The Evolution of Compilers

Priti Shankar

Priti Shankar is with the Department of Computer Science and Automation at the Indian Institute of Science, Bangalore. Her interests are in theoretical computer science and error correcting codes.

The term *compiler* was coined in the late forties, by Grace Murray Hopper, a pioneer who rose to the challenges of programming the first computers. The problem of translation from a source language into a target language was viewed as a 'compilation' of a sequence of machine language subprograms selected from a library. In this article we briefly trace the evolution of compilers from their beginnings as huge sprawling algorithms in the early fifties, to their current elegant, phase ordered forms.

1. Introduction

The year 2006 marked a special event in the history of the Turing award, perhaps the most prestigious award in the field of computer science for a contribution of lasting and major importance to the field. For the first time in its forty year old history, the award went to a woman – Frances Allen. Allen was awarded the prize for her "pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution". She was a member of the team in IBM, responsible for the early development of compiler techniques. Interestingly, the *first* Turing award in 1966 went to Alan Perlis, for "developing a theory of advanced programming languages and compiler construction". That the subject has managed to sustain itself over the last forty years or so, is an indication of its enduring relevance to the field of computer science.

Keywords
Lexical analysis, syntax analysis, parse tree, code generation.

Programming languages are now used for tasks far more general that what we understand to be "ordinary

computation". For example TEX and LATEX use compilers to translate formatting commands into intricate typesetting commands. In fact, this article itself is the result of a program that parsed a LATEX program and converted it into a PDF file. Postscript, generated by text formatters like LATEX and Word is actually a programming language. It is translated by a compiler and executed by laser printers and document previewers to produce a readable form of a document. A language like VHDL supports the creation of VLSI circuits. Just as an ordinary compiler enforces the rules of the syntax of a programming language, a *silicon compiler* enforces the design rules of a circuit, for example the minimum separation between wires and so forth. What is the common framework that enables the same technology to be used for all these tasks? Below the surface, the theme is constant–all compilers execute the *same* sequence of phases, and if at all they go through any optional phases, these have fixed positions in the sequence.

Compiler design is replete with beautiful examples where complicated problems that arise in the real world are solved by constructing a mathematical model which abstracts the essence of the problem. This allows initial ad hoc solutions to a problem to be replaced by elegant algorithms.

Donald Knuth, a pioneer in this field, pointed out in an article in 1962 on the history of compiler writing, that it was difficult in the early stages of development, "to explain the internal mechanism of a compiler as the various phases of translation were all jumbled together into a huge sprawling algorithm. The passage of time has shown how to distinguish the various components of this process, revealing a basic simplicity."

In 1953 IBM introduced an early "automatic programming system": Speedcode for the IBM 701 computer. The idea was to present to an user, an instruction repertoire much larger than what was provided by a given machine, and thus reduce the burden on the programmer.

The first FORTRAN compiler appeared in 1957 and took 18 man years of effort. Initially users were skeptical about its efficacy, as they felt that the quality of code generated could not match that of hand-written code.

In an early paper, John Backus (the scientist featured in this issue of *Resonance*) and Harlan Herrick who were part of the IBM team, stated that there are two methods by which automatic programming systems make these non-machine operations available to the user: the *interpretive* method and the *compiling* method. This statement made about fifty years ago, is valid even today. An interpreter is simply a program which takes as input some *representation* of the source program and simulates the operations that the machine would carry out if it were capable of directly executing programs written in that language. A compiler on the other hand first translates the source program into the machine language and then executes machine level instructions.

By 1954 IBM created the FORmula TRANslation (FORTRAN) System, a set of programs that enabled the IBM 704 to take a formulation of a mathematical problem in precise mathematical notation, and to produce a high speed 704 program for it. The first FORTRAN compiler appeared in 1957 and took 18 man years of effort. Initially users were skeptical about its efficacy, as they felt that the quality of code generated could not match that of hand-written code. While FORTRAN was written for mathematical computing, another programming language COBOL (COmmon Business-Oriented Language) was being developed by Grace Hopper and her group. By 1962 there were 43 different compilers for FORTRAN. While the compilers for COBOL were not as good, it continued to be used by the US Department of Defense, and by the early sixties it was used worldwide for mechanized accounting.

