Projekt C--

Univ.-Prof. Dr. Hanspeter Mössenböck, JKU Linz
Prof. DI Franz Matejka, HTL Braunau

Ziel dieses Maturaprojekts ist die Entwicklung einer Programmierumgebung für eine (leicht veränderte) Untermenge der Sprache C (genannt C--). Das Projekt soll im Rahmen von drei Diplomarbeiten an der HTL Braunau implementiert werden, wobei sich die Aufgaben wie folgt verteilen:

- **Compiler**. Mit Hilfe des Compilergenerators Coco/R (http://ssw.jku.at/Coco) soll ein Compiler entwickelt werden, der C-- Programme in eine Symbolliste und einen abstrakten Syntaxbaum (AST) übersetzt.
- **Interpreter**. Es soll ein Interpreter entwickelt werden, der abstrakte Syntaxbäume ausführt und als Laufzeitdatenstrukturen einen Prozeduren-Stack sowie einen Bereich für globale Daten verwaltet.
- **Umgebung**. Es soll eine Programmier- und Ausführungsumgebung entwickelt werden, die es erlaubt, C-- Programme zu editieren und interpretativ auszuführen. Die Ausführung soll visualisiert werden und dadurch das Verständnis der Programmierung in der Lehre fördern.

Die drei Diplomarbeiten können parallel entwickelt werden. Die Schnittstelle zwischen Compiler und Interpreter bilden die Symbolliste und der abstrakte Syntaxbaum. Die Schnittstelle zwischen Interpreter und Umgebung bilden die Laufzeitdatenstrukturen (Stack und globale Daten). Die Umgebung hat auch Schnittstellen zum Compiler, da sie diesen aufrufen und seine Fehlermeldungen anzeigen muss. Sie muss auch den Syntaxbaum und die Symbolliste kennen, da diese Datenstrukturen für die Visualisierung der Daten sowie für den SingleStep-Betrieb nötig sind.

Die Implementierungssprache des Projekts ist Java.

Ein C-- Programm besteht aus einer einzigen Datei. Als Datentypen gibt es *int*, *float*, *char* (Ascii) sowie (mehrdimensionale) Arrays und Structs. Als Anweisungen gibt es neben Zuweisungen und Prozeduraufrufen noch Verzweigungen und Schleifen. Prozeduren können void-Prozeduren oder Funktionen sein. Bei der Parameterübergabe wird Call by Value und Call by Reference unterstützt. Die Ein-/Ausgabe beschränkt sich auf das Lesen und Schreiben von Zeichen.

Das Projekt wird vom Institut für Systemsoftware der Johannes Kepler Universität Linz (http://ssw.jku.at) mitbetreut. Die drei Maturanten verbringen in der Anfangsphase zumindest zwei Wochen am Institut, um mit den Techniken des Compilerbaus und der Interpretation von Programmen vertraut zu werden. Die Hauptbetreuung und Beurteilung der Arbeiten erfolgt seitens der HTL Braunau.

Dieses Dokument ist derzeit lediglich eine Projektskizze, die die wesentlichen Eckpunkte der Aufgabenstellung festlegt. Eine detailliertere Projektbeschreibung wird noch ausgearbeitet bzw. soll im Rahmen des Projekts als Dokumentation entstehen.

