Typen, Type Checking und Attributierte Grammatiken

BC George (FH Bielefeld)

Motivation

Ist das alles erlaubt?

Operation erlaubt?

Zuweisung erlaubt?

Welcher Ausdruck hat welchen Typ?

(Welcher Code muss dafür erzeugt werden?)

- a = b
- a = f(b)
- a = b + c
- a = b + o.nummer
- if (f(a) == f(b))

Taschenrechner: Parsen von Ausdrücken wie 3*5+4

```
expr : expr '+' term
     | term
     ;
term : term '*' DIGIT
     | DIGIT
     ;

DIGIT : [0-9] ;
```

=> Wie den Ausdruck **ausrechnen**?

Semantische Analyse: Die Symboltabellen nutzen

Das haben wir bis jetzt (1/2)

Wir haben Einträge in den Symboltabellen angelegt und dafür gesorgt, dass wir in den richtigen Scopes Definitionen in der richtigen Reihenfolge suchen können.

Zum Auflösen von Deklarationen und Zuordnen von Objekten zu Klassen ist mindestens ein zweiter Lauf über Syntaxbaum und/oder Symboltabellen.

Der Parse Tree enthält bei allen Namen Verweise in die Symboltabellen. Die Symboltabelleneinträge für Variablen und Objekte enthalten jetzt die Typen der Variablen und Objekte, bzw. Verweise auf ihre Typen in den Symboltabellen.

Das haben wir bis jetzt (2/2)

Dabei konnten schon einige semantische Eigenschaften des zu übersetzenden Programms überprüft werden, falls erforderlich z. B.:

- Wurden alle Variablen / Objekte vor ihrer Verwendung definiert oder deklariert?
- Wurden keine Elemente mehrfach definiert?
- Wurden alle Funktionen / Methoden mit der richtigen Anzahl Parameter aufgerufen? (Nicht in allen Fällen schon prüfbar)
- Haben Arrayzugriffe auch keine zu hohe Dimension?
- Werden auch keine Namen benutzt, für die es keine Definition / Deklaration gibt?

Was fehlt jetzt noch?

Es müssen kontextsensitive Analysen durchgeführt werden, allen voran Typanalysen. Damit der “richtige” (Zwischen-) Code entsprechend den beteiligten Datentypen erzeugt werden kann, muss mit Hilfe des Typsystems der Sprache (aus der Sprachdefinition) überprüft werden, ob alle Operationen nur mit den korrekten Datentypen benutzt werden. Dazu gehört auch, dass nicht nur Typen von z. B. Variablen, sondern von ganzen Ausdrücken betrachtet, bzw. bestimmt werden. Damit kann dann für die Codeerzeugung festgelegt werden, welcher Operator realisiert werden muss (Überladung).

Analyse von Datentypen

Typisierung

- stark oder statisch typisierte Sprachen: Alle oder fast alle Typüberprüfungen finden in der semantischen Analyse statt (C, C++, Java)
- schwach oder dynamisch typisierte Sprachen: Alle oder fast alle Typüberprüfungen finden zur Laufzeit statt (Python, Lisp, Perl)
- untypisierte Sprachen: keinerlei Typüberprüfungen (Maschinensprache)

Ausdrücke

Jetzt muss für jeden Ausdruck im weitesten Sinne sein Typ bestimmt werden.

Ausdrücke können hier sein:

- rechte Seiten von Zuweisungen
- linke Seiten von Zuweisungen
- Funktions- und Methodenaufrufe
- jeder einzelne aktuelle Parameter in Funktions- und Methodenaufrufen
- Bedingungen in Kontrollstrukturen

Typinferenz

Def.: *Typinferenz* ist die Bestimmung des Datentyps jedes Bezeichners und jedes Ausdrucks im Code.

Der Typ eines Ausdrucks wird mit Hilfe der Typen seiner Unterausdrücke bestimmt.

Dabei kann man ein Kalkül mit sog. Inferenzregeln der Form

$$\frac{f : s \to t \qquad x : s}{f(x) : t}$$

(Wenn f den Typ $s \to t$ hat und x den Typ s, dann hat der Ausdruck f(x) den Typ t.)

benutzen. So wird dann z. B. auch Überladung aufgelöst und Polymorphie zur Laufzeit.

