Compiler Construction and Formal Languages
Exercises

Berthold Hoffmann
Informatik, Universität Bremen, Germany
hof@informatik.uni-bremen.de

January 2013

Abstract

By doing the following exercises, the students attending the course *Compiler Construction and Formal Languages* shall apply the concepts and algorithms taught in the course. Some of these exercises will be indicated as assignments, to be done by students, either alone or in groups of at most two, in order to get *credit points* and *marks* for the course. Some of the other exercises will be done in class.

1 Static or Dynamic?

A property P of a programming language is called

- *static* if it can be checked already while the program is compiled, and
- *dynamic* if it can, in general, only be checked at the time when the compiled program is executed.

Please consider the programming languages C, Java, and Haskell (or other languages that you know). Find out which of the following properties is static, and dynamic in these languages, respectively:

1. The *binding* of names (identifiers) to the declarations introducing them.
2. The type of a variable.
3. The bounds of an array.
4. The value of a variable.

Please compare the languages wrt. your findings.

2 Bootstrapping

Implementing Java

The language Java is implemented by a compiler and an interpreter:

- The compiler transforms Java programs into *Java Byte Code* (JBC).
- JBC-programs are executed by an interpreter, known as the *Java Virtual Machine* (JVM).

A Portable Java Implementation

Portable implementations of Java – like the *Java Developer Kit* by SUN systems are based on the following components:

1. A master compiler from Java to JBC, written in Java itself. This compiler can be assumed to use only a small portion of the JVM library.
2. A compiler from Java to JBC, written JBC, which is derived from the master compiler. (You get this compiler for free as soon as you are able to execute Java on some platform.)
3. A *Java Virtual Machine*, an interpreter for JBC, written in Java.

Tasks

1. *Port to machine M.* How can Java be ported to a new machine M, given the components above? Do this by implementing a component that is as simple and small as possible.

   You may assume that a native C-compiler, producing M machine code, and "written" in M machine code, exists for M. The resulting implementation need not run efficiently – it may involve an interpreter.
2. *Bootstrap on machine M.* How can the compiler be made native? Again, do this by implementing a small component.

3 Regular Definitions

3.1 Identifiers

shall be formed according to the following rules:

1. Identifier are strings (of arbitrary length) that consist of letters, digits, and "_" (underscore).
2. Identifiers start with a letter.
3. An underscore may appear only between two letters or digits.[1]

Examples: `x` `Hallo_Welt` `Identifier_koennen_lang_sein`
Counterexamples: `1x` `H_a_l_l_o__W_e_l_t` `Name_`

3.2 String Literals

shall be defined as follows:

1. String literals are enclosed in quotes "`"`".
2. They consist of an arbitrary number of *printable characters*.
3. They must not contain a newline character.
4. If one quote shall be made part of a string literal, it has to appear twice in it.[2]

Examples: The string literals `""`, `" Hallo, Welt!"`, and `""" Hallo, Welt!"""` contain 0, 12, and 14 characters, respectively.

[1] These are the rules for identifiers in the programming language Ada.
[2] These are the rules for string literals in the programming language Pascal.

Task

Give regular definitions for identifiers and string literals.

Tip

- *PrintableChar* shall be the set of all printable characters, *including* newline.
- The regular expression"\n" shall represent newline.
- The regular expression "$\sim[c_1, \ldots, c_n]$" shall denote the set of characters that contains all printable characters, except for c_1 bis c_n; i.e., the set $\mathsf{PrintableChar} \setminus \{c_1, \ldots, c_n\}$.

4 Finite Automata

Use the rules discussed in the course to construct, for the regular definitions of identifiers and string literals, resp.,

1. a non-deterministic automaton,
2. the equivalent deterministic automaton, by the quotient set construction, and
3. the minimal deterministic automaton.

5 Recognition of Comments

Comments shall be enclosed in `/*` und `*/`; they may include newlines.

