Praktikum Compilerbau

Wintersemester 2007/08

Martin Hofmann, Andreas Abel, Hans-Wolfgang Loidl

Einführung

- Organisatorisches
- Aufgaben und Aufbau eines Compilers
- Überblick über das Praktikum
- Interpretation geradliniger (Straightline) Programme

Organisatorisches

- Das Praktikum richtet sich nach dem Buch *Modern Compiler Implementation in Java* von Andrew W Appel, CUP, 2005, 2. Aufl.
- Es wird ein Compiler für eine Teilmenge von Java: MiniJava entwickelt.
- Jede Woche wird ein Kapitel durchgenommen; ca. 30% VL und 70% Programmierung im Beisein der Dozenten.
- Die beaufsichtigte Programmierzeit wird i.A. nicht ausreichen; Sie müssen noch ca. 4h/Woche für selbstständiges Programmieren veranschlagen.
- Die Programmieraufgaben werden in Gruppen à zwei Teilnehmer bearbeitet (Extreme Programming).
- Scheinvergabe aufgrund erfolgreicher Abnahme des Programmierprojekts durch die Dozenten. Die Abnahme wird mündliche Fragen zum in der VL vermittelten Stoff enthalten.

Struktur des Kurses

Struktur des Kurses

Do	18.10.	Einführung
Do	25.10.	Lexikalische Analyse und Parsing
Do	8.11.	Abstrakte Syntax
Do	15.11.	Semantische Analyse
Do	22.11.	Activation records
Do	29.11.	Zwischensprachen
Do	6.12.	Basisblöcke
Do	13.12.	Speicherverwaltung
Do	20.12.	Instruktionsauswahl
Do	10.1.	Aktivitätsanalyse (liveness analysis)
Do	17.1.	Funktionale Sprachen
Do	24.1.	Registerverteilung
Do	31.1.	Objekt-orientierte Sprachen
Do	7.2.	Optimierungen

Aufgaben eines Compilers

- Übersetzt Quellcode (in Form von ASCII Dateien) in Maschinensprache (“Assembler”)
- Lexikalische und Syntaxanalyse
- Semantische Analyse (Typüberprüfung und Sichtbarkeitsbereiche)
- Übersetzung in Zwischencode: keine lokalen Variablen, Sprunganweisungen und Funktionsaufrufe als einzige Kontrollstruktur
- Erzeugung von Maschineninstruktionen (architekturabhängig)
- Registerzuweisung
- Ausgabe in Binärformat

Zu verschiedenen Zeitpunkten können Optimierungen vorgenommen werden.

Geradlinige (Straightline) Programme

- Bestehen aus Zuweisungen, arithmetischen Ausdrücken, mehrstelligen Druckanweisungen.
- Beispiel:

  ```
  a := 5+3; b := (print(a, a-1), 10*a); print(b)
  ```

  Ausgabe:

  ```
  8 7
  80
  ```

  BNF Grammatik:

$$
\begin{array}{lcl}
\mathit{Stm} & ::= & \mathit{Stm}\,\texttt{;}\,\mathit{Stm} \mid \mathit{ident}\,\texttt{:=}\,\mathit{Exp} \mid \texttt{print(}\,\mathit{ExpList}\,\texttt{)} \\
\mathit{Exp} & ::= & \mathit{ident} \mid \mathit{num} \mid \texttt{(}\,\mathit{Stm}\,\texttt{,}\,\mathit{Exp}\,\texttt{)} \mid \mathit{Exp}\ \mathit{binop}\ \mathit{Exp} \\
\mathit{ExpList} & ::= & \mathit{Exp} \mid \mathit{Exp}\,\texttt{,}\,\mathit{ExpList}
\end{array}
$$

Abstrakte Syntax in Java

```
abstract class Stm {}

class CompoundStm extends Stm {
   Stm stm1, stm2;
   CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}
}

class AssignStm extends Stm {
   String id; Exp exp;
   AssignStm(String i, Exp e) {id=i; exp=e;}
}

class PrintStm extends Stm {
   ExpList exps;
   PrintStm(ExpList e) {exps=e;}
}
```

Abstrakte Syntax in Java

```
abstract class Exp {}

class IdExp extends Exp {
   String id;
   IdExp(String i) {id=i;}
}

class NumExp extends Exp {
   int num;
   NumExp(int n) {num=n;}
}

class OpExp extends Exp {
   Exp left, right; int oper;
   final static int Plus=1,Minus=2,Times=3,Div=4;
   OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}
}
```

Abstrakte Syntax in Java

```
class EseqExp extends Exp {
   Stm stm; Exp exp;
   EseqExp(Stm s, Exp e) {stm=s; exp=e;}
}

abstract class ExpList {}

class PairExpList extends ExpList {
   Exp head; ExpList tail;
   public PairExpList(Exp h, ExpList t) {head=h; tail=t;}
}

class LastExpList extends ExpList {
   Exp head;
   public LastExpList(Exp h) {head=h;}
}
```

Abstrakte Syntax in Haskell

```
data Program = Prog { body :: [Statement] }

data Statement = CompoundStatement { stm1 :: Statement,
                                     stm2 :: Statement }
               | AssignStatement   { id :: String,
                                     exp :: Expression }
               | PrintStatement    { exps :: [Expression] }

data Expression = IdExpression  { id :: String }
          | NumExpression { num :: Integer }
          | OpExpression  { left :: Expression, op :: Binop,
                                   right :: Expression }
          | EsepExpression{ stm :: [Statement], exp :: Expression

data Binop =  PLUS |  MINUS  | TIMES  | DIVIDE
```

Beispiel-Programm

```
new CompoundStm(new AssignStm("a",new OpExp(new NumExp(5), OpExp.Plus,
                                             new NumExp(3))),
 new CompoundStm(new AssignStm("b",
     new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
                         new LastExpList(new OpExp(new IdExp("a"), OpExp.Min
                                                 new NumExp(1))))),
              new OpExp(new NumExp(10), OpExp.Times, new IdExp("a")))),
  new PrintStm(new LastExpList(new IdExp("b"))));
}
```

Programmieraufgabe: Straightline Interpreter

- Implementieren einer Klasse `Table`, die Zuordnungen Bezeichner $\mapsto$ Werte, d.h. Umgebungen, modelliert.

  Umgebungen werden funktional implementiert, z.B. als Listen oder und können nicht imperativ verändert werden.

- Bereitstellen und Implementieren einer Methode

  `Table interp(Table t)`

  in der Klasse `Stm`.

  Der Aufruf `Table tneu = s.interp(t)` soll das Programm `s` in der Umgebung `t` auswerten und die daraus resultierende neue Umgebung in `tneu` abspeichern.

  Hierzu deklarieren Sie diese Methode in `Stm` als abstrakt und implementieren sie dann jeweils in den konkreten Unterklassen. Sie benötigen entsprechende Hilfsmethoden in der Klasse `Exp`.

Empfohlene Programmiertechniken

Der Interpreter dient nur als Aufwärmübung. Grundkenntnisse in Java (oder der gewählten Implementationssprache) sind Voraussetzung.

Besonders nützlich für den eigentlichen Compiler sind:

- **Datenstrukturen:** Modellierung einer Umgebung (environment) als eine (rein funktionale) Tabelle (table).
- **Tools:** Modellierung der abstrakten Syntax mittels einer Klassen-Hierarchie. Diese kann automatisch mittels `dataGen` erzeugt werden.
- **Programmiertechnik:** Iteration über diese Datenstruktur mittels eines *Visitor Pattern*.

Visitor Pattern

Eine generische Struktur zum Ausführen von Operationen auf allen Elementen einer komplexen Datenstruktur.

- Fördert eine *funktionsorientierte Sichtweise*: der Code für eine Operation auf einer gesamten Datenstruktur wird in einem Modul zusammengefasst (z.B. Typcheck auf der abstrakten Syntax)
- Es werden 2 Klassen-Hierarchien aufgebaut: eine für die Objekt-Klassen (Daten) und eine für die Visitor-Operationen (Code).
- Die Objekt-Klassen verwenden eine generische *accept* Methode, um Code des Visitors (Parameter) auszuführen.
- Geeignet für Anwendung mit fixer Daten-Struktur (z.B. abstrakter Syntax) und verschiedenen Operationen die darauf arbeiten.

Visitor Pattern (cont'd)

Objekt-Klasse:

```
class AssignStatement extends Statement {
    public String id;    public Expression exp;
    public AssignStatement (String id, Expression exp) { this.id = id;
                                                         this.exp = exp; }
    public <A> A accept (StatementVisitor<A> v) {
        return v.visit (this);
    }
}
```

Visitor-Klasse:

```
class PPVisStm implements StatementVisitor<String> {
   ...
   public String visit(AssignStatement s) {
     return s.id + " = " + s.exp.accept(new PPVisExpr());
   }
   ...
}
```

Tools

Folgende Tools sind auf den Maschinen das CIP Pools installiert:

- `javac`: Java compiler (v1.6)
- `dataGen`: für das automatische Erzeugen einer Klassen-Hierarchie aus einer Haskell-artigen Datenstruktur
- `jflex`: Lexer Generator (ab PK 2)
- `javaCUP`: Parser Generator (ab PK 2)
- `muHwI'`: Interpreter für die Zwischensprache (ab PK 6)

Links zu den verwendeten Tools unter

`http://www.tcs.informatik.uni-muenchen.de/lehre/WS07-08/Compiler/software.`

Lexikalische Analyse

- Was ist lexikalissche Analyse
- Durchführung mit endlichen Automaten
- Bedienung von Werkzeugen: JLex, JFlex.
- Praktikum: Lexer für MiniJava mit JLex.

Was ist lexikalische Analyse?

- Erste Phase der Kompilierung
- Entfernt Leerzeichen, Einrückungen und Kommentare
- Die Eingabe wird in eine Folge von *Token* umgewandelt.
- Die *Tokens* spielen die Rolle von Terminalsymbolen für die Grammatik der Sprache.

Was sind Tokens?

Beispiele:

`foo` (ID), `73` (INT), `66.1` (REAL), `if` (IF), `!=` (NEQ),
`(` (LPAREN), `)` (RPAREN)

Keine Beispiele:

`/* huh? */` (Kommentar),
`#define NUM 5` (Präprozessordirektive),
`NUM` (Makro)

Beispiel für die lexikalische Analyse

```
void match0(char *s) /* find a zero */
{if (!strncmp (s, "0.0", 3))
    return 0.;
}
```

VOID ID(`match0`) LPAREN CHAR STAR

ID(`s`) RPAREN LBRACE IF LPAREN

BANG ID(`strncmp`) LPAREN ID(`s`)

COMMA STRING(`0.0`) COMMA INT(`3`)

RPAREN RPAREN RETURN REAL(`0.0`)

SEMI RBRACE EOF

Reguläre Ausdrücke und NEA/DEA

`(`	$\rightarrow$	LPAREN
$digit\,digit^*$	$\rightarrow$	INT(ConvertToInt(yytext()))
`print`	$\rightarrow$	PRINT
$letter(letter + digit + \{\texttt{\_}\})^*$	$\rightarrow$	ID(yytext())
...		

wobei $digit = \{\texttt{0}, \ldots, \texttt{9}\}$ und $letter = \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}\}$.

Motto: Tokens werden durch reguläre Ausdrücke spezifiziert; Verarbeitung konkreter Eingaben mit DEA. Automatische Lexergeneratoren wie (lex, flex, JLex, JFlex, Ocamllex) wandeln Spezifikation in Programm um.

Auflösung von Mehrdeutigkeiten

I.a. gibt es mehrere Möglichkeiten eine Folge in Tokens zu zerlegen.
Lexergeneratoren verwenden die folgenden zwei Regeln:

Längste Übereinstimmung: Das längste Präfix, das zu irgendeinem der regulären Ausdrücke passt, wird das nächste Token.

Regelpriorität: Wenn das nicht hilft, kommt die weiter oben stehende Regel zum Zug. Die Reihenfolge der Regeln spielt also eine Rolle.

`print0`	$\rightarrow$	ID(`print0`) *nicht* PRINT INT(`0`)
`print`	$\rightarrow$	PRINT *nicht* ID(`print`)

Implementierung

Man baut mit Teilmengenkonstruktion einen Automaten, dessen Endzustände den einzelnen Regeln entsprechen. “Regelpriorität” entscheidet Konflikte bei der Regelzuordnung.

Zur Implementierung von “längste Übereinstimmung” merkt man sich die Eingabeposition, bei der das letzte Mal ein Endzustand erreicht wurde und puffert jeweils die darauffolgenden Zeichen, bis wieder ein Endzustand erreicht wird. Kommt man dagegen in eine Sackgasse, so bestimmt der letzte erreichte Endzustand die Regel und man arbeitet zunächst mit den gepufferten Zeichen weiter.

Lexergeneratoren

Ein Lexergenerator erzeugt aus einer `.lex`-Datei einen Lexer in Form einer Java-Klasse (bzw. C, ML Modul), das eine Methode (Funktion) `nextToken()` enthält.

Jeder Aufruf von `nextToken()` liefert das "Ergebnis" zurück, welches zum nächsten verarbeiteten Teilwortes der Eingabe gehört.

