Languages and Compilers (SProg og Oversættere)

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to Hanne Riis Nielson and Flemming Nielson whose book this lecture is based on.

The missing link

The connection
Between
Syntax and Semantics
And
Languages and Compilers

The Language While

n will range over numerals, **Num**,

x will range over variables, **Var**,

a will range over arithmetic expressions, **Aexp**,

b will range over boolean expressions, **Bexp**, and

S will range over statements, **Stm**.

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & ::= & \mathtt{true} \mid \mathtt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \mathtt{skip} \mid S_1 \,;\, S_2 \mid \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \\
 & \mid & \mathtt{while}\ b\ \mathtt{do}\ S
\end{array}
$$

The Language While can be considered mini version of Mini-Triangle
While is almost the same as the BIMS language

Single step operational semantics for While

$$[\mathrm{ass}_{\mathrm{ns}}] \qquad \langle x := a,\ s\rangle \rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$[\mathrm{skip}_{\mathrm{ns}}] \qquad \langle \mathtt{skip},\ s\rangle \rightarrow s$$

$$[\mathrm{comp}_{\mathrm{ns}}] \qquad \frac{\langle S_1,\ s\rangle \rightarrow s',\ \langle S_2,\ s'\rangle \rightarrow s''}{\langle S_1;S_2,\ s\rangle \rightarrow s''}$$

$$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}] \qquad \frac{\langle S_1,\ s\rangle \rightarrow s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}] \qquad \frac{\langle S_2,\ s\rangle \rightarrow s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}] \qquad \frac{\langle S,\ s\rangle \rightarrow s',\ \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ s'\rangle \rightarrow s''}{\langle \mathtt{while}\ b\ \mathtt{do}\ S,\ s\rangle \rightarrow s''} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}] \qquad \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ s\rangle \rightarrow s \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

A stack based virtual machine - AM

The *instructions* of **AM** are given by the abstract syntax

$$
\begin{array}{lcl}
\mathit{inst} & ::= & \textsc{push-}n \mid \textsc{add} \mid \textsc{mult} \mid \textsc{sub} \\
& \mid & \textsc{true} \mid \textsc{false} \mid \textsc{eq} \mid \textsc{le} \mid \textsc{and} \mid \textsc{neg} \\
& \mid & \textsc{fetch-}x \mid \textsc{store-}x \\
& \mid & \textsc{noop} \mid \textsc{branch}(c, c) \mid \textsc{loop}(c, c) \\
c & ::= & \varepsilon \mid \mathit{inst}{:}c
\end{array}
$$

States:

$$\langle c, e, s \rangle \in \mathbf{Code} \times \mathbf{Stack} \times \mathbf{State}$$

Transitions:

$$\langle c, e, s \rangle \triangleright \langle c', e', s' \rangle$$

Operational semantics for AM

$$\langle \text{PUSH-}n\text{:}c,\ e,\ s\rangle \triangleright \langle c,\ \mathcal{N}[\![n]\!]\text{:}e,\ s\rangle$$

$$\langle \text{ADD:}c,\ z_1\text{:}z_2\text{:}e,\ s\rangle \triangleright \langle c,\ (z_1+z_2)\text{:}e,\ s\rangle \quad \text{if } z_1,\ z_2\in\mathbf{Z}$$

$$\langle \text{MULT:}c,\ z_1\text{:}z_2\text{:}e,\ s\rangle \triangleright \langle c,\ (z_1\star z_2)\text{:}e,\ s\rangle \quad \text{if } z_1,\ z_2\in\mathbf{Z}$$

$$\langle \text{SUB:}c,\ z_1\text{:}z_2\text{:}e,\ s\rangle \triangleright \langle c,\ (z_1-z_2)\text{:}e,\ s\rangle \quad \text{if } z_1,\ z_2\in\mathbf{Z}$$

$$\langle \text{TRUE:}c,\ e,\ s\rangle \triangleright \langle c,\ \mathbf{tt}\text{:}e,\ s\rangle$$

$$\langle \text{FALSE:}c,\ e,\ s\rangle \triangleright \langle c,\ \mathbf{ff}\text{:}e,\ s\rangle$$

$$\langle \text{EQ:}c,\ z_1\text{:}z_2\text{:}e,\ s\rangle \triangleright \langle c,\ (z_1=z_2)\text{:}e,\ s\rangle \quad \text{if } z_1,\ z_2\in\mathbf{Z}$$

$$\langle \text{LE:}c,\ z_1\text{:}z_2\text{:}e,\ s\rangle \triangleright \langle c,\ (z_1\leq z_2)\text{:}e,\ s\rangle \quad \text{if } z_1,\ z_2\in\mathbf{Z}$$

