The University of Melbourne

Department of Computer Science and Software Engineering

COMP90045 Programming Language Implementation

Semester 1, 2012

Project

Aim

The main aim of this project is to give you experience with all the main parts of compilers: lexical analysis, syntax analysis, semantic analysis, code generation, and symbol tables.

A secondary aim is to give you experience with cooperative software development. While the project is an individual project in which all submitted files must be your own individual work, the test suite required to test those submitted files will be developed cooperatively by all the students in the class.

Another secondary aim is to give you experience with software maintenance. Your first stage will build upon the supplied code, and your second stage will build upon your code for the first stage.

Deadline

This project has two separate stages, each with its own deadline.

- The deadline for stage 1 is 4pm on Thursday, April 26.
- The deadline for stage 2 is 4pm on Thursday, May 17.

Specification overview

The purpose of the project is to write a simple but complete compiler. The job of this compiler is to translate programs written in the toy language PLI12 to the assembly language of the target machine T12.

- Stage 1 of the project requires you to write a scanner, a parser and a prettyprinter for PLI12. The scanner and parser together should build an abstract syntax tree (AST), which the prettyprinter should then print out in a standard format to prove the correctness (or otherwise) of the abstract syntax tree.
- Stage 2 of the project requires you to extend your code from stage 1 by adding to it a semantic analyzer and a code generator. The semantic analyzer should decorate the AST by recording the types of all expressions, precisely identifying all operators (e.g. whether + represents integer addition or real addition), and making explicit any implicit required conversions (e.g. int to real). The code generator should translate the decorated AST to the assembly language of T12. The generated assembly code can then be run on test inputs using a simulator of the T12 machine, which allows the outputs of the simulator to be compared with the expected outputs. This allows you to judge the correctness of the generated code. To make finding bugs in the generated code easier, the T12 simulator performs several data integrity checks that can reveal errors in generated instructions as soon as they are executed, instead of waiting to check the output at the end of the program. For example, the simulator reports an error when an instruction performs a load from a stack slot that has not been stored to yet.

Please read the *entirety* of this project specification before you start stage 1. You don't want to write code to perform some task only to find out later that another part of this specification requires you do that task some other way.

You should also look at the supplied code before you start stage 1, for two reasons. First, you need to know what code is already there so you can avoid wasting your time writing code that duplicates existing functionality. Second, the supplied code of the T12 simulator contains a complete and working scanner and

parser that you can use as models for the scanner and parser you have to write.

Specification of the source language PLI12

PLI12 is a toy programming language. All programs are in a single module, and consist only of functions; there are no global variables. Function definitions may contain only a very limited range of statements. There are no user-defined types and only four builtin types, all atomic: "int", "real", "bool", and "string".

Every PLI12 program consists of one or more function definitions. Each function definition contains a function header and a function body. The function header must contain the following components in this order:

- The keyword "function".
- An identifier giving the name of the function. In PLI12, an identifier consists of a sequence of one or more lower or upper case letters, underscores and numbers starting with an lower or upper case letter.
- A comma-separated list of zero or more parameters within a pair of parentheses. Each parameter consists of an identifier, a colon and a type name. Each type name is one of these four keywords: "int", "real", "bool", or "string".
- The keyword "returns".
- The name of the return type.

A function body begins with the keyword "begin" and ends with the keyword "end". In between there is a list of zero or more declarations, and a list of one or more statements. Declarations have the form

```
declare <id> <type>;
```

or the form

```
declare <id> <type> initialize to <const>;
```

where

- "declare", "initialize" and "to" are keywords;
- <id> is an identifier;
- <type> is a type name;
- <const> is an integer, real, boolean, or string constant;
- an integer constant is a sequence of one or more digits;
- a real constant is a sequence of one or more digits, a decimal point, and another sequence of one or more digits;
- a boolean constant is either of the keywords "true" and "false";
- a string constant is a sequence of characters between a pair of double quote characters, the string may not contain newline characters or unescaped double quote characters, but may contain escape sequences of the form \n to represent newline characters and/or escaped double quote characters (double quote characters preceded by a backslash).

Statements have one of the following forms:

```
<id> := <expr>;
read <id>;
write <expr>;
if <expr> then <stmtlist> endif
if <expr> then <stmtlist> else <stmtlist> endif
while <expr> do <stmtlist> endwhile
return <expr>;
```

where

- "read", "write", "if", "then", "else", "endif", "while", "do", "endwhile" and "return" are keywords;
- <stmtlist> is a list of one or more statements, with no separator between statements (though the atomic statements in the list will have their semicolon terminators).

