The Skeleton

Contents

This document describes the skeletons you will be using in exercises three and four. The descriptions in here should be enough as long as everything goes as planned, but if for some reason you need more information than you can find in this description, you'll just have to read the source code.

1 The Files

You'll find the following files in the `lab3-4` directory:

`Makefile` : This file is used by the `make` utility. You'll only have to modify it if you add new source files.

`Makefile.dependencies` : This file is generated by the `make` utility. There is absolutely no point in modifying or removing this file since it will be regenerated the next time you compile.

`ast.cc` : This file contains the C++ code that implements the functions that print abstract syntax trees. The code is an absolute mess, but if you add a new class to `ast.hh`, you should add code to print it here. You can probably copy one of the other functions.

`ast.hh` : This file declares the classes used to implement the abstract syntax trees. When you start writing code generation functions you'll have to know about many of the classes and their instance variables. There are comments before the declarations that describe what the classes are for.

`codegen.cc` : This file implements the `GenerateCode`, `GenerateCodeAndJump` and `GenerateAssignment` methods of the classes in `ast.hh`. It also implements functions related to the classes in `codegen.hh`. This is the file you'll modify in exercise four, when you write code generation functions.

`codegen.hh` : This file declares the classes that are used for intermediate code. You shouldn't have to modify this file at all, but you may want to refer to it to see which quads are legal.

`main.cc` : This file contains the `main` function, which starts the parsing process. If you want to add a predefined function or command-line option,

this is the place to do it, but otherwise there should be no reason to modify this file.

`parser.y` : This is the parser specification.

`scanner.l` : This is the scanner specification.

`string.hh, string.cc` : These files implement the `string` class, which is used in a bunch of places. You encountered this class in the first exercise.

`symtab.cc` : This file implements all symbol table management functions. You shouldn't need to modify anything here, but one can never know. You will, however, use most of the functions in this file.

`symtab.hh` : This file declares the classes used in the symbol tables, and the symbol tables themselves. You'll encounter the `SymbolInformation` class and the classes derived from it a lot, so become friends with the code in this file.

`parser.cc` : This is the generated parser. It might be fun to look inside to see what it looks like. Keep in mind that this is the `simple` skeleton file. There's one that's much nastier.

`parser.cc.output` : This file is generated when `bison` is run with the `--verbose` option. It contains information about the generated parse table, including all states in DPDA used for parsing. This file is really useful when trying to identify conflicts in the grammar.

`scanner.cc` : This is the generated scanner. It might be fun to look inside to see what it looks like.

`parser.h` : This file is also generated from `parser.y` when `bison` is run with the `--defines` option. It contains C declarations of the union used to return semantic values in the parser, and numeric values for all tokens that the parser recognizes.

2 The Scanner

The scanner is generated using `flex`. You added some of the missing rules in exercise two, so you should be reasonably familiar with it by now. There should be no need to further modify the scanner in exercises three and four.

The scanner is only called from the generated parser.

3 The Parser

The parser is generated using `bison`. You'll add to it in exercises three and four.

When `bison` generates a parser from a grammar specification, it creates parsing tables which are inserted into a *skeleton* file. There are two skeleton files, `bison.simple` and `bison.hairy` included in the `bison` distribution. It is also possible to write your own.

The parser is called from the function `main`, which is in the file `main.cc`. This function initializes the symbol tables, processes the command-line options and then calls the parser function `yyparse`.

4 Symbol Tables

The symbol table is one of the most important data structures in a compiler. The symbol table defines the meaning of all symbols, defines which symbols are visible from where, what type variables have, which functions have been declared and so on.

The skeleton for this exercise has a global symbol table and one symbol table for each function that is defined. The global table records every type defined in any function and all functions and variables defined at the top level. Each function also has a symbol table, in which the symbols defined immediately within the scope of the function are recorded.

The symbol tables map strings to objects of subclasses of `SymbolInformation`. There are currently three subclasses, `FunctionInformation`, which stores information about functions, `VariableInformation`, which stores information about variables and `TypeInformation`, which stores information about types. Each of these classes is described in detail below.

Everything related to symbol tables is defined in `symtab.hh` and `symtab.cc`.

4.1 Information About All Symbols

The `SymbolInformation` class is a superclass for all classes representing different kinds of symbols. It defines a few methods and instance variables that are used by all subclasses. Most of the definition of `SymbolInformation` is shown in figure **??**.

Figure 1 Definition of `SymbolInformation`.

```
class SymbolInformation
{
protected:
    // [...]
public:
    SymbolInformationType       tag;
    string                      id;
    SymbolTable                *table;

    SymbolInformation(SymbolInformationType t, const string &i) :
        tag(t),
        id(i) {};

    virtual FunctionInformation *SymbolAsFunction(void) { return NULL; };
    virtual VariableInformation *SymbolAsVariable(void) { return NULL; };
    virtual TypeInformation     *SymbolAsType(void)     { return NULL; };
    // [...]
};
```

The following fields are of interest:

`id` : This is the name of the symbol, under which it is registered in the symbol table.

`table` : This is a pointer to the symbol table in which the symbol is recorded.

The following methods may also be of use. They are all described in more detail below, in the section about casting from `SymbolInformation*` to pointers to subclasses.

`SymbolAsFunction` : This method is used for casting values of type `SymbolInformation*` to `FunctionInformation*`.

`SymbolAsVariable` : This method is used for casting values of type `SymbolInformation*` to `VariableInformation*`.

`SymbolAsType` : This method is used for casting values of type `SymbolInformation*` to `TypeInformation*`.

4.2 Information About Functions

Information about functions is stored in objects of the class `FunctionInformation`. Most of the definition of `FunctionInformation` is shown in figure **??**.

The private instance variables have the following uses:

`parent` : A pointer to the `FunctionInformation` object for the function in which this function was defined. For example, if function `g` is defined within the scope of function `f`, The `parent` variable in `g` will point to `f`.

`returnType` : A pointer to the `TypeInformation` object representing the return type of the function.

`lastParam` : A pointer to the `VariableInformation` for the *last* formal parameter defined.

`lastLocal` : A pointer to the `VariableInformation` object representing the *last* local variable defined in the function.

`symbolTable` : The function's symbol table. This is where all local variables, formal parameters, local functions and temporary variables are stored.

`body` : A pointer to the abstract syntax tree representing the function's body, or `NULL` if the body has not been defined or there were compilation errors.

`quads` : A pointer to the intermediate code representation of the function body, or `NULL` if it has not yet been generated or there was no function body.

