Prof. Dr. M. Jäger FB MNI

Praktikumshinweise Compilerbau SS 2018

Laufzeitstack-Organisation für SPL

Unterprogrammaufrufe in SPL

SPL hat ein einfaches Unterprogrammkonzept, das die Definition beliebig vieler Prozeduren auf der globalen Ebene erlaubt. Ineinander verschachtelte Prozedurdefinitionen sind nicht möglich. Da es in der Sprache auch keine globalen Variablen gibt, kann eine Prozedur ausschließlich auf ihre Parameter und ihre lokalen Variablen zugreifen.

Es gibt zwei Arten der Parameterübergabe: Wertübergabe („call by value“) und Referenzübergabe („call by reference“).

Begriffe

Im Zusammenhang mit Unterprogrammen (Prozeduren, Funktionen, Methoden) muss zwischen den Parametern unterschieden werden, die als Teil der Unterprogrammdeklaration definiert werden, und den Argumenten, die an der Aufrufstelle angegeben werden. Erstere werden manchmal als *formale Parameter* und letztere als *aktuelle Parameter* bezeichnet. Wir benutzen im folgenden stattdessen die Begriffe „Parameter“ (formale Parameter) und „Argumente“ (aktuelle Parameter).

Bei Unterprogrammaufrufen gibt es eine klare Rollenverteilung. Das aufrufende Unterprogramm bezeichnen wir im folgenden als *Caller*, das aufgerufene Unterprogramm als *Callee*.

Parameterübergabeverfahren

Die Deklaration der Parameter legt den *Typ* und das *Parameterübergabeverfahren* fest.

Beispiel:

```
proc modulo ( op1: int, op2: int, ref result: int) {
   result := op1 - op1 / op2 * op2;
}
```

Die Parameter von *modulo* sind *op1*, *op2* und *result*. Die Parametertypen sind alle `int`. Die ersten beiden Parameter sind *Wertparameter*: Ein Wertparameter ist eine lokale Variable des Callee, die vom Caller mit einem Initialwert versorgt wird. Als Argument kann an der Aufrufstelle ein beliebiger Ausdruck angegeben werden. Zum Aufrufzeitpunkt wird dessen Wert berechnet und im Parameter gespeichert. Eine Wertänderung von *op1* durch *modulo* würde nichts am Wert von *i* ändern.

Der Parameter *result* ist ein *Referenzparameter*: Ein Referenzparameter ist, vereinfacht gesagt, ein vom Callee benutzter Verweis auf eine Variable des Caller. Im Beispiel hat *main* eine lokale Variable *i* und übergibt beim Aufruf von *modulo* einen Verweis auf *i* als drittes Argument. Genauer betrachtet ist ein Referenzparameter eine lokale Variable der aufgerufenenen Prozedur, die ähnlich wie eine Zeigervariable (Pointer) genutzt wird. Sie wird bei Aufruf mit der Adresse der Argument-Variablen initialisiert.

Aufrufbeispiel:

```
proc main () {
   var i: int;
   i := 21;
   modulo(i, i-16, i);
   printi(i);
}
```

Die Argumente sind *i*, *i-16*, und *i*. Der erste Parameter von *modulo* (*op1*) wird mit 21 (Wert von *i*) initialisiert, der zweite (*op2*) mit 5. Da der dritte Parameter (*result*) ein Referenzparameter ist, wird nicht der Wert von *i*, sondern die Adresse von *i* an *modulo* übergeben. Dadurch kann *modulo* die lokale Variable *i* von *main* verändern: Im Beispielaufruf wird nach der Rückkehr *i* den Wert 1 (21 modulo 5) haben.

