MACHINE-INDEPENDENT CODE GENERATORS SPECIFIED

IN PREDICATE LOGIC

by

LIU CHUN NIAN

Dissertation submitted to the Division of Computer Science, The Norwegian Institute of Technology, in partial fulfillment of the requirement for the degree of Doctor of Engineering.

DIVISION OF COMPUTER SCIENCE
THE NORWEGIAN INSTITUTE OF TECHNOLOGY
UNIVERSITY OF TRONDHEIM
ABSTRACT

The specification and prototype implementation of a Code Generator Writing System (CGWS) based on the predicate logic language PROLOG is presented. The CGWS accepts PROLOG descriptions of the target machine architecture and the compiler implementation decisions and creates a code generator that generates assembly code for the target machine from IL programs. The IL is a high level intermediate language and the created code generator covers very large part of the synthesis phase of compilation: storage allocation, data accessing, instruction selection, register allocation and procedure handling.

This work tries to combine two useful tools together to facilitate the code generator construction: the concept of CGWSs and the logic programming. The major contributions provided by this thesis are:


2. To do more optimization at code generator generation time, resulting in a smaller, faster code generator.

3. To include all the machine dependent issues of the code synthesis phase of the compilation into a single package, making the interfaces between compiler modules much easier.

4. A logic programming language has been introduced into the area of CGWSs for the first time.

A prototype CGWS hosted on a ND-100 computer using NTH PROLOG has produced code generators for PDP-11, ND-100 and MOTOROLA-68000.
ACKNOWLEDGMENT

First, I wish to express my deepest gratitude to my supervisor, Prof. Kristen Rekdal. He has directed my study since I came to NTH. For this dissertation, he has given the motivation of the topic, long hours of discussion and thoughtful criticism of earlier drafts.

Tore Amble escorted my first steps in PROLOG. I thank him not only for the indispensible research tool, the NTH PROLOG interpreter he has developed, but for the countless hours he spent with me in discussion of various ideas, in reviewing earlier drafts and making suggestive comments.

I got a lot of support from the faculty in the Division of Computer Science, NTH. Specially, the help and encouragement of Prof. Arne Sølvberg is appreciated. The researchers in The Group of languages and translators at RUNIT (The Computing Center at the University of Trondheim) have helped me in various ways during the last few years. I have learned a lot from discussions with them.

In addition to the individuals mentioned above, the very favorable working condition provided by RUNIT and the scholarship provided by NORAD, The Norwegian Agency of International Development, are gratefully acknowledged.
The thesis with the title **MACHINE-INDEPENDENT CODE GENERATORS SPECIFIED IN PREDICATE LOGIC**

by **Liu Chun nian**

for the degree of **Doctor of Engineering**

has been approved by the censor committee consisting of

**Professor Knut S. Skog (sign.)**

**Professor Stein-Åke Tänlund (sign.)**, and

**Professor Kristen Rekdal (sign.)**

Date: **December 1983**
# TABLE OF CONTENTS

1. INTRODUCTION  
1.1 The motivation  .................................. 1  
1.2 The overview of this work  ....................... 3  
  1.2.1 Design decisions on various aspects  ...... 4  
  1.2.2 Research goals  .................................. 6  
  1.2.3 The thesis organization  ....................... 8  
1.3 The related work  
  1.3.1 A brief historical review  ................... 9  
  1.3.2 The general tree pattern matching and the heuristic search 10  
  1.3.3 The completeness of retargetable code generators 12  

2. FIRST ORDER PREDICATE LOGIC AND PROLOG  
  2.1 First order predicate logic  ............... 14  
    2.1.1 Syntax  ...................................... 14  
    2.1.2 Semantics  ................................... 15  
  2.2 Logic programming  
    2.2.1 Inference viewed as computation  .......... 17  
    2.2.2 The clause form of predicate logic  ...... 17  
    2.2.3 The syntax of PROLOG - Horn clauses  .... 18  
    2.2.4 The semantics of PROLOG - unification and backtracking 22  
  2.3 Running a PROLOG program  .................... 23  
  2.4 Nonstandard features in NTH PROLOG  .... 24  

3. TARGET MACHINE DESCRIPTION IN PROLOG  
  3.1 The design principle  .......................... 27  
  3.2 The implementation decisions and the hardware support 28  
    3.2.1 The memory and the data type mapping  .... 29  
    3.2.2 The (user) allocatable registers  ........ 30  
    3.2.3 Dedicated registers and run-time conventions 32  
    3.2.3.1 Keeping track of the positions of activation records 33  
    3.2.3.2 The layout of activation records  .... 34  
  3.3 Machine operations  
    3.3.1 The access modes  ............................ 37  
    3.3.2 The operand classes  ........................ 37  
    3.3.3 The instruction set  .......................... 39  
  3.4 The guidance to the pattern matching  ........ 41
4. CODE GENERATOR GENERATION IN PROLOG

4.1 Introduction

4.2 Code derivation
4.2.1 Theoretical background
4.2.2 Code derivation and AI Production Systems (AIPSs)
4.2.2.1 The Production Systems in AI
4.2.2.2 A backtracking algorithm for the state-space search
4.2.2.3 Code derivation formalized as an AIPS
4.2.3 Code derivation described in PROLOG
4.2.3.1 The state space
4.2.3.2 Production rules
4.2.3.2.1 The choice of evaluation order (EVO rules)
4.2.3.2.2 Instructions (INS rules)
4.2.3.2.3 Fetch/store subtargeting (SUB rules)
4.2.3.2.4 Flow-control decompositions (DEC rules)
4.2.3.2.5 The arithmetic and Boolean laws (AXI rules)
4.2.3.2.6 Common subexpression merging (CSE rules)
4.2.3.3 The control regime
4.2.3.3.1 The cost estimation
4.2.3.3.2 The search limits
4.2.3.3.3 The rule ordering
4.2.3.3.4 Other heuristics

4.2.4 The experimental results

4.3 Pattern selection and MT building
4.3.1 A pattern selection scheme
4.3.2 The current implementation of MT building

5. CODE GENERATION

5.1 The Intermediate Language (IL)
5.1.1 The form of IL programs
5.1.2 The level of the IL
5.1.3 The flexibility of the IL
5.1.4 An example

5.2 Machine-independent storage allocation

5.3 Data accessing

5.4 Instruction selection

5.5 Register allocation

5.6 Procedure-linkage and parameter-passing

5.7 An example of code generation
6. RESULTS, CONTRIBUTIONS AND FUTURE RESEARCH

6.1 Results

6.2 The major contributions

6.3 Future research

REFERENCES

Appendix 1. The definition of the IL
Appendix 2. The Pattern File (PF)
Appendix 3. The TMDs for PDP-11 and ND-100
Appendix 4. Extracts of the MTs for PDP-11 and ND-100
Appendix 5. Examples of code derivation for complex IL patterns
Appendix 6. The CG in PROLOG
1. INTRODUCTION

1.1 The motivation

The subject of this thesis is machine-independent code generators in compilers.

A code generator is called machine-independent (or retargetable) if it can easily be adapted to changes in the target machine architecture and/or (in the broad sense) the compiler implementation decisions. In compiler construction, independency of both source language and target machine is highly desirable. Compiler-compilers (or compiler writing systems) have been investigated for many years to reduce the effort involved in compiler construction. The idea is that, by providing tools for automatic generation of various compiler modules, one can construct a compiler for any new source-language / target-machine pair routinely, cheaply and reliably.

The most successful aspects of this attempt has been parser generators and lexical analyzer generators. For example, /Johnson 1978/ reports that YACC (a parser generator) and LEX (a lexical analyzer generator) on the UNIX system have proved to be very satisfactory tools in the construction of the Portable C Compiler. At RUNIT, the GRANADA system (/Wessel & Rekdal 1976/) has successfully generated the CHILL syntax analyzer in the Nordic CHILL Compiler (/Rekdal et al. 1979/).

Unfortunately, no such success has been seen in any production compiler for the back-end of the compilation process. Any one developing a production compiler recognizes that syntax analysis is not a large portion of the compiler as far as construction effort is concerned. The lack of success in automating other phases makes compiler writing systems much less attractive in the commercial world than they are in a research environment. In particular, rapid development of new machine architectures makes rapid construction of compilers more and more desirable. So it is not surprising that in the
past few years there has been an increasing interest in automating code generation and optimization. The general name for the developed systems in this area is Code Generator Writing Systems (CGWSs), and it is fair to say that most of the research activity reported so far has been in the U.S.A.

As code generation and optimization phases of compilers normally apply a great deal of heuristics, techniques from Artificial Intelligence (AI), among others, have begun to find their way into the research area of CGWSs. Some authors introduce AI-techniques to realize (part of) CGWSs (e.g. /Cattell 1978/, /Kozlak 1981/). Others even use AI-languages like LISP to specify and implement their experimental CGWSs (e.g. /Newcomer 1975/, /Fraser 1977a,b/, /Kessler 1981/).

In this same period, logic programming and its languages (e.g. PROLOG) have been gaining an enthusiastic band of users, mainly in Europe and Japan. There are now many implementations of PROLOG, including the one developed at NTH (The Norwegian Institute of Technology) by B.M. Johnsen and T. Amble (/Amble 1983/). There are also many promising results in various PROLOG application areas including language implementation (/Warren 1980/). But as far as my knowledge goes, no attempt at using logic programming in CGWSs has been reported before.

In a CGWS, the key issue is the pattern matching. This fits directly the basic, built-in mechanism of PROLOG - the unification process. (As the matter of fact, unification is a stronger concept than pattern matching). In light of this observation, I believe CGWSs could be another promising application area of PROLOG.

This work is based on a study of previous CGWSs. The study identifies the areas which are important but have gained little attention before, and proposes solutions to them. To demonstrate the feasibility of the ideas proposed, a prototype CGWS has been implemented. To test the applicability of PROLOG to CGWSs, the whole system is specified in PROLOG. The NTH PROLOG interpreter supports this specification, giving an experimental implementation.
1.2 The overview of this work

Figure 1.1. gives an overview of the problems tackled in this work and the top-level structure of the prototype CGWS:

Figure 1.1. The CGWS model
The way chosen here to present the overview of our CGWS is in two steps. First in section 1.2.1, we identify the main components and give our design decisions. Then in section 1.2.2, we state the research goals of this work. Section 1.2.3 is about the thesis organization.

1.2.1 Design decisions on various aspects

From Figure 1.1, we can see that this CGWS consists of

Two action modules: The Code Generator Generator (CGG),
The Code Generator (CG);
Three data modules: The Target Machine Description (TMD),
The Mapping Table (MT);
The IL Pattern File (PF);
Two interfaces: The Intermediate Language (IL),
The Symbolic Assembly Language (SAL).

For each of them, we state its purpose and our design decision as follows.

1. The Target Machine Description (TMD)

The TMD describes the various hardware components of the target machine as well as the compiler implementation decisions (such as the run-time conventions). In short, it describes the compiler writer's virtual machine for which we are going to generate code. The CGG uses it to produce the MT, which in turn will be used by the CG.

The description is written by hand from the target machine manual, etc. It is in the form of PROLOG clauses, each clause asserts a fact about the target machine.

2. The Code Generator Generator (CGG)

The CGG reads IL patterns (parameterized IL expressions) from the IL Pattern File (PF), consulting with the TMD to derive the "best" target code sequences for them. From each pattern and the corresponding code
sequence it builds a template and sends it to the MT.

The CGG takes the general tree pattern matching approach to the code derivation. The IL patterns can be as complex as basic blocks. A depth-first heuristic search algorithm derives the "best" code sequence for each pattern. The CGG is a PROLOG program, running (ideally) once for each target machine.

3. The Mapping Table (MT)

The MT defines the mapping from IL patterns to target machine code. First, the CGG "copies" the TMD to the MT, modifying the instruction description into templates (because each instruction does implement a simple IL pattern and should be regarded as a template). Then the CGG adds new templates built from more complex IL patterns. Thus the MT also consists of PROLOG clauses.

The MT contains all machine-dependent and implementation-dependent information needed by the CG.

4. The Code Generator (CG)

The CG is a skeleton code generator. It receives IL programs as input and produces as output symbolic assembly code for the target machine.

The CG covers a very large part of the synthesis phase of compilation: storage allocation, data accessing, instruction selection, register allocation, procedure-linkage and parameter-passing are all performed in a machine-independent way, driven by the MT.

The CG is also a PROLOG program, running once for each IL program.

5. The Intermediate Language (IL)

A source program is first translated into an intermediate language form. This translation is not included in this work. The CG generates target code from this IL form. Our IL is on a high level with respect to the target languages, as a consequence of the completeness of the CG. This is a very desirable feature for a CGWS.

Syntactically, an IL program consists of a sequence of PROLOG terms,
which can be read by the CG one after another.

6. The IL Pattern File (PF)

To build the MT from the TMD, we must decide which IL patterns should be sent to the CGG as input. The chosen IL patterns reside on the IL Pattern File (PF), from which the CGG reads the patterns one after another. Section 4.3 gives the criteria for choosing the IL patterns, the PF building methods and how to use it in the CGG.

7. The Symbolic Assembly Language (SAL)

The SAL code is chosen as the output of the CG instead of binary machine code, partly because of the time constraints of this work, partly because of its readability for human beings. The methods used in this CGWS, however, are in no way limited to this form of output. E.g. binary relocatable code could have been generated as well.