The following methods are defined in `FunctionInformation`:

`SymbolAsFunction` : This method is used to cast values of type `SymbolInformation*` to `FunctionInformation*`. It is described in more detail below.

`SetParent` : Sets the `parent` instance variable.

Figure 2 Definition of `FunctionInformation`.

```
class FunctionInformation : public SymbolInformation
{
protected:
    // [...]
    long temporaryCount;

private:
    FunctionInformation         *parent;
    TypeInformation             *returnType;
    VariableInformation         *lastParam;
    VariableInformation         *lastLocal;
    SymbolTable                  symbolTable;

    StatementList               *body;
    QuadsList                   *quads;

public:

    FunctionInformation(const string& i) :
        SymbolInformation(kFunctionInformation, i),
        parent(NULL),
        returnType(NULL),
        lastParam(NULL),
        lastLocal(NULL),
        body(NULL),
        quads(NULL) {};

    virtual FunctionInformation *SymbolAsFunction(void) { return this; };

    void SetParent(FunctionInformation *);
    void SetReturnType(TypeInformation *);
    void SetBody(StatementList *);

    FunctionInformation *GetParent(void);
    TypeInformation     *GetReturnType(void);
    VariableInformation *GetLastParam(void);
    StatementList       *GetBody(void);

    FunctionInformation *AddFunction(const string&, FunctionInformation *);
    VariableInformation *AddParameter(const string&, TypeInformation *);
    VariableInformation *AddVariable(const string&, TypeInformation *);
    SymbolInformation   *AddSymbol(SymbolInformation *);
    TypeInformation     *AddArrayType(TypeInformation *, int);

    VariableInformation *TemporaryVariable(TypeInformation *type);

    void GenerateCode(void);
    char OkToAddSymbol(const string&);
    SymbolInformation *LookupIdentifier(const string&);
};
```

`SetReturnType` : Sets the `returnType` instance variable.

`SetBody` : Sets the `body` instance variable.

`SetQuads` : Sets the `quads` instance variable.

`GetParent` : Returns the `parent` instance variable.

`GetReturnType` : Returns the `returnType` instance variable.

`GetLastParam` : Returns the `lastParam` instance variable.

`GetBody` : Returns the `body` instance variable

`GetQuads` : Returns the `quads` instance variable.

`AddFunction` : Adds a function to the function's symbol table. The first argument to `AddFunction` is the name of the function to add. The second argument is a pointer to a `FunctionInformation` object representing the function.

`AddVariable` : Adds a variable to the function's symbol table. The first argument to `AddVariable` is the name of the variable to add. The second argument is a pointer to a `VariableInformation` object representing the variable.

`AddSymbol` : Adds a symbol to the function's symbol table. You should never have to call this directly. Use one of the `Add`*Something* functions instead.

`AddArrayType` : Adds an array type to the global symbol table. The first argument is the element type of the array, and the second is the number of elements in the array. This function will construct a symbol for the array type. For example, an array of ten integers will be named `integer<10>`.

`TemporaryVariable` : Returns a freshly allocated `VariableInformation` object with the type set according to the `TypeInformation` given as an argument to `TemporaryVariable`. The new variable is guaranteed to have a name that is unique within the symbol table of the function.

`GenerateCode` : Generates code for the function body and places a pointer to the code in the `quads` field of the `FunctionInformation` record.

`OkToAddSymbol` : Returns non-zero if the argument given to `OkToAddSymbol` is not already occupied by a symbol in the function's symbol table or a type in the global symbol table.

`LookupIdentifier` : Looks up a symbol in the function's scope (its own symbol table and those of it's parents and ancestors) and returns the `SymbolInformation` record representing that symbol. You may have to case the result to one of the `SymbolInformation` subclasses. There's a section below that describes how to do that.

4.2.1 Formal Parameters

The formal parameters of a function are kept in a list in the symbol table. The `lastParameter` variable in a `FunctionInformation` object points to the *last* formal parameter of a function. The list of parameters is kept together by the `prev` field of the `VariableInformation` objects representing the parameters.

Note that this list is stored in reverse order. The last formal parameter is the first element of the list, and the first formal parameter is the last element of the list. This may seem a bit counter-intuitive, but it is practical when building such lists using left-recursive grammars.

To add a formal parameter to this list, call the `AddParameter` method, which adds a parameter to the beginning of the formal parameters list.

4.2.2 Local Variables

The local variables, excluding all temporary variables and formal parameters, are also kept in a list, in a manner similar to the formal paramters. The `lastLocal` instance variable points to the last local variable declared in the function.

Variables that are generated using the `TemporaryVariable` function are not included in this list.

To add a local variable to a function's symbol table, call the `AddVariable` method, which adds the variable to the front of the local variables list.

4.3 Information About Variables

Variables are represented by objects of the `VariableInformation` class. Most of the definition of `VariableInformation` is shown in figure **??**.

Figure 3 Definition of `VariableInformation`.

```
class VariableInformation : public SymbolInformation
{
protected:
    // [...]
public:
    TypeInformation             *type;
    VariableInformation         *prev;

    virtual VariableInformation *SymbolAsVariable(void) { return this; };

    VariableInformation(const string& i) :
        SymbolInformation(kVariableInformation, i) {};
    VariableInformation(const string& i, TypeInformation *t) :
        SymbolInformation(kVariableInformation, i),
        type(t) {};

};
```

The fields of `VariableInformation` have the following uses:

`type` : A pointer to the `TypeInformation` record representing the variable's type. This field *must* be set at all times, or horrible things may happen.

`prev` : A pointer to the previous variable in some list. There are currently two kinds of list: the list of formal parameters in a function and the list of local variables of a function. `prev` is `NULL` in the last element of the list.

The `VariableInformation` class defines the following methods:

`SymbolAsVariable` : This method is used for casting values of type `SymbolInformation*` to `VariableInformation*`. There is a section below that describes how this works.

4.4 Information About Types

Types are represented by objects of the `TypeInformation` class. These are all stored in the global symbol table. When the parser is called the global symbol table contains two types, `integer` and `real`. More types are added when the program declares arrays; each kind of array is given its own type. For example, all variables that are arrays of 10 integers will have the type `integer<10>.`, and all variables that are arrays of 5 reals will have the type `real<5>.`

The global variable `kIntegerType` points to the `TypeInformation` object for the integer type and the global variable `kRealType` points to the `TypeInformation` object for the real type.

If you need to check if two variables have the same type, simply retrieve pointers to the `TypeInformation` records for the types and compare the pointers. Most of the definition of `TypeInformation` is shown in figure ??.