$$\langle \text{AND:}c,\ t_1\text{:}t_2\text{:}e,\ s\rangle \triangleright \begin{cases} \langle c, \mathbf{tt} : e, s\rangle & \text{if } t_1=\mathbf{tt} \text{ and } t_2=\mathbf{tt} \\ \langle c, \mathbf{ff} : e, s\rangle & \text{if } t_1=\mathbf{ff} \text{ or } t_2=\mathbf{ff},\ t_1,\ t_2\in\mathrm{T} \end{cases}$$

$$\langle \text{NEG:}c,\ t\text{:}e,\ s\rangle \triangleright \begin{cases} \langle c, \mathbf{ff} : e, s\rangle & \text{if } t=\mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{if } t=\mathbf{ff} \end{cases}$$

$$\langle \text{FETCH-}x\text{:}c,\ e,\ s\rangle \triangleright \langle c,\ (s\ x)\text{:}e,\ s\rangle$$

$$\langle \text{STORE-}x\text{:}c,\ z\text{:}e,\ s\rangle \triangleright \langle c,\ e,\ s[x\mapsto z]\rangle \quad \text{if } z\in\mathbf{Z}$$

$$\langle \text{NOOP:}c,\ e,\ s\rangle \triangleright \langle c,\ e,\ s\rangle$$

$$\langle \text{BRANCH}(c_1,\ c_2)\text{:}c,\ t\text{:}e,\ s\rangle \triangleright \begin{cases} \langle c_1 : c, e, s\rangle & \text{if } t=\mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{if } t=\mathbf{ff} \end{cases}$$

$$\langle \text{LOOP}(c_1,\ c_2)\text{:}c,\ e,\ s\rangle \triangleright \langle c_1\text{:BRANCH}(c_2\text{:LOOP}(c_1,\ c_2),\ \text{NOOP})\text{:}c,\ e,\ s\rangle$$

Translation of While to AM

$$
\begin{array}{lcl}
\mathcal{CA}[\![n]\!] & = & \text{PUSH-}n \\
\mathcal{CA}[\![x]\!] & = & \text{FETCH-}x \\
\mathcal{CA}[\![a_1 + a_2]\!] & = & \mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]:\text{ADD} \\
\mathcal{CA}[\![a_1 \star a_2]\!] & = & \mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]:\text{MULT} \\
\mathcal{CA}[\![a_1 - a_2]\!] & = & \mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]:\text{SUB} \\
\\
\mathcal{CB}[\![\texttt{true}]\!] & = & \text{TRUE} \\
\mathcal{CB}[\![\texttt{false}]\!] & = & \text{FALSE} \\
\mathcal{CB}[\![a_1 = a_2]\!] & = & \mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]:\text{EQ} \\
\mathcal{CB}[\![a_1 \leq a_2]\!] & = & \mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]:\text{LE} \\
\mathcal{CB}[\![\neg b]\!] & = & \mathcal{CB}[\![b]\!]:\text{NEG} \\
\mathcal{CB}[\![b_1 \wedge b_2]\!] & = & \mathcal{CB}[\![b_2]\!]:\mathcal{CB}[\![b_1]\!]:\text{AND}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{CS}[\![x := a]\!] & = & \mathcal{CA}[\![a]\!]:\text{STORE-}x \\
\mathcal{CS}[\![\texttt{skip}]\!] & = & \text{NOOP} \\
\mathcal{CS}[\![S_1;S_2]\!] & = & \mathcal{CS}[\![S_1]\!]:\mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathcal{CB}[\![b]\!]:\text{BRANCH}(\mathcal{CS}[\![S_1]\!],\mathcal{CS}[\![S_2]\!]) \\
\mathcal{CS}[\![\texttt{while } b \texttt{ do } S]\!] & = & \text{LOOP}(\mathcal{CB}[\![b]\!],\mathcal{CS}[\![S]\!])
\end{array}
$$

Note similarity with code generation templates for Mini-Triangle

Example

$$\mathcal{CS}[\![\mathtt{y:=1;\ while}\ \neg(\mathtt{x=1})\ \mathtt{do}\ (\mathtt{y:=y}\star\mathtt{x;\ x:=x-1})]\!]$$

$$= \mathcal{CS}[\![\mathtt{y:=1}]\!]{:}\mathcal{CS}[\![\mathtt{while}\ \neg(\mathtt{x=1})\ \mathtt{do}\ (\mathtt{y:=y}\star\mathtt{x;\ x:=x-1})]\!]$$

