Script generated by TTT

Title: Simon: Compilerbau (17.06.2013)

Date: Mon Jun 17 14:17:18 CEST 2013

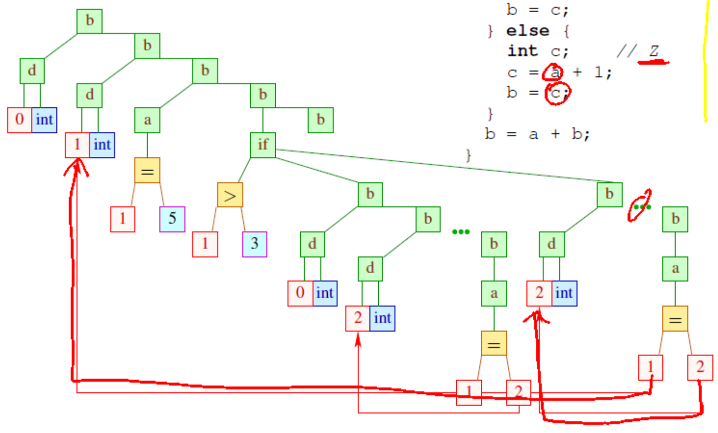
Duration: 88:10 min

Pages: 50

Resolving: Rewriting the Syntax Tree

- **d** declaration node
- **b** basic block
- **a** assignment

```
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```



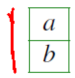
Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers

Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers

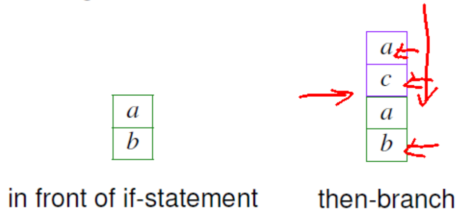
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



in front of if-statement

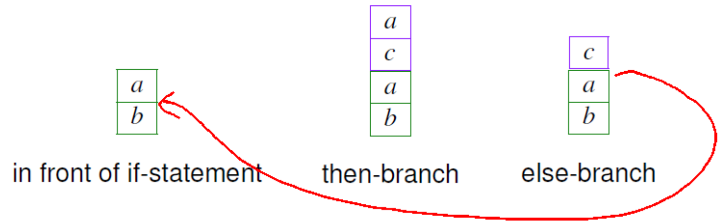
Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



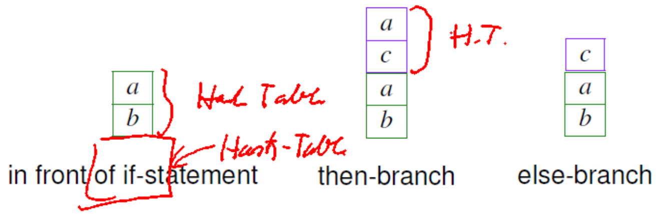
Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



Alternative Resolution of Visibility

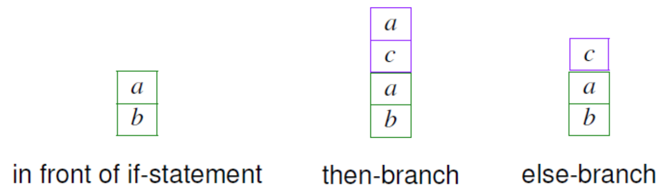
- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



- instead of lists of symbols, it is possible to use a list of hash tables ⇝ more efficient in large, shallow programs

Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



- instead of lists of symbols, it is possible to use a list of hash tables ⇝ more efficient in large, shallow programs
- a more elegant solution is to use a *persistent tree* in which an update returns a new tree but leaves all old references to the tree unchanged
  - a persistent tree t can be passed down into a basic block where new elements may be added; after examining the basic block, the analysis proceeds with the unchanged t

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

Consider the declaration of an alternating linked list in C:

```
struct list1;
struct list0 {
  int info;
  struct list1* next;
}
```

```
struct list1 {
  double info;
  struct list0* next;
}
```


Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

Consider the declaration of an alternating linked list in C:

```
struct list1;
struct list0 {
  int info;
  struct list1* next;
}
```

```
struct list1 {
  double info;
  struct list0* next;
}
```

⇝ the first declaration `struct list1;` is a forward declaration.

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

Consider the declaration of an alternating linked list in C:

```
struct list1;
struct list0 {
  int info;
  struct list1* next;
}
```

```
struct list1 {
  double info;
  struct list0* next;
}
```

⇝ the first declaration `struct list1;` is a forward declaration.

Alternative: automatically add a forward declaration into the symbol table and check that all these entries have been declared by the time the symbol goes out of scope

Declarations of Function Names

An analogous mechanism is need for (recursive) functions:

- in case a *recursive function* merely calls itself, it is sufficient to add the name of a function to the symbol table before visiting its body; example:

