Compiler Construction
Solutions to Exercises

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

January/February 2011

These are solutions to most of the exercises. The explanations may be very brief, because of lack of time for translating them to English.

Solution 1: Static or Dynamic?

For C, Java,and Haskell folds:

1. *Binding* is of names to types, classes, variables, procedures is *static*, with one notable exception: in Java, the binding of names to method bodies is, in general, dynamic.

2. In C the type of a variable is *static*, even if it is not always checked.

   In Haskell, variables are "single-assignment", and *static* as well. In Java, variables have a static base type b, and may contain, dynamically, values of some subtype of b.

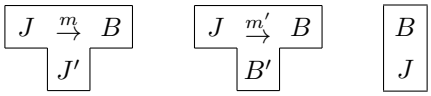
3. In C or Java, an *array* of the form"$[n]\langle type\rangle a$" has a *static* length. In C, arrays can be declared as pointers, and be allocated (with `malloc`) a storage of dynamic size.

   The predefined arrays of Haskell are of dynamic size.

4. Values of a variable are dynamic, in general. Only if the variable is "single assignment", and the expression defining the variable is constant, the value itseölf is constant as well.

Solution 2: *Bootstrapping*

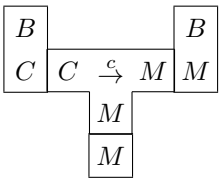
The portable Java Implementation



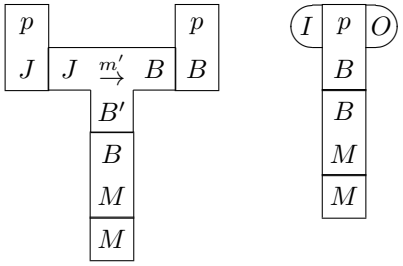
The compilers m und m' use only subsets $J' \subseteq J$ bzw. $B' \subseteq B$ of Java and JBC, respectively.

1: Porting to M

The JVM could be ported to C, and be translated with the native C compiler:



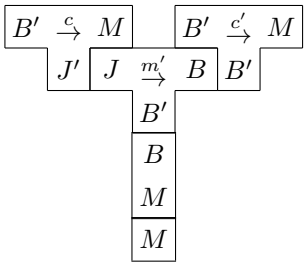
Now the compiler m' can be interpreted on M, as well as every program p translated with it:



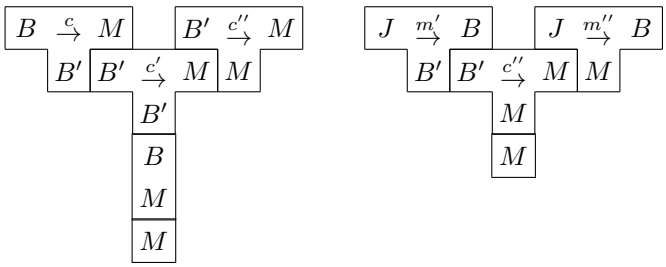
However, the compiler will be slow on M as it is interpeted.

2: Bootstrap on M

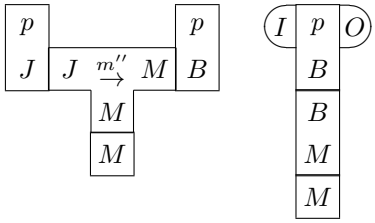
Instead of an interpreter, we write a compiler c of B' into M code; c will just be used to translate m' so that not all of the JVM needs t obe translated; c coulkd be done in C.



The result c' can be used to compile c "with itself", in order to translate, with the resulting c'', the master compiler m' to a "semi-native" Utranslatoe for Java to B in M machinecode:



Now we have the required native compiler. The B programs produced by it can be interpreted:



Solution 3: Regular Definitions

Identifiers

$$\begin{aligned} \mathit{identifier} &= \mathit{letter}\left((\_)^{?}\,(\mathit{letter} \mid \mathit{digit})\right)^{*} \\ \mathit{letter} &= \texttt{a} \mid \cdots \texttt{z} \mid \texttt{A} \mid \cdots \texttt{Z} \\ \mathit{digit} &= \texttt{0} \mid \cdots \texttt{9} \end{aligned}$$

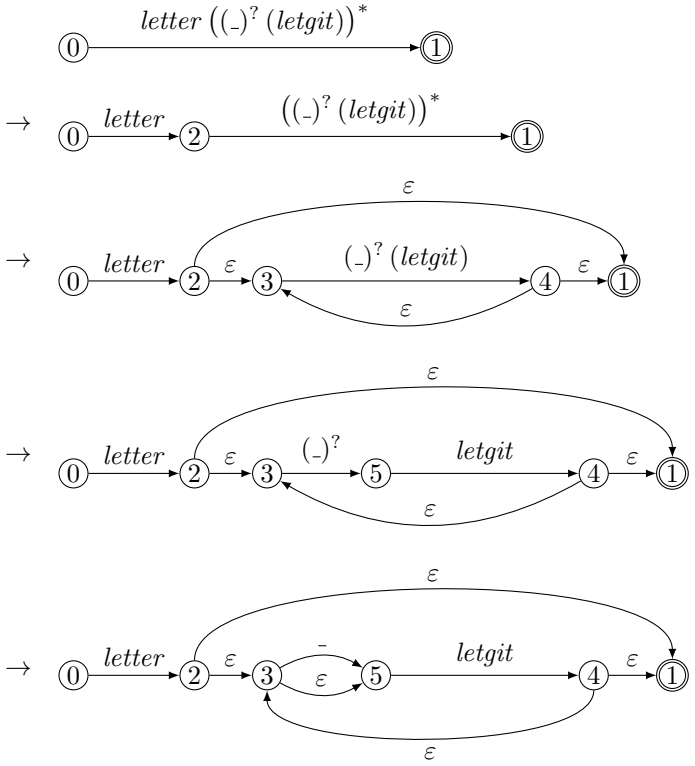
String Literals

$$\mathit{string} = \texttt{"}\,(\texttt{""} \mid \sim[\texttt{"}, \backslash\texttt{n}])^{*}\,\texttt{"}$$

Solution 4: Finite Automata

Identifiers

NFA B

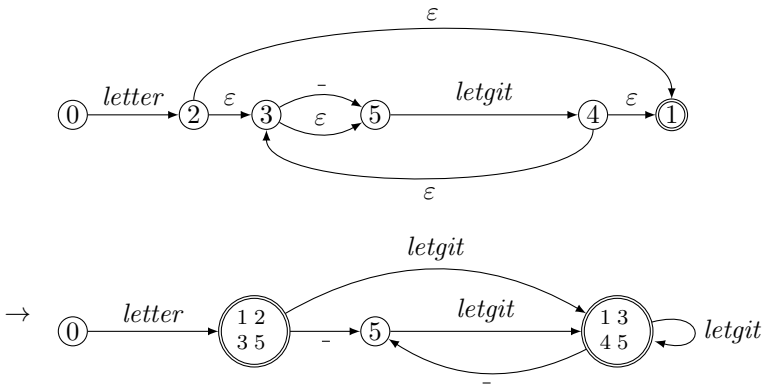


DFA The quotient set construction gives the following states and transitions:

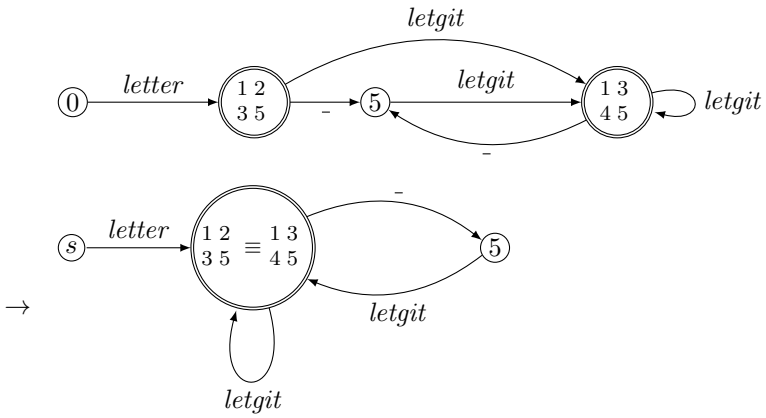
1. $P_0 = E(0) = \{0\}$.
2. $P_1 = F(P_0, \mathit{letter}) = \{1, 2, 3, 5\}$ with transition under *letter*.
3. $P_2 = F(P_1, \_) = \{5\}$ with transition under _.
4. $P_3 = F(P_1, \mathit{letgit}) = \{1, 3, 4, 5\}$ with transition under *letgit*.
5. $F(P_3, \_) = \{5\} = P_2$ with transition under _.

6. $F(P_3, letter|digit) = \{1, 3, 4, 5\} = P_3$ with transition under *letgit*.

