Nolife: A Language for Practice Implementation

1 Purpose

This document describes the Nolife programming language.[1] It is intended to provide enough detail to allow implementation of a parser (context-free analyzer) and a type checker (context-sensitive analyzer). Because this document will serve as a reference for two labs in the course, it contains information that is irrelevant to the first lab. The wording has been carefully chosen to make clear the distinction between the context-free and context-sensitive properties of Nolife. For example, most of the "Section Notes" describe context-sensitive issues or code generation issues. Read the lab description carefully to ensure that you understand the scope of each lab.

2 Introduction

Nolife is a programming language designed for practice implementation. Nolife is a simplified version of Pascal in which one may perform simple integer and floating point calculations, as well as simple string manipulation. Nolife is intended to be simple enough to implement in a single semester, but powerful enough to illustrate many common features of modern programming languages. Nolife avoids many complications, like block structure, that are of little instructional value while retaining some features, like recursive procedures, that illuminate fundamental problems of compiler design. Each feature included in the language was added specifically to illustrate some problem that arises in the design and implementation of a compiler.

Nolife supports three basic data types: *integer*, *floating point* and *character*. Each of these types may be aggregated into one dimensional arrays. A number of operators are defined for each type. You can assume that the underlying hardware supports integers with 32 bit twos complement arithmetic and floating point with a 32 bit implementation of the IEEE floating point standard.

Control structures in Nolife are limited. It has an *if* statement, a *while* statement, a *case* statement, and a *compound* statement. Procedures may be recursive and they may be declared within the main program, but not within other procedures. (Nolife supports but one level of lexical nesting.)

Nolife does not include support for separate compilation. The compiler is given the entire program to compile in a single compilation step. This makes possible certain types of context-sensitive checking that is difficult otherwise. For example, the compiler can check argument lists at call sites against the definitions of the corresponding formal parameters.

The language is intended to be *strongly typed*; that is, the type of each expression should be determinable at compile time. However, some *coercions* from one type to another will be permitted. Since there is no *boolean* data type, integers are used as logicals in a manner similar to APL and C.

3 Lexical Properties of Nolife

1. In Nolife, blanks are significant.

2. In Nolife, keywords always consist of capital letters. All keywords are reserved; that is, the programmer cannot use a Nolife keyword as the name of a variable. The valid keywords are: `AND`, `ARRAY`, `BEGIN`, `CHARACTER`, `DO`, `ELSE`, `END`, `FLOAT`, `FUNCTION`, `IF`, `INTEGER`, `MOD`, `NOT`, `OF`, `OR`, `PROCEDURE`, `PROGRAM`, `READ`, `RETURN`, `THEN`, `VAR`, `WHILE`, `WRITE`. (Note that Nolife is *case sensitive,* that is, the variable `X` differs from `x`. Thus, `END` is a keyword, but `end` can be a variable name.)

3. The following special characters have meanings in a Nolife program. (See the grammar and notes for details.)

[1] In the words of one of my former students, the languages is called Nolife because you will have no life while working on this project.

{ } ' < > = + − * [ ] ( ) . , ; :

4. Comments are delimited by the characters { and }. A { begins a comment; it is valid in no other context. A } ends a comment; it cannot appear inside a comment. (This means comments may not be nested. { can appear in a comment; the first } closes the comment.) Comments may appear before or after any other token.

5. Identifiers are written with upper and lowercase letters and are defined as follows:

   Letter → a | b | c | ··· | z | A | B | ··· | Z
   Digit → 0 | 1 | 2 | ··· | 9
   Identifier → *Letter* (*Letter* | *Digit*)*

   The implementor may restrict the length of identifiers so long as identifiers of at least 31 characters are legal.

