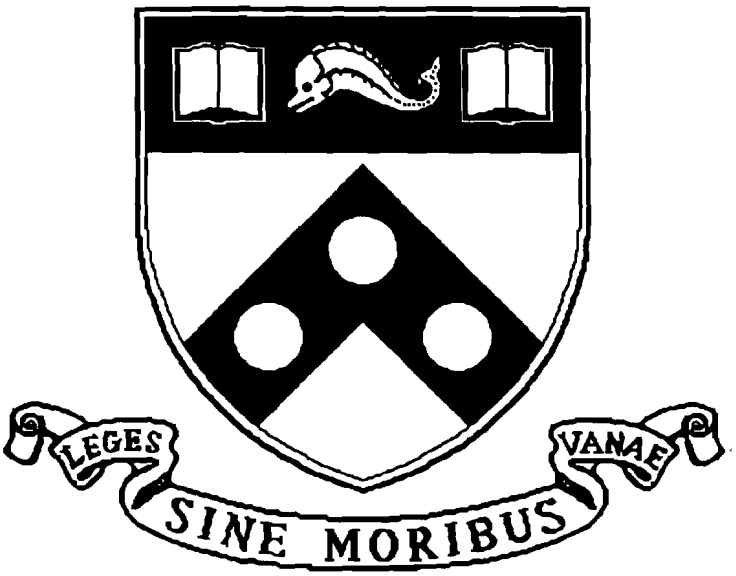
A Compiler Project for Translating a C Subset to SPARC Assembly Language

MS-CIS-93-89
GRASP LAB 364

Duncan E. Clarke
Richard P. Paul



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

November 1993

A Compiler Project for Translating a C Subset to SPARC Asssembly Language

Duncan E. Clarke and Richard P. Paul
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

November 2, 1993

Abstract

We present a complete description of a project for a compiler that translates a subset of the C programming language to SPARC assembler language. The project is suitable for a one semester undergraduate course on compilers and interpreters based on the text of Aho, Sethi, and Ullman, and has been used successfully in that context at the University of Pennsylvania. Output that facilitate scoring, and checkpoints for monitoring the students' progress are integral to the project description.

Preface

Many undergraduate computer science curricula include a one semester introduction to theoretical and implementation aspects of programming language compilers. Although a textbook, written assignments, and in-class exams are an efficient means for imparting the theory of compiler design, implementation issues are best learned by the students firsthand, by attempting their own appropriately sized implementation. This document describes one such implemention project, a compiler translating a subset of the ANSI C programming language to SPARC assembly language, used successfully at the University of Pennsylvania.

The project is intended for a one semester course for advanced undergraduates based on the text by Aho et al.[ASU86], with the text by Paul[Pau94] as a reference for material covered in past course work. The implementation as presented is based on the UNIX system Lex and Yacc tools, and the C programming language, but there is no reason why programming language curricula with a more functional flavor could not incorporate the same project using ML-Lex, ML-Yacc, and the ML programming language.

At the University of Pennsylvania distribution of the project assignment followed introductory discussions of compiler structure, detailed lectures on lexical analysis, and tutorial material on the Lex lexical analyzer generator. Students were encouraged to work in pairs, especially those deficient in either C or SPARC assembly language, although students who preferred to work individually were allowed to do so. The due dates given in the project description assume distribution of the project during the fourth week of a 15 week semester. Future courses based on this project will probably cover less introductory material so that distribution can be moved up by a week, adding more slack time to the implementation schedule, as students found the code generation phase more time consuming than originally anticipated.

Features of the source language were chosen based on their ability to create an interesting language, with a spectrum of data types and control constructs broad enough to touch on as many interesting implementation issues as possible, while keeping the students' coding task to a manageable size. Past experience has shown all of the students capable of implementing all of the features through the semantic checking phase, with the completeness of projects being determined largely by the number of features for which code can be generated. Instructors using this project are encouraged to size the code generation task appropriately, based on the progress of students up to that point in the course.

The project structure is based on the flow of data through a typical compiler. A lexical analyzer and symbol table are implemented first, followed by a parser, various checks on the semantics of correctly parsed programs, and finally code generation. The project does not include a phase for generating intermediate code or optimizing same, due to time limitations within the semester. Instead, the target language (SPARC assembly code) is generated directly from the internal tree representation of statements. Creation of the symbol table during the lexical analysis phase is perhaps earlier than some will think logical, but the even distribution of work across the semester required it, and forward definition of all variables within the source language makes it possible.

Grading of projects is facilitated by debugging directives in the language, which make it possible to display internal state and actions as compilation progresses. Test strings for grading-checkpoints based on this feature are given in Appendix A.

The project described was conceived and implemented at the University of Pennsylvania. It is however heavily influenced by past experience with the Aho et al.[ASU86] text, and borrows some features (most notably debugging directives) from an Algol/FORTRAN compiler project used in courses taught at Michigan State University in 1985 by M. Keeney, M. McCullen, and R. Whitehair. Past success with the project also owes a great deal to the bright and hard working juniors and seniors who have enrolled in CSE 341 on an elective basis.

Project Description

A Compiler for the CS Programming Language

1 Introduction

In order to give you practical experience with the theory offered by [ASU86] you will be implementing a compiler for a subset of the ANSI standard version of the C programming language[KR88]. Our C subset, hereupon referred to as `CS`, will include many of the control structures of ANSI C, the integer and floating point data types, the character data type, and at most one level of pointers to each of these types.