1.2.2 Research goals

The CGWS structure shown in Figure 1.1 and explained briefly in the previous subsection fits the general model of CGWSs. In this subsection we state the research goals of this work, i.e. what are the new problems to tackle, and what are the new ideas and new methods to test in this CGWS.

1. To do as much optimization as possible at code generator generation time. The result is a simple, fast skeleton code generator which can produce quite good code without expensive optimization algorithms. In other word, we try to gain both efficiency of the CG and good quality of the target code.

Our approach to this goal is to include any necessary IL patterns in the MT. No matter how complex they are, the CGG must derive the "best" code sequences for them (the quotation marks emphasize the fact: optimal code derivation in its full complexity is an NP-complete
problem, so we can only expect a near-optimal solution within a reasonable search time). Then a code generator using a simple tree pattern matching algorithm can give good code because each successful matching always returns the "best" local code.

This work proposes a depth-first heuristic search algorithm in the CGG to tackle the optimal code derivation problem. Unlike the classical A* algorithm (see /Nilsson 1971/, for example), this algorithm causes no space explosion problem, and the search time is also within reasonable limits.

We also propose the criteria for the IL pattern selection, based on the knowledge of the source language and the source programs.

2. To make a "complete" retargetable code generator. Unlike most of the previous CGWSs, in which the CGs are essentially instruction selectors, this work tries to isolate all machine-dependent issues in a single package and to deal with them in a machine-independent way. The advantage of this approach is obvious: other phases of the compiler need not bother about the target machine, and the interface with the CG would be much easier.

This work gives a model of the compiler writer's virtual machine description, and proposes machine-independent algorithms for various code synthesis tasks using the description.

3. To test the applicability of logic programming and its language PROLOG to CGWSs. All action modules and data modules are specified in PROLOG. Three real machines - PDP-11, NO-100, MOTOROLA-68000 - representing three typical architectures and various compiler implementation decisions are tested in the experimental system. The performance will be discussed in the conclusion chapter (chapter 6).
1.2.3 The thesis organization

Chapter 1 is the introduction. After the motivation and the overview of this work presented above, the rest of this chapter gives a summary of the related work.

Chapter 2 introduces predicate logic and the PROLOG language, with a "mini code generator" to show the working style of PROLOG programs.

Chapter 3 presents the virtual machine description in PROLOG, showing how to specify the various hardware components of the target machine as well as the compiler writer’s decisions.

Chapter 4 describes the CGG, with the emphasis on the proposed algorithm of optimal code derivation, taking evaluation order, target operand selection, and common subexpression merging into account. The IL pattern selection is another important subject in this chapter.

Chapter 5 is devoted to the code generation problem, showing how to tackle various code synthesis tasks in a machine-independent way.

The conclusion chapter summarizes the research, evaluates the prototype CGWS and proposes fields for the future research.

The prototype system in its PROLOG code is presented in appendices.

1.3 The related work

In this section, we give a brief review of the history of CGWSs and a summary of those CGWSs to which the present work has direct relations. A more comprehensive survey of this research area can be found in /Cattell 1977/, /Ganapathi 1981,1982/ and specially in /Lunell 1983/, which devotes more than 300 pages to this topic, providing very systematic information.
1.3.1 A brief historical review

The history of CGWSs (or the research in reducing code generation effort) has been through 5 major evolutions:

1. The two-stage translation scheme.

Since the very early time of the research in reducing code generation effort (in the late 1950s), the two-level translation scheme (by introducing the Intermediate Language IL) has been suggested. With this IL, to implement p programming languages on m machine architectures only p*m translators have to be written instead of p*m. This scheme has been kept thereafter. Current examples of ILs are PASCAL P-code (/Ammann et al. 1977/), SIMULA S-code (/Jensen et al. 1982/) and CHILL/MARY MIIL-code (/Rekdal & Hallsteinsen 1982/, and for MARY language see /Conradi & Holager 1979/), to name a few.

2. Code generation languages.

In the two-stage translation scheme, the IL is regarded as the language of an ideal machine, and the task of code generation is to expand the IL "instructions" into the real target machine instructions. To facilitate this process, since the middle of the 1960s some compiler writing systems provided programmers with specialized languages with built-in machinery to deal with the common details. Examples are /Elson & Rake 1970/ and /Wilcox 1971/. They can be regarded as predecessors to real CGWSs.


The method taken in 2 was indeed a distinct improvement over ad-hoc methods, but the code generation algorithms used there were still intermixed with machine dependent issues. One can not retarget the code generator to new machines without changing the algorithms. /Miller 1971/ made the first attempt at this separating. He describes the target machine operations as macros written in the Object Machine Language (OMML), and the code generation algorithm in macros written in the Machine-Independent Macro Language (MIML). Retargeting his compiler to a new machine means changing OMML macros, while the MIML macros remain unchanged.

One of the major goals of CGWSs is that, we try to discover and use a unified, general model of code generation. The traditional interpretation method (enumerating the cases to be dealt with in code generation) has been not satisfactory in this aspect. On the other hand, if we describe machine operations as tree patterns and regard IL programs as tree instantiations, we can formalize code generation as pattern matching. From this new point of view we can understand better what code generation is, and the code generation construction can be automated much easier. Nowadays most CGWSs take this approach.

5. Automatic analysis of a formal machine description.

In recent years (since the middle of the seventies), the research has concentrated on automatic analysis of formal machine description and code generator generation. That means, given a formal description of the target machine, we can derive automatically a mapping table which specifies the mapping from IL patterns to target machine code sequences. The code generator is machine-independent, driven by the machine dependent mapping table. There have been two major branches in this approach: methods borrowed from AI (such as this work), and methods from language grammar study.

In RUNIT, there is a project (Heensaasen et al. 1983/) to make a retargetable code generator for various target machines utilizing the recent research results in CGWSs.

In the following subsection we give a short summary of 4 CGWSs to which this work is closely related.

1.3.2 The general tree pattern matching and the heuristic search

/Cattell 1978, 1980/ and /Kozlak 1981/ are the two examples in this area. They were taken as the starting points of this work.

The original work of Cattell, putting the emphasis on machine formalization and code generator generation, does not do very much in
code generation itself. The model fits only the CODE phase of the BLISS system (/Wulf et al. 1975/) or the PQCC system (/Cattell et al. 1979/, /Wulf et al. 1980/).

He uses the Tree-based Common Language (TCOL) representation to describe three aspects:

- the machine instruction actions in the Machine Description (MD),
- the IL programs input to the CG,
- the IL patterns input to the CGG.

So the general tree pattern matching can be used both in the CGG and the CG. In the CG, the matching uses simple heuristics to solve the multiple matching problem. In the CGG, the means-ends analysis is employed to derive the optimal code sequences for the IL patterns which have no corresponding instructions in the target machine. The tree equivalence axioms form the basis for this analysis.

While his system is regarded as the first successful attempt at automating code generation in the practical sense, there are a few weak points:

1. The IL patterns are no more complex than single IL operators. The CG, using the mapping table consisting only of these IL patterns cannot guarantee good code quality. (In PQCC, there are many optimization phases before and after the CODE phase).

2. Even if more complex IL patterns were sent to this CGG, the latter could not derive as optimal code sequences for them as it does for the primitive IL patterns. The means-ends analysis used in this CGG fails to keep track of processor state values, and does not recognize common subexpressions. In cases involving complex IL patterns the redundant LOADs and STOREs are inevitable.

3. His code generator is only an instruction selector.

/Kozlak 1981/ tries to improve the point 2. In his system, the IL
patterns can be of potentially any complexity. He uses a breadth-first algorithm to derive the optimal code sequences for these patterns. But his machine model is over-simplified, containing only simple arithmetic and Boolean instructions and three simple access modes: register, memory and constant. As for how to choose IL patterns, and how to use the mapping table produced in the CG, there are only very vague statements.

The present work also takes the general tree pattern matching approach, but the matching is done automatically by the PROLOG unification mechanism. We attack all the three problems mentioned above and deal with real machines. In problem 2 we propose a depth-first algorithm, which can avoid the space explosion problem and fits the PROLOG programming style naturally.

1.3.3 The completeness of re-targetable code generators

If the CG in a CGWS turns to be merely an instruction selector, the CGWS can not claim that it really produces a re-targetable code generator. Because in that case there are other machine-dependent and the implementation-dependent issues which must be dealt with by other compiler phases. /Fraser 1977a,b/ and /Ganapathi 1980/ are two CGWSs producing comparatively complete code generators. Their methods are quite different: The former is an AI knowledge-based system, while the latter takes attribute grammars as the starting point. Here we investigate only one question: how complete the created code generators in these two CGWSs are.

/Fraser 1977a,b/ takes the knowledge based approach, using the Instruction Set Processor notation (ISP) as his machine description. (For the ISP, see for example /Digital 1971/). Here we are only concerning with the completeness of the created code generator. The instruction selection is quite complete, including high level control operations. The procedure linkage is also included. But his IL is in the form of tuples, one operation per tuple. He generates code for the tuples one after another, without inter-tuple optimization, resulting in many redundant LOADs and STOREs. As for the storage allocation,
his system supports only static allocation directly, while the dynamic allocation must be built from primitives such as stack operations in his IL programs.

/Ganapathi 1980/ is an extension of /Glanville 1977/, using attributed grammars to describe the target machine as well as the IL. His system produces code of high quality at least for the examples given in his thesis. He tries also to tackle all machine-dependent issues in one package. However, because his mapping table is a parsing table, it seems to drive only the code selection and machine-dependent optimization phases. How the storage allocation and register allocation are driven by the table (or whether by other means) is not clear. Besides, in his system, each IL operator must have a corresponding machine instruction with the same opcode. Complex source operations must be decomposed into these IL operations before used as the input to his system.

The present work produces a retargetable CG covering the following code synthesis tasks and all these tasks are driven by the mapping table:

- the storage allocation (both static and dynamic),
- the data accessing (under various implementation decisions about the non-local data access methods),
- the code selection (with all source operation undecomposed),
- the register allocation (driven by the mapping table also),
- the procedure-like call and parameter-passing (supporting different commonly used strategies).
2. FIRST ORDER PREDICATE LOGIC AND PROLOG

In this chapter we introduce the basic concepts of first order predicate logic and the logic programming language PROLOG. The main purpose is to make it easier to understand the PROLOG programs appearing in this thesis as examples. Therefore the readability takes priority over the formality in this presentation.

2.1 First order predicate logic

First order predicate logic (/Nilsson 1982/) is a system of logic capable of expressing much of mathematics and common human reasoning.

2.1.1 Syntax

The data in predicate logic are called terms. Terms can be:

Constants - any strings of characters, e.g. a, acc, #M, #R, +, <-

Functions - an n-ary function consists of a function name followed by n terms called its arguments in a pair of parentheses. For example ADD(a) is a unary function.

Lists - a list is a parenthesized sequence of terms of any length. It is in fact an abbreviated representation of an anonymous binary function cons( _, _ ) with right recursion. (_ stands for a term whose identification is not relevant). Examples are: (a #M), (<- acc (+ acc (a #M))).

The basic class of logic expressions are the atomic formulas. An n-ary (n>0) atomic formula is a predicate symbol followed by n terms in a pair of parentheses.
For example, $P(a, b, c)$ is a trinary atomic formula, $Q(e, f(c, d))$ is a binary atomic formula.

With several logic operators we can build more interesting expressions - the well-formed formulas (wffs) recursively from atomic formulas in a way analogous to ordinary arithmetic expressions.

- (unary operator) is read "not"
+ (binary operator) is read "implied by"
& (binary operator) is read "and"
V (binary operator) is read "or"

In theory, - and + are enough, & and V are introduced to improve the readability of some wffs involving too many - and +.

Here are some examples of wffs:

$$
\neg P(a, b, c) \\
Q(e, f(c, d)) + P(a, b, c) \\
( \neg P(a, b, c) ) V ( Q(e, f(c, d)) + P(a, b) )
$$

2.1.2 Semantics

To define the semantics of a wff, we associate with the wff a domain $D$ in the following way:

<table>
<thead>
<tr>
<th>in the wff</th>
<th>in the domain $D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant terms</td>
<td>elements</td>
</tr>
<tr>
<td>function terms</td>
<td>functions</td>
</tr>
<tr>
<td>predicates</td>
<td>relations among the elements</td>
</tr>
</tbody>
</table>

This association constitute an interpretation of the wff. Then we can talk about the value of the wff under this interpretation. First, we define the value for each atomic formula:

An atomic formula has the value TRUE, if the elements associated
with the terms of the formula satisfy the relation associated with the predicate, otherwise, it has the value FALSE.

Then the values for general wffs can be defined recursively by the truth-table

<table>
<thead>
<tr>
<th>α</th>
<th>β</th>
<th>-α</th>
<th>β•α</th>
<th>α∨β</th>
<th>α&amp;β</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

The greek letters α and β are any wffs, T and F are the abbreviations of TRUE and FALSE

If a wff has the value TRUE under some interpretation, we say that the interpretation satisfies the wff. And the interpretation is said satisfying a set of wffs S if it satisfies every wff in the set.

Further, we can use logic variables (represented by the letters x, y, z, ... and ranging over the domain of the interpretation) and the universal quantifier ∀ (meaning "for all") and the existential quantifier ∃ (meaning "there exists") to express the ideas about "any objects" or "at least one object" neatly, without listing all possibilities explicitly.

Logic variables are used in wffs as terms just like constant or function terms syntactically. However, the semantics are quite different. Their meanings are associated with the quantifiers they follow. A quantifier prefixes a wff or its subexpression and is followed by one or more variables. Within the scope of the quantifier, any occurrences of the following variables are quantified by it. A variable quantified by ∀ means "for all elements in D", while one quantified by ∃ meaning "there exists in D".