6. Constants are defined as follows:

   Constant → *Intnum* | *Floatnum* | *CharConstant*
   Positive → 1 | 2 | 3 | ... | 9
   Sign → + | - | ϵ
   Intnum → *Positive Digit** | 0
   Floatnum → *Intnum* .
   | *Intnum* . *Intnum*
   | *Intnum* . E *Sign Intnum*
   | *Intnum* . *Intnum* E *Sign Intnum*
   CharConstant → 'Letter'

   Special string constants are acceptable in WRITE statements:

   StringConstant → '*Letter**'

7. Relational operators are defined:

   Relop → < | <= | >= | > | = | <>

   Note: <> denotes inequality.

8. Operators:

   Addop → + | -

   Mulop → * | MOD

   Logop → OR | AND

4 Nolife Syntax

This section gives a syntactical description of Nolife. The sections following the grammar provide implementation notes on the various parts of the grammar.

The grammar, as stated, defines the language. It may require some massaging before implementation with any particular parser generator system. For example, the grammar uses "empty" productions in the *Declarations* and *SubprogramDeclarations* sections to concisely express the legal options. In an actual parser, these empty productions might be elevated into alternate forms of the *program* production.

4.1 BNF

The following grammar describes the context-free syntax of Nolife:

Program → `PROGRAM` *Identifier*;
Decls
SubprogramDecls
CompoundStatement

{*Decls*} → `VAR` *DeclList*
| ϵ

DeclList → *IdentifierList* `:` *Type* `;`
| *DeclList IdentifierList* `:` *Type* `;`

IdentifierList → *Identifier*
| *IdentifierList* `,` *Identifier*

Type → *StandardType*
| *ArrayType*

StandardType → `INTEGER`
| `FLOAT`
| `CHARACTER`

ArrayType → `ARRAY [` *Dim* `] OF` *StandardType*

Dim → *Intnum* `..` *Intnum*
| *CharConstant* `..` *CharConstant*

SubprogramDecls → *SubprogramDecls SubprogramDecl* `;`
| ϵ

SubprogramDecl → *SubprogramHead Decls CompoundStatement*

SubprogramHead → `FUNCTION` *Identifier Arguments*: *StandardType* `;`
| `PROCEDURE` *Identifier Arguments* `;`

Arguments → `(` *ParameterList* `)`
| ϵ

ParameterList → *IdentifierList* `:` *Type*
| *ParameterList* `;` *IdentifierList* `:` *Type*

Statement → *Assignment*
| *IfStatement*
| *WhileStatement*
| *CaseStatement*
| *ProcedureInvocation*
| *IOStatement*
| *CompoundStatement*
| *ReturnStatement*

Assignment → *Variable* `:=` *Expr*

IfStatement → `IF` *Expr*
`THEN` *Statement*
`ELSE` *Statement*
| `IF` *Expr* `THEN` *Statement*

WhileStatement → `WHILE` *Expr* `DO` *Statement*

CaseStatement → `CASE` *Expr* `OF` *Cases* `END`

Cases → *CaseElement*
| *Cases* `;` *CaseElement*

CaseElement	→	*CaseLabels* : *Statement*
	|	ϵ
CaseLabels	→	*Constant*
	|	*CaseLabels* , *Constant*
ProcedureInvocation	→	*Identifier*`()`
	|	*Identifier* `(` *ExprList* `)`
IOStatement	→	`READ (` *Variable* `)`
	|	`WRITE (` *Expr* `)`
	|	`WRITE (` *StringConstant* `)`
CompoundStatement	→	`BEGIN` *StatementList* `END`
StatementList	→	*Statement*
	|	*StatementList* `;` *Statement*
ReturnStatement	→	`RETURN` *Expr*
ExprList	→	*Expr*
	|	*ExprList* `,` *Expr*
Expr	→	*Expr Op Expr*
	|	`NOT` *Factor*
	|	*Factor*
Op	→	*LogOp*
	|	*Relop*
	|	*AddOp*
	|	*MulOp*
Factor	→	*Variable*
	|	*Constant*
	|	`(` *Expr* `)`
	|	*ProcedureInvocation*
Variable	→	*Identifier*
	|	*Identifier* `[` *Expr* `]`