Statische Typprüfungen

Bsp.: Der + - Operator:

Typ 1. Operand	Typ 2. Operand	Ergebnistyp
int	int	int
float	float	float
int	float	float
float	int	float
string	string	string

Typkonvertierungen

- Der Compiler kann implizite Typkonvertierungen vornehmen, um einen Ausdruck zu verifizieren (siehe Sprachdefiniton).
- In der Regel sind dies Typerweiterungen, z.B. von *int* nach *float*.
- Manchmal muss zu zwei Typen der kleinste Typ gefunden werden, der beide vorhandenen Typen umschließt.
- Explizite Typkonvertierungen heißen auch *Type Casts*.

Nicht grundsätzlich statisch mögliche Typprüfungen

Bsp.: Der ^ - Operator (a^b):

Typ 1. Operand	Typ 2. Operand	Ergebnistyp
int	int ≥ 0	int
int	int < 0	float
int	float	float
...	...	...

Attributierte Grammatiken

Was man damit macht

Die Syntaxanalyse kann keine kontextsensitiven Analysen durchführen.

- Kontextsensitive Grammatiken benutzen: Laufzeitprobleme, das Parsen von cs-Grammatiken ist *PSPACE − complete*.
- Der Parsergenerator *Bison* generiert LALR(1)-Parser, aber auch sog. *Generalized LR (GLR) Parser*, die bei nichtlösbaren Konflikten in der Grammatik (Reduce/Reduce oder Shift/Reduce) parallel den Input mit jede der Möglichkeiten weiterparsen.
- Ein weiterer Ansatz, kontextsensitive Abhängigkeiten zu berücksichtigen, ist der Einsatz von attributierten Grammatiken, nicht nur zur Typanalyse, sondern evtl. auch zur Codegenerierung.

Syntax-gesteuerte Übersetzung: Attribute und Aktionen

Berechnen der Ausdrücke

```
expr : expr '+' term ;
```

```
translate expr ;
translate term ;
handle + ;
```

Attributierte Grammatiken (SDD)

auch "*syntax-directed definition*"

Anreichern einer CFG:

- Zuordnung einer Menge von Attributen zu den Symbolen (Terminal- und Nicht-Terminal-Symbole)
- Zuordnung einer Menge von *semantischen Regeln* (Evaluationsregeln) zu den Produktionen

Definition: Attributierte Grammatik

Eine *attributierte Grammatik AG = (G,A,R)* besteht aus folgenden Komponenten:

- $G = (N, T, P, S)$ ist eine cf-Grammatik
- $A = \bigcup_{X \in (T \cup N)} A(X)$ mit $A(X) \cap A(Y) \neq \emptyset \Rightarrow X = Y$
- $R = \bigcup_{p \in P} R(p)$ mit $R(p) = \{X_i.a = f(\ldots) | p : X_0 \rightarrow X_1 \ldots X_n \in P, X_i.a \in A(X_i), 0 \leq i \leq n\}$

Abgeleitete und ererbte Attribute

Die in einer Produktion definierten Attribute sind

$AF(P) = \{X_i.a \mid p : X_0 \to X_1 \dots X_n \in P, 0 \leq i \leq n, X_i.a = f(\dots) \in R(p)\}$

Wir betrachten Grammatiken mit zwei disjunkten Teilmengen, den abgeleiteten (synthesized) Attributen $AS(X)$ und den ererbten (inherited) Attributen $AI(X)$:

$AS(X) = \{X.a \mid \exists p : X \to X_1 \dots X_n \in P, X.a \in AF(p)\}$

$AI(X) = \{X.a \mid \exists q : Y \to uXv \in P, X.a \in AF(q)\}$

Abgeleitete Attribute geben Informationen von unten nach oben weiter, geerbte von oben nach unten.

Die Abhängigkeiten der Attribute lassen sich im sog. *Abhängigkeitsgraphen* darstellen.

Beispiel: Attributgrammatiken

Produktion	Semantische Regel
`e : e1 '+' t ;`	`e.val = e1.val + t.val`
`e : t ;`	`e.val = t.val`
`t : t1 '*' D ;`	`t.val = t1.val * D.lexval`
`t : D ;`	`t.val = D.lexval`

Produktion	Semantische Regel
`t : D t' ;`	`t'.inh = D.lexval`
	`t.syn = t'.syn`
`t' : '*' D t'1 ;`	`t'1.inh = t'.inh * D.lexval`
	`t'.syn = t'1.syn`
`t' :` ϵ `;`	`t'.syn = t'.inh`

Wenn ein Nichtterminal mehr als einmal in einer Produktion vorkommt, werden die Vorkommen nummeriert. (t, t1; t', t'1)

S-Attributgrammatiken und L-Attributgrammatiken

S-Attributgrammatiken und L-Attributgrammatiken

S-Attributgrammatiken: Grammatiken mit nur abgeleiteten Attributen, lassen sich während des Parsens mit LR-Parsern bei beim Reduzieren berechnen mittels Tiefensuche mit Postorder-Evaluation:

```
def visit(N):
    for each child C of N (from left to right):
        visit(C)
    eval(N)     # evaluate attributes of N
```

L-Attributgrammatiken: Grammatiken mit Attributen, die nur von einem Elternknoten oder einem linken Geschwisterknoten abhängig sind. Sie können während des Parsens mit LL-Parsern berechnet werden. Ein links-nach-rechts-Durchlauf ist ausreichend.

