Kapitel 7: Semantische Analyse

Aufgabe

Überprüfen und Auswerten der Typinformationen (Deklarationen)

Themen

- Typ – Ausdrücke
- Symboltabelle
- Typ – Überprüfung
- Overloading

Typ-Ausdrücke

Strukturierte Typen können durch Ausdrücke bzw. Bäume beschrieben werden

- **Ein Basistyp ist ein Typ-Ausdruck, z.B.** `bool,int,real,char`
- **Ein Typ-Konstruktor, angewandt auf Typ-Ausdrücke T, ist ein Typ-Ausdruck**

`array(n,T)`

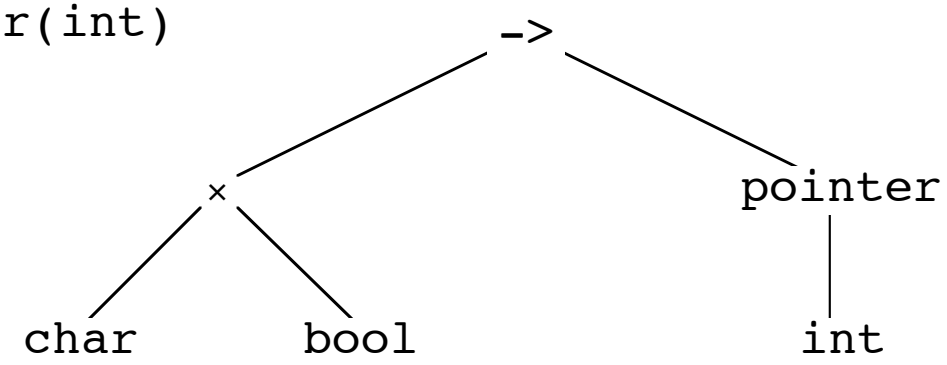
$T_1 \times \ldots \times T_k$

`pointer(T)`

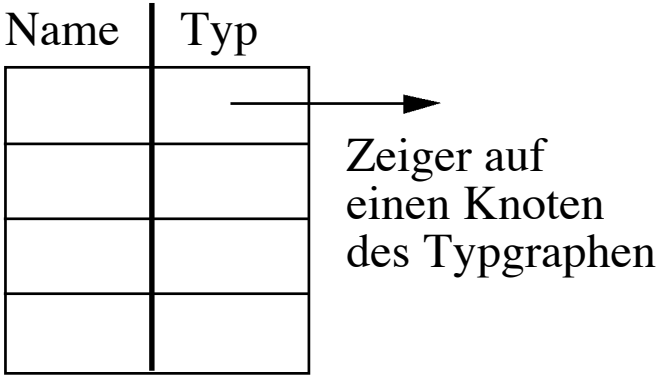
T_1 `->` T_2

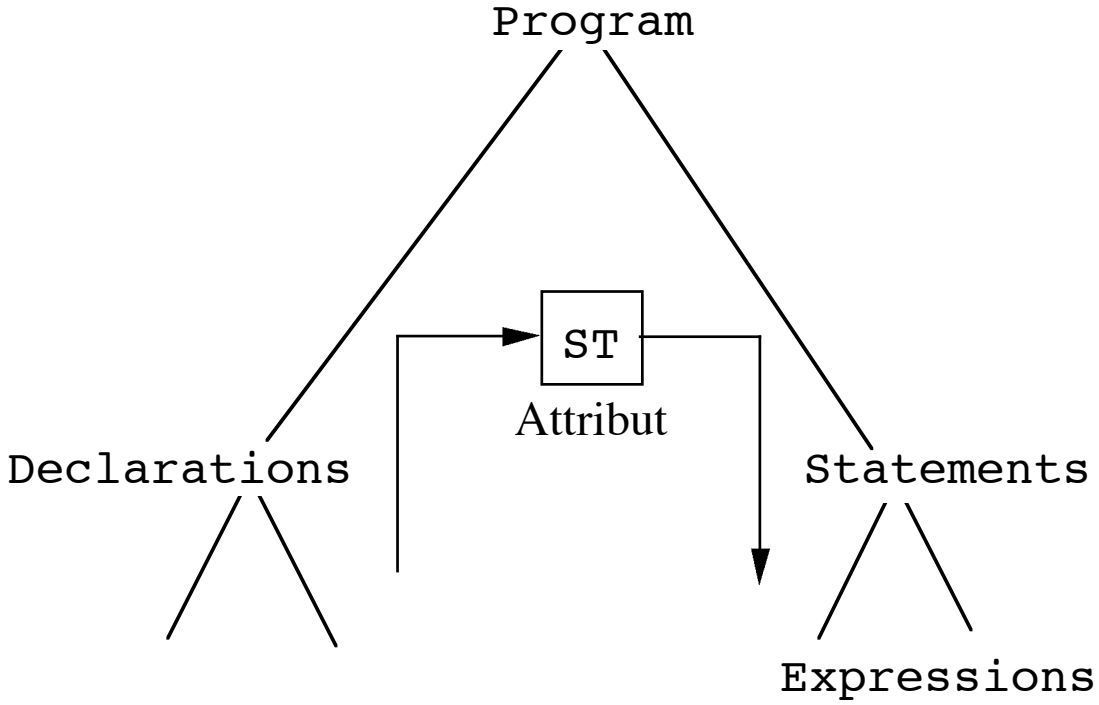
Typ-Ausdruck als Graph

`char` × `bool -> pointer(int)`



Symboltabelle