4.2 Section Notes

4.2.1 Declarations

Nolife has three standard types: `INTEGER`, `FLOAT` and `CHARACTER`. Integers and floats occupy in a single machine "word", while a character is stored in a single "byte." These standard types may be composed into the structured `ARRAY` type. An identifier may represent one of four types of objects:

1. an integer variable or array
2. a floating point variable or array
3. a character variable or array
4. a procedure or function name

Identifiers are declared to be variables or arrays by a `VAR` declaration, while they are declared to be procedure names by `PROCEDURE` and `FUNCTION` declarations. Only singly dimensioned arrays are permitted in Nolife, but arbitrary upper and lower index bounds are permitted. Arrays may be indexed by characters.

As mentioned earlier, Nolife is case-sensitive. Procedure names are drawn from the same set as variable names. Thus, `foo` can be either a variable or a procedure, but no single name scope can contain both a procedure named `foo` and a variable named `foo`.

Example:

```
VAR x,y :  INTEGER;
   c1, c2, c3 :  CHARACTER;
   a :  ARRAY [ 1 ..  15 ] OF INTEGER;
   s1, s2 :  ARRAY [0 ..  79 ] OF CHARACTER;
   table :  ARRAY [ 'a' ..  'z' ] OF INTEGER;
```

4.2.2 Procedure Declarations

The semantics of function definition are simple. A function returns the value of the expression specified in the first `RETURN` statement that it executes. Note that the grammar allows a `RETURN` statement inside procedures that are not functions.

Example:

```
FUNCTION max ( a, b:  INTEGER ) : INTEGER;
   BEGIN
      IF a < b
         THEN RETURN b
         ELSE RETURN a
      END;
```

4.2.3 Assignment Statement

The assignment statement requires that the *left hand side* (the *Variable* non-terminal) and *right hand side* (the *Expr* non-terminal) evaluate to have the same type. If they have different types, either coercion is required or a context-sensitive error has occurred. The coercion rules for assignment are simple. If both sides are numeric (of type `INTEGER` or `FLOAT`), the right hand side is converted to the type of the left hand side. If either side is is of type `CHARACTER`, both sides must be `CHARACTER` (or else the procedure contains a context-sensitive error).

4.2.4 If Statement

The grammar for the `IF-THEN-ELSE` construct embodies one of the classical solutions to the dangling else ambiguity. It provides a unique binding of the *else-part* to a corresponding *if* and *then-part*.

To evaluate an if statement, the expression is evaluated. If the expression's type is `CHARACTER`, the procedure contains a context-sensitive error. If its type is `FLOAT`, it should be converted to an `INTEGER`. For an integer value, Nolife defines `0` as *false*; any other value is equivalent to *true*.

Examples:

```
IF c=d THEN d := a
IF b=0 THEN b := 2*a ELSE b := b/2
```

4.2.5 While Statement

The while statement provides a simple mechanism for iteration. Nolife's while statement behaves like the while statement in many other languages; it executes the statement in the loop's body until the controlling expression becomes false.

The controlling expression will be treated as a boolean value encoded into an `INTEGER` expression. If the expression is not of type `INTEGER`, the same coercion rules apply as in the if statement.

4.2.6 Case Statement

There are several insteresting implementation issues that arise in the translation of the `CASE` statement.

1. The `CASE` statement executes by picking the case whose label matches the expressionon's run-time value. It executes the code corresponding to the label. It then branches to the first statement *after* the `CASE` statement. There is no "`BREAK`" statement in Nolife; thus the phenomenon of "fall-through" behavior like in C cannot occur. Case labels *must* be unique.
2. The language specifies no ordering on the case labels. You will need to select an evaluation strategy.
3. The expression in the `CASE` statement must evaluate to the same type as the case label constants.
4. The compiler must decide whether to implement a case with multiple labels using a single copy of the code for that case or to replicate the code.