This gives the following DFA:

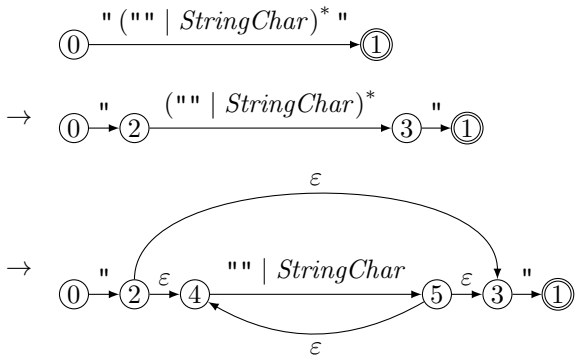


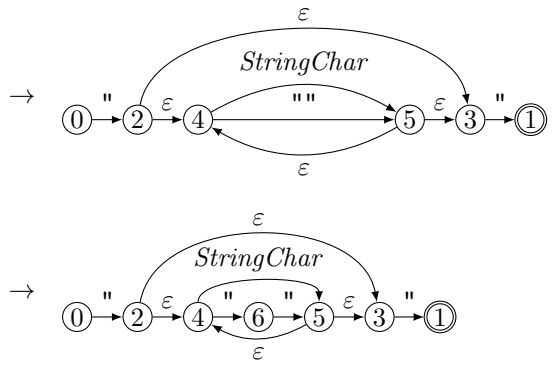
Minimal DEA States P_1 and P_3 turn out to be equivalent. There are no dead or unreachable states. also:



String Literals

NFA

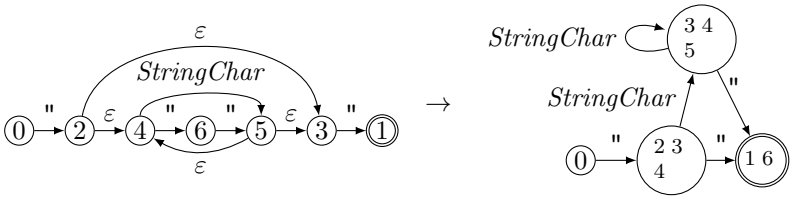




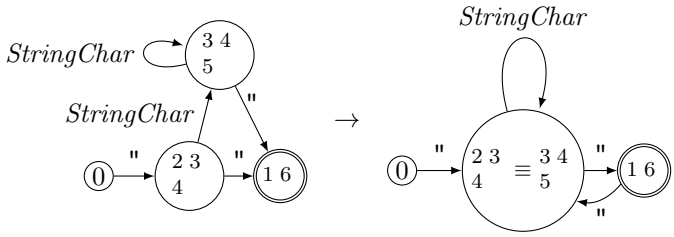
DFA The quotient set construction gives the following states and transitions:

1. $P_0 = E(0) = \{0\}$.
2. $P_1 = F(P_0, \texttt{"}) = \{2, 3, 4\}$ with transition under ".
3. $P_2 = F(P_1, \texttt{"}) = \{1, 6\}$ with transition under ".
4. $P_3 = F(P_1, \mathit{StringChar}) = \{3, 4, 5\}$ with transition under *StringChar*.
5. $F(P_3, \texttt{"}) = \{1, 6\} = P_2$ with transition under ".
6. $F(P_3, \mathit{StringChar}) = \{3, 4, 5\} = P_3$ with transition under *StringChar*.

The DFAD has following transition diagram:



Minimal DFA States P_1 und P_3 turn out to be equivalent. No state is dead or unreachable:



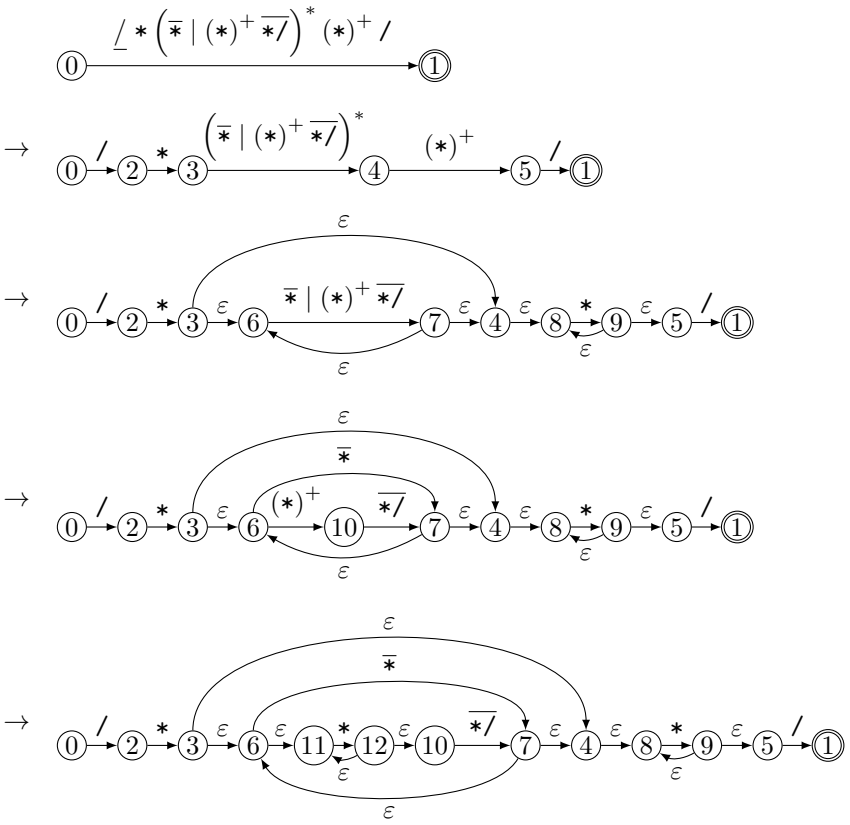
Solution 5: Recognition of Comments

1. Regular Definition :

$$\mathit{comment} \quad = \quad \underline{/}\,*\left(\overline{*} \mid (*)^+\,\overline{*/}\right)^*(*)^+\,/$$

Here, $\overline{t_1 \ldots t_n}$ denotes the *complement* of the set $\{t_1, \ldots, t_n\}$, that is, all characters except $t_1, \ldots, t_n$.

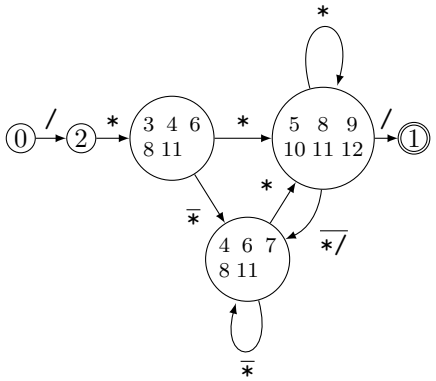
2. NFA



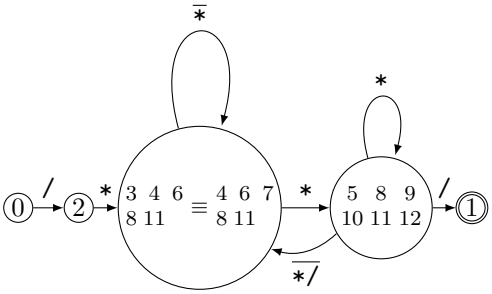
3. DFA The quotient set construction gives the following states and transitions:

1. $P_0 = E(0) = \{0\}$.
2. $P_0 \xrightarrow{/} P_1 = F(P_0, /) = \{2\}$.
3. $P_1 \xrightarrow{*} P_2 = F(P_1, *) = \{3, 4, 6, 8, 11\}$.
4. $P_2 \xrightarrow{*} P_3 = F(P_2, *) = \{5, 8, 9, 10, 11, 12\}$.
5. $P_2 \xrightarrow{\overline{*}} P_4 = F(P_2, \overline{*}) = \{4, 6, 7, 8, 11\}$.
6. $P_3 \xrightarrow{/} P_5 = F(P_3, /) = \{1\}$.
7. $P_3 \xrightarrow{*} F(P_3, *) = \{5, 9, 8, 10, 11, 12\} = P_3$.
8. $P_3 \xrightarrow{\overline{*/}} F(P_3, \overline{*/}) = \{4, 6, 7, 8, 11\} = P_4$.
9. $P_4 \xrightarrow{*} F(P_4, *) = \{5, 8, 9, 10, 11, 12\} = P_3$.
10. $P_4 \xrightarrow{\overline{*}} F(P_4, \overline{*}) = \{4, 6, 7, 8, 11\} = P_4$.

The DFA is as follows:



4. Minimal DFA States $P_2 = \{3, 4, 6, 8, 11\}$ and $P_4 = \{4, 6, 7, 8, 11\}$ turn out to be equivalent, since both are not final, and their successor states (under $*$ and $\overline{*}$) are the equivalent:

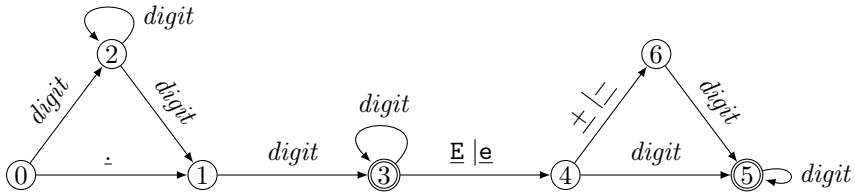


Recognition of Float Literals

The start state of the DFA for float literals represents the expression:

$$
\begin{array}{rcl}
R_0 & = & \mathit{digit}^*\,\underline{.}\;\mathit{digit}^+\;\mathit{exp}^? \\
 & \equiv & \varepsilon\,\underline{.}\;\mathit{digit}^+\;\mathit{exp}^? \\
 & | & \mathit{digit}\,\mathit{digit}^*\,\underline{.}\;\mathit{digit}^+\;\mathit{exp}^? \\
 & \equiv & \underline{.}\;\mathit{digit}^+\;\mathit{exp}^? \\
 & | & \mathit{digit}\;\mathit{digit}^*\,\underline{.}\;\mathit{digit}^+\;\mathit{exp}^? \\
R_1 & = & \mathit{digit}^+\;\mathit{exp}^? \\
 & \equiv & \mathit{digit}\;\;\mathit{digit}^*\;\mathit{exp}^? \\
R_2 & = & \mathit{digit}^*\,\underline{.}\;\mathit{digit}^+\;\mathit{exp}^? = R_0 \\
R_3 & = & \mathit{digit}^*\;\mathit{exp}^? \\
 & \equiv & \varepsilon\;\;\mathit{exp}^? \\
 & | & \mathit{digit}\;\mathit{digit}^*\;\mathit{exp}^? \\
 & \equiv & \varepsilon \\
 & | & (\underline{\mathtt{E}}\mid\underline{\mathtt{e}})\,(\underline{+}\mid\underline{-})^?\;\;\mathit{digit}^+ \\
 & | & \mathit{digit}\;\mathit{digit}^*\;\;\mathit{exp}^?
\end{array}
$$

$$
\begin{array}{rcl}
R_4 & = & (\underline{+}\mid\underline{-})^?\;\;\mathit{digit}^+ \\
 & \equiv & \varepsilon\;\;\mathit{digit}^+ \\
 & | & (\underline{+}\mid\underline{-})\;\;\mathit{digit}^+ \\
 & \equiv & \mathit{digit}^+ \\
 & | & (\underline{+}\mid\underline{-})\;\;\mathit{digit}^+ \\
 & \equiv & \mathit{digit}\;\;\mathit{digit}^* \\
 & | & (\underline{+}\mid\underline{-})\;\;\mathit{digit}^+ \\
R_5 & = & \mathit{digit}^*\;\;\mathit{exp}^? = R_3 \\
R_5 & = & \mathit{digit}^* \\
 & | & (\underline{+}\mid\underline{-})\;\;\mathit{digit}^+ \\
 & \equiv & \varepsilon \\
 & | & \mathit{digit}\;\;\mathit{digit}^* \\
 & | & (\underline{+}\mid\underline{-})\;\;\mathit{digit}^+ \\
R_6 & = & \mathit{digit}^* = R_5 \\
R_7 & = & \mathit{digit}^+ \\
 & \equiv & \mathit{digit}\;\;\mathit{digit}^* \\
R_8 & = & \mathit{digit}^* = R_6
\end{array}
$$



Solution 6: Lexical Analysis of OOPS-0 mit `lex`

1. Scanner without Keywords

```
%option noyywrap
/* Lexeme von OOPS, ohne Schlüsselwörter
  (Berthold Hoffmann, hof@informatik.uni-bremen.de)  */
/* Übersetzer Sommer 2011, Übungszettel 5            */

/* needed for output of numbers */
%{
#include <math.h>
%}

/* character classes */
layout     [ \t\n\f]
letter     [a-zA-Z]
digit      [0-9]
noast      [^\*]
noclose    [^\)]
noastclose [^\*\)]

/* lexeme definitions by regular expressions */
whitespace {layout}+
comment    \(\*({noast}|\*+{noastclose})*\*+\)
prchar     [^\\\']|\\n|\\t|\\\'
string     \'({prchar})*\'
identifier {letter}({letter}|{digit})*
number     {digit}+

%%
/* screener transformation - rather trivial  */
{whitespace}  {printf("Whitespace: []%s[]\n", yytext);}
{comment}     {printf("A comment: %s\n", yytext);}
{identifier}  {printf("An identifier: %s\n", yytext);}
{string}      {printf("A string literal: %s\n", yytext);}
{number}      {printf("An integer literal: %s (%d)\n", yytext, atoi(yytext));}
"<="|">="|":="|"*"|"/"|"+"|"-"|"="|"#"|"<"|">"|"?"|"!"          {
               printf("An operator: %s\n", yytext);
              }
","|":"|";"|"("|")"           {
               printf("A delimiter: %s\n", yytext);
              }
(.)           {printf("an erroneous character: %s\n", yytext);}
```

```
%%
main(argc, argv)
int argc;
char **argv;
    {
++argv; --argc; /* skip over the program name */
if (argc > 0 )
    yyin = fopen(argv[0], "r");
else
    yyin = stdin;
yylex();
    }
```

2. Scanner with Keywords

Keyword can be defined with the following regular expression:

```
keyword     (AND|ATTRIBUTE|BEGIN|CLASS|ELSE|ELSEIF|END|EXTENDS|IF|IS
            |METHOD|MOD|NEW|NOT|OR|OVERRIDE|READ|RETURN)
```

The line

```
{keyword}     {printf("A keyword: %s\n", yytext);}
```

has t oappear before the line for identifiers.

3. Comparing Scanner Sizes

Called with option `-v`, `flex` reports (without keywords):

```
...
116/2000 NFA-Zustände
30/1000 DFA-Zustände (165 Wörter)
...
71 Epsilon-Zustände, 40 Doppel-Epsilon-Zustände
...
532 Tabelleneinträge insgesamt benötigt
```

With keywors, it reprits:

```
...
216/2000 NFA-Zustände
79/1000 DFA-Zustände (470 Wörter)
...
90 Epsilon-Zustände, 58 Doppel-Epsilon-Zustände
...
828 Tabelleneinträge insgesamt benötigt
```

Keywords enlarge the scanner by 60%.

Solution 7: Ambiguity in Context-Free Grammars

The rules

$$R \to R\,\underline{|}\,R \mid R\,R \mid R\,\underline{+} \mid R\,\underline{*} \mid R\,\underline{?} \mid \underline{(}\,R\,\underline{)} \mid \underline{\texttt{a}} \mid \underline{\texttt{b}} \mid \underline{\texttt{c}}$$

define regular expressions over three terminal symbols.

The grammar is ambiguous as the $\underline{\texttt{a|ba}}$ und $\underline{\texttt{a|b|c}}$ und $\underline{\texttt{abc}}$ have two leftmost derivations each, because it fails to define the associativity of alternatives and concatenations, as well as their precedence.

As with expressions, two subcategories T and F of R have to be introduced, where the former does not generate alternatives, and the latter does generate neither alternetives, nor sequences.

$$\begin{aligned} R &\to R\,\underline{|}\,T \mid R \\ T &\to T\,F \mid F \\ F &\to F\,\underline{+} \mid F\,\underline{*} \mid F\,\underline{?} \mid \underline{(}\,R\,\underline{)} \mid \underline{\texttt{a}} \mid \underline{\texttt{b}} \mid \underline{\texttt{c}} \end{aligned}$$

Then the example words above have only one leftmost derivation.

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

For the rules of statements, and the first and last rule rules of statement, the *First* sets are not disjoint, and thus violate the $SLL(1)$ condition.

2. We factorize these three rule pairs, where the conditional rules are factorized in the middle, in oder to get an intuitive rule.

program ::= statement $\underline{\square}$

statements ::= statement statement_rest

statement_rest ::= ε | $\underline{;}$ statement statement_rest

statement ::= **if** expression **then** statements else_part **end if**
| **while** expression **do** statements **end while**
| id assignment_rest

else_part ::= ε | **else** statements

assignment_rest ::= ε | **:=** expression

Then the *First* and *Follower* sets are as follows:

Nichtterminal	$Prefix_1$	$Succ_1$
program	–	$\bot$
statements	$Prefix_1$(statement)	$\underline{\texttt{elsif}}$, $\underline{\texttt{else}}$, $\underline{\texttt{end}}$
statement_rest	ε, $\underline{;}$	$Succ_1$(statements)
statement	$\underline{\texttt{id}}$, $\underline{\texttt{if}}$, $\underline{\texttt{while}}$	$\square$, $\underline{;}$, $\underline{\texttt{elsif}}$, $\underline{\texttt{else}}$, $\underline{\texttt{end}}$
else_part	ε, $\underline{\texttt{else}}$	$\underline{\texttt{end}}$
assignment_rest	ε, $\underline{:=}$	$Succ_1$(statement)

The grammar now satisfies the $SLL(1)$ condition.

Solution 9: Recursive-Descent Parsing of Commands

1. We use a version of the syntax with EBNF operators:

$$
\begin{array}{rcl}
\text{program} & ::= & \text{statement } \underline{\square} \\
\text{statements} & ::= & \text{statement } \{\underline{;}\ \text{statement}\} \\
\text{statement} & ::= & \text{id } [\ \textbf{:=}\ \text{expression }] \\
 & | & \textbf{while } \text{expression } \textbf{do } \text{statements } \textbf{end} \\
 & | & \textbf{if } \text{expression } \textbf{then } \text{statements} \\
 & & \{\textbf{elseif } \text{expression } \textbf{then } \text{statements }\} \\
 & & [\textbf{else } \text{statements}]\ \textbf{end}
\end{array}
$$

1. Transformation in rekursive Parsierfunktionen

⟦statements ::= ⋯⟧

= **procedure** statements;
begin
⟦statement {statement }⟧
end ;

= **procedure** statements;
begin
⟦statement ⟧;
while $\ell = \underline{;}$
loop scan;
⟦statement ⟧
end loop;
end ;

= **procedure** statements;
begin
statement;
while $\ell = \underline{;}$
loop scan;
statement
end loop;
end ;

⟦statement ::= ⋯⟧

= **procedure** statement;
begin
⟦id ... end ⟧
end ;