Alle Kanten im Abhängigkeitsgraphen gehen nur von links nach rechts.

S-attributierte SDD sind eine Teilmenge von L-attributierten SDD.

Beispiel: S-Attributgrammatik

Produktion	Semantische Regel
`e : e1 '+' t ;`	`e.val = e1.val + t.val`
`e : t ;`	`e.val = t.val`
`t : t1 '*' D ;`	`t.val = t1.val * D.lexval`
`t : D ;`	`t.val = D.lexval`

Beispiel: Annotierter Syntaxbaum für `5*8+2`

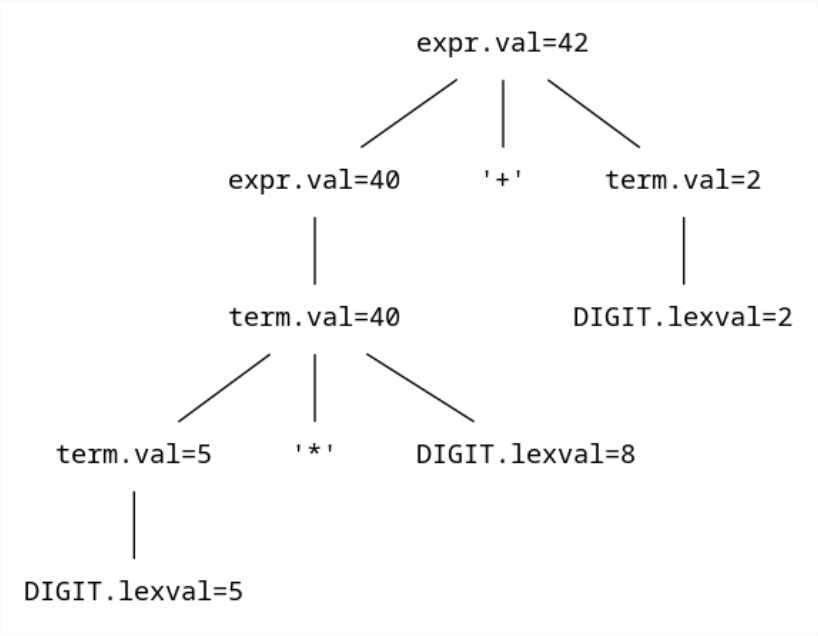


Abbildung 1: Annotierter Parse-Tree

Erzeugung des AST aus dem Parse-Tree für 5*8+2

Produktion	Semantische Regel
`e : e1 '+' t ;`	`e.node = new Node('+', e1.node, t.node)`
`e : t ;`	`e.node = t.node`
`t : t1 '*' D ;`	`t.node = new Node('*', t1.node, new Leaf(D, D.lexval));`
`t : D ;`	`t.node = new Leaf(D, D.lexval);`

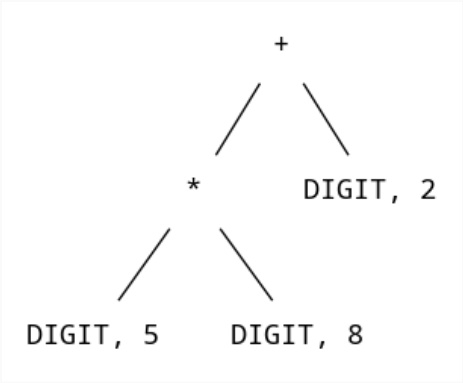
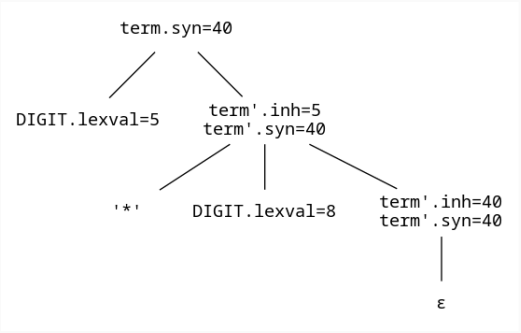


