrow t$
 - Cartesian product \times : $s \times t$
 - ...

CFG for Function Declaration

$$\begin{aligned} F &\rightarrow B \text{ id } (OptL) \\ B &\rightarrow \text{int} \\ &\quad | \text{float} \\ OptL &\rightarrow \epsilon \\ &\quad | Ps \\ Ps &\rightarrow P \\ &\quad | Ps, P \\ P &\rightarrow T \text{ id} \end{aligned}$$

CFG for Function Declaration

F	\rightarrow	$B \text{ id } (OptL)$	$\{ F.type = \rightarrow (OptL.type, B.type); \}$
B	\rightarrow	int	$\{ B.type = integer; \}$
		float	$\{ B.type = float; \}$
$OptL$	\rightarrow	ϵ	$\{ OptL.type = void; \}$
		Ps	$\{ OptL.type = Ps.type; \}$
Ps	\rightarrow	P	$\{ Ps.type = P.type; \}$
		Ps_1, P	$\{ Ps.type = \times (Ps_1.type, P.type); \}$
P	\rightarrow	$T \text{ id}$	$\{ P.type = T.type; \}$

6.3.2 Type Equivalence

$$S \rightarrow \mathbf{id} = E \quad \{\mathbf{if} \ (id.type == E.type) \\ \mathbf{then} \ \dots; \ \mathbf{else} \ \dots\}$$

When are type expressions equivalent?

- Structural equivalence
- Name equivalence
- Use graph representation of type expressions to check equivalence
 - Leaves for basic types and type names
 - Interior nodes for type constructors
 - Cycles in case of recursively defined types...

Structural Equivalence

- Same basic type:
integer is equivalent to *integer*
- Formed by applying same constructor to structurally equivalent types
pointer(integer) is equivalent to *pointer(integer)*
- One is type name of other

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

Name equivalence ...

6.3.3 Declarations

- We need symbol tables to record global and local declarations in procedures, blocks, and structs to resolve names
- Symbol table contains type and relative address of names

Example:

$$\begin{aligned} D &\rightarrow T \mathbf{id}; D \mid \epsilon \\ T &\rightarrow B C \mid \mathbf{record} \{ D \} \\ B &\rightarrow \mathbf{int} \mid \mathbf{float} \\ C &\rightarrow \epsilon \mid [\mathbf{num}] C \end{aligned}$$

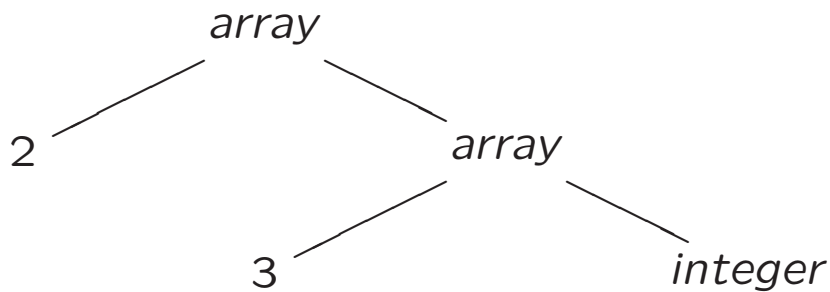
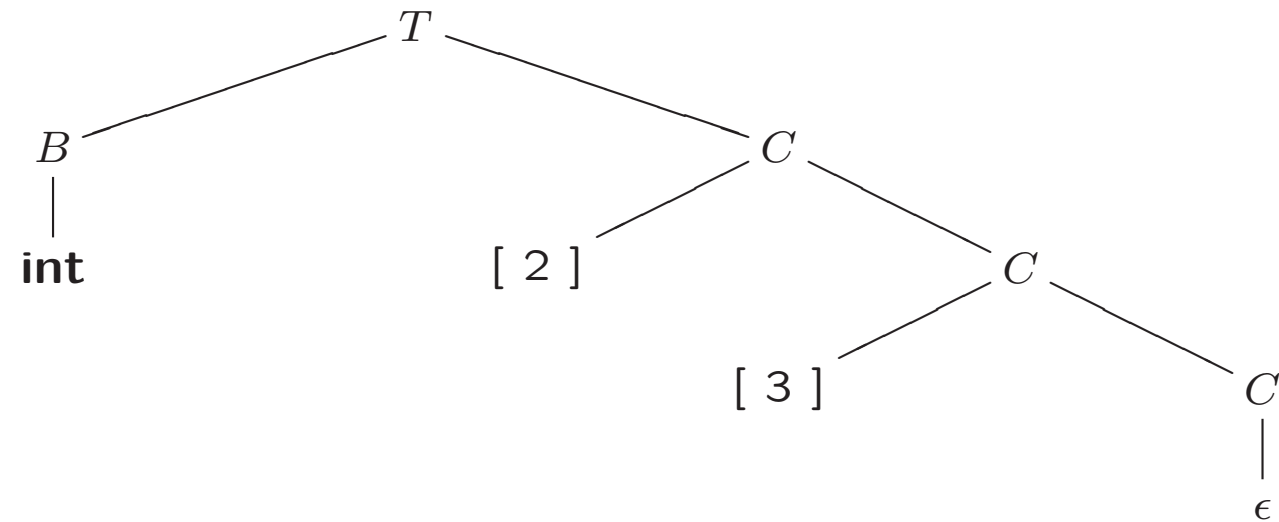
Structure of Types (Example)