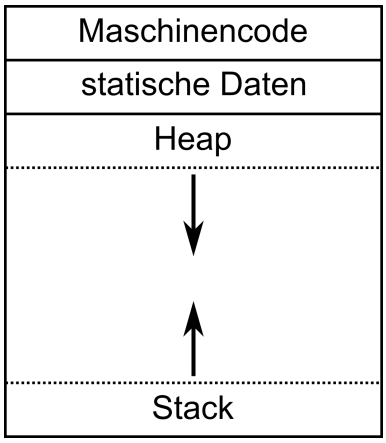
Den Bezug von Referenzparametern zu Pointervariablen verdeutlicht die entsprechende C-Implementierung. Da C keine Referenzparameter kennt, müssen stattdessen Pointer verwendet werden.

```
#include <stdio.h>

void modulo ( int op1, int op2, int *result) {
       *result = op1 - op1 / op2 * op2;
}

int main () {
       int i = 21;
       modulo(i, i-16, &i);
       printf("%d\n", i);
    }
```

Im Gegensatz zu der SPL-Version muss in der C-Implementierung beim dritten Argument im Aufruf von *modulo* der explizite Adressoperator verwendet werden, um die Adresse von *i* zu übergeben. Auch innerhalb von *modulo* ist der Unterschied sichtbar: Um auf das *i* von *main* zugreifen zu können, muss in C der Dereferenzierungsoperator verwendet werden (`*result`), der beim Referenzparameter in der SPL-Variante weggelassen wird. Auf der Maschinenebene sind beide Lösungen vollkommen äquivalent, die Unterschiede liegen nur auf der Quelltextebene.

Hauptspeicher-Layout und Laufzeitstack

Der Hauptspeicher eines Prozesses ist (vereinfacht) wie folgt in logische Teilbereiche (Segmente) gegliedert:



Für SPL wird der Heap nicht benutzt.

Der Stack enthält für jeden aktiven Unterprogramm-Aufruf einen Eintrag, den Aktivierungsrahmen (auch „activation record“ oder „frame“) des Unterprogramm-Aufrufs. Dieser Speicherbereich bietet Platz für die Speicherung der Parameter, der lokalen Variablen und aller sonstiger Werte die mit einem Unterprogrammaufruf assoziiert sind. Detail dazu folgen weiter unten.

Bei jedem Unterprogramm-Aufruf wird ein neuer Rahmen auf dem Stack reserviert und beim Rücksprung wieder freigegeben. Die Anordnung der Aktivierungsrahmen gibt die Aufruf-Verschachtelung zu einem bestimmten Zeitpunkt wieder.

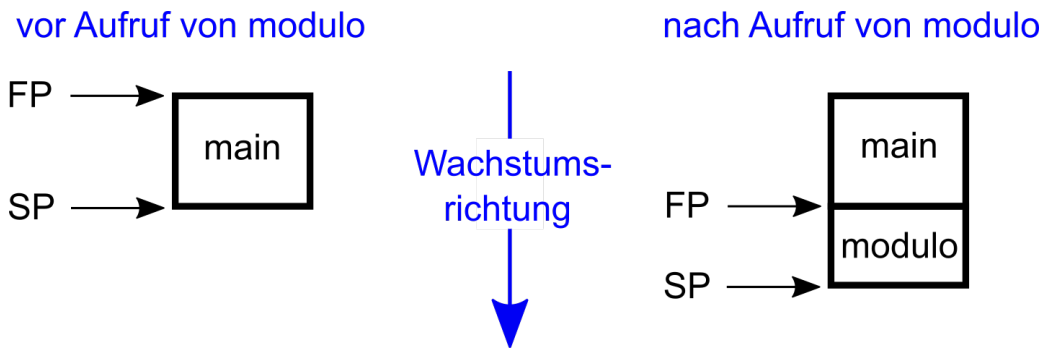
Stackpointer und Framepointer

Damit eine Prozedur ihren Aktivierungsrahmen findet, werden zwei Zeiger genutzt:

1. Stackpointer (SP): Der SP verweist auf das „obere“ Ende des Stacks und damit gleichzeitig auf das Ende des aktuellen Rahmens.
2. Framepointer (FP): Der FP verweist auf den Beginn des aktuellen Rahmens.

Für SP und FP werden zwei Register des Prozessors verwendet (Puck VM: SP=$31, FP=$25). Die Adressen der in einem Rahmen gespeicherten Daten können jeweils in der Form *FP+Offset* oder *SP+Offset* angegeben werden. *Offset* kann vom Compiler bestimmt werden. SP und FP müssen bei jedem Aufruf und jedem Rücksprung aktualisert werden. Dazu muss der Compiler Maschinenbefehle generieren.

