YAPL Compiler Assignment Report

Shahryar Bahmaei

July 27th 2024

3.1 Answers

a) YAPL Data Types Representation

YAPL data types are represented in the compiler using a `Type` object. Both `Symbol` and `Attrib` classes use this `Type` object to represent data types:

```
protected Type type;
```

The `Type` interface/class defines different types such as `IntType`, `BoolType`, `ArrayType`, and `RecordType`.

Why this information is needed in both `Symbol` and `Attrib` objects

In `Symbol` objects:

- Represents the declared type of variables, constants, parameters, and function return types in the symbol table.
- Stores additional type-related information like whether it's a reference type, readonly, or global.
- Used for scope management and type checking during the semantic analysis phase.

In `Attrib` objects:

- Carries type information during parsing and semantic analysis of expressions and statements.
- Used for intermediate representation during code generation.
- Stores additional information like constant status, memory offset, and register allocation.

The relationship between `Symbol` and `Attrib` is that `Attrib` objects can be created from `Symbol` objects, transferring type information seamlessly.

```
public Attrib(Symbol sym) throws YAPLException {
    type = sym.getType();
    // ...
}
```

This dual representation allows the compiler to efficiently perform type checking and code generation throughout the compilation process.

b) Type Compatibility Checks for Arrays and Records

ArrayType

The type compatibility check for arrays is implemented in the `isCompatible` method:

```
@Override
public boolean isCompatible(Type start, Type type) {
    if (!(type instanceof ArrayType))
        return false;
    ArrayType other = (ArrayType) type;
    return base.isCompatible(other.base) &&
        (len < 0 || other.len < 0 || other.len == len);
}
```

RecordType

The type compatibility check for records and prevention of infinite recursion in recursive record declarations is implemented as follows:

```
@Override
protected boolean isCompatible(Type start, Type other) {
    if (!(other instanceof RecordType))
        return false;
    RecordType o = (RecordType) other;
    Iterator<Symbol> i1 = this.fields.iterator();
    Iterator<Symbol> i2 = o.fields.iterator();
    while (i1.hasNext() && i2.hasNext()) {
        Symbol s1 = i1.next();
        Symbol s2 = i2.next();
        if (!s1.getName().equals(s2.getName()))
            return false;
        /* avoid infinite recursion */
        if (s1.getType() == start) {
            if (s2.getType() != start)
                return false;
        } else if (!s1.getType().isCompatible(start, s2.getType()))
            return false;
    }
    return i1.hasNext() == i2.hasNext();
}
```

c) Type Check Errors in Test Files

Test File 1 (test33)

```
Program test33
Procedure void proc(int k)
Begin
   If k < 3 Then Return; EndIf;
   Return k / 10;
End proc;
Begin
End test33.
```

Expected error: `ERROR 37 (line 10, column 15) - illegal return value in procedure proc (not a function)`

Test File 2 (test30)

```
Program test30
Procedure int proc1(int m)
Begin
    Return m % 31;
End proc1;
Procedure int proc2(int m)
Begin
    writeint(m);
End proc2;
Begin
End test30.
```

Expected error: `ERROR 35 (line 15, column 1) - missing Return statement in function proc2`

d) Location of Type Check Implementation

Test File 1 (test33) The type check for return values in a void procedure is implemented in the `Procedure()` method where the return type of the procedure is checked against the type of the returned expression.
Test File 2 (test30) The type check for ensuring a return statement is present in a non-void function is also implemented in the `Procedure()` method where it ensures that all paths in the procedure have a return statement if the return type is non-void.

e) Expressions Requiring Many MIPS Registers

Complex arithmetic expressions, function calls with multiple arguments, array indexing in complex expressions, boolean expressions with short-circuit evaluation, and nested method calls are some expressions that require many MIPS registers allocated simultaneously.

f) Local Variables Storage

Local variables are not always stored on the stack; they can be kept in registers if:

- The variable is used frequently within a small scope.
- The variable has a short lifespan.
- The variable is read-only and not reassigned.
- The function is small with few local variables.

g) Code Generation for If Statements

The target code for If statements is generated using conditional jumps. The jump label for the Else branch is generated unconditionally for simplicity and consistency in code generation.

h) Activation Record for Procedure Call

Caller:

- Parameters
- Return address

Callee:

- Old frame pointer
- Local variables
- Temporary storage

i) Memory Size for Run-time Allocation of Records

The memory size for records is determined by summing up the sizes of all fields in the record and aligning the total size to the word boundary.

