Errors and Warnings

Errors may arise during the compiler's analysis of the input program because the program is incorrect. We distinguish between **fatal errors**, in which the compiler is unable to ascertain the meaning of a piece of the program and will therefore be unable to generate code for it, and **warnings**, where the compiler notices something which may be an error, but still has a meaningful way of continuing.

Errors and warnings may arise from any of the analysis stages. Lexical and syntactic fatal errors will be caught by lexers and parsers generated by tools, such as lex/flex and yacc/bison. Consider the simple expression syntax which has been used repeatedly as an example. If the input sequence is `x = a / / b ;` there is no valid derivation corresponding to the input, and therefore the parser must declare a fatal error.

Semantic errors are caught by the code which the compiler writer adds as part of the semantic analysis phase. As an example, consider the expression statement `(b+a)=c;` Although it is syntactically valid, it is a fatal semantic error, because there is no meaningful way that the compiler can generate code to assign to `(b+a)`. (I.e. (b+a) is not a lvalue). On the other hand, assigning a pointer to an integer is a warning; the compiler can generate meaningful and correct code, but there is a possibility that the programmer has written the wrong thing, and that the program will crash or produce invalid results.

Error Reporting and Recovery

It is important to give the programmer good information about what the error or warning is and where in the program to go about finding it. At the very least, the file name and line number at or near the location of the detection of the error should be reported. The error/warning message should be clear and specific.

Giving the exact location of an error can be problematic. A given language construct can span several lines. Furthermore, the parser looks ahead to the next token, so the lexer's idea of what the current line is may be one off. E.g.:

```
int i;
char j              //semicolon missing
int k;
```

The compiler will probably report this error at line 3, even though common sense tells us that it is really at line 2.

For reporting semantic errors, it is a good idea to track where things are defined. Then, e.g., if there is a conflicting definition, the error message can give both the first and the second places.

The bison parser generator includes optional support for adding a third parse stack. In addition to the state stack and the semantic value stack, a **location stack** can be used to

track information about the location of terminals and non-terminals in a given rule. By default, Bison uses a data structure which uses line:column pairs, and tracks both the starting and ending position of a symbol. It includes automatic rules to construct these as a result of each shift/reduce. Errors can then be reported as start_line:start_col - end_line:end_col. This can be especially useful when the compiler is front-ended by a GUI and the user finds it important to see the errored code highlighted on the screen. The reader is referred to the Bison documentation for more information.

Error Recovery

When a fatal error occurs, everything from that point onward is suspect. One possible approach is simply to halt at the first fatal error. But, in many cases, it is helpful to the programmer to see all of the errors which would have been caught had compilation been allowed to continue. In order to do this, the compiler must perform **error recovery** to get itself back to a state where it can meaningfully process input.

For semantic errors, recovery might be as simple as ignoring the offending declaration, expression, statement, etc. For syntax errors, the compiler writer needs to code special error recovery into the grammar. In yacc/bison, one can create a rule with the special "error" token:

```
stmt:expr ';'
   |IF '(' expr ')' stmt ';'
   |/*other instances of statement productions */
   |/*                 ...                     */
   |/*                 ...                     */
   |/*                 ...                     */
   |error ';'
   ;
```

When parser reaches a state in which the lookahead token can produce no valid shifts or reductions, it starts popping states (and semantic values, and locations, if enabled) until it reaches a state in which the introduction of the `error` token would be a valid shift. Then, if the old lookahead token is still no good, the parser consumes and discards tokens until it finds one which leads to a valid shift or reduce. In the example above, let us say the input is `if () x=1;` An error will be detected at the closing parenthesis. The parser will pop back to the state `stmt: . error ';'` and then shift the fictitious `error` token. It will then discard the rest of the flawed if statement until it reaches the semicolon, and then use the error production to reduce to stmt. In this way, the mangled input is recognized as if it had been a valid statement, and therefore subsequent statements are syntically in the correct place to continue parsing.

When the syntax error happens, `yyerror()` is called. Typically this function is supplied by the compiler writer and outputs the error message. Note that yacc/bison pass as an argument a default message, usually "syntax error", although Bison can be

instructed to given better errors including which tokens would have been acceptable, using the `%error-verbose` directive.

