jaac
CS 444

Andrew Kane, 96411902
Ondřej Lhoták, 96040603

April 8, 2000

Contents

Chapter 1

User Documentation

1.1 Introduction

jaac stands for Just Another Ada Compiler. It is a compiler for the Ada/CS subset of Ada. The current implementation includes the scanning, parsing, context-sensitive analysis (type checking), code generation and register statistics.

1.2 Command Line Options

The jaac sources and executable can be found on the undergrad environment in the directory `/u/olhotak/cs444/jaac`. The name of the executable is `jaac`. It takes the input Ada/CS source from standard input. It accepts the following command-line options:

`-s` Prints the output of the scanner

`-p` Prints the output of the parser (the parse tree)

`-t` Prints the output of the context-sensitive analysis (a type tree)

`-c` Prints the output of the code generator, the assembly language code for the program, suitable as input to an assembler (`as`)

`-r` Prints statistics on register allocation

`-d` Turns on additional debugging output

Any error output is sent to standard error. This includes scanning errors, parsing errors, type errors, and unsupported features.

The following section shows the output of the compiler for a very simple example. For more complicated examples (with both input and output), please see the directories under `/u/olhotak/cs444/jaac/test`. The `scan` directory contains examples of the `-s` option. The `parse` directory contains examples of the `-p` option. The `type` directory contains examples of the `-t` option. The `code`, `course`, and `fischer` directories contain examples of the `-c` option.

1.3 Hello World Example

Listing of hello.ada

```
-- basic 'hello world'

package Main is body
begin

        write("Hello World!\n");

end Main;
```

Scanning

Command: `./jaac -s < hello.ada`

Output:

```
package
id : Main
is
body
begin
id : write
lp
string : "Hello World!"
rp
semicolon
```

```
end
id : Main
semicolon
eof
```

Parsing

Command: `./jaac -p < hello.ada`

Output:

```
-+Compiln
 |-+CompilnUnit
 | \-+PkgDec
 |   |--package (L3 C7)
 |   |-+PkgSpecOrBody
 |   | |--id : Main (L3 C12)
 |   | |--is (L3 C15)
 |   | |--SpecDecS
 |   | |--PvtPartO
 |   | |-+BodyOpt
 |   | | |--body (L3 C0)
 |   | | |--BodyDecS
 |   | | \-+BeginStmtsO
 |   | |   |--begin (L4 C0)
 |   | |   |-+StmtS
 |   | |   | |-+Stmt
 |   | |   | | \-+CallStmt
 |   | |   | |   |-+Name
 |   | |   | |   | |-+Name
 |   | |   | |   | | \-+SimpName
 |   | |   | |   | |   \--id : write (L6 C13)
 |   | |   | |   | |--lp (L6 C14)
 |   | |   | |   | |-+ExprList
 |   | |   | |   | | \-+Expr
 |   | |   | |   | |   \-+Reln
 |   | |   | |   | |     \-+SimpExpr
 |   | |   | |   | |       \-+SimpExprNoUnary
 |   | |   | |   | |         \-+Term
```

```
 |   | |   | |   | |           \-+Factor
 |   | |   | |   | |             \-+Primary
 |   | |   | |   | |               \-+Literal
 |   | |   | |   | |                 \--string : "Hello World!" (L6 C28)
 |   | |   | |   | \--rp (L6 C29)
 |   | |   | |   \--semicolon (L6 C30)
 |   | |   | \--StmtS
 |   | |   \--XptnPart0
 |   | |--end (L8 C3)
 |   | \-+IdOpt
 |   |   \--id : Main (L8 C8)
 |   \--semicolon (L8 C9)
 \--CompilnUnitS
```

Type Checking

Command: `./jaac -t < hello.ada`
Output:

```
--Function call  has type Type: VOID
 |--String Literal Hello World!
```

Code Production

Command to compile to assembly:
`./jaac -c < hello.ada > hello.s`
Output (`hello.s`):

```
    .section ".data"
    .align 4
INTREADBUF: .word 0
    .section ".text"
    .align 4
PERCENTS: .asciz "%s"
PERCENTI: .asciz "%i"
TRUE: .asciz "true"
FALSE: .asciz "false"
LS0x81872c0_LITERAL:
    .asciz "Hello World!\n"
```

```
    .align 4

    .align 4
    .global main
main:
    save %sp,-104,%sp

    mov %fp, %r7 ! %r7 is reserved for global frame pointer
! PLEASE DO NOT OVERWRITE %r7

    st %fp, [%fp-4]

  ! Calling StringWriteRef
    set LS0x81872c0_LITERAL, %r9
    set PERCENTS, %r8
    call printf,0
    nop
    ret
    restore
```

Command to assemble and link hello.s:

`gcc -o hello hello.s`

Execution of hello:

Command: `./hello`

Output: `Hello World!`

Register Statistics

Note that in this case, the sample program uses no variables, parameters, or subexpression temporaries, so all the statistics are zero. If this were a more realistic program, the statistics would reflect the allocation of memory to its variables, parameters, and temporaries.

Command: `./jaac -r < hello.ada`

Output:

```
==========================================================
Register allocation statistics:
Type of value                          Number Total bytes
==========================================================
```

```
Must be on stack                           0           0
Placed in register                         0           0
Could be in %i reg but none free           0           0
Could be in any reg but none free          0           0
==========================================================
```

1.4 Source File List

`jaac.h` common declarations used everywhere

`adacs.slr` SLR(1) grammar for Ada/CS used as input to slr

`adacs.h` output from SLR containing all the information in the grammar in a form recognizable by C++, along with the parser transition table

`scan.{cc,h}` scanner

`parse.{cc,h}` LR parser with error recovery

`error.cc` code to report errors and print debugging information

`symbols.{cc,h}` base classes for symbols and productions with implementation of base class member functions – this includes the attribute grammar evaluator

`attributes.h` declaration of all the attributes in the attribute grammar, along with their type (synthetic, inherited, etc.) and default values

`terms.{cc,h}` class declarations of terminal symbol classes and their implementation

`nonterms.{cc,h}` class declarations of non-terminal symbol and production rule classes and their implementation – this includes the implementation of the attribute grammar rules

`cactus.h` cactus stack template used for the symbol table and parser error recovery

`ST.{cc,h}` symbol table

`symref.{cc,h}` declarations and implementation of symbol records representing anything appearing as an identifier or literal in the source – also includes definitions of primitive Ada/CS functions and operators

`symcode.cc` This file includes all the functions that produce assembly code.

`cstring.{cc,h}` tree-based implementation of an efficient string class which can concatenate strings in constant time, used to build up long strings such as a parse table or the output code

`FT.{cc,h}` The FrameTop class which assigns space to variables (either on the stack or in registers)

`jaac.cc` main program

`makefile` GNU makefile

1.5 Language Features

Deciding which features to include was a knapsack problem constrained by the limited time which we had to complete the project. We tried to assign a value to features based on how interesting they were to implement, how much we would learn from implementing them, and how important they would be to someone trying to use our compiler. Attempting to use an unsupported feature causes the compiler to halt with a message specifying the unsupported feature which the user was attempting to invoke.

1.5.1 Supported Features

Our compiler supports the following features of Ada/CS:

- declare blocks (scopes)
- built-in types
  - integer
  - string [1]
  - boolean

[1] Only string constants are supported; they cannot be manipulated

- all Ada/CS built-in operators
  - overloading of operators
- enumeration types
- record types
- variables
- named constants
- initialization of variables and constants within declarations
- loops
  - for [2]
  - while
  - exit and exit when statements
  - named loops with exit out of specific named loop
- subprograms
  - recursion
  - nested subprograms and potentially recursive calls between them
  - overloading and overload resolution
- Read for built-in types (except string)
- Write for all built-in types

1.5.2 Unsupported Features

We did not have sufficient resources to make our compiler support the following features of Ada/CS. We have avoided design decisions which would make it difficult to add these features.

