COS**341**, 2022

Project **Part C**

SPECIFICATION

Topic:

Intermediate-Code Generation

Motivation

- In the foregoing Project Part C, which belonged to the **Static Semantics Analysis** phase, we have tried to make sure (as far as we could) that the input program is not only syntactically correct but also “makes sense” – such that we would not generate any output code for meaningless input programs.
- **Now the time has come to generate output code for our meaningful input programs** ☺

Preliminaries

- Our Intermediate Code language will be the ancient programming language BASIC
- It has the advantage that students can see how their generated code can actually run!
- The permissible BASIC commands will be listed on the following slides.
- The principle of translation is the one from Chapter #6 of our book: only a few extra hints will be provided in this slide-show.

Preliminaries: BASIC

- BASIC = **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode
  - **Study it from the Internet!**
- **Only the simplest commands from the original "ancient" BASIC may be used in this Practical** – *not* the "high-level" instructions of nowadays BASIC dialects!
- These most simple commands will be listed on the following slides:

Preliminaries (continued)

- PRINT
- ***** (for multiplication)
- **–** (for subtraction)
- **+** (for addition)
- **“ ”** (for strings)
- GOTO ***n*** (where ***n*** is a line-number)
- GOSUB ***p*** (to call a sub-***procedure***)
- RETURN (to continue with the caller)

➔

Preliminaries (continued)

- Variable names for number-variables: in CAPITAL Letters and Digits according to the BASIC rule-book.
- $ as prefix for String-Variable names, in accordance with the BASIC rule-book.
- Digits for the Integer Numbers, according to the BASIC rule-book.
- < , > , = (smaller, greater, eq *Comparison*)
- LET ... = ..., for value assignments

→

Preliminaries (continued)

- IF *simpleCondition* THEN ***n*** (whereby ***n*** is a line-number) – ***NO*** **“else”!!**
- INPUT *variable*
- END (to indicate the last line of the program in the generated code-file)
- STOP (to enforce the halting of the run)
- **( )** (to structure arithmetic expressions)

➔

TWO Translation Phases

- **Phase 1**:
  - With **Symbolic Label-Addresses**, such as in Chapter #6 of our Textbook
- **Phase 2**:
  - **Replacement of Symbolic Label Addresses by proper Line Numbers** according to the Conventions of BASIC Syntax

Grammar-based Translation

Some additional advice is given on the subsequent slides

- **SPLProgr ➔ ProcDefs main { Algorithm halt ; VarDecl }**
  - Translate the Algorithm as in Chapter #6;
  - Translate the body of the ProcDefs as in Chapter #6, concluded by the RETURN command;
  - The VarDecls do not need any translation at all; they were needed only for the foregoing Semantic analysis phase.
  - finally write the <u>END</u> keyword to indicate the end of the code file.

- **PD → proc *UserDefinedName* { ProcDefs Algorithm return ; VarDecl }**
  - Similar advice as on the previous slide: Translate the Algorithm as in chapter #6 of the book; and append BASIC's RETURN command as the final line of the Algorithms-code.
  - Also here the VarDecl does not need to be translated at all.

- **halt**
  - The corresponding BASIC command is: STOP

- **UnOp ➔ input(VAR)**
  - – Use the INPUT command of BASIC

- **LHS ➔ output**
  - The corresponding BASIC command is PRINT

- **PCall ➔ call *UserDefinedName***
  - – Use BASIC’s command GOSUB address
  - – The address must be the start-address of the code of the body of the named procedure.

- **VAR ➔ *UserDefinedName***
  - Generate target-variable-names as shown in Chapter #6; just make sure that your generated names do not violate the BASIC syntax
  - For String-Typed variables, do not forget the additional BASIC symbol **$**

- **BinOp → add(EXPR,EXPR)**
  - Use BASIC’s **+** for additions

- **BinOp ➔ sub(EXPR,EXPR)**
  - Use BASIC’s **–** for subtractions

- **BinOp ➔ mult(EXPR,EXPR)**
  - Use * for multiplications

- **BRANCH ➔ if(Expr) then { Algorithm } Alternat**
  - To be translated as shown in Chapter #6

- **Alternat ➔ else { Algorithm }**
  - *You are <u>NOT</u> allowed to use the “Else” of nowadays modern BASIC!*
  - *You <u>MUST</u> use the “Jumping” Technique from Chapter 6!*