For example, if R( _, _ ) is a binary predicate and D={a,b,c}, then

(∀x) R(x,b) is the abbreviation of R(a,b) & R(b,b) & R(c,b),
(∃x) R(x,b) is the abbreviation of R(a,b) V R(b,b) V R(c,b).
2.2 Logic programming

2.2.1 Inference viewed as computation

In first order predicate logic, there are inference rules that allow us to derive new assertions from a set of given ones. Logic programming has grown out of the idea that this logic inference procedure could be automated and viewed as computations.

Given a set of wffs S and another wff W, if any interpretation satisfying S also satisfies W, then W is said a logic consequence of S. Proving that W is a logic consequence of S equals demonstrating that no interpretation can satisfy the union of S and (¬W).

This task sounds quite formidable at the first glance. As the matter of fact, in its full complexity, when the domain D is an infinite set and logic variables are involved, this is an undecidable problem. There can exist no effective procedure that will always decide whether or not an arbitrary wff is a logical consequence of an arbitrary set of wffs. Fortunately, it has been proved that, if W is a logical consequence of S, such procedures do exist being able to report this fact. In practice, this is useful enough.

Even more interesting is the fact that some of these procedures can be carried out mechanically, highly suited to computer implementation. We are talking about the resolution inference. In fact, the research activity in this area following the seminal paper /Robinson 1965/ eventually gave birth to logic programming and various logic programming languages among which PROLOG is the dominant one.

The resolution inference demands that the wffs be first put into a special, convenient form - the clause form. In the following, we introduce the clause form and its Horn subset upon which PROLOG is based. And we will explain the resolution procedure in the context of PROLOG. In this way we demonstrate the PROLOG working style as well.
2.2.2 The clause form of predicate logic

It has been proved that any wff can be put into the clause form which involves only \( \land \) and \( \lor \) through a sequence of simple transformations (see Nilsson 1982/ for the transformations). If we use \( P_1, P_2, \ldots, P_m \) and \( Q_1, Q_2, \ldots, Q_n \) for any atomic formulas, and the involved variables are \( x, y, z, \ldots \) the general clause form looks like:

\[
(\forall x, y, z \ldots) \left( Q_1 \lor Q_2 \lor \ldots \lor Q_n \lor P_1 \lor P_2 \lor \ldots \lor P_m \right)
\]

But we prefer the equivalent form:

\[
(\forall x, y, z \ldots) \left( (Q_1 \lor Q_2 \lor \ldots \lor Q_n) \land (P_1 \land P_2 \land \ldots \land P_m) \right)
\]

(0)

Here the existential quantifier \( \exists \) is eliminated. All logic variables \( x, y, z \ldots \) involved in (0) mean "for all". We will omit \( (\forall x, y, z \ldots) \) in the following presentation. Because every clause is always prefixed by it, there is no need to write it explicitly.

For most applications of logic (and also for ours), a restricted class of clauses is sufficient. These are the so-called Horn clauses (with \( n \leq 1 \) in the expression (0)). A Horn clause is one of:

1. \( Q_1 \) \((n=1, m=0)\)

2. \( Q_1 \land (P_1 \land P_2 \land \ldots \land P_m) \) \((n=1, m>1)\)

3. \( \neg (P_1 \land P_2 \land \ldots \land P_m) \) \((n=0, m>1)\)

(1) and (2) are easy to understand, while (3) needs some explanation. The original form of (3) is

\[
\neg (P_1 \land P_2 \land \ldots \land P_m)
\]

which means negation. However we can also interpret it in another way. Suppose we are given a set of wffs \( S \) and \( (P_1 \land P_2 \land \ldots \land P_m) \) is another wff \( W \). As mentioned before, if we can demonstrate that the
union of $S$ and $\neg W$ is unsatisfiable, we have proved that $W$ is a logical consequence of $S$ (so we have solved the problem $W$). Now as (3) exactly expresses the $\neg W$, we can regard it as the question:

Is $(P_1 \land P_2 \land \ldots \land P_m)$ true for certain instances of the variables?

This is exactly the way a PROLOG program states a problem to be solved.

2.2.3 The syntax of PROLOG - Horn clauses

Using the predicate logic as a programming language was advocated by Kowalski 1974/. Various versions of PROLOG have different representations, but in essence all are based syntactically on the Horn clauses and semantically on the resolution principle. In this section we describe the syntax of PROLOG and leave the semantics in the next one. The notation and terminology about PROLOG used in the rest of this thesis are those for the NTH PROLOG, and from now on we call Horn clauses simply clauses.

A PROLOG program is a set (a conjunction, in fact) of clauses. There are three kinds of clauses which in essence are the same as (1), (2), (3) above respectively. But there are some changes in the syntax and terminology:

<table>
<thead>
<tr>
<th>in predicate logic</th>
<th>in the NTH PROLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>the universal quantifier $\forall$ and variables $x, y, z \ldots$</td>
<td>$\forall$ is omitted, identifiers ending with ' are used for variables: e.g. $x', y', z' \ldots$ meaning &quot;all&quot;</td>
</tr>
<tr>
<td>atomic formulas $+$</td>
<td>literals</td>
</tr>
<tr>
<td>(1) $Q_1$</td>
<td>$Q_1$,</td>
</tr>
<tr>
<td>(2) $Q_1 + (P_1 \land P_2 \land \ldots \land P_m)$</td>
<td>$Q_1 : P_1, P_2, \ldots, P_m$.</td>
</tr>
<tr>
<td>(3) $+ (P_1 \land P_2 \land \ldots \land P_m)$</td>
<td>PROBLEM : $P_1, P_2, \ldots, P_m$.</td>
</tr>
</tbody>
</table>
In the following, for each of the three kinds of clauses we give examples showing PROLOG counterparts. Taken together these examples form a complete PROLOG program specifying a toy code generator. In the next section we will give the trace of the execution of this program, showing the working style of PROLOG.

The first one is the unconditional clause (1).

Examples:

\[ ACCESS\_MODE(\{r' \#R\}, r'). \quad \text{\texttt{ZZ} any object taking the form \{r' \#R\}} \]
\[ ACCESS\_MODE(\{m' \#M\}, m'). \quad \text{\texttt{ZZ} similarly for the memory mode.} \]

The second one is the conditional clause (2).

Examples:

\[ INSTRUCTION(\{<- \text{acc} (+ \text{acc x'})\}, \text{ADD}(y')): ACCESS\_MODE(x', y'). \]

\text{i.e. the meaning is that, the instruction \text{ADD}(y') implements the operation expressed in the list (\{<- \text{acc} (+ \text{acc x'})\}), if \text{x'} is an access mode and \text{y'} is its assembly representation.} \]

\[ CODE\_GENERATION(\text{exp'}, \text{ins'}): INSTRUCTION(\text{exp'}, \text{ins'}). \]

\text{i.e. meaning: the instruction \text{ins'} can be generated as the assembly code for the operation expressed in \text{exp'}, if \text{ins'} implements \text{exp'} as the literal INSTRUCTION(\text{exp'}, \text{ins'}) asserts.} \]

In a conditional clause, the literal on the left side of ":=" is called the conclusion, while those on the right side are the conditions.
The third one is the question (3). Every PROLOG program has the unique \texttt{PROBLEM} clause giving the question to solve. For example,

\begin{verbatim}
PROBLEM:CODE GENERATION((<- acc (+ acc (a #M))), code'),
  WRITE(TERMINAL, code'). % predefined, always solvable.
\end{verbatim}

This clause asks to find the code for the expression \((<- acc (+ acc (a #M)))\) and output the solution on the terminal.

Putting all of the examples so far together we make a complete PROLOG program

\begin{verbatim}
<1> PROBLEM:CODE GENERATION((<- acc (+ acc (a #M))), code'),
    WRITE(TERMINAL, code').
<2> CODE GENERATION(exp' ins'):INSTRUCTION(exp' ins').
<3> INSTRUCTION((<- acc (+ acc x')) ADD(y'):ACCESS MODE(x', y').
<4> ACCESS MODE(x' #R), r').
<5> ACCESS MODE(m' #M), m').
\end{verbatim}

With respect to predicate logic, \(<2> - <5>\) constitute a set of wffs \(S\), and \texttt{CODE GENERATION((<- acc (+ acc (a #M))), code')} in \(<1>\) is the wff \(W\) to be proved. \(<1>\) is the \(-W\), and the whole PROLOG program \(<1> - <5>\) is the union of \(S\) and \(-W\). In demonstrating the unsatisfiability of this union, we must find a counterexample of the logic variable \texttt{code}', which gives \(-W\) the value FALSE (therefore gives \(W\) itself the value TRUE) if \(<2> - <5>\) all have the value TRUE. Thus, this counterexample in turn serves as a solution.

The question is how to demonstrate the unsatisfiability (so-called \texttt{re}futation process), that is, how to execute the PROLOG program. That is the subject of the next subsection.
2.2.4 The semantics of PROLOG - unification and backtracking

The execution of a PROLOG program is a process of refutation based on the resolution principle. The main action involved is the unification. But to be a computer program, we should also decide the execution order. In the NTH PROLOG the order is top-down, left-to-right, depth-first and with backtracking. We will explain the refutation process briefly as follows.

Starting with the unique PROBLEM clause,

```
PROBLEM:P1, P2, ... Pm.
```

we execute P1 - Pm from left-to-right. To execute P1, the system searches for the first clause (top-down)

```
Q1:R1, R2, ... Rn.
```

whose conclusion Q1 matches P1. The matching process is called the unification, finding the most general common instance of the two, which is unique if it exists.

/Nilsson 1971/, for example, gives the detailed unification algorithm on page 175-178. Here it is enough to state that, the matching of the two literals P1 and Q1 is performed on pairs of symbols occupying the same positions on P1 and Q1, one pair after another. Two identical predicate names, two identical function names or two identical constants match each other. While a variable matches any term in which it is not involved and the corresponding term is substituted for the variable, and any other occurrences of the variable in the two clauses are bound to the same item.

After a successful unification we replace P1 by the conditions R1, R2, ... Rn (if n > 1) of the unified clause and try to solve them (again from left to right) before the original P2, ... Pm. That is the meaning of depth-first.

If the unified clause is unconditional, it has been executed (or
solved), and we proceed to solve P2. Here we have found certain instances of the variables which make P1 true.

This process will continue until all of P1 - Pm are executed. Then the refutation is completed, because we have found certain instances of the variables to make all of P1 - Pm true, a fact directly refusing the negation over them as the PROBLEM clause claims. And the instances of the variables involved in P1 - Pm thus constitute a solution to the original problem.

If at any point of the execution we fail to find any matching clause (in this example we did not meet this situation), we backtrack from that point, rejecting the most recently executed clause, undoing the substitutions made in the unification process for the rejected clause, trying to find the subsequent matching clause.

In summary, in a PROLOG program, the clauses specify the logic of the algorithm (what to do), and the search strategy described above provides a simple control mechanism (how to do it efficiently). In most cases, by wisely ordering of the clauses within a PROLOG program and the conditions within each conditional clause, the natural specification of an algorithm and a quite good implementation of the algorithm are one and the same. This is one of the main attractions of logic programming (WARREN 1980/). And in our CGWS, we will take full advantage of this feature.

2.3 Running a PROLOG program

In this section we list the running trace of the toy code generation program <1> - <5>, showing more concretely the resolution process as well as the working style of PROLOG program. The English sentences after % are comments.
The Running Trace of the mini-CG (1) - (5)

\[ \text{CODE\_GENERATION} \{ (\neg \text{acc } (\land \text{acc } (\text{a } \#M))) \text{, code'} \} \quad \text{\% P1 of (1)} \]

\[ \text{UNIFY } (\neg) \text{ with } (\neg) \quad \text{\% successful unification, exp' in} \]
\[ \text{UNIFIED in the list, while} \]
\[ \text{CALL } (\neg) \text{ FROM } (\neg) \quad \text{\% code' in bound to ins'} \]

\[ \text{INSTRUCTION} \{ (\neg \text{acc } (\land \text{acc } (\text{a } \#M))) \text{, ins'} \} \quad \text{\% derived from (2)} \]

\[ \text{UNIFY } (\neg) \text{ with } (\neg) \quad \text{\% successful unification, x' in} \]
\[ \text{UNIFIED in the list, while} \]
\[ \text{CALL } (\neg) \text{ FROM } (\neg) \quad \text{\% ins' is bound to ADD(y')} \]

\[ \text{ACCESS\_MODE} \{ (\text{a } \#M), y' \} \quad \text{\% derived from (3)} \]

\[ \text{UNIFY } (\neg) \text{ with } (\neg) \quad \text{\% failed unification} \]
\[ \text{UNIFY } (\neg) \text{ with } (\neg) \quad \text{\% successful unification, m' in} \]
\[ \text{UNIFIED in the list, while} \]
\[ \text{CALL } (\neg) \text{ FROM } (\neg) \quad \text{\% also bound to a} \]

\[ \text{RETURN to (3)} \quad \text{\% P5 is unconditional,} \]
\[ \text{RETURN to (2)} \quad \text{\% P1 is solved, and code' is} \]
\[ \text{RETURN to (1)} \quad \text{\% finally bound to ADD(a)} \]

\[ \text{WRITE(TERMINAL,ADD(a))} \quad \text{\% P2 of (1)} \]

\[ \text{RETURN} \quad \text{\% it is always solvable,} \]
\[ \text{OUTPUTting the solution ADD(a)} \]

2.4 Nonstandard features in NTH PROLOG

To be a practical programming language, any PROLOG version should provide something more than the standard features given above. The most important nonstandard features in the NTH PROLOG are:
1. List processing

We have already introduced the concept list. Here are more about list processing specific to the NTH PROLOG. Basically, this PROLOG treats lists in the same way as the programming language LISP does:

\[
\begin{align*}
() & \quad \text{stands for NIL, a constant;} \\
(a.b) & \quad \text{stands for cons(a,b);} \\
(a) & \quad \text{stands for (a.NIL), or cons(a,NIL);} \\
(a \ b) & \quad \text{stands for (a.(b.NIL)) or cons(a, cons(b,NIL))} \\
(a \ b \ c) & \quad \text{stands for (a.(b.(c.NIL)))} \\
(a \ b.c) & \quad \text{stands for (a.(b.c)).}
\end{align*}
\]

Lists are powerful data structures in PROLOG. Specially, according to the definition above, \( (t_1 \ldots t_n.\ x') \) may match any list with at least \( n (> 0) \) elements, and \( x' \) is bound to the rest of the list. This is a great flexibility.