- floating point numbers

[2] Only simple ranges of the form `Expression .. Expression` are supported

- access types
- array types
- subtypes
- aggregates
- case statements
- quoted attributes ('first, 'last, etc)
- (use) packages
- exceptions

Chapter 2

Design Documentation

2.1 Design Goals

The purpose of this project was for us to learn how to write a compiler for a full-featured language like Ada, and to demonstrate that we did in fact learn what we did. With this in mind, our priorities in making design decisions were, in general, as follows:

1. Ease of maintenance and ease of implementation – We mention ease of maintenance first, as it is the most significant contributor to ease of implementation in a large project. We put a lot of thought into making our compiler easy to maintain and to write. For example, we went to a great deal of effort to make it possible to locally modify parts of the grammar with very little effort even after the code generation phase was complete (see sections 2.3 and 2.4). Our formal automatic memoizing attribute grammar evaluator made it much easier to write all the semantic functionality – the bulk of the implementation effort – than it would have been with ad hoc methods. We wanted it to be possible to implement the compiler in the six man-months that we had available, without having to work more than we would have had to at a full-time job. At 8715 lines of (non-automatically-generated) source, and 4166 lines of distinct test programs, we were not fully successful in meeting this goal. However, we did manage to implement the most important features thoroughly enough to fulfil our main purpose as stated above.

2. Extendibility – For every design decision we made, we considered how it would affect the implementation of additional features, especially the ones that we did not end up implementing. We wanted it to be possible to implement the rest of Ada/CS given additional time without having to undo any of the decisions that we made. In a broader sense, we wanted the pieces of our compiler to be reusable for completely different languages. Our parser table generator, parser, and attribute grammar evaluator are completely general. The general design of our scanner can be used for other languages with slight modifications. Our symbol table, register and stack allocator, and value description objects can be also be reused for other languages with only slight modifications.

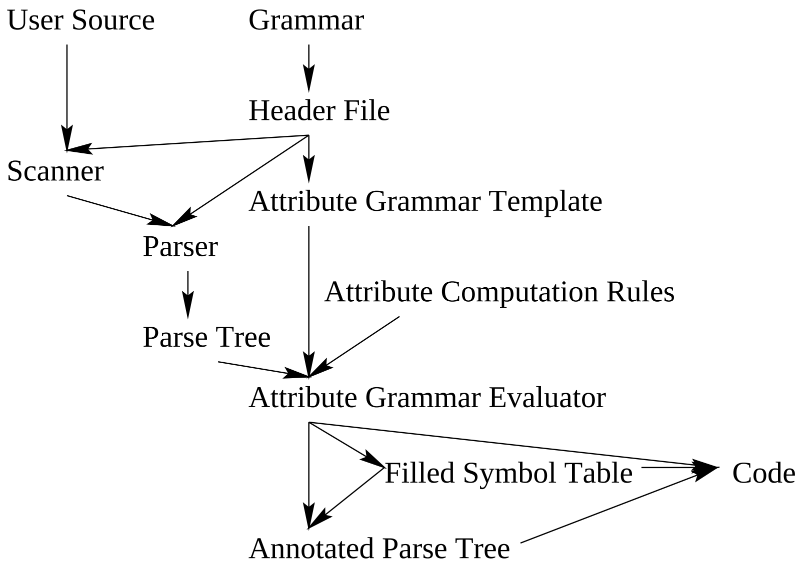
3. Educational and demonstrative value – In choosing features to implement, we chose ones which would teach us the most for the time required to implement them, and which would show what we had learned. For example, we implemented records as an example of a non-scalar type that does not fit in a register. We did not implement case statements, because they involve many tedious details, and they only require slightly more thought than if statements.

4. Usefulness of implemented features – We wanted to have a compiler complete enough to be able to compile real programs. We therefore considered essential, commonly-used features first, such as variables, subprograms, and control flow structures, before considering more unusual features.

5. Runtime efficiency of generated code – This is an issue with almost any compiler, and we felt it important to consider this issue in order to understand the construction of real compilers. Although this objective is listed last, in many cases, it did not conflict with the other objectives. For example, our static chaining implementation of nested subprograms is very efficient, yet also simple and easy to maintain. Also, in some cases, we made efficiency a priority. For example, we didn't have to use registers at all to store values (and store them all in main memory instead), but we felt that if we did this, we would be missing out on an important part of compiler design.

2.2 Structural Overview

The following diagram depicts the structure of jaac:



The creation of the compiler begins with the grammar. The SLR Generator creates an SLR parse table from the grammar, and encodes all the information contained in the grammar in a C header file. This header file is then included in the scanner and parser, and is also used to create a template for the attribute grammar computation rules. The scanner and parser create a parse tree from a user's source program. The attribute grammar evaluator applies the attribute computation rules to the parse tree, annotating it with attributes describing the semantic properties of the source program. These attributes include a symbol table, which contains symbol objects for all symbols used in the program. Finally, there is a code attribute, which uses all the semantic information from the annotated parse tree, including the symbols, and creates assembly language code, which can then be used as input to an assembler.

2.3 Grammar

The grammar is based on the Ada/CS grammar found in the file `/u/cs444/AdaCS.Grammar` on `undergrad.math` machines. The first step in adapting the grammar was to convert it to standard Backus-Naur form. The resulting grammar is not SLR(1); in fact, it is ambiguous. The next step was therefore to convert it to an equivalent unambiguous SLR(1) grammar, taking care to make it reflect the associativity and precedence rules for Ada/CS. In order to remove the ambiguities, we had to loosen the grammar, so it accepts certain strings not accepted by the original grammar. Although it is likely that an SLR(1) grammar exists which accepts the same language as the original grammar, it would be much more complicated than the original, and the structure of the parse tree created using such a grammar would not reflect the meaning of the source program. Furthermore, being context-free, the original grammar necessarily accepts a wider language than the language of all valid Ada/CS programs, because this language is not context-free. This means that the validity of an Ada/CS program cannot be fully tested with a context-free parser. A context-sensitive analysis stage is necessary. By loosening the grammar to accept a larger language, we move some of the analysis from the parser to the context-sensitive analysis stage.

2.4 SLR Table Generator

The SLR table generator is a modified version of the one given in slr.c on the course web page. The only modification is the addition of a function called `writehdr()` which writes out the information produced by the SLR table generator in the format of a C header file (`adacs.h`). This header file includes:

- Enumerated type with numeric values for each terminal and non-terminal symbol
- A class declaration for each terminal and non-terminal symbol
- Enumerated type with numeric values for each production rule in the grammar
- A class declaration for each production rule in the grammar
- A preprocessor macro containing code to fill a parser transition table
- A preprocessor macro which converts the numeric value of each symbol into a test string describing the symbol type

- A preprocessor macro which creates a new non-terminal symbol object of a given type specified by the numeric value of the type
- A preprocessor macro which creates a new production rule object of a given type specified by the numeric value of the type

Each class declaration includes two functions which return the type and string representation of the symbol or production rule. These allow us to determine the type of an object derived from a base class, giving us the same functionality as the `typecase` command in Modula-3, which is unavailable in C++. The classes for production rules also contain a function returning the type of the left-hand-side non-terminal symbol of that rule. This is used in constructing the parse tree from the bottom up.

We want to be able to add functions to the classes for specific symbols and production rules to represent attributes and perform attribute computations. To facilitate this, the header file defines preprocessor macros containing the class declarations rather than the class declarations themselves. The macros actually get expanded in other header files (currently `terms.h` and `nonterms.h`), and other member variables and member functions may be added in these header files. This design allows us to modify the member functions generated by `slr.c` for each class and regenerate the header file without affecting the additional members for specific classes introduced in `terms.h` and `nonterms.h`.

2.5 Scanner

The purpose of the scanner is to convert a file of Ada/CS source into a stream of tokens.

We decided to code the scanner directly in C++ rather than write a generic finite state machine interpreter and a finite state machine for it. The main reason for this was simplicity; since the tokens in Ada/CS have a reasonably simple structure, the finite state token recognizer is simple enough that it is less work to write it in C++ than to write a formal transition function along with a general driver. To reduce this work further, we wrote preprocessor macros and functions to easily handle the kinds of tokens that commonly appear in programming languages, such as integers, identifiers, and operators composed of two symbols of which each is a symbol on its own. Although this approach is not as general as a formal finite state machine, our functions and macros can be reused to write scanners for many common languages. A formal finite state machine would not normally permit such reuse. Furthermore, writing the scanner in C++ makes it easier to handle special language features which cannot be expressed effectively in a finite state machine, such as the '..' problem in Ada/CS (if you read 1., you don't know whether the '.' is part of the floating point literal '1.0', or part of the '..' operator, as in the range '1..5'). Because the scanner does not use a formal finite state construction, it is infeasible to recognize keywords directly while scanning. Instead, we first recognize them as identifiers, and then look them up in a table to return the correct keyword token.