Expressions have one of the following forms:

```
<id>
<const>
( <expr> )
<expr> <binop> <expr>
<unop> <expr>
<id> ( <exprlist> )
```

The list of operators is

```
or
and
not
= != < <= > >=
+ -
* /
-
```

The "not" operator is an unary prefix operator, The "-" operator is an unary prefix operator as well as a binary operator. All the other operators are binary. All the operators on the same line have the same precedence, and the ones on later lines have higher precedence; the highest precedence is unary minus. The relational operators are not associative; the other operators are all left associative.

<exprlist> is a list of one or more expressions separated by commas. Note that this means that although you can *define* functions with zero parameters in PLI12, you cannot *call* such functions. This is because in the absence of global variables, a function without parameters cannot be told by its caller what to do. The ability to define functions with no parameters is nevertheless needed because every PLI12 program needs to have an entry point (see below).

PLI12 supports comments, which start at a # character and go on until the end of the line. White space is not significant, but you will need to keep track of line numbers for use in error messages.

The semantic rules of PLI12 are as follows.

- It is an error to define more than one function with the same name in the program.
- It is an error to define more than one variable, including parameters, with the same name in any given function. (It is ok to declare variables of the same name in different functions.)
- In assignment statements, the identifier on the left hand side and the expression on the right hand side must have the same type, with one exception: an integer expression may be assigned to a real variable (the compiler must convert the value to be assigned from integer to real).
- Initializations in declarations should be treated as assignments to be executed when the function is called, assigning the given constant to the declared variable, so they have the same rules about types as assignments.
- The conditions of "if" and "while" statements must be boolean expressions. The boolean expression is always fully evaluated; there is no short circuit evaluation.
- An expression that is a constant has the type appropriate to the constant.
- An expression that is a variable has the declared type of the variable. It is an error for a function to use an undeclared variable.

- The two operands of the operators "and" and "or" must each be booleans, and their result is boolean.
- The one operand of the operator "not" must be boolean, and its result is boolean.
- The one operand of the unary minus operator may be either integer or real, and its result has the same type as the operand.
- The two operands of the operators "=", "!=", "<", "<=", ">" and ">=" must either have the same type, or one must be an integer and the other a real, in which case the compiler must convert the integer to a real before the comparison. The result is a boolean.
- The two operands of the operators "+", "-", "*" and "/" must be either two integers, two reals or one of each. If both operands are integers, the operation operates on integers and the result is an integer. If at least one operand is a real, the operation operates on reals and the result is a real. If one operand is an integer and the other is a real, the compiler must convert the integer to a real before the operation.
- Every function called by a PLI12 program must be either a function defined in that program or a builtin function of the PLI12 language. The list of builtin functions is as follows.
  - "string_concat" takes two string arguments, and returns a string, which will be the concatenation of the two input strings.
  - "string_length" takes a single string argument, and returns an integer, which will be the length of the input string.
  - "substring" takes three arguments, the first being a string and the second and third being integers, and returns a string. substring(s, i, j) returns the substring of s that starts at offset i and continues on for j characters, unless the string s ends first. The first character of a string is considered to be at offset 0.
  - "sqrt" takes a single real argument, and returns a real, which will the square root of the input argument.
  - "round" and "trunc" both take a single real argument and return an integer. "round" rounds the input real to the nearest integer, while "trunc" truncates it to the nearest integer that is not greater than the real.
- The number of actual parameters must match the number of formal parameters.
- The types of the actual parameters and the corresponding formal parameters must be the same, with one exception: If an actual parameter is an integer and the formal parameter is a real, the compiler must convert the integer to a real before the call. This rule applies both to user-defined functions and builtin functions.
- The type of the expression supplied to a return statement must be the same as the return type of the function containing the return statement.
- The program must have a function named "main", which must have no parameters, and whose return type must be "int" (although the return value of main is ignored by the T12 simulator).

Specification of the target machine T12

T12 is an artificial target machine that closely resembles a subset of the intermediate representations used by many compilers.

T12 has 1024 registers named r0, r1, r2, ... r1023. This is effectively an unlimited set of registers, and your compiler may treat it as such; your compiler may generate register numbers without checking whether they exceed 1023. Every register may contain a value of any type.

