Compilerconstructie

najaar 2018

http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/

Rudy van Vliet
kamer 140 Snellius, tel. 071-527 2876
rvvliet(at)liacs(dot)nl

college 5, vrijdag 12 oktober 2018

Syntax DAG / Types

The Phases of a Compiler

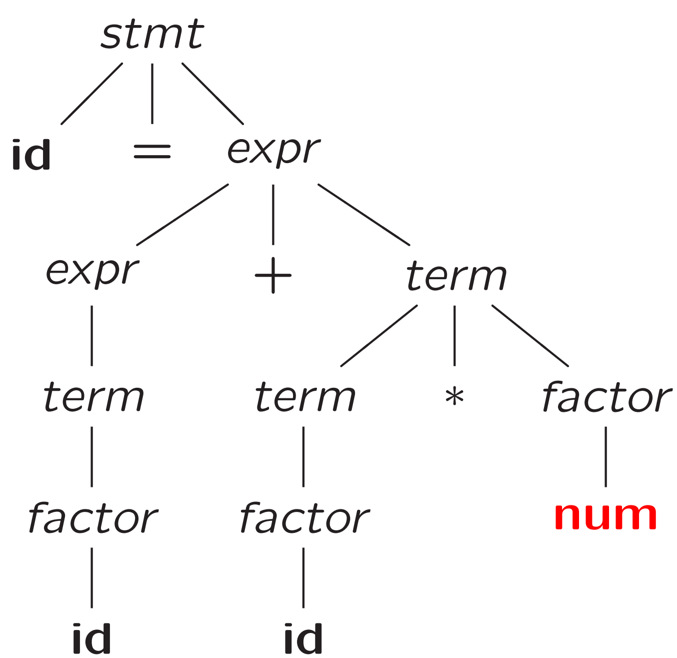
(from lecture 1)

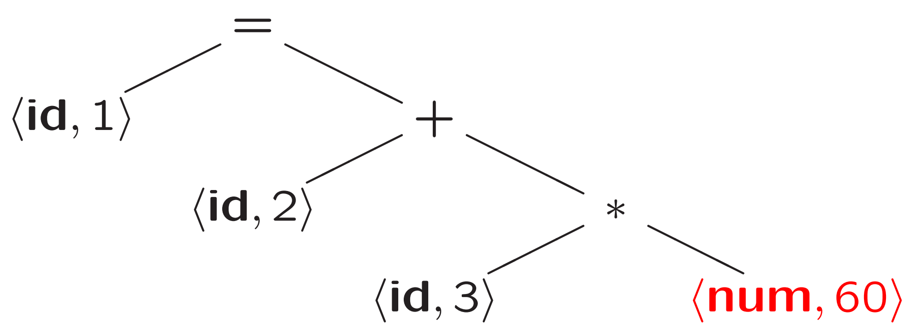
Token stream:

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

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

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

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

$\ldots$

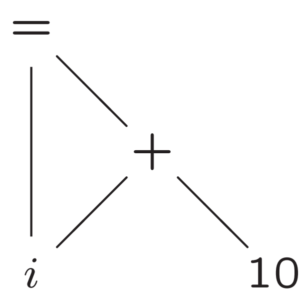
Producing Syntax Trees or DAG's

1) $p_1 = getLeaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = getLeaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = getLeaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = getLeaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = getNode('-', p_3, p_4)$
6) $p_6 = getNode('*', p_2, p_5) = getNode('*', p_1, p_5)$
7) $p_7 = getNode('+', p_1, p_6)$
8) $p_8 = getLeaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = getLeaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = getNode('-', p_8, p_9) = getNode('-', p_3, p_4) = p_5$
11) $p_{11} = getLeaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = getNode('*', p_{10}, p_{11}) = getNode('*', p_5, p_{11})$
13) $p_{13} = getNode('+', p_7, p_{12})$