The scanner keeps track of the current line and column for the purpose of reporting errors to the user. It provides detailed error messages when it is unable to group the source text into a token. These include unterminated string literals, malformed floating point literals, and unrecognized characters. When such an error occurs, the scanner returns a token containing the longest set of characters it could read before encountering a problem, and continues scanning at the following character in an attempt to recover from the error. In this way, the scanner will hopefully uncover and report all the errors rather than just the first one.

For each token recognized, the scanner creates an object of the appropriate type, and returns it to the parser.

2.6 Parser

The parser constructs a parse tree from the terminals returned to it by the scanner. It uses the LR parsing algorithm given in class, and the SLR(1) parse table produced by SLR and stored in the `adacs.h` header file. The transition table contains about 100,000 entries, so no attempt is made to compress it, since it only takes 200 kB of memory. About 4,500 of the entries are non-error entries.

Whenever the parser finds a handle, it creates an object for the production rule for that handle, as well as a non-terminal object for the left-hand side of the rule. It links these objects together with the objects for the right-hand side, so that at the end of the parse, the parser is left with a linked parse tree, which it returns.

We chose to use an LR parser because an LL(1) grammar is not expressive enough to describe certain constructs nicely, such as operator associativity.

Also, an LR parser is simple and fast, and the 200 kB memory requirement for the parser transition table is easily justified on today's machines with tens or hundreds of megabytes of memory. If memory were scarce, the transition table could have been compressed in some sort of sparse matrix representation. Out of the LR parsers, we chose SLR(1) because an easy-to-use tool for creating SLR tables was readily available, because it produced the smallest transition table, and because it was powerful enough to accept our grammar.

2.6.1 Parse Error Recovery

When a parse error occurs, the parse tables provide no way for the parser to continue. If this happens, we try to locally modify the input token stream into something that will parse without error in order to continue the parse. We do this by backing up in the input, inserting tokens, and deleting tokens. We try all possible combinations of backing up various amounts, inserting all strings of tokens up to a maximum length, and deleting up to a maximum sequence of tokens. For each modification attempted, we check how far the parser can continue before another parse error occurs. We have a minimum threshold for how far the parser must get in order for it to accept a modification. We also have a maximum number of tokens that we try to parse after a modification to judge the quality of the modification.

Of all the possible modifications, we first select those which allow the parser to parse the longest number of tokens after the modification, up to the maximum threshold. From these, we select the ones which minimize the sum of the number of tokens backed over, the number of insertions, and the number of deletions. From the remaining set of modifications, we select the ones with the minimum amount of backtracking, then with the minimum number of insertions. Finally, from the remaining modifications, we select one arbitrarily as the modification to use. The parameters of the error recovery mechanism can be set in the defines in `parse.cc`. We decided to backtrack up to 5 tokens, insert up to 3 tokens, delete up to 5 tokens, and continue parsing from between 2 and 30 additional tokens. With 74 non-terminals, this means that we search a space of almost 15 million possible strings of up to 30 tokens each. However, many of these strings cause parse errors very early. Since the proabability of a parse error at each token is about 95.5%, the expected number of strings to actually check for each error is only around 1400. These parameters ensure that recovery time is well under half a second per error even on a slow machine like undergrad.

Along with the error message, we give a line number and column, a listing of the piece of code that was modified and a listing of the modified version, with one extra token on each end for context. Our experience has been that an overwhelming majority of syntax errors which we've made in our test programs were corrected in the way in which we would have corrected them ourselves. Also, in all programs that we were able to write, the parser always found modifications which allowed it to parse to the end of the input, reporting all additional errors. There was one test program, `test/parse/parse_test_error.ada`, with a large number of deliberate parse errors for which the parser could not parse to the end of the input, but it was able to parse 110 of the 143 lines.

The most unusual feature of our error-recovery mechanism is the backtracking. With this feature, we can recover from about 90% of errors correctly, compared to about 55% without it. In order to implement backtracking, we used the cactus stack abstraction that we use in our symbol table to efficiently keep track of the parse stack after each token we parse. This allows us to easily move the parser into a previous state, while using time and space linear in the size of the input.

Example

Input:

```
package Main = body -- equal instead of is

    x : integer := 6;

    function f is begin -- no return type
        if x = 7 them -- misspelled then
            x := 5;
        end if;
return x;
    end;

begin
    x := f -- missing semicolon
end Main;
```

Output:

```
================================================================================
Error(L1 C14): Parse Error discovered here
I backtracked 0, inserted 1, and deleted 1 token(s) and I'm continuing.
I replaced
      id : Main   eq   body
with
      id : Main   is   body
================================================================================
Error(L5 C17): Parse Error discovered here
I backtracked 0, inserted 2, and deleted 0 token(s) and I'm continuing.
I replaced
      id : f   is
with
      id : f   return   id   is
================================================================================
Error(L6 C21): Parse Error discovered here
I backtracked 0, inserted 1, and deleted 1 token(s) and I'm continuing.
I replaced
      integer : 7   id : them   id : x
with
      integer : 7   then   id : x
================================================================================
Error(L14 C3): Parse Error discovered here
I backtracked 0, inserted 1, and deleted 0 token(s) and I'm continuing.
I replaced
      id : f   end
with
      id : f   semicolon   end
```

2.7 Attribute Grammar Evaluator

After the parser has produced the parse tree, we apply our attribute grammar evaluator to it to evaluate an attribute grammar which specifies the non-context sensitive analysis of the program. The attribute grammar evaluator uses recursion to automatically resolve dependencies between attributes. It automatically memoizes all results of attribute evaluation to permit efficient evaluation. It also has built-in detection of attempts to evaluate circular references in the attribute grammar. The bulk of the attribute grammar evaluator is implemented in the preprocessor macros in `symbols.h`.

2.7.1 Attributes

All attributes are declared using preprocessor macros in `attributes.h`. We redefine these macros and include this file in various places within the attribute grammar evaluator to generate the appropriate attribute code in each place where it is needed. Every attribute is declared for all non-terminals in the parse tree, so there is no need to declare attributes separately for each non-terminal that uses them. The value of the attribute need only be defined for a subset of the non-terminals, and it only gets calculated for the ones in which it is required by other calculations. Default attribute computations are provided for all attributes; they are to be overridden for certain productions in order to perform actual computation.

Inherited Attributes

By default, inherited attributes are copied from the left-hand side of every production to each non-terminal on the right-hand side.

Synthetic Attributes

The three types of synthetic attributes differ only in their default rule. A normal synthetic attribute is copied from the left-most non-terminal on the right-hand side to the left-hand side. This is useful because many productions only have one non-terminal on their left-hand side, so it makes it easy to pass things up from deep down the parse tree. A default value can be specified in the attribute declaration for productions with no non-terminals on their right-hand side. This default can be any attribute computation. For example, the `SymTblUp` attribute has the default `lhs->SymTbl()`, which makes it easy to implement "bucket brigade" symbol table passing by automatically sending the symbol table up whenever it reaches the bottom of the parse tree.

Summing Synthetic Attributes

A summing synthetic attribute has a default rule that sums the values of the attribute over all the non-terminals on the right-hand side of a production, and assigns the result to the right-hand side. It uses the C++ operator +, which must be defined on the attribute type. Again, a default value may be specified for productions with no non-terminals on the right-hand side. A summing synthetic attribute is useful for creating strings describing the parse tree, where the string for a node is usually the concatenation of the strings for its subtrees. A summing synthetic attribute would be used to construct a printable version of the parse tree, or for putting together the generated code for a program from the code generated for all the subtrees.

Max Synthetic Attributes

This type of attribute is similar to a summing synthetic attribute, but the maximum value from all the right-hand side non-terminals is returned, instead of the sum. This is used for calculating the size of a stack frame.

2.7.2 Computation Rule Definition

Code overriding the default attribute computation for an attribute named `Attribute` is placed in a member function called `CalcAttribute` of the class for the production rule in which the attribute is to be evaluated. The generated classes for the production rules contain pointers to the symbols on the left-hand side and right-hand side of the production. Attributes of any symbol may be accessed by calling the member function `Attribute()`, where `Attribute` is replaced by the name of the attribute being requested. The evaluator takes care of the memoization and circular reference detection automatically, with no additional code. The `CalcAttribute` function for a synthetic attribute is always called on a production rule to calculate the value of the attribute for the left-hand side symbol of the production. For an inherited attribute, a pointer to the symbol for which the attribute is being computed is passed into the `CalcAttribute` function so that it can tell for which right-hand side symbol it is being asked to calculate the attribute.