AG zur Erzeugung der Symboltabelle

Produktion	Regeln
P -> D S	D.i = () S.i = D.s
D -> D_1 ; D_2	D_1.i = D.i D_2.i = D_1.s D.s = D_2.s
D -> id : T	D.s = D.i ‖ (id.x,T.type)
T -> int	T.type = int
T -> real	T.type = real
D -> proc id P	h = D.i ‖ (id.x,proc) D.s = h P.i = h
P -> D S	D.i = P.i S.i = D.s

AG zur Typ-Überprüfung

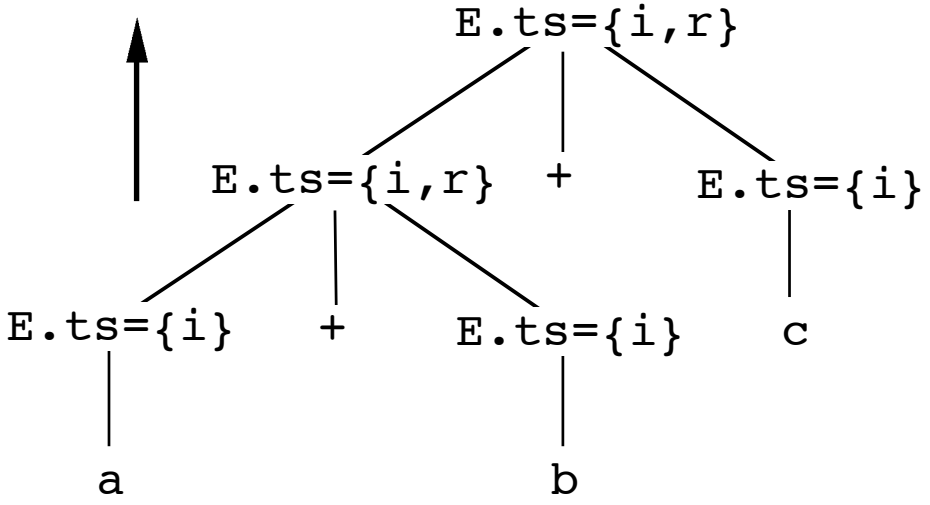
Produktion	Regeln
P -> D S	D.i = () S.i = D.s
S -> S_1 ; S_2	S_1.i = S.i S_2.i = S.i
S -> id := E	E.i = S.i if (looktype(id.x,S.i)<>E.type) error
E -> E_1 + E_2	E_1.i = E.i E_2.i = E.i E.type = if (E_1.type==int && E_2.type==int) int else if (E_1.type==real && E_2.type==real) real else error
E -> id	E.type = looktype(id.x,E.i)