After error recovery, the parser will wait until 3 tokens have been successfully shifted before calling yyerror again (but error recovery will be done). The macro `yyerrorok` can be placed in an action in an "error" rule to restart error messages.

Because semantic values are discarded during error recovery, memory which they consume can be leaked, depending on the memory allocation method used. Error recovery can also leave the value stack in a state where a sensible value would normally be expected in a certain place, but now that value is undefined. The reader should consult the Bison manual for further discussion of these issues, including a way to associate a destructor method with semantic values.

Errors which are detected after the first fatal error are, themselves, suspect. Many times, the compiler's attempt to recover from an error will result in many more spurious errors. Consider what would happen if a missing brace caused the compiler to think an entire function was actually in global scope!

It is up to the compiler writer to make sure that a syntax error is remembered as a fatal error. Although the error recovery allowed further input to be parsed and examined for additional errors, the result (skipping the flawed statement) is most certainly not what the programmer wants the code to do!

Debugging and Warning Levels

It's a good idea to extend the error reporting mechanism to support internal debugging of the compiler, especially during the development stage. Some compilers extend this concept even further, allowing command-line flags to specify either the level of debugging verbosity, or to control debugging on a subsystem basis, e.g. output debugging information about AST construction only.

Likewise, sophisticated compilers offer command-line controls on the level of warnings desired, ranging from almost nothing (the compiler silently ignores potentially dangerous things such as pointer/integer assignments) to almost everything (e.g. reporting unreachable code, possible integer overflows).

Defensive Coding

The compiler is a complicated software system. Bugs can and will happen! It is best to practice "defensive coding" with checks for "can't happen" situations which indicate a bug in your compiler. When a bug is detected, print out as much information as possible to help the developer (that's you) figure out the cause of the bug. Consider using a stack traceback function and/or the abort() function which will leave a core dump. Often these features are tunable from the command line.

Memory Allocation Strategies

A compiler is a difficult problem for efficient memory allocation. Because it builds and discards many temporary data structures, such as those to keep track of types, definitions, values, etc., optimization of the memory allocation process can be a significant source of compile speed improvement.

The default memory allocator, `malloc`, is not very efficient at the allocation of many small, variably-sized objects, either in terms of speed or memory consumption. When processing scoped languages such as C, the pattern of memory allocation and release resembles the ebb and flow of a stack. During the analysis of a function, for example, the compiler consumes a large amount of temporary memory tracking types, expressions, statements and intermediate code. Since most C compilers work on a function-by-function basis, almost all of that memory gets de-allocated after the function has been processed.

This leads to a possible avenue of extreme speed improvement. Rather than taking a traditional nested constructor/destructor approach, we can use a strategy which is called **arena-based memory allocation**, or "**mark/release.**" We associate a growable pool of memory with objects that have to be allocated when looking at a function. To allocate an object out of that pool, we simply dole out this memory linearly, without trying to keep track of which object is which. At the end of the function, the entire pool is released back to the operating system, without any explicit (and potentially nested and costly) destructor/deallocation calls.

We can create multiple arenas, e.g. one which is for global declarations and lasts the entire lifetime of the compilation process, and another which is used on a function-by-function basis. In a typical C compiler, attempts to extend this to a finer level, e.g. a separate arena for structures, blocks or prototypes, do not work out because of the need to save some information in those scopes beyond their lexical end-of-scope marker (e.g. the closing curly brace).

On modern computer systems, speed efficiency tends to be more important than space efficiency for the kinds of problems which compilers tackle. Rarely does the memory consumption of a compiler, even one which is inefficiently coded, begin to tax the resources of the system for most programs of reasonable size and complexity. If, however, the compiler is running not on a general-purpose desktop or server system, but rather in an embedded environment, then optimization for memory usage may become very important. In these environments, strategies such as one-pass compilation, long out of favor for general-purpose compilers, may be preferable.

Not all parsers run just once. Many interpreters run indefinitely. Consider the UNIX command-line "shell" interpreter. Also, parsers may be embedded into other long-running code, such as a web server daemon. In these contexts, it is important to eliminate all memory leaks, because even when the system has a large amount of memory, it is still

finite, and leaks will eventually lead to failure.