Figure 4 Definition of `TypeInformation`.

```
class TypeInformation : public SymbolInformation
{
protected:
    // [...]
public:
    TypeInformation             *elementType;
    int                          arrayDimensions;
    unsigned long                size;

    virtual TypeInformation     *SymbolAsType(void)      { return this; };

    TypeInformation(const string& i, unsigned long s) :
        SymbolInformation(kTypeInformation, i),
        size(s) {};
};
```

The following fields are used:

`elementType` : For array types, this contains a pointer to the `TypeInformation` record representing the type of elements. For non-array types, this is `NULL`.

`arrayDimensions` : For array types, this is the number of elements in the array. For non-array types, this instance variable is undefined.

`size` : This is the number of bytes required to store the type in memory. It is used for calculating the address of an element in an array.

4.5 Casting from `SymbolInformation*` to a Subclass

Sometimes it is necessary to cast a value that has the type "pointer to `SymbolInformation`" to a pointer to one of the subclasses. For example, the `LookupSymbol` method returns a pointer to a `SymbolInformation`, but in order to use it, it may be necessary to cast it to a pointer to `VariableInformation`, `FunctionInformation` or `TypeInformation`.

This kind of cast is called a *downcast* in C++, and isn't something to be done lightly. Consider the scenario in figure **??**.

Figure 5 Downcasting; first version.

```
class A {
public:
    int x;
};

class B : public A {
public:
    int y;
    B() : y(1) {};
};

class C : public A {
public:
    char *y;
    C() : y("FOO") {};
};
```

```
void f(A *ptr) {
    C *x = (C *)ptr;

    cout << x->y << '\n';
}

void g(void) {
    B var;

    f(&var);
}
```

When the programmer calls `g`, `g` will create an object of type `B`. The call to f is perfectly legal, since `A` is a superclass of `B`. In `f`, the program converts the parameter to a pointer to `C`, and since this is C++, that works too, even though the object pointed to is actually not of type `C`, but of type `B`. The next statement, which tries to print the `y` field, will cause the program to crash.

This is why downcasting is tricky business. You can shoot yourself in your foot by downcasting to the wrong type, and the compiler probably won't even warn you that the gun is loaded.

In recent drafts of the C++ standard, there is a mechanism for safe downcasting, which is based on something called RTTI, which stands for Run-Time Type Information. Since many compilers don't support this, we'll have to try a different trick.

One trick is to include a type tag in the superclass, an instance variable that indicates which type the object really has. Then we can check this variable before downcasting. It looks something like the code in figure **??**.

This solution works fine as long as we don't forget to set the tag, set it to the right value every single time and never forget to check it. Unfortunately, if we have a good compiler, the compiler will warn us about the downcast, even though we know it's safe.

Another solution, which is the one used in the skeleton program, is to do casting using virtual methods. The superclass defines one casting method for each subclass, and the subclasses redefined the casting method that applies to that subclass. The methods are designed to return `NULL` when we attempt an illegal downcast. The solution might look something like figure **??**.

This solution is more elegant in many ways. It does not require us to set a

Figure 6 Downcasting; second version.

```
class A {
public:
    int tag;
    A(int t) : tag(t) {};
};

class B : public A {
public:
    int y;
    B() : A(0), y(1) {};
};

class C : public A {
public:
    char *y;
    C() : A(1), y("FOO") {};
};
```

```
void f(A *ptr) {
    C *x;

    if (ptr->tag == 1) {
        x = (C *)ptr;
        cout << x->y << '\n';
    }
    else
    {
        // Do something else
    }
}

void g(void) {
    B var;

    f(&var);
}
```

Figure 7 Downcasting; final version.

```
class B;
class C;

class A {
public:
    virtual B* Cast_A_to_B(void)
                  { return NULL; };
    virtual C* Cast_A_to_C(void)
                  { return NULL; };
};

class B : public A {
public:
    int y;
    B() : y(1) {};
    virtual B* Cast_A_to_B(void)
                  { return this; };
};
```

```
class C : public A {
public:
    char *y;
    C() : y("FOO") {};
    virtual C* Cast_A_to_C(void)
                  { return this; }
};

void f(A *ptr) {
    C *x = ptr->Cast_A_to_C();

    if (x != NULL)
        cout << x->y << '\n';
    else
        ; // Do something else
}

void g(void) {
    B var;

    f(&var);
}
```

special tag value; the compiler will keep track of the types through the virtual method mechanism. There are no explicit downcasts. The downside is that we have to define a lot of virtual methods in the base class.

In the skeleton for the exercise you can convert a `SymbolInformation` pointer into a `FunctionInformation` pointer using the `SymbolAsFunction` method; into a `VariableInformation` pointer using the `SymbolAsVariable` method; and into a `TypeInformation` pointer using the `SymbolAsType` method.

4.6 Adding Symbols

Symbols are added to symbol tables by calling the `AddParameter`, `AddVariable`, `TemporaryVariable`, `AddFunction` or `AddArrayType` methods of the `FunctionInformation` object to whose symbol table the symbol is to be added. In other words, you'll never have to call methods directly on the `SymbolTable` class.

Before adding a symbol it is a good idea to check if you are allowed to add it at all, by calling the `OkToAddSymbol`. This needs to be done on functions and variables, not on array types.

The functions that add symbols return `VariableInformation`, `FunctionInformation` and `TypeInformation` records. In most cases you can ignore the return value, since it will be identical to one of the arguments you supplied to the function. The exceptions are the functions `AddArrayType` and `TemporaryVariable`, which allocate `TypeInformation` or `VariableInformation` objects of their own.

4.7 Finding Symbols

When you want to look up a symbol, use the `LookupIdentifier` method of the `FunctionInformation` whose symbol table you want to examine. This function searches all the parents of the function, right up to the global symbol table, and returns the first `SymbolInformation` object it finds that has the requested name, or `NULL` if no symbol was found.

You'll probably have to cast the result from `LookupSymbol` to one of the subclasses of `SymbolInformation`. See the section above on casting for information on how to do this.

5 Abstract Syntax Trees

The other really important data structure in the compiler is the abstract syntax tree. As the parser reads its input it constructs an abstract syntax tree, or AST for short, that represents the program. The tree is simple for a second phase to translate to intermediate code.

In an AST there will be nodes of many different kinds, representing different kinds of program constructs. Each type of node carries slightly different information and may have any number of children. In the program skeleton each kind of node has a class of its own, and all classes are derived (sometimes in several steps) from the abstract class `ASTNode`.

