Minisprache Core

Kleine imperative Programmiersprache, statisch typisiert.
Spezifikation der Analyse liegt vollständig vor.

Ziele:

- **Spezifikation der Typanalyse** kennenlernen
  -> Spezifikation verstehen, Sprache erweitern
- **Technik der Source-To-Source-Übersetzung** erlernen
  -> Abschnitt 7, Spezifikation vervollständigen

Core Beispielprogramm:

```
program
let
     a:    int;
     x, y: real;
     i = 10;
in
     input a;
     x := 3.5;
     y := a * i + x;
     output y;
end
```

Sprachkonstrukte:

- Block
- Variablen- und Konstantendeklarationen
- Zuweisungen
- E/A Anweisungen
- Ausdrücke

Erweiterungen

- Ablaufstrukturen
- geschachtelte Blöcke
- weitere Operatoren

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 601

Ziele:

Beispielsprache vorstellen

im Vorlesungsteil:

Ziele und Vorgehen erläutern

nachlesen:

Core-Spezifikation

Verständnisfragen:

Welche Aufgaben der Typanalyse erkennen Sie an dem Beispielprogramm?

Core: Struktur, Namensanalyse

Konkrete Syntax:

```
Program:         Source.
Source:          'program' Block.
Block:           'let' Declarations 'in'
                       Statements 'end'.
Declarations:    Declaration*.
Statements:      Statement*.

Declaration:     DefIdents ':' TypeDenoter ';'.
DefIdents:       DefIdent // ','.
DefIdent:        Ident.
TypeDenoter:     UseTypeIdent.
UseTypeIdent:    Ident.

Statement:       UseIdent ':=' Expression ';'.
Statement:       'input' UseIdent ';'.
Statement:       'output' Expression ';'.
...
Operand:         UseIdent.
Operand:         IntLiteral.
Operand:         FloatLiteral.
UseIdent:        Ident.
```

Ausdruckssyntax folgt später.
Symbolnotation wie in Pascal.

Namensanalyse (siehe Folie 5.4):

- prüfen: alle Bezeichner definiert, alle Definitionen eindeutig
- Bezeichner `int, real` mit Modul `PreDefine` vordefinieren:

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
```

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 602

Ziele:

Syntax- und Aufgabenübersicht

im Vorlesungsteil:

Erläuterung der Namensanalyse

nachlesen:

Core-Spezifikation

Übungsaufgaben:

Arbeiten Sie die I-Aufgaben der Core-Spezifikation bis einschließlich der Namensanalyse durch.

Aufgaben und Begriffe der Typanalyse

Typisierte Programmobjekte (z. B. Variable)

- haben **Typ als Eigenschaft**;
- Typ wird durch Definition des Objektes bestimmt,
- Typ wird bei Benutzungen verwendet (Prüfung, Analyse)

Typisierte Programmkonstrukte (z. B. Ausdrücke)

- haben **Typ als Eigenschaft (Attribut);**
- sprachspezifische Regeln zur Bestimmung und Prüfung der Typen

Typen sind Programmobjekte

- Sie haben charakterisierende Eigenschaften; diese können wieder Typen enthalten.
- **vordefinierte Typen** (z. B. `int, real`)
- Typen, die durch **Typangaben** im Programm eingeführt werden (z. B. `record ... end`)
- **Typbezeichner** benennen einen Typ;
  Bindung Name - Typ durch **Typdefinition;**
  mehrere Namen können an denselben Typ gebunden sein

Relationen über Typen

- zwei Typen sind **gleich**,
  wenn sie dasselbe Typobjekt sind oder benennen, oder
  wenn sie sprachspezifische Regeln erfüllen
- ein Typ `tn` ist **verträglich** mit einem Typ `tw`,
  wenn Werte von `tn` als Werte von `tw` aufgefasst oder
  in solche umgewandelt werden können (**Coercion**).
- evtl. weitere sprachspezifische Relationen über Typen

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 603

Ziele:

Übersicht zu Typanalyseaufgaben in statisch typisierten Programmiersprachen

im Vorlesungsteil:

Erläuterungen anhand von Core-Beispielen

nachlesen:

Abschnitt Datentypen im GdP-Skript; Abschnitt Typanalyse im Übersetzer-Skript (U-40 bis U-43)

Verständnisfragen:

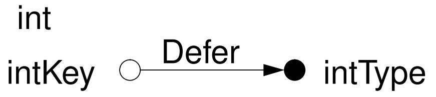
Geben Sie nichttriviale Beispiele für

- Restriktionen zum Typ eines Ausdruckes,
- Restriktionen in Typangaben,
- Typgleichheit,
- Typverträglichkeit.

Typrepräsentation des Moduls **`Typing`**

Typen werden durch **`DefTableKey`**s repräsentiert
`NoKey`: fehlerhafter, unbekannter Typ.

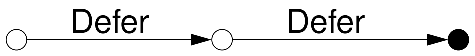
Relation `Defer` bindet Typ-Keys an die Keys von Typnamen:




```
type t = record ... end
```

Indirekte Bindungen werden erst bei Anwendungen des Typs aufglöst; vermeidet Analysezyklen:

```
type t = s; s = record ... end
```



Die Funktion **`TransDefer`** angewandt auf einen Typnamens-Key oder Typ-Key liefert den Typ-Key am Ende der Defer-Kette. Voraussetzung: alle Defer-Relationen existieren.

Defer-Ketten werden vom Modul **nicht zyklisch** erzeugt.

Eigenschaften von Typen werden immer den **Typ-Keys** (am Ende der Defer-Kette) zugeordnet.

Typ-Keys werden durch die Eigenschaft **`IsType`** von anderen Keys unterschieden.

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 604

Ziele:

Nutzen und Gebrauch der Defer-Ketten verstehen

im Vorlesungsteil:

- kurze Erläuterung des Prinzips;
- Solange es in Core keine Typdefinitionen gibt, wird das Defer-Prinzip nicht ausgenutzt

nachlesen:

Dokumentation des Moduls Defer

Verständnisfragen:

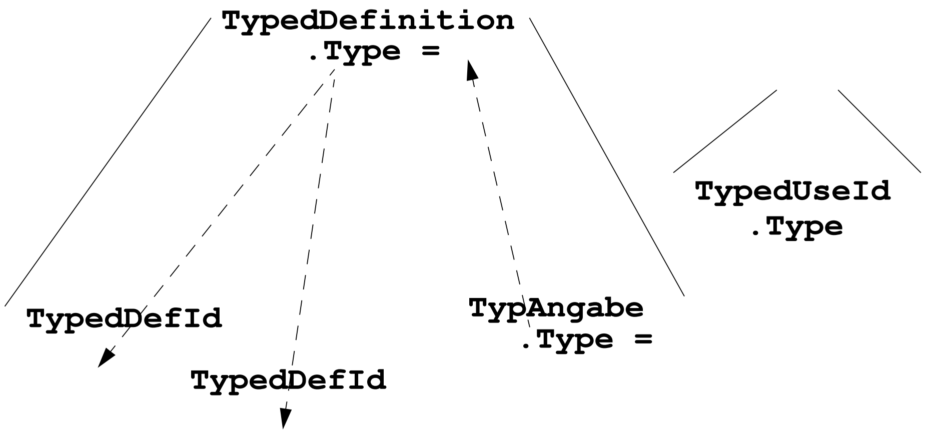
- Skizzieren Sie die Repräsentation eines korrekten, rekursiv definierten Typs.
- Geben Sie fehlerhaft rekursiv definierte Typen an.

Typisierte Objekte

Programmobjekte (Variable, Parameter, usw.) haben einen Typ als Eigenschaft **`TypeOf`**.

Symbolrollen des Moduls Typing

- setzen die `TypeOf`-Eigenschaft (**`TypedDefId`**)
- im Kontext von Definitionen (**`TypedDefinition`**) und
- machen sie bei Anwendungen (**`TypedUseId`**) als Attribut **`Type`** verfügbar (Wert: Typ-Key am Ende der Defer-Kette)
- kennzeichnet Wurzel der Grammatik (**`RootType`**)



Zusammenwirken der Rollen garantiert die Reihenfolge für das Setzen und Lesen von `TypeOf` und Benutzen der Defer-Kette.

Beliebige **weitere Eigenschaften** können zusammen mit dem Typ gesetzt werden. Sie sind zusammen mit dem Typ verfügbar

```
SYMBOL DefIdent INHERITS TypedDefId COMPUTE
  SYNT.GotType = ResetKind (THIS.Key, ...);
END;
SYMBOL UseIdent INHERITS TypedUseId COMPUTE
  SYNT.Kind = GetKind (THIS.Key, ...)
    <- THIS.Type;
END;
```

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 605

Ziele:

Schema typisierter Definitionen verstehen

im Vorlesungsteil:

Erläuterungen mit Beispielen der Core-Spezifikation

nachlesen:

Dokumentation des Moduls Typing

Übungsaufgaben:

Arbeiten sie in der Core-Spezifikation die I-Aufgaben bis vor den Abschnitt "Operatoren in Ausdrücken" durch.

Verständnisfragen:

Geben Sie Beispiele, für die Notwendigkeit einer Kind-Eigenschaft.

Typnotationen

Typangaben z. B. Deklarationen, Signaturen, usw. sind **angewandte Typbezeichner** (Rolle **`TypeDefUseId`**) oder **Konstrukte, die einen Typ beschreiben** (z. B. Record, Array) (Rolle **`TypeDenotation`**)

Die Rolle **`TypeDenotation`** bildet ein **neues Typobjekt** (Type-Key). Beliebige **Eigenschaften des Typs** können in diesem Kontext gesetzt werden. Das Attribut **`GotType`** garantiert ihre Verfügbarkeit an Anwendungsstellen.