2. Backtrack controlling

The predefined zero-argument predicates ACCEPT and REJECT are used to cut remaining alternative solutions for the clause which contains them as part of the conditions. ACCEPT is unconditionally true, and REJECT false. They are used for saving unnecessary search efforts.

3. Built-in arithmetic operations

For example the predicate PLUS(x',y',z') solves the equation \( x' + y' = z' \) when at most one variable is unbound. From the arithmetic operations we can define the comparison operations. For example,

\[
\begin{align*}
\text{EQUAL}(x',x'). \\
\text{LESS}(x', y') : \text{PLUS}(x', \text{some}', y'), \text{NO}(\text{EQUAL}(\text{some}' 0)). \\
\text{NO}(x'): x', \text{REJECT}. \\
\text{NO}(x').
\end{align*}
\]

The definition of \( \text{NO}(\_) \) is beyond first order logic, allowing logic variables to be predicates.
To improve readability, the following constructs are used in the remainder of this thesis:

\[ x' + y' = z' \text{ for } \text{PLUS}(x', y', z'), \]
\[ A(x' + y') \text{ for } A(z'), \text{PLUS}(x', y', z'), \]
\[ x' < y' \text{ for } \text{LESS}(x', y'), \text{ and so on.} \]

4. Dynamic modification of programs

There are two more predefined predicates \texttt{ASSERT(\_)} and \texttt{RETRACT(\_)} capable of modifying PROLOG programs dynamically. The first one takes its argument as an unconditional clause and inserts it at the end of the program. The second checks if there are unconditional clauses in the program being the same as its argument, and deletes the matched ones from the program or does nothing if there is none matched.

We will give explanations for other nonstandard features along with their first appearances in this thesis.
3. TARGET MACHINE DESCRIPTION IN PROLOG

3.1 The design principle

The Target Machine Description (TMD) in our CGWS consists of the following information:

- that part of the target machine architecture, which is relevant to code generation; (This is necessary for any CGWSs).
- the compiler implementation decisions; (Because our CGWS produces a "complete" code generator, not only the instruction selector).
- some guidance for the pattern matching which is the main strategy for the CGG and the CG, such as in what order to perform the pattern matching.

In short, the description is code-generation-oriented, in a format suitable to the tree pattern matching. The TMD includes the compiler implementation decisions, describing a compiler writer's virtual machine. The purpose is to do storage allocation, data accessing, register allocation and procedure handling in a machine-independent way as we do for the instruction selection. The isolation of all machine-dependent and implementation-dependent information facilitates the retargetability to a great extent. For example if the compiler designer changes his mind about the layout of activation records, what he needs to do is simply to modify the description about the changed layout instead of rewriting many parts in the compiler.

The TMD is not a general hardware description language like ISP. ISP contains too many details of the architecture, which are important for other uses but irrelevant to code generation. This would give a CGWS directly based on ISP a poor performance. On the other hand, ISP does not contain anything about compiler implementation decisions. Previous work (e.g. /Fraser 1977/, /Cattell 1978/) has shown that ISP must be symbolically manipulated before it can be used as a starting point of
automatic generation of software. This symbolic manipulation is an interesting subject in itself, but not for our purpose.

Our machine description model contains the sufficient and necessary information for a CGWS. In this chapter we show that the logic programming language PROLOG is a suitable tool for this description. The resulting representation is precise, compact, both machine and human readable, and easy for the user (the code generator writer) to build from the machine manuals.

The range of the target architecture investigated in this work is broad. They are widespread register machines:

- single-accumulator, one-address machines such as ND-100;
- multiple-register, two-address machines with at least one operand in register such as MOTOROLA-68000;
- multiple-register, two-address machines such as PDP-11;
- three-address machines such as VAX.

The scope of run-time conventions considered is broad too, including various implementation schemes of block-structured languages like ALGOL or PASCAL.

In the following section we will show how to specify all these components in PROLOG, taking PDP-11 and ND-100 as examples. The complete TMDs for these two machines are found in Appendix 3.

3.2 The implementation decisions and the hardware support

We must first describe the hardware resources (such as the memory and registers) before the machine operations (such as the addressing capability and the instruction set) can be described properly. However, how to utilize these resources in the code generation process is up to the compiler designer. In this section we deal with the compiler implementation decisions and the hardware resources, postponing the description of the machine operations to the next section.
3.2.1 The memory and the data type mapping

The first hardware resource we consider is the memory. It is an array of locations holding data objects as well as instructions. To code generation, one of the most important things about the memory is how to use memory units of different sizes to hold different kinds of data objects. That is the mapping between machine data types and the IL data types. First we define the machine data types in a machine independent way:

- BIT - 0 or 1.
- BYTE - a byte is the smallest memory unit which has an integer address. For example on PDP-11 a byte has 8 bits. On ND-100, a byte consists of 16 bits and is usually called a "word";
- WORD - two successive bytes;
- THREE - three successive bytes; (for example on ND-100 for real)
- DOUBLE - four successive bytes or two successive words;

and so on. Each machine data type defined thus has a size: the number of bytes it takes. Some machines demand that, for example a word must start from a byte with an even address. We express this fact by saying that "the align of WORD is 2".

If there are machine operations directly upon a machine data type, we say the machine supports this type. The mapping from IL data types onto the machine data types supported by the machine is a simple decision and can be expressed in PROLOG as

\[
\text{DATA_TYPE_CONVERSION(IL_type', machine_type', size', align').}
\]

For example on PDP-11/70 we have:

\[
\begin{align*}
\text{DATA_TYPE_CONVERSION(BYTE, BYTE, 1, 1).} \\
\text{DATA_TYPE_CONVERSION(BOOL, BYTE, 1, 1).} \\
\text{DATA_TYPE_CONVERSION(INT, WORD, 2, 2).} \\
\text{DATA_TYPE_CONVERSION(REAL, DOUBLE, 4, 2).} \\
\text{DATA_TYPE_CONVERSION(POINTER, WORD, 2, 2).}
\end{align*}
\]
Each unconditional clause defines a data type conversion, giving the size and the align of the corresponding machine data type.

Given this information in the TMD (and copied into the MT later) the CG can determine the displacement of each name in the IL program under compilation within the activation record the name belongs to.

If an IL data type has no corresponding machine supported data type, as REAL in PDP-11/20, then a virtual machine data type must be provided via run-time code for each operation on that virtual data type. This run-time code in principle can be derived automatically using the axiomatic approach (described in the next chapter), but in practice it is very hard and time-consuming to do so (for example, deriving REAL code in PDP-11/20). We have a practical alternative: providing a library of hand-written run-time routines and adding them as new templates to the automatically derived basic part of the mapping table (MT).

3.2.2 The (user) allocatable registers

The register resources are divided into two groups: the (user) allocatable and the dedicated. The former is used in the register allocation phase of the CG, while the latter is mainly for manipulating the run-time stack.

The information about the allocatable registers necessary for the CG includes:

- types - data registers (each holds a data item) or location registers (each holds an address). In the case of data registers which machine data types can be held in the registers; some registers such as R0 - R5 in PDP-11 are both data and location registers.
- status - FREE or BUSY. Initially all user allocatable registers are set FREE, and this will be modified along with the code generation process.
So the proper data structure in PROLOG to describe one register is a list of three terms specifying its name, type and status respectively:

\[
\text{(name_of_register', TYPE(type1', type2'), status')} \]

The second term is a binary function. The first argument type1' is a list of DATA and/or LOCA, showing that the register can be used as a data register or location register or both. The second argument is a list of data types which can be held in this register when it is used as a data register.

The description of all allocatable registers is a unary predicate ALLOCATABLE_REG(_), and the argument is a list of all allocatable registers. For example, on PDP-11, we have

\[
\text{ALLOCATABLE_REG( \{ (R0 TYPE((DATA LOCA), (BYTE WORD)) FREE) (R1 TYPE((DATA LOCA), (BYTE WORD)) FREE) (R2 TYPE((DATA LOCA), (BYTE WORD)) FREE) (R3 TYPE((DATA LOCA), (BYTE WORD)) FREE) (R4 TYPE((DATA LOCA), (BYTE WORD)) FREE) \}) .}
\]

This unconditional clause asserts that on PDP-11, R0 - R4 are the 5 registers which the code generator writer can use for temporary calculation of values or addresses, and the legal data types are BYTE and WORD. The fact that R5 is reserved for the frame-pointer (the base register) by the compiler designer will be expressed in the next subsection. However the above description alone excludes the possibility of the allocation of R5 by the user.

On ND-100, we have

\[
\text{ALLOCATABLE_REG( \{ (A TYPE((DATA), (BYTE)) FREE) (X TYPE((DATA LOCA), (BYTE)) FREE) (D TYPE((DATA), (BYTE)) FREE) (T TYPE((DATA), (BYTE)) FREE) (DA TYPE((DATA), (WORD)) FREE) (FA TYPE((DATA), (THREE)) FREE) \}) .}
\]
But here is a special matter to take care of. On ND-100, the register DA (for the long integer operations) is in fact a concatenation of the A register and the D register, while the register FA (for the real operations) is a concatenation of T, A and D. We express this fact in another predicate OVERLAY(_,_):

OVERLAY(DA, (A D)).
OVERLAY(FA, (T A D)).

The register allocation routines will check this information to be sure, for example, if some registers are allocated or freed implicitly, and if so, set their status accordingly.

3.2.3 Dedicated registers and run-time conventions

The virtual data type operations are not the only run-time routines. Other in-line or out-of-line routines include procedure linkage, parameter passing, dynamic storage allocation and operating system interface. They are dependent not only on hardware support (such as the dedicated registers) but the run-time conventions as well. Of course the conventions themselves depend on the architecture to a great extent. However, there is still some design flexibility left. Our CGWS design principle is to provide as many schemes as possible of those that are commonly used in compilers, and let the user choose a convenient one.

The run-time conventions mainly concern the mechanism for keeping track of the positions of various activation records within the run-time stack and the layout of activation records. For each of these two aspects, we list the usual schemes, the necessary hardware support for individual schemes and the description methods.
3.2.3.1 Keeping track of the positions of activation records

There are four commonly used mechanisms:

1. By the static links;

To store with each procedure a pointer (called a static link) in each activation record for the procedure, pointing to the currently effective activation record of that procedure which physically surrounds the said procedure in the program (/Aho & Ullman 1977/, also used in RUNIT MARY compiler and CHILL compiler). Thus at any time of the execution of the program, we maintain a chain of static links. To reference non-local data, we descend the chain to find all necessary statically enclosing procedures, and the data accessed appear eventually as local in a proper activation record. (See section 5.3.).

The hardware support needed for this scheme is simple: a dedicated location register called the frame-pointer or base register pointing to the base of the topmost activation record (as a matter of fact, this is a necessity for all schemes of implementing block-structured languages). Referencing local data is made by indexing off this register. However for efficiency it is desirable for the machine to provide another dedicated register called the stack-pointer SP. We will describe these hardware resources in the next subsection connected with the layout of activation records.

2. By register display;

This scheme uses a set of dedicated location registers as display array. In this case we should have a clause to specify the registers. For example in the PASCAL 6000 compiler described in /Wirth 1977/, the decision can be expressed as

`DISPLAY_REGISTER((R1,R2,R3,R4,R5)).`

The static links remain as in the scheme 1, but this time they are used for updating the display (one of the tasks of so-called run-time system routines) instead of referencing to data. Here the data referencing is much simpler than in scheme 1:

`absolute_address = display_array[static_level] + offset`

This calculation can be carried out by an ordinary indexing access mode which is supported by almost any modern computer.
3. By memory display;

Unfortunately, not many mini- or micro- machines can afford as many as 5 registers dedicated to this purpose. So we have the alternative scheme 3. It works basically in the same way as scheme 2, but as there are not so many registers in the target machine, we place the display in fixed memory locations that are indexed off the DISPLAY_POINTER (usually a location register). References to data objects now involve double indexing. The following clause

DISPLAY_POINTER(R1).

is an example description for the display-pointer.

4. By stack display;

To store the display on the run-time stack and create a new copy (modified somehow) at each procedure entry. (/Gries 1971/, /Aho & Ullman 1977/). In this case, the contents of the linkage-locations in activation records have some change: no static links needed, but the display resides in these locations.