Besides the classes that represent actual node types, there are several abstract classes that represent classes of similar node types. For example, the

`Expression` class is a base class for all classes that represent node types that are part of expressions, and the `LeftValue` class is a base class for all types of nodes that can sit on the left-hand side in an assignment.

5.1 The Abstract Classes

This section describes the abstract classes that are included in the hierarchy derived from `ASTNode`. The classes in question are `ASTNode`, `Statement`, `Expression`, `BinaryOperation`, `LeftValue`, `Condition`, `BinaryRelation` and `BinaryCondition`.

5.1.1 `ASTNode`

Base class for all AST node types. Defines the methods and variables needed to print syntax trees, and defines abstract virtual methods implemented by subclasses.

5.1.2 `Statement`

Base class for all types of statements. Its only use is in declarations where all kinds of statements are valid, such as the `statement` field of the `StatementList` class.

5.1.3 `Expression`

Base class for all kinds of expressions. All expressions have a type, which is stored in the `valueType` field of the `Expression` class. The value type is set automatically by the constructors for the derived classes. When manipulating expressions in the parser you will sometimes have to look at the `valueType` field of the objects involved.

The `Expression` class is also used extensively in declarations of other classes.

5.1.4 `BinaryOperation`

Base class for all binary operations within expressions. This class defines two instance variables, `left` and `right`, which represent the left-hand side and the right-hand side of the expression, respectively.

5.1.5 `LeftValue`

Base class for anything that can appear on the left-hand side of an assignment. `LeftValue`s are also expressions, so this class is derived from `Expression`.

`LeftValue` declares a new method, `GenerateAssignment`, which in the subclasses is used to generate code for assigning into whatever the class represents.

5.1.6 `Condition`

Base class for all kinds of conditions. This class is used only in declaring other classes.

5.1.7 `BinaryRelation`

Base class for all binary relations, such as greater than, less than and equals. `BinaryRelation` defines two variables, `left` and `right`, which represent the left-hand side and right-hand side of the relation, respectively.

5.1.8 `BinaryCondition`

Base class for all binary conditions. Currently the only ones are `and` and `or`. `BinaryCondition` defines two instance variables, `left` and `right`, which represent the left-hand side and right-hand side of the condition, respectively.

5.2 The Concrete Classes

These are the classes that you might have reason to create. If you need more detail than is provided here, read the source code in `ast.hh` and `ast.cc`.

All classes have constructors that initialize the instance variables of the object. Unless otherwise mentioned, the constructor takes the same arguments, in the same order, as there are fields listed for each class. For example, if there are two fields, a `Statement` and an `Expression` listed, then the constructor will take a `Statement` pointer and a `Expression` pointer as arguments, and use those to initialize the fields of the object.

5.2.1 `StatementList`

Represents a statement within a list of statements. This class derives from `ASTNode` and defines the following fields:

`statement` : The statement itself (a pointer to a `Statement`.)

`precedingStatements` : The preceding statement in the list (a pointer to a `StatementList`.)

A statement list, like most other lists in the AST and the symbol table, is stored backwards. For example, a three-statement program will be stored as three StatementList objects, each one pointing to a statement through the `statement` field and the list of *preceding* statements through the `precedingStatements` field. Figure **??** shows a simple example of how a statement list is represented.

5.2.2 `IfStatement`

This class represents `if` statements in the program. It derives from `Statement`. An `IfStatement` the following fields:

`condition` : The condition of the if statement (a `Condition`.)

`thenStatements` : The statements in the `then` part of the if statement (a pointer to a `StatementList`.)

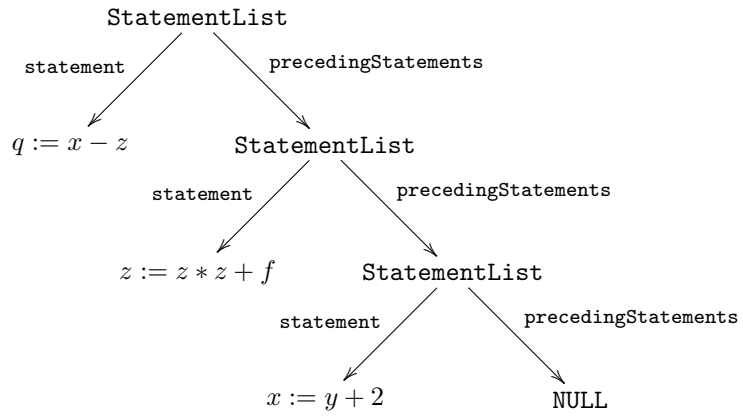
`elseIfList` : The list of `elseif` statements, in reverse order (a pointer to an `ElseIfList`.)

Figure 8 Example of a statement list.

The following small program...

```
begin
    x := y + 2;
    z := z * z + f;
    q := x - z;
end;
```

...is translated to the following sequence of `StatementLists`:



`elseStatements` : The statements in the `else` part of the if statement (a pointer to a `StatementList`.)

See **??** for an example of how an `if` statement is represented.

5.2.3 `ElseIfList`

This class represents an `elseif` branch of an `if` statement. It derives directly from `ASTNode`. The class `ElseIfList` has the following fields:

`preceding` : A pointer to the `ElseIfList` object that represents the previous `elseif` branch of the `if` statement (a pointer to an `ElseIfList`.)

`condition` : The condition in the `elseif` branch (a pointer to a `Condition`.)

`body` : The statements to execute if the condition is true (a pointer to a `StatementList`.)

Note that like statement lists, the list of `elseif` branches is in reverse order. Figure **??** shows an example of an `if` statement with two `elseif` branches.

5.2.4 `Assignment`

This class represents an assignment statement. It derives from the `Statement` class. An `Assignment` has the following fields

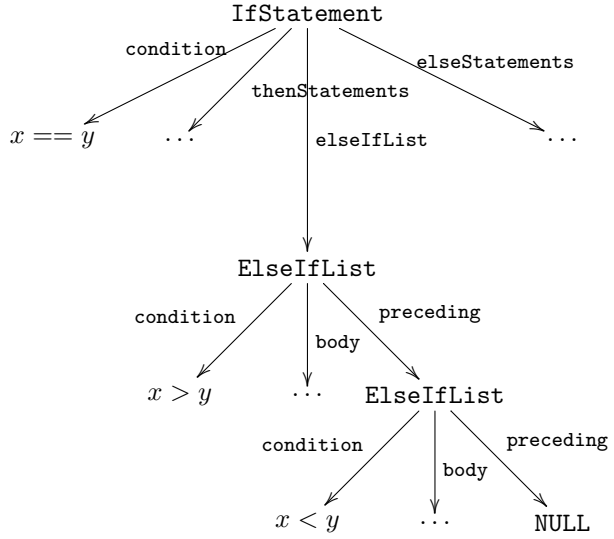
`target` : The object being assigned into (a pointer to a `LeftValue`.)