Stack zum Beispiel von oben:



Aus dem Gesamtlayout des Adressraums wird deutlich, dass der Stack in Richtung kleinerer Adressen wächst: Der Wachstumsrichtungspfeil in der letzten Abbildung zeigt nach unten, in der Abbildung von Seite 2 allerdings nach oben. SP ist also immer kleiner als FP, genauer gesagt gilt im Beispiel nach dem Aufruf von *modulo* :

$$SP = FP - Rahmengroesse(modulo)$$

Die Rahmengröße einer Prozedur wird vom Compiler berechnet. Beim Aufruf von *modulo* werden die beiden Verweise also offensichtlich wie folgt aktualisiert:

```
FP := SP ;
SP := FP - Rahmengröße(modulo) ;
```

Es stellt sich die Frage, ob Caller oder Callee für die Aktualisierung sorgen. In Sprachen mit *getrennter Übersetzung* (z.B. C) kann der Compiler einzelne Komponenten eines Programms auch dann übersetzen, wenn andere Komponenten noch nicht vorliegen. Auf unser Beispiel übertragen heißt das: Die Übersetzung von *main* muss ohne genaue Kenntnis von *modulo* möglich sein. Insbesondere kennt der Compiler beim Übersetzen des Callers i.A. die Rahmengröße des Callee nicht und umgekehrt. Daher wird wie folgt verfahren:

Der Callee ist für die Reservierung und die Freigabe seines Rahmens zuständig. Wenn der Compiler allerdings bei der Übersetzung von *modulo* den Maschinencode für die Freigabe des Rahmens vor Rücksprung nach *main* erzeugt, sind folgende Aktionen nötig:

```
SP := FP ;
FP := SP + Rahmengröße (main) ;
```

Das Problem liegt hier in der Bestimmung des alten FP, denn die Rahmengröße von *main* ist bei der Übersetzung von *modulo* i.A. nicht bekannt. Daher wird wie folgt verfahren:

Bei Reservierung eines neuen Rahmens wird der alte FP gerettet, d.h. in den neuen Rahmen kopiert (FP-alt). Bei Freigabe des Rahmens wird dann diese Kopie verwendet, um den FP auf den alten Wert zu restaurieren.

Aufruf von modulo:

```
FP-alt := FP;
FP := SP ;
SP := FP - Rahmengröße(modulo) ;
```

Rücksprung zu main:

```
SP := FP ;
FP := FP-alt ;
```

Man beachte, dass FP-alt ein Speicherplatz im Callee-Rahmen ist, sodass bei verschachtelten Aufrufen zu jeder Aufrufebene ein eigener FP-alt-Wert existiert.

Rücksprungadressen

Sowohl der Unterprogrammaufruf als auch die Rückkehr werden auf der Maschinenebene durch Sprungbefehle realisiert. Während der Compiler bei der Übersetzung des Callers die Zieladresse kennt, ist bei der Übersetzung des Callees die Aufrufstelle und damit die Zieladresse für den Rücksprung nicht bekannt.

Vielmehr wird die Rücksprungadresse zum Aufrufzeitpunkt vom Aufrufer ähnlich wie ein Argument an den Callee übergeben. Bei SPL wird dazu das Register $30 (RETURN) verwendet. Im Caller muss die Ausführen nach dem Rücksprung mit dem Maschinenbefehl fortgesetzt werden, der unmittelbar hinter dem Sprung in den Caller-Code steht. Spezielle Sprungbefehle (Puck VM: CALL und CALLR) setzen RETURN auf den korrekten Wert und schreiben dann das Sprungziel in den Programmzähler (PC) des Prozessors.

Wenn der Callee nun seinerseits eine Prozedur aufruft, wird das RETURN-Register überschrieben. Damit der Callee trotzdem korrekt zurückspringen kann, muss er den alten Wert vorher retten (in seinen Rahmen kopieren). Mit dem geretteten Wert kann er vor dem Rücksprung das RETURN-Register restaurieren. Das Retten und Restaurieren entfällt, wenn der Callee selbst keine Unterprogrammaufrufe durchführt.