Input files for your `CS` compiler will have two sections. An optional external declarations section will be followed by the definition of exactly one function. The input will not include # directives (`#include`, `#line`, etc.). After lexical analysis, parsing, semantic processing, and code generation, your compiler will produce a SPARC[1] assembler source file that can be assembled using `as`.

The implementation of the compiler has been decomposed into four major phases that build upon one another. The structure of the compiler to be constructed is shown in Figure 1. The phases of the implementation correspond to construction of each of the boxes labeled "Lexical Analyzer," "Parser," "Semantic Checking," and "Code Generator." The "Symbol Table" data structure will be constructed during the "Lexical Analysis" phase and augmented during the "Parser," "Semantic Checking," and "Code Generator" phases. The "Syntax Trees" will be defined during the "Parser" phase, augmented by the "Semantic Checking" phase, and used as input for the "Code Generator" phase.

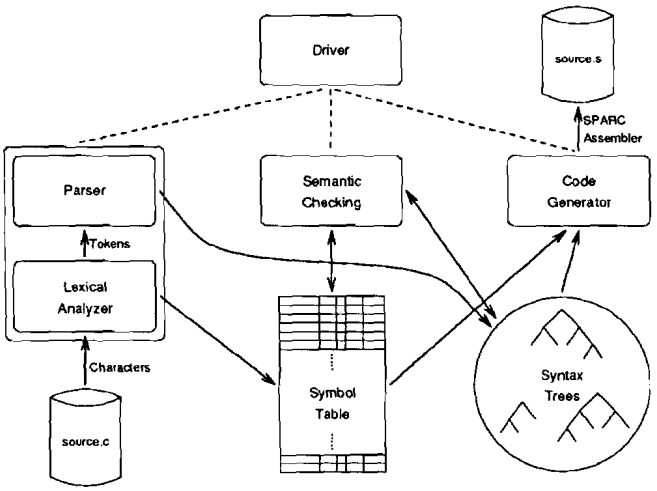


Figure 1: Project structure

Notably absent from the project are phases for error recovery during the parse and any optimization. While these are both interesting phases to implement, the time that would be required to explain the underlying theory, and design and code the implementation would make it impossible to finish the entire project in one semester. If you like, you can add these features on your own once the semester is over, since they can be added to your existing compiler in a modular fashion.

[1] SPARC is a registered trademark of SPARC International, Inc.

1.1 Goals

The goal of the project is for each team to construct a working compiler capable of generating code for a few simple programs by the end of the semester. The phrase "working compiler" is key, and you should use it as a guiding principle throughout your project. If you are concerned that implementing some obscure feature is likely to keep you from finishing, then consult the professor or the TA and see if it is reasonable to restrict your input language to eliminate pathological cases exploiting arcane features.

An assumption that you may adopt from the start is that you will only be asked to process syntactically "correct" input, meaning there will be no parsing errors.

1.2 A Few Words About Due Dates

There are always questions about all the different shades of meaning that can be inferred from due dates. So here are the policies that we're all going to abide by, in black and white.

The due dates are to be taken seriously, and you should not expect them to be extended. The pace of work that is implicit in the due dates is necessary if you're going to finish by the end of the semester, and extensions of individual assignments will only extend the final completion date beyond the semester's end. Work is to be turned in at class by the end of the week it is due. Any time after that will be considered late. Work is not to be left in mailboxes (electronic or otherwise) or slipped under doors, as it tends to get lost or trodden upon.

1.3 Organization and References

The remainder of this document is divided into four sections describing the specifications, requirements, and deliverables for each phase of the project. These descriptions and the more detailed discussion in [KR88] are to be considered your primary reference for all questions regarding syntax and semantics of the language. The course text[Pau94] will be your primary reference for the SPARC assembler code to be generated by your code generator. The article by Muchnick et al.[MAG+88] touches on a number of important points without going into much detail, but provides pointers to other more technical papers.

2 The Lexical Analyzer

You will be using the `lex` lexical analyzer generator tool (or the GNU version, `flex` if you prefer) described in lecture, on-line in the `lex(1)` man' page, briefly in [KP84], and in detail in [LMB92]. As you have already learned, the primary purpose of the lexical analyzer is to divide the input stream of characters into aggregate groups, called tokens, in order to simplify the operation of the parser. The construction of your lexical analyzer will also require you to define and construct a symbol table and possibly several auxiliary literal tables to record the values of identifiers and literals that are passed to the parser.

2.1 The Symbol Table

Your symbol table should provide all of the classic dictionary services, including efficient lookup and insertion. For the most part symbol table organization is left at your discretion, including such issues as data structure, representation of literals, and whether literals are stored in the main symbol table or separate literal tables. Care should be taken in the choice of your symbol table data structure as some structures are too inefficient to be used (e.g. a single linear list) and some are more complex than their efficiency merits for a project of this size (e.g. B-trees, red-black trees, etc.).

Since you will be augmenting the symbol table entries throughout the project, document their structure carefully and always keep the need for future expansion in mind. The symbol table is the single most important data structure in your compiler, so give it the time and thought it deserves.