References to non-local data objects are made by first finding the appropriate display element in the activation record, and then the data. Again, double indexing is involved.

The routines setting up the linkage information for the schemes described above (e.g. setting up the static links, modifying the displays) belong to the run-time system. They can be automatically derived from the TMD in principle. However, because most of them require extremely efficient object code, the hand-written approach is recommended. The same is true for the in-line code to deal with procedure call, parameter passing, etc. They are hand-written and added to the MT.

3.2.3.2 The layout of activation records

The number of possibilities is in this case much larger than for those cases described in the previous subsection, too large to list them all. Fortunately, the description methods are quite uniform, we will explain them by examples.
Whatever the layout is, an activation record consists of three parts: the locations for the linkage information, for the local data and for the parameters and the return value (if any). The question is the relative position. The factors relevant to the CG are:

- the identifications of the frame-pointer and the stack pointer,
- the direction in which the stack grows,
- the position of the base of an activation record,
- the starting position for local data,
- the position of the static link,
- usually the linkage information is at the beginning of a record, but what about the parameters? are they located before or after the linkage locations?
- the position for the return value.

In the following, we give two examples of the layout of activation records, and show how to describe them in PROLOG.

Example 1. (a virtual machine on the top of PDP-11)

<table>
<thead>
<tr>
<th>The layout</th>
<th>The PROLOG description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter1</td>
<td>PARAMETER(BEFORE). % parameters are pushed % onto the stack before % the linkage locations. % (The push instruction % is available).</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>parameter2</td>
<td>FRAME_POINTER(R5).</td>
</tr>
<tr>
<td></td>
<td>STATIC_LINK(+, 0).</td>
</tr>
<tr>
<td>STATIC_LINK</td>
<td>+R5</td>
</tr>
<tr>
<td>return value</td>
<td>DATASTART(-,6). % data start here and % the first one is the % return value.</td>
</tr>
<tr>
<td>local data</td>
<td>STACK_POINTER(SP).</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>=SP</td>
</tr>
<tr>
<td>000000</td>
<td>FRAME_DIRECTION(DOWN).</td>
</tr>
</tbody>
</table>
Example 2. (a virtual machine on the top of ND-100)

<table>
<thead>
<tr>
<th>The layout</th>
<th>The PROLOG description</th>
</tr>
</thead>
</table>
| STATIC_LINK | STATIC_LINK(-, 128).  
|            | %% the base is in the  
|            | %% middle of the record |
|            | %% of 256 bytes.        |
|            | PARAMETER(AFTER).       |
|            | %% para. are after linkage |
| parameter1 | DATASTART(-, 125).     |
|            | %% because of lack of SP,|
|            | %% and treated as local|
|            | %% data.                |
| parameter2 | RETURN_VALUE           |
|            | local data             |
|            | FRAME_POINTER(B).       |
|            | %% pointing to the      |
|            | %% middle of records    |
|            | FRAME_DIRECTION(UP).    |
|            | 177777                  |
3.3 Machine operations

The instruction set and the access mode set constitute the operation part of the target machine. For each instruction or access mode the information relevant to our CGWS is:

- The input - the operation carried out;
- The output - the assembly format;
- The cost - the time consumed for the execution and/or the space taken for the static representation in memory;
- The data type - the machine data type of the operands.

3.3.1 The access modes

Access modes specify the available addressing capability on the target machine. In PROLOG each access mode is expressed as an unconditional clause. The predicate has two arguments. The first argument is the input: a prefix expression tree (in PROLOG it is a nested list) specifying the address calculation (implicit to the user) to access a constant, a (data) register, or a memory location. We call this kind of expressions the location trees. The second argument specifies the assembly output for this mode.

Now let us examine the location trees more closely. These expressions are specific for address calculation, consisting of operators and operands. The leaves (the bottom operands) are parameterized using PROLOG variables, with the following patterns:

\[(c \ #C): \text{ the integer constant } c';\]
\[(r' \ TYPE(\text{DATA}, t')): \text{ data register } r' \text{ for data type } t';\]
\[(r' \ TYPE(\text{LOCA}, t')): \text{ a memory location whose address is held in the location register } r'; \text{ here } t' \text{ is irrelevant}\]
\[(a' \ #AM): \text{ a memory location with absolute address } a';\]
\[(l' \ #RM): \text{ a memory location with relative (to PC, the program counter) address } l'.\]
In theory, we need only two kinds of leaves to construct all access modes: constants and registers. The last two in the above set are just the abbreviations of some complicated modes, and introduced for practical convenience.

The set of operators used in location trees is very small. It consists of only three operators: the address-addition (or pointer-addition) " + " and " - " and the "contents of" (or indirection) " <> ", the latter takes a memory location as the argument and returns the contents of it.

For example, on PDP-11, the index mode can be expressed by:

\[ \text{INDEX\_MODE}\left( (+ \ (r') \ \text{TYPE}(\text{LOCA}, \ t')) \ (c' \ #C) \), \ c'(r') \right). \quad (1) \]

(In Appendix 3 we use AM2 for this access mode). The second argument c'(r') is the assembly format for this mode. The first argument states how to calculate the address of a memory location by indexing. Most of the access modes describe memory locations, in which, the location trees always give the addresses of the locations, rather than the contents of them. The only two exceptions are the immediate mode and the data register mode.

As another example, consider the "pre- and post- indexing" mode on ND-100:

\[ \text{PRE\_POST\_INDEX\_MODE}\left( (+ \ (<>) (+ \ B \ (d' \ #C))) \ X), \ (<X), \ B \ d') \right). \quad (2) \]

Here the location tree is simplified to improve the readability: omitting the detailed description of locaion registers B and X. But the exact syntax form is not important. The important thing is to understand the address calculation described: First we add a constant d' to the B register, the result is regarded as an address; then adding the contents of this address to the X register, we get the effective address.

So far we haven't mentioned the issue of data types. There are many ways to describe this information. We prefer to isolate it in the operators. All operators in the description of instructions and access
modes are unambiguous with respect to the data types of their operands. In expression (1) above the operator + is the address-addition which is the same as the integer-addition. But the byte-addition is different, it is the operator +B. Similarly, <> in expression (1) takes an address as the argument, and returns an integer which can be used as an address or pointer. (Another indirect operator <>B also takes an address as the argument, but returns a byte).

The description of an access mode usually returns an address - the address of the first byte of the memory location accessed. In the description of instructions, we use unambiguous operators to indicate the data types of the operands.

On the other hand, in the CG, the IL programs use generic operators like in the source programs. But to do the pattern matching, these generic operators must be disambiguated according to the data types of their operands before the instruction selection. (See section 5.3).

At code generation time, data accessing consists of two steps. Each name in the statements of the IL program will first be extended into a location expression consisting of the same set of leaves and operators given above, but maybe with multiple indexing and/or multiple indirections (see section 5.3.). Then the mapping from this machine-independent "virtual access modes" to the available access modes described here, probably involving register allocation and extra code for the address calculation, is performed together with instruction selection.

3.3.2 The operand classes

Many instructions can use not only one but a set of alternative access modes as their operand, and what are really used may be the addresses specified in the access modes or the contents of them. For example, on PDP-11, the instruction ADD has two operands, each can be one of the 8 access modes. This means we need 64 clauses to describe ADD. To reduce the number of instructions to be described, we give a name for each set of alternative access modes, and in the instruction description we use these names as well as access modes to
represent operands (see the next subsection). These sets are called operand classes.

There are two operand classes of special importance. One is called the destination operand class. We use the predicate DST_OC(_, _, _) to describe it, and for each access mode included in this set such as the index mode (1) on PDP-11, there is a conditional clause:

$$\text{DST\_OC}(x', y', 1) : \text{INDEX\_MODE}(x' y').$$

(Indeed there are 8 such clauses for DST\_OC on the TMD of PDP-11). The third argument specifies the cost to access this operand. In our case it is the space requirement for the operand specification, and it is 1 word for the constant c’ in (1) on PDP-11. This is only one of the possible definitions of "cost", which is generally defined by the user as a weighted combination of space and time.

The other important class is the source operand class described by the predicate SRC\_OC(_, _, _). This time it is the value (the contents of the location) to be used, and we use

$$\text{SRC\_OC}((\langle\rangle x'), y', 1) : \text{INDEX\_MODE}(x', y').$$

and so on to describe this class. Remember that, \langle\rangle returns an integer. So we have another source operand class

$$\text{SRC\_OC\_B}((\langle\rangle B x', y, 1)$$

which returns a byte.

### 3.3.3 The instruction set

Each instruction is expressed in PROLOG as a conditional clause. The conclusion predicate is always CODE(_, _, _), with three arguments specifying the operation (again prefixed expression tree, but the set of operators is much larger than in access modes), the assembly output format and the total cost of this instruction (including the explicit operation and the implicit operand accessing). The conditions give the operand classes and the rule to calculate the cost.
For example, in PDP-11, the integer addition instruction is described by

```
CODE($1' (+ $2' $3')), ADD(src', dst'), c'

SCR_OC($2', src', c'''),
DST_OC($1', dst', c'''),
CONTENTS($1' $3'),
c' = 1 + c'' + c'''
```

This clause asserts that if parameter $1'$ is any access mode belonging to the operand class DST_OC with assembly format dst' and cost c''', parameter $2'$ belongs to SCR_OC with assembly format src' and cost c''', and $3'$ is the contents of $1'$, then the instruction ADD(src' dst') implements the integer addition $1' <- (2' + 3')$ and the total cost c' is $(1 + c'' + c'''')$. Here the cost is defined as the number of words occupied by this instruction. Of course there should be the definition of the predicate CONTENTS, which is quite obvious: $(<> x')$ is the contents of x' except for data registers, in which case we use the same symbol to express the register itself as well as the value held in it.

In the example above the operator "+" is the integer addition by default. In specifying the byte addition, we will use a clause like the above, but using "+B" instead of "+" and ADDB instead of ADD. With this unambiguous operator + we can determine that $1'$ is a location, $3'$ is an integer like the contents of $1'$, and $2'$ is also an integer.

3.4 The guidance to the pattern matching

The main strategy for code derivation in the CG6 and the instruction selection in the CG is the general tree pattern matching. This strategy naturally has an effect on the TMD. As we have seen in the previous section, in the description of access modes and instructions we specify the implemented operations as prefixed expression trees.
represented by PROLOG lists. In the CGG, the IL patterns have the same representation. The code derivation in the CGG are realised basically as tree pattern matching which is in turn implemented by using the PROLOG unification process. The same is true for the instruction selection phase in the CG.

In the description of access modes and instructions, for every commutative operator we add a new clause to specify the same access mode or instruction with the commuted expressions. We do so in order to test different target paths (see chapter 4).

In PROLOG, the order in which the unconditional clauses with the same predicate name or the conditional clauses having the same predicate name in their conclusions is important. This order has an effect on the search efficiency (for the CGG) as well as the code quality (for the CG). This ordering problem will be discussed in connection with the CGG and the CG in the following chapters.

The last question about pattern matching guidance is how to estimate the cost of an IL pattern. The reason is the following. Our code derivation algorithm described in chapter 4 uses exhaustive search. After one solution has been found, we proceed to find better ones. In searching for the latter, we would like to cut search paths as soon as they are obviously leading to worse solutions. The method to detect those paths is to estimate the cost c of the solution to which a search path leads. If c is an underestimate but still bigger than the cost of the best solution found so far, we are confident to cut the path to improve the search efficiency without loosing any promising solutions.

The cost estimate is a highly machine dependent issue. So we should include it in the TMD. For example, in PDP-11, suppose we have defined the number of words taken by an instruction (including the space for its operands) as its cost. Then, for an IL assignment with m operands which are not in registers, we can estimate the cost of the IL tree as c = 1 + m. This estimate is an underestimate. Of course we can choose some more sophisticated underestimate.
We use a binary predicate ESTIMATE(_, _) to describe this information. The definition of ESTIMATE(tree', cost') is included in the TMD and performs the calculation of cost' from tree' according to a scheme like the above one.
4. CODE GENERATOR GENERATION IN PROLOG

4.1 Introduction

In this chapter, we consider the specification of code generator generators. There are two major problems, none of them are trivial.

1. A heuristic search algorithm for code derivation (section 4.2).

The problem can be formalized as follows:

Given a machine M with instruction set \{m_1, m_2, \ldots, m_n\} and an IL pattern \( G \), find a sequence of instruction instantiations which implements the semantics of \( G \) and has the minimum cost (defined as a function of space and time).

The problem is a hard one in its full complexity. In section 4.2.1 we review some theoretical results which give the insight into optimal code generation problem, explaining why we should restate our aim as near-optimal code. Then in section 4.2.2. we formalize our code derivation problem as a general AI Production System and present our search algorithm in a general, skeleton form within this framework. How to instantiate this algorithm for code derivation is given in section 4.2.3.

2. The code generator generation scheme — how to select the IL patterns which constitute (together with their code derived) the templates driving the code generation phase.

This is by no means easier than the first problem. If we want to do the complete case selection (to detect all special cases of IL operator combinations or even the most common ones), it is indeed a very hard problem and has been treated very little in the literature so far. In section 4.3 we propose our scheme for code generator generation.
4.2 Code derivation

4.2.1 Theoretical background

The optimal code generation problem involves at least the following complications:

1. Some IL operators may not appear in a particular instruction set. If we take a formal approach to this problem, for example using program equivalence axioms to transform these operators into sequences of more primitive ones, we can not guarantee optimal code or any code at all because of the unsolvability of program equivalence: no set of axioms can express all the equivalences over program trees. On the other hand, for some IL operators there is more than one instruction or one instruction with more than one access mode which can implement them, a situation requiring exhaustive or heuristic search for the optimal solutions.