Over the years, programming languages have been augmented with all kinds of features, like block structure, variable declaration sections, procedures and functions with different kinds of capabilities, and various forms of parameter passing, recursion, information encapsulation and so forth. While a discussion of how all these

features are handled by a compiler is beyond the scope of this article, we will concentrate on *imperative* languages, where languages are structured as a sequence of statements. The various models are described below:

1. *Imperative:* Here the program is viewed as modifying a set of states which are characterized by values in registers, memory and external storage, and the goal is to reach a desired final state via the execution of a sequence of statements. Languages of this kind include FORTRAN, Pascal, C, COBOL, Algol and Ada.

2. *Functional:* Functional programming languages are described in the article by Madhavan in this issue of *Resonance*. Here we describe the function that must be applied to the initial state to get the final result. Complex functions are built from simpler ones until we obtain a final function that gets us the desired result. Examples of functional languages are LISP, ML and Haskell.

3. *Rule-based:* Here actions are predicated on certain conditions in the form of logical expressions. Prolog is an important language in this category.

4. *Object-Oriented:* These can be considered an extension of imperative languages, where primary entities are classes and objects, and where every object has a class which defines its data and its behaviour. Complex objects can be defined from simple objects by allowing objects to inherit properties of other objects. Interaction between objects is possible in carefully defined ways. Examples are Smalltalk, C++ and Java.

A coarse classification of the tasks performed by a compiler are *analysis* of the source program being compiled and *synthesis* of a target program. The analysis part breaks up a source program into its syntactic components and embeds them into a hierarchical grammatical structure. This process is termed parsing. If the analysis detects that the program is syntactically incorrect or

A coarse classification of the tasks performed by a compiler are *analysis* of the source program being compiled and *synthesis* of a target program.

Almost all modern compilers are *syntax directed*. This means that the compilation process is governed by the syntax of the programming language. The context-free grammar formalism originated with Noam Chomsky in the mid fifties.

semantically unsound, then it provides error messages to the user at the appropriate points. It also constructs an intermediate form of the program, and collects information about various entities in the program and stores it in a data structure called a *symbol table*. The intermediate form and the symbol table are passed on to the synthesis part, which constructs the desired target program from the intermediate program and the symbol table. The analysis part is often called the *frontend* and the synthesis part the *backend*.

Almost all modern compilers are *syntax directed*. This means that the compilation process is governed by the syntax of the programming language. The context-free grammar formalism originated with Noam Chomsky in the mid fifties. The formalism was used by Backus, in the formalization of the syntax of FORTRAN, and by Peter Naur in the formalization of Algol 60 in 1960. Centuries ago, between 400 BC and 200 BC the ancient Indian scholar Panini had devised an equivalent syntactic notation to specify the rules of Sanskrit grammar.

Since the syntax is supposed to direct the parsing and the translation of the language, *ambiguity* of the grammar poses a problem. Ambiguity means that there is more than one way to parse a sentence of the language and therefore more than one translation. The diferent phases of the compiler evolved, as the descriptive limitations of each formalism became evident. We will elaborate on this in the following paragraphs.

The structure of most modern compilers can be described by means of the block diagram displayed in *Figure* 1. We will briefly describe the function of each block.

2. Lexical Analysis

The lexical analyzer views the input program as a stream of characters. It scans the program character by charac-

