Functional Specification: CS335 Compilers Project
Group No: 8

Ashish Gupta, Y8140
Shitikanth, Y8480
Anindya Jyoti Roy, Y8078
Sanjay Kumar, Y8448
Rajeev Rathore, Y8398

OUTLINE

This document contains the functional specification of the OBERON-2 compiler built as part of the CS335 course at Indian Institute Of Technology Kanpur.

Source Language: OBERON-2
Target Language: MIPS
Implementation Language: C++
Library Used: STL

BASIC TYPES

The following basic types are included:

1. **Type Assignments** have been provided. For example: Type A = Array N of INTEGER and VAR X: A.
2. **Integer** : Implemented as 4 bytes. Takes values between MIN(INTEGER) and MAX(INTEGER).
3. **Real**: Provided support for the real type upto the intermediate code including type checking and semantic analysis.
4. **Boolean** : Boolean comprises the truth values TRUE or FALSE. We have allocated 4 bytes for this type as well. It can take any complex boolean expression as value. A special back-patching algorithm is used for assignment of boolean values to variables at run time.
   - BOOLEAN V = Expr
   - Generate the code for Expr as usual (maintaining Truelist and Falselist)
   - Create Two Statements-
     statement 1 : *V:=TRUE*
     statement 2 : *V:=FALSE*
   - Backpatch **Truelist** with statement 1 and **Falselist** with stat2

COMPLEX TYPES

1. **Array Type** :
   - **Run time array bound** checking : We maintain the array dimensions during runtime.
   - **Multidimensional array:** Represented in the memory as row major form.
   - **Complex expression** can be used as array indices.
   - Array can passed as **array arguments**: Required declaration of a stack dedicated to saving information of the parent array as the parsing follows the same rule in the grammar to avoid the loss of information.
2. **Record Type** : Consisting of elements which can be of different kind.
3. **Procedure Type**: If a procedure P is assigned to vaiable of type T, the formal parameter list of P and T must match.

TYPE-CHECKING

1. The Oberon2 language is a strongly typed language i.e most of the type checking has to be implemented by the compiler.
2. We are representing types by a tree structure.
3. The root node contains information about the extension or the basic type and the children are the types from which it is derived.
4. Checking for type equivalence requires recursive checking if the trees are identical.
5. Type expansion for arithmetic types has been implemented for type checking phase

CONSTANTS

1. **Integer Constants**: can be both in decimal or hexa-decimal form
2. **Real Constants**: are supported by the scale-factor option
3. **Constant folding** carried on at compile time

OPERATORS

1. **Arithmetic Opertors**
   - + Addition
   - - Subtraction
   - * Multiplication
   - / Real division
   - DIV Integer division
   - MOD Modulus
2. **Logical Oprarotors**

- OR Logical disjunction
- & Logical conjuction
- ˜ neagation

3. **Relational Operators**

   - = equal (assignment is done by :=)
   - # unequal
   - < less
   - <= less or equal
   - > greater
   - >= less or equal

EXPRESSIONS

1. **Designator**: If a designates an array, then a[e] denotes that element of a whose index is the current value of the expression e. The type of e must be an integer type. A designator of the form a[e0, e1, ..., en] stands for a[e0][e1]...[en]. If r designates a record, then r.f denotes the field f of r or the procedure f bound to the dynamic type of r.
2. **Constant Declarations**: A constant declaration associates an identifier with a constant value. A constant expression is an expression that can be evaluated at compile time, including constants and predeclared functions.
3. Real Arithmetic Expression, Integer Arithmetic Expression, Boolean Expression

COMMENTS

They are arbitrary character sequences opened by the bracket (* and closed by *). Comments may be nested.

KEYWORDS

```
ARRAY RETURN
BEGIN THEN
CASE LOOP TYPE
CONST MOD DIV
MODULE VAR
DO NIL WHILE
ELSE OF ELSIF OR
END POINTER
EXIT PROCEDURE
FOR RECORD REPEAT
```

CONDITIONAL STATEMENTS

1. IF-THEN-ELSEIF-THEN-ELSE-END
2. Maintained the true list and false list accordingly with the back patching algorithm

CONTROL FLOW

1. WHILE STATEMENT
2. REPEAT-UNTIL STATEMENT

PROCEDURES

1. Type of a procedure is represented as:
   (return-type)*(arg1-type)*(arg2-type)*...*(argN-type)
2. **Multiple Argument Procedure** : Any number of arguments.
3. The return value of procedure can be directly used in the expression.
4. **Recursive function** call support
5. Activation Record
   - Parameters : passed by the caller
   - Return Address (set by the callee)
   - Frame Pointer (set by the callee)
   - Access Link (set by the caller)
6. Support for output provided by PRINT function.

SCOPING SYSTEM

1. **Static Scoping system** implemented.
2. Support for **identical name for procedures and variables** in different scopes available.
3. Takes help for the **access links** maintained at the time of insertion of activation record.
4. Access link is set up by the caller for the caller activation record.
5. **Accessing parent variables**: Variables is searched for depth recursively in the symbol table of the current and previous environments. After finding the depth, access links are used to generate the corresponding stack address.

OPTIMIZATION: DEAD CODE ELIMINATION

1. We converted intermediate code to **basic blocks** by converting into a **Flow Graph** creating new blocks on a *jump, cjump, call and reachable labels.*
2. After a jump statement, searched for the **first reachable label** (from another jump statement). Till that delete the code.
3. Do a **BFS** on the basic blocks to find reachabilty from the starting block (Main). Delete the unreachable blocks.
4. Constant folding has been used wherever possible, i.e., operations with constant operands are done by the compiler during code-generation itself. These constants are not pushed onto the stack during run-time and space is optimized.

FEATURES NOT TO BE IMPLEMENTED

1. Type Bound Procedures: Globally declared procedures may be associated with a record type declared in the same module.
2. Extensible Record Types: Defining new Records based on existing Records
3. SET data type and SET operators
4. Long, Longint, Pointer type
5. Export, import from modules
6. Dynamic Arrays, Run-time Data Structures
7. WITH STATEMENT