2.8 Attribute Computations

2.8.1 Attribute Computations For Declarations

The attributes for computing declarations each pass information up the parse tree for other attributes, and the last attribute gets put into the symbol table.

The attributes are as follows:

Synthetic:`string Id` and

Synthetic:`list<string>* IdList` These attributes allow easy access to the names of a symbol.

Synthetic:`TypeRef* TypeSubt` This attribute creates the type of a declaration. If the declaration is a type declaration, then this is put into the symbol table; otherwise, it is passed up the parse tree.

Synthetic:`ParmList* AParmList` This attribute takes each identifier in a declaration and creates a variable symbol object (ParmRef) which contains the TypeSubt attribute and the identifier. This is then passed up the parse tree.

Synthetic:`ST* SymTblUp` and

Inherited:`ST* SymTbl` These attributes pass the symbol table up and down the parse tree. The inherited attribute passes it down, the synthetic passes it up. The symbol table passes through the declaration parts of the parse tree before the code/statements section, since the declarations are used in the code.

Synthetic:`FieldTList* FieldTs` This synthetic attribute produces the list of field types from the definition of a record type. The list of fields is then inserted into the record type object for the record.

Synthetic:`FunRef* Fun` This synthetic attribute creates a subprogram symbol object which describes the subprogram defined by the subtree for which the attribute is defined. The subprogram symbol object contains information such as the name of the subprogram, the types of its parameters and return value, and the label at which its code is found.

Inherited:`FunRef* FunDown` This inherited attribute passes the subprogram symbol object computed by the `Fun` attribute down the parse tree, so that the computations which generate code for the subprogram have access to it. This is needed mainly to generate code for the return statement, which needs to know where to place the return value.

Inherited:`int NestingLevel` This attribute calculates the static nesting level of the current procedure. It is zero for the global scope, one for a subprogram within the global scope, and one higher for each level of nesting of subprograms.

Synthetic Sum:`int ContainedProcedures` and

Inherited:`bool ContainsProcedures` `ContainedProcedures` calculates the number of subprograms nested within the current subprogram. `ContainsProcedures` is an inherited attribute passed down to the implementation portion of a subprogram. It is true if and only if the current subprogram contains nested subprograms. If it does, no variables are allocated in registers; they are all allocated on the stack in case one of the nested subprograms needs to access them.

If the declaration is an object declaration, then each element in the ParmList is put into the symbol table. If the declaration is a subprogram, enumeration or record, then an appropriate symbol object is created and the ParmList is embedded in it. In the case of subprograms, the ParmList is also added to the branched symbol table for the body of that subprogram. The enumeration declaration also adds its EnumLiterals (contained in the ParmList) to the symbol table, but they are inserted into the non-branched table.

2.8.2 Attribute Computations for Expression Type Checking and Overload Resolution

These attributes perform type checking on expressions, and select the types of subexpressions which may not be immediately obvious due to function overloading. The overload resolution algorithm is the one given in class. We make a set of all the possible types of an expression, and build it up based on the possible types of subexpressions. Once we get to the top of an expression, we examine the set of possible types. If there is exactly one type which can be used, we go back down the expression tree narrowing down the set of possible types. If at any point there is not exactly one possible type for a subexpression, then that subexpression is incorrectly typed, and we report an error, listing the subexpression that caused the error. The following attributes are used:

Synthetic:`string FunName` Propagates the name of a function up to the expression involving the function.

Synthetic:`ValRef* ExprSym` This attribute calculates a final symbol object for the subexpression. The symbol object contains all the details about the value represented by that expression.

Synthetic:`TypeSet* PossibleTypes` This attribute calculates a set of type symbol objects for all the possible types of a subexpression. This is built up from the return types of any functions involved in the expression.

Synthetic:`vector<TypeSet*>* PossibleArgs` This attribute combines the PossibleTypes attributes of the parameters to a function into one unit, so that we can look for functions having these types as arguments.

Inherited:`TypeRef* RequiredType` Once the calculation of possible types reaches the top of an expression tree, a unique type is determined for the tree, and this attribute is used to calculate it. It is calculated on all the subexpressions to determine their actual types.

Synthetic:`vector<TypeRef*>* RequiredArgs` and

Inherited:`vector<TypeRef*>* RequiredArgsDown` Once a unique function has been found for a potentially overloaded function identifier, these attributes give the argument list of the function. They are used to do type-checking on the expressions which are the arguments.

2.8.3 Attributes For Code Production

Stack and register allocation

Inherited:`FT* FrameTop` and

Synthetic:`FT* FrameTopUp` These attributes pass around the FrameTop class which assigns locations for variables in registers or on the stack. See section 2.9.6 for more details.

Synthetic Max:`int FrameSize` This attribute calculates the maximum of the stack portion of the `FrameTop` attribute in the subprogram defined by the subtree. This is used to calculate the stack needs of the entire subprogram, so that the stack pointer can be adjusted above the top of the frame when the subprogram starts executing.

Code generation

Synthetic Sum:`cstring Code` This is the generated SPARC assembly code that forms the final output.

Synthetic Sum:`cstring InitCode` This is the variable initialization code. It is separate from the `Code` attribute because variable initialization code must be placed at the beginning of the body of the subprogram, after any code for other subprograms nested within it.

Synthetic Sum:`cstring IncrementCode` This is the code to increment or decrement the loop variable in a for loop. The loop statement computation for `Code` places it at the appropriate place at the end of the loop.

Synthetic Sum:`cstring LiteralPool` This contains the code for all the literals that appear in the source, each with a label. It is placed before all of the code, so that the code can refer to these literals by their labels.

Control structures

Inherited:`string BeginLabel` Computes the label at the beginning of the loop, so that the code can branch to it at the end of the loop.

Inherited:`string EndLabel` Computes the label at the end of the loop, so that the code can branch to it when it needs to exit the loop (in an exit statement or a false loop condition).

Synthetic:`Variable* LoopVar` For a for loop, computes the variable symbol object of the loop index variable.

Synthetic:`Variable* EndVar` For a for loop, computes the variable symbol object of the temporary holding the end value of the range of the loop.

Synthetic:`ValRef* StartSym` For a range expression, computes the value symbol object of the expression specifying the start of the range.

Synthetic:`ValRef* EndSym` For a range expression, computes the value symbol object of the expression specifying the end of the range.

2.9 Major Data Structures

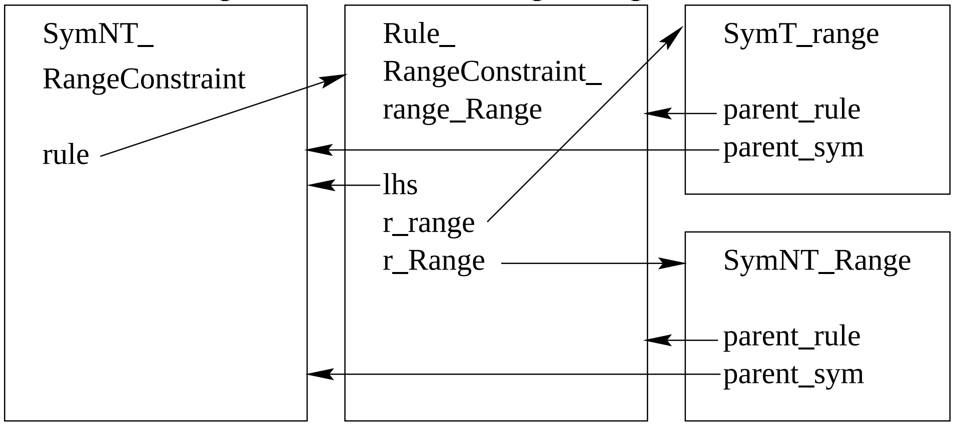
2.9.1 Parse Tree

The main data structure in the compiler is the parse tree, which is a linked collection of symbol and production rule objects. The following diagram gives the inheritance structure.