T12 also has an area of main memory representing the stack, which contains zero or more stack frames. Each stack frame contains a number of stack slots. Each stack slot may contain a value of any type, and you specify a stack slot by its stack slot number. Stack slot numbers start at zero.

A program in T12 assembler consists of a sequence of instructions, some of which may have labels. Although the simulator does not require it, good style dictates that each instruction should be on its own line. As in most assembly languages, you can attach a label to an instruction by preceding it with an identifier (the name of the label) and a colon. The label and the instruction may be on the same line or different lines. Identifiers have the same format in T12 as in PLI12.

The following lists all the opcodes of T12, and for each opcode, shows how many operands it has, and what they are. The destination operand is always the leftmost operand.

```
push_stack_frame     framesize
pop_stack_frame      framesize

load                 rN, slotnum
store                slotnum, rN

int_const            rN, intconst
real_const           rN, realconst
bool_const           rN, boolconst
string_const         rN, stringconst

add_int              rN, rN, rN
add_real             rN, rN, rN
sub_int              rN, rN, rN
sub_real             rN, rN, rN
mul_int              rN, rN, rN
mul_real             rN, rN, rN
div_int              rN, rN, rN
div_real             rN, rN, rN

cmp_eq_int           rN, rN, rN
cmp_ne_int           rN, rN, rN
cmp_gt_int           rN, rN, rN
cmp_ge_int           rN, rN, rN
cmp_lt_int           rN, rN, rN
cmp_le_int           rN, rN, rN
cmp_eq_real          rN, rN, rN
cmp_ne_real          rN, rN, rN
cmp_gt_real          rN, rN, rN
cmp_ge_real          rN, rN, rN
cmp_lt_real          rN, rN, rN
cmp_le_real          rN, rN, rN
cmp_eq_bool          rN, rN, rN
cmp_ne_bool          rN, rN, rN
cmp_gt_bool          rN, rN, rN
cmp_ge_bool          rN, rN, rN
cmp_lt_bool          rN, rN, rN
cmp_le_bool          rN, rN, rN
cmp_eq_string        rN, rN, rN
cmp_ne_string        rN, rN, rN
cmp_gt_string        rN, rN, rN
cmp_ge_string        rN, rN, rN
cmp_lt_string        rN, rN, rN
cmp_le_string        rN, rN, rN

and                  rN, rN, rN
or                   rN, rN, rN
not                  rN, rN
```

```
int_to_real          rN, rN

move                 rN, rN

branch_on_true       rN, label
branch_on_false      rN, label
branch_uncond        label

call                 label
call_builtin         builtin_function_name
return

debug_reg            rN
debug_slot           slotnum
debug_stack

halt
```

The "push_stack_frame" instruction creates a new stack frame. Its argument is an integer specifying how many slots the stack frame has; for example the instruction "stack_frame 5" creates a stack frame with five slots numbered 0 through 4. (In the simulator, it also reserves an extra slot, slot 5, to hold the size of the previous stack frame, for error detection purposes.)

The "pop_stack_frame" instruction deletes the current stack frame. Its argument is an integer specifying how many slots that stack frame has; it must match the argument of the "push_stack_frame" instruction that created the stack frame being popped.

The "load" instruction copies a value from the stack slot with the given number to the named register. The "store" instruction copies a value from the named register to the stack slot with the given number.

The "int_const", "real_const", "bool_const" and "string_const" instructions all load a constant of the specified type to the named register. The format of the constants is exactly the same as in PLI12.

The "add_int", "add_real", "sub_int", "sub_real", "mul_int", "mul_real", "div_int" and "div_real" instructions perform arithmetic. The first part of the instruction name specifies the arithmetic operation, while the second part specifies the shared type of all the operands.

The "cmp_eq_int", "cmp_ne_int", "cmp_gt_int", "cmp_ge_int", "cmp_lt_int" and "cmp_le_int" instructions, and their equivalents for the other builtin types, perform comparisons, generating boolean results. The middle part of the instruction name specifies the comparison operation, while the last part specifies the shared type of both input operands.

The "and", "or" and "not" instructions each perform the boolean operation of the same name.

The "int_to_real" instruction converts the integer in the source register to a real number in the destination register.

The "move" instruction copies the value in the source register (which may be of any type) to the destination register.

The "branch_on_true" instruction transfers control to the specified label if the named register contains the boolean value "true". The "branch_on_false" instruction transfers control to the specified label if the named register contains the boolean value "false". The "branch_uncond" instruction always transfers control to the specified label.