Syntax

```
Program     =  { ConstDecl | VarDecl | StructDecl | ProcDecl }.

ConstDecl   =  "const" Type ident "=" (intCon | floatCon | charCon) ";".
VarDecl     =  Type ident {"," ident } ";".
StructDecl  =  "struct" ident "{" {VarDecl} "}".
ProcDecl    =  (Type | "void") ident "(" [FormPars] ")"
               ( "{" { ConstDecl | VarDecl | Statement } "}"
               | ";" "forward" ";"
               ).
FormPars    =  FormPar {"," FormPar}.
FormPar     =  ["ref"] Type ident.
Type        =  ident { "[" intCon "]" }.
Statement   =  Designator ("=" Expr | ActPars) ";"
            |  "if" "(" Condition ")" Statement ["else" Statement]
            |  "while" "(" Condition ")" Statement
            |  "print" "(" Expr ")" ";"
            |  "{" {Statement} "}"
            |  "return" Expr ";"
            |  ";".
ActPars     =  "(" [ ["ref"] Expr {"," ["ref"] Expr} ] ")".

Condition   =  CondTerm {"||" CondTerm}.
CondTerm    =  CondFact {"&&" CondFact}.
CondFact    =  Expr Relop Expr
            |  ["!"] "(" Condition ")".
Relop       =  "==" | "!=" | ">" | ">=" | "<" | "<=".

Expr        =  Term {Addop Term}.
Term        =  Factor {Mulop Factor}.
Factor      =  Designator [ActPars]
            |  intCon
            |  floatCon
            |  charCon
            |  "read" "(" ")"
            |  "-" Factor
            |  "(" Type ")" Factor
            |  "(" Expr ")".
Designator  =  ident {"." ident | "[" Expr "]"}.
Addop       =  "+" | "-".
Mulop       =  "*" | "/" | "%".
```

Lexikalische Struktur

Zeichenklassen:

```
letter      = 'a' .. 'z' | 'A' .. 'Z'.
digit       = '0' .. '9'.
whiteSpace  = ' ' | '\t' | '\r' | '\n'.
```

Terminalklassen:

```
ident       = letter {letter | digit | '_'}.
intCon      = digit {digit}.
floatCon    = digit {digit} '.' {digit} ['E' ['+'|'-'] digit {digit}].
charCon     = "'" char "'".  // einschließlich '\r', '\n', '\'' und '\\'
```

Kommentare: // bis Zeilenende
oder `/*` . . . `*/` können auch geschachtelt werden

Typ- und Deklarationsregeln

Typen

- Es gibt die drei vordefinierten primitiven Typen *float, int* und *char*. Ferner gibt es als strukturierte Typen Arrays und Structs.
- Zahlentypen dürfen in arithmetischen Ausdrücken gemischt werden. Der Ergebnistyp ist der größere der beiden Operandentypen, zumindest aber *int*.
- Vergleichsoperationen (==, !=, >, >=, <, <=) sowie logische Operationen (&&, || und !) führen zu booleschen Werten. Auf Sprachebene gibt es aber keinen expliziten Typ *boolean*. Logische Ausdrücke werden mit Kurzschlussauswertung berechnet, d.h.

  x && y => if x then y else false
  x || y => if x then true else y

Typgleichheit
Zwei Typen sind gleich, wenn sie durch denselben Typnamen ausgedrückt sind.

Zuweisungskompatibilität
Gleiche Typen sind zuweisungskompatibel, sofern sie nicht strukturiert sind. Ferner ist *int* zuweisungskompatibel mit *float*, und *char* ist zuweisungskompatibel mit *int* und *float*.

Deklarationsregeln
Jeder Name muss vor seiner Verwendung deklariert werden. Indirekt rekursive Prozeduren können durch Forward-Deklarationen aufgelöst werden. Jeder Name darf in seinem Scope (Programm, Prozedur oder Struct) nur ein einziges Mal deklariert werden (außer forward-deklarierte Prozedurnamen). Die Deklaration eines Namens in einer Prozedur oder in einem Struct verdeckt eine eventuelle Deklaration eines gleichen globalen Namens im Programm.

Kontextbedingungen

Program = {ConstDecl | VarDecl | StructDecl | ProcDecl}.

- Es muss eine parameterlose void-Prozedur namens *Main*() geben.

ConstDecl = "const" Type ident "=" (intCon | floatCon | charCon) ";".

- Der Typ von *inCon*, *floatCon* und *charCon* muss gleich sein wie der Typ von *Type*.

VarDecl = Type ident {"," ident } ";".

StructDecl = "struct" ident "{" {VarDecl} "}".

- Die Struct-Deklaration darf nicht leer sein.
- Der Name des Struct-Typs darf nicht als Typ eines der Felder verwendet werden.

ProcDecl = (Type | "void") ident "(" [FormPars] ")" "{" { ConstDecl | VarDecl | Statement } "}".