- Rule
  - Rule_Start_bof_Compiln_eof
  - Rule_Compiln_CompilnUnit_CompilnUnitS
  - ⋮
- Symbol
  - SymT (terminal)
    - SymT_integer
    - SymT_id
    - ⋮
  - SymNT (non-terminal)
    - SymNT_Expr
    - SymNT_Stmt
    - ⋮

There are two main base classes, Rule and Symbol. Rule represents a production rule in the grammar, and Symbol represents a terminal or non-terminal symbol. There is one class for each production rule that inherits from Rule. There are two classes, SymT and SymNT, which inherit from Symbol. For each terminal and non-terminal symbol, there is a class which inherits from one of these two. By convention, non-terminal symbols start with a capital letter, and terminal symbols start with a lowercase letter.

The following diagram is an example of part of a parse tree representing the production `RangeConstraint -> range Range`:



There are pointers to allow each object access to its parent and children in the tree, so that attribute calculations can be done in the attribute computation phase of the compiler. Please see section 2.7 for more information.

2.9.2 Symbol Table

The symbol table is based on a cactus stack, described later. All identifiers declared in the program are pushed onto the branch of the cactus representing the current scope. When a new scope is to be created, a new branch is added to the cactus, and a scope barrier is inserted to separate the scope from the old scope. This is necessary when checking for duplicately declared identifiers, since an identifier can be declared with the name of a pre-existing identifier only if the pre-existing identifier appears in a scope other than the most local one, (or the identifier represents a subprogram being overloaded, in which case special rules apply).

The symbol table includes functions for retrieving either the top-most symbol with a given name, or a set of all the symbols with a given name (for finding overloaded subprograms). The search is done linearly through the cactus stack, so search time is proportional to the number of symbols visible in the current scope. Though we have found no evidence to suggest that this is unacceptably slow, it could easily be speeded up by representing the symbol table as a hash table, with each bucket being a separate cactus stack.

When the symbol table is first created, we insert symbols for all of the built-in types, operators, and functions (Read and Write). That way, these are available in every scope, and may be overloaded. The built-ins can be found at the end of the file `symref.cc`.

2.9.3 Cactus Stack

The cactus stack is a C++ template for a stack which can be branched in constant time. Branching creates two stacks with the same contents as the original stack, but they can be modified (both pushed and popped) independently. It is implemented as a linked list, with the stack being identified by its top. Branching creates a new top pointing to the old stack. We use reference counts to allow us to free popped nodes only if there are no other branches coming out of them.

The cactus stack is used in the parser error recovery to allow backtracking. A new branch is made after each token is processed, effectively creating a list of the state of the parse stack after each token, making backtracking possible.

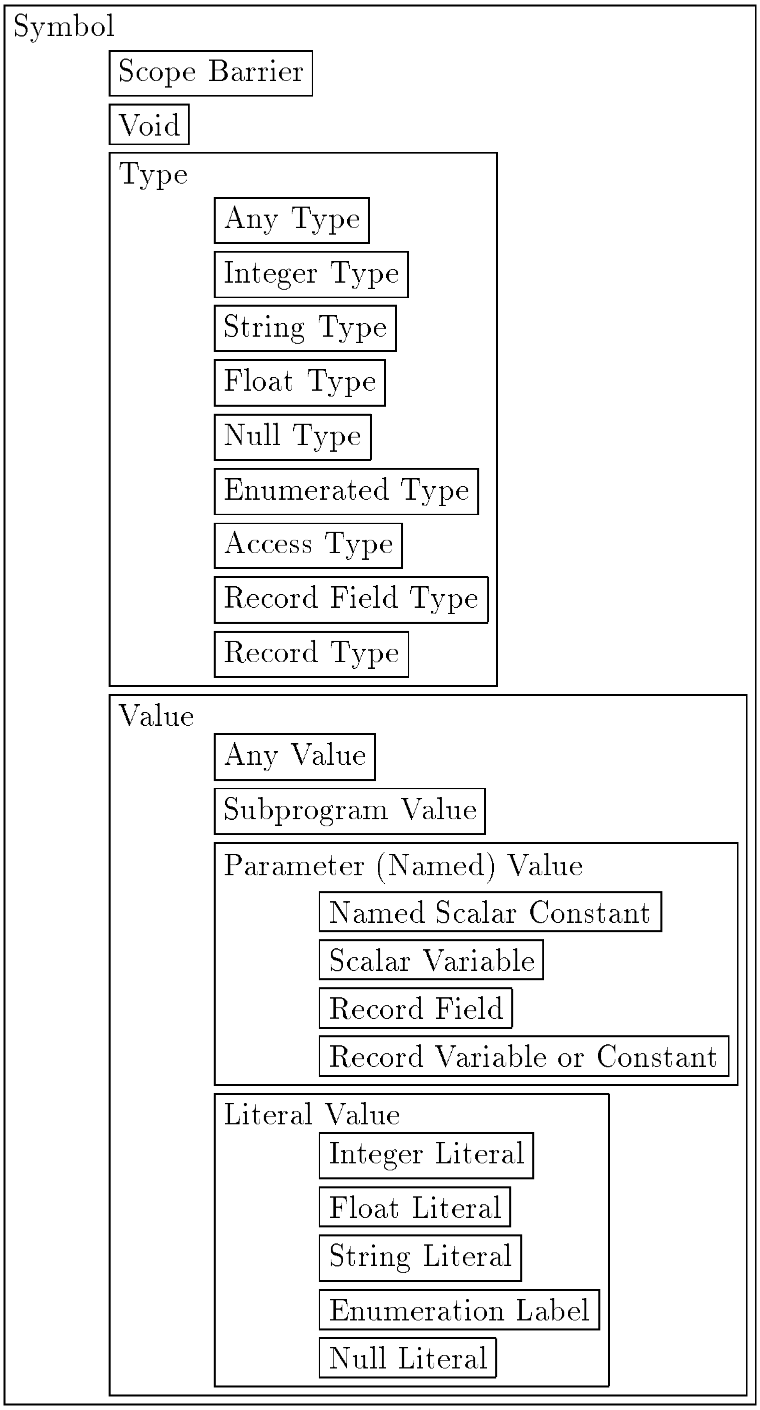
The cactus stack is also used to implement the symbol table. Here, branches in the stack represent the creation of new scopes.

2.9.4 Inheritance Hierarchy Of Symbols

A symbol object (declared in `symref.h`) is used to represent anything that appears as an identifier or a literal in the source. The figure on page 31 gives the inheritance hierarchy of these symbol objects. Symbol objects contain a name for the object, and pointers to other symbol objects. For example, every value has a pointer to its type, a subprogram contains a list of pointers to its parameters, a record contains a list of pointers to its fields, and so on. Each type is identified by a single type symbol. Since in Ada/CS, every newly-defined type gets a new cookie, and types are equivalent if and only if they have the same cookie, we simply use the address of the type symbol as the cookie, and type equivalence can be determined with just a pointer comparison. If we were implementing a language which, unlike Ada/CS, had true subtyping, we would need a more complicated type equivalence and type assignability function.

The symbol objects for values (including subprograms) play a major role in code generation. Each value symbol contains all of its compile-time information, such as where in memory (or in which register) it is stored. Value symbols also have methods such as `Fetch` and `Store` into registers (for scalar values), `Copy` to copy between two values (not necessarily scalar), `CopyIn` and

`CopyOut` to pass parameters into and out of subprograms, and `Call` on subprogram symbols to generate code for a function call. The implementation of all these methods is found in the file `symcode.cc`.


Symbol
Scope Barrier
Void
Type
Any Type
Integer Type
String Type
Float Type
Null Type
Enumerated Type
Access Type
Record Field Type
Record Type
Value
Any Value
Subprogram Value
Parameter (Named) Value
Named Scalar Constant
Scalar Variable
Record Field
Record Variable or Constant
Literal Value
Integer Literal
Float Literal
String Literal
Enumeration Label
Null Literal

2.9.5 Efficient String Class

The motivation for this class was the `Code` attribute. It has to be passed around the parse tree, with code fragments for branches being concatenated together and passed up the tree. This concatenation can be done with very large strings since the strings represent compiled output (assembler) of the compiler. The STL `string` class does not do efficient concatenation; it creates a new buffer and copies both strings into it.

The new string class (`cstring`) takes advantage of the fact that the `Code` attribute never gets manipulated after it is first created (a code fragment is only concatenated with other fragments of code). The `cstring` class creates a binary tree with the leaves having STL `string` values. With this structure, we can concatenate strings in constant time by just creating a new node with the two strings to be concatenated as its children. To output the string, we perform an in-order traversal of the binary tree, outputting the strings at the leaves.