2.2 Lexical Entities

There are ten classes of lexical entities, eight of which generate tokens to be returned to the parser. The remaining two classes (comments and white space) are to enhance the readability of the program and may be "skipped," meaning that they will not be returned as tokens.

Comments Comments are delimited by `/*` and `*/`, and they are skipped except for information on debugging flags. Flags will be used to control debugging aids and diagnostic information throughout the project. The flag field immediately follows the opening delimiter and may not be preceded by, or include white space. The format of the flag field is:

`/*debug(`*debug_directive*`)` *...remainder of comment...* `*/`

where, for the lexical analysis phase of the project, *debug_directive* is one of:

- `token_listing_on`—Begin listing tokens to the standard output as they are created. The output format is: <*token_code*,*token_value*>. For example, if the token code for the keyword `while` is 289, then immediately before returning this token to the parser you write `<289,while>` to the standard output.
- `token_listing_off`—Stop listing tokens until directed to do so again.
- `symtab_stats`—Report information on symbol table usage and performance. Values that might be of interest include the number of symbols inserted in the table, the number of unique identifiers, the number of each type of constant, the number of look-ups, the average number of items searched on each look-up, etc. This is by no means an exhaustive list, and you should report any and all values that are appropriate for your symbol table organization.
- `symtab_dump`—Display the contents of the symbol table. Your output listing should include each symbol's value, its reference count, and any attributes.
- `halt`–Halt execution of the compiler without processing any further input.

White Space Spaces, tabs, newlines, and formfeeds are used throughout the input text to delimit tokens and enhance readability. White space outside the context of character or string constants should be skipped.

Identifiers An identifier is any string of characters made up of `a-z`,`A-Z`,`0-9`, and underscore (_). The first character of an identifier may not be a digit or underscore, there is no bound on length, and all characters are significant.

Reserved Words Table 1 contains the reserved words, or keywords, of our language. Each one should be assigned a unique token code, and you may differentiate them from ordinary identifiers by whatever means you like.

`char`	`do`	`float`	`else`
`extern`	`for`	`if`	`int`
`return`	`void`	`while`	`register`

Table 1: Reserved words

Special Characters Table 2 contains the special characters, and atomic sequences of special characters that your lexical analyzer should recognize.

Integer Constants Your lexical analyzer should recognize the three types of integer constants defined in section A2.5.1 of [KR88]. You needn't handle the prefixes for unsigned and long types however, since our language will not use these representations.

;	,	:	&	=	&&	||
+	-	*	/	%	++	--
<	>	==	!=	<=	>=	!
(	)	{	}	[	]	

Table 2: Special characters

Floating Point Constants Floating point constants should be recognized in keeping with the definition of section A2.5.3 of [KR88]. Again, there is no need to handle the suffixes for "float" and "long" types since our language will compute all floating point quantities in single precision.

Character Constants You are responsible for recognizing character constants as defined in section A2.5.2 of [KR88], including escape sequences, with the exception of the \x*hh* format. The choice of internal representation for character constants is at your discretion. We will be treating characters as signed integers when we reach the code generation phase.

String Constants A sequence of ASCII characters delimited by double quotes ("...") represents a string constant, which is in truth just an array of type `char` padded with a `NULL` character. Double quotes may appear in strings if they are preceded by \, as may all of the escape sequences defined for character constants above. A newline may not appear in a string unless it is preceded by \, in which case it is ignored.

Illegal Characters Any character not included in the token classes described above ("?" for example) is illegal as part of the input. You should assign a special code to this class of characters and return that code to the parser when an illegal character is encountered.

2.3 Food for Thought

- The debugging flags serve a dual purpose. They are intended both to make it easier to produce output that demonstrates the operation of your project for the purpose of grading, and to provide you with invaluable data during the debugging of each project phase. Implement them *first*, not last.

- Even without the embedded debugging flags comments are difficult to describe to `lex`. Consequently, comments are typically recognized by their opening sequence (`/*`), and then a C function is called to search for the closing "`*/`." The `lex(1)` man' page has a good example of this.

- Two values are returned to represent each token—the return value of the `lex(void)` function, and auxiliary data returned in the `yylval` variable. The `yylval` variable is typically defined as a `union` and upon return from the lexical analyzer usually contains a pointer to an appropriate symbol table or literal table entry.

  Once the construction of the parser has begun, you will `#include` an output file produced by `yacc` to define symbols that correspond to token codes. To save you the work of building your own file full of `#define`'s for token codes, you can start building your `yacc` input file now. The format of the file is shown in figure 2. If you process the file using the command

  `yacc -d` *token_file*`.y`

  the files `y.tab.c` and `y.tab.h` will be produced. The `y.tab.c` file can be ignored for now, and `y.tab.h` should be `#include`'ed at the top of your lexical analyzer description file.