Figure 9 Example of an `if` statement with `elseif` branches.

The following small program...

```
begin
    if x == y then begin
        ...
    end
    elseif x < y then begin
        ...
    end
    elseif x > y then begin
        ...
    end
    else
        ...
    end if;
end;
```

...is translated to the following structure, rooted in an `IfStatement` (note the reverse order of the `elseif` branches):

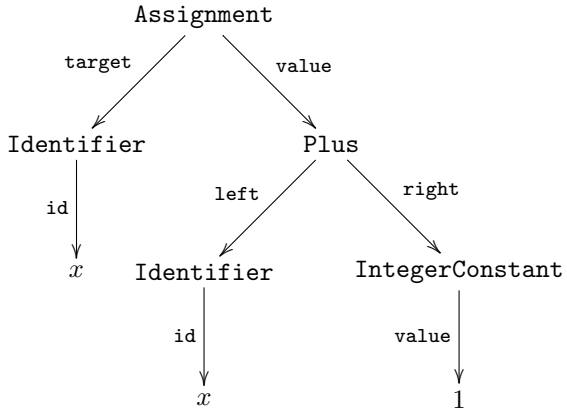


`value` : The `Expression` to assign to the target (a pointer to an `Expression`.)

Figure **??** shows an example of what this structure can look like.

Figure 10 Example of an assignment statement.

The statement `x := x + 1` is translated to the following structure:



5.2.5 `CallStatement`

This class represents a function call. It derives from `Statement`. A `CallStatement` has the following field:

`call` : A pointer to the `FunctionCall` object that represents the actual function call.

The reason there is a `CallStatement` class at all is that the `FunctionCall` derives from `Expression`, and expressions can't appear as statements. We could have let `FunctionCall` inherit from both `Statement` and `Expression`, but multiple inheritance is a nuisance, so we preferred this solution.

5.2.6 `ReturnStatement`

This class represents a `return` statement. It derives from `Statement`. A `ReturnStatement` object has the following fields:

`value` : A pointer to the `Expression` object that represents the value to return.

5.2.7 `WhileStatement`

This class represents a `while` statement. It derives from `Statement`. A `WhileStatement` object has the following fields:

`condition` : The condition for executing the loop (a pointer to a `Condition`.)

`body` : The look body (a pointer to a `StatementList`.)

5.2.8 `ExpressionList`

This class represents an expression within a list of expressions. It is currently only used for parameters in a function call, but it could also be used for things like array constants. It derives directly from `ASTNode` and has the following fields:

`precedingExpressions` : This is a pointer to the `ExpressionList` object representing the *preceding* expression in the list.

`expression` : This is the expression itself (a pointer to an `Expression`.)

Note that expression lists, just like most other lists, are stored in reverse order. Again, this is because it makes the semantic actions in the parser simpler and faster.

There's an example of an `ExpressionList` in figure **??**.

5.2.9 `FunctionCall`

This class represents a function call. It derives from `Expression`. Function calls as individual statements are represented by the `CallStatement` class. A `FunctionCall` object has the following fields:

`function` : The function to call (a pointer to a `FunctionInformation` object.)

`arguments` : The arguments to pass to the function (a pointer to an `ExpressionList`.

Figure **??** shows what a function call might look like.

5.2.10 `IntegerConstant`

This class represents an integer constant. It derives from the `Expression` class. An `IntegerConstant` has the following fields:

`value` : The integer value (a signed long integer.)

5.2.11 `RealConstant`

This class represents a real constant. It derives from the `Expression` class. A `RealConstant` has the following fields:

`value` : The value of the constant (a double.)

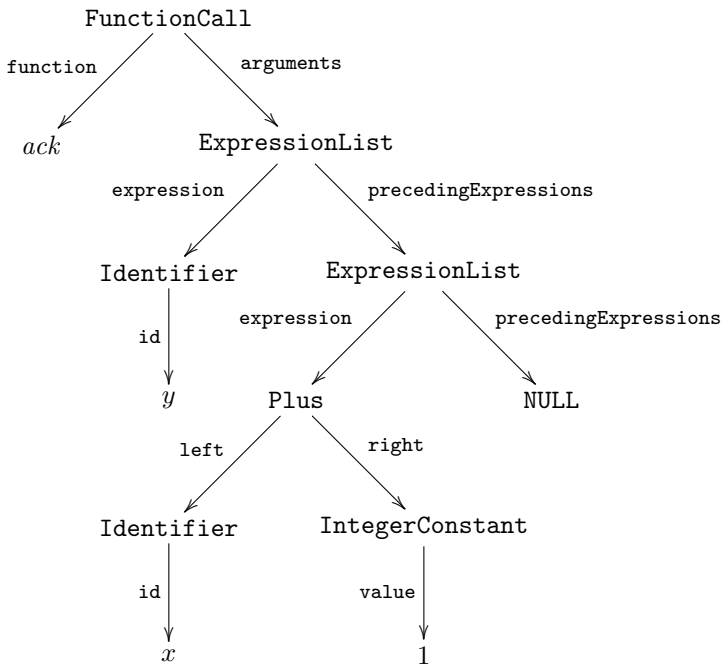
5.2.12 `IntegerToReal`

This class represents the implicit conversion of an integer to a real. It is used in expressions to ensure that the types of the left-hand side and the right-hand side are the same, and in function calls to convert integer values to reals before passing then as real arguments. `IntegerToReal` is derived from `Expression`. The `IntegerToReal` class has the following fields:

`value` : The integer-valued expression whose result needs to be converted to a real (a pointer to an `Expression`.)

Figure 11 Example of a function call.

The function call `ack(x + 1, y)` is translated to the following structure:



5.2.13 `TruncateReal`

This class represents the implicit conversion of a real to an integer. It is used in function calls and assignments when a real value needs to be assigned to an integer variable or parameter. `TruncateReal` is derived from `Expression`. A `TruncateReal` object has the following fields:

`value` : The real-valued expression whose result needs to be converted to an integer (a pointer to an `Expression`.)

5.2.14 `Plus`

This class represents the addition of two expressions. It derives from `BinaryOperator` and does not add any new fields.

5.2.15 `Minus`

This class represents subtraction of two expressions. It derives from `BinaryOperator` and does not add any new fields.