Abbildung 2: AST

Beispiel: L-Attributgrammatik, berechnete u. geerbte Attribute, ohne Links-Rekursion

Produktion	Semantische Regel
`t : D t' ;`	`t'.inh = D.lexval`
	`t.syn = t'.syn`
`t' : '*' D t'1 ;`	`t'1.inh = t'.inh * D.lexval`
	`t'.syn = t'1.syn`
`t' :` ϵ `;`	`t'.syn = t'.inh`

`5*8` $\Rightarrow$



Beispiel: Typinferenz für `3+7+9` oder `"hello"+"world"`

Produktion	Semantische Regel
`e : e1 '+' t ;`	`e.type = f(e1.type, t.type)`
`e : t ;`	`e.type = t.type`
`t : NUM ;`	`t.type = "int"`
`t : NAME ;`	`t.type = "string"`

Syntax-gesteuerte Übersetzung (SDT)

Erweiterung attributierter Grammatiken

Syntax-directed translation scheme:

Zu den Attributen kommen **Semantische Aktionen**: Code-Fragmente als zusätzliche Knoten im Parse Tree an beliebigen Stellen in einer Produktion, die, wenn möglich, während des Parsens, ansonsten in weiteren Baumdurchläufen ausgeführt werden.

```
e : e1  {print e1.val;}
    '+' {print "+";}
    t   {e.val = e1.val + t.val; print(e.val);}
  ;
```

S-attributierte SDD, LR-Grammatik: Bottom-Up-Parsierbar

Die Aktionen werden am Ende jeder Produktion eingefügt (“postfix SDT”).

Produktion	Semantische Regel
`e : e1 '+' t ;`	`e.val = e1.val + t.val`
`e : t ;`	`e.val = t.val`
`t : t1 '*' D ;`	`t.val = t1.val * D.lexval`
`t : D ;`	`t.val = D.lexval`

```
e : e1 '+' t  {e.val = e1.val + t.val; print(e.val);} ;
e : t         {e.val = t.val;} ;
t : t1 '*' D  {t.val = t1.val * D.lexval;} ;
t : D         {t.val = D.lexval;} ;
```

L-attributierte SDD, LL-Grammatik: Top-Down-Parsierbar (1/2)

Produktion	Semantische Regel
`t : D t' ;`	`t'.inh = D.lexval`
	`t.syn = t'.syn`
`t' : '*' D t'1 ;`	`t'1.inh = t'.inh * D.lexval`
	`t'.syn = t'1.syn`
`t' :` ϵ `;`	`t'.syn = t'.inh`

```
t  : D {t'.inh = D.lexval;} t' {t.syn = t'.syn;} ;
t' : '*' D {t'1.inh = t'.inh * D.lexval;} t'1 {t'.syn = t'1.syn;} ;
t' : e {t'.syn = t'.inh;} ;
```

L-attributierte SDD, LL-Grammatik: Top-Down-Parsierbar (2/2)

- LL-Grammatik: Jede L-attributierte SDD direkt während des Top-Down-Parsens implementierbar/berechenbar
- SDT dazu:
  - Aktionen, die ein berechnetes Attribut des Kopfes einer Produktion berechnen, an das Ende der Produktion anfügen
  - Aktionen, die geerbte Attribute für ein Nicht-Terminalsymbol A berechnen, direkt vor dem Auftreten von A im Körper der Produktion eingefügen
- Implementierung im rekursiven Abstieg:
  - Geerbte Attribute sind Parameter für die Funktionen für die Nicht-Terminalsymbole
  - berechnete Attribute sind Rückgabewerte dieser Funktionen.

```
T t'(T inh) {
    match('*');
    T t1inh = inh * match(D);
    return t'(t1inh);
}
```

Bison: Attribute und Aktionen

Berechnete (*synthesized*) Attribute

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

Berechnete Attribute sind der Defaultfall in Bison.

Erinnerung: Keine Typen deklariert:

- Bison verwendet per Default `int` für alle Symbole (Terminalsymbole (Token) und Regeln).

Keine Aktionen an den Regeln angegeben:

- Bison nutzt die Default-Aktion `$$ = $1`. Diese Aktionen werden immer dann ausgeführt, wenn die rechte Seite der zugehörigen Regel/Alternative reduziert werden konnte.