- For the sake of readability in your parser description, don't define token names for the single-character tokens like ";". just use their character value for the token code. There won't be a conflict with the `lex` assigned token codes because the codes assigned by `lex` are all outside the ASCII character range.

```
%{
#include "your_globals.h"
%}

/* One %token line for every token
 * you define.  Here is an example:
 */

%token IDENT

%%

/* One dummy rule, so that yacc thinks
 * this is really a parser definition.
 * There is a tab before : and ;.
 */

dummy    : IDENT
         ;

%%
```

Figure 2: Dummy **yacc** input file

- When looking at **lex.yy.c** for compile-time errors, be sure to make the corrections to the input file, not the **lex.yy.c** file. A good way to make sure you don't slip up is to have **make** turn off write permission on the file as soon as it is created.
- If you keep your strings and identifiers in the same table be sure to differentiate between the identifier **index** and the string constant **"index"**. Similar problems arise if you maintain your character or numeric constants as strings.
- The **yywrap()** routine is defined in **/usr/lib/libl.a**. You can link it with your lexical analyzer by including the **-ll** option at the end of the **cc** command that builds your final executable.
- For small changes, using **make** can save you as much as five minutes per compile by the end of the project. Why not start using it now?

2.4 Deliverables and Due Dates

Week 5 Submit a detailed description of the symbol table organization you intend to use. Include a discussion of why you have chosen this organization over others, where you intend to store the various classes of symbols (identifiers, string literals, etc.) and C declarations for any structures or arrays.

Submit a list of regular expressions that you intend to use to represent each class of token. The regular expressions can be given in the notation of §3.3 of [ASU86], and need not be in a form acceptable to **lex**.

Week 7 Construct a **main()** function that accepts the input file name as a single parameter, and calls **lex** repeatedly to break the input up into tokens. Run your lexical analyzer, driven by this main routine, on a test file to be made available shortly before the due date.

You are to submit the source for your lexical analyzer description file and any support routines along with the output from the test run. If you made any simplifying assumptions in constructing your lexical analyzer, describe them in a written summary. If your lexical analyzer failed to scan the test file correctly, describe the nature of your problems in a written summary. Make any notations on your test run output necessary to explain the format of your trace and dump output.

3 The Parser

You will use the **yacc** parser generator tool (or the GNU version, **bison** if you prefer) described in lecture, on-line in the **yacc(1)** man' page, briefly in [KP84], and in detail in [LMB92]. In addition to designing the **yacc** input specification for the language grammar, you will also be augmenting your symbol table to retain attributes of identifiers and constants defined by the program being parsed, and constructing an intermediate representation of the parsed statements in the form of binary trees.

3.1 Project Grammar

The grammar defining the syntax of **CS** is shown in Figure 3. In the grammar specification, typewriter font is used to represent tokens and the normal Roman font is used for non-terminal symbols. The **ident**, **icon**, **fcon**, **ccon** and **scon** tokens represent identifiers, integer constants, floating point constants, character constants, and string constants respectively, as defined in section 2.2.

The grammar as presented is not LALR(1), and consequently you will have to make several adjustments before **yacc** will generate a useful parser. It also does not correctly represent operator precedence in all cases, and steps will have to be taken to describe the precedence in terms that make the language described by your parser consistent with the operator precedences defined in §A7 of [KR88].

You are free to alter the productions of the grammar in any way you like, provided your grammar accepts the same syntax as the grammar of Figure 3, and implements the appropriate C semantics.

3.2 Semantic Actions

Once your grammar specification is complete, you will augment most of your productions with semantic actions. The purpose of these actions is to record information related to identifier attributes that are specified by declarations, and to construct an internal representation of the source statements using binary trees. The identifier attributes to be recorded in the symbol table will require you to add fields to your existing symbol table data structures.

3.3 Debug Directives

In addition to augmenting your **symtab_dump** code to output any fields you add to your symbol table entries, you should also implement the following new directives:

- **variable_dump**—List variable names, function names, and array names, and their scopes, types, dimensions, and parameter types, etc., as appropriate.
- **statement_dump**—Print each syntax tree for all statements fully parsed at the point this directive is encountered. The output needn't be pretty, just complete enough that you can verify the internal representation.

3.4 Food for Thought

- Be sure to record every detail of each declaration in your symbol table, and every detail of each statement's syntax in your trees, because once the parse is complete you won't be able to refer to the input file for additional data.

program	→	external_decls
external_decls	→	declaration external_decls | function_def
declaration	→	modifier type_name var_list ;
modifier	→	**extern** | **register** | ϵ
type_name	→	**void** | **int** | **float** | **char**
var_list	→	var_list , var_item | var_item
var_item	→	array_var | scalar_var | * scalar_var
array_var	→	**ident** [**icon**]
scalar_var	→	**ident** | **ident** (parm_type_list)
function_def	→	function_hdr { function_body }
function_hdr	→	type_name **ident** (parm_type_list)
	|	type_name * **ident** (parm_type_list)
	|	**ident** (parm_type_list)
parm_type_list	→	**void** | parm_list
parm_list	→	parm_list , parm_decl | parm_decl
parm_decl	→	type_name **ident** | type_name * **ident**
function_body	→	internal_decls statement_list
internal_decls	→	declaration internal_decls | ϵ
statement_list	→	statement statement_list | ϵ
statement	→	compound_stmt | null_stmt | expression_stmt | if_stmt
	|	for_stmt | while_stmt | dowhile_stmt | return_stmt
compound_stmt	→	{ statement_list }
null_stmt	→	;
expression_stmt	→	expression ;
if_stmt	→	**if** (expression) statement
	|	**if** (expression) statement **else** statement
for_stmt	→	**for** (expression ; expression ; expression) statement
while_stmt	→	**while** (expression) statement
dowhile_stmt	→	**do** statement **while** (expression) ;
return_stmt	→	**return** expression ; | **return** ;
expression	→	assignment_expr
assignment_expr	→	unary_expr = expression | binary_expr
binary_expr	→	binary_expr binary_op unary_expr | unary_expr
binary_op	→	boolean_op | rel_op | arith_op
boolean_op	→	&& | ||
rel_op	→	== | != | < | > | <= | >=
arith_op	→	+ | - | * | / | %
unary_expr	→	unary_op unary_expr | postfix_expr
unary_op	→	! | + | - | ++ | -- | & | *
postfix_expr	→	postfix_expr [expression]
	|	**ident** (argument_list)
	|	**ident** ( )
	|	postfix_expr ++
	|	postfix_expr --
	|	primary_expr
primary_expr	→	**ident** | constant | (expression)
constant	→	**icon** | **fcon** | **ccon** | **scon**
argument_list	→	argument_list , expression | expression