Normalerweise besteht dieses "Ergebnis" aus dem Namen des Tokens und seinem Wert; es kann aber auch irgendetwas anderes sein.

Die `.lex` Datei enthält Regeln der Form

$$regexp \rightarrow \{code\}$$

wobei *code* Java (C, ML)-code ist, der das "Ergebnis" berechnet. Dieses Codefragment kann sich auf den (durch *regexp*) verarbeiteten Text durch spezielle Funktionen und Variablen beziehen, z.B.: `yytext()` und `yypos`. Siehe Beispiel + Doku.

Zustände

- In der `.lex` Datei können auch Zustände angegeben werden:
- Regeln können mit Zuständen beschriftet werden; sie dürfen dann nur in diesem Zustand angewandt werden.
- Im *code* Abschnitt kann durch spezielle Befehle der Zustand gewechselt werden, z.B. `yybegin()`.
- Man verwendet das um geschachtelte Kommentare und Stringliterale zu verarbeiten:
  - `/*` führt in einen “Kommentarzustand”.
  - Kommt `/*` im Kommentarzustand vor, so inkrementiere einen Tiefenzähler.
  - `*/` dekrementiert den Tiefenzähler oder führt wieder in den “Normalzustand” zurück.
  - Erzeuge Fehlermeldung, wenn `/*` oder `*/` unerwartet vorkommen.
  - Ebenso führt `"` (Anführungszeichen) zu einem “Stringzustand”...

Ihre Aufgabe

Schreiben eines Lexers (mit JLex/JFlex) für MiniJava.

- Fehlerausgabe mit Zeile/Spalte (vorgefertigte Klasse auf Homepage).
- Geschachtelte Kommentare, sowie Kommentare bis zum Zeilenende (`//`).
- Dateiende (`EOF`) korrekt behandeln.

Syntaxanalyse

- Zweite Phase der Kompilierung
- Konversion einer Folge von Tokens in einen Syntaxbaum (abstrakte Syntax, AST (*abstract syntax tree*) anhand einer Grammatik.
- Laufzeit linear in der Eingabegröße: Einschränkung auf spezielle Grammatiken (LL(1), LR(1), LALR(1)) für die effiziente Analysealgorithmen verfügbar sind.
- Parsergeneratoren (yacc, bison, ML-yacc, JavaCUP, JavaCC, ANTLR) erzeugen Parser (Syntaxanalysatoren) aus formaler Grammatik. JavaCC, ANTLR verwenden LL(1), die anderen LALR(1).

LL(1)-Grammatiken

Betrachte folgende Grammatik:

1. $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
2. $S \rightarrow \text{begin } S\ L$
3. $S \rightarrow \text{print } E$
4. $L \rightarrow \text{end}$
5. $L \rightarrow ;\ S\ L$
6. $E \rightarrow \text{num} = \text{num}$

Ein Parser für diese Grammatik kann mit der Methode des rekursiven Abstiegs (*recursive descent*) gewonnen werden:

Für jedes Nichtterminalsymbol gibt es eine Funktion, die gegen dieses analysiert:

In C

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();eat(ELSE); S(); b
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();}}
void L(void) {switch(tok) {
        case END:  eat(END); break;
        case SEMI: eat(SEMI); S(); L(); break;
        default:   error();}}
void E(void) { eat(NUM); eat(EQ); eat(NUM); }
```

Manchmal funktioniert das nicht:

$$S \to E\,\$ \qquad E \to E + T \qquad T \to T * F \qquad F \to \mathrm{id}$$
$$E \to E - T \qquad T \to T \,/\, F \qquad F \to \mathrm{num}$$
$$E \to T \qquad T \to F \qquad F \to (\ E\ )$$

```
void S(void) {  E(); eat(EOF);  }
void E(void) {switch(tok) {
        case ?: E(); eat(PLUS); T(); break;
        case ?: E(); eat(MINUS); T(); break;
        case ?: T(); break;
        default:   error(); }}
void T(void) {switch(tok) {
        case ?: T(); eat(TIMES); F(); break;
        case ?: T(); eat(DIV); F(); break;
        case ?: F(); break;
        default:   error();  }}
```

LL(1)-Parsing

- Eine Grammatik $G = (\Sigma, V, P, S)$ heißt LL(1), wenn ein Parse-Algorithmus basierend auf dem Prinzip des rekursiven Abstiegs für sie existiert.
- Nicht jede Grammatik ist LL(1), etwa die aus dem zweiten Beispiel ist es nicht. Problem: der Parser muss schon anhand des ersten Tokens (und der erwarteten linken Seite) in der Lage sein, zu entscheiden, welche Produktion zu wählen ist.

  Manchmal kann man eine Grammatik in die LL(1)-Form bringen. Man kann auch mehr als ein Zeichen weit vorausschauen: LL(k).
- Wie erzeugt man einen LL(1)-Parser automatisch aus der Grammatik?

Die First- und Follow-Mengen

Seien X, Y, Z Nichtterminalsymbole, α, β, γ Folgen von Terminal- und Nichtterminalsymbolen.

- $\mathrm{nullable}(\gamma)$ bedeute, dass $\gamma \rightarrow^* \epsilon$.
- FIRST(γ) ist die Menge aller Terminalsymbole, die als Anfänge von aus γ abgeleiteten Wörtern auftreten.
  (FIRST$(\gamma) = \{a \mid \exists w.\gamma \rightarrow^* aw\}$.
- FOLLOW(X) ist die Menge der Terminalsymbole, die unmittelbar auf X folgen können. (FOLLOW$(X) = \{a \mid \exists \alpha, \beta.S \rightarrow^* \alpha X a \beta\}$).

Berechnung der First- und Follow-Mengen

Die First- (für rechte Seiten von Produktionen) und Follow-Mengen, sowie das Nullable-Prädikat, lassen sich mithilfe der folgenden Regeln iterativ berechnen:

- $\text{FIRST}(\epsilon) = \emptyset, \text{FIRST}(a\gamma) = \{a\}, \text{FIRST}(X\gamma) =$
  if $\text{nullable}(X)$ **then** $\text{FIRST}(X) \cup \text{FIRST}(\gamma)$ **else** $\text{FIRST}(X)$.
- $\text{nullable}(\epsilon) = \mathit{true}$, $\text{nullable}(a\gamma) =$ *false*,
  $\text{nullable}(X\gamma) = \text{nullable}(X)) \wedge \text{nullable}(\gamma)$.

Für jede Produktion $X \to \gamma$ gilt:

- Wenn $\text{nullable}(\gamma)$, dann auch $\text{nullable}(X)$.
- $\text{FIRST}(\gamma) \subseteq \text{FIRST}(X)$.
- Wenn $\gamma = \alpha Y \beta$ und $\text{nullable}(\beta)$, dann
  $\text{FOLLOW}(Y) \subseteq \text{FOLLOW}(X)$.
- Wenn $\gamma = \alpha Y \beta Z \delta$ und $\text{nullable}(\beta)$, dann
  $\text{FIRST}(Z) \subseteq \text{FOLLOW}(Y)$.

Konstruktion des Parsers

Man tabuliert durch sukzessive Anwendung der Regeln die First- und Follow-Mengen, sowie das Nullable-Prädikat für alle Nichtterminalsymbole. Erweiterung auf Phrasen erfolgt “on-the-fly”.

Soll die Eingabe gegen X geparst werden (also in der Funktion X) und ist das nächste Token a, so kommt die Produktion $X \to \gamma$ infrage, wenn

- $a \in \text{FIRST}(\gamma)$ oder
- $\text{nullable}(\gamma)$ und $a \in \text{FOLLOW}(X)$.

Kommen aufgrund dieser Regeln mehrere Produktionen infrage, so ist die Grammatik nicht LL(1). Ansonsten kann anhand der Regeln der Parser konstruiert werden.

Von LL(1) zu LL(k)

Der LL(1)-Parser entscheidet aufgrund des nächsten Tokens und der erwarteten linken Seite, welche Produktion zu wählen ist. Bei LL(k) bezieht man in diese Entscheidung die k nächsten Token mit ein.

Dementsprechend bestehen die First- und Follow-Mengen aus Wörtern der Länge k und werden dadurch recht groß.

Durch Einschränkung der k-weiten Vorausschau auf bestimmte benutzerspezifizierte Stellen, lässt sich der Aufwand beherrschbar halten. Das passiert bei den Parsergeneratoren ANTLR und JavaCC.

Zusammenfassung LL(1)-Syntaxanalyse

- Das nächste Eingabesymbol und die erwartete linke Seite entscheiden, welche Produktion (für das nächste Eingabesymbol und die folgenden) zu wählen ist.
- Ist diese Entscheidung in eindeutiger Weise möglich, so liegt eine LL(1) Grammatik vor und die Entscheidung lässt sich mithilfe der First- und Follow-Mengen automatisieren.
- Der große Vorteil des LL(1)-Parsing ist die leichte Implementierbarkeit: ist kein Parsergenerator verfügbar (etwa bei Skriptsprachen), so kann ein rekursiver LL(1)-Parser leicht von Hand geschrieben werden. Beachte, die First- und Follow-Mengen müssen ja nur einmal pro Grammatik berechnet werden, was wiederum von Hand geschehen kann.

LR-Syntaxanalyse

- LL(1)-Syntaxanalyse hat den Nachteil, dass allein aufgrund des nächsten Tokens die zu verwendende Produktion ermittelt werden muss.
- Bei LR(1)-Syntaxanalyse braucht diese Entscheidung erst gefällt werden, wenn die gesamte rechte Seite einer Produktion (plus ein Zeichen Vorausschau) gelesen wurde.
- Der Parser arbeitet mit einem Keller, dessen Inhalt aus Grammatiksymbolen besteht und (das ist die Invariante) stets den bisher gelesenen Symbolen entspricht (sich also zu denen reduzieren lässt mithilfe der Grammatikproduktionen).
- Enthält der Keller nur das Startsymbol und wurde die gesamte Eingabe eingelesen, so ist die Analyse zuende.

Aktionen des LR(1)-Parsers

Der LR(1)-Parser kann zu jedem Zeitpunkt eine der folgenden beiden Aktionen durchführen:

- Ein weiteres Eingabesymbol lesen und auf den Keller legen (*Shift*-Aktion)
- Eine Produktion auf die oberen Kellersymbole rückwärts anwenden, also den Kellerinhalt $\sigma\gamma$ durch σX ersetzen, falls eine Produktion $X \to \gamma$ vorhanden ist. (*Reduce*-Aktion)

Eine Grammatik ist per definitionem LR(1), wenn die Entscheidung, welche Aktion durchzuführen ist, allein aufgrund der bisher gelesenen Eingabe, sowie dem nächsten Symbol, getroffen werden kann.

Wann soll man mit $X \to \gamma$ reduzieren?

A: Wenn γ oben auf dem Keller liegt ($\gamma \in (\Sigma \cup V)^*$) und außerdem angesichts der bisher gelesenen Eingabe und des nächsten Symbols keine Sackgasse vorhersehbar ist:

$$L^{\text{Reduce by } X \to \gamma \text{ when } a} = \{\sigma\gamma \mid \exists w \in \Sigma^*. S \to_{\mathbf{rm}} \sigma X a w\}$$

Hier bezeichnet $\to_{\mathbf{rm}}$ die Rechtsableitbarkeit: die Existenz einer Ableitung, in der immer das am weitesten rechts stehende Nichtterminalsymbol ersetzt wird.

Wann soll man das nächste Symbol a einlesen?

A: Wenn es eine Möglichkeit gibt, angesichts des aktuellen Kellerinhalts später eine Produktion anzuwenden.

$$L^{\text{Shift } a} = \{\sigma\alpha \mid \exists w \in \Sigma^*.\exists \text{Produktion} X \to \alpha a \beta.S \to_{\mathbf{rm}} \sigma X w\}$$

Für festes nächstes Eingabesymbol müssen diese Shift- und Reduce-Mengen disjunkt sein, sonst kann eben die erforderliche Entscheidung nicht eindeutig getroffen werden und es liegt keine LR(1)-Grammatik vor.

Das Wunder der LR(1)-Syntaxanalyse

Die Mengen $L^{\text{Reduce by } X \to \gamma \text{ when } a}$ und $L^{\text{Shift } a}$ sind regulär.

Es existiert also ein endlicher Automat, der nach Lesen des Kellerinhaltes anhand des nächsten Eingabesymbols entscheiden kann, ob ein weiteres Symbol gelesen werden soll, oder die obersten Kellersymbole mit einer Produktion reduziert werden sollen und wenn ja mit welcher.

Beispiel

(1)E →E + T (2)E→T

(3)T →T * F (4)T→F

(5)F →id (6)F→(E)

Beispiel

	0	1	2	3	4	5	6	7	8	9	10	11
0												
1	E											
2	T				T							
3	F				F		F					
4	(				(		(	(				
5	id				id		id	id				
6		+							+			
7			*							*		
8					E							
9							T					
10								F				
11									)			
	S	S	R2*	R4	S	R5	S	S	S	R1*	R3	R6

S = Shift. Rx = Reduce mit (x). Rx* = Shift, falls nächstes Symbol `*`, sonst Reduce mit (x).

Optimierung: LR(1)-Tabelle

- Annotiere Kellereinträge mit erreichtem Zustand (in der Praxis lässt man die ursprünglichen Kellersymbole ganz weg und arbeitet mit Automatenzuständen als Kelleralphabet).
- Konstruiere Tafel, deren Zeilen mit Zuständen und deren Spalten mit Grammatiksymbolen indiziert sind. Die Einträge enthalten eine der folgenden vier *Aktionen*:

Shift(n)	"Shift" und gehe in Zustand n;
Goto(n)	Gehe in Zustand n;
Reduce(k)	"Reduce" mit Regel k;
Accept	Akzeptiere.

  Leere Einträge bedeuten Syntaxfehler.

Der LR(1)-Algorithmus

- Ermittle Aktion aus der Tabelle anhand des obersten Kellerzustands und des nächsten Symbols.
- Ist die Aktion...

  Shift(n): Lies ein Zeichen weiter; lege Zustand n auf den Keller.

  Reduce(k):
  - Entferne soviele Symbole vom Keller, wie die rechte Seite von Produktion k lang ist,
  - Sei X die linke Seite der Produktion k:
  - Finde in Tabelle unter dem nunmehr oben liegenden Zustand und X eine Aktion "**Goto(n)**";
  - Lege n auf den Keller.

  Accept: Ende der Analyse, akzeptiere die Eingabe.

  Error: Ende der Analyse, erzeuge Fehlermeldung.

Beispiel

(1)E →E + T (2)E→T (3)T →T * F (4)T→F (5)F →id (6)F→(E) (7)S→E$

	0	1	2	3	4	5	6	7	8	9	10	11
0												
1	E											
2	T				T							
3	F				F		F					
4	(				(		(	(				
5	id				id		id	id				
6		+							+			
7			*							*		
8					E							
9							T					
10								F				
11									)			
	S	S	R2*	R4	S	R5	S	S	S	R1*	R3	R6

	id	(	)	+	*
0	s5	s4			
1				s6	
2	r2	r2	r2	r2	s7
3	r4	r4	r4	r4	r4
4	s5	s4			
5	r5	r5	r5	r5	r5
6	s5	s4			
7	s5	s4			
8			s11	s6	
9	r1	r1	r1	r1	s7
10	r3	r3	r3	r3	r3
11	r6	r6	r6	r6	r6

Konstruktion des Automaten (und der Tabelle)

Der Automat wird zunächst nichtdeterministisch konzipiert.

Die Zustände haben die Form $(X \to \alpha.\beta, a)$ wobei $X \to \alpha\beta$ eine Produktion sein muss und a ein Terminalsymbol ist.

Solch ein Zustand heißt “LR(1)-Item”.

Die Sprache, die zum Erreichen des Items $(X \to \alpha.\beta, a)$ führen soll, ist:

$$L(X \to \alpha.\beta, a) = \{\gamma\alpha \mid \exists w \in \Sigma^*. S \to_{\mathbf{rm}} \gamma X a w\}$$

also gilt insbesondere:

$$L^{\text{Reduce by } X \to \gamma \text{ when } a} = L(X \to \gamma., a)$$
$$L^{\text{Shift } a} = \bigcup_{X \to \alpha a \beta \text{Produktion}} L(X \to \alpha a \beta., a)$$

Jetzt muss man nur noch die Transitionen so bestimmen, dass tatsächlich diese Sprachen “erkannt” werden:

Transitionen des Automaten

Erinnerung:

$$L(X \to \alpha.\beta, a) = \{\gamma\alpha \mid \exists w \in \Sigma^*.S \to_{\mathbf{rm}} \gamma X a w\}$$

- $(X \to \alpha.s\beta, a) \longrightarrow^{s} (X \to \alpha s.\beta, a)$, $s \in \Sigma \cup V$,
- $(X \to \alpha.Y\beta, a) \longrightarrow^{\epsilon} (Y \to .\gamma, b)$, falls $Y \to \gamma$ und $b \in \text{FIRST}(\beta a)$

Startzustände: $(S \to .\gamma\$, ?)$, wobei $S \to \gamma\$$ die einzige Produktion ist, die S involviert.

Man zeigt durch Induktion, dass der so definierte Automat tatsächlich die gewünschten Sprachen erkennt.

Aktionen

Nunmehr determinisiert man den Automaten und erhält so Mengen von LR(1)-Items als Zustände.

Enthält ein Zustand das Item $(X \to \gamma., a)$ und ist das nächste Symbol a, so reduziert man mit $X \to \gamma$.

Enthält ein Zustand das Item $(X \to \alpha.a\beta, c)$ und ist das nächste Symbol a, so wird ge-Shiftet.

Gibt es mehrere Möglichkeiten, so liegt ein Shift/Reduce, beziehungsweise ein Reduce/Reduce-Konflikt vor und die Grammatik ist nicht LR(1).

LALR(1) und SLR

LR(1)-Tabellen sind recht groß (mehrere tausend Zustände für typische Programmiersprache).

LALR(1) ist eine heuristische Optimierung, bei der Zustände, die sich nur durch die Vorausschau-Symbole unterscheiden, identifiziert werden. Eine Grammatik heißt LALR(1), wenn nach diesem Prozess keine Konflikte entstehen.

Bei SLR wird auf Vorausschau-Symbole in den Items verzichtet, stattdessen verwendet man FOLLOW-Mengen, um Konflikte aufzulösen.

Zusammenfassung LR-Syntaxanalyse

- LR-Parser haben einen Keller von Grammatiksymbolen, der der bisher gelesenen Eingabe entspricht.
- Sie legen entweder das nächste Eingabesymbol auf den Keller (“Shift”) oder wenden auf die obersten Kellersymbole eine Produktion rückwärts an (“Reduce”).
- Die Shift/Reduce-Entscheidung richtet sich nach der bereits gelesenen Eingabe und dem nächsten Symbol. Sie kann durch einen endlichen Automaten vorgenommen werden, der auf dem Keller arbeitet.
- Um Platz zu sparen, werden nur die Automatenzustände auf dem Keller gehalten und jeweils aktualisiert.
- LR-Parser sind allgemeiner und auch effizienter als LL-Parser.
- LALR(1) und SLR sind heuristische Optimierungen von LR(1)

Parsergeneratoren

- Parsergenerator: Grammatikspezifikation $\rightarrow$ Parser (als Quellcode).
- JavaCUP ist ein LALR(1)-Parsergenerator, der Java-Code generiert.
- Grammatikspezifikationen sind in die folgenden Abschnitte gegliedert:

  Benutzerdeklarationen (z.B. `package` statements, Hilfsfunktionen)

  %%

  Parserdeklarationen (z.B. Mengen von Terminal- / Nichtterminalsymbolen)

  %%

  Produktionen

- Die Produktionen haben die folgende Form (exemplarisch):

  `exp : exp PLUS exp` (*Semantische Aktion*)

  Die “semantische Aktion” wird ausgeführt, wenn die entsprechende Regel “feuert”.

Beispielgrammatik

Stm→*Stm*;*Stm*	(CompoundStm)		
Stm→`id :=`*Exp*	(AssignStm)	*ExpList*→*Exp* `,` *ExpList*	(PairExpLis
Stm→`print (`*ExpList*`)`	(PrintStm)	*ExpList*→*Exp*	(LastExpLis
Exp→`id`	(IdExp)	*BinOp*→`+`	(Plus
Exp→`num`	(NumExp)	*BinOp*→`-`	(Minus
Exp→*Exp BinOp Exp*	(OpExp)	*BinOp*→`*`	(Times
Exp→`(`*Stm* `,` *Exp*`)`	(EseqExp)	*BinOp*→`/`	(Div

Implementierung in CUP