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 Types and Declarations

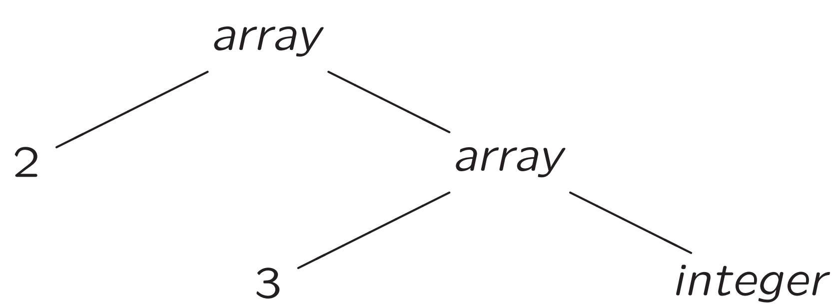
- Type expressions
- Function declaration
- Type equivalence
- Declarations of variables
- Storage layout
- Records and classes

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 $\times$: $s \times t$
  - ...

6.3 Types and Declarations

- Type expressions
- Function declaration
- Type equivalence
- Declarations of variables
- Storage layout
- Records and classes

CFG for Function Declaration

$$\begin{array}{rcl} F & \rightarrow & B\ \textbf{id}\ (\mathit{OptL}) \\ B & \rightarrow & \textbf{int} \\ & | & \textbf{float} \\ \mathit{OptL} & \rightarrow & \epsilon \\ & | & \mathit{Ps} \\ \mathit{Ps} & \rightarrow & P \\ & | & \mathit{Ps}, P \\ P & \rightarrow & T\ \textbf{id} \end{array}$$

CFG for Function Declaration

$$
\begin{array}{llll}
F & \rightarrow & B\ \textbf{id}\ (\mathit{OptL}) & \{\ F.\mathit{type} = \rightarrow (\mathit{OptL.type}, B.\mathit{type});\} \\
B & \rightarrow & \textbf{int} & \{\ B.\mathit{type} = \mathit{integer};\} \\
 & | & \textbf{float} & \{\ B.\mathit{type} = \mathit{float};\} \\
\mathit{OptL} & \rightarrow & \epsilon & \{\ \mathit{OptL.type} = \mathit{void};\} \\
 & | & \mathit{Ps} & \{\ \mathit{OptL.type} = \mathit{Ps.type};\} \\
\mathit{Ps} & \rightarrow & P & \{\ \mathit{Ps.type} = P.\mathit{type};\} \\
 & | & \mathit{Ps}_1, P & \{\ \mathit{Ps.type} = \times(\mathit{Ps}_1.\mathit{type}, P.\mathit{type});\} \\
P & \rightarrow & T\ \textbf{id} & \{\ P.\mathit{type} = T.\mathit{type};\}
\end{array}
$$

6.3 Types and Declarations

- Type expressions
- Function declaration
- Type equivalence
- Declarations of variables
- Storage layout
- Records and classes

6.3.2 Type Equivalence

$$S \rightarrow \textbf{id} = E \ \{\textbf{if}\ (id.\textit{type} == E.\textit{type}) \ \textbf{then}\ \ldots;\ \textbf{else}\ \ldots\}$$

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 Types and Declarations

- Type expressions
- Function declaration
- Type equivalence
- Declarations of variables
- Storage layout
- Records and classes