Figure 3: BNF description of project grammar

- Each unique node type you create in your intermediate representation is going to require unique code during the code generation phase. The fewer node types you have, the fewer different code templates you will have to generate. One way to reduce the number of unique nodes is to identify operators or statements that can be redefined in terms of other, more basic operators or statements. Section A9.5 of [KR88] offers a hint for one possible reduction. There are others.
- Beware of redefining `++` or `--` in terms of assignment. If you must, consider an expression like `(*p++)++`, where p is a pointer to an integer, before attempting an implementation.
- If you corrupt `yacc`'s private data structures at run time, it will occasionally be reported as a syntax error.
- Scopes of identifiers turn out to be a little more difficult to manage than you might expect. Consider them carefully before you start coding, and make sure you have a plan to (1) identify the point in the parse when scopes change, and (2) accommodate name clashes between internal and external declarations. Also plan how you will handle the names of parameters that appear in function declarations, and what scope they should have.

3.5 Deliverables and Due Dates

Week 8 Submit a copy of the final LALR(1) grammar you intend to use as input to `yacc`. Describe in writing all changes you made, including how you eliminated the conflicts in the original grammar, and how you enforced the appropriate precedences. If any conflicts remain, explain why `yacc`'s default action for resolving conflicts is appropriate.

Week 10 Submit a detailed description of the data structure you intend to use for the internal representation of statements. Your description should include a copy of the C declarations of your node and leaf types and any supporting structures. You should also include a complete description of how your trees are used to represent each expression and statement type. For example, Figure 4 shows the type of diagram you might use to represent `if/else` statements, and `%` expressions.

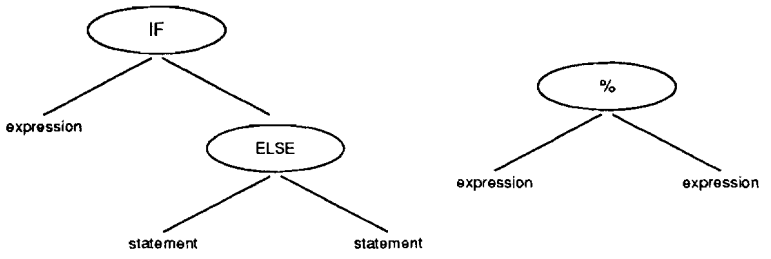


Figure 4: Sample syntax trees

Week 11 Construct a `main()` function that accepts the input file name as a single parameter, and calls `yacc` to parse the input. Run your parser, driven by this main routine, on a test file to be made available shortly before the due date.

You are to submit the source for your parser description file and any supporting routines along with the output from the test run. If you made any simplifying assumptions in constructing your parser, describe them in a written summary. If your parser failed to interpret the test file correctly, describe the nature of your problems in a written summary. Make any notations on your test run output necessary to explain the format of your trace and dump output.

4 Semantics and Data Type Checking

In this phase of your project you will write functions to analyze your syntax trees. You will report any semantic errors to the user with appropriate messages written to the standard output, and you will augment the syntax trees with any type conversions necessary to make the expressions and statements consistent with respect to data types.

The semantic rules that you are expected to enforce are as follow:

1. `CS` is strongly typed. All variables, arrays, and functions must be appropriately defined.
2. Names may not be overloaded. That is, if there is an integer variable named "f", there cannot be another variable, array, or function of the same scope also named "f".
3. Arguments specified in function calls must agree in number and type with the function's prototype, either directly or through conversion.
4. The value of any expression returned by a function must agree in type with the declared type of the function, either directly or through conversion.
5. The types of arguments and results for unary and binary arithmetic operators are defined in §A6.5 of [KR88].
6. You may ignore the special case for conversion of `NULL` pointer values described in §A6.6 of [KR88].
7. The `&` and unary `*` operators should have their usual meaning, except that `CS` does not allow the `&` operator to be applied to an expression that is already a pointer type.
8. The left hand side of all assignment statements should be checked to verify that any expressions are in fact lvalue's, as discussed in §A5 of [KR88].
9. Arrays will be represented as pointers to appropriately sized blocks of storage. Therefore array names should be treated as pointers to the appropriate type.

In order to prepare for the code generation phase, you should also allocate local, temporary variables for each intermediate node in the trees representing expressions. Allocate the temporaries in the symbol table as you would ordinary automatic variables, assigning names such that they do not conflict with existing variables. Assign the temporaries the appropriate data type and any other attributes you deem necessary.