1. Define comments by a regular definition.
2. Construct a minimal finite automaton for comments from this definition. You may do this "intuitively", without taking the formal steps discussed in the course.
3. Check whether your definition gets simpler if you use a *lookahead operator* R_1/R_2, which generates the expression R_1 only if it is followed by the expression R_2.

Recognition of Float Literals

String literals shall be defined by

$$num \quad = \quad digit^{*} \,\underline{.}\; digit^{+} \; exp^{?}$$

1. Construct a minimal finite automaton for comments from this regular definition. You may do this "intuitively", without taking the formal steps discussed in the course.

6 Lexical Analysis of OOPS-0 with `lex`

The language OOPS, which is used in the compiler project at Universität Bremen, has the following lexemes:

- non-empty sequences of layout characters *blank*, *tabulator*, and *newline*,
- comments, also extending over several lines, enclosed in `(*` and `*)`,
- identifiers,

- integer literals,
- string literals,
- the delimiters `,`, `:`, `;`, `(`, and `)`,
- the operator symbols `<=`, `>=`, `:=`, `*`, `/`, `+`, `-`, `=`, `#`, `<`, `>`, and `.`, as well as
- the keywords `AND`, `ATTRIBUTE`, `BEGIN`, `CLASS`, `ELSE`, `ELSEIF`, `END`, `EXTENDS`, `IF`, `IS`, `METHOD`, `MOD`, `NEW`, `NOT`, `OR`, `OVERRIDE`, `READ`, `RETURN`, `THEN`, `WITH`, `WHILE`, and `WRITE`.

Tasks

1. Define the scanner for OOPS in `lex`. Here, we are not interested in the translation of lexemes into tokens, so the `return` statements can be omitted.)

   The regular definitions developed in the course can be used for the lexemes.

2. Produce tow versions of the scanner: one without keywords, and one including them (Then their definitions should go before that of identifiers.)

3. Compare the size of the scanner tables?

   So, should be keywords rather be recognized by the screener?

7 Ambiguity of Context-Free Grammar

Consider the following context-free rules, where terminals are underlined, and R is the start symbol:

$$
\begin{array}{rcl}
R & \rightarrow & R\,\underline{|}\,R \\
 & | & R\,R \\
 & | & R\,\underline{+} \quad | \; R\,\underline{*} \quad | \; R\,\underline{?} \\
 & | & \underline{(}\,R\,\underline{)} \\
 & | & \underline{\mathtt{a}} \;|\; \underline{\mathtt{b}} \;|\; \underline{\mathtt{c}}
\end{array}
$$

Tasks

1. Which language is defined here?

2. Show that the grammar is ambiguous.

3. Make it unambiguous. Derivations and derivation trees shall respect that the precedences of the operators increases from top to bottom The operators $\underline{*}$, $\underline{+}$ und $\underline{?}$ have the same precedence.

8 $SLL(1)$-Property for a Grammar of Commands

Let the syntax of commands be defined as follows:

program	::=	statement $\underline{\square}$
statements	::=	statement
	|	statement $\underline{;}$ statements
statement	::=	**if** expression **then** statements **end if**
	|	**if** expression **then** statements **else** statements **end if**
	|	**while** expression **do** statements **end while**
	|	id **:=** expression
	|	id
expression	::=	$\underline{\text{e}}$

Tasks

1. Check whether the grammar satisfies the $SLL(1)$ condition.
2. Determine the *First* and *Follower* sets for that purpose.
3. If the condition is violated, transform the grammar by left factorization until it does satisfy it.

9 Recursive-Descent Parsing of Commands

Consider the $SLL(1)$ syntax for commands from the previous exercise.

Tasks

1. Generate a recursive descent parser.
2. Construct an abstract syntax for commands.
3. Add tree-constructing instructions to the parser.
4. Incorporate error handling into the parser, based on deletion and insertion of symbols.