Rahmen-Layout

Informationen, die im Rahmen eines Unterprogrammaufrufs zwischen Caller und Callee ausgetauscht werden müssen, können prinzipiell sowohl auf dem Laufzeitstack als auch in den Registern des Prozessors gespeichert werden. Die gleiche Wahl besteht auch für lokale Variablen sowie für Zwischenergebnisse, die bei der Berechnung komplexer Ausdrücke gespeichert werden müssen.

Für den SPL-Compiler werden folgende Konventionen festgelegt:

1. **Argumente**

   Alle Argumente stehen im Hauptspeicher. Die Argumente für den Callee stehen am Ende des Caller-Rahmens in umgekehrter Reihenfolge: Das erste Argument steht ganz am Ende des Rahmens. Falls der Caller unterschiedliche Callees aufruft, wird der maximal benötigte Speicher für die Argumente ermittelt und am Rahmenende reserviert. Die Rahmengröße ist fix.

2. **lokale Variablen**

   Alle lokalen Variablen stehen im Hauptspeicher am Anfang des Aktivierungsrahmens.

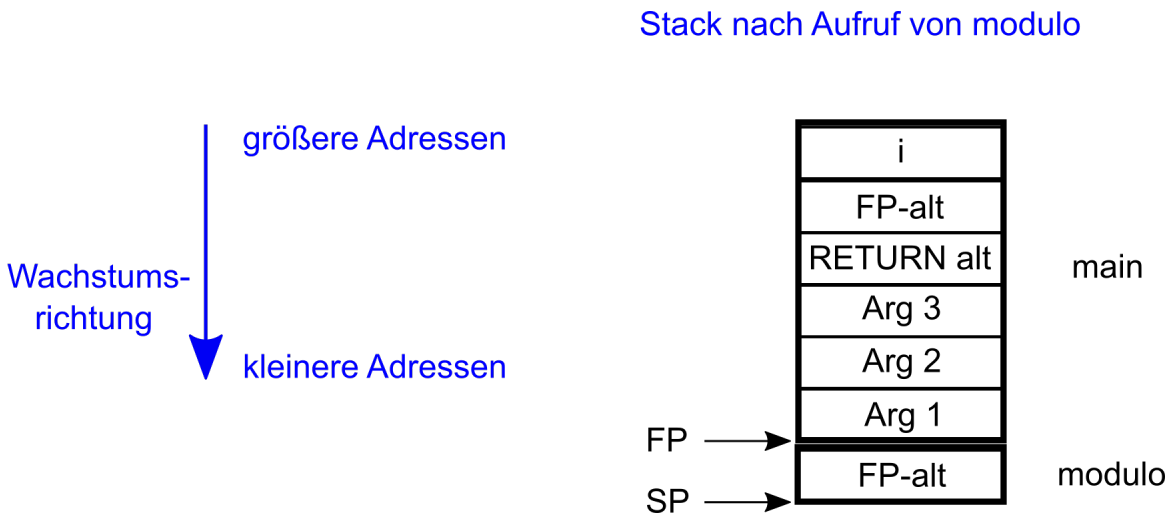
3. **gerettete Registerinhalte**

   - hinter den lokalen Variablen steht FP-alt
   - hinter FP-alt steht RETURN-alt, falls RETURN gerettet werden muss

4. **Zwischenergebnisse**

   Alle Zwischenergebnisse stehen in Mehrzweckregistern. Wenn diese nicht ausreichen, bricht die Programmausführung mit einer Fehlermeldung ab.

Im Beispiel hat der Callee *modulo* keine lokalen Variablen und enthält keine Unterprogrammaufrufe. Daher besteht der Rahmen nur aus dem Speicherplatz für FP-alt.