The cstring class is now used in many of the functions for outputting information including dumping the symbol table, the parse tree, the TypeTree attribute and the TypeTreeIndent attribute.

2.9.6 Stack and Register Allocation (FrameTop) class

The FrameTop class (`FT`) is responsible for assigning space to variables. It keeps a list of available registers and the current size of the stack frame in the current block. Currently the registers are assigned on a first come first served basis (see section 2.10.4 for details), but the framework could be extended to use a more sophisticated method. This class is passed around the parse tree using the `FrameTop` and `FrameTopUp` attributes, whose propagation around the parse tree controls the lifetime of variables and temporaries.

2.10 Implementation of Language Features

This compiler produces SPARC assembly code. Comments have been added to the produced code so that it may be easily read, understood, and debugged.

2.10.1 Scalar Values

Whenever a scalar value is encountered in the source file, a value symbol object is created for it. This symbol object may be a literal symbol object, or a named value symbol object, depending on the value. The object stores all the compile-time information about the value, including its type, its size, its memory or register location (as assigned by the FrameTop stack and register allocation object), its constantness, and, in the case of a literal, its value. Scalar variables are assigned space either in a register, or, if none are available, on the stack, at a specific negative offset from the frame pointer. The synthetic attribute `FrameSize` calculates the stack needs of each subprogram, and at the start of the subprogram, the stack pointer is set to point above any values potentially used by the subprogram. This means that the amount of stack space required by a subprogram must be known at compile time, which may not be the case if the subprogram contains local variables whose size is not known at compile time. In order to support this, we would have to generate code to calculate the frame size at runtime and adjust the stack pointer accordingly. There is nothing to prevent us from adding this except lack of time.

2.10.2 Literal Pool

We do not keep a literal pool for scalar values; instead, we simply insert them into the code where they are used. Our compiler supports string literals, and could be easily extended to support other non-scalar types of literals. These are all collected using a synthetic attribute, and included together as constants at the beginning of the assembly output. The associated literal value objects contain the unique label assigned to each literal, so they can generate the code to access the literals.

2.10.3 Temporaries

Temporary variables (both scalar and aggregate types) are allocated using the FrameTop object just like any other variables, so they may be placed in registers, or on the stack. Instead of a complicated temporary allocation and freeing algorithm, we decided to simplify things by making the lifetime of a temporary be a single statement. This makes temporaries very easy to allocate: we allocate them using the FrameTop object, the same as variables.

We go back to the same FrameTop object at the start of every statement, so any temporaries allocated for preceding statements don't appear in it, and their space is free to be used.

No temporary ever needs to be around for longer than a single statement. This simplification is only slightly wasteful in the case of very complicated expressions, in which some memory locations (which could be valuable registers) could be reused for more than one temporary. However, in most programs, very complicated expressions are usually split up over multiple statements for readability. Also, on the SPARC, there are usually enough registers to accommodate most expressions found in typical programs.

2.10.4 Register Allocation

Only the `%i` and `%l` registers are used for scalar values, as these registers are automatically saved and restored on function call by the SPARC's sliding registers. Variables which are formal parameters (in, out, or both) to a subprogram must use the `%i` registers to allow the caller to copy into them directly using what it sees as `%o` registers. When the function call is made, the `%o` registers slide to become `%i` registers. The return value from a function is just like an out parameter, so it too can be placed in an `%i` register which slides back to an `%o` register when the function returns.

The global registers, `%g1-7`, are used for global values which never change, or for temporary values which need never be preserved. Registers `%g1-4` are used to do computations such as addition, subtraction, and exponentiation. Registers `%g6` and `%g7` are used to implement nested subprograms. See the section on nested subprograms below for details. Register `%g5` is never used.

For scalar variables and subexpressions, we first try to use an `%l` register to keep the `%i` registers free for formal parameters. When all the `%l` registers are in use, we try to allocate an `%i` register, and if all of those are in use as well, we put the value on the stack. This decision is made in the FrameTop class in `FT.cc`. The lifetime of a variable allocation is the duration of the scope in which it is defined. This is very simple, though not necessarily optimal, since it relies on the programmer to limit the size of the scope to the actual required lifetime of the variable. The lifetime of a subexpression is the statement in which it appears, as discussed in "Temporaries", above.

Any value which does not fit in a register (records) is always placed on the stack. We make no attempt to place specific fields of records in registers, because we keep record fields consecutive in memory in order to make it possible to implement access types and allow cyclic linked structures. If the fields of a record were not all in memory, a pointer to a record would have to be represented as a pointer to each field, and cyclic linked structures would be impossible to represent.

If a value is accessed by a subprogram nested within the subprogram in which the value is declared, the value cannot be placed in a register, because when we call the nested subprogram, the value would be slid out out with the register window, and the nested subprogram would not be able to access it. Therefore, any variable declared in a scope in which other subprograms are nested is declared on the stack, and never in a register. This is simple, but less than optimal. A less conservative and more efficient solution would be to check for each variable whether it is actually referenced by any nested subprograms, and try to place it in a register if it is not. This would not even be much more complicated to implement given the structure of our compiler, but we did not have enough time to do so.

We introduced the `-r` option to our compiler in order to evaluate our very simple register allocation strategy. An example of the output from this option is shown below:

```
==========================================================
Register allocation statistics:
Type of value                           Number Total bytes
==========================================================
Must be on stack                           291        1336
Placed in register                         469        1876
Could be in %i reg but none free             2           8
Could be in any reg but none free            8          32
==========================================================
```

This output is actually a summary of the register allocation statistics for our entire test suite. Notice that because we are compiling for a non-Intel processor, there are many registers, and almost all the values that we can place in registers actually fit there. Notice also, however, that there are 291 values that could not be placed in registers. Some of these are records, and the rest are values allocated in scopes containing nested subprograms. It may actually be possible to place some of the latter values in registers if we could ensure that the nested subprograms do not access them.

2.10.5 Records

We represent a record as the values of its fields, placed consecutively in memory. We allocate memory for the entire record. Each field has a special memory addressing object which does not actually own any memory, but contains a pointer to the record type, as well as an offset of the field value within the record. All of this information is compile-time information. If we were to implement variant records, we would have to add a new kind of addressing object which would do the offset calculation at runtime. There is nothing which would prevent us from doing this other than lack of time.

We support and test the use of records in all places where scalar values can be used, including assignment, fields nested within other records, subprogram parameters (both in and out), and function return values.

2.10.6 Subprograms

A subprogram is represented at compile time by a function symbol object. This contains information such as the subprogram's name, a label at which the subprogram's code is found, and a list of the subprogram's parameters and/or return value. The parameters and return value are themselves value symbol objects, so we can read and write to them both within the subprogram, and at the call site.

In order to allow parameter passing to subprograms, we have three types of value copy functions. The first type does a simple assignment, copying values as offsets from `%fp`. The other two copy in and out, respectively, of a subprogram, by copying to or from an offset of `%sp`. In the called subprogram, `%fp` takes on the value of `%sp` of the caller, so these copy functions allow us to effectively copy into the frame of the called subprogram before we call it, and copy back out of it after the subprogram returns. The copy functions work with special types of values: values in registers, and record types. For registers, accessing the value in the called subprogram's frame means accessing the corresponding `%o` register instead of the `%i` register. Our register allocation ensures that parameters and return values are never placed in any other types of registers. For record types, the copy functions generate code to copy all the space used by the record, one word at a time.

The procedure for a subprogram call is therefore as follows:

1. Copy in parameters into the subprogram's frame.

2. Set up the display based on the nesting levels of the caller and the called subprogram (see "Nested Subprograms", below).

3. Make a call using the label of the procedure.

4. Copy out parameters out of the subprogram's frame.

5. Copy the return value (if any) out of the subprogram's frame.

2.10.7 Scope and declare blocks

Any new scope is represented only as a new scope marker object placed in the symbol table. This is to allow us to distinguish symbols declared in the current scope from symbols declared in outer scopes, in order to detect duplicate declarations. The symbol table takes care of finding the innermost symbol with a given name when it is referenced. A new scope does not get its own frame on the stack, because there is no need for one, creating one would take time at runtime, and it would be more complicated to implement. Instead, a new scope inherits the FrameTop register and stack allocation object from its surrounding scope, so that it does not overwrite any values allocated by the surrounding scope.