4.1 Debug Directives

In addition to augmenting your `symtab_dump` and `variable_dump` code to output any fields you add to your symbol table entries, and verifying that your `statement_dump` code is capable of outputting any type coercion nodes you add, you should also implement the following new directive:

- **`trace_types`**—Trace the type checking of expressions and statements. For example, when type-checking a `+` operator with integer and floating point operands, you might output the following:

  ```
  Types:  + int float -> float
  ```

  When type-checking an integer function named "f" that accepts two float arguments that were provided as an integer and char respectively, you might output the following:

  ```
  Types:  f(int,char) -> (float,float)
  ```

4.2 Food for Thought

- Before you go to the trouble of converting float's to int's for use in the boolean operands of `if`, `&&`, etc., take a look at some code generated by `gcc -S`. You may be surprised to learn that the semantics described in §A9.4 of [KR88] (and elsewhere) are meant to be taken quite literally.
- Don't forget that adding integers to pointers requires scaling the integer operand by the size of items of the data type referred to by the pointer operand.

4.3 Deliverables and Due Dates

Week 12 Submit tables describing type conversions and errors resulting from every combination of operator and operands. For each operator provide a matrix showing what conversions are performed and what the resulting type is for every possible pair of operands. See Figure 5 for an example.

+	void	int	float	char	pointer
void	error	error	error	error	error
int	error	int	int → float float	char → int int	int → pointer pointer
float	error	int → float float	float	char → float float	error
char	error	char → int int	char → float float	char	char → pointer pointer
pointer	error	int → pointer pointer	error	char → pointer pointer	pointer

Figure 5: Type matrix for binary + operator

Week 13 Run your parser and semantic checker on several test files to be made available shortly before the due date.

You are to submit annotated output for each of the test runs. If you made any simplifying assumptions in your semantic checking, describe them in a written summary. If your semantic checker failed to check the test file correctly, or failed to perform any appropriate type conversions, describe the nature of your problems in a written summary. Make any notations on your test run output necessary to explain the format of your trace and dump output.

5 The Code Generator

The final phase of your compiler is the code generator, which will convert the abstract syntax trees constructed during the parse into SPARC assembler code. We will not be generating an intermediate representation such as three address code, since it would involve an extra translation step that is unnecessary given that we won't be performing any optimizations.

There are several different issues to be addressed during code generation, including allocation of storage for automatic, temporary, and global variables; establishment of the appropriate linkage from the callee of the compiled function, and to any called functions; efficient register allocation; allocation of constants and literals; and, finally, generation of executable code to correspond to the control structures and expressions in the input program.

The key to simplifying the generation of actual executable code is carefully designing and implementing your register allocator. If you define a concise, minimal interface that works correctly, the most difficult part of generating code will be all the typing involved. Take the time to plan your register allocator carefully and the entire code generation phase will be the least challenging of the project. If your register allocator is constantly requiring attention, code generation can turn out to be the most time consuming part of the project.

5.1 Debug Directives

In addition to insuring that your previous debugging flags can produce meaningful output for any new fields added to the symbol table and syntax trees, provide the following new debugging flags:

- **`trace_regs`**—List register allocation requests, deallocation requests, purge actions, and spill actions as they occur. Identify by name the temporary, variable, or constant that the allocation or deallocation involved.
- **`register_dump`**—Print the contents of the register allocation table at termination.

5.2 Food for Thought

- Design, code, and test your register allocator first. Once it is working, then start working on generating assembler output. Start small, generating the prologue, return instruction, and epilogue for a program consisting entirely of:

  ```
  void null(void) { return; }
  ```

  Once you've got the skeleton of your output programs done, start working on generating code to compute expressions by adding an expression to your return statement. Once you can compute simple constant expressions, start passing variables in. And so on, continually growing the complexity of the inputs your program can process. This way errors are easily diagnosed and tracked down.

- Very few people are going to be able to implement code generation for all the features of the entire language, so start out by setting realistic goals for yourself. Limit the incoming and outgoing parameter count to six or fewer so that all parameters can be passed in registers. Select a subset of the control structures to implement first, and implement the rest once your original goal is achieved. Select a subset of the operators to implement first, and implement the rest once your original goal is achieved. If you try to implement the entire language all at once, you'll probably end up with a compiler that can't compile anything. If you grow the set of programs you can compile, from the very small to the very complex, you're likely to be able to compile a significant subset of `CS` by the time you're finished.
- Use the `-S` option of the C compiler to see how others have attacked these problems before.
- Generating nested function calls on an architecture that passes parameters in fixed registers can be tricky. If you attempt to tackle this problem, remember that some innocent looking expressions generate implicit function calls.

5.3 Deliverables and Due Dates

Week 14 You will be provided with source for a short `CS` program called `null.c`. This source file will include variable declarations that will use `extern` storage from another program, allocate its own global variables, and allocate parameters and automatic variables on the stack. The function `null()` will consist of exactly one expressionless `return` statement.

You are to run your compiler with `null.c` as the input, and produce an error-free SPARC assembler output file for the program. Submit a printed copy of the results of your compilation. If your code generator fails to generate correct output, provide a written explanation of any problems.

Week 15 Code, compile, and run two non-trivial algorithms of your choice. Good candidates would be (1) use Newton's method to find roots of a polynomial, (2) Quick Sort, (3) Bubble Sort, etc. You may use separately compiled C programs to drive your `CS` programs and produce printed results if you wish.