$$= \mathcal{CA}[\![1]\!]{:}\textsc{store-y}{:}\textsc{loop}(\mathcal{CB}[\![\neg(\mathtt{x=1})]\!],\mathcal{CS}[\![\mathtt{y:=y}\star\mathtt{x;\ x:=x-1}]\!])$$

$$= \textsc{push-1}{:}\textsc{store-y}{:}\textsc{loop}(\mathcal{CB}[\![\mathtt{x=1}]\!]{:}\textsc{neg},\mathcal{CS}[\![\mathtt{y:=y}\star\mathtt{x}]\!]{:}\mathcal{CS}[\![\mathtt{x:=x-1}]\!])$$

$$\vdots$$

$$= \textsc{push-1}{:}\textsc{store-y}{:}\textsc{loop}(\textsc{push-1}{:}\textsc{fetch-x}{:}\textsc{eq}{:}\textsc{neg},$$

$$\textsc{fetch-x}{:}\textsc{fetch-y}{:}\textsc{mult}{:}\textsc{store-y}{:}$$

$$\textsc{push-1}{:}\textsc{fetch-x}{:}\textsc{sub}{:}\textsc{store-x})$$

Correctness Proof

Theorem 3.20 For every statement S of **While** we have $\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$.

This theorem relates the behaviour of a statement under the natural semantics with the behaviour of the code on the abstract machine under its operational semantics. In analogy with Theorem 2.26 it expresses two properties:

- If the execution of S from some state terminates in one of the semantics then it also terminates in the other and the resulting states will be equal.
- Furthermore, if the execution of S from some state loops in one of the semantics then it will also loop in the other.

The theorem is proved in two stages as expressed by Lemmas 3.21 and 3.22 below. We shall first prove:

Soundness and completeness

Lemma 3.21 For every statement S of **While** and states s and s', we have that

$$\text{if } \langle S, s\rangle \to s' \text{ then } \langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \varepsilon, s'\rangle$$

So if the execution of S from s terminates in the natural semantics then the execution of the code for S from storage s will terminate and the resulting states and storages will be equal.

Lemma 3.22 For every statement S of **While** and states s and s', we have that

$$\text{if } \langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^{\mathbf{k}} \langle \varepsilon, e, s'\rangle \text{ then } \langle S, s\rangle \to s' \text{ and } e = \varepsilon$$

So if the execution of the code for S from a storage s terminates then the natural semantics of S from s will terminate in a state being equal to the storage of the terminal configuration.

Getting closer to TAM

Exercise 3.7 **AM** refers to variables by their *name* rather than by their *address*. The abstract machine $\mathbf{AM}_1$ differs from **AM** in that

- the configurations have the form $\langle c, e, m \rangle$ where c and e are as in **AM** and m, the *memory*, is a (finite) list of values, that is $m \in \mathbf{Z}^*$, and
- the instructions FETCH-x and STORE-x are replaced by instructions GET-n and PUT-n where n is a natural number (an address).

Specify the operational semantics of the machine. You may write $m[n]$ to select the nth value in the list m (when n is positive but less than or equal to the length of m). What happens if we reference an address that is outside the memory? □

And closer

Exercise 3.8 The next step is to get rid of the operations BRANCH(···,···) and LOOP(···,···). The idea is to introduce instructions for *defining labels* and for *jumping to labels*. The abstract machine $\mathbf{AM}_2$ differs from $\mathbf{AM}_1$ (of Exercise 3.7) in that

- the configurations have the form $\langle pc, c, e, m\rangle$ where c, e and m are as before and pc is the program counter (a natural number) pointing to an instruction in c, and
- the instructions BRANCH(···,···) and LOOP(···,···) are replaced by the instructions LABEL-l, JUMP-l and JUMPFALSE-l where l is a label (a natural number).

The idea is that we will execute the instruction in c that pc points to and in most cases this will cause the program counter to be incremented by 1. The instruction LABEL-l has no effect except updating the program counter. The instruction JUMP-l will move the program counter to the unique instruction LABEL-l (if it exists). The instruction JUMPFALSE-l will only move the program counter to the instruction LABEL-l if the value on top of the stack is **ff**; if it is tt the program counter will be incremented by 1.

Specify an operational semantics for $\mathbf{AM}_2$. You may write $c[pc]$ for the instruction in c pointed to by pc (if pc is positive and less than or equal to the length of c). What happens if the same label is defined more than once? □

Conclusion

- With a bit of hard work it is possible to connect
  - The high level operational semantics for a language
  - With the low level implementation
  - And prove correctness of the translation

- This is called:

Provably correct implementations

- For more information read
  - Semantics with applications,
  - Hanne Riis Nielson and Flemming Nielson
  - Wiley 1992, ISBN 0 471 92980 8