The "call" instruction calls the user defined function whose code starts with the label whose name is the operand of the instruction, while the "call_builtin" instruction calls the builtin function whose name is the operand of the instruction. All functions, whether user defined or builtin, take their first argument from r0, their second from r1, and so on. During the call, the function may destroy the values in all the registers, so

they contain nothing meaningful when the function returns. The exception is that functions that return a value put their return value in r0. When the called function executes the "return" instruction, execution continues with the instruction following the call instruction.

The list of builtin functions is the following:

```
read_int
read_real
read_bool
read_string

print_int
print_real
print_bool
print_string

string_concat
string_length
substring
sqrt
trunc
round
```

The read functions take no argument. They read a value of the indicated type (using scanf) from standard input, and return it in r0.

The print functions take a single argument of the named type in r0, and print it to standard output; they return nothing.

The string_concat function takes two string arguments in r0 and r1, and returns a string representing their concatenation in r0.

The string_length function takes a single string argument in r0, and returns an integer giving its length in r0.

The substring function takes a string argument in r0, and two integer arguments (the position and length) in r1 and r2, and returns a string in r0. substring(s, i, j) returns the substring of s that starts at offset i and continues on for j characters, unless the string s ends first. (The first character of the string is at offset 0.)

The sqrt function takes a single real argument in r0 and returns the square root of that argument (a real) in r0.

The trunc and round functions each take a single real argument in r0 and return an integer in r0. That integer will be the truncated or rounded integer version of the input real.

Each "call" instruction pushes the return address (the address of the instruction following it) onto the stack. The "return" instruction transfers control to the address it pops off the top of the stack.

The "debug_reg", "debug_slot" and "debug_stack" instructions are T12's equivalent of debugging printfs in C programs: they print the value in the named register or stack slot or the entire stack. They are intended for debugging only; your submitted compiler shouldn't generate them. If your code generator generates T12 code that does the wrong thing but you don't know why, you can manually insert these instructions into the T12 code and execute the modified program in the simulator to gain insight.

The "halt" instruction stops the program.

T12 supports comments, which start at a # character and go on until the end of the line.

Specification of stage 1

For stage 1 of the project, you must write a *prettyprinter* whose job is to read in programs written PLI12, accepting syntactically correct programs with any formatting, and writing them out again in a standard format.

The main part of this stage is learning how to use flex and bison and writing the scanner and parser specifications. Since PLI12 is a toy language and only toy programs will be written in it, you don't have to worry about the performance of the scanner and parser; for example, you don't have to worry about copying strings unnecessarily in the scanner or using too much stack space in the parser. However, you do have to make sure that your bison specification yields no conflicts.

The task of the prettyprinter is to check whether your scanner and parser work correctly. However, to allow the output of the prettyprinter to be checked automatically against the expected output files of the prettyprinter test cases, that output must follow a defined format.

The output of your prettyprinter should not contain comments. (Most prettyprinters preserve comments, but in this project the prettyprinter is only a stepping stone towards a compiler.) The output of your prettyprinter should replace whatever white space exists in the input with standardized white space, in which tokens are separated from each other by a single space, with the following exceptions/additional rules:

- Successive functions should be separated by a blank line.
- In each function, the list of declarations (if any) and the list of statements should be separated by a blank line.
- Every declaration and every statement should start on a new line.
- Function headings and the "begin" and "end" keywords surrounding the body of a function should begin at the start of a line.
- Declarations and the outermost statements should be indented by four spaces. The statements inside if-then-elses and while loops should be indented four spaces more than the if-then-else or the while loop they are part of.
- There should be no white space before the semicolons terminating atomic statements.
- White space should of course be preserved inside string constants.
- When printing expressions, you should print parentheses around every operand of a unary or binary operator that is not a constant or a variable.

In general, try to make the output of your prettyprinter as close as possible to the stage 1 expected output files of the provided test cases.

While a real prettyprinter would have to ensure that no line in its output is longer than some limit (typically 80 characters), your prettyprinter has no such responsibility.

For syntactically incorrect programs, the prettyprinter should print only an error message. Syntax error handling is not a priority in this project, so the precise text of this error message does not matter, and you do not have to attempt to recover after syntax errors. (Syntax error recovery is very important in practice, but it also happens to be very difficult to do well, so including it in the project would have an unfavorable cost/benefit ratio. This is also the reason why the lectures don't spend much time on this topic.)