Code, compile, and run a third test program that demonstrates all the control structures and expressions your compiler is capable of generating correct code for. The program needn't produce any useful results.

Submit a printed report containing all of the following:

- `CS` source, assembler output, and an execution script from your three test programs.
- A summary describing what works, and what doesn't in your compiler. For those things that don't work, explain why each particular feature in question was omitted, or why it doesn't work.
- A source listing for the entire compiler.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley, 1986.

[KP84] B. W. Kernighan and R. Pike. *The UNIX Programming Environment.* Prentice Hall, 1984.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[LMB92] J. Levine, T. Mason, and D. Brown. *Lex and Yacc.* O'Relly and Associates, Inc., 1992.

[MAG+88] S. S. Muchnick, C. Aoki, V. Ghodssi, M. Helft, M. Lee, R. Tuck, D. Weaver, and A. Wu. Optimizing compilers for the sparc architecture. In *Proceedings of COMPCON '88*, March 1988.

[Pau94] R. P. Paul. *SPARC Architecture, Assembly Language Programming, and C.* Prentice Hall, 1994.

A Test Strings

A.1 Lexical Analyzer (Week 7)

```
/* Lexical Analyzer Test Stream */

/* Before you run the test, sit down and work through it with paper and
 * pencil marking off all of the lexemes and errors.  If you wait and just
 * glance through your output you'll be fooled into believing that a lot
 * of things that are really wrong are correct.  Pay attention to the running
 * commentary--it provides clues regarding what should be happening.
 *
 * There are a lot of tests here.  If it doesn't all work on the first try,
 * choose which cases to fix carefully.  Attend to the common cases first, and
 * leave the truly pathological for last.  For example, correct output from
 * section 15 is much more important than handling escaped newlines in your
 * character strings, or octal integers containing digits 8 or 9.
 *
 * Your debug directives must work correctly.  Your score on the project will
 * be based in large part on the output they generate.
 *
 * Good luck.
 */

/* 1. A single-line comment */
/* 2. A multi-line comment, beginning here
   and
       ending here */

/* 3. Check instrumentation */
/*debug(token_listing_on) */
identifier
/*debug(symtab_dump) */
"If you see this, things can't be all bad." float + ++ 1066 3.141592653598 'c' ?
/*debug(token_listing_off) */

   4.0 these should pass by without being echoed but they had better
find their way into the symbol table once and only once

/*debug(symtab_dump) */

/* 5. identifiers */

/*debug(token_listing_on) */

i j k simple sim_ple sim__ple sim_ple_ _illegal
simp1e s1mple 3just_checking
an_insane1y_10ng_identifier_that_had_better_n0t_10se_any_characters
an_insane1y_10ng_identifier_that_had_better_n0t_10se_any_characterz

/* 6. delimiters */
```

```
6.1 one_token  6.2 two tokens
6.3 two tokens  6.4 two
tokens
6.5 three;tokens  6.6 three!=tokens
6.7 two/**/tokens

 /* 7. Reserved words */

    char do float else extern for if int return void while register
/* *not* reserved words: */ goto Return integer registe fort

/* 8. Special characters */

; , : & = && || + - * / % ++ -- < > == != <= >= ! ( ) { } [ ]
** &&& ++++ ----- =< => !== :-)

/* 9. Integers */

/* decimal */ 0 1 2 3 9 13 840 908566 119918 21557922523136343405
/* octal */ 00 05 07 0711 0177777
/* hex    */ 0x0 0x1066 0xface 0xcafe 0xfeed 0xffffffff
/* trickiness */ 0x 0908566 0xbeadgcf

/* 10. Floats */

0.0 1.0 3.141592653598 1.544e+6 1.2 .2 1.
6e23 6.023e23 6.023e+23 1.0e-9 1e-9 6.023E+23
034.27

/* 11. Characters */

'i' 'I' '<' '1' ' ' ' '\'' '\n' '"' '?''#'
/* errors: */ ''' '!=' ' == '\p'

/* 12. Strings */

"Make sure the following line generates *TWO* strings:"
"a simple string" "a simple string "
"<" "1" "i" "warned you..."
"\\tab\tover\n" "" "\\" "\\\\" "\\\"\n"
"str/* what do you suppose this does? */ing"
"a string with an embedded newline \
 how did you do?"
"a string with an erroneous newline
 how did you do?"
"a string with an embedded \"quote\""
"IBM would call this: "" an embedded quote.  We don't."

/* 13. Illegal */
```

```
@#$^_~`.\"OK string"

/* 14. whatzit? */

/*/ */ */

/* 15. Enough of the pathological; now for the mundane: */

extern int *keys;

int printf(char *s1,int i1);

int bubble_sort(int key_count)
  { /*debug(token_listing_off) Enough already! */

    int i,j,swaps;
    register int temp;

    swaps = 0;

    for(i=0;i<key_count-1;i++)
      for(j=i+1;j<key_count;j++)
if (keys[i] > keys[j])
  { temp = keys[i];
    keys[i] = keys[j];
    keys[j] = temp;

    swaps++;
  }

    printf("Sort complete, %d swaps\n",swaps);

    return swaps;
  }

/* 16. finis */

/*debug(symtab_dump)*//*debug(symtab_stats)*//*debug(halt)*/

/*debug(token_listing_on) */ "You've gone too far!"
```