= **procedure** statement;
begin
if $\ell = \underline{\texttt{id}}$ **then** ⟦id [:= expression]⟧
elsif $\ell = \underline{\texttt{while}}$ **then** ⟦while ... end ⟧
elsif $\ell = \underline{\texttt{if}}$ **then** ⟦if ... end ⟧
else ... **end if** ;
end ;

= **procedure** statement;
begin
if $\ell = \underline{\texttt{id}}$ **then** scan; **if** $\ell = \underline{:=}$ **then** scan; expression **end if**
elsif $\ell = \underline{\texttt{while}}$ **then** ⟦while ... end ⟧
elsif $\ell = \underline{\texttt{if}}$ **then** ⟦if ... end ⟧
else ... **end if** ;
end ;

= **procedure** statement;
begin
if $\ell = \underline{\texttt{id}}$ **then** ... (*wie oben*)
elsif $\ell = \underline{\texttt{while}}$
then scan; expression; match(do); statements; match(end);
elsif $\ell = \underline{\texttt{if}}$ **then** ⟦if ... end ⟧
else ... **end if** ;
end ;

```
=   procedure statement;
    begin
      if ℓ = id then ... (wie oben)
      elsif ℓ = while then ... (wie oben)
      elsif ℓ = if
      then scan; expression; match(then ); statements;
        [[{elsif expression then statements }]] [[[else statements ]]]
        match(end );match(if )
      else ... end if ;
    end ;
=   procedure statement;
    begin
      if ℓ = id then ... (wie oben)
      elsif ℓ = while then ... (wie oben)
      elsif ℓ = if
      then scan; expression; match(then ); statements;
        while ℓ = elsif
        loop scan; expression; match(then ); statements;
        end loop;
        if ℓ = else then scan; statements; end if;
        match(end );match(if )
      else ... end if ;
    end ;
```

2. Construction of Abstract Syntax Trees

We just show how the parser for conditionals can be extended, which is the most interesting case: We assume that new(t1,If) allocates a record with components cond, thenpart, and elsepart. Missing else parts are represented by a node for the empty statement sequence (Skip); **elsif** parts are represented by nested If nodes.

3. Parsing with Tree Construction

Every parsing procedure is turned a function returning its abstract syntax tree. The components of the If node are filled while parsing its components.

```
function statement return Stmt is
  var t1: Stmt;
begin
  if ...
  elseif ...
  elsif ℓ = if
  then new(t1,If);
    scan; t1.cond := expression; match(then ); t1.thenpart := statements;
    while ℓ = elsif
    do t1:= t1.elsepart; new(t1,If);
      scan; t1.cond := expression; match(then ); t1.thenpart := statements
    if ℓ = else then scan; t1.elsepart := statements; else t1.elsepart:= Skip;
    match(end );match(if );
    return t1
  else ...
  end if ;
end ;
```

4. Error Recovery

1. *Insert a symbol.* If procedure match does not find the expected symbol, it acts as if this symbol has been present. This procedure is called if some part of an alternative has already been parsed, e.g., for the keywords **do**, **end**, **then**, **elseif**, and **else**.

2. *Skip symbols.* If no alternative of a rule can be parsed, error handling should skip all symbols until a possible successor of the rule is found, e.g., symbols □, ; , **elseif**, **else**, and **end** in the rule for statement.

3. *Replace a symbol.* This should be done only in situations where

   (a) a specific symbol is expected,

   (b) the parser will not know how to continue if it is missing,

   (c) the input contains a symbol that is *similar* to the expected symbol, and

   (d) the overnext symbol is something that may follow the expected symbol.

   In our example, procedure match(then) could replace words them, than, ... by **then** if the overnext input symbol ia a possible starter of statement.

   With **elsif** or **else**, this is not advisable, ab both symbols are optional, and resemble the next keyword **end**.

Solution 10: A Grammar violating the $SLR(1)$ condition

1. The grammar is *ambiguous*, since expressions like

   $$\underline{\texttt{id}} \mathbin{\underline{+}} \underline{\texttt{id}} \mathbin{\underline{+}} \underline{\texttt{id}} \qquad \underline{\texttt{id}} \mathbin{\underline{*}} \underline{\texttt{id}} \mathbin{\underline{*}} \underline{\texttt{id}} \qquad \underline{\texttt{id}} \mathbin{\underline{+}} \underline{\texttt{id}} \mathbin{\underline{*}} \underline{\texttt{id}}$$

   have two leftmost derivations (and two distinct derivation trees). The grammar does not define precedence and associativity of the operators $\underline{+}$ and $\underline{*}$.

2. The *characteristic finite automaton* for this grammars has states as shown in Figure 1.

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

$$\vdash \xrightarrow[\varepsilon]{0} a-b*c\square \vdash \xrightarrow[a]{0\underline{\texttt{i}}\ 3} -b*c\square$$

$$\vdash \xrightarrow[a]{0\texttt{E}\ 1} -b*c\square \vdash \xrightarrow[a-]{0\texttt{E}\ 1-4} b*c\square$$

$$\vdash \xrightarrow[a-b]{0\texttt{E}\ 1-4\underline{\texttt{i}}\ 3} *c\square \vdash \xrightarrow[a-b]{0\texttt{E}\ 1-4\texttt{E}\ 7} *c\square$$

$$\vdash \xrightarrow[a-b*]{0\texttt{E}\ 1-4\texttt{E}\ 1*5} c\square \vdash \xrightarrow[a-b*c]{0\texttt{E}\ 1-4\texttt{E}\ 1*5\underline{\texttt{i}}\ 3} \square$$

$$\vdash \xrightarrow[a-b*c]{0\texttt{E}\ 1-4\texttt{E}\ 1*5\texttt{E}\ 8} \square \vdash \xrightarrow[a-b*c]{0\texttt{E}\ 1-4\texttt{E}\ 7} \square$$

$$\vdash \xrightarrow[a-b*c]{0\texttt{E}\ 1} \square \vdash \text{yes!}$$

$$\vdash \xrightarrow[\varepsilon]{0}(a-b)*c\square \vdash \xrightarrow[(]{0(2} a-b)*c\square$$

$$\vdash \xrightarrow[(a}{0(2\underline{\texttt{i}}\ 3} -b)*c\square \vdash \xrightarrow[(a}{0(2\texttt{E}\ 6} -b)*c\square$$

$$\vdash \xrightarrow[(a-}{0(2\texttt{E}\ 6-4} b)*c\square \vdash \xrightarrow[(a-b}{0(2\texttt{E}\ 6-4\underline{\texttt{i}}\ 3})*c\square$$

$$\vdash \xrightarrow[(a-b}{0(2\texttt{E}\ 6-4\texttt{E}\ 7})*c\square \vdash \xrightarrow[(a-b}{0(2\texttt{E}\ 6})*c\square$$

$$\vdash \xrightarrow[(a-b)}{0(2\texttt{E}\ 6)9} *c\square \vdash \xrightarrow[(a-b)}{0\texttt{E}\ 1} *c\square$$

$$\vdash \xrightarrow[(a-b)*}{0\texttt{E}\ 1*5} c\square \vdash \xrightarrow[(a-b)*c}{0\texttt{E}\ 1*5\underline{\texttt{i}}\ 3} \square$$

$$\vdash \xrightarrow[(a-b)*c}{0\texttt{E}\ 1*5\texttt{E}\ 8} \square \vdash \xrightarrow[(a-b)*c}{0\texttt{E}\ 1} \square$$

$$\vdash \text{yes!} \vdash$$

$$
\begin{aligned}
M_0 =& \mathsf{S} \ \to \cdot \mathsf{E}\ \$ \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{+} E \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{*} \mathsf{E} \\
& \mathsf{E} \ \to \cdot \underline{(}\ \mathsf{E}\ \underline{)} \\
& \mathsf{E} \ \to \cdot \underline{\mathtt{id}} \\
M_1 =& G(M_i, \mathsf{E}\ ), i = 0,2,4,5 \\
=& \mathsf{S} \ \to \mathsf{E}\ \cdot \$ \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{+} E \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{*} \mathsf{E} \\
M_2 =& G(M_i, \underline{(}\ ), i = 0,2,4,5 \\
=& \mathsf{E} \ \to \underline{(}\ \cdot \mathsf{E}\ \underline{)} \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{+} E \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{*} \mathsf{E} \\
& \mathsf{E} \ \to \cdot \underline{(}\ \mathsf{E}\ \underline{)} \\
& \mathsf{E} \ \to \cdot \underline{\mathtt{id}}
\end{aligned}
\qquad
\begin{aligned}
M_3 =& G(M_i, \underline{\mathtt{id}}\ ), i = 0,2,4,5 \\
=& \mathsf{E} \ \to \underline{\mathtt{id}}\ \cdot \\
M_4 =& G(M_i, \underline{+}\ ), i = 1,6,7,8 \\
=& \mathsf{E} \ \to \mathsf{E} \underline{+}\ \cdot E \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{+} E \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{*} \mathsf{E} \\
& \mathsf{E} \ \to \cdot \underline{(}\ \mathsf{E}\ \underline{)} \\
& \mathsf{E} \ \to \cdot \underline{\mathtt{id}} \\
M_5 =& G(M_i, \underline{*}\ ), i = 1,6,7,8 \\
=& \mathsf{E} \ \to \mathsf{E} \underline{*}\ \cdot E \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{+} E \\
& \mathsf{E} \ \to \cdot \mathsf{E} \underline{*} \mathsf{E} \\
& \mathsf{E} \ \to \cdot \underline{(}\ \mathsf{E}\ \underline{)} \\
& \mathsf{E} \ \to \cdot \underline{\mathtt{id}}
\end{aligned}
\qquad
\begin{aligned}
M_6 =& G(M_2, \mathsf{E}\ ) \\
=& \mathsf{E} \ \to \underline{(}\ \mathsf{E}\ \cdot \underline{)} \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{+} E \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{*} \mathsf{E} \\
M_7 =& G(M_4, \mathsf{E}\ ) \\
=& \mathsf{E} \ \to \mathsf{E} \underline{+} E \cdot \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{+} E \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{*} \mathsf{E} \\
M_8 =& G(M_5, \mathsf{E}\ ) \\
=& \mathsf{E} \ \to \mathsf{E} \underline{*} E \cdot \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{+} E \\
& \mathsf{E} \ \to \mathsf{E}\ \cdot \underline{*} \mathsf{E} \\
M_9 =& G(M_6, \underline{)}\ ) \\
=& \mathsf{E} \ \to \underline{(}\ \mathsf{E}\ \underline{)}\ \cdot \\
M_{10} =& G(M_1, \underline{\$}\ ) \\
=& \mathsf{S} \ \to \mathsf{E}\ \$ \cdot
\end{aligned}
$$