Overloading in Ada

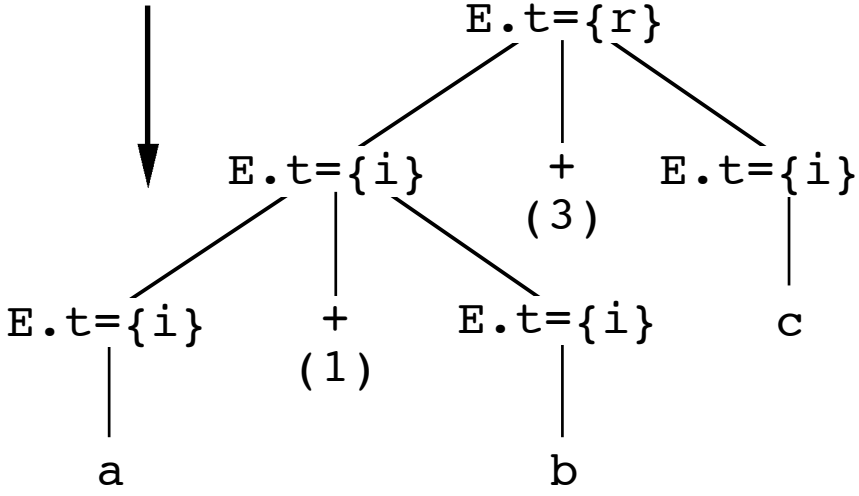
Operator wird durch Argument-Typen und Ergebnis-Typ identifiziert

```
+:(1)int×int->int,(2)real×real->real,(3)int×int->real

a,b,c:int; y:real

y = a+b+c
```

1. Typmengen aufgrund des Ausdrucks



2. Eindeutigkeit durch Ergebnis-Typ



Ox-Lex Beispiel

```
%{
#include "beispiel1.h"
#include "oxout.tab.h"
%}

comment         \/\/.*
number          [0-9]+
register        %r([abcd]x|[sb]p|[sd]i|[89]|1[0-5])
whitespace      [\n\t ]

%%

"+="            return (PLUSASSIGNOP);
"="             return (ASSIGNOP);
"+"             return (PLUSOP);
";"             return (';');
{register}      return (REGISTER);
                        @{ @REGISTER.name@ = strdup(yytext); @}
{number}        return (NUMBER); @{ @NUMBER.val@ = atol(yytext); @}
{whitespace}+   ;
{comment}       ;

.               printf("Lexical error.\n"); exit(1);
```

Ox Beispiel

```
%token REGISTER NUMBER ';' ASSIGNOP PLUSASSIGNOP
%left PLUSOP

@attributes { char* name; } REGISTER
@attributes { long val; } NUMBER
@attributes { treenode *n; } stmt expr constexpr
@traversal @preorder codegen

%{
treenode *newOpNode(int op, treenode *left, treenode *right);
treenode *newRegNode(char* name);
treenode *newNumNode(long num);

extern void invoke_burm(NODEPTR_TYPE root);
%}

%start  stmt_list

%%

stmt_list:      /* empty */
        |       stmt ';' stmt_list
                @{
                        @codegen invoke_burm(@stmt.n@);
                @}
        ;
```

```
stmt    :   REGISTER ASSIGNOP expr
                @{ @i @stmt.n@ = newOpNode(ASSIGN,
                          newRegNode(@REGISTER.name@),@expr.n@); @}
        |       REGISTER PLUSASSIGNOP expr
                @{ @i @stmt.n@ = newOpNode(ADDASSIGN,
                          newRegNode(@REGISTER.name@),@expr.n@); @}
        ;

expr    :       REGISTER
                @{ @i @expr.n@ = newRegNode(@REGISTER.name@); @}
        |       constexpr
                @{ @i @expr.n@ = @constexpr.n@; @}
        ;

constexpr:      NUMBER
                @{ @i @constexpr.n@ = newNumNode(@NUMBER.val@); @}
        |       constexpr PLUSOP constexpr
                @{ @i @constexpr.n@ = newOpNode(ADD,
                              @constexpr.1.n@,@constexpr.2.n@); @}
        ;

%%

int yyerror(char *e) {...}
int main(void) {yyparse(); return 0;}
```