Prettyprinters can work either by emitting a standardized version of the input as it is being parsed, or by building an AST and then printing it when it is complete. Only the second approach is useful as a stepping stone towards a compiler, but during one of the previous offerings of this subject, about 30% of the students chose the first approach. To prevent that from happening again, your prettyprinter must print the functions of the program in a standard order, which is lexicographic order on the function name. If two functions have the same name, which is possible though it is a semantic error, keep those two functions in their original order.

In previous offerings of this subject, some students wrote their own sorting code just for putting the functions in order. There is no need for that, since the Unix standard library has functions for sorting. For an example, see the function "report_all_errors" in the supplied source file "pli12c.c".

Specification of stage 2

For stage 2 of the project, you must write a *semantic analyzer* and a *code generator*.

The job of the semantic analyzer is

- to decorate the AST built by the code for stage 1 with type attributes,
- to identify which instruction should be used to implement each unary and binary operator in every expression,
- to insert the unary operator "int_to_real" at each location where an implicit integer to real conversion is needed,
- to convert every occurrence of the unary minus operator to use the binary subtraction operator and zero, and
- to detect and report every violation of the semantic rules of PLI12. (You don't have to detect and report logic errors in the PLI12 program, such as forgetting to include a return statement at the end of a function.)

If the program contains some semantic errors, try to report as many of them as possible; semantic analyzers should in general try to find and report all the semantic errors in the input program. However, input statements that contain more than one error give the semantic analyzer a choice: it can report any nonempty subset of those errors. The course of action that is most helpful to the user is reporting all errors that aren't a consequence of earlier errors. However, neither compilers nor compiler writers can read programmers' minds, so they cannot tell for sure which errors are consequences of earlier errors. The best compiler writers can do is estimate whether an error is *likely* to be a consequence of another error. For example, if an expression involves an undeclared variable, then the actual error could be either that (1) the variable name is misspelt in the expression, or (2) the variable's declaration is missing. However, any later reference to a variable with the same name means that (2) is more likely, and this in turn means that issuing a second message about the variable being undeclared is likely to more annoying than useful.

We will provide some test cases for your semantic analyzer. In general, you should try to make the format of each kind of error message match the provided expected output files of these test cases (some of the provided code is designed to make this a bit easier), and you may wish to try to match the set of error messages printed for each erroneous statement as well, but these are only goals, not requirements. The *requirement* on your semantic analyzer is that it must report at least one semantic error in every input statement that contains one or more semantic errors, and that report must include a line number that allows the location of the statement. Which one of the errors it reports, which additional errors it reports (if any), exactly what format the error reports take, and (if the statement extends over more than one line) which of these lines the error messages refer to, are all up to you.

If the program contains no semantic errors, then the output of your semantic analysis will be the decorated AST, which will be the input of the code generator. The job of the code generator is to convert to T12 assembly code the decorated AST of any PLI12 program free of syntactic and semantic errors. The generated T12 code, when executed with the T12 simulator, must do what the original PLI12 program says it should do. It must also conform to the following conventions about stack frames, runtime systems and symbol names:

The stack frame for a function with N variables that needs M temporaries should consist of the following slots:

- one slot holding the return address,
- one slot holding the size of the previous stack frame, if any,
- N+M slots holding the values of the function's variables and temporaries.

The slot holding the return address is created and filled in by the call instruction in the caller. The other slots are created by the push_stack_frame instruction in the callee which also fills in the slot holding the size of the previous stack frame (which is remembered by the T12 simulator for error detection purposes, although most real machines do not do this.) The slots created by the push_stack_frame instruction in the function prologue (whose argument should be N+M) should be removed by a matching pop_stack_frame instruction in the function epilogue. The slot holding the return address will be removed by the return instruction.

The T12 simulator starts execution with the first instruction in the program and stops when it executes the halt instruction. Your code generator must therefore ensure that the generated code starts with a fixed two-instruction sequence that represents the T12 runtime system: the first instruction calls main, while the second (executed when main returns) is a halt instruction.

You should make sure that the names of labels generated by the compiler cannot clash with the names of labels representing the starts of functions *without* making any assumptions about the names of functions in PLI12 programs, beyond the obvious fact that they are PLI12 identifiers.