Figure 1: $SLR(0)$ States for an expression grammar

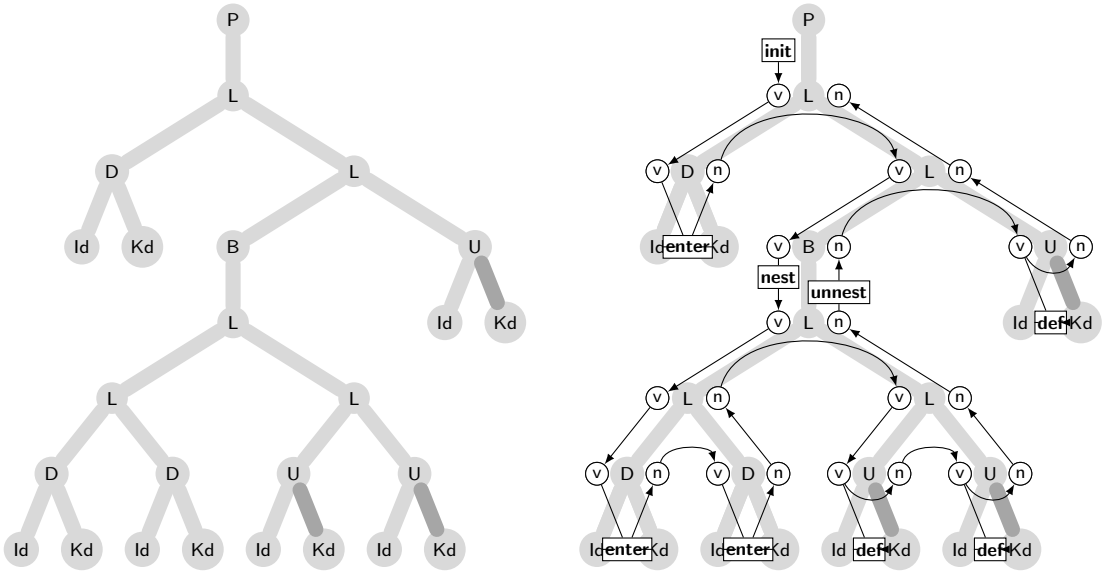
$$
\begin{array}{ll}
\vdash \ \underset{\varepsilon}{0\triangleright}(a-b)c\square \vdash & \underset{(}{0(2\triangleright}a-b)c\square \\
\vdash \ \underset{(a}{0(2\underline{\mathtt{i}}\ 3\triangleright} - b)c\square \vdash & \underset{(a}{0(2\mathsf{E}\ 6\triangleright} - b)c\square \\
\vdash \ \underset{(a-}{0(2\mathsf{E}\ 6-4\triangleright} b)c\square \vdash & \underset{(a-b}{0(2\mathsf{E}\ 6-4\underline{\mathtt{i}}\ 3\triangleright})c\square \\
\vdash \ \underset{(a-b}{0(2\mathsf{E}\ 6-4\mathsf{E}\ 7\triangleright})c\square \vdash & \underset{(a-b}{0(2\mathsf{E}\ 6\triangleright})c\square \\
\vdash \ \underset{(a-b)}{0(2\mathsf{E}\ 6)9\triangleright}c\square \vdash & \text{no!}
\end{array}
$$

$$
\begin{array}{ll}
\vdash \ \underset{\varepsilon}{0\triangleright}(a-b)*-c\square \vdash & \underset{(}{0(2\triangleright}a-b)*-c\square \\
\vdash \ \underset{(a}{0(2\underline{\mathtt{i}}\ 3\triangleright} - b)*-c\square \vdash & \underset{(a}{0(2\mathsf{E}\ 6\triangleright} - b)*-c\square \\
\vdash \ \underset{(a-}{0(2\mathsf{E}\ 6-4\triangleright} b)*-c\square \vdash & \underset{(a-b}{0(2\mathsf{E}\ 6-4\underline{\mathtt{i}}\ 3\triangleright})*-c\square \\
\vdash \ \underset{(a-b}{0(2\mathsf{E}\ 6-4\mathsf{E}\ 7\triangleright})*-c\square \vdash & \underset{(a-b}{0(2\mathsf{E}\ 6\triangleright})*-c\square \\
\vdash \ \underset{(a-b)}{0(2\mathsf{E}\ 6)9\triangleright}*-c\square \vdash & \underset{(a-b)}{0\mathsf{E}\ 1\triangleright}*-c\square \\
\vdash \ \underset{(a-b)*}{0\mathsf{E}\ 1*5\triangleright} - c\square \vdash & \text{no!}
\end{array}
$$

Solution 12: Identification with Linear Visibility

1. The program has the abstract syntaxt tree below left.

2. Applying the attribute rules yields the attrbuted tree below right.



3. The first uses of `y` and `x` refer to `k3` and `k2`, resp.; the last use of `x` refers to `k1`.

Solution 13: Attribute Grammar for Global Visability

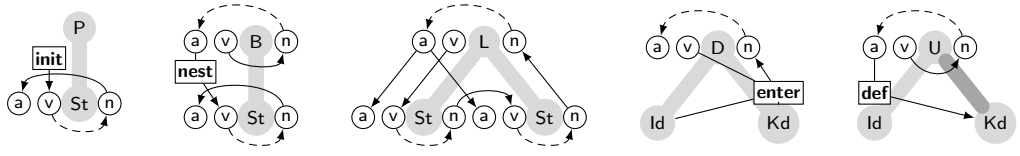
We use the abstract syntax known from Exercise 12. The semantical basis should be extended by one operation:

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

Identification with Global Visibility

If declarations shall be visible if they appaer after (right of) their use, the attribute rulkes have to be extended as follows:

- An attribute (`a`) is added to **St**. It shall contain *all* declarations visible in a block.
- This is the attribute used in **U**.
- Rules **L**, **D**, and **U** collect attributes `v` and `n` as before.
- In rule (**B**), the attribute `a` for the contained statement is defined to be its `n` attribute. Th `v` attribute of the statement is obtained by applying `nest` to the `a` attribute of the block.
- In **L**, the attribute `a` is distributed to all children.



The evaluater for this attribute grammar needs more than one traversal of every node:

1. For each block, the declarations are collected by computing the attributes v and n.
2. Then all uses in that block are evaluated, and all blocks, recursively.

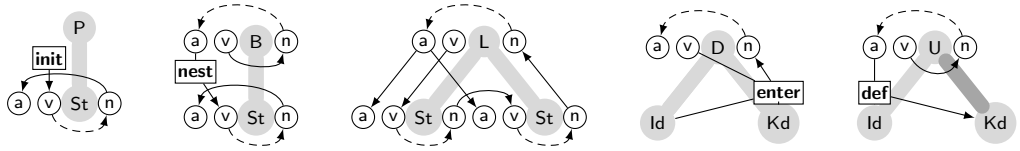
The traversal is nested: The program node and every block has to be traversed twice before the nested blocks are evaluated. In the Haskell function, this is done in two functions, collect: TAB → TAB and idfy: TAB → St → St.

```
idfy' :: Pr -> Pr
idfy' (P s) = P s' where s' = idfy (collect (initial) s)

collect:: TAB -> St -> TAB
collect v (U _ _  ) = n        where n = v
collect v (B s    ) = unnest n where n = collect (nest v) s
collect v (D x k  ) = n        where n = enter v (x,k))
collect v (L s1 s2) = n'       where n  = collect v s1; n' = collect n s2

idfy:: TAB -> St -> St
idfy a (U x _  ) = U x k'    where k' = def a x
idfy a (B s    ) = B s'      where s' = idfy a' s; a' = collect (nest a) s
idfy a (D x k  ) = D x k
idfy a (L s1 s2) = L s1' s2' where s1' = idfy a s1; s2' = idfy a s2
```

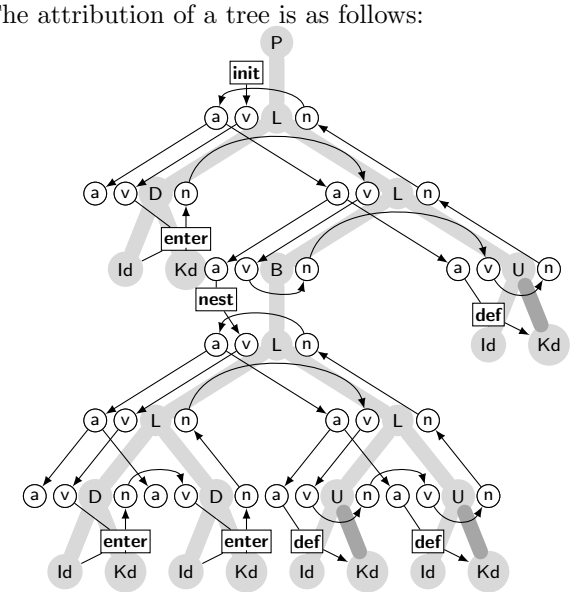
Error handling is not yet included in these definitions.