```
package straight;
import java_cup.runtime.*;

action code  {: <here go helper functions for the actions> :}
parser code  {: <here go helper functions for the parse> :}
scan with    {: return lexer.nextToken(); :};

terminal String ID; terminal Integer INT;
terminal COMMA, SEMI, LPAREN, RPAREN, PLUS, MINUS,
         TIMES, DIVIDE, ASSIGN, PRINT;

non terminal Exp exp;
non terminal ExpList explist;
non terminal Stm prog;

precedence left SEMI;
precedence nonassoc ASSIGN;
```

Implementierung in CUP

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;

start with prog;

prog ::= stm {: :}

exp ::= exp TIMES exp {: :}
       |exp DIVIDE exp {: :}
       |exp PLUS exp {: :}
       |exp MINUS exp {: :}
       |INT {: :}| ID {: :}
       |LPAREN stm COMMA exp RPAREN {: :}
       |LPAREN exp RPAREN {: :}

explist ::= exp {: :}
           |exp COMMA explist {: :}
```

Implementierung in CUP

```
stm ::= stm SEMI stm {: :}
       |PRINT LPAREN explist RPAREN {: :}
       |ID ASSIGN exp {: :}
```

Präzedenzdirektiven

Die obige Grammatik ist mehrdeutig, also insbesondere nicht LR(1).

Präzedenzdirektiven werden hier zur Auflösung der Konflikte verwendet.

Die Direktive

```
%right SEMI COMMA
%left PLUS MINUS
%left TIMES DIV
```

besagt, dass `TIMES, DIV` stärker als`PLUS, MINUS` binden, welche wiederum stärker als `SEMI, COMMA`. `PLUS, MINUS, TIMES, DIV` assoziieren nach links; `SEMI, COMMA` nach rechts.

Also wird $s1; s2; s3, x + y + z * w$ zu $(s1; (s2; s3)), ((x + y) + (z * w))$.

Unäres Minus

Direktiven beziehen sich auf Tokens. Will man sie auf Produktionen beziehen, so führt man ein künstliches Token ein. Beispiel: durch

```
exp: MINUS exp %prec UMINUS
```

erbt diese Produktion die Präzedenz von `UMINUS`. So kann man $-e_1 + e_2$ als $(-e_1) + e_2$ statt $-(e_1 + e_2)$ lesen.

Konflikte

Die verbleibenden Konflikte werden als Shift/Reduce, bzw. Reduce/Reduce-Konflikte gemeldet.

Diese muss man sehr ernst nehmen; in den meisten Fällen deuten sie auf Fehler in der Grammatik hin.

Ein Parser wird auch bei Konflikten generiert; er wird im Zweifel immer Shiften und ansonsten weiter oben geschriebenen Regeln Priorität einräumen (wie beim Lexer).

Ein(ziger) Fall, bei dem solch ein Konflikt sinnvoll ist: “dangling else”.

Typüberprüfung

- Datenstruktur Symboltabelle: funktional und imperativ
- Typüberprüfung in mehreren Durchgängen

Beispiel: Gültigkeitsbereich

```
class C {
  int a; int b; int c;
  public void m () {
    System.out.println(a+c);
    int j = a+b;
    String a = "hello";
    System.out.println(a);
    System.out.println(j);
    System.out.println(b);
 }
}
```

Effiziente Implementierung der Symboltabelle

Datenstruktur für die Symboltabelle:

- imperativ, d.h. die aktuelle Tabelle wird laufend verändert: *hash table* der alte Einträge überdeckt, nicht überschreibt
- funktional, d.h. alte Tabellen bleiben weiterhin verfügbar: *binary search tree*

Realisierung von Gültigkeitsbereichen:

- imperativ: ein Stack von Symbolen wird als Hilfsstruktur verwendet; jedes neue Symbol wird auch auf diesen Stack gelegt; die Operationen `beginScope` und `endScope` legen einen Marker auf den Stack bzw entfernen alle Einträge bis zum letzten Marker
- funktional: da die alte Tabelle noch vorhanden ist, sind `beginScope` und `endScope` nich nötig

Effiziente Implementierung der Symboltabelle

Optimierung von Zugriffen auf Symbole:

- Implementierung einer eigenen Klasse `Symbol` zur Repräsentation von Symbolen.
- Vermeidet langsame String-Vergleiche.

Details dazu in Appel's Buch Seite 106ff.

Inhalt der Symboltabelle

Eine Symboltabelle bildet Bezeichner ab auf “semantische Werte”.

- Semantischer Wert eines *Programms*:
  - Klassennamen mit ihren semantischen Werten.
- Semantischer Wert einer *Klasse*:
  - Feldnamen mit ihren semantischen Werten,
  - Methodennamen mit ihren semantischen Werten
  - Elternklasse — nicht in MiniJava
- Semantischer Wert eines *Feldes*:
  - Typ
- Semantischer Wert einer *Methode*:
  - Ergebnistyp,
  - Vektor der Parameter mit ihren Typen,
  - lokale Variablen mit ihren Typen.

Inhalt der Symboltabelle

Das Wörtchen “mit” wird hier immer durch eine Abbildung, konkret entweder durch einen binary search tree (funktional) oder durch eine HashMap (imperativ) realisiert.

Später werden die semantischen Werte noch um zusätzliche Komponenten erweitert, etwa (relative) Speicheradresse oder Größe (in Byte).