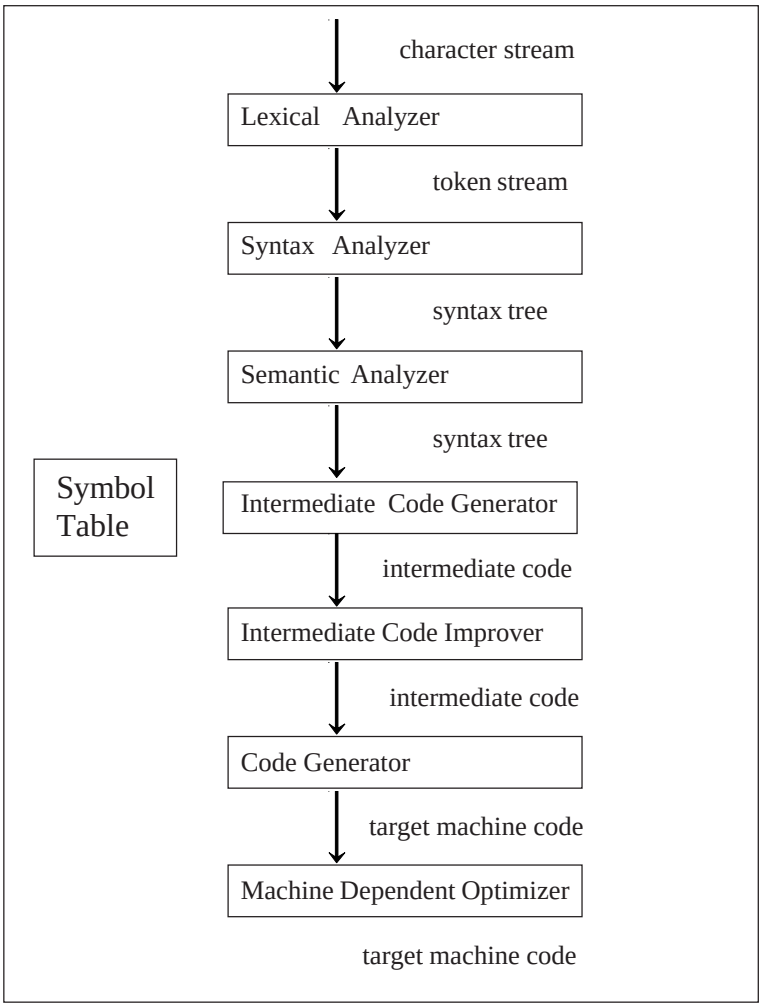


Figure 1. Phases of a compiler.

ter, and produces as output a sequence of *tokens*. Usually tokens are passed one at a time to the parser and represent entities like identifiers, constants, keywords and so forth. Tokens are described by *regular expressions*. For example, if **letter** represents the set of all letters and **digit** the set of all digits, an example of a regular expression for an identifier is **letter(letter|digit)***. The vertical bar stands for union, the parentheses are used to group subexpressions and the star means 0 or more instances of the entity within its scope. Thus the regular expression above describes the set of all identifiers made up of letters and digits that begin with a

The lexical analyzer views the input program as a stream of characters. It scans the program character by character, and produces as output a sequence of *tokens*.

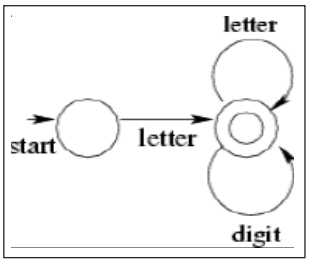


Figure 2. A DFA for identifiers.

letter, which can be followed by 0 or more instances of a letter or digit. Some tokens also carry a lexical value – for example a token representing a constant may also carry a lexical value which is the *value* of that constant. Regular expressions were developed by S C Kleene in the 1950's, and in 1960 R McNaughton and H Yamada devised an algorithm to convert them into *recognizers* called *deterministic finite automata* (DFA). *Figure* 2 is a recognizer for the regular expression for identifiers above. Each circle represents a state. The *start* state is tagged with an arrow and a double circle represents a final state. Each labeled edge from one circle to the other represents a *transition* on the symbol labeling the edge. A string in the *language* represented by the regular expression is said to be recognized by the device whenever it spells out a path consisting of a sequence of labels on edges from the start to a final state.

For compilers, this first phase functionally separates out the task of grouping entities all of which are viewed uniformly by the parser (for example the parser need not distinguish one identifier from another when trying to check the syntactic structure of a sentence).

The theory of *formal languages* developed concurrently with the the theory of compilers. This helped identify which patterns were beyond the power of a particular formalism. For example, the recognition of strings that allowed an arbitrary number of balanced parentheses, like (a), ((a)),(((a))), was proved to be beyond the power of a finite state automaton. A lexical analyzer is then just a simulator for a DFA which may perform a set of tasks as it recognizes each token in the input stream. For example, when it sees a constant in the input it may convert the string of numerals representing the constant into an internal representation and store it somewhere where it can be accessed when needed later.

The recognition of strings that allowed an arbitrary number of balanced parentheses, like (a), ((a)),(((a))), was proved to be beyond the power of a finite state automaton.