$T \rightarrow B C \mid \text{record } \{ D \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \epsilon \mid [\text{num}] C$

`int [2] [3]`



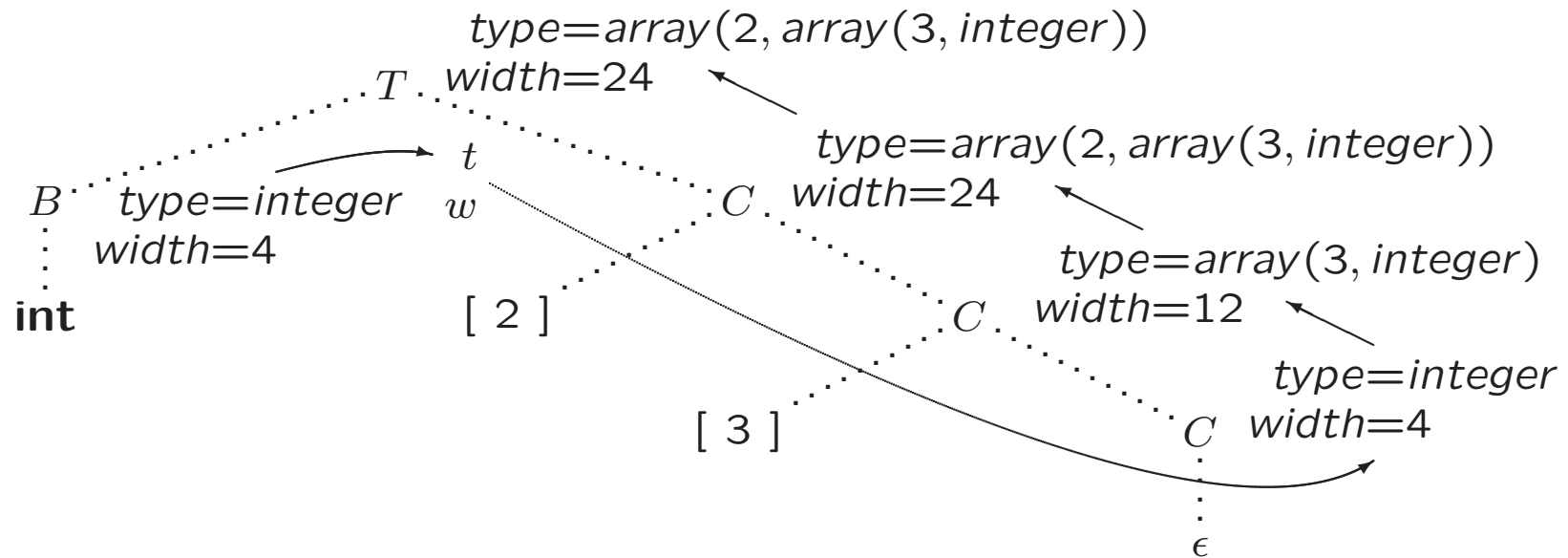
6.3.4 Storage Layout for Local Names

- Storage comes in blocks of contiguous bytes
- **Width** of type is number of bytes needed