Beispiel: Symboltabelle

```
class B {
  C f; int [] j; int q;
  public int start (int p, int q) {
    int ret ; int a ;
    /* ... */
    return ret;
  }
  public boolean stop (int p) {
    /* ... */
    return false;
  }
}

class C { /* ... */
}
```

Die Typprüfung

Die Typüberprüfung selbst besteht nun aus einem Satz rekursiver Methoden, welche in Gegenwart einer Symboltabelle ein Programm / Klasse / Methode / Variablendeklaration / . . . auf Typkorrektheit prüfen (unter Bezugnahme auf die Symboltabelle) und als Seiteneffekt diese ergänzen.

Da Klassen und Methoden sich (in MiniJava) gegenseitig rekursiv verwenden dürfen bietet es sich an, die Typüberprüfung in zwei bis drei Phasen zu gliedern:

- In der ersten Phase werden nur die definierten *Klassen* eingetragen (mit Platzhalter als semantischem Wert). Fehler können auftreten, wenn eine Klasse mehrfach deklariert wird.
- In der zweiten Phase werden die *Methoden* mit ihrem Aufrufmuster (Parameterliste und Rückgabetyp) eingetragen. Fehler können auftreten, wenn ein undefinierter Typ benutzt wird, oder Methoden- oder Parameternamen doppelt deklariert werden. (Overloading erfordert eine komplexere Behandlung — nicht in MiniJava).

Die Typprüfung

- In der dritten Phase werden alle *Rümpfe* auf Typkorrektheit geprüft und die lokalen Variablen eingetragen. Fehlermöglichkeiten: falsche Typen bei Methodenaufruf, Verwendung undeklarierter lokaler Variablen, etc.

Die erste und zweite Phase können zusammengefasst werden (*Aufbau der Symboltabelle*). Man muss dann in der nächsten Phase (*Typprüfung*) nochmal prüfen, ob alle verwendeten Typen tatsächlich deklariert wurden.

Typprüfung einer Variablendeklaration

```
 // Type ty;
 // String id;
 public void visit(VarDecl n) {
  Type   t = n.ty.accept(this);
  String id = n.id.toString();

  if (currMethod == null) {
    if (!currClass.addVar(id,t)) {
       // Fehler
    } else if (!currMethod.addVar(id,t)) {
       // Fehler
    }
  }
}
```

Typprüfung eines Methodenaufrufs

```
//  Expression e;
//  String id;
//  List<Expression> es;
public Type visit(Invoke n) {
  Type t = n.e.accept(this);
  if (! (t instanceof ObjType)){ /* Fehler */ }
  String mname = e.id.toString();
  String cname = ((ObjType) t).c;
  Method calledMethod =
       TypeCheckVisitor.symbolTable.getMethod(mname,cname);
  for ( int i = 0; i < n.es.size(); i++ ) {
      Type t1 = null;  Type t2 = null;
      if (calledMethod.getParamAt(i)!=null)
          t1 = calledMethod.getParamAt(i).type();
      t2 = n.es.elementAt(i).accept(this);
      if (!TypeCheckVisitor.symbolTable.compareTypes(t1,t2)){ /* Fehler
  }
  return TypeCheckVisitor.symbolTable.getMethodType(mname,cname);
}
```

Programmieraufgabe

Implementierung von Typprüfung für MiniJava, bestehend aus:

- Aufbau einer Symboltabelle (imperativ oder funktional).
  - Eine verschachtelte Blockstruktur muss in MiniJava nicht unterstützt werden.
  - Mit der im Buch Seite 111 skizzierten Struktur der Symboltabelle ist keine überschattende hash map nötig, da der Gültigkeitsbereich einer Variable oder Methode immer eine ganze Methode oder Klasse ist.
- Typprüfung mithilfe der Symboltabelle

Optional:

- Verwendung einer Symbol Klasse gemäß Program 5.6.
- Implementierung von `beginScope` und `endScope`.
- Fehlerbehandlung wie auf Seite 114 skizziert.

Aktivierungssätze (Activation Records)

- *Aktivierungssätze (Activation Records)* dienen zur Verwaltung von lokalen Variablen etc in block-strukturierten Programmiersprachen.
- Sie werden als LIFO (last-in-first-order) Datenstruktur mittels eines *Stacks* realisiert.
- In MiniJava müssen Methodenparameter und lokale Variablen in Aktivierungssätzen abgelegt werden.
- Anmerkung: In Sprachen höherer Ordnung, die (verschachtelte) Funktionen als Resultat liefern können, ist eine reine LIFO Verwaltung nicht ausreichend.

Lokale Variablen

```
public int f (int x, int y) {
  int z;
  z=x+y;
  g(z);
  if (y==0) { return x; }
  else      { return f(z,y-1); }
}
```

Wenn `f` aufgerufen wird, so werden neue Instanzen von `x`, `y` und `z` geschaffen. Die formalen Parameter werden vom Aufrufer (*caller*) initialisiert.

Werden lokale Variablen beim Verlassen einer Funktion gelöscht, so kann ein Stack zu deren Verwaltung benutzt werden.

Anmerkung: Manchmal bleiben aber lokale Variablen nach Verlassen einer Funktion am Leben!

Höherstufige Funktionen

```
fun incBy x =
  let fun g y = x+y
   in g
  end
let incBy3 = incBy 3
let z = incBy3 1
```

Wird `incBy` verlassen, so darf `x` noch nicht zerstört werden!

In MiniJava gibt es allerdings keine höherstufigen Funktionen.

Stack

- Der Stack ist ein Speicherbereich, dessen Ende durch den *Stack Pointer* (spezielle Variable) bezeichnet wird. Aus historischen Gründen wächst der Stack nach unten (hin zu niedrigeren Adressen). Genauer bezeichnet der Stack Pointer das erste *freie* Wort über dem Stack.
- Bei Aufruf einer Funktion wird der Stack um einen Aktivierungssatz erweitert (durch Erniedrigen des Stack Pointers), welches Platz für alle lokalen Variablen, Rücksprungadresse, Parameter, etc. bereithält. Dieser Aktivierungssatz heißt *Frame* oder *Activation Record*. Beim Verlassen der Funktion wird der Stack Pointer entsprechend zurück (nach oben!) gesetzt, sodass der Frame entfernt ist.
- Der Prozessorhersteller gibt oft eine bestimmte Aufrufkonvention vor (Parameterlayout etc.), damit Funktionen sprachübergreifend verwendet werden können.

Inhalt eines Aktivierungssatzes

- Administrative Information: static link, return address
- Argumente der Methode
- Lokale Variablen
- Temporäre Variablen und Register

Zur Verwaltung des Stacks als eine Sequenz von Aktivierungssätzen werden zwei spezielle Register verwendet:

- *frame pointer*: zeigt auf den Beginn des aktuellen Aktivierungssatzes;
- *stack pointer*: zeigt auf die nächste freie Stackaddresse.

Der Frame Pointer

- Das Ende des aktuellen Frames wird durch den Stack Pointer SP bezeichnet, genauer gesagt $SP - \texttt{wordSize}$.
- Der Anfang des aktuellen Frames steht in einer weiteren Sondervariablen, dem *Frame Pointer* (FP).
- Ruft $f(x, y)$ die Funktion $g(z)$ auf, so wird der Inhalt von FP auf dem Keller abgelegt (als sog. *dynamic link*) und SP nach FP übertragen, da ja der neue Frame (für g) gerade bei SP anfängt.
- Beim Verlassen von g wird FP wieder nach SP geschrieben und FP aus dem Stack restauriert.
- Ist die Framegröße jeder Funktion zur Compilezeit bekannt, so kann FP jeweils ausgerechnet werden als $SP + \texttt{framesize}$.
- Wird etwa Cs `alloca` (“malloc auf dem Stack”) unterstützt, so ist die Framegröße nicht fest.

Register

Register sind in den Prozessor eingebaute extrem schnelle Speicherelemente. RISC Architekturen (Sparc, PowerPC, MIPS) haben oft 32 Register à ein Wort; der Pentium hat acht Register.

- Lokale Variablen, Parameter, etc. sollen soweit wie möglich in Registern gehalten werden.
- Oft können arithmetische Operationen nur in Registern durchgeführt werden.

Aufrufkonventionen

- Der Hersteller gibt vor, welche Register *caller-save*, d.h., von der aufgerufenen Funktion beschrieben werden dürfen; und welche *callee-save* sind, also von der aufgerufenen Funktion im ursprünglichen Zustand wiederherzustellen sind.
- Befindet sich eine lokale Variable o.ä. in einem caller-save Register, so muss die der Aufrufer vor einem Funktionsaufruf im Frame abspeichern, es sei denn, der Wert wird nach Rückkehr der Funktion nicht mehr gebraucht.
- Der Pentium hat drei caller-save Register und vier callee-save Register.

Parameterübergabe

Je nach Architektur werden die ersten k Parameter in Registern übergeben; die verbleibenden über den Frame.

Und zwar werden sie vom Aufrufer in die obersten Frameadressen hineingeschrieben, sodass sie on der aufgerufenen Funktion dann jenseits des Frame Pointers sichtbar sind. (Die `__cdecl` Konvention). Manchmal werden sie auch in den neuen Frame hineingeschrieben (`__stdcall`).

Beim Pentium werden i.d.R. alle Parameter über den Frame übergeben, allerdings kann man bei GCC auch Werte $0 < k \leq 3$ einstellen.

Escapes

Lokale Variablen und Parameter dürfen nicht in Registern abgelegt werden, wenn vom Programm auf ihre Adresse zugegriffen wird. Das ist der Fall bei Verwendung von Cs Adressoperator, Pascals Call-by-Reference, und bei der Verwendung von Zeigerarithmetik.

Z.B. darf man in C annehmen, dass die Funktionsparameter an aufeinanderfolgenden Adressen liegen.

Solche Parameter (“escapes”) können nur über den Frame übergeben werden.

Zum Glück gibt es aber in MiniJava keine Escapes.

Rücksprung

Wurde g von Adresse a aus aufgerufen, so muss nach Abarbeitung von g an Adresse $a + \texttt{wordSize}$ fortgefahren werden. Diese Adresse muss daher vor Aufruf irgendwo abgespeichert werden.

Eine Möglichkeit ist der Stack; das wird von den Pentium `call` und `ret` Befehlen explizit unterstützt.

Eine andere Möglichkeit besteht darin, die Rücksprungadresse in einem Register abzulegen, welches dann ggf. von der aufgerufenen Funktion im Frame zu sichern ist.

Static link

In manchen Sprachen (z.B.: Pascal ode ML) können lokale Funktionen innerhalb eines Blocks deklariert werden und auf die lokalen Funktionen dieses Blocks Bezug nehmen.

Kann solch eine lokale Funktion nach außen gereicht werden, etwa als Rückgabewert, so hat man höherstufige Funktionen und man muss ganz anders vorgehen. Ist das nicht der Fall, so funktioniert ein Stack nach wie vor, allerdings muss man der inneren Funktion sagen, wo die lokalen Variablen, auf die sie sich bezieht, zu finden sind.

- Entweder in einem weiter unter liegenden Stack Frame. Auf diesen muss dann im aktuellen Frame ein Verweis abgelegt werden: das *static link*.
- Oder man übergibt alle diese lokalen Variablen als extra Parameter. Das bezeichnet man als *lambda lifting* und wird vom Pascal-nach-C compiler `p2c` verwendet.

Interface für Aktivierungssätze

```
package Frame;
public abstract class Frame {
    public Temp.Label name;
    public List<Access> formals;
    public abstract Frame newFrame(Symbol.Symbol name, int n);
    public abstract Access allocLocal();
   /* ... more stuff later ... */
}
public abstract class Access {
    public abstract String toString();
}
```

Implementierung von Aktivierungssätzen

- Die `Access` Klasse beschreibt die Zugriffsmethode auf eine lokale Variable oder einen Parameter. Konkrete Implementierungen sind:
  - `InFrame` (repräsentiert eine relative Position auf dem Stack)
  - `InReg` (repräsentiert ein Register)
- Die Implementierung eines Frames beinhaltet:
  - Zugriffsmethoden (`Access`) der Parameter
  - Anzahl der bisher allokierten lokalen Variablen
  - Label des Beginns des Codes (`name`)
  - Code zum Anpassen der Sicht auf Zugriffsmethoden . . .

Implementierung der abstrakten Klassen

Die abstrakte Klasse `Frame` wird architekturspezifisch implementiert, etwa

```
PentiumFrame extends Frame ...
SparcFrame extends Frame ...
```

Die abstrakte Klasse `Access` (Zugriff auf Variablen) wird durch zwei Subklassen implementiert:

```
InReg extends Access ...
InFrame extends Access ...
```

Alle Klassen werden im Laufe des Projekts noch erweitert. Für den Moment brauchen wir nur die Methode `newFrame`, die ein neues Frame Objekt in Abhängigkeit der Parameterzahl bereitstellt.

Natürlich muss im Frame Objekt dessen Größe (frameSize) mitgefuehrt und aktualisiert werden.

Programmieraufgabe

Implementierung der architekturspezifischen `Frame` Klasse.

Zu beachten:

- Mehrfach vorkommende Methodennamen
- Verwendung eines factory patterns (`newFrame`)
- Zugriffsmethoden: `InFrame`, `InReg`

Optional: Behandlung von Funktionen mit mehr Parametern als zur Verfügung stehenden Registern (verwende eine statische Liste von Registernamen).