Solution 14: Attribution with Simultaneous Visibility

Graphically, the attribute rules look as follows:



The attribution of a tree is as follows:



Solution 15: Type Compatibility

Just for fun, I have studied the rather elaborate type coercions in Algol-68.

Type Coercions

Types can be coerced in the following ways::

1. *Widening* extends the precision of numbers: `short short int` $\rightarrow$ `short int` $\rightarrow$ `int` $\rightarrow$ `long int` $\rightarrow$ `long long int`, or extends the kind of numbers: `int` $\rightarrow$ `real` $\rightarrow$ `compl`.
2. *Dereferencing* yields the base type of a reference (pointer): `ref` $\tau \rightarrow \tau$.
3. *Voidening* coerces types to the unit type: $\tau \rightarrow$ `void`.
4. *Deproceduring*) calls a parameterless procedure: `proc` $\tau \rightarrow \tau$.
5. *Uniting* coerceds a type to a union type containing it: $\tau \rightarrow$ `union` $\tau\ \sigma$.
6. *Rowing* coerces values to one-element arrays: $\tau \rightarrow$ `[i:i]` τ.

Contexts for type coercions

The following program parts are subject to coercions:

1. *strong*: Actual to formal parameter of a procedure: all coercions.
2. *firm*: Operands to argument types of operations: no widening, no voiding, no rowing.
3. *meek*: Conditions of if statements: only dereferencing and deproceduring.
4. *weak*: Operands of structure selections and array subscriptions: only dereferencing and deproceduring.
5. *soft*: right-hand side to left-hand side of an assignment: onlx deproceduring.

Further rules define how single coercions can be combined in the different contexts.

Not bad, isn't it?

Solution 16: Declaration Tables in Java

The signatures of the semantic basis is turned into method signatures by considering the table as their receiver object.

We assume that identifiers are represented as numbers in the range $\{0, \ldots, n\}$.

The contents of a table consists of objects of type `Entry`.

```
class Entry {
   int Id;
   Decl decl;
   int level;
   Entry next, global;
   }
```

```
class TAB {
   int currentLevel;
   LinkedList<Entry> nesting;
   Entry[] entries = new Entry[n];

   public void init() {
     nest();
     enter(x_1, d_1);
-- ...
     enter(x_n, d_n);
     };

   public void nest() {
     currentLevel++;
     nesting.addFirst()
     };

   public void unnest() {
     currentLevel--;
     Entry e = nesting.getfirst();
     while e != null {
        entries[nesting.Id] = entries[nesting.Id].global;
        e = e.next
        };
      nesting.removeFirst();
     };

   public void enter(Id x, Decl d) {
      Entry e = new Entry;
      e.Id = x;
      e.decl = d;
      e.level = currentLevel;
      e.next = nesting.getFirst();
      e.global = entries[x];
      entries[x] = e
     };

   public Decl def(Id x) {
      return entries[x].decl
     };

   public bool isLocal(Id x) {
      return (entries[x] != null)
          && (entries[x].level == currentLevel)
     };
   }
```

Solution 17: Declaration Tables for Overloading

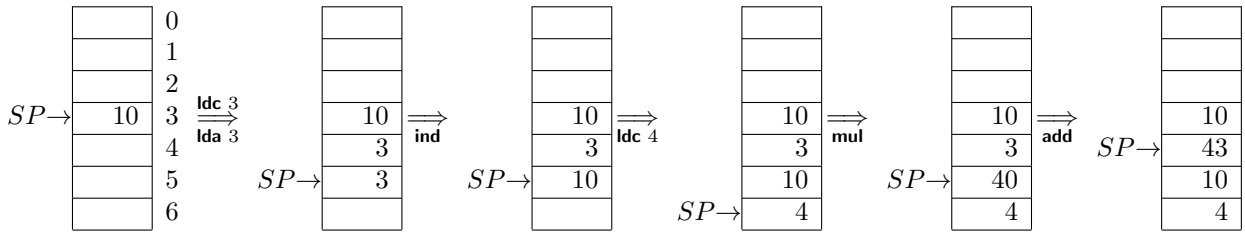
No solution, sorry!

Solution 18: Transforming and Evaluating Expressions

The expression$3 + (x * 4)$ mit $\alpha(x) = 3$ is transformed as follows:

$$\begin{array}{ll} & \text{code}_W(3 + (x * 4))\alpha \\ = & \text{code}_W(3)\alpha; \text{code}_W(x * 4)\alpha; \textsf{add}; \\ = & \textsf{ldc } 3; \text{code}_W(x)\alpha; \text{code}_W(4)\alpha; \textsf{mul}; \textsf{add}; \\ = & \textsf{ldc } 3; \text{code}_A(x)\alpha; \textsf{ind}; \textsf{ldc } 4; \textsf{mul}; \textsf{add}; \\ = & \textsf{ldc } 3; \textsf{lda } 3; \textsf{ind}; \textsf{ldc } 4; \textsf{mul}; \textsf{add}; \end{array}$$

The store of the P machine takes the following states:



Solution 19: Transformating Generalized Expressions

1. *Predefined Functions.*

$$\begin{array}{lll} \text{code}_W(sf(e_1, \ldots e_k))\,\alpha & = & \text{code}_W(e_1)\,\alpha; \\ & & \vdots \\ & & \text{code}_W(e_k)\,\alpha; \\ & & \textsf{jpsf } \alpha(sf) \end{array}$$

Instructions **jpsf** and **retsf** could be defined as follows:

jpsf a	$OPC := PC;\ PC := a$
retsf	$PC := OPC$

For expressions over values that occupy $k > 1$ storage cells, the transformation scheme can stay as it was.

The invariant for code_W must be generalized so that $\text{code}_W(e)$ sets k top cells of the stack for every expression yielding a value of that size. The P-CODE instructions $\textbf{monop}_\oplus$ and $\textbf{dyop}_\otimes$ have to know about the size of their arguments, and how to access their components. The same would hold for predefined functions.

2. *Compound Operands.* The transformation of identifiers and literals would have to be extended as follows:

$$\begin{array}{rcll} \text{code}_W(x)\,\alpha & = & \text{code}_A(x)\,\alpha; \textsf{ind} & \text{wenn } gr(x) = 1 \\ & = & \text{code}_A(x)\,\alpha; \textsf{mvs } gr(x) & \text{sonst} \\ \text{code}_W(l)\,\alpha & = & \textsf{ldc } r_1; \ldots; \textsf{ldc } r_k; & \end{array}$$

Here $(r_1, \ldots, r_k)$ represent the literal l in $k > 1$ storage cells, and the instruction **mvs** ("*move static block*") is defined as follows:

mvs n	**for** $i := n - 1$ **downto** 0 **do** $S[SP + i] := S[S[SP] + i]$ **end** ; $SP := SP + n - 1$

(Copying is done from top to bottom so that the address of the block is overwritten only in the last step.)

Solution 19A Numeric Choice

(*The exercise corresponding to this solution is hidden in the second part of Exercides 18, starting on the last two lines of page 9 in the Exercises.*)
Computed gotos for numeric choices (**case**) are generated as follows:

$$
\begin{array}{ll}
& \text{code}(\textbf{case } e \textbf{ of } u : C_u; \ldots;\ o : C_o \textbf{ default } C_\text{d})\alpha \\
= & \quad \text{code}_W(e)\alpha; \\
& \quad \textbf{ljp } ; u\ l_\text{d} \\
& \quad \textbf{gjp } ; o\ l_\text{d} \\
& \quad \textbf{ldc } u \\
& \quad \textbf{sub}; \\
& \quad \textbf{ijp}; \\
& \quad \textbf{ujp } l_u; \cdots ; \textbf{ujp } l_o; \\
l_u: & \quad \text{code}(C_u)\alpha; \textbf{ujp } l_\text{e}; \\
& \quad \vdots \\
l_o: & \quad \text{code}(C_o\alpha; \textbf{ujp } l_\text{e} \\
l_\text{d}: & \quad \text{code}(C_n)\alpha; \\
l_\text{e}: & \quad \ldots
\end{array}
$$

This kind of transformation is good as long as the range $0..n$ is rather small, since the goto table occupies n code cells. Two instructions have to be executed for every choice, a typical trade-off between storage and time requirements.

Transformation into nested ifs is good for big n, and if many of the selected commands are actually equal.