2.10.8 Built-in operators

The built-in operators are implemented as special function symbol objects which have their `Call` method overridden. Instead of producing code for a procedure call, these `Call` methods perform the required computations in-line. The function symbol objects are placed in the symbol table along with all other functions, so the built-in operators can easily be overloaded for new types just by adding a normal subprogram to handle the computation required of the operator.

All values have a `Suggest` method, which suggests to the built-in procedures which registers they should use. If the value is in a register, it suggests that the built-in use that register for the computation. If the value is not in a register, a default register chosen from `%g1-4` is used. Since many values are in registers, this suggestion mechanism generates much more efficient code than if fixed registers were always used. For example, if the variable X is stored in `%l1`, for the statement `X := X + X;`, we would generate:

```
add %l1,%l1,%l1
```

instead of

```
mov %l1,%g1
mov %l1,%g2
add %g1,%g2,%g3
mov %g3,%l1
```

We should have – but did not have time to – extended the suggestion mechanism to support suggesting small literals which could be placed in the code in-line. For example, for `X := X + 1`, we should generate:

```
add %l1,1,%l1
```

instead of

```
set 1,%g2
add %l1,%g2,%l1
```

2.10.9 If-Then-ElsIf-Else

We represent boolean values the same way as any other enumerated type. We do not try to represent them as branches taken or not taken. Although this would be slightly more efficient, it would take extra work to be able to evaluate complicated boolean expressions using this representation. When we encounter an `if` statement, we first evaluate the conditional expression. We then test whether it is true or false, and if it is false, we branch past the `then` clause. At this point, there may be code for an `elsif` clause, the `else` clause, or simply the end of the `if` statement. At the end of each clause except the `else` clause, we branch out to the end of the whole `if` statement.

The formal attribute grammar evaluator makes it very easy to keep track of all the required labels, requiring only two attributes, `BeginLabel` and `EndLabel`.

2.10.10 And-Then and Or-Else

And-then and or-else differ from the standard and and or in two ways. First, they cannot be overloaded, so they are not implemented as function symbol objects in the symbol table like the other built-in operators. Second, these boolean operators use short-circuit evaluation. If the first clause evaluates to true (or) or false (and), we simply use its value as the result of the overall expression, and skip over the evaluation of the second clause. If we have to evaluate the second clause, then the value of the second clause is the value of the overall expression.

2.10.11 Loops

The overall structure of all loops is the same, for simplicity:

1. Initialization code (optional)
2. Loop start label (optional)
3. Loop condition (optional)
4. Loop body code (may contain exits)
5. Loop increment/decrement code (optional)
6. Branch to start label
7. End label

This may not be the most efficient representation in all cases, but it is general enough to be able to support all types of loops with the same structure. A simple loop does not implement any of the optional parts. A while loop only needs to implement the loop condition section, and branch to the end label if the condition is false. A for loop implements all the optional sections.

The loop start and end labels are propagated inside the loop generation code using the inherited attributes `BeginLabel` and `EndLabel`. Also, a copy of the end label is placed in the symbol table for the loop body as a loop exit symbol object. This is to allow an exit from a named loop, rather than the innermost loop.

A for loop creates a new scope in the symbol table, and the loop variable is placed in the new scope, so it may hide any existing occurrence of the same name.

2.10.12 Nested Subprograms

Our compiler provides full support for nested subprograms, with several optimizations.

We use static chaining to implement nested subprograms. In each frame, at `[%fp-4]`, we store a pointer to the frame of the static parent subprogram. We can follow these pointers to reach the frame of any static ancestor of the currently executing subprogram to access its variables. The `%g6` register is used for this calculation.

In each value symbol object, we store the static nesting level of where that symbol is declared. Also, anywhere in the parse tree where we generate code, we have an inherited attribute calculating the static nesting level of that particular piece of code. For all value accesses, we pass in the static nesting level of the code being generated, and the value object knows its own static nesting level. The value object can therefore generate code to access itself at the correct number of levels up from the level of the code being generated.

The overhead of this scheme is minimal. Values at the current nesting level are accessed with no overhead whatsoever. The overhead for accessing values at higher levels is $1 + n$ instructions, where n is the number of levels we have to go up. As an additional optimization, we set the `%g7` register to point to the frame of the outermost (global) nesting level at the beginning of the program, and we use that register to access all values directly at the global level. This avoids having to follow a potentially long static chain to access these values. The majority of variable accesses in typical programs are to either the innermost nesting level, or to the global nesting level, and both of these can be accessed directly with no overhead. The remaining accesses are often to more inner nesting levels as opposed to more outer ones, and the more inner the nesting level is, the less overhead there is.

A call to a subprogram with a higher nesting level than the current one requires a single instruction to store the current `%fp` in the frame of the called subprogram. A call to a subprogram with the same nesting level as the current subprogram requires two instructions to copy the current static link pointer from the current frame to the new frame. These are the two most common types of subprogram call. A call to a subprogram with a smaller (outer-more) nesting level than the current one requires us to follow the static links to find the static link pointer corresponding to the level of the called subprogram, and copy it into the frame of the called subprogram. This takes a few extra instructions, but these types of calls are rare. Because the static link pointer is stored on the stack in the frame of each subprogram, it is automatically restored to its previous value on return from a subprogram.

Chapter 3

Testing Documentation

3.1 File Locations

Unless otherwise indicated, test files are found in the directory :
`/u/olhotak/cs444/jaac/test/`.

The executable is found at: `/u/olhotak/cs444/jaac/jaac`. See the user documentation for more details.

The script `./runtest` in the directory `/u/olhotak/cs444/jaac/` runs a set of test files through the compiler, producing diffs of the output against expected output. `runtest` is explained in more detail at the and of this section. It takes one parameter, listed below:

scan = run scan test on files in `test/scan/`

parse = run parse test on files in `test/parse/`

type = run type test on files in `test/type/`

code, fischer, course = compile ada programs in `test/<specified>/` directory, outputting .s (SPARC assembly), assemble and link these into executables using gcc, and then run the executables

The programs in the `fischer` directory come from the web page for the course textbook, and were slightly modified to fix syntactic errors in them. The programs in the `course` directory come from the course account on undergrad. The programs in the `code` directory are ones that we wrote ourselves to demonstrate the specific features of our compiler.

`Scan`, `parse` and `type` produce `<filename>.ada.out` and `<filename>.ada.errors.out` files, which are then compared against .correct files.

`Code`, `fischer`, `course` produce `<filename>.s` files, which are then compiled to executable files. These executable files are run with any `<filename>.*.in` files piped in as standard input. The output is piped to `<filename>.*.out`, and compared to the `.correct` files. If the executable has no `<filename>.*.in` files, then it is executed with no standard input piped in. The output is similarly piped to `<filename>.out` and compared to the `.correct` files.

3.2 Scanning

The scan test files can be found in the directory `scan/`.
`<filename>.ada` = Ada files
`<filename>.ada.out` = output for `<filename>.ada`
`<filename>.ada.out.correct` = output has been checked as being correct

Scanning is quite easy to test; one runs Ada programs (or files with similar keywords and symbols) through `jaac -s`, and then checks that the output is correct for the input. There are a few tricky areas: comments, floats, ranges (the .. can easily be confused with a float decimal point), strings and the eof symbol (which is added at the end of the file). These are the areas most of out tests focus on.

Comments: The file `comments.ada` tests comments interspersed with declarations and statements, as well as comments found at the end of a file.

Floats and Ranges: The file `float.ada` tests floating point numbers and numbers with _ in them. Some of the cases are invalid floats like `300e--23` and `1.0e.1`. This file also tests ranges (`<simple expr>..<simple expr>`).

Strings: The file `string.ada` tests strings with double quotes, single quotes, unusual characters and non terminating strings (scanner cuts this to the end of the line).

End of file: All the tests produce the eof symbol.

Other: The other files test a range of Ada code, some containing invalid characters like / in `bad.ada`.

3.3 Parsing

The parse test files can be found in the directory `parse/`.
`<filename>.ada` = Ada files
`<filename>.ada.out` = output for `<filename>.ada`
`<filename>.ada.out.correct` = output has been checked as being correct

3.3.1 Parse Table

There are two levels of testing for our parser: checking if files pass through the parser without producing parse errors, and checking if the parse tree is an accurate representation of the program. All the test cases have been checked for the first level (see next paragraph), but only the Ada files with `<filename>.ada.out.correct` files have been checked for the second level.