Software architecture

The compiler you build will be named pli12c, for PLI12 compiler. It will take some options and two arguments. The first argument will always be the name of a file containing a PLI12 program; by convention, this filename should end with a ".pli12" suffix. The second argument will always be the name of the file where pli12c should put whatever output the user requests.

- When invoked as "pli12c -p prog.pli12 outputfilename", pli12c will do scanning, parsing and prettyprinting only, putting the prettyprinted version of the AST of prog.pli12 in outputfilename if the program has no syntactic errors.
- When invoked as "pli12c prog.pli12 outputfilename", pli12c will do full compilation, putting the generated t12 assembly code in outputfilename if the program has no syntactic or semantic errors. By convention, the names these output files should end with a ".t12" suffix.

If pli12c detects any errors, it won't touch the named output file. Instead, it will print error messages to standard error.

The compiler you build should consist of the following modules. Some modules will be supplied for you to use as is, while others modules will be supplied in skeleton form for you to fill in.

- The pli12c module contains the top level of the compiler. It looks after option handling and errors reporting. (You should call the error reporting functions defined in this module to report any errors detected by your semantic analyzer.) You should not need to modify the supplied version of this module, although you may do so if you wish.
- The pli12c_getopt and pli12c_getopt1 modules are copies of the getopt library. The pli12c module uses these modules instead of whatever version of getopt is built into the system because some versions of Unix do not include getopt in the standard C library. You should not modify the supplied versions of these modules.
- The files scanner.l and parser.y should contain the scanner and parser specifications for PLI12 using flex and bison respectively. You will have to write the code yourself.
- The ast module contains a skeleton of a module for defining the abstract syntax tree (AST) and performing basic operations on it. Most of this module you will have to write yourself. We have supplied some parts of it in order to reduce irrelevant differences between different students' projects, which makes marking easier.
- The pretty module should contain the prettyprinter. You will have to write the code yourself; we have supplied only the prototype (which pli12c.c relies on).
- The analyze module should contain the semantic analyzer. You will have to write the code yourself; we have supplied only the prototype (which pli12c.c relies on).
- The codegen module should contain the code generator. You will have to write the code yourself; we have supplied only the prototype (which pli12c.c relies on).
- The t12 module contains an intermediate representation (IR) with a one-to-one correspondence to T12 instructions. You can choose to use this IR as the target of your code generator, since t12.c contains a function that prints out programs expressed in this IR in the T12 assembly language. You should not need to modify the supplied version of this module, although you may do so if you wish.
- The symbol module should contain the symbol table. You will have to write the code yourself; we have supplied only the header file, which you may alter as you wish.

We also supply a Makefile for building the "pli12c" executable.

The compiler should follow the conventions taught in our second year subjects for type safety and information hiding in all its modules, excepting the ones whose C source code is generated by flex or bison. (The original

lex and yacc programs on which flex and bison are based aren't designed to follow those conventions.) The provided file missing.h acts as the header file for the two C source files generated by flex and bison, since they do not generate header files that follow our type safety conventions. However, even with this header file, compiling the C source files generated by flex and bison may generate warnings from the C compiler, due to issues such as flex-generated code defining some local functions without declaring them first. If the warnings refer to code that you didn't write yourself, you can ignore them.

Supplied files and test suite

We have supplied you with several files to help you concentrate on work relevant to COMP90045 without having to spend too much time on other things. These files should soon be available in the named subdirectories of the directory /home/subjects/comp90045/local/project.

The subdirectory "pli12c" contains the supplied parts of the PLI12 compiler.

The subdirectory "t12" contains the source code of the T12 simulator.

The subdirectory "tests" contains a test suite. The script called "Runtests" allows you to test your compiler by running all the test cases. Each test case consists of a PLI12 source file whose name has the suffix ".pli12" as well as one or more expected output files.

For stage 1, the output of the prettyprinter when run on a test program such as stddev.pli12 will be expected to match either stddev.pretty_exp1, or stddev.pretty_exp2, or stddev.pretty_exp3 and so on. We allow more than one expected output file because the rules for stage 1 don't necessarily resolve all questions about exactly how much white space to put where, although students should strive to minimize spurious differences from the provided expected output files.

For stage 2, the output of the compiler depends on whether the program has any semantic errors or not.

Given a semantically incorrect test program such as bad1.pli12, the error output of the compiler will be expected to match either bad1.error_exp1, or bad1.error_exp2, or bad1.error_exp3 and so on. We allow more than one expected error output file since the rules for stage 2 don't precisely prescribe either the format of error messages (though here also students should strive to minimize spurious differences from the provided expected error output files) or exactly which errors the semantic analyzer should report for a statement that has several errors.