10 A Grammar Violating the $SLR(1)$ Condition

Consider the following grammar for expressions:

$$
\begin{array}{lcl}
\mathsf{S} & \to & \mathsf{E}\ \$ \\
\mathsf{E} & \to & \mathsf{E}\ \underline{+}\ E \\
 & | & \mathsf{E}\ \underline{*}\ \mathsf{E} \\
 & | & \underline{(}\ \mathsf{E}\ \underline{)} \\
 & | & \underline{\texttt{id}}
\end{array}
$$

Z	Action						Goto
	$	+	*	(	)	id	E
0				$s2$		$s3$	1
1	acc	$s4$	$s5$	$s2$		$s3$	
2				$s2$		$s3$	6
3	$r5$	$r5$	$r5$		$r5$		
4				$s2$		$s3$	7
5				$s2$		$s3$	8
6		$s4$	$s5$		$s9$		
7	$r2$	$r2$	$s5$		$r2$		
8	$r3$	$r3$	$r3$		$r3$		
9	$r4$	$r4$	$r4$		$r4$		

Figure 1: An action and goto table for parsing expressions

Tasks

1. Construct the *characteristic finite automaton* for the grammar.
2. Show that the grammar violates the $SLL(k)$ condition, by considering its action table.

11 Bottom-Up $SLR(1)$-Parsing

The grammar

$$
\begin{array}{lcll}
\mathsf{S} & \rightarrow & \mathsf{E}\,\$ & (1) \\
\mathsf{E} & \rightarrow & \mathsf{E} + E & (2) \\
 & | & \mathsf{E} * \mathsf{E} & (3) \\
 & | & (\mathsf{E}) & (4) \\
 & | & \mathtt{id} & (5)
\end{array}
$$

has a parse table as in Figure 1 (Precedences and associativities of the operators have been used to resolve the shift-reduce conflicts for this grammar.) The parser interpretes this table. States and rules are represented by their numbers, and the start state is 0:

```
pop(0);
first symbol;
while Action[top, symbol] ≠ acc
do case Action[top, symbol]
   of s_i ⇒ push(symbol); push(i); nextsymbol;
   | r_n ⇒ for i:= 1 to 2* length(rule(n)) do pop;
           z:= top;
           push(left-hand-side(rule(n))); push(Goto[z])
   otherwise syntaxerror("···");
   end case
end while;
if Action[top, symbol] = acc ∧ symbol = $
then print("yes!") else print("no!") end if.
```

Tasks

Execute the parser with the following programs, and non-programs:

- $a - b * c\square$
- $(a - b) * c\square$
- $(a - b)c\square$
- $(a - b) * -c\square$

Here a, b, c are numbers. Represent the configurations of the parser as

$$\underset{w}{\underline{s_0\, v_1 \cdots s_{k-1}\, v_k\, s_k}} \triangleright t_1 \cdots t_n \square$$

where w is the prefix read, the s_0, s_i are states, the v_i are symbols (for $1 \leqslant i \leqslant k$), and the t_j are terminal symbols (for $1 \leqslant j \leqslant n$).

12 Identification with Linear Visibility

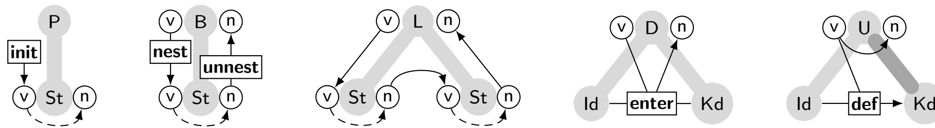
Consider the following program:

```
declare x:k1;
begin
  declare x:k2;
  declare y:k3;
  use y:
  use x
end;
use x
```

Use the attribute grammar presented in the course in the following steps:

- Construct an abstract syntax tree.
- Determine the attribution rules for the program.
- Which declarations are associated with the identifiers `x` and `y`?

