Oberon2 Compiler
CS335: Compiler Project

Ashish Gupta (Y8140)
Shitikanth (Y8480)
Anindya Jyoti Roy (Y8078)
Rajeev Rathore (Y8398)
Sanjay (Y8448)

April 16, 2011

Characterization

- Source Language(S) $\longrightarrow$ **Oberon 2**
- Target Language (T) $\longrightarrow$ **MIPS Assembly Language**
- Implementation Language(I) $\longrightarrow$ $C++$
- Library Used $\longrightarrow$ **STL library**

Oberon 2: Introduction

- Oberon was developed by Prof. Niklaus Wirth as part of the Oberon operating system.
- Oberon was derived from Modula-2 with type extension features and simplified grammar.
- Oberon 2 is an extension of Oberon that adds limited object-oriented features. (Type bound Procedures)

Important features

- Array bounds checking
- Static typing with strong type checking
- Modularity
- Garbage Collection

Organization of Code

oberon.l	lexical analysis
oberon.y	syntax analysis
debug.h, debug.cpp	generating debugging symbols
scope.h, scope.cpp	Symbol table and related functions
type.h, type.cpp	Type tree class and type checking
ic.h, ic.cpp	Intermediate code class representation
icgen.h, icgen.cpp	Generation of intermediate code
reg.h, reg.cpp	Register class, and Register handler
codegen.h, codegen.cpp	Generation of mips32 assembly code
optimize.h, optimize.cpp	Flow graph and dead code elimination

Type Checking

Oberon supports basic types (INTEGER, CHAR, BOOLEAN, REAL etc.) and their extensions with POINTER, ARRAY, RECORD and PROCEDURE.

- We are representing types by a tree structure.
- The root node contains information about the extension or the basic type and the children are the types from which it is derived.
- Graceful error recovery system.
- Type expansion for arithmetic types has been implemented for type checking phase.
- For ex: typeExpansion.m, typeConversion.m, record_type.m

Syntax of Intermediate Language

- LABEL
- JUMP, CJUMP
- CALL, PARAM, RETURN, END
- ARRR (rvalue) , ARRL (lvalue)
- RECR (rvalue) , RECL (lvalue)
- PRINT
- ADD, SUB, MIN, MUL, DIV, MOD, COPY

We generated a strong front end interface which can be easily extended to provide new back end features.

Array and Records

- **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.
- **Record:** Offsets for variables are found using nested symbol tables.
- For ex: arrayArg.m, multiArray.m, record.m

Conditional and Control Structure

- IF, ELSE-IF, ELSE
- Maintained the true list and false list accordingly with the back patching algorithm
- *While statement, Repeat-Until Statements* implemented.
- For ex: ifelse.m, whilearray.m, repeatuntil.m, boolAssign.m

Assigning Boolean Values to variables

- 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

Procedures

- Type of a procedure is represented as: (return-type)*(arg1-type)*(arg2-type)*...*(argN-type)
- **Multiple Argument Procedure** : any number of arguments
- **Recursive function** call support:

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)

The frame pointer is set to point towards the Access Link.
For ex: factorial.m, func_call.m, procArgument.m

Scoping System

- **Static Scoping system** implemented.
- Support for **identical name for procedures and variables** in different scopes available.
- Takes help for the **access links** maintained at the time of insertion of activation record.
- Access link is set up by the caller for the caller activation record.
- **Accessing parent variables**: Variables is searched recursively in the symbol table of the current and previous environments. After finding the depth, access links are used to generate the stack address.
- For ex: scope0.m, scope1.m. scope2.m

Optimization: Dead code elimination

We converted intermediate code to **basic blocks** by converting into a **Flow Graph** creating new blocks on a *jump, cjump, call and reachable labels*.

- After a jump statement, searched for the **first reachable label** (from another jump statement). Till that delete the code.
- Do a **BFS** on the basic blocks to find reachabilty from the starting block (Main). Delete the unreachable blocks.
- Run example: dead1.m, dead2.m

**********Thank You**********