Within a few years it became obvious that instead of

writing lexical analyzers by hand they could be generated from regular expression specifications. The first tool LEX was written by Mike Lesk in 1975. All that a user needed to do was to specify in the appropriate format, a list of regular expressions for each token of the language, and this would be processed by the *lexical analyzer generator* to produce the lexical analyzer. This philosophy would be followed in subsequent stages of the compiler as well. Once the input stream is partitioned into tokens the next step is to check whether the input stream of tokens conforms to the predefined syntax of the language.

All that a user needed to do was to specify in the appropriate format, a list of regular expressions for each token of the language, and this would be processed by the *lexical analyzer generator* to produce the lexical analyzer.

3. Syntax Analysis

When checking an English language sentence for correct syntax, it is necessary to identify the nouns, verbs, adjectives and so forth (these are the tokens in this context), and to verify that they are put together in sentences that follow the grammatical rules of English. *Context-free grammars* are used to specify the syntax of *formal languages* which are far more restricted in scope than natural languages. Context-free grammars have a *start symbol*, and entities called *terminals, nonterminals,* and *rules*. The terminals are the tokens from the lexical analysis phase, and the nonterminals are used to represent syntactic categories or groups of strings. For example, a syntactic category could correspond to the set of all possible **if-then-else** statements of the language. Nonterminals impose a hierarchical structure on the language that is the key to syntax analysis and translation. The choice of these syntactic categories is up to the designer of the grammar. The start symbol is a special nonterminal which always begins a derivation sequence of the grammar, which we will now illustrate.

Nonterminals impose a hierarchical structure on the language that is the key to syntax analysis and translation. The choice of these syntactic categories is up to the designer of the grammar.

The example below shows a snippet of a grammar for arithmetic expressions with operators +(addition) and $*$ (multiplication). The set of terminals is $\{id, +, *, (,)\}$.

Each string in the derivation sequence is called a *sentential form* and the string at the end which consists of terminals only, is called a *sentence*.

The symbol id represents the token returned for an identifier, the $+$ and $*$ represents the tokens for the addition and multiplication operators respectively. The symbols $(,)$ are the tokens for the left and right parentheses. The nonterminals are E, T, F and the start symbol is E.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

A derivation sequence always begins with the start symbol, in this case E and at each step replaces some nonterminal in the current string of symbols with the right side of a rule having that nonterminal on the left. Let us use the symbol $\Rightarrow$ to mean "derives in one step". Then we have the following derivation sequence for the string $id + id * (id + id)$. The nonterminal underlined in each string is the one which will be replaced in the following step.

$\underline{E} \Rightarrow E + \underline{T} \Rightarrow E + T * \underline{F} \Rightarrow E + T * (\underline{E}) \Rightarrow E + T * (E + \underline{T}) \Rightarrow E + T * (E + \underline{F}) \Rightarrow E + T * (\underline{E} + id) \Rightarrow E + T * (\underline{T} + id) \Rightarrow E + T * (\underline{F} + id) \Rightarrow E + \underline{T} * (id + id) \Rightarrow E + \underline{F} * (id + id) \Rightarrow \underline{E} + id * (id + id) \Rightarrow \underline{T} + id * (id + id) \Rightarrow \underline{F} + id * (id + id) \Rightarrow id + id * (id + id)$

Each string in the derivation sequence is called a *sentential form* and the string at the end which consists of terminals only, is called a *sentence*. Note that the derivation strategy above, replaces the *rightmost* nonterminal in a sentential form at each step, and is hence called a rightmost derivation sequence. One could similarly have a *leftmost* derivation sequence for the same input string as illustrated below:

$\underline{E} \Rightarrow \underline{E} + T \Rightarrow id + \underline{T} \Rightarrow id + \underline{T} * F \Rightarrow id + \underline{F} * F \Rightarrow id + id * \underline{F} \Rightarrow id + id * (\underline{E}) \Rightarrow id + id * (\underline{E} + T) \Rightarrow$

$id+id*(\underline{T}+T) \Rightarrow id+id*(\underline{F}+T) \Rightarrow id+id*(id+\underline{T}) \Rightarrow id+id*(id+\underline{F}) \Rightarrow id+id*(id+id)$

In fact one could have several derivation sequences for the same string. Grammars that are used in practice, usually have a unique rightmost or leftmost derivation sequence for each syntactically well-formed sentence of the grammar. For such grammars, there are efficient parsers that parse in time linear in the size of the input sentence.