Numeric choice should be extended to join cases with equal commands. Then the transformation would look as follows:

$$
\begin{array}{lll}
\textbf{case } e & & \textbf{var } t : \mathsf{Integer} := e; \\
\textbf{of } \quad c_{0,1}, \ldots, c_{0,k_0} : C_0; & & \textbf{if } \quad t \in \{c_{0,1}, \ldots, c_{0,k_0}\} \textbf{ then } C_0 \\
\mid \quad c_{1,1}, \ldots, c_{1,k_1} : C_1; & \Rightarrow & \textbf{elsif } t \in \{c_{1,1}, \ldots, c_{1,k_1}\} \textbf{ then } C_1 \\
\mid \quad \vdots & & \quad \vdots \\
\mid \quad c_{n,1}, \ldots, c_{n,k_n} : C_n & & \textbf{elsif } t \in \{c_{n,1}, \ldots, c_{n,k_n}\} \textbf{ then } C_n \\
\textbf{default } C_e & & \textbf{else } C_e
\end{array}
$$

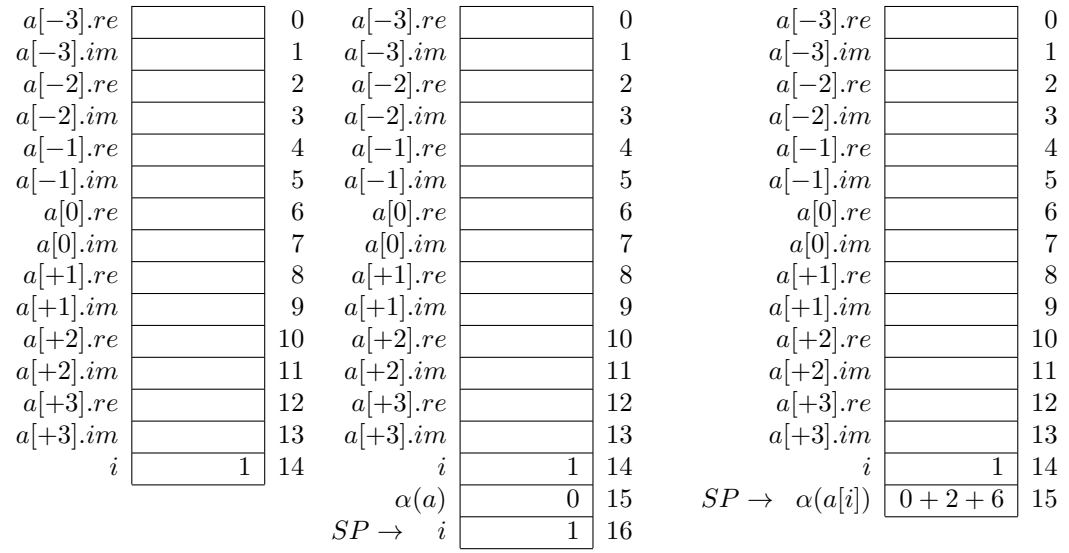
The condition "$t \in \{c_{n,1}, \ldots, c_{n,k_n}\}$" must be expresses as a disjunction "$t = c_{n,1} \vee \cdots \vee c_{n,k_n}\}$" if the language does not provide power sets over the type of e.

Solution 20: Array Transformation

It holds that $gr(\mathsf{Complex}) = 2$ and

$$
\begin{array}{rcl}
gr(\textbf{array } [-3..+3] \textbf{of } \mathsf{Complex}) & = & (o - u + 1) \times gr(\mathsf{Complex}) \\
& = & (3 - (-3) + 1) \times 2 = 7 \times 2 = 14
\end{array}
$$

The address environment defines $\alpha(a) = 0$ and $\alpha(i) = 14$. The transformation is as follows; we



(a) Storing a and i (b) Stack before **ixa** 2; **inc** 6 ... (c) ... and afterwords

Figure 2: Storage and subscription of static arrays

assume staic adresses, and omit the bounds checks "**chk** $u\,o$":

$\text{code}(i := 1)\alpha$
$= \text{code}_A(i)\alpha;\ \text{code}_W(1)\alpha;$ **sto**
$=$ **loa** $\alpha(i)$; **ldc** 1; **sto**
$\text{code}(a[i] := a[i+1])\alpha$
$= \text{code}_A(a[i])\alpha;\ \text{code}_W(a[i+1])\alpha;$ **sto** 2;
$= \text{code}_A(a)\alpha;\ \text{code}_W(i)\alpha;$ **ixa** 2; **inc** 6; $\text{code}_A(a[i+1])\alpha;$ **ind**; **sto** 2;
$=$ **loa** $\alpha(a)$; $\text{code}_A(i)\alpha;$ **ind**; **ixa** 2; **inc** 6; $\text{code}_A(a)\alpha;\ \text{code}_W(i+1)\alpha;$ **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;
$=$ **loa** 0; **loa** $\alpha(i)$; **ind**; **ixa** 2; **inc** 6; **loa** $\alpha(a)$; $\text{code}_W(i)\alpha;\ \text{code}_W(1)\alpha;$ **add**; **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;
$=$ **loa** 0; **loa** 14; **ind**; **ixa** 2; **inc** 6; **loa** 0; $\text{code}_A(i)\alpha;$ **ind**; **ldc** 1; **add**; **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;
$=$ **loa** 0; **loa** 14; **ind**; **ixa** 2; **inc** 6; **loa** 0; **loa** 0; **ind**; **ldc** 1; **add**; **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;

Executing the first command "$i := i + 1$" sets the first storage cell to 1. Then the storage for laocal variables looks as in Figure 2(a). The store operation is as follows, where the modifier is the element size:

sto n	**for** $i := 0$ **to** $n-1$ **do** $S[S[SP-n]+i] := S[SP-n+1+i]$ $SP := SP - n - 1$

Yes, this code is far from optimal.

Solution 21: Storage and Selection of Dynamic Arrays

Code for allocating an array is as it was:

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

Aftewrwards, the storage looks as in the left of Figure 3 ganz links illustriert. The code for `x[i]` is:

$$\begin{aligned}
[\![\texttt{x[i]}]\!]\alpha &= [\![\texttt{x}]\!]_{\mathrm{A}}\,\alpha;\textsf{dpl}\ ;\textsf{ind}\ ;[\![\texttt{i}]\!]_{\mathrm{W}}\,\alpha;\textsf{chd}\ ;\textsf{ixa}\ \mathit{size}(\mathit{type}(\texttt{x[i]}))\ ;\\
&= \textsf{lda}\ \alpha(x)\ ;\textsf{dpl}\ ;\textsf{ind}\ ;\textsf{lda}\ \alpha(\texttt{i})\ ;\textsf{ind}\ ;\textsf{chd}\ ;\textsf{ixa}\ \mathit{size}(\mathit{type}(\texttt{x[i]}))\ ;\\
&= \textsf{lda}\ 10\ ;\textsf{dpl}\ ;\textsf{ind}\ ;\textsf{lda}\ 15\ ;\textsf{ind}\ ;\textsf{chd}\ ;\textsf{ixa}\ 2\ ;\textsf{sli}\ ;
\end{aligned}$$

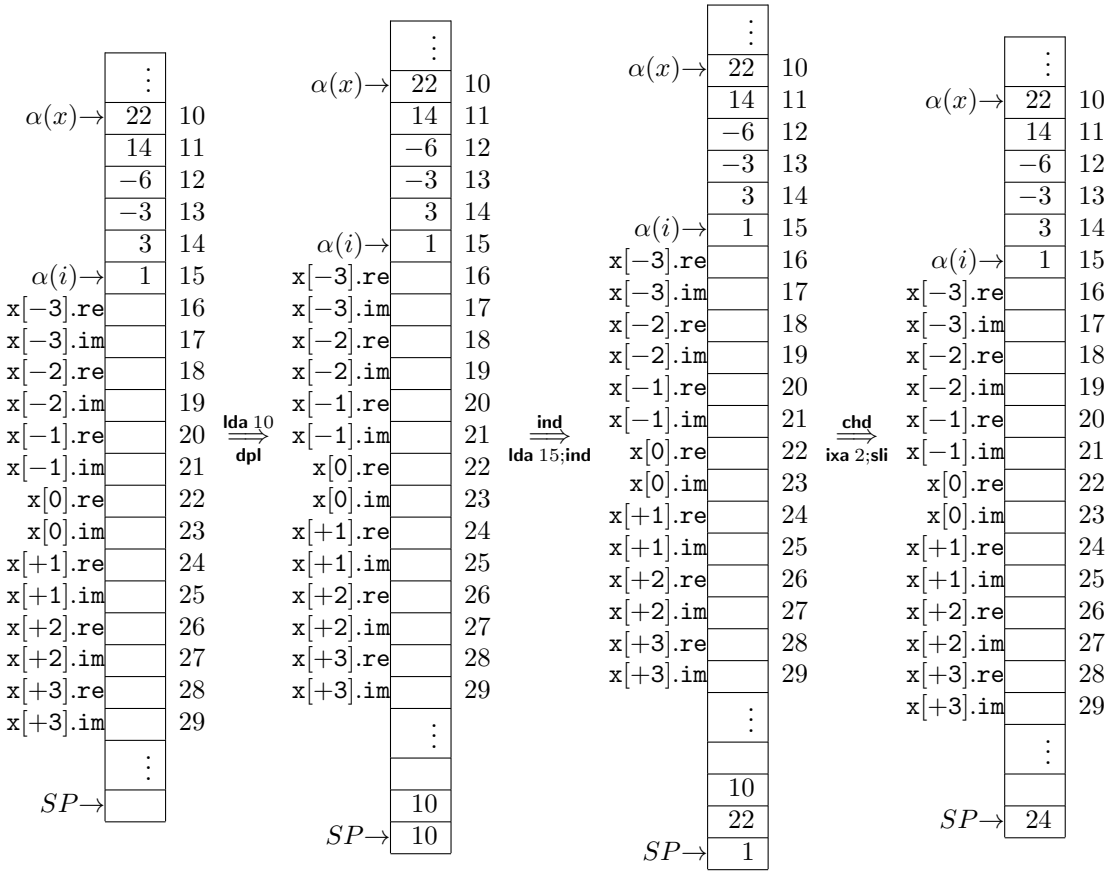


Figure 3: Subscibing a dynamic array

Solution 22: Static Predecessors

The block nesting is fixed, say at $\ell_{\max}$. So static predecessors can be stored in the cells $S[0]$ to $S[\ell_{\max}-1]$.