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 adress of names

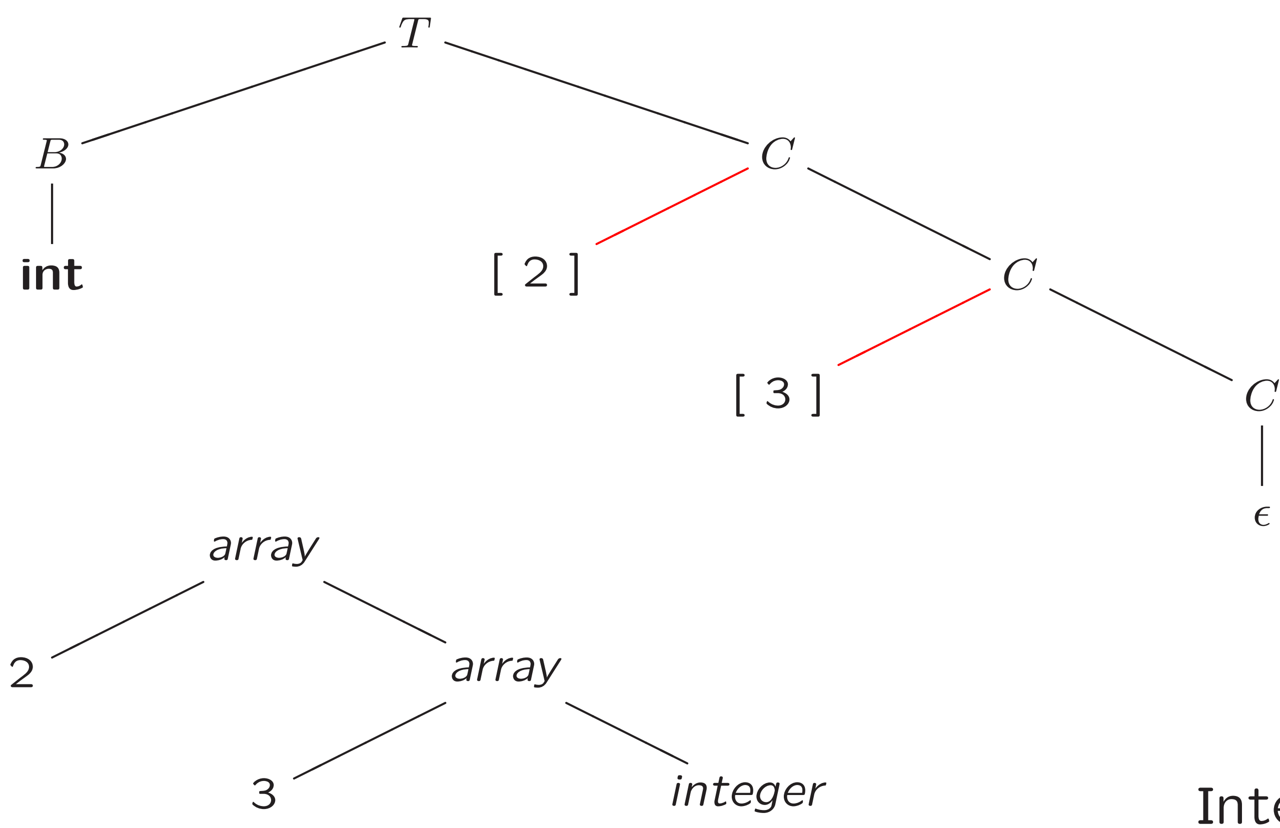
Example:

$$
\begin{array}{lcl}
D & \rightarrow & T \ \textbf{id}; D \ \mid \ \epsilon \\
T & \rightarrow & B \ C \ \mid \ \textbf{record} \ '\{' \ D \ '\}' \\
B & \rightarrow & \textbf{int} \ \mid \ \textbf{float} \\
C & \rightarrow & \epsilon \ \mid \ [\ \textbf{num}\ ] \ C
\end{array}
$$

Structure of Types (Example)

$$
\begin{array}{rcl}
T & \rightarrow & B\ C \ \mid\ \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int}\ \mid\ \textbf{float} \\
C & \rightarrow & \epsilon\ \mid\ [\ \textbf{num}\ ]\ C
\end{array}
$$

`int[2][3]`



Interpretation $C\ldots$

6.3 Types and Declarations

- Type expressions
- Function declaration
- Type equivalence
- Declarations of variables
- Storage layout
- Records and classes