Adressierung der Rahmeninhalte

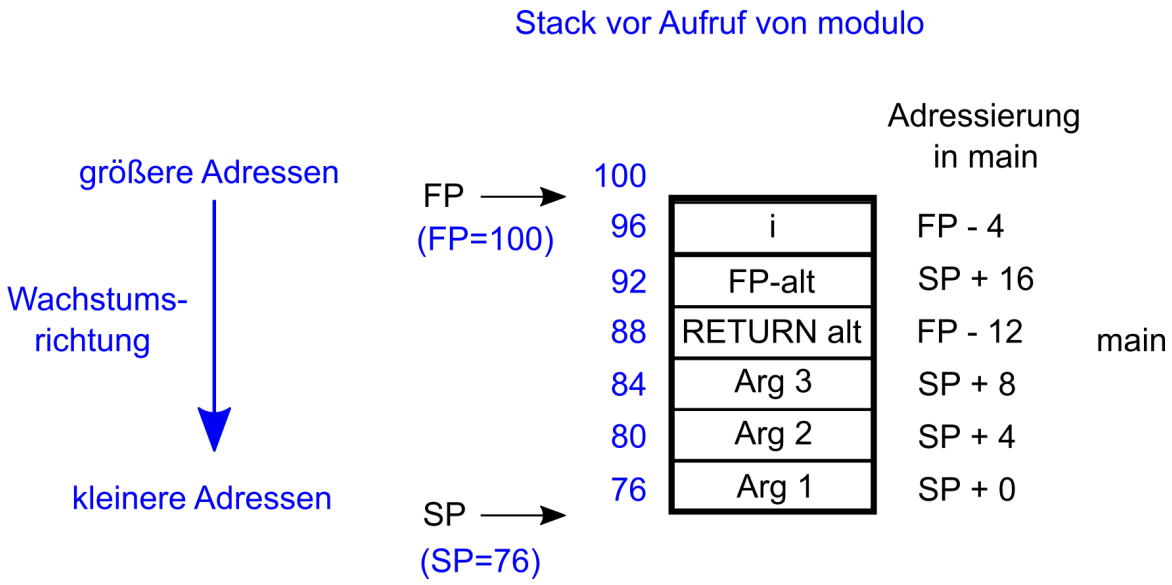
Für die Adressierung der Argumente verwendet der Caller SP als Bezugspunkt und der Callee FP. Da Caller-SP und Callee-FP gleich sind, gelten die gleichen Offset-Werte. Das erste Argument hat

den Offset 0, weitere Argumente haben positive Offsets. Die lokalen Variablen werden relativ zu FP adressiert. Alle lokalen Variablen haben negative Offsets. FP-alt wird relativ zu SP adressiert (Offset positiv), RETURN-alt relativ zu FP (Offset negativ).

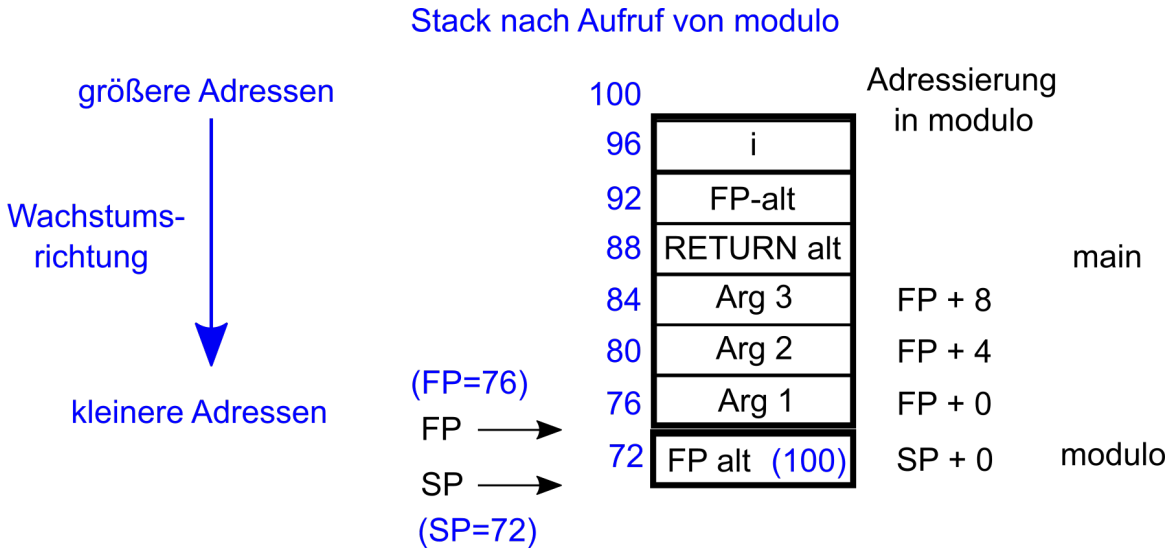
Ein Adressbeispiel:

Die Speicherplätze für Integer-Werte, Referenzen und gerettete Register sind jeweils 4 Byte groß. Die Größe für Array-Variablen ist das Produkt aus Komponentenanzahl und Größe der Komponenten.

Annahme: Der Frame von *main* „beginnt“ bei Adresse 100, d.h. vor dem Aufruf vom modulo ist FP=100. Da er 6 Speicherplätze a 4 Byte enthält, ist er 24 Byte groß. Wegen des Stack-Wachstums in Richtung kleiner werdender Adressen ist der Wert des FP tatsächlich die Adresse des letzten Byte vor dem main-Rahmen. Der main-Rahmen selbst erstreckt sich von Byte 76 bis Byte 99. Die lokale Variable *i* mit der Adresse 96 steht auf dem ersten Speicherplatz im Rahmen und belegt die Bytes 96-99. SP verweist auf das Ende des Rahmens. Dort ist Platz für das erste Argument für *modulo* in den Bytes 76-79.



Nach dem Aufruf von *modulo* sieht das Bild wie folgt aus:



Variablen-Allokation

Vor der Codegenerierung muss der Compiler für alle Parameter und Variablen die Offsets bestimmen. Ebenso muss die Framegröße für alle Prozeduren berechnet werden und das Frame-Layout:

a) **Speicherbedarf für Typen bestimmen**

   Der Speicherbedarf für eine Variable oder einen Parameter hängt vom Typ ab und wird pro Typ nur einmal berechnet. Das Ergebnis wird im Typ gespeichert (*byteSize*). Diese bei komplexen Typen rekursive Berechnung kann innerhalb der Typkonstruktoren erfolgen.

b) **Parameter-Offsets bestimmen**

   Die Offsets der Argumente bzw. Parameter werden sowohl für die Aufrufstelle als auch für den Zugriff innerhalb des Callees benötigt. Daher müssen sie in der lokalen Symboltabelle (für Callee) und in der Parametertypliste (für Caller) abgespeichert werden. Dies lässt sich z.B. in einer Schleife, ausgehend vom AST-Knoten für die Parameterdeklarationen bewerkstelligen. Am Ende der Schleife ist die Gesamtgröße für die Parameter bekannt und wird in der Symboltabelle der Prozedur gespeichert.

c) **Variablen-Offsets bestimmen**

   Die Offsets für die Variablen müssen in die lokale Symboltabelle eingetragen werden. Dies lässt sich z.B. in einer Schleife, ausgehend vom AST-Knoten für die Variablendeklarationen bewerkstelligen. Am Ende der Schleife ist die Gesamtgröße für die Variablen bekannt und wird gespeichert.

d) **Speicherbedarf für Argumentbereich bestimmen**

   In einem zweiten Durchgang kann dann zu jeder Prozedur die Größe des Argumentbereichs am Rahmenende bestimmt werden: Anhand des AST-Knotens für den Prozedurrumpfs ermittelt der Compiler alle darin vorkommenden Aufrufe. Zu jedem Aufruf wird aus dem Symboltabelleneintrag des Callee die benötigte Speichergröße für die Argumente entnommen. Das Maximum für alle Aufrufe wird bestimmt und für den Caller gespeichert.

   Wenn keine Aufrufe gefunden werden, muss der Compiler sich das merken (z.B. outgoingAreaSize = -1), denn in diesem Fall wird später vom Code-Generator das kurze Rahmen-Layout verwendet (ohne RETURN-alt).

Für die Implementierungs der am AST orientierten Algorithmen bietet sich wieder das Visitor-Muster an.