Eine Implementierung von `Temp` ist verfügbar.

Übersetzung in Zwischencode

Warum Zwischencode?

- Ermöglicht Wiederverwendung von front- bzw back-end des Compilers bei einem Wechsel zu anderer Ziel- (*architekturunabhängig*) bzw Quell-sprache (*sprachunabhängig*).
- Modularisierung des Compilers
- Mit Interpreter der Zwischensprache ist rapid-prototyping eines Compilers möglich
- Vergleiche Java/JVM Philosophie: “compile-once, run-everywhere”

Überblick Zwischencode

- Die abstrakte Syntax soll in Zwischencode übersetzt werden.
- Zwischencode ähnelt GOTO-Programmen, hat aber *geschachtelte Ausdrücke* (“tree language”).
- Zur Übersetzung von Funktions- (Methoden-, Prozedur-) aufrufen, müssen *stacks* und *stack frames* implementiert werden.
- Zur Kapselung architekturabhängiger Details verwenden wir Schnittstellen und abstrakte Klassen, die erst später wirklich implementiert werden (z.B. für Parameterübergabe in Methodenaufrufen).

Ausdrücke (“Expressions”) in der Zwischensprache

CONST (i) Die (Integer) Konstante i.

NAME (l) Die symbolische Konstante (z.B. Sprungadresse) l.

TEMP (t) Das abstrakte Register (*temporary*) t. Die Zwischensprache stellt *beliebig viele temporaries* für Zwischenergebnisse o.ä. bereit.

BINOP (o, e_1, e_2) Die Anwendung des binären Operators o auf die Ausdrücke e_1 and e_2, die zuvor in dieser Reihenfolge ausgewertet werden.

Operatoren sind: PLUS, MINUS, MUL, DIV, AND, OR, XOR, LSHIFT, RSHIFT, ARSHIFT.

MEM (e) bezeichnet den Inhalt der Speicheradresse e (Größe: *wordSize*).

CALL ($f, e_1, \ldots, e_n$) Anwendung der Funktion f auf Argumentliste $e_1, \ldots, e_n$, die in dieser Reihenfolge ausgewertet werden.

ESEQ (s, e) Ausführung von Befehl s gefolgt von Auswertung von e.

Befehle (“Statements”) der Zwischensprache

MOVE (TEMP t, e**)** Auswerten von e und abspeichern in *temporary* t.

MOVE (MEM e_1, e_2**)** Auswerten von e_1 zu Adresse a. Den Wert von e_2 in Speicheradresse a abspeichern.

EXP (e**)** Auswerten von e und verwerfen des Ergebnisses.

JUMP ($e, labs$**)** Ausführung bei Addresse e fortsetzen, wobei die Liste *labs* alle möglichen Werte für diese Adresse angibt.

CJUMP (o, e_1, e_2, t, f**)** Bedingter Sprung: Vergleiche die Werte von e_1 und e_2 mit Operator o. Ist der Vergleich wahr, so verzweige nach t, sonst nach f.

Operatoren: EQ, NE, LT, GT, LE, GE, ULT, UGT, ULE, UGE.

SEQ (s_1, s_2**)** Sequentielle Ausführung.

LABEL (n**)** Definiere die Konstante n als aktuelle Maschinencode-Adresse.

Übersetzung

Wir werden alle Phrasen von MiniJava in `Tree.Exp` übersetzen.

Boolesche Ausdrücke werden dabei mit der Konvention *true*=1, *false*=0 übersetzt; Befehle als Ausdrücke mit Dummywert, z.B. 99 wiedergegeben.

Wer will, kann später oder bei entsprechendem Mut auch sofort folgende Verbesserungen implementieren:

- MiniJava Statements werden als `Tree.Stm` übersetzt.
- Boole'sche Ausdrücke in `if` und `while` Statements werden direkt in Sprungbefehle (parametrisiert über den beiden Sprungzielen) übersetzt.

Das Buch schlägt die Verwendung einer Klasse `Translate.Exp` vor, die abhängig von der Verwendung (expression, statement, conditional) unterschiedlichen Code erzeugt.

Einfache Variablen

Eine Variable wird in der Symboltabelle durch einen `Access` repräsentiert.

Die Übersetzung einer Variablen ergibt sich durch Aufruf einer abstrakten Methode in der `Access` Klasse:

```
public abstract Tree.Exp exp(Tree.Exp base)
```

Diese ist in den Unterklassen (`InReg`,`InFrame`,`InMem`) entsprechend zu implementieren.

Der Parameter `base` muss den aktuellen Bezugspunkt, also den Framepointer, bzw. `this` liefern. Bei `InReg` wird er ignoriert.

Für "`this`" sollte in `Frame` eine weitere (neben dem Framepointer) Pseudovariable vorgesehen werden, die dann bei Methodenaufruf entsprechend besetzt wird.

Exkurs: In Sprachen mit verschachtelten Gültigkeitsbereichen, muß bei einem Variablenzugriff die Kette von "static links" verfolgt werden.

Arrays und Objekte

Ein Array der Länge n wird abgespeichert als Block von $n + 1$ aufeinanderfolgenden Wörtern. Das erste Wort enthält die Länge des Arrays, anschließend folgen die Fächer.

Ein Objekt mit n Instanzvariablen wird als Block von n konsekutiven Wörtern repräsentiert.

In echtem Java braucht man einen weiteren Eintrag, meist am Beginn, aus dem die Klasse hervorgeht.

Man erzeugt und initialisiert solche Speicherblöcke durch Systemaufrufe, (ähnlich zu “malloc” in C):

- `L_halloc_obj(n)`: zum Allokieren eines Objektes mit n Feldern
- `L_halloc_arr(n)`: zum Allokieren eines Arrays mit n Fächern
- `L_init_obj(a,n)`: zum Initialisieren eines Objektes mit n Feldern in Adresse a

Arrays und Objekte

- `L_init_arr(a,n)`: zum Initialisieren eines Arrays mit n Fächern in Adresse a

Der Aufruf erfolgt wie für alle Methoden, zB.

```
new CALL(new NAME(new Label("L_init_arr")),
         new CONS<Tree.Exp>(new TEMP(t1),
             new CONS<Tree.Exp>(new CONST(12), new NIL()));
```

Dies soll in folgende Methode abstrahiert werden:

```
public abstract class Frame {
 ...
 abstract public Tree.Exp externalCall(String func,
                                       List<Tree.Exp> args);
 ...}
```

Gute Compiler fordern Speicher in größeren Blöcken an und nicht bei jedem “new” erneut.

Exkurs: Behandlung von Objekten und Arrays

In MiniJava wird ein Objekt oder Array durch einen Pointer repräsentiert.

Zuweisung auf Objekte oder Arrays bedeutet Pointerzuweisung.

```
int [] a;          // Deklaration eines Array-of-int
int [] b;          // Deklaration eines Array-of-int
a = new int[12];   // Initialisierung des arrays
b = new int[12];   // Initialisierung des arrays
a = b;             // Pointer-Zuweisung
```

Sowohl `a` als auch `b` zeigen auf dieselbe Speicheradresse.

Der Speicher, der für `a` angelegt wurde ist Müll (“garbage”).

Bemerkung: in C oder Pascal bezeichnet eine Variable die gesamte Struktur. Eine Zuweisung muß daher die gesamte Struktur kopieren. Um dies zu ermöglichen, müßte man `wordSize()` durch die (jeweils verschiedene) Größe der Arrayeinträge ersetzen.

Arrayzugriff

Ein Array wird repräsentiert durch die Adresse seines Speicherblocks.

Demgemäß befindet sich das i-te Fach des Arrays a an der Adresse $a + i \cdot \texttt{wordSize}$ (oder $a + (1 + i) \cdot \texttt{wordSize}$) und die Übersetzung des Arrayzugriffs erfolgt so:

$$\begin{aligned}\texttt{translate}(a[e]) = \text{MEM}(\text{BINOP}(\text{PLUS}, &\texttt{translate}(a),\\ &\text{BINOP}(\text{MUL}, \text{CONST}(\texttt{frame.wordSize}()), \texttt{translate}(e))))\end{aligned}$$

Im Allgemeinen unterscheidet man zwischen *l-values*, Werte die links einer Wertzuweisung vorkommen (Adresse), und *r-values*, Werte die rechts einer Wertzuweisung vorkommen (Wert).

In der Zwischensprache kann $\text{MEM}(s)$ sowohl als *l-value* als auch als *r-value* verwendet werden. Bei der Übersetzung in Maschinensprache ist der Unterschied zu beachten.

Arraygrenzen

Es ist im Allgemeinen sinnvoll, vor einem Arrayzugriff die Einhaltung der Arraygrenzen zu überprüfen.

Werden sie nicht eingehalten, so kann man mit einem Systemaufruf `L_raise` eine entsprechende Fehlermeldung auslösen.

Aus Gründen der Portabilität ist es sinnvoll, alle erforderlichen Systemaufrufe in einer speziellen “MiniJava-Laufzeitbibliothek” zusammenzufassen.

Der Interpreter der Zwischensprache kennt diese Systemfunktionen bereits!

Feldzugriffe

Ein Feldzugriff $e.f$ erfolgt ähnlich wie ein Arrayzugriff durch Addieren eines geeigneten Offsets zur Übersetzung von e.

Der Offset richtet sich nach dem *statischen Typ* von e, den man ja bei der semantischen Analyse zur Verfügung hat.

Er ist unter der entsprechenden Klasse und Instanzvariable in der Symboltabelle in Form eines `Access` abgespeichert.

Es empfiehlt sich die Einrichtung einer weiteren Unterklasse `InMem` von `Access` mit entsprechender Implementierung der `exp`-Methode.

Arithmetische Ausdrücke und Strings

Arithmetische Ausdrücke lassen sich sehr direkt auf die Zwischensprache abbilden.

Stringliterale werden als NAME(*lab*) übersetzt, wobei *lab* ein “frisches” Label ist, an dem das Stringliteral definiert wird. Diese Definition muss durch einen Aufruf von `frame.string`(*lab*, *string*) erzeugt werden. Die `string`-Funktion ist hierzu architekturabhängig zu implementieren. Hier ist das GCC-Kompilat von `main(){printf("MiniJava!\n");}`

```
.LC0:   .string "MiniJava!\n"
        .text
.globl main
        .type   main, @function
main:   ...
        pushl   $.LC0
        call    printf
        ...
```

If und While

If und While Konstrukte müssen in CJUMP(...) mit frischen Labels für die Zweige übersetzt werden. Vorerst erhalten beide Zweige eines If Konstrukts Labels, später wird der Code zu einem “fall-through” im Else-Zweig umgewandelt.

Methodenaufruf

Methodenaufrufe werden ziemlich direkt in Funktionsaufrufe in der Zwischensprache übersetzt. Zu bemerken ist folgendes:

- In MiniJava gibt es kein “overloading” oder “dynamic dispatch”. Die Methodenauswahl richtet sich daher nach dem *statischen Typ* (durch semantische Analyse ermittelte Klasse) des aufgerufenen Objekts.
- Die Methode m in Klasse C wird durch eine Zwischensprachen-Funktion mit Namen $C\$m$ repräsentiert.
- Der Funktion $C\$m$ ist das aufgerufene Objekt als zusätzlicher Parameter zu übergeben.

Der Aufruf $e.m(e_1, \ldots, e_n)$ wird folgendermaßen übersetzt:

$\text{CALL}(C\$m, [\texttt{translate}(e), \texttt{translate}(e_1), \ldots, \texttt{translate}(e_n)])$

wobei $\texttt{translate}(e)$ den “this”-pointer der aufgerufenen Methode bestimmt.

Methodendefinitionen

Für jede Methodendefinition ist ein *Fragment* Objekt zu erzeugen, das einen `Frame` (zur Verwaltung von lokalen Variablen und Parameter) und den Code der Methode enthält. Struktur des Codes im Fragment:

- *Prolog*:
  - Label der Funktion
  - Anpassen von stack- und frame-pointer
  - Abspeichern der “callee-save” Register
- *Rumpf*: User-code der entsprechenden Methode
- *Epilog*
  - Übertragen des Resultat-Wertes
  - Laden der “callee-save” Register
  - Zurücksetzen von frame- und stack-pointer
  - Eine *return* Instruktion

Methodendefinitionen

Im Fragment-Objekt sind Felder für den Methodenrumpf (vom Typ `Tree.Stm`) sowie für den Frame der Methode (vom Typ `Frame`) vorzusehen. Der Rückgabewert wird in ein spezielles Register `frame.RV()` hineingeschrieben.

Die Symboltabelle bindet jede Methode an das entsprechende Fragment-Objekt.

Variablendefinitionen

Bei einer Variablendefinition wird ein entsprechendes `Access`-Objekt erzeugt, z.B. mit `allocLocal()` und in der Symboltabelle abgelegt.

Instanzvariablen werden ebenso als `Access`-Objekte, konkret als `InMem`, behandelt.