6.3.4 Storage Layout for Local Names

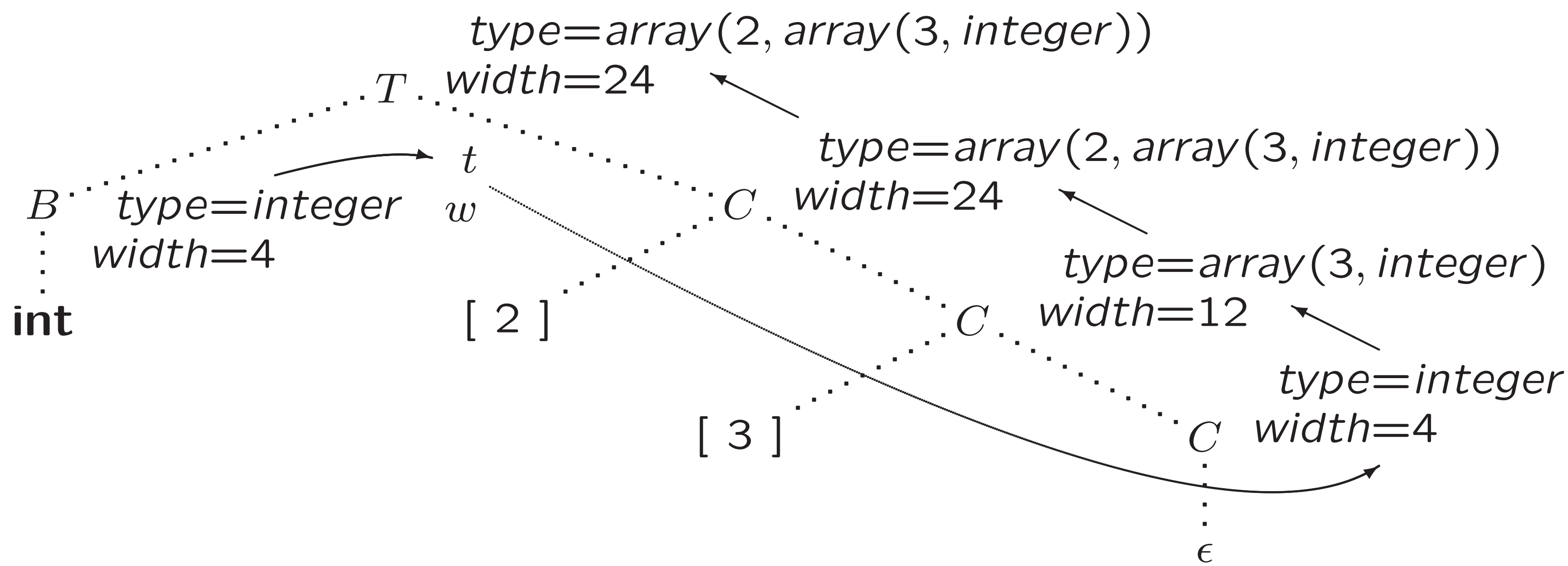
- Storage comes in blocks of contiguous bytes

- Width of type is number of bytes needed

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

Types and Their Widths (Example)

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



6.3 Types and Declarations

- Type expressions
- Function declaration
- Type equivalence
- Declarations of variables
- Storage layout
- Records and classes

6.3.5 Sequences of Declarations

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

Use *offset* as next available address

$$\begin{array}{lll}
P & \rightarrow & \{\ \mathit{offset} = 0; \} \\
 & & D \\
D & \rightarrow & T\ \textbf{id};\ \{\ \mathit{top.put}(\textbf{id}.\mathit{lexeme}, \mathrm{T}.\mathit{type}, \mathit{offset}); \\
 & & \qquad \mathit{offset} = \mathit{offset} + T.\mathit{width}; \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}$$

Example: **int** x; **float** y;