T	\rightarrow	B	$\{$	$t = B.type; w = B.width; \}$
		C	$\{$	$T.type = C.type; T.width = C.width; \}$
B	\rightarrow	int	$\{$	$B.type = integer; B.width = 4; \}$
B	\rightarrow	float	$\{$	$B.type = float; B.width = 8; \}$
C	\rightarrow	ϵ	$\{$	$C.type = t; C.width = w; \}$
C	\rightarrow	$[\text{num}] C_1$	$\{$	$C.type = array(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$

Types and Their Widths (Example)

$T \rightarrow B$ { $t = B.type; w = B.width;$ }
 C { $T.type = C.type; T.width = C.width;$ }
 $B \rightarrow \mathbf{int}$ { $B.type = \mathit{integer}; B.width = 4;$ }
 $B \rightarrow \mathbf{float}$ { $B.type = \mathit{float}; B.width = 8;$ }
 $C \rightarrow \epsilon$ { $C.type = t; C.width = w;$ }
 $C \rightarrow [\mathbf{num}] C_1$ { $C.type = \mathit{array}(\mathbf{num.value}, C_1.type);$
 $C.width = \mathbf{num.value} \times C_1.width;$ }



6.3.5 Sequences of Declarations

$$D \rightarrow T \text{ id}; D \mid \epsilon$$

Use *offset* as next available address

$$\begin{aligned} P &\rightarrow \{ \textit{offset} = 0; \} \\ D &\rightarrow T \text{ id}; \{ \textit{top.put}(\textit{id.lexeme}, T.\textit{type}, \textit{offset}); \\ &\quad \textit{offset} = \textit{offset} + T.\textit{width}; \} \\ D &\rightarrow \epsilon \end{aligned}$$

6.3.6 Fields in Records and Classes

Example

```
float x;  
record { float x; float y; } p;  
record { int tag; float x; float y; } q;  
x = p.x + q.x;
```

$$D \rightarrow T \text{ id}; D \mid \epsilon$$
$$T \rightarrow \mathbf{record} \{ D \}$$

- Fields are specified by sequence of declarations
 - Field names within record must be distinct
 - Relative address for field is relative to data area for that record

Fields in Records and Classes

Stored in separate symbol table t

Record type has form $record(t)$

```
 $T \rightarrow$  record '{' { Env.push(top);  
                        top = new Env();  
                        Stack.push(offset);  
                        offset = 0; }  
  
       $D$  '}' { T.type = record(top);  
            T.width = offset;  
            top = Env.pop();  
            offset = Stack.pop(); }
```

6.5 Type Checking

- **Type system** contains information about
 - Syntactic constructs of language
 - Notion of types
 - Logical rules to assign types to language constructs
 - * if both operands of $+$ are integers, then result is integer
 - * if f has type $s \rightarrow t$ and x has type s , then expression $f(x)$ has type t
- **Sound** type system
- **Strongly typed** implementation of language

A Simple Type Checker

A language example (Pascal-like)

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

A Simple Type Checker

Translation scheme for saving type of identifier

P	\rightarrow	$D; S$	
D	\rightarrow	$D; D$	
D	\rightarrow	id : T	{ <i>addType</i> (id.entry , $T.type$); }
T	\rightarrow	boolean	{ $T.type = \text{boolean}$; }
T	\rightarrow	char	{ $T.type = \text{char}$; }
T	\rightarrow	integer	{ $T.type = \text{integer}$; }
T	\rightarrow	\hat{T}_1	{ $T.type = \text{pointer}(T_1.type)$; }
T	\rightarrow	array [num] of T_1	{ $T.type = \text{array}(1 \dots \text{num.val}, T_1.type)$; }