2. Even when there is a one-to-one correspondence between IL operators and machine instructions, the order in which the subtrees are evaluated may be important. And for commutative operators the choice of target operand (loaded into the same location as the result) is also important. Having ruled out common subexpressions, /Sethi & Ullman 1970/ gives a linear algorithm to generate optimal code for a simple machine model. /Aho & Johnson 1976/ extends this work, proposing a dynamic programming algorithm to produce optimal code for a broad class of machines. This algorithm requires time linear in the number of tree nodes and exponential in the number of instruction and access mode choices at each point. /Ripkin 1977/ uses an extended version of the Aho & Johnson algorithm in his machine-independent code generation scheme (not implemented).
3. If we take common subexpressions into account, the IL patterns are in fact DAGs (Directed Acyclic Graphs). Generating optimal code of DAGs for a single-register machine has been shown to be NP-complete (no polynomial algorithms known) by Bruno & Sethi 1976/. Aho et al. 1977/ shows that even a very simple DAG can require exponential time to produce optimal code for a machine with even an infinite number of registers. For single-register machines they give an optimal algorithm whose time complexity is linear in the size of the DAG and exponential in the amount of sharing. More recently, Carter 1982/ presented a systematic analysis of code generation algorithms for single-register machines, from linear to exponential.

In light of the above, it is not surprising that in most production compilers, the code generator is designed to produce code for one expression tree a time, and common subexpressions are not identified. Highly optimizing compilers, on the other hand, tend to be slow.

In our CGWS, we propose a code derivation algorithm which can accept IL patterns as complex as basic blocks and derive the "best" code for them. The meaning of "best" is: If the pattern is not "very complex", we can expect the optimal code within reasonable search time. But for very complex patterns, the search time could be prohibitively long to find the optimal code. This is inherent in optimal code generation for DAGs, indicating that some heuristics must be used in the search algorithm to cut less promising alternatives, resulting in the near-optimal code instead of the optimal one. We call this kind of code the "best" code.

The algorithm thus shaped may still be impractical to be used in the CG. However it is quite feasible for the CGG. Because in the CGG, the search time is a one-time investment, and therefore not so critical as in the CG. The latter uses faster methods to perform the pattern matching to gain in time efficiency, and is still able to produce reasonably good code, because the CGG has prepared the "best" templates for it.
4.2.2 Code derivation and AI Production Systems (AIPSs)

4.2.2.1 The Production Systems in AI

Many Artificial Intelligence problems can be formalized as a so-called Production System (AIPSs). An ordinary AIPS takes the state-space approach to problem-solving (/Nilsson 1982/, /Nilsson 1971/) consisting of three components:

- a set of problem states.

There is a state called the starting state $S$ - the initial configuration - and one state called the terminating state $T$ in which the problem is considered solved.

The set of states is called the state space consisting of all states reachable from the starting state by applying the rules to the states.

- a set of production rules.

A rule transforms one state into another state. We can regard rules as functions of states whose value are new states. However, each rule is usually defined only over a subset of the state space. That is, the applicability of a rule to a state must be submitted for test.

The problem is now formalized as finding any sequence of rules (or all sequences or the sequence optimizing some criterion, depending on what the problem poses), whose successive applications to $S$ and the derived states eventually transform $S$ into $T$. We call such a sequence a solution. Each rule application constitutes a piece of the solution (or outputs as side effects something which is the piece of solution).

In the case of searching for the optimal solution, we should define the solution cost $c$. Each rule application adds a piece of cost, and the total cost $c$ is accumulated along with the search process. The optimal solution is the one with the minimum $c$. The
definition of cost is a highly problem-dependent issue.

Sometimes we call a solution a path. If \((P_1, P_2, \ldots, P_n)\) is a path we call \((P_1, P_2, \ldots, P_m)\) \((m < n)\) a partial path. The depth of a path or a partial path is the number of \(P_i\) \(n \text{ or } m\).

- a control regime.

The control regime chooses which applicable rule should be applied at each step of the search and ceases the search when the termination condition is satisfied. The termination condition is posed by the problem. It could be when any path, or all paths, or the optimal path have been found.

There are three different control regimes used in AI Production Systems:

1. The first one is the irrevocable control regime which knows the problem so much that it can always choose the only or best rule to apply. Only very few simple problems can use this regime.

2. The second one is the graph search which keeps track of the effects of several sequences of applied rules simultaneously. In particular, the classical graph searching algorithm \(A^*\) was devised to find the minimum cost path. (See /Nilsson 1971/). /Kozlak 1981/ and /Kessler 1981/ both use \(A^*\) (modified slightly) in their CGWSs. The main problem with this approach is the possible space explosion (and hence time explosion) when the number of incomplete sequences which require simultaneous storage is too large. /Pearl & Kim 1982/ introduces three extensions of \(A^*\) which improve the search efficiency by relaxing the optimizing condition.

3. The third one is the backtracking control regime which chooses at each step of the search only one applicable rule, establishes a backtrack point for its choice, and reverts to this point to try other applicable rules if the subsequent computation encounters difficulty in producing a solution. This regime is attractive when the problem at hand is characterized by that: while finding the optimal solution is hard, it is easy to obtain some solution. In this case we
can get the first solution quickly, and go on to search for better ones by forced backtracking. The optimal code derivation problem surely has this property, that is why we choose control regime 3 in this work.

In the next section we present the algorithm in a skeleton form to realize control regime 3 efficiently. An instantiation of it in our code derivation problem will be treated in detail in section 4.2.3.

4.2.2.2 A backtracking algorithm for the state-space search

To realize the control regime 3 efficiently, we must make wise decisions about:

1. How to find all applicable rules at each step of the search;
2. In which order to try them;
3. How to implement the backtracking;
4. After the first solution has been found, we should backtrack also to find possible better ones. How can poor alternatives for this purpose be detected and cut as soon as possible?
5. To improve the search efficiency, we use heuristics to guide the search. Some heuristics guide the search without loosing the optimization such as those used in 2 and 4. They are called the admissible heuristics. However, to deal with NP-complete problems like optimal code generation, other heuristics must be used to keep the space and/or time consumption reasonable, at the expense of giving only a near-optimal solution. They are called semi-admissible heuristics.

Which decisions to make depends heavily on the particular problem at hand. So in the following skeleton form of the algorithm, we just specify the requirements and problem-independent answers, leaving the problem-dependent details to section 4.2.3.
ALGORITHM 4.1. Backtracking search of state-space

INPUT
The starting state S and the terminating state T;

OUTPUT
The near-optimal path P;

METHOD
1. Let CS denote the current state and set CS=S;
   Let CPP denote the current partial path and set CPP=();
   Let c denote the cost of the best solution found so far
   and set c=∞.

2. If any of the following heuristic conditions hold, go to
   to backtrack:
   (1) c1+c2>c. Here c1 is the accumulated cost on
       CPP, c2 is the estimated cost for the remaining
       part of the solution to which CPP will lead;
   (2) the depth of CPP exceeds a pre-defined limit;
   (3) other problem-dependent heuristics to cut the less
       promising alternatives.

3. Otherwise try to find the next applicable rule for CS. (The
   wise ordering of the applicable rules is another
   problem-dependent question.)

4. If any rule is found, derive a new state from CS by
   applying this rule. If the new state is not T, establish a
   backtrack point for the rule choice, append this
   application on CPP, mark the new state as CS and go to 2.

5. If the new state is T, a new solution NS has been found.
   Calculate its cost cc. If cc < c (NS is better), output it
   and set c=cc. Otherwise just ignore it (doing nothing).
   Then mark T as CS and go to 7 to backtrack (hopefully to
   find even better solutions).

6. If no rule is applicable to CS (the search meets
difficulty), go to 7 to backtrack.

7. Backtrack (one step): rejecting the most recently applied
   rule, delete the piece of solution and its cost, remark the
   father of CS as CS and goto 2.
   If there is no way to backtrack (the CS returns to the
   start state S), or the search time since step 1 exceeds the
   predefined limit, then

8. The search is terminated. The last output solution is the
   "best" one.
The advantages of this algorithm can be stated as follows:

First, it is practical. It allows the user to communicate as much knowledge as possible about his particular problem domain into the skeleton form to build a concrete version suitable to that domain. The knowledge can be put in the rules, the ordering of rules and the heuristics.

Secondly, there can be no space explosion. At any time in the search, we need to store only one partial path (the CPP), and the storage for this purpose can be managed conveniently in a stack manner when backtracking occurs. Thus finding all solutions requires the same amount of storage as finding one solution. We can ignore the space problem in our search algorithm and concentrate on reducing the search time.

Thirdly, the algorithm can be tuned to gain the desired trade-off between the time efficiency and the optimization in NP-complete problems. For example, first we can use only the admissible heuristics to guarantee the optimization. We may find the execution time unacceptably long. Then we can insert more and more semi-admissible heuristics into the step 2 until both the time efficiency and the near-optimization are acceptable.

Last but not least, this algorithm can be implemented very easily in PROLOG. Not only the overall structure of the algorithm can be specified in PROLOG in a natural way, several essential parts of the algorithm can also be carried out by PROLOG's built-in mechanisms. In particular, finding the applicable rule in step 3 can be implemented automatically by the unification process, and the backtracking in step 7 by PROLOG backtracking. The programmer need simply to indicate his willing to do these things at proper places. To implement these parts in an ordinary programming language would require a substantial effort.

The following skeleton PROLOG program is an example implementing algorithm 4.1. Note how concise it is and how much is done implicitly by PROLOG itself.
PROLOG program 4.1.

<1> PROBLEM:SOLVE(S, T, (), solution', 0, cost'), ZZ main task.
<2>    COST(c'), ZZ step 5.
<3>    cost' < c', ZZ step 5.
<4>    RETRACT(COST(c')), ZZ step 5.
<5>    ASSERT(COST(cost')), ZZ step 5.
<6>    OUTPUT(solution'), ZZ step 5.
<7>    BACKTRACK., ZZ step 7.
<8>    COST(m'), ZZ step 1.
<9>    SOLVE(x', T, cpp', (cpp'.s'),sofar',sofar'+co'): ZZ step 5.
<10>   RULE(x', T, s', co'), ZZ step 5.
<11>   ACCEPT., ZZ step 5.
<12>   SOLVE(x', y', cpp', solution',sofar',cost'): ZZ step 3.
<13>   RULE(x', z', s', co'), ZZ step 4 or 6
<14>   COST(c'), ZZ step 2.
<15>  sofar' + co' = c1', ZZ step 2.
<16>   <estimate c2'>, ZZ step 2.
<17>   c1' + c2' < c', ZZ step 2.
<18>   <depth within depth-limit>, ZZ step 2.
<19>   <other heuristics>, ZZ step 2.
<20>   SOLVE(z', y', (cpp'.s'), solution', c1', cost'). ZZ step 4.
<21> <definitions of RULE(oldstate', newstate', s', co')>

Only the statements in angular brackets are pseudo-code, requiring further refinement. They are problem-dependent and constitute the main efforts to implement a particular AIPS. (By the way, if we change = in line <8> into some predefined big integer as depth-limit, line <17> will implement line <18> as well, and line <16> can be deleted).
4.2.2.3 Code derivation formalized as an AIPS

Our code derivation problem fits the framework of AIPSs very well. Within the framework given above, the presentation of our code derivation method is much easier, and can be done in a symmetric manner. What we need to specify are:

1. We should have some idea of what the states of the code derivation problem are and choose a proper representation for them. Particularly we should identify $S$ and $T$.

2. Specifying the rules is the main job to extend algorithm 4.1 to a code derivation algorithm. Fortunately, the bulk of this task has been done in the machine operation part of the TMD described in chapter 3. Of course we should explain how to use the TMD and introduce other rules transforming complex IL patterns into primitive ones which can use the TMD directly.

3. Concerning the control regime, we should specify:
   - the ordering of rules;
   - how to estimate the remaining cost $c_2$;
   - set the depth limit and time limit;
   - other heuristics.

In section 4.2.3 we will answer these questions in detail.

4.2.3 Code derivation described in PROLOG

4.2.3.1 The state space

Essentially, the states in the code derivation problem are the processor states. However for the convenience of the presentation, we take another point of view.

First let us have a closer look at the pattern G. As we have
mentioned before, in our CGWS G is a basic block. In other words, G is a collection of IL trees over which a partial ordering is defined:

\[ G = (G_1, G_2, \ldots, G_i, \ldots, G_n) \quad (1) \]

\[ G_i = \{G_{i1}, G_{i2}, \ldots, G_{im_i}\} \quad (1 \leq i \leq n) \quad (2) \]

Here we use \((\ldots)\) for sequences and \(\{\ldots\}\) for sets. Each \(G_i\) is called a tree-set, and each \(G_{ij}\) is called a subgoal. In the above, \((1)\) says that \(G\) is a sequence of \(G_1, \ldots, G_n\). That is, \(G_1\) must be processed before \(G_2\) and so on. \((2)\) states the fact that, each \(G_i\) in turn is a set of simultaneous IL trees whose processing order is of no consequence semantically. That is, in each \(G_{ij}\) all source operands use the old values (before the code implementing \(G_i\) is executed).