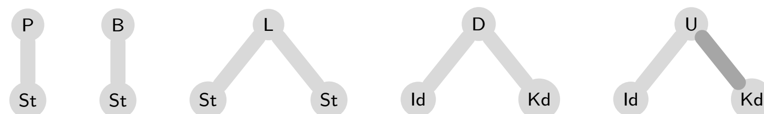
Recall the attribute rules:



13 Attribute Grammar for Global Visibility

Define, for the known abstract syntax and semantical basis, an attribute grammar that defines identification according to *simultaneous visibility*.

The rules of the underlying grammar define the following syntax trees:



The semantical basis should be extended by one operation:

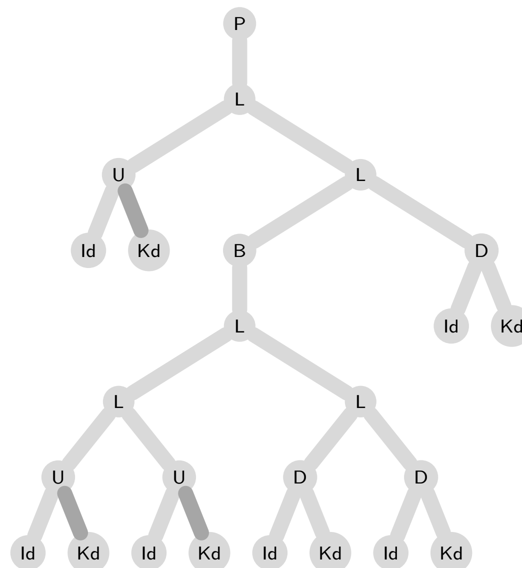
- The predicate `isLocal` can be useful to determine whether an identifier has already be declared within the current block.

In Haskell, the extended interface would be:

```
module DeclarationTable
where type TAB = [[(Id,Kd)]]
      type Id = String
      data Kd = Unbound | Typ | Const | Var | Proc deriving (Eq, Text)
      initial :: TAB
      nest, unnest :: TAB -> TAB
      enter :: TAB -> (Id,Kd) -> TAB
      def :: TAB -> Id -> Kd
      isLocal :: TAB -> Id -> Bool
```

14 Attribution with Simultaneous Visibility

Determine the attribution of the following syntax tree in a language with simultaneous visibility.



15 Declaration Tables in Java

Implement the semantic basis for identification, the declaration table, in Java. The signature is:

```
module DeclarationTable
where type TAB = [[(Id,Kd)]]
      type Id = String
      data Kd = Unbound | Typ | Const | Var | Proc deriving (Eq, Text)
      initial :: TAB
      nest, unnest :: TAB -> TAB
      enter :: TAB -> (Id,Kd) -> TAB
      def :: TAB -> Id -> Kd
      isLocal :: TAB -> Id -> Bool
```

16 Declaration Tables for Overloading

Until now, the grid-like declaration table copes with declarations that are *monomorphic* so that an identifier is associated with at most declaration. In a *polymorphic* language, a set of declarations can be associated with certain identifiers (of operations and functions). Then the function `def` yields a list (`Kinds`) of `Kd`. The function `enter`, when adding a declaration `(x,k)` to a table, has to decide whether

1. the entry can just be added, or
2. whether a similar entry is hidden by the new one, or
3. whether the new declaration is illegal, since the entry to be hidden appeared in the same block.

The predicate former called `isLocal` is not helpful any more. A predicate `distinct` should be provided, which determines whether two declarations (their `Kd`, that is) are distinguishable, and can hence be overloaded. [3] This predicate can be used in the new implementation of `enter`.

In Haskell the changed interface is::

```
module DeclarationTable
where type TAB = ...
      type Kinds = [Kd]
      type Kd  = ...
      type Id  = ...
      initial :: TAB
      nest, unnest :: TAB -> TAB
      enter :: TAB -> (Id,Kd) -> TAB
      def :: TAB -> Id -> Kinds
      distinct :: Kd -> Kd -> Bool
```

Tasks

1. Extend the efficient implementation of declaration tables so that it copes with overloaded declarations.
2. Does the complexity of the operations change?