```
SYMBOL ArrayType INHERITS TypeDenotation END;
RULE: ArrayType ::= TypAngabe '[' Number ']'
COMPUTE
  ArrayType.GotType =
    ORDER (ResetElemType (ArrayType.Type,
                          TypAngabe.Type),
           ResetElemNo (ArrayType.Type, Number));
END;
```

Bei **Typbezeichnern als Typangabe (`TypeDefUseId`)** wird der Namens-Key als Typ verwendet; die Defer-Kette wird erst bei Anwendungen aufgelöst.

Bei **Anwendungen von Typeigenschaften**, die wieder Typen sind, muß **`TransDefer`** angewandt werden;
Vorbedingung: `<- INCLUDING RootType.GotType`
ist hier nicht nötig, da indirekt durch `Variable[2].Type`.

```
RULE: Variable ::= Variable '[' Expr ']' COMPUTE
  Variable[1].Type =
     TransDefer (GetElemType (Variable[2].Type,
                              NoKey));
END;
```

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 606

Ziele:

Analyse von Typangaben verstehen

im Vorlesungsteil:

Typnotationen werden zunächst nicht besprochen, da sie in Core (noch) nicht vorkommen.

Übungsaufgaben:

- Entwerfen Sie Notationen für Typangaben zu den wichtigsten Typen in Programmiersprachen
- Geben sie die charakterisierenden Typeigenschaften dazu an.

Überladene Operatoren

Ein Operatorsymbol (**Quelloperator**) kann verschiedene Bedeutungen (**Zieloperator**) haben, sie werden unterschieden durch die Typen der Operanden.
Werkzeug **OIL** und Modul **`Operator`** unterstützen die Analyse.

Ausdrucksgrammatik, z. B.

```
Expression: Expression AddOpr Factor / Factor.
Factor:     Factor MulOpr Operand / Operand.
Operand:    MonOpr Operand.
```

Beschreibung der **Quelloperatoren**
(Zeichen, konkr. Symbol, abstr. Symbol, eindeutiger Name)

```
SrcOpr ('+', AddOpr, BinOpr, AddKey)
SrcOpr ('*', MulOpr, BinOpr, MulKey)
SrcOpr ('-', MonOpr, UnOpr,  NegKey)
```

Beschreibung der **Zieloperatoren**
(Quellopr., eindeutiger Name, Signatur, Eigenschaftsinit.)

```
TgtOpr (AddKey, iAddKey, (intType,intType):intType,    TtStr={"+"})
TgtOpr (AddKey, rAddKey, (realType,realType):realType, TgtStr={"||"})
TgtOpr (MulKey, iMulKey, (intType,intType):intType,    TgtStr={"*"})
TgtOpr (MulKey, bAndKey, (realType,realType):realType, TgtStr={"&&"})
TgtOpr (NegKey, iNegKey, (intType):intType,            TgtStr={"-"})
TgtOpr (NegKey, bNotKey, (realType):realType,          TgtStr={"!"})
```

Implizite Typanpassung für OIL spezifizieren:

```
  COERCION cFloat (intType):realType;
```

Attributierung benutzt **Modulrollen `BinTgtOpr`, `ChkOpr`**:

```
SYMBOL Expression: Type, ReqType: DefTableKey;
SYMBOL BinOpr INHERITS BinTgtOpr, ChkOpr END;
RULE: Expression ::= Expression BinOpr Expression COMPUTE
  BinOpr.LType = Expression[2].Type;
  BinOpr.RType = Expression[3].Type;
  Expression[1].Type = BinOpr.ResType;
  Expression[2].ReqType = BinOpr.LTType;
  Expression[3].ReqType = BinOpr.RTType;
END;
```

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 607

Ziele:

Benutzung des Moduls Operator verstehen

im Vorlesungsteil:

- Quelloperator - Zieloperator
- Bezug zur konkreten und abstrakten Syntax
- Signaturen

nachlesen:

Dokumentation des Moduls Operator

Übungsaufgaben:

Arbeiten sie in der Core-Spezifikation die I-Aufgaben des Abschnitts "Operatoren in Ausdrücken" durch.

Verständnisfragen:

- Was erzeugen die Operatorbeschreibungen SrcOpr und TgtOpr?