5.2.16 `Times`

This class represents the multiplication of two expressions. It derives from `BinaryOperator` and does not add any new fields.

5.2.17 `Divide`

This class represents the division of two expressions. It derives from `BinaryOperator` and does not add any new fields.

5.2.18 `Power`

This class represents raising one expression to the power of another. It derives from `BinaryOperator` and does not add any new fields.

5.2.19 `UnaryMinus`

This class represents unary negation of an expression. It has the following field:

`right` : A pointer to the expression that is being negated (a pointer to an `Expression`.)

5.2.20 `ArrayReference`

This class represents a reference to an array element, such at `a[x]`. It derives from the abstract class `LeftValue`, and can be used both in assignments and in expressions. An `ArrayReference` object has the following fields:

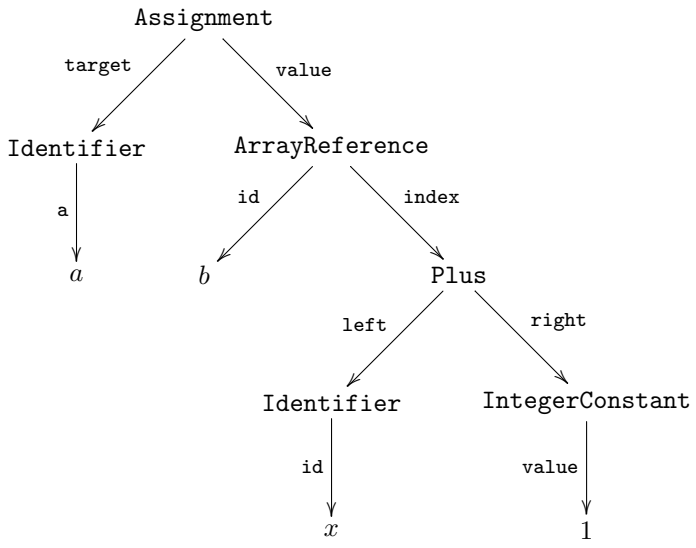
`id` : The variable containing the array being subscripted (a pointer to a `VariableInformation`.)

`index` : The index expression (a pointer to an `Expression`.)

Figure **??** shows an example of this class.

Figure 12 An example of an `ArrayReference` structure.

The expression `a := b[x + 1]` is translated to the following tree:



5.2.21 `Identifier`

This class represents a variable (other types of symbols, such as functions and types, are stored directly in the nodes that use them.) It derives from the `LeftValue` class, so it can appear in assignments and in expressions. An `Identifier` object has the following field:

`id` : A pointer to the `VariableInformation` object for the variable.

5.2.22 `LessThan`

This class represents the less-than (`<`) operator. It derives from the `BinaryRelation` class, and does not define any new fields or methods.

5.2.23 `GreaterThan`

This class represents the greater-than (`>`) operator. It derives from the `BinaryRelation` class, and does not define any new fields or methods.

5.2.24 `GreaterThanOrEqual`

This class represents the greater-than or equal (`>=`) operator. It derives from the `BinaryRelation` class, and does not define any new fields or methods.

5.2.25 `LessThanOrEqual`

This class represents the less-than or equal (`<=`) operator. It derives from the `BinaryRelation` class, and does not define any new fields or methods.

5.2.26 `Equal`

This class represents the equals (`==`) operator. It derives from the `BinaryRelation` class, and does not define any new fields or methods.

5.2.27 `NotEqual`

This class represents the not equal (`<>`) operator. It derives from the `BinaryRelation` class, and does not define any new fields or methods.

5.2.28 `BooleanConstant`

This class represents a boolean constant, `true` or `false`. It derives from the `Condition` class, and as such can be used in conditions. It defined the following field:

`value` : The value of the constant (a `bool`.)

5.2.29 `And`

This class represents the logical `and` operator. It derives from the abstract class `BinaryCondition`, and does not define any new fields or methods.

5.2.30 `Or`

This class represents the logical `or` operator. It derives from the abstract class `BinaryCondition`, and does not define any new fields or methods.

5.2.31 `Not`

This class represents logical negation. It derives from `Condition`, and defines the following field:

`right` : The condition to negate (a pointer to a `Condition`.)

6 Intermediate Code and Quads Lists

Intermediate code is represented by lists of *quads*. Each quad is like a high-level machine instruction. In the general case a quad has three arguments, two operands and one results. Operands and results are usually references to variables in a symbol table, but occasionally an operand will be a constant number.

The quads are generated using the `GenerateCode`, `GenerateCodeAndJump` and `GenerateAssignment` methods of the subclasses of `ASTNode`. The resulting code is stored in the `quads` field of the `FunctionInformation` object for which the code was generated.

6.1 `Quads` and `QuadsLists`

There are two classes you'll encounter when dealing with intermediate code, `Quad` and `QuadsList`. There are other classes as well, but you probably won't be using any of them. Finally, there is a type named `tQuadType` which defines all the names of the quads.

6.1.1 `Quad`

The definition of `Quad` is shown in figure **??**.

The most important field in a `Quad` is `opcode`, which defined what kind of quad the object represents. The remaining fields are the arguments to the quad. Each quad can have three arguments, and each argument can be a reference to a symbol, an integer or a double. You'll probably never have to look directly at the fields in a quad.

There are enough constructors to handle all kinds of operand combinations. When you need to create a quad with fewer than three operands, such as a `label` quad, just pass `NULL` for the unused arguments.

6.1.2 `QuadsList`

The definition of the `QuadsList` is shown in figure **??**.

`QuadsList` contains a nested class named `QuadsListElement`, which is used to link individual quads into a list. You will never have to use this class directly (I hope.)