We take \(G\) as the starting state \(S\). All new states reachable from \(S\) are derived by rule application (see section 4.2.3.2). Each new state is somewhat different (possibly with \(n\) decreased) from the old one, but remains a partially ordered collection of IL trees and therefore keeps the general form of \((1)\) and \((2)\). The terminating state \(T\) is the empty set \((n=0)\).
The figure 4.1 shows a pattern $G$ in DAG and the representation given above:

(a) the DAG

(b) partitioned into trees

$G = \{G_1, G_2\}$

$G_1 = \{G_{11}\}$

$G_2 = \{G_{21}, G_{22}\}$

$G_{11} = \langle - A (+ TN_1 (* TN_1 (+ TN_2 (+ TN_2 F)))) \rangle$

$G_{21} = \langle - TN_1 (- B C) \rangle$

$G_{22} = \langle - TN_2 (- D E) \rangle$

(c) the representation in our system

Figure 4.1 An example of pattern $G$
So eventually,

\[ G = \{ \{ \langle - A \rangle \left( + \ TN1 \ \langle * \ TN1 \ \langle + \ TN2 \ \langle + \ TN2 \ F \rangle \rangle \rangle \} \} \}
\{ \{ \langle - \ TN1 \ \langle - B \ C \rangle \} \} \{ \langle - \ TN2 \ \langle - D \ E \rangle \} \} \}

In our system the code derivation is performed top-down over IL trees (utilizing the operators as the index of rules and thereby accelerating the search). The code is derived in the reverse order as it would be executed. Also, when we say "the tree-set G1 is processed before the tree-set G2", we mean that the code for G2 will be executed before the code for G1.

4.2.3.2 Production rules

The code derivation process can be described as:

\[ G \rightarrow G^1 \rightarrow G^2 \rightarrow \ldots \rightarrow G^m = T = \{ \}
\]

rule-1 \quad rule-2 \quad rule-3 \quad rule-m

Some rules (INS rules, see section 4.2.3.2.2) eliminate a subgoal and output as a side effect the semantically equivalent assembly code and its cost. A sequence of the output code constitutes a solution of the code derivation problem.

In this section we describe various rules separately, leaving the rule ordering to section 4.2.3.3.

Our description is like this: for each class of rules we give the description of states over which these rules are applicable, the description of the derived states and the side effects of the their application (the assembly code, the cost of the code, compiler actions or nothing). The description is given first figuratively, then in PROLOG clauses. These clauses, put together, refine line (21) in PROLOG program 4.1.

We will use (1) and (2) as the general form of states. Because the processing order is inherent in (1), any rule acts upon G1, unless it
has been reduced to empty. In the latter case, rules are applied to \( G_2 \)
and so on until all \( G_i \) are reduced to empty (and we get the
terminating state \( T \)). So in the following description of rules, we
specify only \( G_1 \). Within \( G_1 \), usually \( G_{11} \) is the only subgoal affected
by rules, so we use "..." for the unaffected subgoals.

There are 6 different kinds of rules described in the
following subsections respectively.

### 4.2.3.2.1 The choice of evaluation order (EVO rules)

A set of simultaneous IL trees is in fact representing the set of
subexpressions of a certain big expression. The evaluation order of
these subexpressions is of no consequence semantically, but matters
very much for the code optimization. Our strategy is basically the
exhaustive search, trying every possible order. The EVO rules make
this choice.

Suppose \( G_1 = \{G_{11}, G_{12}, G_{13}\} \). The first time the EVO rule is applied
to \( G_1 \), it picks out \( G_{11} \), making a new state \( \{G_{11}, \{G_{12}, G_{13}\}\} \). Here \( G_{11} \)
is not in \( \{} \). This means, \( G_{11} \) will be processed first. Whenever the
backtracking is up to here, the next subgoal will be selected, the
previously selected one will be put back to the remaining set. In this
way, we can try every possible evaluation order.

\[
\begin{align*}
&\{G_{11}, \{G_{12}, G_{13}\}\} \quad \text{or} \\
&\{G_{11}, G_{12}, G_{13}\} \quad \text{---} \\
&\text{EVO rules} \\
&\text{(nothing)}
\end{align*}
\]

Referring to the PROLOG program 4.1., we enrich line <21> with the
following definition of RULE (_, _, _, _):

\[
\text{RULE}(G_1', G_1'', (\), 0): \text{CHOICE}(G_1', G_1'').
\]

The literal \( \text{CHOICE}(G_1', G_1'') \) which builds a new state \( G_1'' \) from \( G_1' \)
is defined recursively:
CHOICE({x'.y'}, {x' {y'}}).
CHOICE({x'.y'}, {z' {x'.y'}}):CHOICE({y'}, {z' {y'}}).

These two short lines of PROLOG implement the intended function of EVO described above.

Of course we must avoid applying the EVO rule in succession. After the EVO rule application to G1' we must apply other kinds of rules to G1'' trying to reduce the chosen subgoal instead of reordering again. Reordering makes sense only when backtracking occurs. This will be taken care of automatically: because the description of G1'' is (x' {...}) to which the EVO rule could not be applied at all. On the other hand, as we can see below, in the description of all other rules, RULE(oldstate',_,_,_), the description of oldstate' is always like (x' {...}).

Exhaustively applying the EVO rule in backtracking results in combinatorial explosion. Some semi-heuristics must be used to cut less promising permutations. They will be given in section 4.2.3.3.

4.2.3.2.2 Instructions [INS rules]

As we just mentioned, after an application of an EVO rule, we have chosen one subgoal to process first. If we are lucky, there may be a machine instruction implementing it directly and reducing it to empty. The INS rules express this reduction:

\{(G11 \{G12 G13\}) \rightarrow {G12 G13}\}

INS rule
(code for G11)
(cost of code)

In PROLOG:

RULE((G11' \{rest\'}), \{rest\'}, code', cost'): CODE(G11', code', cost').
Remember, the definitions of CODE(pattern', code', cost') have been included in the TMD to describe the machine instructions. We need not specify anything machine-dependent here!

But the situation is somewhat more complicated. Because the code sequence is derived in the reverse order, any locations whose original values are used in the INS rule application cannot be used as destination locations of {rest'}. So the PROLOG program maintains a data structure called the destination-prohibited location set (the DPL). Later, any attempt at assigning new values to the locations in the DPL is regarded as failure and causes backtracking.

However, suppose we have the following pattern G,

\[
\{(\leftarrow A (\ast A B)) \leftarrow B (\ast B A))\} 
\]

When we use (ADD B A) to reduce the first tree, B is put into the PDL set, and the second tree becomes unsolvable; choosing the second tree to be reduced first cannot help either. Introducing a Temporary Name (TN) is necessary to break this dilemma.

4.2.3.2.3 Fetch/store subtargeting (SUB rules)

If there is no INS rule which can eliminate G11 (because it is too complicated or in the dilemma above), we introduce one or more Temporary Names (TNs) representing some specific locations (such as the ACC in single-accumulator machines) or user-allocatable registers or temporary memory locations to hold intermediate results. Supposing \(a, \beta, \gamma, \ldots\) are disjoint subtrees of G11, we denote this fact by \(G11 = a[\beta, \gamma, \ldots]\). Then it is equivalent semantically to \(a[TN1, TN2 \ldots]\) and \(\leftarrow TN1 \beta, \leftarrow TN2 \gamma, \ldots\). The first one must be processed before the other assignments and hopefully will be eliminated by an INS rule. On the other hand, the evaluation order among those assignments and G12 \(\ldots\) can be chosen freely:

\[
(a[\beta] \{G12 \ldots\}) \rightarrow (a[\beta] \{G12 \ldots\}) \\
\text{SUB rule} \\
\{\text{alloc}(TN)\} 
\]
(For readability only one TN is involved here). Here we are essentially performing the register allocation. If αβ is an assignment α:=β, the rule is called the store subtargeting, in all other cases, it is a fetch subtargeting.

If the TN must be a special location (such as the A register in ND-100) to make α[TN] solvable, but at the same time the special location is "busy" (used as the destination of some remaining subgoal of [G12 ...]), a conflict arises. In addition to the ideal case above we should have another rule to detect these conflicts and generate the necessary subgoals for register spilling.

4.2.3.2.4 Flow-control decompositions (DEC rules)

These rules are concerned with flow-control structures. In our code generation model the IL contains high-level flow-control operations such as (IF b' THEN s') or (WHILE b' DO s'). Here b' is any Boolean expression, and s' is any operation. On the other hand, machines usually have only primitive jump instructions and/or skip instructions. Before using these instructions we must decompose the high-level structures into a sequence of primitive-ones. DEC rules do this job, and the sequence produced has a definite processing order.

Because we have no intention to do optimization beyond a basic block, we can assume that these high-level flow-control operations stand alone. That is, such an operation alone constitutes the original pattern 6.

This group of rules covers all possible decompositions to explore all (but the variety is very limited) hardware capability for flow-control operations. Let us take the example of (IF b' THEN s'). There are two possible decompositions for it:

(1) (IF_THEN b' s') -----------------> (s' (→ (NOT b') (+ PC m)))

DEC rule
(nothing)
"->" is the conditional jump operator: (-> b' l') means if the result of the evaluation of b' is TRUE, go to the location described by l'. Here this location is described by the expression (+ PC m) (m is the number of bytes occupied by the code implementing s'). Some machines like ND-100 do have the skip instructions implementing this pattern (in the case of ND-100, m=1).

(2) (IF_THEN b' s') -------------> (s' (-> (NOT b') (l' #RM)))
DEC rule
(compiler-generated
label l', after
the code of s')

The ordinary conditional branch instructions can implement the second subgoal.

After this decomposition, we first process s' as we do for G1, and the second subgoal takes the position of G2. To process a subgoal of the form (-> b' l'), we first try the INS rules. But only a very simple b' can fit directly the patterns of jump instructions. So usually we should first evaluate b' and set the result into condition code (if any), or transform b' into the normal forms supported by the target machine. In this process the SUB rules can do some help, but we need new rules for the evaluation or transformation, the rules expressing the usual Boolean laws.

4.2.3.2.5 The arithmetic and Boolean laws (AXI rules)

The situation just mentioned is only one example of the applications of the AXI rules (expressing the tree equivalence AXIoms). If some arithmetic operation does not appear in the target machine, neither INS rules nor SUB rules can derive code for it. Again, we need AXI rules to transform it into an equivalent operation or operations which are supported by the target machine. Furthermore, even for 'normal' operations the application of these AXI rules sometimes result in
better code!

A general AXI rule can be described as:

\[(G11 \{G12 \ldots\}) \rightarrow (Q11 \{G12 \ldots\})\]

\[AXI \text{ rule}\]
\[\text{nothing}\]

Here Q11 is semantically equivalent to G11.

In PROLOG, we have:

\[\text{RULE}((G11' \{\text{rest}'\}), (Q11' \{\text{rest}'\}), (\), (0)):\]
\[\text{AXIOM}(G11, Q11).\]

The definitions of AXIOM(\_\_) cover as many equivalence axioms as necessary. The search space defined by these axioms is a combinatorial explosion. So we must have some semi-heuristics to control the application of this kind of rules. And this is again one of the topics of section 4.2.3.3.

4.2.3.2.6 Common subexpression merging (CSE rules)

Every time we choose a subgoal to reduce, we first attempt at possible common subexpression (cse) merging:

\[\text{(\text{- x cse}) \{\ldots (\text{- y ces}) \ldots\} \rightarrow (\text{- x y}) \{\{\text{- y cse}} \ldots \ldots\}\}
\[\text{or(\text{- y x}) \{\{\{- x cse}\} \ldots \ldots\}\}
\]

\[\text{CSE rule}\]
\[\text{(nothing)}\]

All the occurrences of cse will ultimately be merged into one.
4.2.3.3 The control regime

Algorithm 4.1 and PROLOG program 4.1 outlines the control regime. However there are some very important problem-dependent issues left for further refinements. In this section, we complete the description of the control regime for our code derivation problem.

4.2.3.3.1 The cost estimation

The methods to estimate the cost for an IL expression are machine-dependent issues and described in the TMD by the literal ESTIMATE(\text{...}). (See section 3.4). So, the refinement of line (16) in PROLOG program 4.1 simply is:

\text{ESTIMATE}(z', c2'),

here \(z'\) is the set of the remaining subgoals, \(c2'\) is its estimated cost.

In the TMD the estimate \(c2'\) is the underestimate. So the following line (17) \((c1' + c2' < c')\) cuts those solutions whose cost will be at least as big as the cost \(c'\) of best solution obtained so far. This before-hand cutting improves the search efficiency and does no harm to the code quality.

Sometimes this method can only reduce the search effort moderately. To improve the search efficiency further we must relax the requirement of optimal code somehow, and accept the concept of semi-admissible heuristics. For example, we can increase the cost of the previous solution \(c'\) by a factor of, say, 10 percent before the comparison. In this way we can accelerate the search process while keeping the deviation from the optimal solution at no more than 10 percent.
4.2.3.3.2 The search limits

Because of the incompleteness of depth-first search, we must set up a depth limit. When the depth of the current partial path exceeds this limit, we discard it. On the other hand because of the NP-completeness of optimal code derivation, we have to set a time limit as well. In doing so we lose the control over the deviation of the solution found from the optimal one. But they are inevitable by the nature of the problem at hand.

Fortunately, these limits are parameters and can be adjusted easily to trade off between the time efficiency and the code quality as mentioned before.

4.2.3.3.3 The rule ordering

The philosophy behind our rule ordering strategy is that, we want to get at the first solution as quick as possible, and this solution should be as good as possible. The ordering among different classes of rules is important for quick derivation of the first solution, while the rule ordering within the same classes has effects on the quality of the first solution.