Beispiel: Übersetzung von Methodendefinitionen

```
public Stm visit (MethDecl d) {
  curr_mthd=d.id;
  Frame frame = frameFactory.newFrame(d.id);
  curr_frame=frame;
  bindMthdToFrame(frame);           // store frame in symbol table
  d.ds.accept(new TransDeclList()); // adds local vars to symbol table
  List<Stm> ss = d.ss.accept(new TransStatementList());
  Exp res_code = d.e.accept(new TransExpression());
  frame.procEntryExit1(ss);         // wraps entry/exit code around ss
  Stm body_code = SEQ(mkSeq(ss), MOVE(TEMP(frame.RV()),res_code));
  bindMthdToCode(body_code);        // store code in symbol table
  curr_mthd="";  curr_frame=null;
  return body_code ;
}
```

Zum Schluss

Am Ende dieser Phase ist die Symboltabelle folgendermaßen verändert:

- Semantischer Wert eines *Programms*:
  - Klassennamen mit ihren semantischen Werten.
- Semantischer Wert einer *Klasse*:
  - Elternklasse (optional),
  - Feldnamen mit ihren semantischen Werten,
  - Methodennamen mit ihren semantischen Werten.
- Semantischer Wert eines *Feldes*:
  - Typ,
  - *Zugriffsmethode* (`Access`-Objekt).
- Semantischer Wert einer *Methode*:
  - Ergebnistyp,
  - Vektor der Parameter mit ihren Typen,

Zum Schluss

- lokale Variablen mit ihren Typen und *Zugriffsmethoden*,
- ein Code-*Fragment* bestehend aus
  - einem `Frame`-Objekt mit architekturspezifischer Information, und
  - einem Methodenrumpf vom Typ `Tree.Stm`, der den Rückgabewert im speziellen Register `frame.RV()` zurückgibt.

Programmieraufgabe

Implementierung der Übersetzung von abstrakter Syntax in Zwischensprache.

Interface der `Fragment` Klasse:

```
package Translate;
...
public abstract class Fragment { public Fragment next; }
public ProcFragment(Tree.Stm body, Frame.Frame frame);
public DataFragment(String data);
...
public class Translate {
 ...
 private Fragment fragments;  // linked list of accumulated frags
 public void procEntryExit(Exp body);  // wrap in prologue&epilog
 public Fragment getResult(); // extract fragment list
 ...
}
```

Garbage collection

Ziel: Entfernung nicht mehr erreichbarer Objekte aus der Halde.

- Beispiel: wo braucht man GC?
- Mark-and-sweep: Markieren aller erreichbaren Objekte mit Tiefensuche; Entfernung aller Nichtmarkierten.
- Reference counting: jedes Objekt erhält einen Zähler, in dem die Zahl der auf es verweisenden Referenzen gespeichert wird. Wird dieser Zähler Null, so kann das Objekt freigegeben werden.
- Copying Collection: Aufteilung des Speichers in zwei gleichgrosse Bereiche (*from-space*, *to-space*). Von Zeit zu Zeit: Kopieren aller erreichbaren Objekte vom *from-space* nach *to-space*. Vertauschen von *from-space* und *to-space*.

- Generational Garbage Collection: Die Objekte werden je nach ihrem Alter in Generationen eingeteilt. Auf der jungen Generation wird öfter GC durchgeführt, als bei der alten.
- Concurrent Garbage Collection: Die GC wird während der Programmabarbeitung parallel im Hintergrund durchgeführt.

Warum Garbage Collection?

```
List y = null;
{
 List x = new List(5,3);
} // x ist jetzt Garbage
{
 List x = null;
 for(int i;i <= 10;i++) {
    x = new List(i,x);
} // x und alles von x Erreichbare ist Garbage
{
 List x = new List(5,new List(4,null));
 y = x.tl;
} // x ist Garbage, x.tl aber nicht.
```

Mark and Sweep

Jedes Objekt erhält ein zusätzliches Boolesches Feld “`marked`”.

Rekursive Implementierung:

```
DFS(x) { if (x == null) ;
         else if (! x.marked) {
                    x.marked = true;
                    for (f in Felder(x))
                        DFS(x.f);} }
SWEEP(){ p = erste Adresse im Heap;
         while (p < letze Adresse im Heap) {
            if (p.marked)
                p.marked = false;
            Fuege p in Freelist ein;
            p = p+SIZE;
        }   } }
```

Wir gehen davon aus, dass alle Objekte dieselbe Größe `SIZE` haben. Falls nicht: mehrere Freelists. Problem dann: Fragmentierung.

Optimierung von Mark-and-Sweep

Stackhöhe bei DFS = Länge des längsten Pfades im Heap. Sehr groß bei “langen” Datenstrukturen, z.B. Listen.

Verbesserung: Expliziter Stack statt Rekursion: Eine Adresse pro Stackeinheit statt ein ganzer Stackframe.

Verbesserung: Eliminierung des Stacks durch Pointer-Reversal; Nach dem ersten Besuch eines Knotens verweist sein erstes Feld auf den Elternknoten. Man merkt sich in jedem Knoten die Anzahl der bereits bearbeiteten Felder und geht rückwärts, wenn alle Felder bearbeitet wurden.

Referenzzählen

- Jedes Objekt erhält ein zusätzliches Feld `count`, in dem die Anzahl der Verweise auf es gespecihert werden.
- Bei Allokierung "`new`" wird `count` auf 1 gesetzt.
- Bei Zuweisung `x.f = y` setze `y.count++;x.f.count--;`.
- Wird ein Referenzzähler Null, so füge das Objekt der Freelist hinzu und dekrementiere die Zähler aller Felder.

Nachteile: Weniger effizient; Zyklische, aber von außen nicht errecihbare Datenstrukturen schwer zu entdecken.

Copying Collection

Die Halde wird in zwei gleichgroße Teile aufgeteilt: *from-space*, *to-space*. Nur der *from-space* enthält aktive Objekte.

Bei GC werden alle erreichbaren Objekte vom *from-space* nach *to-space* kopiert. Anschließend werden *from-space* und *to-space* vertauscht.

Vorteile: Keine Freelist nötig, Keine Fragmentierung tritt auf.

Nachteil: Doppelter Platzbedarf.

Durchführung des Kopierens

- Der *to-space* erhält zwei Zeiger: *next* (nächste freie Adresse), *scan* (Objekte unterhalb dieser Adresse wurden komplett verarbeitet, d.h., ihre Kinder sind auch schon im *to-space*.
- Zu Beginn: Kopiere alle Wurzelobjekte (erreichbar von Variablen) nach *to-space*; schreibe Adresse des neuen Objekts in das erste Feld des alten Objekts im *from-space*. “Forward-Pointer”. Setze *scan*=Beginn des *to-space*.
- Solange $scan < next$:
- Verarbeite das Objekt an der Stelle *scan* durch . . .
- Rüberkopieren seiner Kinder bzw.
- Aktualisieren der Feldeinträge, falls diese auf ein bereits im *to-space* befindliches Objekt verweisen, erkennbar daran, dass der erste Feldeintrag des Objekts ein Forward-Pointer ist.

Durchführung des Kopierens

Vorteil: Keine Fragmentierung Nachteil: Nahe zusammenliegende Objekte werden i.a., nicht auf nahe zusammenliegende Objekte kopiert : schlechte Cache-Performance. Abhilfe: Verwendung von DFS statt der (impliziten) BFS.

Nebenläufige GC

Der Programmablauf zerfällt in zwei “Prozesse”:

Kollektor. Er sammelt unerreichbare Objekte auf, führt also die eigentliche GC durch.

Mutator: Er fügt neue Objekte hinzu und verändert Referenzen; er entspricht dem eigntlichen Programm.

Bei den bisherigen Verfahren führt der Kollektor immer einen ganzen Ablauf durch; der Mutator wird während dieser Zeit angehalten.

Bei nebenläufiger GC darf der Mutator auch während der Arbeit des Kollektors tätig werden, muss dabei aber mit diesem in geeigneter Weise kooperieren.

Abstrakte GC

Alle bisherigen GC Algorithmen teilen die Objekte in drei Klassen auf:

- Weiße Objekte: Noch gar nicht besucht.
- Graue Objekte: Schon besucht, Kinder aber noch nicht vollst. verarbeitet (auf Stack, bzw. zwischen *scan* und *next*).
- Schwarze Objekte: Schon besucht, Kinder grau oder schwarz.

Grundalgorithmus:

- Mache Wurzelobjekte grau.
- Solange es graue Objekte gibt:
- Wähle graues Objekt, mache es schwarz und seine weißen Kinder grau.

Invarianten: (1) Schwarze Objekte verweisen nie auf Weiße. (2) Graue Objekte befinden sich in einer Datenstruktur des Kollektors.

Der Mutator darf diese Invarianten nicht verletzen.

Konkrete Verfahren

- Dijkstra, Lamport, et al.: Will der Mutator ein weißes Objekt in ein Feld eines schwarzen Objekts schreiben, so ist ersteres Objekt grau zu machen (durch zusätzliche Instruktionen).
- Steele: Will der Mutator ein weißes Objekt in ein Feld eines schwarzen Objekts schreiben, so ist letzteres Objekt wieder grau zu machen (durch zusätzliche Instruktionen).
- Baker: Referenziert der Mutator ein weißes Objekt, egal zu welchem Zweck, so wird dieses sofort grau gemacht.

Weiterführendes

- Viele Verfahren, insbes. nebenläufige können auf der Ebene von Seiten des virtuellen Speichers implementiert werden. Dabei kann spezielle OS-Infrastruktur, z.B. Read-only Bits benutzt werden.
- Man kann den Heap in zwei oder mehrere Generationen aufteilen. Die “junge Generation” wird öfter GC’ed, als die ältere.

  Wird z.B. bei OCAML verwendet.

Instruktionsauswahl

Ziel: Übersetzung der IR-Bäume in Assemblerinstruktionen mit (beliebig vielen) abstrakten Registern.

- Die ix86 (Pentium) Architektur.
- Abstrakter Datentyp für Assemblerinstruktionen.
- Instruktionsauswahl als Kacheln von Bäumen.
- Greedy-Algorithmus (Maximal-Munch) und optimaler Algorithmus mit dyn. Prog.
- Implementierungsaufgabe.

ix86 Assembler: Register

- Der ix86 (im 32bit Modus) hat 8 Register: `%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp`
- `%esp` ist der stack pointer; nicht anderweitig verwendbar.
- `%ebp` kann als frame pointer benutzt werden. Wird kein frame pointer benötigt, so wird `%ebp` allgemein verfügbar.
- `%eax, %edx, %ecx` sind caller-save
- `%ebx, %esi, %edi, %ebp` sind callee-save

ix86 Assembler: Operanden

Ein *Operand* ist entweder

- ein Register, oder
- eine Konstante, oder
- eine Speicherstelle.
- Eine *Konstante* wird geschrieben als $\$n$ wobei n eine Zahl ist. Der *Assembler* gestattet hier auch arith. Ausdr. mit Konstanten und Abkürzungen.
- Eine *Speicherstelle* wird geschrieben als $n(reg)$, wobei n Zahl oder arith. Ausdr. wie oben ist und *reg* ein Register ist. Es bezeichnet die Adresse $reg + n$.

ix86 Assembler: Verschiebebefehle

Ein Verschiebebefehl hat die Form

`movl` src, dst

Er verschiebt den Inhalt von `src` nach `dst`. Hier kann `src` ein beliebiger Operand sein; `dst` darf natürlich keine Konstante sein. Außerdem dürfen `src`, `dst` nicht beide zugleich Speicherstellen sein.

```
framesize = 48
     movl     %ebx,-4+framesize(%esp)
     movl     $33,%eax
     movl     (%ecx),%esi
```

ix86 Assembler: Arithmetische Befehle

- Plus, Minus, Mal haben das allgemeine Format

  $$op\ src, dst$$

  wobei `op` entweder `addl`, `subl`, `imull` ist. Der Effekt ist

  $$src\ op\ dst \rightarrow dst$$

  Die Operanden sind denselben Einschränkungen wie bei den Verschiebebefehlen unterworfen. Bei `imull` muss `dst` ein Register sein.

ix86 Assembler: Division

Das Format des Divisionsbefehls ist:

```
cdq
idivl op
```

Dies platziert das Ergebnis der Division `%eax` durch `op` nach `%eax`. Außerdem wird das Register `%edx` überschrieben (mit Müll) und *op* darf keine Konstante sein.

ix86 Assembler: Sonderbefehle

Als Beispiel für einen der vielen “barocken” Sonderbefehle geben wir hier

$$\texttt{leal}\ n_1(,reg,n_2),\ dst$$

Der Effekt ist

$$\texttt{dst} \leftarrow n_1 + reg \times n_2$$

Hier muss $n_2 \in \{1, 2, 4, 8\}$ sein.

ix86 Assembler: Bitverschiebebefehle

Diese haben die Form

$$sop\ op,n$$

sop ist `sall` (left shift), `shrl` (right shift), `sarl` (arithmetic right shift).
Der Operand `op` wird um *n* Bits verschoben.

ix86: Labels und unbedingte Sprünge

Jeder Befehl kann durch Voranstellen eines Bezeichners mit Kolon markiert (*labelled* werden.

```
f:    subl    $main_framesize,%esp
L73:  movl    $35,%eax
```

Nach dem Kolon ist ein Zeilenumbruch erlaubt.