A.2 Parser (Week 11)

```
/* Hacked up quick sort, from 2nd ed. K&R, page 87. */

float left; /* Should be hidden by parameter left       */
char i; /* Should be hidden by internal declaration */
int ext,ext2; /* Should not be hidden by anything         */
void dummy(void);
void swap(int *v, int i, int j);
```

```
void qsort(int *v,int left,int right)
  {
    register int i,last;
    int new_partition_element;
    int *ext_ptr;
    int everything_a_ok, bad_news;
    float array[64];

    if (left >= right)
      return;

/*debug(token_listing_on) */
    new_partition_element = (left+right)/2;
/*debug(token_listing_off) */
    swap(v,left,new_partition_element);
    last = left;
    for (i=left+1;i<=right;i++)
      if (v[i] < v[left])
swap(v,++last,i);
    swap(v,left,last);
    qsort(v,left,last-1);
    qsort(v,last+1,right);

    return;

    /* Some hokey statements: */

    ext = 34;
    while(--ext)
      { /* do nothing */
      }

    ext_ptr = &ext;
    *ext_ptr = 34;
    do {
         *ext_ptr--;
       } while(ext_ptr);

    everything_a_ok = bad_news = 0;
    if (ext == ext_ptr[0])
      everything_a_ok = 1;
    else
      bad_news = 1;

    if (5*(bad_news > 9-13-64+68))
      return dummy();
/*debug(symtab_dump)*/
  }
/*debug(variable_dump)*/
/*debug(statement_dump)*/
```

A.3 Semantic Checking (Week 13)

A.3.1 First Test

```
/* Quick sort, from 2nd ed. K&R, page 87. */

void swap(int *v, int i, int j);

void qsort(int *v,int left,int right)
  {
    int i,last;
    int new_partition_element;

    if (left >= right)
      return;

    new_partition_element = (left+right)/2;
    swap(v,left,new_partition_element);
    last = left;
    for (i=left+1;i<=right;i++)
      if (v[i] < v[left])
swap(v,++last,i);
    swap(v,left,last);
    qsort(v,left,last-1);
    qsort(v,last+1,right);
  }
/*debug(trace_types)*/
```

A.3.2 Second Test

```
float f(float x);
float f_prime(float x);
int dprint(char *s,float d);

float newton(float start)
  {
    float x[100];
    int i;

    i = 0;
    x[i] = start;
    for(i=1;i<100;i++)
      {
x[i] = x[i-1] - f(x[i-1])/f_prime(x[i-1]);
      }

    return x[i-1];
  }
/*debug(trace_types)*/
```

A.3.3 Third Test

```
void foo(void);
```

```
int bar(int i,float f,char c);
float baz(int *ip,float *fp,char *cp);
char qux(int i,float f);

int quux[32];
float corge[64];
char grault[128];

float garply(int ivar,float fvar,char cvar,int *ipvar,float *fpvar)
  { int i,j,k,l,m,n;
    float f,g,h;
    char a,b;

    /* 1. (c*i)->((c->i)*i)
  (i*f)->((i->f)*f)
  (i=f)->(i=(f->i))->i
  (c=i)->(c=(i->c))->i
     */

    cvar = ivar = 'x' * 1 * 3.5;
    cvar = ivar = cvar * ivar * fvar;
    cvar = ivar = ivar * cvar * fvar;

    /* 2. A-OK
     */

    foo();

    /* 3. (i,f,c)\(c,i,f)->(i,f,c)\((c->i),(i->f),(f->c))
  and result is ignored.
     */

    bar('x',3,32.0);

    /* 4. A-OK
     */

    if ((0.5 && i) - 1)
      { /* 4.1 (i->f)
         */
        return 0;
      }
    else
      { /* 4.2 (c-c)->((c->i)-(c->i))->(((c->i)-(c->i))->f)
 */
return '0'-'0';
      }

    /* 5. A-OK
     */
```

```
    while(qux(bar(*quux,*corge,*grault),baz(&ivar,&fvar,&cvar)))
      { /* 5.1 A-OK
 */

ivar++;
++fvar;
/* ++cvar++; */

/* 5.2 lhs: (*((fp+i)->fp))->f
       rhs: (((&i)->ip)[]i)->i
       (f=i)->f
 */

*(corge+34) = (&ivar)[34];
      }

    /* X. O-ton-o errors.  Catch at least one and halt.
     */

    return 1+2.0+foo(ipvar % 10.0);
  }
/*debug(trace_types)*/
```

A.4 Code Generation (Week 14)

```
/* null.c
 *
 * You are expected to generate:
 *   i. Declarations for the external variables
 *  ii. A .global declaration for the entry point
 * iii. Appropriate code to adjust the stack upon entry, taking into
 *      account the space needed for automatic variables
 *  iv. A ret/return instruction pair
 *
 * In fact, just run gcc -S on it, and that's [very] roughly what you're
 * expected to produce.
 *
 */

int a,c; /* Some simple definitions */
char b,e,g;
char *h;

extern int shared; /* Declare, but don't define */

int array[64]; /* Allocate pointer, and array */
char string[16];

void null(int i,char k)
  {
    int l;
```

```
  char array2[128];
  int m,n,o;
  char *p,q;

  return;
}
```