As you may notice there is no way to get an individual quad or modify the list. This would be a serious omission in a real compiler, but for this

Figure 13 Definition of `Quad`.

```
class Quad
{
private:
    // [...]
public:
    tQuadType        opcode;
    SymbolInformation *sym1;
    SymbolInformation *sym2;
    SymbolInformation *sym3;
    long               int1;
    long               int2;
    long               int3;
    double             real1;
    double             real2;
    double             real3;

    Quad(tQuadType o,
         SymbolInformation *a, SymbolInformation *b, SymbolInformation *c) :
        opcode(o),
        sym1(a),
        sym2(b),
        sym3(c)
        {};

    Quad(tQuadType o, long a, SymbolInformation* b, SymbolInformation* c) :
        opcode(o),
        sym2(b),
        sym3(c),
        int1(a)
        {};

    Quad(tQuadType o, SymbolInformation *a, long b, SymbolInformation *c) :
        opcode(o),
        sym1(a),
        sym3(c),
        int2(b)
        {};

    Quad(tQuadType o,
         double a, SymbolInformation *b, SymbolInformation *c) :
        opcode(o),
        sym2(b),
        sym3(c),
        real1(a)
        {};
    // [...]
}
```

Figure 14 Definition of `QuadsList`.

```
class QuadsList
{
    class QuadsListElement
    {
    public:
        Quad                *data;
        QuadsListElement    *next;

        QuadsListElement(Quad *d, QuadsListElement *n) :
            data(d),
            next(n) {};
        ~QuadsListElement() { delete data; next = NULL; }
    };

    QuadsListElement        *head, *tail;
    static long              labelCounter;

public:
    QuadsList() : head(NULL), tail(NULL) {};

    QuadsList& operator+=(Quad *q);
    long       NextLabel(void) { return (labelCounter += 1); };
};
```

skeleton we've tried to keep the amount of code to a minimum. There is a class named `QuadsListIterator` that can be used to iterate over all elements in a `QuadsList`.

There are two methods of interest defined in `QuadsList`. The first one is `NextLabel`, which simply returns a number that can be used as a label in a `label` quad. The method guarantees that the label will be unique across all `QuadsList`s.

The other method is the operator `+=`. This operator is used to append a quad to the end of a quads list. The example in figure **??** shows how to generate code for the expression `1 + 3`. Note the three statements reading `quads += ...;`. These add quads to the `QuadsList` argument to the function.

6.2 Methods for Code Generation

Code generation is implemented by three virtual methods defined on subclasses of `ASTNode`. The first, `GenerateCode` just generates code for an object. The second, `GenerateCodeAndJump` generates code the same way as `GenerateCode`, but ensures that control is transferred to a specific label and does not continue with the next quad in the quads list. The third method, `GenerateAssignment` is used to generate code for assignments, and is only defined on subclasses of `LeftValue`.

6.2.1 `GenerateCode`

This is the basic code generation method. It has one argument, the `QuadsList` to which new quads are to be appended and it returns one result, a pointer to a `VariableInformation` in which the last result computed value is placed (this only applies to code generation for subclasses of `Expression` and `Condition`.)

Figure 15 Example of appending quads to a `QuadsList`.

```
VariableInformation *OnePlusThree(QuadsList& quads)
{
    VariableInformation     *t1, *t2, *res;

    // Get some temporary variables to work with

    t1 = currentFunction->TemporaryVariable(kInteger);
    t2 = currentFunction->TemporaryVariable(kInteger);
    t3 = currentFunction->TemporaryVariable(kInteger);

    // Add the quads to the quads list

    quads += new Quad(iconst, 1, NULL, t1);
    quads += new Quad(iconst, 3, NULL, t2);
    quads += new Quad(iadd, t1, t2, res);

    return res;
}
```

Figure **??** shows what `GenerateCode` might look like for the `Plus` node. In reality it also checks the types of its operands in order to catch errors made in the parser.

Figure 16 Code generation procedure for `Plus`.

```
VariableInformation::Plus(QuadsList& q)
{
    VariableInformation     *leftInformation, *rightInformation;
    VariableInformation     *result;

    leftInformation = left->GenerateCode(q);
    rightInformation = right->GenerateCode(q);

    result = currentFunction->TemporaryVariable(left->type);
    if (result->type == kIntegerType)
        q += new Quad(iadd, leftInformation,
                            rightInformation,
                            result);
    else if (result->type ==  kRealType)
        q += new Quad(radd, leftInformation,
                            rightInformation,
                            result);
    else
        // An error in the parser. Tell the user.

    return result;
}
```

This example is fairly typical. The `GenerateCode` function first generates code for all its children, then computes a result by combining the results returned from the children, and then returns the `VariableInformation` object for the variable where it stored that result. Note how this example checks the type of the operands and generates different quads depending on the types.

6.2.2 `GenerateCodeAndJump`

`GenerateCodeAndJump` is similar to `GenerateCode`, but it guarantees that the generated code will jump to a specified label and not just continue with the next quad in the quads list. This method is used when an object is generating code for its children and wants to make sure that the code the children generate exit to a loop the object specifies. An example of this is the `if` statement, which calls `GenerateCodeAndJump` for the `elseif` branches.

In addition to the `QuadsList` argument, `GenerateCodeAndJump` takes a second argument, `lbl`, which is the label number to jump to. It returns a `VariableInformation` pointer, just like `GenerateCode`.

6.2.3 `GenerateAssignment`

This method is implemented by subclasses of `LeftValue` and is used to generate the code for an assignment. For assignments, the standard template of "generate code for all children, then add some quads" isn't practical. After all the code required to change an array element is very different from the code required to read it.

Therefore, the `GenerateCode` method for `Assignment` first generates for the right-hand side of the assignment, then calls the `GenerateAssignment` method of the object on the left-hand side.

A `GenerateAssignment` method takes two arguments. The first is the `QuadsList` object to which it should add instructions. The second is a pointer to a `VariableInformation` which contains the result of the expression on the right-hand side of the assignment.

The `GenerateAssignment` does not have a return value.

6.3 The Quads Revealed

This section lists all the quads in some semi-logical order. Each entry starts with the name of the quad and its operands. There are three types of operands, symbols, integers and reals. In the operand listing a symbol operand is listed as `sym`x, an integer operand is listed as `int`x and a real operand is listed as `real`x. An operand titled `lbl` is a label number (integer constant.) The last operand is always the result, unless otherwise stated.

`iconst`	`int1`	`-`	`sym1`

Loads the integer constant `int1` into the variable represented by `sym1`, which must have type `kIntegerType`.

`rconst`	`real1`	`-`	`sym1`

Loads the real constant `real1` into the variable represented by `sym1`, which must have type `kRealType`.