A Simple Type Checker

Type Checking of Expressions

E	\rightarrow	true	$\{E.type = boolean;\}$
E	\rightarrow	false	$\{E.type = boolean;\}$
E	\rightarrow	literal	$\{E.type = char;\}$
E	\rightarrow	num	$\{E.type = integer;\}$
E	\rightarrow	id	$\{E.type = lookup(id.entry);\}$
E	\rightarrow	E_1 and E_2	$\{if (E_1.type == boolean) and (E_2.type == boolean)$ \quad then $E.type = boolean$; else $E.type = type_error$;\}
E	\rightarrow	E_1 mod E_2	$\{if (E_1.type == integer) and (E_2.type == integer)$ \quad then $E.type = integer$; else $E.type = type_error$;\}
E	\rightarrow	$E_1[E_2]$	$\{if (E_2.type == integer) and (E_1.type == array(s, t))$ \quad then $E.type = t$; else $E.type = type_error$;\}
E	\rightarrow	E_1^{\wedge}	$\{if (E_1.type == pointer(t))$ \quad then $E.type = t$; else $E.type = type_error$;\}

A Simple Type Checker

Type Checking of Statements

$S \rightarrow \mathbf{id} := E$ {**if** ($id.type == E.type$)
 then $S.type = void$; **else** $S.type = type_error$; }

$S \rightarrow \mathbf{if} E \mathbf{then} S_1$ {**if** ($E.type == boolean$)
 then $S.type = S_1.type$; **else** $S.type = type_error$; }

$S \rightarrow \mathbf{while} E \mathbf{do} S_1$ {**if** ($E.type == boolean$)
 then $S.type = S_1.type$; **else** $S.type = type_error$; }

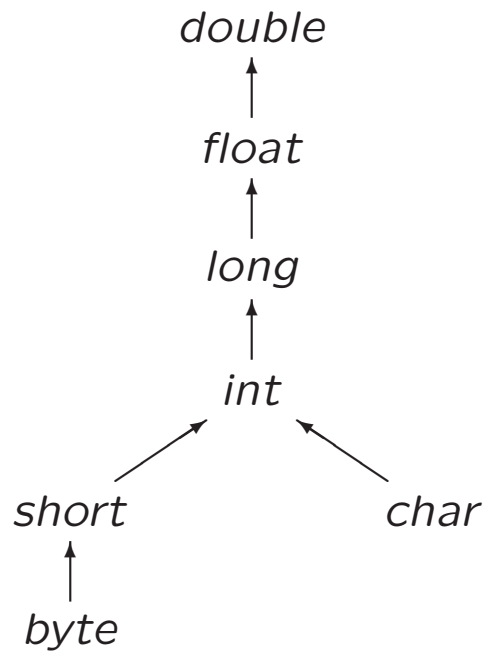
$S \rightarrow S_1; S_2$ {**if** ($S_1.type == void$) **and** ($S_2.type == void$)
 then $S.type = void$; **else** $S.type = type_error$; }

6.5.2 Type Conversions

$y = x + i$ with x float and i integer

- widening conversion
- narrowing conversion
- explicit conversion (= cast)
- implicit conversion (= coercion), automatically by compiler

Conversions in Java



Widening conversions

Coercion (Example)

Semantic action for $E \rightarrow E_1 + E_2$ uses two functions:

- $max(t_1, t_2)$
- $widen(a, t, w)$

$$E \rightarrow E_1 + E_2 \quad \left\{ \begin{array}{l} E.type = max(E_1.type, E_2.type); \\ a_1 = widen(E_1.node, E_1.type, E.type); \\ a_2 = widen(E_2.node, E_2.type, E.type); \\ E.node = \mathbf{new} \text{ Node}('+', a_1, a_2); \end{array} \right. \}$$

In book with three-address code

Coercion (Example)

```
Node widen(Node a, Type t, Type w)
{
    if (t==w) return a;
    else if (t == integer and w == float)
        { temp = new Node(inttofloat, a);
          return temp;
        }
    else error;
}
```