The parse sequence implicitly constructs a parse tree for the sentence. For a leftmost derivation sequence the tree is constructed in a *top-down* manner and the corresponding parsers are called top-down parsers. For rightmost derivation sequences, the construction of the parse tree is *bottom-up* and mimics a rightmost derivation sequence in reverse. *Figure* 3 displays the parse tree (there is only one) for the sentence of the example above.

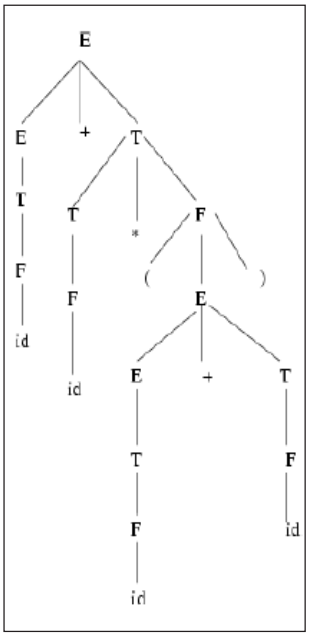
Each internal node of the parse tree is a nonterminal. The labels on the children of a node corresponding to a nonterminal say A, spell out from left to right, a right hand side of a production for A. The labels on the leaves of a parse tree read from left to right spell out a sentence of the grammar.

When actually generating code, the process ensures that code for all subtrees of a tree is generated before the code for the root of the tree and therefore precedence for $*$ over $+$, and () over everything else is automatically taken care of by the way the grammar is written. This is no accident, as the grammar is deliberately designed that way. If, for example, we had specified the grammar as

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow (E)$
4. $E \rightarrow id$

Each internal node of the parse tree is a nonterminal. The labels on the children of a node corresponding to a nonterminal say *A*, spell out from left to right, a right hand side of a production for *A*.

Figure 3. The parse tree for the sentence id + id *(id +id).



In addition to being equipped with a finite state automaton, a pushdown automaton is also equipped with a *stack* which is a device into which information can be written, and from which the information most recently written is read out first.

then even though it generates the same set of sentences as the previous grammar, it is ambiguous as there is more than one parse tree for a sentence like $id + id * id$, (the interested reader can check this) and one of them would generate code for the addition before the multiplication.

The recognizer for a context-free language is a *pushdown automaton*, a precise description of which is beyond the scope of this article. In addition to being equipped with a finite state automaton, a pushdown automaton is also equipped with a *stack* which is a device into which information can be written, and from which the information most recently written is read out first. For example, if a,b,c, are written into the stack in that order, they are read out in the sequence c,b,a. It was soon recognized that a program that simulated a pushdown automaton could be automatically be generated from a context-free grammar and tools for generating parsers from grammars became available. One of the first such tools was YACC (Yet Another Compiler Compiler) designed in 1975. In addition to generating the syntax analyzer, these tools were also capable of spotting syntactic ambiguities and other problems that might have slipped through the early design phase of the language. Donald Knuth provided the basis for the theory that underlie parser generators like YACC.

Formal language theorists soon observed that context-free grammars were not sufficient to describe all the constraints that a programming language had to satisfy. For example, a context-free grammar was not powerful enough to specify the constraint that any identifier in a program had to be declared before it was used, or that the number of parameters in a procedure *call* had to match the number in the *declaration* of the procedure. In order to show that a particular language construct *can* be handled by a context-free grammar, it is enough to display a context-free grammar for it. On the other

hand, to show that some construct is beyond the scope of a context-free grammar, requires a proof that a certain property that is *always* true of any context-free language cannot possibly hold for the construct. Herein lies the power of formal models. Not only do they enable the construction of recognizers, but they also provide techniques to define their own limitations.

Error *recovery* also needed to be handled by the parser, that is the ability to go on with the parse even after a syntactic error was detected, with insertions of error diagnostics at the appropriate places. The ingenious techniques designed, actually mimic the actions of a human who is trying to understand a grammatically incorrect sentence. Error recovery schemes usually analyze the grammar to find 'synchronizing' symbols for each nonterminal; thus if the phrase encapsulated by the nonterminal is syntactically incorrect, the parser can recover when the synchronizing symbol is seen in the input stream, by assuming that the parsing of that phrase was over, after putting out a diagnostic message indicating that *that* particular phase is in error. This scheme can easily be integrated into the machinery of a parser. What is more, a parser augmented with error recovery can actually be *generated* just from the grammar specification alone.