```
int fac(int i) {
  return i*fac(i-1);
}
```


Declarations of Function Names

An analogous mechanism is need for (recursive) functions:

- in case a *recursive function* merely calls itself, it is sufficient to add the name of a function to the symbol table before visiting its body; example:

```
int fac(int i) {
  return i*fac(i-1);
}
```


- for *mutually recursive functions* all function names at that level have to be entered (or declared as forward declaration). Example ML and C:

```
fun odd 0 = false
  | odd 1 = true
  | odd x = even (x-1)
and even 0 = true
  | even 1 = false
  | even x = odd (x-1)
```

```
int even(int x);
int odd(int x) {
  return (x==0 ? 0 :
    (x==1 ? 1 : even(x-1)));
}
int even(int x) {
  return (x==0 ? 1 :
    (x==1 ? 0 : odd(x-1)));
}
```


Overloading of Names

The problem of using names before their declarations are visited is also common in object-oriented languages:

- for object-oriented languages with inheritance, the base class must be visited before the derived class in order to determine if declarations in the derived class are correct
- in addition, the signature of methods needs to be considered ()
  - qualify a function symbol with its parameters
  - may also require type checking

Overloading of Names

The problem of using names before their declarations are visited is also common in object-oriented languages:

- for object-oriented languages with inheritance, the base class must be visited before the derived class in order to determine if declarations in the derived class are correct
- in addition, the signature of methods needs to be considered ()
  - qualify a function symbol with its parameters
  - may also require type checking

Once the names are resolved, other semantic analyses can be applied such as *type checking* or *type inference*.

Multiple Classes of Identifiers

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

Multiple Classes of Identifiers

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

In some cases a declaration may *change* the class of an identifier; for example, a **`typedef`** in C:

- the scanner generates a different token, based on the class into which an identifier falls
- the parser informs the scanner as soon as it sees a declaration that changes the class of an identifier
- the parser generates a syntax tree that depends on the semantic interpretation of the input so far

Multiple Classes of Identifiers

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

In some cases a declaration may *change* the class of an identifier; for example, a **`typedef`** in C:

- the scanner generates a different token, based on the class into which an identifier falls
- the parser informs the scanner as soon as it sees a declaration that changes the class of an identifier
- the parser generates a syntax tree that depends on the semantic interpretation of the input so far

the interaction between scanner and parser is *problematic*!

Fixity-Declarations in Haskell

Haskell allows for *arbitrary* binary operators over $(?!\hat{}\&|=+-\_\star/)^+$.
In Standard Library of Haskell:

```
infixr 8 ^
infixl 7 *,/
infixl 6 +,-
infix  4 ==,/=
```

The grammar is *generic*:

$$
\begin{array}{lcl}
Exp_0 & ::= & Exp_0 \; LOp_0 \; Exp_1 \\
 & | & Exp_1 \; ROp_0 \; Exp_0 \\
 & | & Exp_1 \; Op_0 \; Exp_1 \\
 & | & Exp_1 \\
 & & \vdots \\
Exp_9 & ::= & Exp_9 \; LOp_9 \; Exp \\
 & | & Exp \; ROp_9 \; Exp_9 \\
 & | & Exp \; Op_9 \; Exp \\
 & | & Exp \\
Exp & ::= & \textbf{ident} \mid \textbf{num} \\
 & | & (\; Exp_0 \;)
\end{array}
$$

Fixity-Declarations in Haskell

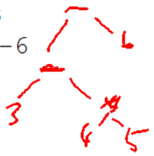
Haskell allows for *arbitrary* binary operators over `(?!^&|=+-_*/)`$^+$.
In Standard Library of Haskell:

```
infixr 8 ^
infixl 7 *,/
infixl 6 +,-
infix  4 ==,/=
```

The grammar is *generic*:

$$
\begin{array}{lcl}
Exp_0 & ::= & Exp_0\ LOp_0\ Exp_1 \\
 & | & Exp_1\ ROp_0\ Exp_0 \\
 & | & Exp_1\ Op_0\ Exp_1 \\
 & | & Exp_1 \\
 & \vdots & \\
Exp_9 & ::= & Exp_9\ LOp_9\ Exp \\
 & | & Exp\ ROp_9\ Exp_9 \\
 & | & Exp\ Op_9\ Exp \\
 & | & Exp \\
Exp & ::= & \textbf{ident} \mid \textbf{num} \\
 & | & \textbf{(}\ Exp_0\ \textbf{)}
\end{array}
$$

- parser enters an `infix` declaration into a table
- scanner checks table and produces:
  - operator `-` turns into token LOp_6.
  - operator `*` turns into token LOp_7.
  - operator `==` turns into token Op_4.
  - etc.
- ⇝ parser recognizes `3-4*5-6` as `(3-(4*5))-6`



Fixity-Declarations in Haskell: Observations

Troublesome changes:

- the scanner has a *state* which the parser determines
  ⇝ grammar no longer *context-free*, needs global data structure

Fixity-Declarations in Haskell: Observations

Troublesome changes:

- the scanner has a *state* which the parser determines
  ⇝ grammar no longer *context-free*, needs global data structure
- a code fragment may have several semantics
- syntactic correctness may depend on imported modules

Fixity-Declarations in Haskell: Observations

Troublesome changes:

- the scanner has a *state* which the parser determines
  ⇝ grammar no longer *context-free*, needs global data structure
- a code fragment may have several semantics
- syntactic correctness may depend on imported modules
- error messages difficult to understand

The GHC Haskell Compiler parses all operators as LOp_0 and transforms the AST afterwards.

Type Synonyms and Variables in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Type Synonyms and Variables in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced

Type Synonyms and Variables in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced
- parser state has at least one look-ahead token

Type Synonyms and Variables in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced
- parser state has at least one look-ahead token
- the scanner has already read `point_t` in line two as identifier

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator `;`
declaration-specifier	→	`static` | `volatile` ··· `typedef`
	|	`void` | `char` | `char` ··· `typedef-name`
declarator	→	`identifier` | ···

Solution is difficult:

- try to fix the look-ahead inside the parser

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator `;`
declaration-specifier	→	`static` | `volatile` ··· `typedef`
	|	`void` | `char` | `char` ··· `typedef-name`
declarator	→	`identifier` | ···

Solution is difficult:

- try to fix the look-ahead inside the parser
- add the following rule to the grammar:
  typedef-name → `identifier`

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator `;`