For a semantically correct source file such as stddev.pli12, the output of the compiler should be a T12 program. Each such test case should have one or more test inputs stored in files named stddev.run_in1, stddev.run_in2, and so on. For each of these input files, running the simulator on the T12 program generated by the compiler and giving it e.g. stddev.run_inN as input should generate output that precisely matches stddev.run_expN, for every N. In this stage, there is no prettyprinting and no ambiguity about what is correct output.

The test suite directory will initially contain a few test cases to get you started. However, the suite of test cases should evolve and grow during the semester. Every student doing this project has write access as well as read access to this repository, and is expected to contribute to the development of the test suite. Such contributions are part of your mark, although we want to encourage students to contribute more than the marking scheme strictly requires. Cooperative development of a test suite is good practice for work on significant-sized software development projects either in your further studies or in industry after your graduation.

Any files you add to the test suite will need to be usable by your classmates, and you must set up their permissions accordingly. If you have any concerns or questions about any test case, you should post them on the LMS.

Although every effort has been made to ensure the quality of the code we have provided to you, it may still contain bugs. If you find one, please report it to me (zs); I will fix it, make the fixed version available, and announce that fact.

The supplied pli12c.c has limits on how many error messages a PLI12 program can generate. The T12 simulator has some fixed limits on how big the stack can get and on how many labels and instructions the program may contain. These limits are high enough that they should not be a problem, but if they are, please

report it to me as well as fixing it for yourself in your own copy.

Questions

If you have any questions about any aspect of the project, you should ask on the LMS, so that other students can also read the answers.

Submission and marking

The files that must be included in each submission are the following.

for stage 1	for stage 2
ast.c	ast.c
ast.h	ast.h
scanner.l	scanner.l
parser.y	parser.y
pretty.c	pretty.c
	symbol.c
	analyze.c
	codegen.c

These will be put together with the supplied versions of the other files involved in the pli12c program to generate an executable. If you have modified any of the supplied files in the pli12c directory (e.g. pretty.h, pli12c.c or the Makefile), then you must submit the modified file as well. It is ok for analyze.c, symbol.c and codegen.c to remain in their skeleton states for stage 1; that's why you don't have to submit those files for those stages. For stage 1, pli12c needs to work only when invoked with -p. For stage 2, pli12c needs to work in all cases: when invoked with -p *and* when invoked without options.

We require you you to use a software configuration management tool such as RCS, CVS or subversion to record your progress during the development of your code for the project. Which tool you use is up to you, but you must specify the pathname of the repository (which must be on a CSSE machine) in a comment at the top of ast.c. While we will normally look at only the last submitted version of each file, we reserve the right to look at the development history in cases of disputed authorship, and we will be actively looking for submissions that are too similar. Your submission is supposed to reflect your own unaided work.

You should submit your files for stage *N* by first packaging all the files you want to submit into a single archive (using either *zip* or *tar*) and then sending an email to *zs@unimelb.edu.au* containing that archive file as an attachment. The email should have the subject line *COMP90045 SUBMIT N*; if it does not, it is at great risk of being discarded as spam.

To get full marks, you must submit your work before the deadline. Late submissions will incur a penalty of 1 mark per day (or part thereof) for the first two days after the deadline and a penalty of 2 marks per day (or part thereof) after that, unless you get permission for a late submission in advance. Note that machine breakdowns (within reason) are an accepted fact of life, and you must learn to deal with them. (One way you can do this is by starting work on your projects as soon as possible.) Therefore they will not be acceptable as an excuse for not having a project completed by the due date.

A working program, while important, is not the only criterion for assessment. You may also be assessed on your understanding of the project, as evidenced by the design of your solution; your programming style with respect to clarity, logical structure, and modularity; the robustness of your program; and your standard of documentation. The last is especially important: it is your responsibility to help us understand your programs.

Efficiency is not a major concern in this project; using e.g. linear search in the symbol table is fine.

Please avoid using any software libraries that aren't universally available. If you aren't sure about a library, please check with me first.

The project is worth 30% of the marks for the subject. Stage 1 is worth 10%, while stage 2 is worth 18%; the final 2% is for adding a test case to the test suite at least 72 hours before the due date of each of the two stages.