`iaddr`	`sym1`	`-`	`sym2`

Loads the address to the array represented by `sym1` into the variable represented by `sym2`, which must have type `kIntegertype`.

itor	sym1	-	sym2

Converts the integer in the variable `sym1` to a real, and stores the result into `sym2`. `sym1` must have type `kIntegerType` and `sym2` must have type `kRealType`.

rtrunc	sym1	-	sym2

Truncates the real in variable `sym1` and stores the result into `sym2`. `sym1` must have type `kRealType` and `sym2` must have type `kIntegertype`.

iadd	sym1	sym2	sym3

Add the integers in variables `sym1` and `sym2` and store the result into `sym3`. All variables must have type `kIntegerType`.

isub	sym1	sym2	sym3

Subtract the integer in `sym2` from the one in `sym1` and store the result into `sym3`. All variables must have type `kIntegerType`.

imul	sym1	sym2	sym3

Multiply the integer in `sym1` with the one in `sym2` and store the result into `sym3`. All variables must have type `kIntegerType`.

idiv	sym1	sym2	sym3

Divide the integer in `sym1` by the one in `sym2` and store the result into `sym3`. All variables must have type `kIntegerType`.

ipow	sym1	sym2	sym3

Raise the integer in `sym1` to the power of the integer in `sym2` and store the result into `sym3`. All variables must have type `kIntegerType`.

radd	sym1	sym2	sym3

Add the reals in variables `sym1` and `sym2` and store the result into `sym3`. All variables must have type `kRealType`.

rsub	sym1	sym2	sym3

Subtract the real in `sym2` from the one in `sym1` and store the result into `sym3`. All variables must have type `kRealType`.

rmul	sym1	sym2	sym3

Multiply the real in `sym1` with the one in `sym2` and store the result into `sym3`. All variables must have type `kRealType`.

rdiv	sym1	sym2	sym3

Divide the real in sym1 by the one in sym2 and store the result into sym3. All variables must have type kRealType.

rpow	sym1	sym2	sym3

Raise the real in sym1 to the power of the real in sym2 and store the result into sym3. All variables must have type kRealType.

igt	sym1	sym2	sym3

If sym1 is greater than sym2, then store 1 into sym3, otherwise store 0 into sym3. All variables must have type kIntegerType.

ilt	sym1	sym2	sym3

If sym1 is less than sym2, then store 1 into sym3, otherwise store 0 into sym3. All variables must have type kIntegerType.

ieq	sym1	sym2	sym3

If sym1 is equal to sym2, then store 1 into sym3, otherwise store 0 into sym3. All variables must have type kIntegerType.

rgt	sym1	sym2	sym3

If sym1 is greater than sym2, then store 1 into sym3, otherwise store 0 into sym3. sym1 and sym2 must have type kRealType and sym3 must have type kIntegerType.

rlt	sym1	sym2	sym3

If sym1 is less than sym2, then store 1 into sym3, otherwise store 0 into sym3. sym1 and sym2 must have type kRealType and sym3 must have type kIntegerType.

req	sym1	sym2	sym3

If sym1 is equal to sym2, then store 1 into sym3, otherwise store 0 into sym3. sym1 and sym2 must have type kRealType and sym3 must have type kIntegerType.

iand	sym1	sym2	sym3

If sym1 and sym2 are both nonzero, store 1 into sym3, otherwise store 0 into sym3. All variables must have type kIntegertype.

ior	sym1	sym2	sym3

If at least one of sym1 and sym2 is nonzero, store 1 into sym3, otherwise store 0 into sym3. All variables must have type kIntegertype.

inot	sym1	-	sym2

If sym1 is nonzero, store 0 into sym2, otherwise store 1 into sym2. All variables must have type kIntegertype.

jtrue	lbl	sym1	-

Jump to label lbl if sym1 is nonzero. sym1 must be of type kIntegerType.

jfalse	lbl	sym1	-

Jump to label lbl if sym1 is zero. sym1 must be of type kIntegerType.

jump	lbl	-	-

Jump to label lbl.

clabel	lbl	-	-

Place label lbl in the code. This ensures that jumps to lbl start executing the next quad after the label in the quads list (in other words, it does what you would expect.)

istore	sym1	-	sym2

Store the integer in variable sym1 into the memory location indicated by sym2. Both variables must be of type kIntegerType.

iload	sym1	-	sym2

Load the integer in the memory location indicated by sym1 to the variable sym2. Both variables must be of type kIntegerType.

rstore	sym1	-	sym2

Store the real in variable sym1 into the memory location indicated by sym2. sym1 must be of type kRealType and sym2 of type kIntegerType.

rload	sym1	-	sym2

Load the integer in the memory location indicated by sym1 to the variable sym2. sym1 must be of type kRealType and sym2 of type kIntegerType.

creturn	-	-	sym1

Causes the currently executing function to return the value in sym1. The type of sym1 must be identical with the return type of the currently executing function.

param	sym1	-	-

Indicate that sym1 is the next parameter to the next function call. If a function takes three parameters, then you need to generate three param quads, one for

each parameter, and then a `call` quad to call the function. The type of `sym1` must be identical to the type of the next parameter in turn.

`call`	`sym1`	-	`sym2`

Call the function in `sym1` and have the result returned in `sym2`. `sym1` must be a `FunctionInformation` symbol and the type of `sym2` must be identical to the return type of `sym1`. most recently executed and unused `param` quads. For example, if `f` is a function taking two arguments and `g` takes one argument, then the sequence `param 1`, `param 2`, `call g`, `param 3`, `call f` will cause `g` to be called with parameter 2 and `f` to be called with parameters 1 and 3.

`iassign`	`sym1`	-	`sym2`

Assign the integer value in `sym1` to `sym2`. Both variables must have type `kIntegerType`.

`rassign`	`sym1`	-	`sym2`

Assign the real value in `sym1` to `sym2`. Both variables must have type `kRealType`.

`aassign`	`sym1`	`int1`	`sym2`

Copy array elements from the array `sym1` to the array `sym2`. `int1` is number of elements to copy. The element types of `sym1` and `sym2` must be identical, and `sym2` must have room for all the copied elements.

`hcf`	-	-	-

Halt and catch fire. This quad triggers a bug in many versions of the SuperSPARC chip that causes the CPU to execute a tight microcode loop which causes the data cache, microcode controller and ALU to generate sufficient heat to cause a short-circuit between the A.119-66.4471-53 and B.122-41.3212-01 wires in the instruction decoder. If your compiler generates this quad, you have a problem.

`nop`	-	-	-

A do-nothing operation.

7 Global Variables and Constants

The skeleton uses a couple of global variables and constants. There aren't very many, and only three that are used a lot.

`currentFunction` : A pointer to the `FunctionInformation` object that represents the function currently being compiled. This object may not yet be in a symbol table.

`kIntegerType` : A pointer to the `TypeInformation` object that represents the `integer` type.

`kRealType` : A pointer to the `TypeInformation` object that represents the `real` type.

`yytext` : A pointer to a string that contains the text of the most recently scanned token. This variable is supplied by `flex`.

`yylineno` : The line number where the most recently scanned token ends. This variable is supplied by `flex`.

`errorCount` : The number of error messages issued so far.

`warningCount` : The number of warning messages issued so far.