In book with three-address code

Constructing Type Graphs in Yacc

```
enum Types {Tint, Tfloat, Tpointer, Tarray, ...};
typedef struct Type
{ Types type;
  struct Type *child
} Type;
```

- `Type *mkint()` construct int node if not already constructed
- `Type *mkfloat()` construct float node if not already constructed
- `Type *mkarray(Type*, int)` construct array-of-type node if not already constructed
- `Type *mkptr(Type*)` construct pointer-of-type node if not already constructed

Yacc Specification (Example)

from lecture 4

```
expr    : expr '+' term    { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor  { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'     { $$ = $2; }
        | DIGIT
        ;

%%
/* auxiliary functions section */
yylex()
{   int c;
    c = getchar();
    if (isdigit(c))
    {   yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Constructing Type Graphs in Yacc

```
%union
{ Symbol *sym;
  int num;
  Type *typ;
}
%token INT
%token <sym> ID
%token <num> NUM
%type <typ> typevar

%%
decl : typevar ID          { addType($2, $1); }
     | typevar ID '[' NUM ']' { addType($2, mkarr($1,$4)); }
     ;
typevar : INT             { $$ = mkint(); }
        | typevar '^'    { $$ = mkptr($1); }
        | /* empty */    { $$ = mkint(); }
        ;
```


Type Checking in Yacc

```
%{
enum Types {Tint, Tfloat, Tpointer, Tarray, ...};
typedef struct Type
{ 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 +");
                        else
                        { $$ = mkint();
                          gen(int-add instruction for $1 and $3);
                        }
}
```

Type Coercion in Yacc

```
%{ ... %}  
%%  
expr : expr '+' expr  
    { if ($1->type == Tint && $3->type == Tint)  
      { $$ = mkint(); gen(int-add instruction for $1 and $3);  
      }  
      else if ($1->type == Tfloat && $3->type == Tfloat)  
      { $$ = mkfloat(); gen(float-add instruction for $1 and $3);  
      }  
      else if ($1->type == Tfloat && $3->type == Tint)  
      { $$ = mkfloat(); gen(int2float instruction for $3);  
        gen(float-add instruction for $1 and $3);  
      }  
      else if ($1->type == Tint && $3->type == Tfloat)  
      { $$ = mkfloat(); gen(int2float instruction for $1);  
        gen(float-add instruction for $1 and $3);  
      }  
      else  
      { semerror ("type error in +");  
        $$ = mkint();  
      }  
    }  
}
```

L-Values and R-Values

- $E_1 = E_2;$
- What can E_1 and E_2 be?
 - $i = i + 1;$
 - $i = 5;$
- L-value: left side of assignment, location
Example: identifier i , array access $a[2]$
- R-value: right side of assignment, value
Example: identifier i , array access $a[2]$, constant 5, addition $i + 1$

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;
      gen(...);
    }
| expr '=' expr
    { if ( !$1->islval || $1->typ != $3->typ )
      semerror ("invalid assignment");
      $$->typ = $1->typ;
      $$->islval = FALSE;
      gen(...);
    }
| ID
    { $$->typ = lookup($1);
      $$->islval = TRUE;
      gen(...);
    }
}
```

L-Values and R-Values in Yacc

Alternative

```
%{
typedef struct Node
{ Type *typ;
  int islval;
} Node;
%}
%union
{ Node *rec;
}

expr : expr '+' expr
      { if ($<rec>1->typ->type != Tint ||
            $<rec>3->typ->type != Tint )
          semerror ("non-int operands in +");
        $<rec>$->typ = mkint();
        $<rec>$->islval = FALSE;
        gen(...);
      }
| expr '=' expr
  ...

%%
```

Volgende week

- Practicum over opdracht 2
- Direct naar 302/304
- Staat al online
- Inleveren 27 oktober

Over twee weken

- 's Ochtends hoorcollege
- 's Middags werkcollege

Compilerconstructie

college 5

Static Type Checking

Chapters for reading: 6.1, 6.3, 6.5.1, 6.5.2