Ein unbedingter Sprungbefehl hat die Form

`jmp` *label*

ix86 Assembler: Bedingte Sprünge

Ein bedingter Sprung hat die Form:

`cmp` op_1,op_2

jop $label$

Hier ist *jop* eines von `je` (equal), `jne` (not equal), `jl` (less than), `jg` (greater than), `jle` (less or equal), `jge` (greater or equal), `jb` (unsigned less), `jbe` (unsigned less or equal), `ja` (unsigned greater), `jae` (unsigned greater or equal).

Es wird gesprungen, falls die entsprechende Bedingung zutrifft, ansonsten an der nächsten Instruktion weitergearbeitet.

ix86 Assembler: Funktionsaufruf

Die Instruktion

`call` *label*

springt nach *label* und legt die Rücksprungadresse auf den Keller, subtrahiert also 4 vom Stack Pointer.

Die Instruktion

`ret`

springt zur Adresse ganz oben auf dem Keller und pop-t diese.

Rückgabewerte sollten in `%eax` übergeben werden.

Zusätzlich kann man die Instruktionen `enter` und `leave` benutzen, die den Stackframe verwalten. Siehe Doku.

Ausführung von Assemblerprogrammen

Ein ausführbares Assemblerprogramm braucht ein Label `main` und die Direktive

```
.globl main
```

Bei `main` wird dann die Ausführung begonnen.

Mit `call` können andere Funktionen aufgerufen werden, z.B. in C geschriebene. Argumente für diese werden in `(%esp)`, `4(%esp)`, `8(%esp)` übergeben.

Um ein Assemblerprogramm `prog.s` mit einem C Programm, z.B. `library.c` zu verbinden führe man aus

```
cc -c library.c
cc -o runme prog.s library.o
```

Dann ist `runme` eine ausführbare Datei.

Abstrakte Assemblerinstruktionen

Wir verwenden eine abstrakte Klasse `Instr` (in Package `Assem`) von Assemblerbefehlen über `temporaries`. Jede Instruktion spezifiziert außerdem die Liste der Quell- und Zielregister.

```
package Assem;
import Temp.TempList;
public abstract class Instr {
  public String assem;
  public abstract TempList use();
  public abstract TempList def();
  public abstract Targets jumps();
  public abstract String format(Temp.TempMap m);
}
```

Wir verwenden konkrete Unterklassen mit den folgenden Konstruktoren:

```
public OPER(String a, Temp.TempList d, Temp.TempList s,
            Temp.LabelList j);
public OPER(String a, Temp.TempList d, Temp.TempList s);
```

Abstrakte Assemblerinstruktionen

```
public MOVE(String assem, Temp dst, Temp src);
public LABEL(String assem, Temp.Label label)
```

Im `assem` String kann man mit
`‘s0,‘s1,...,‘d0,‘d1,...,‘j0,‘j1,...` auf die Komponenten von `dst, src,jumps` Bezug nehmen.

Einen Additionsbefehl würde man so repräsentieren:

```
emit(new OPER("addl ‘s0 ‘d0\n",new TempList(t1,null),
                               new TempList(t1,new TempList(t2,null))))
```

Für die spätere Registerzuweisung ist es wichtig, Quell- und Zielregister richtig anzugeben.

Auch müssen bei Sprungbefehlen alle Sprungziele angegeben werden. Hierbei ist bei durchfallenden Sprungbefehlen die Folgeinstruktion einzuschließen.

Emission

`emit` ist eine Methode, die die Instruktion in eine globale Liste einhängt, die dann später weiterverarbeitet und schließlich ausgegeben wird.

Spezielle Register

Die besonderen Register `%eax, %ebp, %edx` sollten im Frame-Modul als solche repräsentiert werden, etwa `Frame.eax`, etc.

Man kann dann z.B. beim Divisionsbefehl `Frame.edx` zu den Zielregistern hinzufügen.

Bei der Registerverteilung werden diese dann auf die entsprechenden konkreten Register abgebildet.

Dreiadressbefehle

Allgemeine (Dreiadress) arithmetische Befehle können durch zusätzliche Erzeugung von Verschiebebefehlen simuliert werden. Etwa für $t_1 \leftarrow t_2 + t_3$ erzeugen wir drei Instruktionen:

```
Temp hilf = new Temp();
emit(new MOVE("movl ‘s0 ‘d0\n", [hilf], [t2]));
emit(new OPER("addl ‘s1 ‘d0\n", [hilf], [hilf,t3]));
emit(new MOVE("movl ‘s0 ‘d0\n", [t1], [hilf]));
```

Die spätere Phase der Registerverteilung ist in der Lage, redundante Verschiebebefehle wieder zu entfernen.

Funktionsaufrufe

Funktionsaufrufe können als `CALL` Instruktion übersetzt werden.

Es ist zu beachten, dass hierbei die caller-save Register überschrieben werden; man muss diese daher der `dst`-Liste hinzufügen, ebenso wie auch das Return-Register. Die Return Adresse wird beim Pentium auf dem Stack abgelegt, daher muss diese nicht über Register behandelt werden.

Auf der anderen Seite muss bei Funktionsrümpfen am Ende eine leere Instruktion angefügt werden, die als `src` alle callee-save Register, den Stackpointer und das Returnregister enthält, damit der Registerverteiler nicht denkt, diese würden nicht mehr gebraucht.

Dies kann durch Implementierung einer Methode

```
Assem.InstrList procEntryExit2(Assem.InstrList body);
```

geschehen.

Instruktionsauswahl

Wir schreiben zwei Methoden

```
Temp.Temp munchExp(Tree.Exp exp);
void munchStm(Tree.Stm stm);
```

Diese emittieren eine Folge von Assemblerinstruktionen, die dem übergebenen Argument entsprechen. Bei `munchExp` enthält das zurückgelieferte Temporary den Wert.

Beide Funktionen werden rekursiv implementiert.

Man wählt diejenige Assemblerinstruktion (oder Idiom von solchen, wie bei Dreiadressbefehlen), die von der Wurzel des übergebenen Baums das meiste abdeckt.

So kann etwa ein `leal`-Befehl eine Addition und eine Multiplikation auf einmal abdecken.

Auch kann ein Verschiebebefehl mit Speicheroperanden ein `MOVE` gefolgt von `MEM` abdecken.

Instruktionsauswahl

Solch eine Asssemblerinstruktion oder Folge von ihnen bezeichnen wir als *Kachel*, die eine oder mehrere IR-Befehle überdeckt.

Es gilt also den gegebenen Baum vollständig so mit Kacheln zu überdecken, dass möglichst wenig Kacheln verwendet werden.

Wir benutzen folgenden einfachen Greedy-Ansatz (“Maximal Munch”):

Wähle die größte (im Sinne der Zahl der überdeckten IR-Knoten) Kachel, die auf die Wurzel platziert werden kann.

Überdecke rekursiv die verbleibenden Kinder.

Konkret geht man die Kacheln der Größe nach in einer großen Fallunterscheidung durch; nimmt die erste, die passt, und tätigt dann rekursive Aufrufe auf den verbleibenden Teilausdrücken.

Dieser Algorithmus findet nicht immer die beste Kachelung. Diese gewinnt man mit dynamischer Programmierung.

Optimale Instruktionsauswahl mit dynamischer Programmieru

Man legt eine Tabelle an, in der für jeden Knoten des IR-Baumes die optimale Kachelung des dort beginnenden Teilbaumes abgelegt werden soll.

Die Tabelle wird bottom-up, oder aber rekursiv aufgefüllt. Für die Blätter ist es klar; für die höheren Knoten geht man der Reihe nach alle prinizipiell passenden Kacheln durch und berechnet jeweils die Größe der so (unter Verwendung der bereits berechneten optimalen Kachelungen der Unterbäüme) die entstehenden Kosten und wählt die beste aus.

Das Prinzip ist recht leicht; die Implementierung aber etwas aufwendig, weswegen es auch für diese Aufgabe codeerzeugende Werkzeuge gibt (“code generator generator”).

Diese lösen sogar eine etwas allgemeinere Aufgabe, die sich stellt, wenn Register nicht allgemein verwendbar sind.

Zusammenfassung

- Abstrakte Assemblerinstruktionen beinhalten Assemblerbefehle über Temporaries
- Sie erlauben eine Stringausgabe, geben aber auch Auskunft über die verwendeten und beschriebenen Temporaries, sowie Sprungziele. Diese Information wird für spätere Kompilerphasen gebraucht.
- Die Instruktionsauswahl erfolgt top-down rekursiv durch Wahl der jeweils größten (i.S.d. abgedeckten IR-Knoten) Assemblerinstruktion.
- Bessere Instruktionsauswahl lässt sich mit dynamischer Programmierung erreichen.

Aktivitätsanalyse (*liveness analysis*)

Ziel: Feststellen, welches Temporary wann aktiv ist ("*live*" ist).

- Aufstellen eines Kontrollflussgraphen (ähnlich wie Flussdiagramm).
- Temporary in einem Knoten aktiv, wenn sein aktueller Wert später gebraucht wird (genauer: werden könnte).
- Temporaries, die nie gleichzeitig aktiv sind, können auf dasselbe physikalische Register abgebildet werden.
- Bedingungen an die Vertauschbarkeit von Instruktionen können abgelesen werden.

Kontrollflussgraph

- Jede Instruktion (`Assem.Instr`) entspricht einem Knoten des Kontrollflussgraphen.
- Instruktionen, die keine Sprunganweisungen sind, haben die jeweils nächste Instruktion als *Nachfolger*
- Sprungbefehle haben ihre (ein oder zwei) Sprungziele als Nachfolger.
- Eine (gerichtete) Kante im Kontrollflussgraphen verbindet jeweils eine Instruktion mit ihren (ein oder zwei) Nachfolgern.
- Jede Ausführung des Programms entspricht also einem *Pfad* im Kontrollflussgraphen, aber nicht immer umgekehrt.

Datenstruktur für Graphen

```
package Graph
public class Graph {
  public List<Node> nodes();
  public Node newNode();
  public void addEdge(Node from, Node to);
  public void rmEdge(Node from, Node to);
  public void show(java.io.PrintStream out);
}
```

Man könnte schon an dieser Stelle mit Knoten und Kanten Information assoziieren, also etwa

```
package Graph
public class Graph<NodeInfo,EdgeInfo> {
  public void addEdge(Node from, Node to, EdgeInfo);
  ... }
```

Im Buch wird empfohlen, hierfür externe Hashtabellen zu verwenden.

```
public class Node {
    public Node(Graph g);
    public List<Node> succ(); public List<Node> pred();
    public List<Node> adj();
    public int inDegree(); public int outDegree(); public int degr
    public boolean goesTo(Node n);
    public boolean comesFrom(Node n);
    public boolean adj(Node n);
    public String toString();
}
```

Nach der “Buchmethode” wäre also ein Graph, dessen Knoten mit Instruktionen beschriftet sind, so zu realisieren:

```
class GraphWithInstr {
   Graph graph;
   Map<Node,Assem.Instr> nodemap;
}
```

Kontrollflussgraph

Der Kontrollflussgraph ist also ein Graph, dessen Knoten mit Instruktionen beschriftet sind.

Das Aufstellen des Kontrollflussgraphen bewerkstelligt eine Methode, die aus einer Liste von Instruktionen solch einen Graphen berechnet.

Beachte: an dieser Stelle benutzen wir die in den abstrakten Assemblerinstruktionen enthaltenen Sprungziele.

Aktivität (Liveness)

Ein Temporary r wird in einem Knoten n des Kontrollflussgraphen *definiert* (def), wenn die dort stehende Instruktion das Temporary r beschreibt.

Ein Temporary r wird in einem Knoten n des Kontrollflussgraphen *benutzt* (use), wenn die dort stehende Instruktion das Temporary r liest.

Jede abstrakte Assemblerinstruktion beinhaltet bereits eine Liste der benutzten und der definierten Temporaries.

Ein Temporary r ist entlang einer Kante e des Kontrollflussgraphen aktiv (*live*), wenn es einen Pfad im Kontrollflussgraphen durch e gibt, der auf eine Benutzung von r führt, ohne vorher durch eine Definition von r zu führen.

Der Wert des Temporaries ist also entlang einer solchen Kante von Belang.

Beispiel

$$
\begin{array}{lll}
 & a \leftarrow 0 & \\
L_1: & b \leftarrow a + 1 & \\
 & c \leftarrow c + b & \\
 & a \leftarrow b * 2 & \\
 & \text{if } a < N \text{ goto } L_1 & \\
 & \text{return } c &
\end{array}
\qquad
\begin{array}{lll}
 & a \leftarrow 0 & \\
L_1: & a \leftarrow a + 1 & \\
 & c \leftarrow c + a & \\
 & a \leftarrow a * 2 & \\
 & \text{if } a < N \text{ goto } L_1 & \\
 & \text{return } c &
\end{array}
\qquad
\begin{array}{lll}
 & b \leftarrow 0 & \\
L_1: & b \leftarrow b + 1 & \\
 & c \leftarrow c + b & \\
 & b \leftarrow b * 2 & \\
 & \text{if } b < N \text{ goto } L_1 & \\
 & \text{return } c &
\end{array}
$$

Die Variablen a und b haben disjunkte Aktivitätsbereiche und können daher zusammengefasst werden.