By the late seventies there were excellent tools that could generate both top-down and bottom-up parsers with error recovery from grammar specifications. A user with access to these tools just needed to specify the grammar in a specified format and the tool would check the grammar, and if there were no violations of constraints, output a parser with error recovery for the grammar. As mentioned earlier there are tasks that are beyond the power of recognizers for context-free languages and these are performed by the semantic analysis phase described next.

In order to show that a particular language construct *can* be handled by a context free grammar, it is enough to display a context free grammar for it. On the other hand, to show that some construct is beyond the scope of a context free grammar, requires a proof that a certain property that is *always* true of any context free language cannot possibly hold for the construct. Herein lies the power of formal models.

Attribute grammars as the name suggests, associate attributes with syntactic entities, and formulate rules whe- reby these attributes may be computed.

4. Semantic Analysis

The semantic analysis of a sentence typically involves tasks like checking if identifiers have been declared before use, checking that the types of operands match and so forth. For example a syntax checker would pass an expression where two identifiers are multiplied without worrying about what they represented, but a semantic analyzer would tag as erroneous, an expression where an identifier which represents a string was multiplied by one representing a floating point number. Tasks like these are usually performed in a syntax directed manner, using the syntax to inductively define rules that enforce these checks. A formalism used here is that of *attribute grammars* first defined by Donald Knuth in 1968.

Attribute grammars as the name suggests, associate attributes with syntactic entities, and formulate rules whereby these attributes may be computed. Attributes may be passed *down* from ancestors to descendants in the parse tree, or passed *up* from descendants to ancestors. The former are termed *inherited attributes* and the latter *synthesized attributes*. In more complex specifications, attributes may be passed among siblings in the parse tree. A problem that might arise here is that of *circularity*. For example there may be a parse tree for which there is no order in which the attributes may be computed because the rules are written in a circular fashion. There are tools that check this and which can generate a proper sequence of attribute computations if an ordering of computations is possible.

Once a program is syntactically and semantically checked it is cleared for the generation of intermediate code.

5. Intermediate Code Generation and Code Improvement

Typically compilers first translate source code into an intermediate representation(IR), mainly to implement

code improvement techniques–sometimes referred to as optimizations. An intermediate representation in terms of simple statements, or in the form of abstract syntax trees, permits an easy analysis to uncover potential machine independent code improvement transformations, and makes translation to target code easy. The process of syntax directed translation itself, because it is so general, introduces some redundant computatons, and perhaps statements that are unreachable in any execution of the program. These can be removed by analyzing the code after constructing a suitable model of the program.

An intermediate representation in terms of simple statements, or in the form of abstract syntax trees, permits an easy analysis to uncover potential machine independent code improvement transformations, and makes translation to target code easy.

A casual programmer might have put statements that are actually loop invariant within loops that may be executed millions of times thereby incurring unecessary execution time overheads. Also addressing of arrays when performed in low level code may result in expressions being recomputed when they have already been computed once and may be reused. The techniques for gathering information to perform these kind of tasks are broadly called *data-flow analyses*. These are algorithms to gather information about a program.

The fundamental treatise on techniques for loop optimization is by Frances Allen in 1969. A systematic study of data-flow analysis was initiated by a pair of papers by Frances Allen and J Cocke in 1970. The idea here is to first identify *basic blocks* and then represent the program as a graph called a *control flow graph*. Each node represents a basic block, which is a sequence of instructions that is always entered at the first instruction and exited at the last instruction with no intermediate entries or exits. There is an edge from block A to block B if control can flow from block A to B during execution. The analysis is now carried out using an iterative traversal of this graph with each node associated with a *function* that represents the effect of the instruction sequence for the basic block on the information we seek to gather. The information we seek to gather, for example, might

Early versions of FORTRAN had no recursion which meant that a procedure or function could not call itself either directly or indirectly. This allowed the use of a "static" memory allocation scheme, where all the memory allocation for variables could be done before the program was actually executed.

be all the *definitions* of a variable (of the form $v = e$ where v is a variable and e an expression) that reach a particular point in a program. There is an elegant theoretical framework developed by G Kildall in the early seventies and in 1977, J B Kam and J D Ullman set up the conditions under which a dataflow analysis algorithm will rapidly converge.