17 Type Compatibility

For the type analysis of expressions, we have introduced a relation $t \trianglelefteq u$ expressing that a type u is compatible with a type t.

Tasks

Consider a programming language you know (Java, Haskell, C, C# ...).

1. At which program places, type compatibility is required?
2. How is type compatibility defined in the language?
3. Does a single notion of type compatibility hold throughout the language, or are there different notions, depending on the context?

[3] Languages differ wrt. the conditions under which functions or operations are considered to be distinguishable. This will be discussed in the course.

18 Transforming and Evaluating Expressions

Determine the transformation $c = \text{code}_W(3+(x*4))\alpha$. Assume that $\alpha(x) = 3$, and that $\mathsf{dop}_* = \mathsf{mul}$ has the meaning:

$$S[SP] := S[SP-1] * S[SP]; SP := SP - 1$$

Execute c with the P-Machine. Assume the code c starts at location 10; $S[3]$ shall contain the value 10 , and SP shall have the value 3.

The equations for transformation are on slide 169/171. The sketch of the P-machine is on page 166. [There is a minor error on this page. Do you find it?]

A simple form of numerical choice takes the following form:

$$\mathsf{C} \quad ::= \quad \underline{\texttt{case}}\ \mathsf{E}\ \underline{\texttt{of}}\ \underline{u}\ \underline{=>}\ \mathsf{C}_u\ \underline{;}\ \ldots\ \underline{;}\ \underline{o}\ \underline{=>}\ \mathsf{C}_o\ \underline{\texttt{default}}\ C_{\mathrm{d}}$$

The value v of the expression E is used to select the command C_v labeled with the corresponding value, from a contiguous integer range $u \ldots o$. If v is not in the range $u \ldots o$, the *default* command C_{d} is selected. (The pre-historic **switch** of ALGOL-60, C, C++, and JAVA does not necessarily perform a unique selection (only if all commands are terminated with **break**).

Numerical choice can be transformed, on the source level, into an equivalent cascade of Boolean choices (`if` – `then` – `else`). It can also be implemented with a goto table: n unconditional jumps `ujp` l_i are inserted in the code – not in the data – and a *computed goto* jumps to the right place in that table. This instruction takes the following form:

$$\underline{\texttt{ijp}} \quad \equiv \quad PC := PC + S[SP];\ SP := SP - 1$$

The instruction expects the goto table to be placed in the directly succeeding code locations. Following instructions might be useful.

$$\underline{\texttt{ljp}}\ u\ l \quad \equiv \quad \underline{\texttt{if}}\ \ s[SP] < u\ \underline{\texttt{then}}\ \ PC := PC + l$$
$$\underline{\texttt{gjp}}\ o\ l \quad \equiv \quad \underline{\texttt{if}}\ \ s[SP] > o\ \underline{\texttt{then}}\ \ PC := PC + l$$

Tasks

1. Translate numeric choice, either to equivalent `if` -cascades, or with a goto table.

2. Which of the implementations do you consider to be better? More precisely, under which circumstances could one be better than the other one?

3. If the same command shall be selected for different numerical values, the simple form of `case` forces to duplicate code, which should be avoided for several reasons.

   How could you change the transformation, if a list of literals can be placed in front of the commands?

19 Transforming Generalized Expressions

1. *Predefined Functions.* Allow calls to standard functions of the form $f(e_1, \ldots e_k)$ in expressions, and extend the transformations to handle these calls. Assume that that $\alpha(f)$ contains the address of code(f).