4.2.7 Procedure Invocation

Nolife uses parentheses to indicate invocation and square brackets to indicate subscripting of an array. This simplifies the grammar — many languages use parentheses for both purposes. Of course, this introduces an ambiguity at the context-free level between a function invocation and an array reference.

Nolife passes all parameters as call-by-reference formal parameters. At execution, actual parameters are evaluated left-to-right.

4.2.8 Input-Output Statements

Nolife provides two primitives for input and output. The `READ` and `WRITE` statements are intended to provide direct access to primitives implemented in the target abstract machine.

Examples:

```
READ (x)
WRITE (x+y)
WRITE ('error')
```

4.2.9 Expressions

Nolife expressions compute simple values of type `INTEGER`, `FLOAT`, or `CHARACTER`. For both integer and floating point numbers, addition, multiplication, and comparison are defined. (Division is omitted deliberately.) For characters, comparison is the only defined operation. The standard ASCII collating sequence is assumed.

Coercion: If an expression contains operands of only one type, evaluation is straight forward. When an operand contains mixed types, the situation is more complex. Characters cannot appear as operands of any *Addop* or *Mulop*. Such usage constitutes a context-sensitive error. If an *Addop* or *Mulop* has an `INTEGER` operand and a `FLOAT` operand, the `INTEGER` operand should be converted to a `FLOAT` before the operation is performed.

Relational operators always produce an integer. Comparisons between characters and numbers make no sense; they are illegal. Comparisons between integers and floats produce integer results. To perform the comparison, the integer is converted to a float. For the numbers, comparison is based on both sign and

magnitude. For characters, comparison is based on location in the standard ASCII collating sequence. Any comparison between unlike types constitutes a context-sensitive error.

Note: in an assignment, the value of a numeric expression gets converted to match the type of the variable that appears on its left hand side, except for characters. (See Section 4.2.3)

Booleans: Because Nolife has no booleans, relational expressions are defined to yield integer results. Thus, a relational expression of the form `a = b` is considered to be an arithmetic expression whose value is `1` if the relation holds and `0` otherwise. Hence, both the `IF-THEN-ELSE` and `WHILE` statements test integer values; the expression is considered *false* if it evaluates to `0` and to *true* if it evaluates to anything else. Consider the following example which tests for either of two conditions being true:

```
BEGIN
    READ (a); READ (b); READ (c); READ (d);
    IF (a = b) + (c < d) THEN WRITE ('error')
END
```

Note that relational expressions must be enclosed in parentheses because they have very low precedence. In the above example, `a`, `b`, `c`, and `d` may be variables of any type.

In the above example, the special operator `OR` could have been used. In Nolife the operator `OR` takes two integer operands, two character operands, two floating-pointer operands, or an integer operand and a floating-pointer operand. `OR` produces the result `0` if both operands evaluate to `0`; otherwise, it produces `1`. The operator `AND` evaluates to `1` if both operands are nonzero; otherwise it evaluates to `0`. The unary logical operator `NOT` evaluates to `1` if its argument is zero and to `0` otherwise. The operand of `NOT` must be an integer.

Unary Minus: Nolife does not include either a unary minus operator or an optional negative sign on the front of a numeric constant. If you finish your parser early, consider adding a unary minus to the grammar. Of course, it should have highest precedence.

5 An Example Program

The following program represents a simple example program written in Nolife. This program successively reads pairs of integers from the input file and prints out their greatest common divisor.

```
PROGRAM example;
    VAR x, y :  INTEGER;
    FUNCTION gcd (a,b:  INTEGER):INTEGER;
        BEGIN
            IF b=0
                THEN RETURN a
                ELSE RETURN gcd(b, a MOD b)
        END;
        BEGIN
            READ (x);
            READ (y);
            WHILE (x <> 0) OR (y <> 0) DO
                BEGIN
                    WRITE (gcd (x,y));
                    READ (x);
```

```
            READ (y)
        END
END
```