- Wenn die Prozedur eine Funktion ist, muss sie mit *return* verlassen werden.
- *Type* muss ein primitiver Typ sein.

ProcDecl = (Type | "void") ident "(" [FormPars] ")" ";" "forward" ";"

- Zu jeder Forward-Deklaration muss es eine volle Prozedurdeklaration geben.
- Eine Forward-Deklaration muss die gleiche Signatur haben wie die später folgende volle Prozedurdeklaration, d.h. der Prozedurname, der Funktionstyp, die Anzahl der Parameter, die Parametertypen und die Parameterarten (*ref* oder nicht *ref*) müssen übereinstimmen.

FormPars = FormPar {"," FormPar}.

FormPar = ["ref"] Type ident.

- *Type* muss ein primitiver Typ sein.

Type = ident { "[" intCon "]" }.

- *ident* muss einen Typ bezeichnen.
- *intCon* darf nicht 0 sein.

Statement = Designator "=" Expr ";".

- *Designator* muss eine Variable, ein Struct-Feld oder ein Arrayelement bezeichnen.
- Der Typ von *Expr* muss mit dem Typ von *Designator* zuweisungskompatibel sein.
- Der Typ von *Expr* muss ein primitiver Typ sein.

Statement = Designator ActPars ";".

- Designator muss eine *void*-Prozedur bezeichnen.

Statement = "if" "(" Condition ")" Statement ["else" Statement]
| "while" "(" Condition ")" Statement
| "{" {Statement} "}"
| ";".

Statement = "print" "(" Expr ")" ";".

- *Expr* muss vom Typ *char* sein.

Statement = "return" Expr ";".

- Die *return*-Anweisung darf nur in einer Funktionsprozedur vorkommen.
- Der Typ von *Expr* muss gleich sein wie der Typ der umschließenden Funktionsprozedur.

ActPars = "(" [["ref"] Expr {"," ["ref"] Expr}] ")".

- Die Anzahl der aktuellen und formalen Parameter muss übereinstimmen.
- Wenn ein formaler Parameter mit *ref* gekennzeichnet ist, muss der entspr. aktuelle Parameter
  - ebenfalls mit *ref* gekennzeichnet sein
  - eine Variable, ein Struct-Feld oder ein Arrayelement sein
  - den gleichen Typ haben wie der formale Parameter.

Condition = CondTerm "||" CondTerm.

CondTerm = CondFact "&&" CondFact.

CondFact = $Expr_1$ Relop $Expr_2$.

- Die Typen von $Expr_1$ und $Expr_2$ müssen gleich sein.
- Die Typen von $Expr_1$ und $Expr_2$ müssen primitive Typen sein.

CondFact = ["!"] "(" Condition ")".

Relop = "==" | "!=" | ">" | ">=" | "<" | "<=".

Expr = $Term_1$ Addop $Term_2$.

- Die Typen von $Term_1$ und $Term_2$ müssen primitive Typen sein.

Term = Factor$_1$ Mulop Factor$_2$.

- Die Typen von *Factor$_1$* und *Factor$_2$* müssen primitive Typen sein.

Factor = Designator | intCon | floatCon | charCon | "(" Expr ")".

Factor = Designator ActPars.

- *Designator* darf nicht strukturiert sein.
- *Designator* muss eine Methode bezeichnen.
- Der Typ der Methode darf nicht *void* sein.

Factor = "read" "(" ")".

- Der Typ des eingelesenen Werts ist *char*.
- Wenn nichts mehr gelesen werden kann, wird der Wert 0 geliefert.

Factor$_1$ = "-" Factor$_2$.

- Der Typ von *Factor$_2$* muss *int*, *float* oder *char* sein.
- Der Typ von *Factor$_1$* ist der Typ von *Factor$_2$*, aber zumindest *int*.

Factor$_1$ = "(" Type ")" Factor$_2$.

- Der Typ von *Type* muss ein primitiver Typ sein.
- Der Typ von *Factor$_2$* muss ein primitiver Typ sein.

Designator = ident.