   Transform function calls into machine code. You may wish to introduce an instruction that jumps to the code of standard functions. This instruction could use a register, say OPC, to save the program counter. An instruction for returning from a standard function concludes the code of standard functions. Assume that this instruction restores the program counter with "$PC := OPC$".

2. *Compound Operands*. Allow that the infix operations and standard functions in an expression may have operands and results that occupy several storage cells, not just one.

   Extend the transformation scheme accordingly.

20 Array Transformation

Consider the following Pascal program:

```
program Feld;
  type Complex = record re, im: Real; end;
  var a: array [-3..+3] of Complex;
  var i: Integer;
begin
  i:= 1;
  a[i] := a[i+1]
end.
```

Determine the size of the types, and the address environment of the program.

Translate the program and execute the translated P-code. Take care: the transformation presented in the course does only cover array ranges $0 \dots k$. Invent an extension for the more general case (or just look it up in the presentations).

21 Storage and Selection of Dynamic Arrays

So far assumed that the range $[u..o]$ of an array is static. Then its size is static as well, like of any other type discussed so far. Values of static arrays can be allocated on the stack.

An array is *dynamic* if its range is $[E_{\mathrm{u}}..E_{\mathrm{o}}]$ is given by the values of expressions. The size of a dynamic array is dynamic (hence the name), so that its values cannot be stored on the stack like values of static size.

Many modern languages allocate dynamic arrays on the heap. This has the drawback that the space for an array has to be recovered by garbage collection if it is no longer alive. It would be better to allocate it on the stack where the storage is reclaimed automatically. For this purpose, he representation of a dynamic array is divided into two parts:

1. Its *descriptor* contains all information that is needed to perform the operations on it:
   - The upper bound *og* (the value of E_{o}).
   - The lower bound *ug* (the value of E_{u}).
   - The subtrahend *su* (the difference to the 0th element).
   - The size *gr*, as the number of its storage cells.
   - The fictive address *fa* (the address of the 0th element).
2. Its proper values are kept in a dedicated area *DYN* of the stack, between the area (LOC) for values of static size and the stack for intermediate calculations in expressions ($ZWERG$).

The *size* of a dynamic array is just the (static) size of its descriptor:

$$size(\mathbf{row}\ [E_{\mathrm{u}}..E_{\mathrm{o}}]\ \mathbf{of}\ t) = 5$$

Then the address environment α can be determined as before.

Furthermore, the elaboration of a dynamic array declaration must produce code that, at runtime, defines the values of the descriptor, and allocates space for its proper value in *DYN*.

Table 1: Instructions allocating and accessing dynamic arrays

Command	Meaning Explanation
sad a g	$S[a+4] := S[SP];$ (og) $S[a+3] := S[SP-1];$ (ug) $S[a+2] := S[a+3] * g;$ (su) $S[a+1] := S[a+4] - S[a+3] + 1;$ (gr) $S[a] := SP - S[a-2] - 1;$ (fa) $SP := SP + S[a+1] - 2$ (allocate space)
dpl	$SP := SP + 1;$ $S[SP] := S[SP-1];$
ind	$S[SP] := S[S[SP]]$
chd	**if** $S[SP] < S[S[SP-2]+3]$ **or** $S[SP] > S[S[SP-2]+4]$ **then** error "index error";
ixa g	$S[SP-1] := S[SP-1] + (S[SP] * g); SP := SP - 1;$

We assume that the elaboration of local declarations calls an instruction **isp** g that increases register SP by the size g of all local declarations – including the descriptors of dynamic arrays.