declaration-specifier	→	`static` | `volatile` ··· `typedef`
	|	`void` | `char` | `char` ··· `typedef-name`
declarator	→	`identifier` | ···

Solution is difficult:

- try to fix the look-ahead inside the parser
- add the following rule to the grammar:
  typedef-name → `identifier`
- register type name earlier

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator `;`
declaration-specifier	→	`static` | `volatile` ··· `typedef`
	|	`void` | `char` | `char` ··· `typedef-name`
declarator	→	`identifier` | ···

Solution is difficult:

- try to fix the look-ahead inside the parser
- add the following rule to the grammar:
  typedef-name → `identifier`
- register type name earlier
  - separate rule for `typedef` production

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

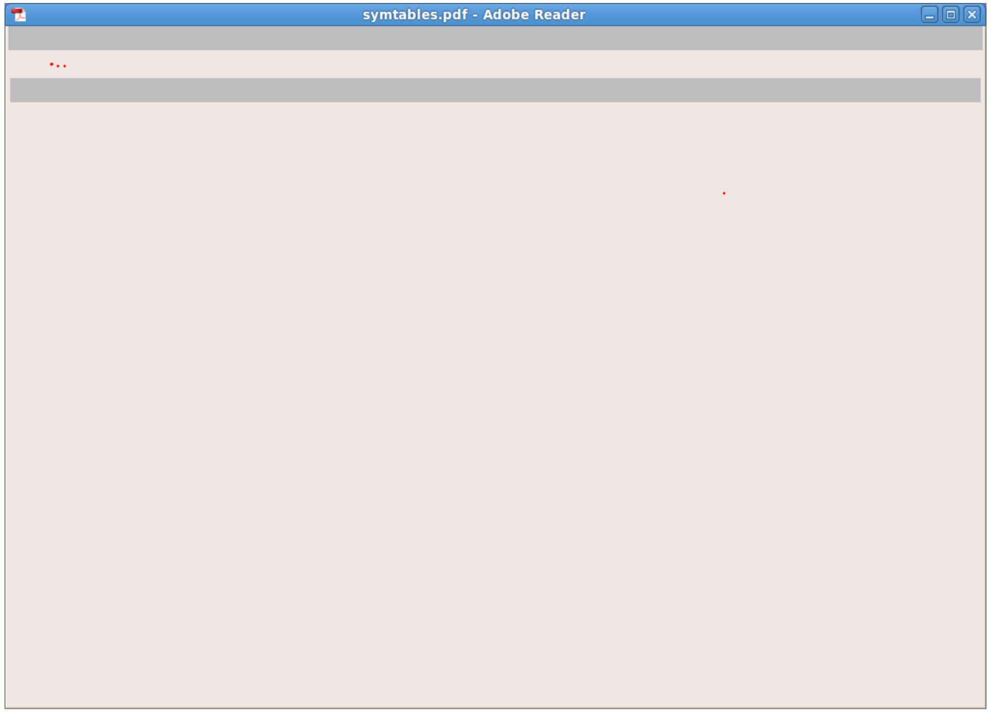
declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator `;`
declaration-specifier	$\rightarrow$	`static` | `volatile` ··· `typedef`
	|	`void` | `char` | `char` ··· `typedef-name`
declarator	$\rightarrow$	`identifier` | ···

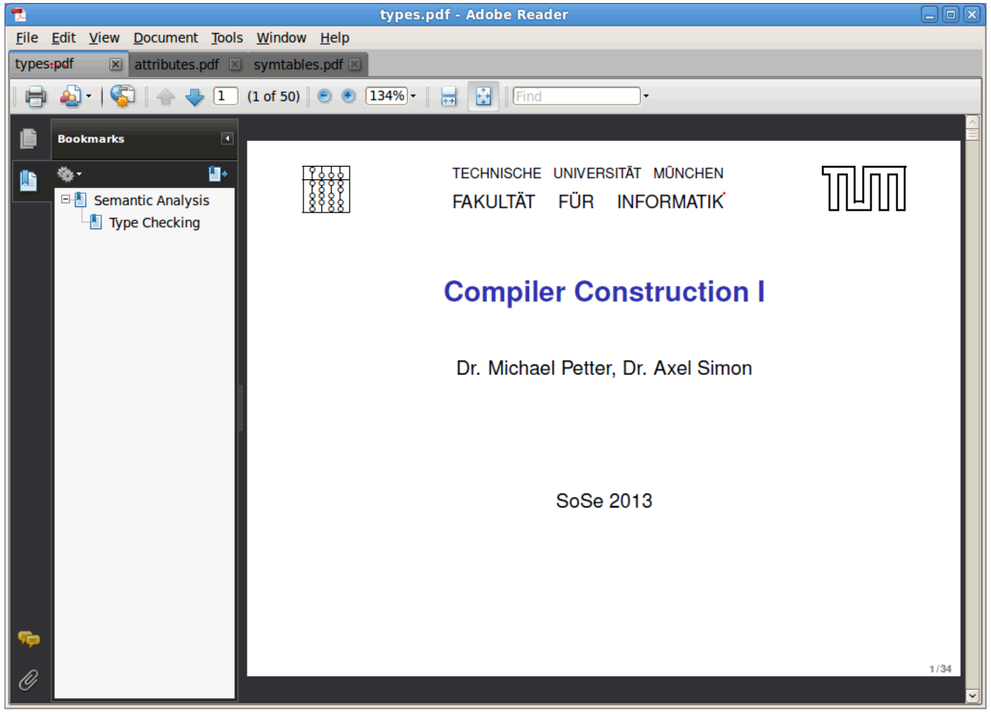
Solution is difficult:

- try to fix the look-ahead inside the parser
- add the following rule to the grammar:
  typedef-name $\rightarrow$ `identifier`
- register type name earlier
  - separate rule for `typedef` production
  - call alternative declarator production that registers `identifier` as type name

Outlook

- seminar: implement symbol tables for C
- lecture: check types of programs





Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type.
for example: `int`, `void*`, `struct { int x; int y; }`.

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type.
for example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type.
for example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.

Type Expressions

Types are given using type-*expressions*.
The set of type expressions T contains:

1. base types: **int**, **char**, **float**, **void**, ...
2. type constructors that can be applied to other types

Type Expressions

Types are given using type-*expressions*.
The set of type expressions T contains:

1. base types: **int**, **char**, **float**, **void**, ...
2. type constructors that can be applied to other types

example for type constructors in C:

- records: **struct** $\{\ t_1\ a_1; \ldots t_k\ a_k;\ \}$
- pointer: $t\ \star$
- arrays: $t\ [\,]$
  - the size of an array can be specified
  - the variable to be declared is written between t and $[n]$
- functions: $t\ (t_1, \ldots, t_k)$
  - the variable to be declared is written between t and $(t_1, \ldots, t_k)$
  - in ML function types are written as: $t_1 * \ldots * t_k \rightarrow t$

Type Definitions in C

A type definition is a *synonym* for a type expression.
In C they are introduced using the **typedef** keyword.
Type definitions are useful

- as abbreviation:

```
typedef struct { int x; int y; } point_t;
```

- to construct *recursive* types:

Possible declaration in C:

```
struct list {
  int info;
  struct list* next;
}

struct list* head;
```

more readable:

```
typedef struct list list_t;
struct list {
  int info;
  list_t* next;
}
list_t* head;
```


Type Checking

Problem:

Given: a set of type declarations $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
Check: Can an expression e be given the type t?

Type Checking

Problem:

Given: a set of type declarations $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
Check: Can an expression e be given the type t?

Example:

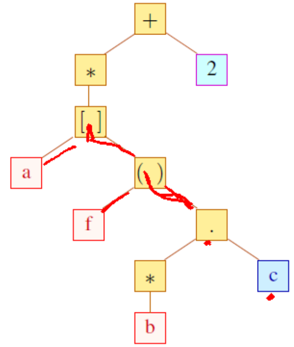
```
struct list { int info; struct list* next; };
int f(struct list* l) { return 1; };
struct { struct list* c;}* b;
int* a[11];
```

Consider the expression:

```
*a[f(b->c)]+2;
```


Type Checking using the Syntax Tree

Check the expression `*a[f(b->c)]+2`:



Idea:

- traverse the syntax tree bottom-up
- for each identifier, we lookup its type in Γ
- constants such as 2 or 0.5 have a fixed type
- the types of the inner nodes of the tree are deduced using *typing rules*

Type Systems

Formal consider *judgements* of the form:

$$\Gamma \vdash e : t$$

// (in the type environment Γ the expression e has type t)

Axioms:

Const: $\Gamma \vdash c : t_c$ (t_c type of constant c)
Var: $\Gamma \vdash x : \Gamma(x)$ (x Variable)

Regeln:

Ref: $\dfrac{\Gamma \vdash e : t}{\Gamma \vdash \&e : t*}$ Deref: $\dfrac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t}$

Type Systems for C-like Languages

More rules for typing an expression:

Array: $\dfrac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1[e_2] : t}$

Array: $\dfrac{\Gamma \vdash e_1 : t[\,] \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1[e_2] : t}$

Struct: $\dfrac{\Gamma \vdash e : \mathbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$

App: $\dfrac{\Gamma \vdash e : t\,(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}$

Op: $\dfrac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$

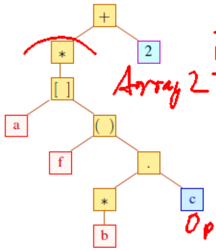
Cast: $\dfrac{\Gamma \vdash e : t_1 \qquad t_1 \text{ can be converted to } t_2}{\Gamma \vdash (t_2)\ e : t_2}$

Example: Type Checking

Given expression `*a[f(b->c)]+2` and $\Gamma = \{$

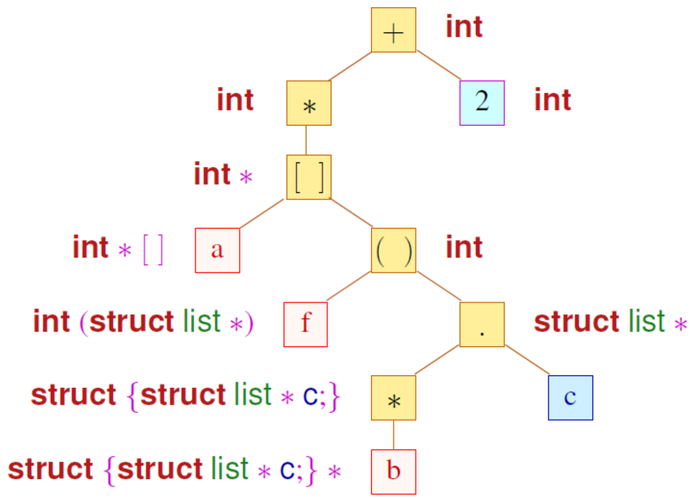
```
struct list { int info; struct list* next; };
int f(struct list* l);
struct { struct list* c;}* b;
int* a[11];
```

$\}$;



Example: Type Checking

Expression `*a[f(b->c)]+2`:



Equality of Types

Summary type checking:

- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- ⇝ determining the rule requires a check for *equality* of types

type equality in C:

- **struct** A {} and **struct** B {} are considered to be different
  - ⇝ the compiler could re-order the fields of A and B independently (*not* allowed in C)
  - to extend an record A with more fields, it has to be embedded into another record:

```
typedef struct B {
          struct A a;
          int field_of_B;
        } extension_of_A;
```

- after issuing **typedef int** C; the types C and **int** are the same