Designator$_1$ = Designator$_2$ "." ident.

- *Designator2* muss eine Variable, ein Arrayelement oder ein Struct-Feld bezeichnen.
- Der Typ von *Designator2* muss ein Struct sein.
- *ident* muss ein Feld von *Designator$_2$* sein.

Designator$_1$ = Designator$_2$ "[" Expr "]".

- *Designator2* muss eine Variable, ein Arrayelement oder ein Struct-Feld bezeichnen.
- Der Typ von *Designator2* muss ein Array sein.
- Der Typ von *Expr* muss *int* sein.

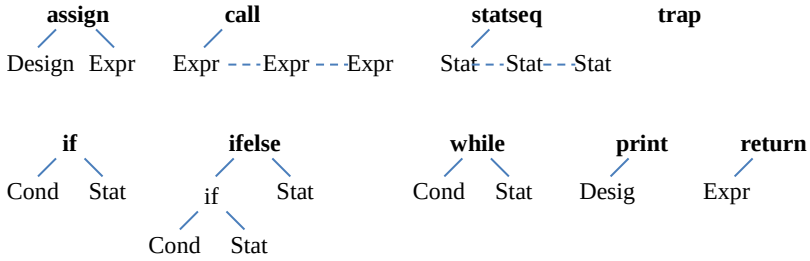
Addop = "+" | "-".

Mulop = "*" | "/" | "%".

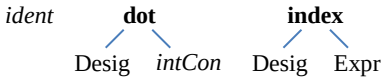
Syntaxbäume

```
class Node {
    int kind;               // assign, call, if, while, ..., plus, minus, ...
    int line;               // line number (only used for statements)
    Struct type;
    Node left, right, next;
    Obj obj;                // für ident
    int val;                // für intCon und charCon
    float fval;             // für floatCon

    // constructors
    Node (Obj obj) {...}    // ident (leaf)
    Node (int val) {...}    // intCon (leaf)
    Node (char val) {...}   // charCon (leaf)
    Node (float val) {...}  // floatCon (leaf)
    Node (int kind, Node left, Node right, Struct type) {...}
}
```