Once the dataflow analysis is over then code improvement techniques can be applied. These may move statements out of loops, eliminate intermediate code that is never reached and remove unecessary copying instructions. Once possibilities for code improvement have been explored the 'target independent' part of the compilation is over and the compiler 'backend' takes over. This part is much more difficult to automate.

Before we go to the compiler backend, we will say a little about how storage is organized at execution time.

6. Runtime Storage Administration

Early versions of FORTRAN had no recursion, which meant that a procedure or function could not call itself either directly or indirectly. This allowed the use of a "static" memory allocation scheme, where all the memory allocation for variables could be done before the program was actually executed. However the inclusion of block structure and recursion in Algol 60 required the use of stacks as different invocations of a procedure could simultaneously be active and could therefore not share the same space. Each procedure body in the source program is represented by a section of instructions in the target program. When a procedure has been called and before it returns, various entities represent the state of the current activation of the procedure. There is firstly an *instruction pointer*, that indicates a position in the code, then there is a *data frame* that holds all the values of parameters and local data objects of the activa-

tion and lastly, the *environment* consisting of those data frames which the current activation may legally access. When execution of the procedure body reaches a point at which the current procedure calls another procedure, or calls itself recursively, the current activation is halted, all the entities that define its state are saved and relevant information is set up for the new procedure. In a recursive language, the acquisition of space for the new procedure is usually from a stack.

7. Code Generation

This phase actually generates code for the target machine from the intermediate form. With architectures becoming more and more sophisticated and having hardware features like caches, multiple functional units, special purpose registers and so forth, that cannot be easily be described in a formal specification, generating tools for this phase of the compiler is much more difficult. Typically there are three major phases in code generation – *instruction selection* which selects a sequence of target machine operations that implement the operations in the intermediate code, *register allocation* which decides which variables of the program should reside in registers at each point in the program, and *instruction scheduling* which decides an order in which the instructions should execute. Instruction selection becomes important when there are several instruction sequences of the target machine that can implement the same sequence of intermediate code instructions. Thus some criterion for selecting the best sequence has to be decided on. For instance, the total execution time of the sequence can be used as a criterion.

Typically there are three major phases in code generation – *instruction selection* which selects a sequence of target machine operations that implement the operations in the intermediate code, *register allocation* which decides which variables of the program should reside in registers at each point in the program, and *instruction scheduling* which decides an order in which the instructions should execute.

Registers are physical devices in the machine which may hold intermediate values. Operands residing in registers can be accessed faster than if they reside in memory. Since these are limited in number and this number is much smaller than the number of potential candidates,

Many processors especially those used on embedded devices leave the dependence checking between consecutive instructions to the compiler which is responsible for inserting no-op instructions in the code whenever necessary.

the problem is to decide which operands should reside in registers. Clearly if the ranges of instructions for which two operands are required do not intersect, they can both share the same register. R Sethi showed in 1975 that the register allocation problem was unlikely to have an optimal solution that could be computed in polynomial time. Gregory Chaitin ingeniously formulated this problem as a graph colouring problem in 1982 and it is this formulation that is most commonly used in heuristic algorithms for tackling the problem even today.

Many modern processors have pipelines. With a pipeline, a new instruction can be fetched every clock cycle while the preceding instructions are still going through the pipeline. Thus even if the processor can issue only one operation per clock, several instructions can be in execution in different stages of the pipeline at the same time. Many processors especially those used on embedded devices leave the dependence checking between consecutive instructions to the compiler which is responsible for inserting no-op instructions in the code whenever necessary. If several instructions can be issued per clock cycle then in machines known as Very Long Instruction Word (VLIW) machines, it is the responsibility of the compiler to schedule operations that can execute simultaneously and delay those whose operands are not yet ready. Scheduling instructions so that the schedule consumes the minimum number of cycles is another instance of a problem in compilers for which a polynomial time algorithm is unlikely to exist, so usually heuristic algorithms are used in practice.