Actions upon Procedure Entry

Let us assume that procedure p calls q which is declared on level ℓ_q.

1. $S[FP+SVV] := S[\ell_q]$; The old static predecessor is stored in the frame of q.
2. $S[\ell_q] := FP$; the current FP of q is the base address for q.

Actions upon Procedure Exit

1. $S[\ell_q] := S[FP + SVV]$; The previous static predecessor is restored.

Load Instruction for Non-Local Variables

The access operation uses a cell of the displax vector as a basis.

$$\begin{array}{lcl} \textbf{lda}\ a\ \ell & \hat{=} & SP := SP + 1; \\ & & S[SP] := S[S[\ell]] + a; \end{array}$$

ZThis is more efficient if the variable is declared more than one level above its acccess.

Load Instruction for Local Variables

The register FP can be used instead of the static predecessor.

$$\begin{array}{lcl} \textbf{ldl}\ a & \hat{=} & SP := SP + 1; \\ & & S[SP] := S[FP] + a; \end{array}$$

(Access to a register is – a little – more efficient than acces to the storage celle $S[currentLevel]$.

Load Instruction for Global Variables

Global variables have completely static addresses.

$$\begin{array}{lcl} \textbf{ldg}\ a & \hat{=} & SP := SP + 1; \\ & & S[SP] := a; \end{array}$$

Solution 23: Translating Recursive Procedures

The PASCAL function

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

is translated as follows:

$$\begin{array}{l} \text{code}(\textbf{function}\ \text{fac}\ldots)\alpha \\ = \quad f:\ \textbf{ssp}\ l';\ \textbf{sep}\ k';\ \textbf{ujp}\ m'; \\ \quad\ m':\ \text{code}(\textbf{if}\ \mathsf{n} \leqslant 0\ \textbf{then}\ \text{fac} := 1\ \textbf{else}\ \text{fac} := \mathsf{n} * \text{fac}(\mathsf{n}-1))\alpha;\ \textbf{retp} \end{array}$$

The variable fac for the result of the function is on address -2, and its value parameter n on address -1. The size l' of local storage is 4 (for the organization cells) – there are no local variables. The maximal stack growth, needed for register EP, equals 4. The jump to m' can be omitted. The code continues as follows:

$$\begin{array}{rl} = & \textbf{ssp}\ 4;\ \textbf{sep}\ 3; \\ & \text{code}_W(\mathsf{n} \leqslant 0)\alpha;\ \textbf{jpf}\ l''; \\ & \text{code}(\text{fac} := 1)\alpha;\ \textbf{ujp}\ l'''; \\ l'': & \text{code}(\text{fac} := \mathsf{n} * \text{fac}(\mathsf{n}-1))\alpha; \\ l''': & \textbf{retp} \end{array}$$

$=$ **ssp** 4; **sep** 3;
loa 0 (−1); **ind**; **ldc** 0; **leq** ; **jpf** l'';
loa 0 (−2); **ldc** 1; **sto** 1; **ujp** l''';
l'' : **loa** 0 (−2); **loa** 0 (−1); **ind** ; $\text{code}_W(\text{fac}(\mathsf{n}-1))\alpha$; **mul** ; **sto** 1;
l''' : **retp**
$=$ **ssp** 4; **sep** 3;
loa 0 (−1); **ind**; **ldc** 0; **leq** ; **jpf** l'';
loa 0 (−2); **ldc** 1; **sto** 1; **ujp** l''';
l'' : **loa** 0 (−2); **loa** 0 (−1); **ind** ; **mst** 0; $\text{code}_W(\mathsf{n}-1)\alpha$; **cup** 2 f; **mul** ; **sto** 1;
l''' : **retp**
$=$ **ssp** 4; **sep** 3;
loa 0 (−1); **ind**; **ldc** 0; **leq** ; **jpf** l'';
loa 0 (−2); **ldc** 1; **sto** 1; **ujp** l''';
l'' : **loa** 0 (−2); **loa** 0 (−1); **ind** ; **mst** 0; **loa** 0 (−2); **ind** ; **ldc** 1; **sub**; **cup** 2 f; **mul** ; **sto** 1;
l''' : **retp**

In Figure 4 we show the complete code on the left, and the state of the storage in several instances (calls) of fac. On the right, we see the state immediatly before returning from recursion (n equals 0).

Solution 24: Parameter Passing

Constant Parameters

1. Constant parameters are names for immutible values. In contrast to value parameters, no value may be assigned to them.
2. If procedure calls are sequential, the value of the actual parameter need not be copied, as it cannot be changed during the call.
3. Value parameters may save to allocate one local varibale.

Name Parameters

1. We must generate code for name parameters. Every use of the formal parameter jumps to that code in order to evaluate it.
2. This resembles procedure parameters.

   Example: *Jensen's device*explains a rather unexpected feature of name parameters:

```
var a: array [1..n] of Integer;
var i: Integer;
function sum (name a:Integer) : Integer;
  begin
  sum := 0;
  for i:= 1 to n do sum := sum + a;
  end;

  begin
  write (sum(a[i]));
  end.
```

   Function `sum` sums up the elements of the array `a` – not `n*a[i]` for the value of `i` when the procedure is entered. So one avoids the "expensive" passing of arrays.

   This obfuscates the definition of `sum`.

0	**ssp** l
1	**sep** k
2	**ujp** m
$f = 3$	**ssp** l'
4	**sep** k'
5	**loa** $0\,(-1)$
6	**ind**
7	**ldc** 0
8	**leq**
9	**jpf** l''
10	**loa** $0\,(-2)$
11	**ldc** 1
12	**sto** 1
13	**ujp** l'''
$l'' = 14$	**loa** $0\,(-2)$
15	**loa** $0\,1$
16	**ind**
17	**mst** 0
18	**loa** $0\,(-2)$
19	**ind**
20	**ldc** 1
21	**sub**
22	**cup** $2\,f$
23	**mul**
24	**sto** 1
$l''' = 25$	**retp**
$m = 26$	**loa** $0\,0$
27	**mst** 0
28	**ldc** 1
29	**cup** $2\,f$
30	**sto** 1
31	**stp**

0		x
1	0	$\alpha(x)$
2		fac_0
3	2	n_0
4	0	SV
5	0	DV
6	3	EP
7	30	RSA
8	2	$\alpha(\text{fac}_0)$
9	2	$val(n_0)$
10		fac_1
11	1	n_1
12	0	SV
13	4	DV
14	4	EP
15	23	RSA
16	10	$\alpha(\text{fac}_1)$
17	1	$val(n_1)$
18		fac_2
19	0	n_2
20	0	SV
21	12	DV
22	4	EP
23	23	RSA
24	18	$\alpha(\text{fac}_2)$
25	1	

0		x
1	0	$\alpha(x)$
2		fac_0
3	2	n_0
4	0	SV
5	0	DV
6	3	EP
7	30	RSA
8	2	$\alpha(\text{fac}_0)$
9	2	$val(n_0)$
10		fac_1
11	1	n_1
12	0	SV
13	4	DV
14	4	EP
15	23	RSA
16	10	$\alpha(\text{fac}_1)$
17	1	$n_1 * \text{fac}_2$

0		x
1	0	$\alpha(x)$
2		fac_0
3	2	n_0
4	0	SV
5	0	DV
6	3	EP
7	30	RSA
8	2	$\alpha(\text{fac}_0)$
9	2	$n_0 * \text{fac}_1$

Figure 4: Code and procedure frame for `fac`

Solution 25: Translating a Complete Program

The Pascal Program is translated as follows:

$$
\begin{aligned}
&\text{code}(\textbf{program}\ \mathsf{fakultaet}\ \ldots)\alpha \\
=\ &\textbf{ssp}\ l;\ \textbf{sep}\ k;\ \textbf{ujp}\ m;\ \text{code}(\textbf{function}\ \mathsf{fac}\ \ldots)\alpha;\ m : \text{code}(x := \text{fac}(2))\alpha;\ \ \textbf{stp} \\
=\ &\textbf{ssp}\ l;\ \textbf{sep}\ k;\ \textbf{ujp}\ m;\ \text{code}(\ldots)\alpha;\ m : \textbf{loa}\ 0\,0;\ \textbf{mst}\ 0;\ \textbf{ldc}\ 1;\ \textbf{cup}\ 2\,f;\ \textbf{sto}\ 1;\ \textbf{stp}
\end{aligned}
$$

Here x has the address 0, the extension l of local variables is 1, and the maximal stack growth k is 3. (For translating"x := fac(2)", the address of x, the space for the result of fac, and the value 2 of the actual parameter has to be pushed on the stack.)

The translation of fac can be found in Solution 23.

Solution 26: Multiple Inheritance

No solution, sorry!