- **LOOP ➔ while(Expr) do {Algorithm}**
  - Translate it as in Chapter #6
- **LOOP ➔ do { Algorithm } until (Expr)**
  - *is semantically equivalent to:*

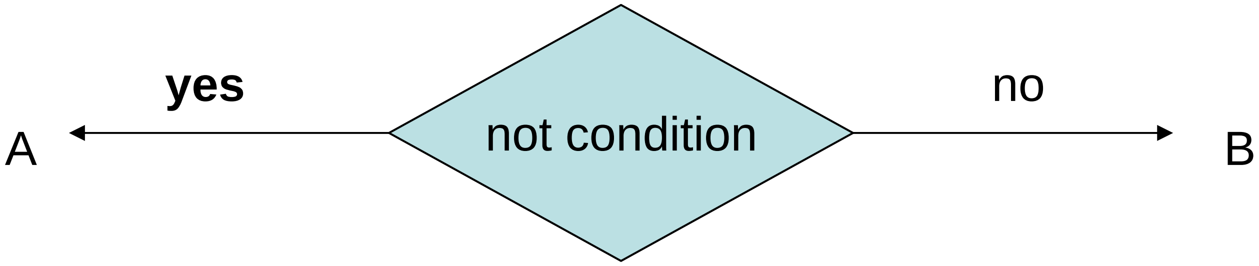
    ```
    exactCopyOfAlgorithm ;
    while(Expr) do {Algorithm}
    ```

  - After this transformation you can translate the transformed program as in Chapter #6

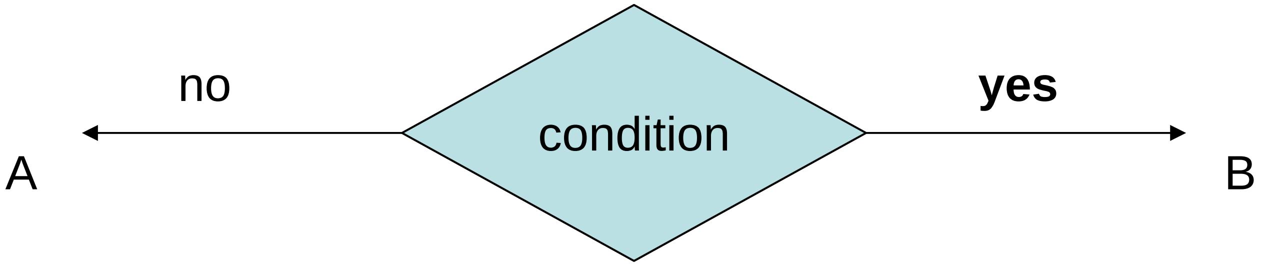
- **BinOp ➔ eq(... , ...)**
  - Use BASIC’s **=**

- **BinOp ➔ larger(Expr, Expr)**
  - Use BASIC’s **>**

- **UnOp ➔ not (Expr)**
  - Translate by way of Branch-Swapping



is *computationally equivalent* to:



- **BinOp ➔ or(Expr,Expr)**
  - Translate with “cascading” GOTO jumps, as in book’s Section 6.6.1 (Figure 6.8)

- **BinOp ➔ and(Expr,Expr)**
  - Translate with “cascading” GOTO jumps, as in Section 6.6.1 (Figure 6.8)

Dead Code Elimination

- In Part B of our project, the Boolean type had special sub-types, **T**,**F**, which the Semantic Analysis could *possibly* reveal.
- In such special cases, we can now re-use this special subtype-information for some code-optimisation (dead code elimination) as follows:

- If some **Expr** is of guaranteed subtype T, then
  - **not(Expr)** is of guaranteed subtype F
  - **or(Expr, ...)** is of guaranteed subtype T
  - **or(..., Expr)** is of guaranteed subtype T

- If some **Expr** is of guaranteed subtype F, then
  - **not(Expr)** is of guaranteed subtype T
  - **and(Expr, ...)** is of guaranteed subtype F
  - **and(..., Expr)** is of guaranteed subtype F

➔

Dead Code Elimination

- **while (Expr) do { Algorithm }**

  is equivalent to **no_code** *if* Expr is of guaranteed subtype **F**

  - Translation can be omitted!
  - Also note: *if* Expr would be of guaranteed subtype **T**, then you could emit an *"infinite loop warning"* already at compile-time!

Dead Code Elimination

- **if (Expr) then { Algorithm } else ALTERNAT**

  is equivalent to **ALTERNAT** *if* Expr is of guaranteed subtype **F**

  – Translation can be simplified!