3.3 Final Report

a) Implemented Extensions and Test Cases

We have implemented the following extensions:

- While statements
- Multi-dimensional arrays
- Records

b) Questions Regarding Extended YAPL Compiler

1. Translation of While Statements

While statements are translated into MIPS assembler code using conditional jumps. The implementation does not support lazy evaluation of While conditions because it branches if the whole condition expression is false.

2. `allocArray()` Procedure in `run-timelib.yapl`

The `allocArray()` procedure in `run-timelib.yapl` allocates multi-dimensional arrays. It does this by recursively allocating each dimension of the array.

Type Checking Suppression: Type checking is suppressed in the return statement of the procedure to allow returning the base address of the allocated array.

Recursive Array Allocation: The procedure checks if the current dimension is less than the total number of dimensions and recursively allocates the next dimension if true.

```
Procedure int _allocArray(int[] len, int dim, int nDims)
Declare
    int[] a;
    int i;
Begin
    a := new int[len[dim]];
    dim := dim + 1;
    If dim < nDims Then
        i := 0;
        While i < #a Do
            a[i] := _allocArray(len, dim, nDims);
            i := i + 1;
        EndWhile;
    EndIf;
    Return a;        /* return base address of a, suppress type checking! */
End _allocArray;
```

3. Translation of Expressions Involving Selectors

The compiler tracks memory addresses of array and record elements by maintaining a base address and applying offsets for each field or array index. This is implemented in methods that calculate the address of a specific field or array element based on its position within the record or array.

c) Overall Design and Functionality of the Compiler

High-level Explanation:

The compiler translates a YAPL source program to MIPS assembler code through the following stages:

- **Lexical Analysis:** Converts source code into tokens.

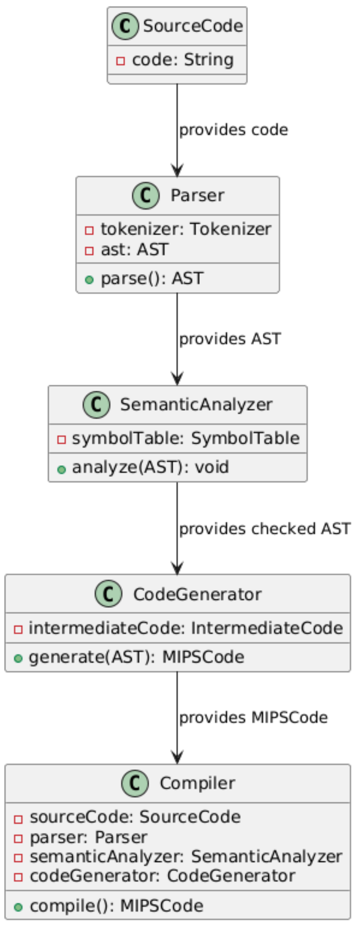

Figure 1: High-level Diagram of Compiler Design

- **Syntax Analysis:** Parses tokens into an Abstract Syntax Tree (AST) based on grammar rules.
- **Semantic Analysis:** Checks for semantic errors and builds symbol tables.
- **Intermediate Code Generation:** Translates the AST into an intermediate representation.
- **Target Code Generation:** Converts the intermediate representation to MIPS assembler code.

Structure of Generated Code:
The generated MIPS assembler code consists of:

- **Data Section:** Declares static data, string literals, and space for global variables.
- **Text Section:** Contains the executable code, including predefined procedures and user-defined functions.
- **Prolog and Epilog:** Manage the procedure call stack, saving and restoring registers.

The overall structure ensures efficient execution by adhering to MIPS conventions and optimizing for common cases such as local variable storage and conditional branching.