Aktivität in Knoten

Ein Temporary r ist in einem Knoten n des Kontrollflussgraphen *ausgangsaktiv* (*live-out*), wenn es eine Kante e von n aus gibt, entlang der r aktiv ist.

Mit anderen Worten gibt es einen Pfad durch n, der auf eine Benutzung führt ohne vorher durch eine Definition von r zu führen. Der Knoten n selbst ist hierbei jeweils *ausgeschlossen*.

Ein Temporary r ist in einem Knoten n des Kontrollflussgraphen *eingangsaktiv* (*live-in*), wenn es eine Kante e nach n gibt, entlang der r aktiv ist.

Mit anderen Worten gibt es einen Pfad durch n, der auf eine Benutzung führt ohne vorher durch eine Definition von r zu führen. Der Knoten n selbst ist hierbei jeweils *eingeschlossen*.

Berechnung der Aktivität

Man kann für jedes Temporary r die Menge derjenigen Knoten in denen es ausgangsaktiv ist direkt aus der Definition berechnen.

Hierzu verfolgt man ausgehend von einer Benutzung von r die Kanten im Kontrollflussgraphen *rückwärts* und nimmt alle besuchten Knoten hinzu, bis man schließlich auf eine Definition von r stößt.

Für diese Rückwärtsverfolgung lässt sich die Tiefensuche verwenden.

Natürlich muss dies für jede Benutzung von r separat gemacht werden, wodurch sich möglicherweise ein recht hoher Aufwand ergibt.

Eine Alternative liegt in der gleichzeitigen Berechnung der eingangs- und ausgangsaktiven Temporaries für jeden Knoten.

Berechnung der Aktivität

Man kann für jeden Knoten die Menge der eingangs- und ausgangsaktiven Temporaries aufgrund folgender Beobachtungen iterativ berechnen:

- Wird ein Temporary r in n *benutzt*, so ist es in n eingangsaktiv.
- Ist r in einem Nachfolger von n eingangsaktiv, so ist es in n selbst ausgangsaktiv.
- Ist r ausgangsaktiv in n und wird in n nicht definiert, so ist r auch eingangsaktiv in n.

In Zeichen:

$$\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] \setminus \mathit{def}[n])$$
$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}[n]} \mathit{in}[s]$$

Die $\mathit{in}[n]$ und $\mathit{out}[n]$ Mengen bilden die kleinste Lösung dieser Gleichung und können mit deren Hilfe iterativ berechnet werden:

Algorithmus

for each $n \quad in[n] \leftarrow \emptyset; \quad out[n] \leftarrow \emptyset;$

repeat

  for each n

    $in'[n] \leftarrow in[n]; \qquad out'[n] \leftarrow out[n];$

    $in[n] \leftarrow use[n] \cup (out[n] \setminus def[n]);$

    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s];$

until $in'[n] = in[n]$ **and** $out'[n] = out[n]$ **for all** n

Es empfiehlt sich, die “for each” Schleifen so zu durchlaufen, dass Nachfolgerknoten möglichst vor ihren Vorgängern verarbeitet werden. Durch Tiefensuche im umgedrehten Kontrollflussgraphen kann solch eine Reihenfolge bestimmt werden.

Interferenzgraph

Aus der Aktivitätsinformation kann ein Interferenzgraph berechnet werden.

Zwei Temporaries t_1, t_2 *interferieren* wenn es nicht zulässig ist, sie auf ein und dasselbe physikalische Register abzubilden.

Gründe für Interferenz:

1. t_1 wird in einem Knoten n definiert und gleichzeitig ist t_2 in t_1 ausgangsaktiv.
2. t_1 ist ein Maschinenregister und t_2 darf aus anderen Gründen nicht auf t_1 abgebildet werden.

Von der ersten Bedingung gibt es noch eine wichtige Ausnahme:

t_1 und t_2 interferieren nicht, wenn ihre Aktivitätsbereiche *nur* wegen einer Verschiebeinstruktion $t_i \leftarrow t_{3-i}$ überlappen. Dann kann man sie nämlich doch zusammenfassen und die Verschiebeinstruktion entfernen!

Die zweite Bedingung ist für uns weniger relevant.

Algorithmus

Diese Betrachtungen führen auf folgenden Algorithmus:

for each n

  if n *enthält keine Verschiebeinstruktion*

    for each $t \in$ *def* $[n]$

      for each $u \in$ *out* $[n]$

        füge Interferenzkante (t, u) *ein*

  else if n *enthält Verschiebeinstruktion* $a \leftarrow c$

      for each $u \in$ *out* $[n] \setminus \{c\}$

        füge Interferenzkante (t, u) *ein*

Programmierprojekt

- Implementierung der Datenstrukturen für Graphen.
- Aufstellung der Kontrollflussgraphen (separat für jede Prozedur).
- Berechnung der Aktivitätsmengen.
- Berechnung des Interferenzgraphen für jede Prozedur.

Registerverteilung

Ziel: Verteilung der Temporaries auf physikalische Register, bzw. Adressen im Frame (spilling).

Dabei:

- Zusammenfassung von Temporaries, wenn möglich.
- Elimination von Verschiebeanweisungen.
- Verwendung von hauptspeicheradressierenden Instruktionen bei *spilling*.
- Minimierung des Spilling.

Registerzuweisung als Graphenfärbung

Lassen wir einmal Spilling außer acht, nehmen wir also an, wir könnten alle Temporaries auf Register abbilden.

Eine legale Registerzuweisung weist jedem Knoten des Interferenzgraphen ein Register zu, so dass durch eine Kante verbundene Knoten verschiedene Register zugewiesen bekommen.

Es geht also darum, den Interferenzgraphen mit den Registern zu *färben*. Die Register sind also die Farben.

Graphenfärbung ist i.a. NP-vollständig (siehe Info IV). Für die Interferenzgraphen, die von strukturierten Programmen herrühren, ist das Problem aber in P. Für unsere Zwecke genügt eine einfache Heuristik von Kempe (1879):

Heuristik für Graphenfärbung

Um einen Graphen G mit K Farben zu färben gehe wie folgt vor:

- Hat G einen Knoten v vom Grad $< K$ also mit weniger als K Nachbarn, so entferne v und färbe $G \setminus \{v\}$ rekursiv. Gelingt das, so kann auch ganz G gefärbt werden, da ja noch mindestens eine Farbe für v frei ist.
- Hat G nur noch Knoten vom Grad $\geq K$, so antworte "nicht färbbar".

Alternativ könnte man im auch zweiten Fall einen beliebigen Knoten entfernen und dann hoffen, dass die Nachbarn mit weniger als K Farben gefärbt werden.

Spilling: Was ist, wenn der Interferenzgraph nicht färbbar ist?

Kann der Interferenzgraph nicht mit K Farben (K die Anzahl der Register) gefärbt werden, so müssen einige Temporaries in den Frame ausgelagert werden (spilling). Dazu verfeinert man obige Heuristik wie folgt:

- Hat G einen Knoten v vom Grad $< K$ also mit weniger als K Nachbarn, so entferne v und färbe $G \setminus \{v\}$ rekursiv. Gelingt das, so kann auch ganz G gefärbt werden, da ja noch mindestens eine Farbe für v frei ist.
- Hat G nur Knoten vom Grad $\geq K$, so entferne einen beliebigen Knoten v und markiere ihn als "Spill-Kandidat" und färbe den verbleibenden Graphen rekursiv (durch das Entfernen des Spillkandidaten können nun wieder Knoten mit Grad $< K$ entstanden sein, ansonsten sind weitere Knoten als Spillkandidaten zu benennen)
- Wurden die Nachbarn eines Spillkandidaten mit weniger als K Farben gefärbt, so kann man diesen doch noch färben; er ist dann kein

Spilling: Was ist, wenn der Interferenzgraph nicht färbbar ist?

Spillkandidat mehr.

- Verbleiben nach Ablauf des Algorithmus noch Spillkandidaten, so müssen die Spillkandidaten in den Frame ausgelagert werden; hierzu muss das Maschinenprogramm entsprechend umgeschrieben werden. Die gesamte Prozedur einschließlich Aktivitätsanalyse ist dann erneut durchzuführen.
- Dies alles wird solange wiederholt, bis keine Spills mehr auftreten.

Verschmelzung (*Coalescing*)

Nach der Registerverteilung kann der Code redundante Verschiebeinstruktionen der Form $r \leftarrow r$ enthalten. Diese können natürlich entfernt werden.

Unter Umständen ist es sinnvoll, das Auftreten dieser Situation dadurch zu begünstigen, dass man schon vor der Registerverteilung nicht-interferierende Knoten a und b verschmilzt, sofern zumindest eine Verschiebeinstruktion $a \leftarrow b$ vorhanden ist.

Dadurch erhöht sich aber i.a. der Knotengrad und zusätzliche Spills könnten resultieren, in diesem Fall muss das Verschmelzen unbedingt vermieden werden.

Dieser ungünstige Fall tritt sicher dann *nicht* auf, wenn eine der folgenden Bedingungen zutrifft:

Heuristiken für die Verschmelzung

Briggs: Knoten a und b können verschmolzen werden, wenn der resultierende Knoten ab weniger als K Nachbarn vom Grad $\geq K$ hat.

George: Knoten a und b können verschmolzen werden, wenn jeder Nachbar t von a entweder mit b interferiert, oder aber Grad $< K$ hat.

Man überlegt sich, dass das Verschmelzen in diesen Fällen keine Verschlechterung der Färbbarkeit nach sich zieht.

Es empfiehlt sich, während des Verlaufs der Registerverteilungsprozedur immer wieder zu prüfen, ob Verschmelzung nach diesen Kriterien möglich ist. Schließlich könnte ja durch das sukzessive Entfernen von Knoten eine der Bedingungen erfüllt werden.

Spilling

Die verbleibenden Spillkandidaten sind im Frame unterzubringen.

Allerdings sind auch diese i.a. nicht alle gleichzeitig aktiv, so dass sie aggressiv verschmolzen (bei Vorhandensein einer entsprechenden Verschiebeinstruktion) und sodann mit möglichst wenigen Farben gefärbt werden.

Hierzu entfernt man jeweils einen Knoten mit minimalen Grad und färbt den Restgraphen.

Die verwendeten Farben entsprechen dann Positionen im Frame.

Vorgefärbte Knoten

Die Knoten, die den Maschinenregistern entsprechen, sind vorab bereits gefärbt. Sie interferieren miteinander und können nicht in den Frame ausgelagert werden.

Es empfiehlt sich, deren Aktivitätsbereich durch sofortiges Verlagern in andere Temporaries vorab zu minimieren.

Die entsprechenden Verschiebeinstruktionen können ja später wieder entfernt werden, sofern sie am Ende nicht benötigt werden.

Zusammenfassung

Prozedur “Färben”:

Eingabe: Interferenzgraph.

Ausgabe: Liste von Spill-Knoten, Liste von verschmolzenen Knotenpaaren, Färbung des Interferenzgraphen (Spill-Knoten sind nicht gefärbt, bei verschmolzenen Paaren ist nur eine Komponente gefärbt).

Verwendung: Bilde Interferenzgraph, führe Färbung durch. Ist die Spill-Liste leer, so verteile Register entsprechend der Färbung, berücksichtige verschmolzene Knoten, entferne redundante Verschiebeinstruktionen.

Anderenfalls verteile Spill-Knoten auf Framepositionen, verschmelze aggressiv und minimiere Zahl der Framepositionen.

Identifiziere Möglichkeiten für hauptspeicheradressierende Instruktionen (optional).

Wende gesamte Prozedur auf entstehenden Code erneut an.

Prozedur “Färben” (rekursive Version)

Eingabe: Interferenzgraph. Knoten, die Teil einer Verschiebeinstruktion sind als “verschmelzbar” zu markieren.

Ausgabe: Spill-Liste, Liste von verschmolzenen Knotenpaaren, Färbung.

Simplify Entferne nicht “verschmelzbaren” Knoten von Grad $< K$ und färbe den Restgraphen. Färbe dann den entfernten Knoten passend.

Coalesce Ist kein nicht “verschmelzbarer” Knoten von Grad $< K$ vorhanden, so verschmelze Knoten gemäß der Briggs oder George Heuristik. Färbe den resultierenden Graphen.

Freeze Applizieren weder Simplify, noch Coalesce, so deklariere einen “verschmelzbaren” Knoten von Grad $< K$ als “nicht verschmelzbar” und färbe den resultierenden Graphen.

Spill Gibt es überhaupt keinen Knoten von Grad $< K$, so entferne einen beliebigen Knoten und färbe den Restgraphen. Haben die Nachbarn des entfernten Knoten weniger als K Farben, so färbe ihn entsprechend,

Prozedur “Färben” (rekursive Version)

anderenfalls füge ihn der Spill-Liste hinzu.