Then ever declaration of a dynamic array produces the following code:

$$[\![x : \textbf{row}\ [E_\text{u}..E_\text{o}]\ \textbf{of}\ t]\!]\alpha = [\![V]\!]_\text{A}\ \alpha; [\![E_\text{u}]\!]_\text{W}\ \alpha; [\![E_\text{o}]\!]_\text{W}\ \alpha; \textsf{sad}\ \alpha(x)\ \textit{size}(t);$$

The access to an element of a dynamic array is translated as follows:

$$[\![V[E]]\!]\alpha = [\![V]\!]_\text{A}\ \alpha; \textsf{dpl}\ ; \textsf{ind}\ ; [\![E]\!]_\text{W}\ \alpha; \textsf{chd}\ ; \textsf{ixa}\ \textit{size}(\textit{type}(x[E]))\ ; \textsf{sli}$$

The code $[\![V]\!]_\text{A}\ \alpha$ yields the descriptor address of a dynamic array x, not the address of the values themselves. The descriptor address is duplicated with the instruction **dpl** first, and then used to calculate the address of the array values with **ind** (their fictive address, rather), before the Index is calculated. The original of the descriptor is needed to check array bounds with the command **chd** . If the variable access V is not just an identifier x, but a component, say $y.s$, of a product, the address of the descriptor cannot be passed to the instruction directly. The original descriptor is two cells below SP – above is the address of the array values and the values of the index. Only when the address of the array element is calculated, it will be slided down on the stack, overwriting the original descriptor, by the instruction **sli**.

In the following Pascal program:

```
procedure Feld;
  var a: array [-n..+n] of Complex;
  var i: Integer;
  begin
    i:= 1;
    a[i] := a[i+1]
  end;
```

the global variable `n` shall have the value 3. Let $\textit{size}(\texttt{Complex}) = 2$ and $\alpha = \{\texttt{n} \mapsto 5, \texttt{a} \mapsto 10, \texttt{i} \mapsto 15\}$. After elaboration of the declarations, the stack pointer shall point to the last cell occupied by local variables, namely 15.

The code for allocating the array looks as follows:

$$\begin{aligned}[\![\texttt{x: array [-n..+n] of Complex}]\!]\alpha &= [\![\texttt{x}]\!]_\text{A}\ \alpha; [\![-\texttt{n}]\!]_\text{W}\ \alpha; [\![\texttt{n}]\!]_\text{W}\ \alpha; \textsf{sad}\ \alpha(x)\ \textit{size}(\texttt{Complex});\\ &= \textsf{lda}\ \alpha(x)\ ; \textsf{lda}\ \alpha(\texttt{n})\ ; \textsf{ind}\ ; \textsf{neg}\ ;\\ &\quad\ \textsf{lda}\ \alpha(\texttt{n})\ ; \textsf{ind}\ ; \textsf{sad}\ \alpha(x)\ \textit{size}(\texttt{Complex});\\ &= \textsf{lda}\ 10\ ; \textsf{lda}\ 5\ ; \textsf{ind}\ ; \textsf{neg}\ ; \textsf{lda}\ 5\ ; \textsf{ind}\ ; \textsf{sad}\ 10\ 2;\end{aligned}$$

By these instructions, the store is manipulated as shown in Figure2.

