Deadline: The final exam’s day

Language Abstract Syntax

Non-terminal Symbols

Command	**Declaration**	**Expression**
Program	**Type-denoter**	**V-Name**

Start Symbol

Program

Syntax

Program	::=	Command	Program
Command	::=	V-Name **:=** Expression	AssignCommand
		| Identifier **(** Expression **)**	CallCommand
		| **if** Expression **then** Command **else** Command	IfCommand
		| **while** Expression **do** Command	WhileCommand
		| **let** Declaration **in** Command	LetCommand
		| **begin** Command **end**	Command
		| Command **;** Command	SequentialCommand
Expression	::=	Integer-Literal	IntegerExpression
		| V-Name	VNameExpression
		| Operator Expression	UnaryExpression
		| Expression Operator Expression	BinaryExpression
V-Name	::=	Identifier	SimpleVName
Declaration	::=	**const** Identifier **~** Expression	ConstDeclaration
		| **var** Identifier **:** Type-denoter	VarDeclaration
		| Declaration **;** Declaration	SequentialDeclaration
Type-denoter	::=	Identifier	SimpleTypeDenoter

1) Extend Triangle with additional loops as follows.

(a) A repeat-command:

repeat C until E

is executed as follows. The *subcommand* C is executed, then the expression E is evaluated. If the value of the expression is *true*, the loop terminates, otherwise the loop is repeated. The *subcommand* C is therefore executed at least once. The type of the expression E must be *Boolean*.

(b) A for-command:

for I from E_1 to E_2 do C

is executed as follows. First, the expressions E_1 and E_2 are evaluated, yielding the integers *m* and *n* (say), respectively. Then the *subcommand* C is executed repeatedly, with identifier I bound in successive iterations to each integer in the range m through n. If m > n, C is not executed at all.
(The scope of I is C, which may use the value of I but may not update it. The types of E_1 and E_2 must be Integer.) Here is an example:

```
for n from 2 to m do
    if prime(n) then
        putint(n)
```

---

2) Extend Triangle with a case-command of the form:

```
case E of
  IL1:  C1;
  ….;
  Iln:  Cn •
  else: C0
```

This command is executed as follows. First E is evaluated; then if the value of E matches the integer-literal IL_i;, the corresponding subcommand Q is executed. If the value of E matches none of the integer-literals, the subcommand C_0 is executed. (The expression E must be of type Integer, and the integer-literals must all be distinct.) Here is an example:

```
case today.m of
  2:   days := if leap then 29 else 28;
  4:   days := 30;
  6:   days := 30;
  9:   days := 30;
 11:   days := 30;
else:  days := 31
```

---

3) Extend Triangle with an initializing variable declaration of the form:

var I := E

This declaration is elaborated by binding / to a newly created variable. The variable's initial value is obtained by evaluating E. The lifetime of the variable is the activation of the enclosing block. (The type of I will be the type of E.)

---

4) Extend Triangle with a new family of types, string n, whose values are strings of exactly n characters (n >= I). Provide string-literals of the form "$c_1...c_n$". Make the generic operations of assignment, '=', and '\ = ' applicable to strings. Provide a new binary operator '<<' (lexicographic comparison).

Finally, provide an array-like string indexing operation of the form 'V[E]', where V names a string value or variable. (Hint: Represent a string in the same way as a static array.)

---

5) Modify the Triangle language processor to perform run-time index checks, wherever necessary. (**Hint**: Add a new primitive routine *rangecheck*. To TAM, as suggested in the following illustration.)

Illustration (could help in solving problem no. 5):

The following Triangle program fragment illustrates array indexing:

```
        let
            var name: array 4 of Char;
            var i: Integer
        in
            begin
            …..
(1)         name [i ] := '   ' ;
            …..
            end
```

Assume that characters and integers occupy one word each, and that the addresses of global variables name and i are 200 and 204, respectively. Thus name occupies words 200 through 203; and the address of name [i] is 200 + i, provided that 0 < i < 3.

The Triangle compiler does not currently generate index checks. The assignment command at **(1)** will be translated to object code like this (omitting some minor details):

```
LOADL  4 8        - fetch the blank character
LOAD   204        - fetch the value of i
LOADL  200        - fetch the address of name [ 0 ]
CALL add          - compute the address of name [ i ]
STOREI            - store the blank character at that address
```

This code is dangerous. If the value of i is out of range, the blank character will be stored, not in an element of name, but in some other variable - possibly of a different type. (If the value of i happens to be 4, then i itself will be corrupted in this way.) We could correct this deficiency by making the compiler generate object code with index checks, like this:

```
LOADL      48             - fetch the blank character
```

```
LOAD      204          - fetch the value of i
LOADL     0            - fetch the lower bound of name
LOADL     3            - fetch the upper bound of name
CALL      rangecheck   - check that the index is within range
LOADL     200          - fetch the address of name [ 0 ]
CALL      add          - compute the address of name [ i ]
STOREI                 - store the blank character at that address
```

The index check is italicized for emphasis. The auxiliary routine *rangecheck*, when called with arguments *i, m,* and *n*, is supposed to return *i* if *m* =< *i* <= *n*, or to fail otherwise. The space cost of the index check is three instructions, and the time cost is 0three instructions plus the time taken by *rangecheck* itself.

Software run-time checks are expensive in terms of object-program size and speed. Without them, however, the object program might overlook a run-time error, eventually failing somewhere else, or terminating with meaningless results. And, let it be emphasized, if a compiler generates object programs whose behavior differs from the language specification, it is simply incorrect. The compiler should, at the very least, allow the programmer the option of including or suppressing run-time checks. Then a program's unpredictable behavior would be the responsibility of the programmer who opts to suppress run-time checks.

Whether the run-time check is performed by hardware or software, there remains the problem of generating a suitable error report. This should not only describe the nature of the error (e.g., 'arithmetic overflow' or 'index out of range'), but should also locate it in the source program. An error report stating that overflow occurred at instruction address 1234 (say) would be unhelpful to a programmer who is trying to debug a high-level language program. A better error report would locate the error at a particular line in the source program.

The general principle here is that error reports should relate to the source program rather than the object program. Another example of this principle is a facility to display the current values of variables during or after the running of the program. A simple storage dump is of little value: the programmer cannot understand it without a detailed knowledge of the run-time organization assumed by the compiler (data representation, storage allocation, layout of stack frames, layout of the heap, etc.). Better is a symbolic dump that displays each variable's source-program identifier, together with its current value in source-language syntax.

General Instructions:

- Teams of two to three students are allowed.
- You must submit source code (in Java, of course). **You should clearly mark with comments the code you wrote.**

- The code will be compiled and run on a Windows machine using Eclipse. Please modify the Compiler.java code to use the following main method (properly formatted, of course):

```
public static void main(String[] args) {
    boolean compiledOK;
```

```
    if (args.length != 1) {
        System.out.println("Usage: tc filename");
       System.exit(1);
    }
    String sourceName = args[0];
    compiledOK = compileProgram(sourceName, objectName, false, false);
    if(compiledOK) {
        String [] arg={"obj.tam"};
        Disassembler.main(arg); Interpreter.main(arg);
     }
}
```

This will allow me to test your code more easily.

- The code above assumes that you imported the TAM package; otherwise, use TAM.Disassembler and TAM.Interpreter.

It is very difficult to evaluate properly code that does not compile. Therefore, you may receive a grade of zero on parts of the term project that do not compile. The same holds for code that compiles but does not run at all. Submit your code as a zip archive using the departmental dropbox. Use the Term Project dropbox. Include instructions on how to compile and run your code. Include an example of use.