Geerbte (*inherited*) Attribute (1/2)

```
functiondecl : returntype fname paramlist ;

returntype  : REAL    { $$ = 1; }
            | INT     { $$ = 2; }
            ;

fname : IDENTIFIER;

paramlist : IDENTIFIER           { mksymbol($0, $-1, $1); }
          | paramlist IDENTIFIER { mksymbol($0, $-1, $2); }
          ;
```

Geerbte (*inherited*) Attribute (2/2)

Hier:

- `returntype` und `fname` haben normale berechnete Attribute
- `paramlist`: Funktionsaufruf mit den erzeugten Werte für `returntype` und `fname` als Parameter $\Rightarrow$ der Wert von `paramlist` ist ein "geerbtes Attribut".

Zugriff auf die Werte der Symbole auf dem Stack links vom aktuellen Symbol: `$0` ist das erste Symbol links vom aktuellen (hier `type`), `$-1` das zweite (hier `class`) usw. ...

Probleme mit geerbten Attributen

```
functiondecl : returntype fname paramlist ;
functiondecl : STRING paramlist ;  /* Autsch! */

...

paramlist : IDENTIFIER           { mksymbol($0, $-1, $1); }
          | paramlist IDENTIFIER { mksymbol($0, $-1, $2); }
          ;
```

Wenn vor `paramlist` ein `STRING` steht, ist `$0` der Wert von `STRING`, nicht `fname`. Analog für `$-1`, ...

Dies ist eine Quelle für schwer zu findende Bugs!

Typen für geerbte Attribute

```
functiondecl : returntype fname paramlist ;

paramlist : IDENTIFIER           { mksymbol($0, $-1, $1); }
          | paramlist IDENTIFIER { mksymbol($0, $-1, $2); }
          ;
```

Achtung: Für geerbte Attribute funktioniert die Deklaration von Typen mit `%type` nicht mehr!

Das Symbol, auf das man sich mit `$0` bezieht, steht nicht in der Produktion, sondern im Stack. Bison kann zur Compilezeit nicht den Typ des referenzierten Symbols bestimmen. Falls oben die Typen von `returntype` und `fname` jeweils `rval` und `fval` wären, müsste man die Aktion manuell wie folgt anpassen:

```
paramlist : IDENTIFIER           { mksymbol($<fval>0, $<rval>-1, $1); }
          | paramlist IDENTIFIER { mksymbol($<fval>0, $<rval>-1, $2); }
          ;
```

Bison und Aktionen

Regeln ohne Aktion ganz rechts: die Default-Aktion ist `$$ = $1;` (Vorsicht: Die Typen von `$$` und `$1` müssen passen!)

Aktionen mitten in einer Regel:

```
xxx : A { dosomething(); } B ;
```

wird übersetzt in:

```
xxx : A dummy B ;
dummy : /* empty */ { dosomething(); }
```

Da nach dem Shiften von `A` nicht klar ist, ob diese Regel matcht und `dosomething` ausgeführt werden soll, übersetzt Bison die Regel `xxx` in zwei Regeln, wobei `dosomething()` ganz rechts in der Dummy-Regel steht. `dummy` ist ein normales referenzierbares Symbol.

Beispiel:

```
xxx : A { $$ = 42; } B C { printf("%d", $2); } ;
```

=> Hier wird “42” ausgegeben, da mit `$2` auf den Wert der eingebetteten Aktion zugegriffen wird.

`$3`: Der Wert von `B`

`$4`: Der Wert von `C`

Bison: Konflikte durch eingebettete Aktionen

```
xxx : a | b ;

a : 'a' 'b' 'a' 'a' ;
b : 'a' 'b' 'a' 'b' ;
```

Diese Grammatik ist ohne Konflikte von Bison übersetzbar.

```
xxx : a | b ;

a : 'a' 'b' { dosomething(); } 'a' 'a' ;
b : 'a' 'b' 'a' 'b' ;
```

Nach dem Lesen von "`ab`" gibt es wegen des identischen Vorschauzeichens (`'a'`) einen Shift/Reduce-Konflikt.

Wrap-Up

Wrap-Up

- Die Typinferenz benötigt Informationen aus der Symboltabelle
- Einfache semantische Analyse: Attribute und semantische Regeln (SDD)
- Umsetzung mit SDT: Attribute und eingebettete Aktionen
- Reihenfolge der Auswertung u.U. schwierig

  Bestimmte SDT-Klassen können direkt beim Parsing abgearbeitet werden:

  - S-attributierte SDD, LR-Grammatik: Bottom-Up-Parsierbar
  - L-attributierte SDD, LL-Grammatik: Top-Down-Parsierbar

  Ansonsten werden die Attribute und eingebetteten Aktionen in den Parse-Tree integriert und bei einer (späteren) Traversierung abgearbeitet.

LICENSE