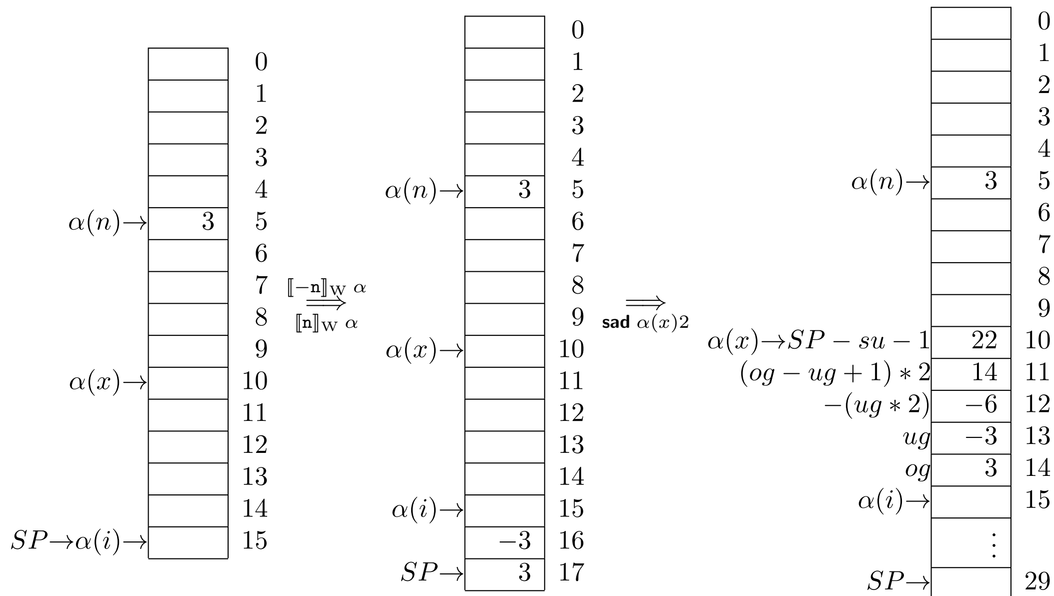


Figure 2: Allocation of an array descriptor

Task

1. Define the code for $[\![\mathtt{x[i]}]\!]\alpha$, assuming that `i` has the value 1.
2. Demonstrate how the code is executed.

22 Static Predecessors

In the course, we have discussed how static predecessors can be organized as a linked list. This leads, with deeply nested blocks, to a certain overhead in the access to non-local identifiers.

This can be avoided if static predecessors are organized in a *display vector*. Here the static predecessors are held in a (small) global stack at the start of the storage area. As the depth of block nesting is a static value, the maximal extension of the display vector can be determined in advance.

Tasks

Organize static predecessors in a display vector.

1. What has to be done when entering and leaving a procedure?
2. Redefine the instruction **lda**. You may change the meaning of the modifier ℓ for that purpose.
3. If a variable is global, i.e., declared on level $\ell = 0$, or local, i.e., declared on $\ell =$ *current level*, the display vector is not needed at all. Define the instructions **ldg** and **ldl** for access to global and local variables, respectively.

23 Translating Recursive Procedures

Translate the following Pascal function:

```
function fac (n: Integer): Integer;
begin
   if n <= 0 then fac := 1 else fac := fac(n-1)
end
```

Task

The statement "`fac` $:= E$" defines the result of the function `fac` (instead of "**return** E" in other languages). Execute the code.

24 Parameter Passing

This is about endangered ways of parameter passing. Let **proc** $p(x : t); B$ be a procedure that is called as $p(E)$.

Constant Parameters

If x is handled as a *constant parameter*, x acts as a constant name for the value of E while p is executed, as if a declaration **const** $x = E$; would be elaborated before entering p.

Questions

1. In what differs this passing mode to *value parameters* ("*call by value*")?
2. Consider the advantages and drawbacks of constant wrt. value parameters.
3. What code has to be generated for passing a dynamic array as a constant parameter?

Name Parameters

This way of parameter passing resembles textual substitution, and has been used in the lambda calculus. It is defined as follows: If x is passed as a name parameter, it denotes the expression E while the body B of p is executed. Whenever x is used, the expression E will be evaluated; this is done in the context of the block B, considering all declarations for B.

Questions

1. Consider advantages and drawbacks of name parameters.
2. Sketch how name parameters could be transformed into code.
3. What is the difference of name parameters to pure textual substitution?

25 Translating a Complete Program

Translate the following program:

```
program fakultaet;
  var x : Integer;
  function fac (n: Integer): Integer;
  begin
   if n <= 0 then fac := 1 else fac := n * fac(n-1)
  end
begin
  x := fac(2)
end.
```

26 Multiple Inheritance

Consider how the organization of object-oriented code must be changed in the presence of “proper” multiple inheritance (not only of interfaces, as in Java).

1. How can method tables be organized?
2. How can dynamic binding of methods be done?