Dead Code Elimination

- **if (Expr) then { Algorithm } else ALTERNAT**

  is equivalent to **Algorithm** *if* Expr is of guaranteed subtype **T**

  – Translation can be simplified!

OUTPUT of your Compiler

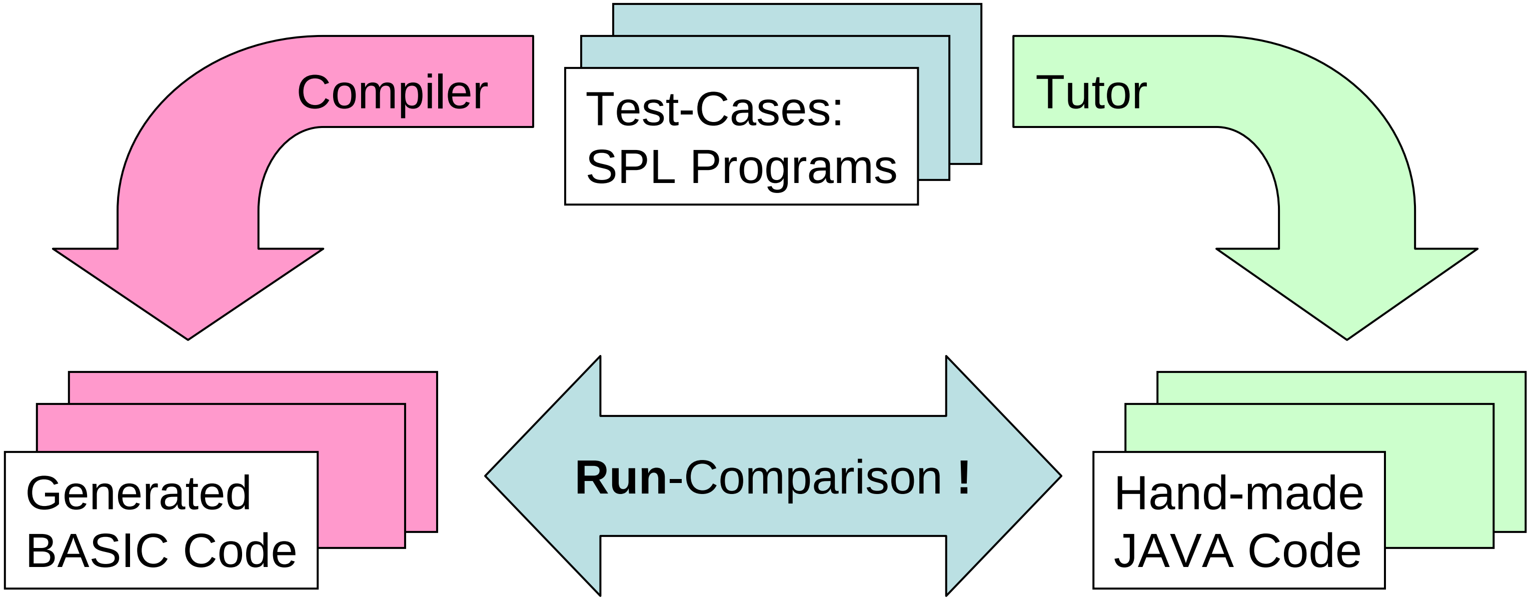
- **For any given SPL Test-Program the syntactically correct BASIC program (with line numbers) must be emitted by your compiler.**
  - **Important:** The Tutor will not only look at the BASIC program as such; he will also let it run to see if it really "works"!
    - Zero marks if you are using "forbidden" syntax of nowadays high-level BASIC! You may only emit the simplistic old-fashioned early BASIC code

GOOD TIP ☺

Before submission,

- Get yourself a FREE “emulator” for BASIC from the Internet, and test-run your own generated BASIC programs already before the Tutor will run it ☺

The Tutor’s Testing Procedure



For each Test-SPL-Program, and for each Input, Tutor will check if your automatically generated BASIC code will produce the same output as the equivalent hand-crafted JAVA program **at run-time!** The Tutor will also check if your BASIC code was generated with the Translation Method described and explained in Chapter #6.

Submission Date = as announced via ClickUp

And **now**…

HAPPY CODING!

☺

Note: Plagiarism is *forbidden!*
Code sharing with other students
is also *not allowed*