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;
```

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id};D \ \mid\ \epsilon \\
T & \rightarrow & \textbf{record}\ '\{'\ D\ '\}'
\end{array}
$$

- 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)

$$
\begin{array}{llll}
T & \rightarrow & \textbf{record}\ '\{' & \{\ \mathit{Env}.\mathit{push}(\mathit{top}); \\
 & & & \quad \mathit{top} = \textbf{new}\ \mathit{Env}(); \\
 & & & \quad \mathit{Stack}.\mathit{push}(\mathit{offset}); \\
 & & & \quad \mathit{offset} = 0; \} \\
 & & D\ '\}' & \{\ T.\mathit{type} = \mathit{record}(\mathit{top}); \\
 & & & \quad T.\mathit{width} = \mathit{offset}; \\
 & & & \quad \mathit{top} = \mathit{Env}.\mathit{pop}(); \\
 & & & \quad \mathit{offset} = \mathit{Stack}.\mathit{pop}(); \}
\end{array}
$$

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

A Simple Type Checker

A language example (Pascal-like)

- $P \rightarrow D; S$

- $D \rightarrow D; D \ \mid \ \textbf{id} : T$

- $T \rightarrow \textbf{boolean} \ \mid \ \textbf{char} \ \mid \ \textbf{integer} \ \mid \ \textbf{array [num] of } T \ \mid \ \hat{\ } T$

- $S \rightarrow \textbf{id} \ := E \ \mid \ \textbf{if } E \textbf{ then } S \ \mid \ \textbf{while } E \textbf{ do } S \ \mid \ S; S$

- $E \rightarrow \textbf{true} \ \mid \ \textbf{false} \ \mid \ \textbf{literal} \ \mid \ \textbf{num} \ \mid \ \textbf{id} \ \mid \ E \textbf{ and } E$
  $\mid \ E \textbf{ mod } E \ \mid \ E[E] \ \mid \ E\hat{\ }$

A Simple Type Checker

Translation scheme for saving type of identifier

$$
\begin{array}{lcll}
P & \rightarrow & D; S & \\
D & \rightarrow & D; D & \\
D & \rightarrow & \textbf{id} : T & \{\mathit{addType}(\textbf{id}.\mathit{entry}, T.\mathit{type});\} \\
T & \rightarrow & \textbf{boolean} & \{T.\mathit{type} = \mathit{boolean};\} \\
T & \rightarrow & \textbf{char} & \{T.\mathit{type} = \mathit{char};\} \\
T & \rightarrow & \textbf{integer} & \{T.\mathit{type} = \mathit{integer};\} \\
T & \rightarrow & \hat{}\, T_1 & \{T.\mathit{type} = \mathit{pointer}(T_1.\mathit{type});\} \\
T & \rightarrow & \textbf{array}\ [\textbf{num}\ ]\ \textbf{of}\ T_1 & \{T.\mathit{type} = \mathit{array}(1 \ldots \textbf{num}.\mathit{val}, T_1.\mathit{type});\}
\end{array}
$$

A Simple Type Checker

Type Checking of Expressions

$E \rightarrow$ **true** $\{E.\mathit{type} = \mathit{boolean};\}$

$E \rightarrow$ **false** $\{E.\mathit{type} = \mathit{boolean};\}$

$E \rightarrow$ **literal** $\{E.\mathit{type} = \mathit{char};\}$