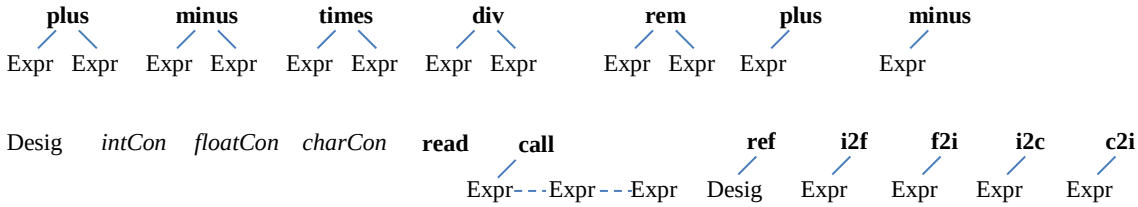
Statements



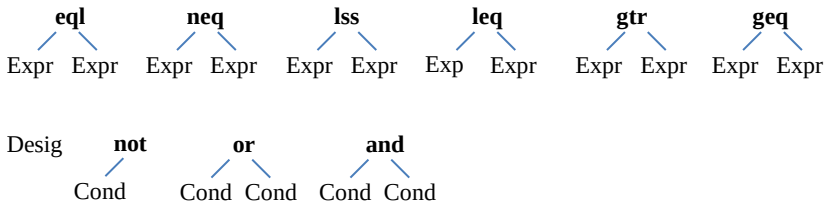
Designators



Expressions



Conditions



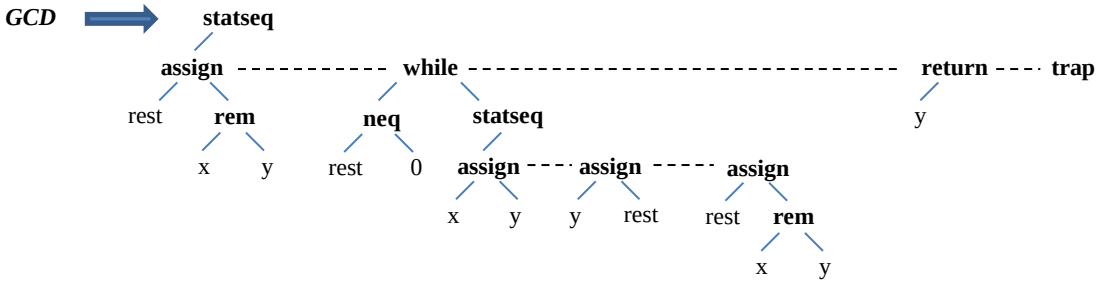
Procedures

Das Prozedurobjekt in der Symboltabelle enthält einen Verweis auf den AST der Prozedur.

Beispiel

```
int GCD (int x, int y) {
    int rest = x % y;
    while (rest != 0) {
        x = y; y = rest; rest = x % y;
    }
    return y;
}
```

Syntaxbaum dafür



Jede Funktion wird mit einer *trap*-Instruktion beendet, die einen Laufzeitfehler bewirkt, falls die Fuktion nicht mittels *return* verlassen wird.

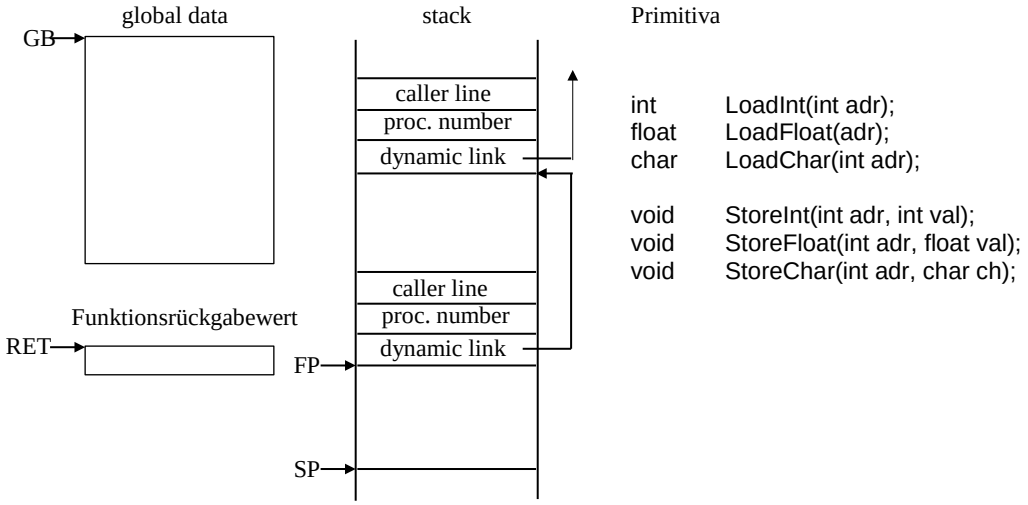
Skizzierung des Interpreters

```
void StatSeq (Node p) {
    for ( p = p.left; p != null; p = p.next) Statement(p);
}

void Statement (Node p) {
    switch (p.kind) {
        case assign:
            switch (p.right.type.kind) {
                case int_: StoreInt(Adr(p.left), IntExpr(p.right)); break;
                case float_: StoreFloat (Adr(p.left), FloatExpr(p.rigth)); break;
                ...
            }
            break;
        case if_:
            if (Condition(p.left)) Statement(p.right);
            break;
        case ifelse:
            if (Condition(p.left)) Statement(p.left.rigth); else Statement(p.right);
            break;
        case while_:
            while (Condition(p.left)) Statement(p.right);
            break;
        ...
    }
}

int IntExpr (Node p) {
    switch (p.kind) {
        case ident: return LoadInt(IdentAdr(p.obj));
        case intCon: return p.val;
        case dot: return LoadInt(Adr(p.left) + p.right.val);
        case index: return LoadInt(Adr(p.left) + IntExpr(p.right));
        case plus: return IntExpr(p.left) + IntExpr(p.right);
        ...
        case f2i: return (int) FloatExpr(p.left);
        case call: Call(p); return Load(IntRET);
    }
}

boolean Condition (Node p) {
    switch (p.kind) {
        case ident: return BoolLoad(IdentAdr(p.obj));
        case dot: return BoolLoad(Adr(p.left) + p.right.val);
        case index: return BoolLoad(Adr(p.left) + IntExpr(p.right));
        default:
            switch (p.left.type.kind) {
                case int_:
                    switch(p.kind) {
                        case eql: return IntExpr(p.left) == IntExpr(p.right);
                        case neq: return IntExpr(p.left) != IntExpr(p.right);
                        ...
                    }
                case float_:
                    switch (p.kind) {
                        case eql: return FloatExpr(p.left) == FloatExpr(p.right);
                        ...
                    }
            }
    }
}
```

```
void Call (Node p) {
    CreateFrame(p.obj);
    Obj formPar = p.obj.locals;
    for (Node actPar = p.left; actPar != null; formPar = formPar.next, actPar = actPar.next) {
        if (formPar.kind = RefPar) {
            StoreInt(FP + formPar.adr, Adr(actPar));
        } else { // value parameter
            switch (formPar.type.kind) {
                case int_: StoreInt(FP + formPar.adr, IntExpr(actPar)); break;
                case float_: StoreFloat(FP + formPar.adr, FloatExpr(actPar)); break;
                ...
            }
        }
    }
    StatSeq(p.obj.ast);
    DisposeFrame();
}

int Adr (Node p) {
    switch (p.kind) {
        case ident: return IdentAdr(p.obj);
        case dot: return Adr(p.left) + p.right.val;
        case index: return Adr(p.left) + IntExpr(p.right);
        default: error(); return FP;
    }
}

int IdentAdr (Obj obj) {
    if (obj.level == 0) return GB + obj.adr;
    else if (obj.kind == RefPar) return LoadInt(FP + obj.adr);
    else return FP + obj.adr;
}

void CreateFrame (Obj proc) {
    StoreInt(SP++, curLine);  // current line (caller line)
    StoreInt(SP++, proc.val); // procedure number
    StoreInt(SP++, FP);       // dynamic link
    FP = SP;
    SP += proc.varSize;
}

void DisposeFrame() {
    SP = FP - 12;
    FP = LoadInt(SP+8);
}
```