8. Retargetable Compilers

Systematic approaches to code generation have facilitated the building of *retargetable compilers*. A retargetable compiler typically has the machinery to handle the instruction set of any machine provided the appropriate specification for the machine is written. Most of

the burden for such portable compilers is placed on the instruction selector. *Figure* 4 shows how such a system works. The machine is specified in terms of tree patterns which indicate how target code is generated for intermediate code constructs. A *code generator generator* (CGG) preprocesses these patterns into tables and a driver uses these tables to guide the selection of target instructions for each intermediate code operaton. The driver and the tables constitute the code generator. Since preprocessing is done offline, i.e. at compiler generation time, sophisticated algorithms can be used to generate the CGG as it is a one time operation for a given target. Retargeting the compiler to another machine would require running the CGG on a new specification.

A retargetable compiler typically has the machinery to handle the instruction set of any machine provided the appropriate specification for the machine is written.

9. The GCC Collection

Without a doubt, the most popular compiler collection is GCC, the Gnu Compiler Collection. GCC was started by Richard Stallman in 1984 in order to promote freedom and cooperation among computer users and programmers. The GNU Compiler Collection includes frontends for C, C++, FORTRAN and Java. Most importantly it is *free* software distributed by the *Free Software Foundation*. It is the standard compiler for the free software Unix operating system and Apple MAC OS X. The original version which handled only C programs was released in 1987 and for C++ at the end of that year.

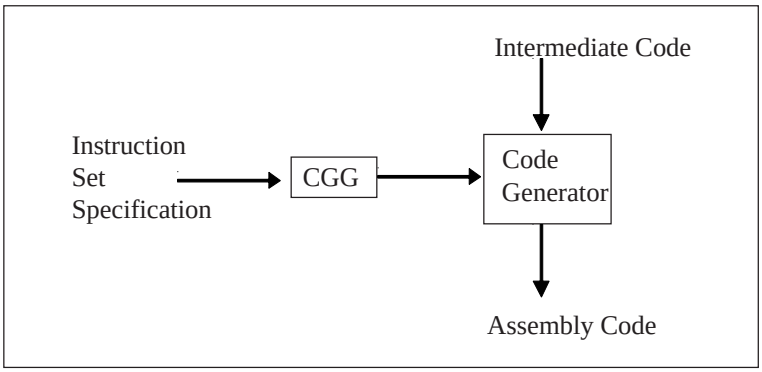


Figure 4. Retargetable Code Generation.

Frontends were later developed for languages like FORTRAN, Java, Ada and others. It is maintained by a group of programmers around the world and has been ported to more processors and operating systems than any other compiler.

10. Some New Challenges

Encouraged by the success of tools that generate parts of the compiler automatically, compiler designers are trying to push the technology to solve problems that were typically in the domain of hardware engineers. Around 2001 HP Labs started a new project called PICO (Program In Chip Out). Embedded PICO Express takes algorithm descriptions expressed in terms of sequences of nested loops and maps this into a highly optimized pipeline processor array architecture. A powerful compiler analysis exploits parallelism at multiple levels to find the best implementation. It is claimed that this technology can reduce Register Transfer Level (RTL)[1] creation from months to days. Thus this project will let users create C-language algorithms for applications such as MPEG or MP3 decoders. It will evaluate alternatives and generate synthesizable register-transfer-level code. Hardware designers can use a VLIW processor, a configurable pipeline of processor arrays or both.

Compiling for embedded systems is another area of active research. In the embedded systems area, compiler technology is faced with new challenges, including code generation for special architectural features, a highly flexible degree of retargetability, memory aware code generation, that can exploit the structure of the memory of the embedded system, and the generation of code that satisfies real-time and performance constraints.

[1] RTL is a special low level intermediate representation.

Suggested Reading

[1] A V Aho, M S Lam, Ravi Sethi, J D Ullman, *Compilers–Principles, Techniques and Tools*, Second Edition, Addison Wesley, 2007.

[2] See articles on Grace Murray Hopper, *Resonance*, Vol.6, No.2, 2001.

Address for Correspondence
Priti Shankar
CSA
Indian Institute of Science
Bangalore 560 012, India.
Email:priti@csa.iisc.ernet.in;
priti@aditya.csa.iisc.ernet.in