The following is a list of some of the parsing errors from our test cases, along with a reason for each error:

`<filename>`:Error(`<location in file>`)	Reason for error
`decl04.ada.out`:Error(L40 C35):	a = b, should be a := b
`parse_test.ada.out`:Error(L23 C0):	no semicolon to end record
`parse_test_error.ada.out`:	many different types of common parse errors; good demonstration of error recovery
`test26.ada.out`:Error(L16 C58):	; instead of ,
`test38.ada.out`:Error(L3 C6):	pragmas not supported
`test39.ada.out`:Error(L12 C15):	Ada/CS does not allow object declarations in a private item

The error recovery system recovers sensibly from most of the parse errors. For example, the unsupported pragma statement is just removed and the parsing continues.

The following lists parts of the grammar and an Ada file which tests it. Not all the Ada files actually used in testing are listed here.

Part of Grammar	<filename>.ada
Package Declarations (interface and body)	decl03.ada
Declarations of Variables	test03.ada
	decl03.ada
Subprograms	test30.ada
	test28.ada
	decl03.ada
Types, subtypes	decl03.ada
Ranges	test11.ada
If Statements	test20.ada
Case Statements	test25.ada
Loops Statements	test22.ada
	test23.ada

3.4 Declarations

3.4.1 Type Declarations

It is quite hard to test type declarations by themselves, since they just create entries in the symbol table. They do use each other, but almost never in a complicated way. In fact, the top level of declarations never get tested or used within other declarations. There is a small amount of type checking here, for example, `a : bla;` checks that `bla` is actually a type that has already been declared and inserted into the symbol table before creating a value symbol object for `a` and embedding the type `bla` within it. Since declarations are hard to test by themselves, testing of declarations usually involves type checking and overload resolution (section 3.5).

Below are the different types of declarations with explanations of how and where they are tested.

Subprogram Declaration

Subprograms have a name, a parameter list (ordered), a return type and some code. They can be overloaded to have the same name, but within one scope, functions of the same name must differ in their return type or their parameter list. (See file `type/multiDef.ada`)

The return type of a function must type check with the variable you are assigning it to. The type checker can select between functions overloaded with different parameters, or with the same parameters, but different return type. Functions may not be called as procedures, with the return value thrown away (this is allowed in C, but not in languages like Modula-3 and Ada/CS). (See file `type/procOverload.ada`)

Arrays

Arrays are not yet supported, but if they are used, jaac will say so and exit cleanly. (See file `type/arrays.ada`)

Records

Record declarations associate a name with a list of field declarations. We must test both the naming and the list of fields. We also test creation of variables of type record, assignment of variables of type record to each other (if they are not the same record type, this should fail, even if they have the same structure) and referring to fields in the record. Currently, the variant part of records is not implemented, and the compiler gives an error indicating this. (See file `type/records.ada`)

Enumerations

Enumerations associate a name (which becomes a type) to a list of other names (which become constant variables). A variable given the type of the enumeration can be assigned only values from the list associated with that enumeration. (See file `type/enums.ada`)

Access Types

An access type is a pointer type. These can currently be declared, but not instantiated, because we do not currently support dynamic memory allocation. A message to that effect is printed if someone tries to create a new object of some type to assign to an access variable. (See file `type/access.ada`)

3.4.2 Object Declarations

Object declarations assign a variable name to a type. Only one variable can have the same name in each scope. (See file `type/multiDef.ada`) These are used everywhere, so almost every one of our test cases will use an object declaration.

3.5 Expression Type Checking and Overload Resolution

Overload resolution has already been discussed in the subprogram testing section above.

Expression type checking is used in most of the files referred to that test declarations. The ones used in these files are simple (usually just assignment or addition). More complicated type checking involves combining all the declarations, scope and built ins. (See all `.ada` files in the `type/` directory as well as the `/u/cs444/Test3.ada/` directory)

3.5.1 Built ins

There are many built in functions and variables. For example Boolean: (True, False) is a built in enumeration, + is built in for integers, read, write, etc. Each of these is defined for a certain set of built-in types, as defined by the language. All built-in operators can be overloaded. (See file `type/builtin.ada`)

3.6 Scope

There are three main areas where scope comes into play: packages, subprogram declarations (functions/procedures) and loops/blocks.

Packages have declarations that are visible within the code for that package. Packages can export their declarations to other packages with the `use` clause. This is not currently implemented and jaac will return an error stating this. Packages can also have private declarations which are visible only within the package. Since `use` is not implemented this cannot be tested.

The parameters of a function are put into the scope of the function's body and are only visible within that body.

The for loop defines a variable to iterate on and this variable is only visible within the scope of the loop.

See file `type/scopeOfVars.ada` for examples of all of these.

3.7 Code Production

Testing code production is done through exhaustive test input. The input is compiled, and the produced executables are checked to see if they behave as expected. Below is a list of language features with Ada files which test them. Only a few Ada files are listed for each feature; there may in fact be other test files that use the feature indirectly. Some of the tests below produce NYI (not yet implemented) errors, showing that jaac will gracefully fail for unimplemented features.

The files in the `code/` directory are tests we created, in `course/` are tests from the course account, and in `fischer/` are tests from the textbook web site.

3.7.1 Declarations

packages
- declaration before implementation
- use clause (NYI)
  - `course/test41b.ada`
  - `course/test50.ada`
- private(NYI)/public
  - `course/decl01.ada`
  - `fischer/test29.ada`

subprograms
- return values
  - `course/factorial.ada`
  - `type/function.ada`
  - `code/function2s.ada`
- parameters
  - in, out, in/out
    - `code/inout1.ada`
    - `code/display2.ada`
- no parameters
  - `code/functions3.ada`
- non-scalar parameters/return values
  - `code/records1.ada`
- declaration before implementation (forward declaration)

code/functions1.ada
code/display3.ada
code/factorial.ada

declaration w/o implementation (should give error)
code/functions1.ada

operators
code/operators.ada

operator overloading
code/overloading2.ada

overloading
code/overloading1.ada

recursion
code/factorial.ada

procedure has no return type
code/functions2.ada

object declarations (variables)

single name/multiple names (given same type)
code/decl.ada

initial value (with multiple names)
code/initialize.ada
code/readwrite.ada
code/decl.ada

access types (NYI)
course/test05.ada

constant variables
fischer/test18.ada
course/test13.ada

type declarations

enums
code/enums.ada

records
code/records1.ada

records as fields of other records
code/records1.ada

variant records
course/test46.ada

arrays
course/decl02.ada

`course/test05.ada`

incomplete type declaration (NYI)

`course/test08.ada`

subtypes (NYI)

`course/decl03.ada`

range constraint

`course/test09.ada`

exceptions (NYI)

compile time

`course/test43.ada`

runtime

`Not Implemented`

3.7.2 Statements

pragma (NYI)

`parse/test38.ada`

null statement

`code/null.ada`

assign statement

`code/factorial.ada`

assignment of non-scalars

`code/records1.ada`

call statement

`fischer/test12.ada`

declare block

`code/block.ada`

loop

for loops (forward/reverse)

`code/loops.ada`
`code/loops3.ada`

while loops

`code/loops.ada`

loop w/o while or for

`code/loops4.ada`

loop exits (w/o id, with id)

code/loops2.ada
code/loops4.ada

if statements (if, else, elsif (multiple))

code/enums.ada

exit statements

code/loops2.ada

return statement

code/enums.ada
code/factorial.ada
code/records1.ada

case statement (NYI)

course/test25.ada

raise statement (NYI)

course/test43.ada

aggregates (NYI)

short-circuit evaluation (and then; or else)

code/shortcircuit.ada

string literals

code/readwrite.ada
code/test01.ada
code/test04.ada

builtin functions

operators

code/operators.ada
code/power.ada

enum '=' and '/=' operators

code/enums.ada

read/write

code/readwrite.ada

3.7.3 Scope

hiding variables and subprograms in scope

code/display*.ada
code/block.ada

nested procedures/functions

`code/display*.ada`

scope of for loop variable

`code/loops3.ada`

3.8 Statistics

Test file statistics:

	lines	words	bytes
scan/*.ada	290	759	5797
parse/*.ada	2500	7051	40187
type/*.ada	280	804	5408
code/*.ada	978	2665	17271
course/*.ada	2156	5928	36448
fischer/*.ada	1032	4705	27560
total:	7236	21912	132671