Laufzeitdatenstrukturen

Liegen am C-Heap und werden über Primitiva angesprochen.



Umgebung und Visualisierung

Dieser Teil wird in einem eigenen Dokument näher beschrieben. Hier sind einige erste Ideen dazu.

In einem *Editor-Fenster* kann der Benutzer ein Programm eingeben, das bei Verlassen des Editorfensters sofort in einen abstrakten Syntaxbaum, eine Symbolliste und Laufzeitdatenstrukturen für Variablen übersetzt wird. Der Editor soll wenn möglich Syntax-Coloring unterstützen sowie Syntax- und Semantikfehler geeignet kennzeichnen. Programme sollen auf Datei abgespeichert und wieder geladen werden können.

In einem *Shell-Fenster* kann der Benutzer globalen Variablen Werte zuweisen sowie Prozeduren aufrufen. Er kann damit Eingabewerte spezifizieren und Prozeduren einzeln ausprobieren. Bei der Ausführung der Prozeduren kann er SingleStep-Modus einschalten, wodurch nach jeder Anweisung angehalten wird und die Variablen wie in einem Debugger inspiziert werden können. Die Art der Visualisierung des SingleStep-Betriebs ist im Laufe der Arbeit zu spezifizieren. Eventuelle Programmausgaben erscheinen im Shell-Fenster. Eventuelle Programmeingaben werden unmittelbar nach der aufgerufenen Prozedur im Shell-Fenster eingegeben.

In einem *Daten-Fenster* sieht der Benutzer die globalen Variablen, die Prozeduraufrufkette sowie die lokalen Variablen der aktiven Prozeduren. Die Art der Visualisierung ist im Laufe der Arbeit zu erarbeiten.