$E \rightarrow$ **num** $\{E.\mathit{type} = \mathit{integer};\}$

$E \rightarrow$ **id** $\{E.\mathit{type} = \mathit{lookup}(\textbf{id}.\mathit{entry});\}$

$E \rightarrow E_1$ **and** E_2 $\{\textbf{if } (E_1.\mathit{type} == \mathit{boolean}) \textbf{ and } (E_2.\mathit{type} == \mathit{boolean})$
$\textbf{then } E.\mathit{type} = \mathit{boolean}; \textbf{ else } E.\mathit{type} = \mathit{type\_error};\}$

$E \rightarrow E_1$ **mod** E_2 $\{\textbf{if } (E_1.\mathit{type} == \mathit{integer}) \textbf{ and } (E_2.\mathit{type} == \mathit{integer})$
$\textbf{then } E.\mathit{type} = \mathit{integer}; \textbf{ else } E.\mathit{type} = \mathit{type\_error};\}$

$E \rightarrow E_1[E_2]$ $\{\textbf{if } (E_2.\mathit{type} == \mathit{integer}) \textbf{ and } (E_1.\mathit{type} == \mathit{array}(s,t))$
$\textbf{then } E.\mathit{type} = t; \textbf{ else } E.\mathit{type} = \mathit{type\_error};\}$

$E \rightarrow E_1\hat{}$ $\{\textbf{if } (E_1.\mathit{type} == \mathit{pointer}(t))$
$\textbf{then } E.\mathit{type} = t; \textbf{ else } E.\mathit{type} = \mathit{type\_error};\}$

A Simple Type Checker

Type Checking of Statements

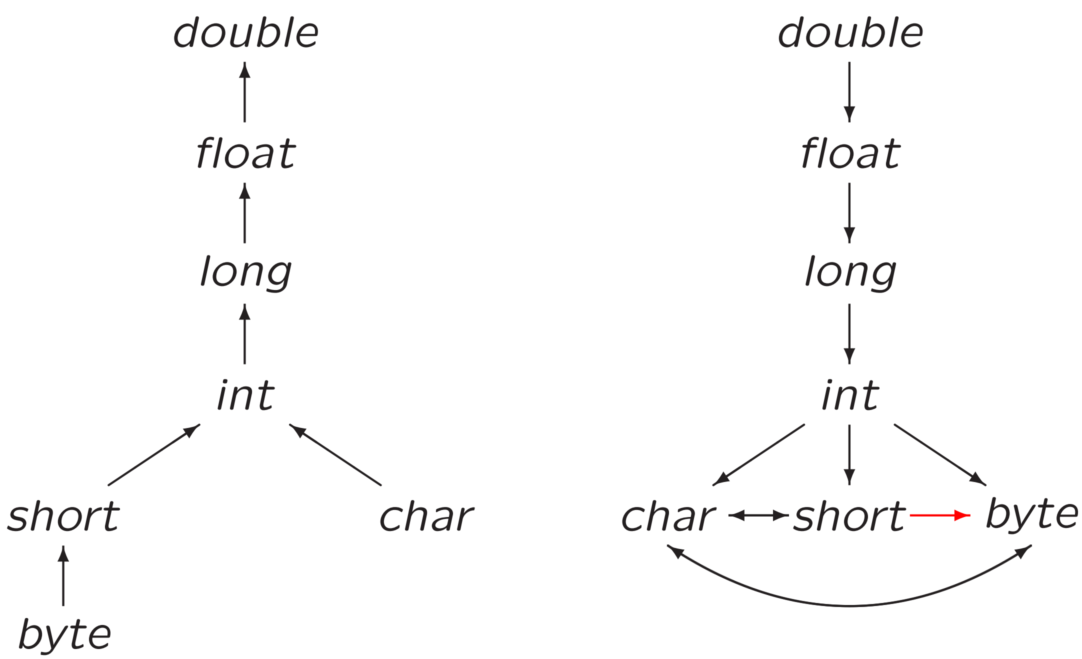
S	$\rightarrow$	**id** $:=$ E	{**if** (id.*type* == E.*type*) **then** S.*type* = *void*; **else** S.*type* = *type_error*; }
S	$\rightarrow$	**if** E **then** S_1	{**if** (E.*type* == *boolean*) **then** S.*type* = S_1.*type*; **else** S.*type* = *type_error*; }
S	$\rightarrow$	**while** E **do** S_1	{**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 Narrowing conversions

Coercion (Example)

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

- $\mathit{max}(t_1, t_2)$
- $\mathit{widen}(a, t, w)$

$$
\begin{array}{ll}
E \rightarrow E_1 + E_2 & \{ \; E.\mathit{type} = \mathit{max}(E_1.\mathit{type}, E_2.\mathit{type}); \\
& \quad a_1 = \mathit{widen}(E_1.\mathit{node}, E_1.\mathit{type}, E.\mathit{type}); \\
& \quad a_2 = \mathit{widen}(E_2.\mathit{node}, E_2.\mathit{type}, E.\mathit{type}); \\
& \quad E.\mathit{node} = \textbf{new}\ \mathit{Node}('+', a_1, a_2); \\
& \}
\end{array}
$$

Example: $x + i$ ...

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 acces $a[2]$

- R-value: right side of assignment, value
  Example: identifier i, array acces $a[2]$, 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 1 november

Over twee weken

- Vrijdagochtend: hoorcollege
- Woensdagochtend daarna: practicumbijeenkomst

Compilerconstructie

college 5

Static Type Checking

Chapters for reading: 6.1, 6.3, 6.5.1, 6.5.2