First, the ordering among different classes of rules is expressed in Figure 4.2. The arrowed goal-flow shown in this figure is valid only when we are searching for the first solution. To search for more solutions, the backtracking controls the flow. As the figure shows, in searching for the first solution, if there is an applicable INS rule, we apply it without delay except the possible cse merging. In other words, the first solution will be found as quickly as possible.
Figure 4.2 The ordering of rule classes

In PROLOG the implementation of the above scheme is easy: To arrange the clauses describing different classes of rules in the order shown in the figure will do the most. The only other problem is to submit a new state to a certain point shown in the figure. This is taken care
of automatically: we have met this in section 4.2.3.2.1.

On the other hand, within each class, the rule ordering is also important, specially for the quality of the first solution:

1. Within INS.
   In the TMD, the clauses representing instructions are indexed by the operators and arranged in such a way that the instructions denoting special cases of operands of more general instructions are placed before the latter (e.g. INC before ADD, CLR before MOVE in PDP-11). Furthermore the clauses representing one operand class are arranged by increasing cost per node in the location trees. The effect is to first use the access mode which can "bite off" the largest possible portion of the IL pattern at each reduction step.

2. Within DEC.
   For the clauses representing flow-control decompositions, those attempting to use the (less expensive) skip instructions are placed before those of jump instructions.

3. Within EVO.
   It presents more challenge to choose the "favorite" subgoal to reduce first. We set up the following criteria for the "wise" choice:

3.1. The locations whose values are used in the computation of the chosen subgoal are not used as destination locations for the remaining subgoals. For example if the set is

\[
\{ (\neg A \ (\neg A \ B)) \ (\neg A \ (\neg C \ D)) \}
\]

we choose the second subgoal to reduce first.

3.2. The expression in the chosen subgoal is not a (true) subtree of any remaining subgoal's. Here we are simply trying not to loose the possibility of common subexpression merging.

3.3. In some machines some special location is required for store subtargeting (e.g. the ACC in single-accumulator machines).
If there is a subgoal whose destination is the special location, choose it. Because if we choose another subgoal, we should perform the store subtargeting for it and we need the ACC which is already the destination of some other subgoal: a lot of extra data moving has to be done. This condition encourages keeping to reduce one subgoal until it has been eliminated totally.

Ideally, we choose a subgoal satisfying all these conditions to reduce first (and in this case we can even cut other evaluation order choices, see section 4.2.3.3.4). Unfortunately in many cases these conditions contradict each other (this kind of conflicts must be handled by the exhaustive search). If so, we choose one satisfying these conditions partly (as many as possible).

In this way the first solution is made as good as possible. And when we go on to find even better solutions we can with more confidence cut search paths as soon as they seem to lead to no better solutions: We have got a fairly good solution already.

4.2.3.3.4 Other heuristics

After the first solution has been found, we backtrack to find more (hopefully better). As we mentioned before, semi-heuristics are inevitable here to keep the search time reasonable in the practical sense. Intuitively, they are valuable for reducing the search efforts but doing very little harm to the code quality. But it is very hard to do any quantitative analysis. We'd better evaluate these heuristics by experiments.

1. The store subtargeting is performed only when necessary in spite of some rare cases where its application can give better code.

2. As mentioned earlier, if we can luckily choose one "favorite" subgoal satisfying all the three conditions to reduce first, we don't consider other evaluation orders involving this subgoal. The reasoning is: given that reducing it first causes no extra data
moving (i.e., everything done is necessary), putting it back would not give better solutions.

3. The heuristics we employ in applying the arithmetic and Boolean axioms are as follows: for any subgoal, each applicable axiom (i.e., whose first parameter matches the subgoal) is applied once, and the new state created by this application cannot use the same axiom in either direction. Otherwise we will get into infinite loops like

\[
x' \rightarrow x' + 0 \rightarrow x' + 0 + 0 \rightarrow ...
\]
or
\[
x' \rightarrow x' + 0 \rightarrow x' \rightarrow ...
\]

Mainly we use these axioms in order to recover from the hopeless situations involving irreducible operators rather than to do code optimization.

4.2.4 The experimental results

The proposed code derivation algorithm has been implemented in PROLOG and tested for non-trivial IL patterns. /Kozlak 1981/ presents a code generator generator called the ACG and tests it for a benchmark of 14 basic blocks. We have tested our CGG using the same benchmark as the IL patterns. The comparison and comments are the following:
<table>
<thead>
<tr>
<th>Aspects</th>
<th>ACG</th>
<th>CGG</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>PDP-11/50</td>
<td>ND-100</td>
<td>similar</td>
</tr>
<tr>
<td>target machines</td>
<td>A,B,C, (three ideal machine models)</td>
<td>A,B,C, PDP-11, PDP-8, NO-100, MOTOROLA-68000</td>
<td>A,B,C are model machines We deal with real machines</td>
</tr>
<tr>
<td>tool</td>
<td>ordinary progr. language</td>
<td>PROLOG AI lang.</td>
<td>PROLOG can be improved easily by a factor of 10 in execution time</td>
</tr>
<tr>
<td>algorithm &amp; performance</td>
<td>A*</td>
<td>depth-first</td>
<td>1. ACG takes pains to reduce the storage requirement, CGG needs not bother; 2. timing figures are in the same range: 1sec - 1min for one IL pattern. This means, CGG can be 10 times faster than ACG if PROLOG itself is improved</td>
</tr>
<tr>
<td>code quality</td>
<td>near-optimal</td>
<td>better</td>
<td>CGG gets better code in 4 out of 14 examples. Others are the same.</td>
</tr>
</tbody>
</table>

This comparison convinces us that, our ALGORITHM 4.1 and the heuristics used are competent to produce near-optimal code sequences for complex basic blocks, within reasonable time and causing no space problem. It is also showing that, for the CGG, PROLOG is a good tool not only of specification but also of implementation.

Appendix 5 gives three code derivation examples for complex IL patterns.
4.3 Pattern selection and MT building

The term "code generator generation" means to build the MT which drives the CG. In other words, the CGG is really a MT generator. To do this, we must start with a set of IL patterns. For each pattern we should derive a code sequence. To improve the overall code quality, we should make better pattern selections as well as improve the quality of the code sequences. The latter was the topic of the previous section (4.2). Now we turn to the former. In this section we propose a pattern selection scheme, and present the current implementation of the MT building.

4.3.1 A pattern selection scheme

As mentioned before, optimal pattern selection in the generation of the MT is a difficult problem. Let's first examine how it was treated in the previous literature.

In /Newcomer 1975/ both the selection of the pattern trees and the use of the templates in the code generator are left for further research, let alone the optimal selection. /Cattell 1978/ selects all IL operators as the patterns, but no combination of IL operators have been included in the MT except those which can be implemented by a single machine instruction. /Kozlak 1981/ suggests that the IL operators should be as high level as the source operators. /Slanville 1977/ and /Ganapathi 1980/ both assume the one-to-one correspondence between the IL operators and the machine operators, so their patterns are exactly the machine instructions.

In summary, so far no CGWS select patterns more complicated than a single IL operator. /Kozlak 1981/ seems to go further. However, if we examine some source languages, we will find that most really complex source operators (such as the assignment between strings in PASCAL) can be efficiently implemented only by exotic instructions (see /Morgan 1982/), but these exotic instructions are not included in the machine model of /Kozlak 1981/. Other source operators chosen as the patterns usually cannot provide the CGG with non-trivial sets of
assignments as /Kozlak 1981/ claims.

Ideally, we would like a scheme to automatically determine what special combinations of IL operators should go into the MT to improve the overall code quality without the tedious compile-time case analysis. The task is really difficult in its full generality. However if we confine ourselves to particular IL programs, which represent the most frequent application problems of the CGWS, it can be achieved to a certain extent. In our CGWS, the CGG can derive the near-optimal code sequence for a basic block. Now we propose a scheme to incrementally build a set of patterns which ultimately represents the most important basic blocks of the programs under consideration. We will discuss how this can be done, although a full implementation has been left to future research.

First we make a necessary minimum set of IL patterns and the Pattern File (PF) contains only these patterns. The next subsection will describe what patterns are in this minimum set and how to build the MT from it. The MT created thus is the kernel which guarantees that the CG can work. If we need to improve the overall code quality for a particular program, we should tune the MT to better suit the problem under consideration. The tuning is based on run-time statistics:

1. Divide the program into basic blocks at IL level.
2. Measure execution frequency $f$ of each basic block.
3. Select the basic blocks with the maximal value of $f$.
4. Make IL patterns corresponding to the selected basic blocks.
5. If there are no such patterns in the PF, add them to the PF and call the CGG to derive new templates. Thus the MT is tuned, and the CG can now produce better code for the program.

It should be noted that the MT is "learning". Each time the tuning process has been done, it is more "intelligent" in the sense that the CG using it may produce better code even for other programs.
4.3.2 The current implementation of MT building

The current implementation of MT building is essentially to build the kernel of the MT. The task is mainly done by the CGG with limited human help:

1. Prepare the PF by hand. The PF consists of patterns. Each pattern is a PROLOG term and the PF is a sequence of terms. The PROLOG program CGG can read patterns one after another from the PF. There are the following different patterns:

1.1. For each arithmetic or (bit-wise) Boolean operator (binary) \( \alpha \) of IL, there is a pattern like \( (<- \text{dst} \ (\alpha \ \text{src1} \ \text{src2})) \). Here src1, src2 and dst are parameterized operands. For unary or trinary \( \alpha \) there are similar patterns.

1.2. For each relational operator \( \beta \), there is a pattern like \( (<- \ (\beta \ \text{src1} \ \text{src2}) \ 1) \). Again src1, src2 and 1 are parameterized operands.

1.3. There are patterns like \( (<- \text{dst} \ \text{src}) \), for each pair of distinct access modes dst and src. They are necessary for the subtargeting in the CG (see next chapter).

1.4. For each existing pattern, add more patterns with the same operator but with special operands before the general one. For example, starting from \( (<- \text{dst} \ (+ \ \text{src1} \ \text{src2})) \) we derive new patterns \( (<- \text{dst} \ (+ \ (<\text{dst}) \ \text{src2})) \) and \( (<- \text{dst} \ (+ \ (<\text{dst}) \ 1)) \). For high-level flow-control operators (IF, WHILE ...) and the Boolean operators (AND, OR, NOT), because their processing is basically machine independent, we leave it to the CG and need not generate corresponding patterns.

2. Expand the PROLOG code for Algorithm 4.1 to do reading from the PF and writing to the MT. Joining the resulting program and the TMD, we have got the complete CGG program which behaves like this:

2.1. First it copies the virtual part of the VMD to the MT, which is the machine-dependent information the CG needs to perform tasks other than instruction selection.

2.2. Then it copies the operation part of the TMD to the MT, deleting the cost information (which is not necessary for the CG). Each instruction copied thus serves a simple template.
2.3. Now it starts to build more complex templates. It reads one pattern from the PF, checks if it is included in the MT already. If not, it derives the near-optimal code sequence and builds a PROLOG clause similar to the instruction description. This clause is a new template and is written into the MT. This process is repeated until the end of the file PF is reached.

3. To complete the MT, human help is needed again. We write the code for operations on virtual data types, for procedure linkage, for parameter passing and for whatever not included in the MT but necessary for the CG to work in the MT driven way. For each of them, write a template and put it into the MT. Note that the templates and the MT are both human and machine manipulatable.

We see from the above that the MT is essentially an extension of the TMD and is a sequence of PROLOG clauses. The CG is an uncompleted PROLOG program. To make a complete code generator we simply join the MT to the CG in the same way we did for the TMD and CGG.

Appendix 2 is the minimum PF, and Appendix 4 gives some extractions of the MTs produced by the CGG from the PF for PDP-11 and ND-100. Appendix 5 contains three examples of code derivation for complex IL patterns (non-trivial basic blocks).
5. **CODE GENERATION**

The Code Generator (CG) proposed in our CGWS takes tree-like Intermediate Language (IL) programs, and produces as output symbolic assembly code (other output forms like BRF could be produced as well, we choose assembly code in this work just for convenience). The CG is machine-independent, all the machine dependent information required by the CG is specified in the Mapping Table (MT), which is the output of the Code Generator Generator (CGG) as mentioned in the previous chapter.

The CG has the following strong points:

1. The IL is an intermediate language on a quite high level, the translation from a source language to this IL is therefore easy.

2. The CG is a complete retargetable code generator in the sense that it covers a very large part of the synthesis phase of compilation such as storage allocation, data accessing, instruction selection, register allocation and procedure handling, and they are all performed in a machine-independent way, driven by the MT. Other phases (except the machine-dependent optimization) of the compiler need not bother about machine-dependent issues.

3. Data accessing consists of two steps. First, each name occurring in the IL action part is replaced by a machine-independent (virtual) access mode (specifying the necessary address calculation to access this object). Then the mapping of this virtual access mode to the actual mode supported by the target machine is considered as normal instruction selection. In this way we get a more unified view of code generation.

4. Instruction selection uses the same general tree pattern matching approach as the CGG. However, this time the matching is performed on trees in a fixed top-down, left-to-right order and without recognition of common subexpressions instead of performing an
exhaustive search. This simple strategy still produces quite good code because of:

- some heuristics solving the multiple matching problem;
- the MT including any necessary IL patterns and the "best" code;
- full exploitation of the target machine's addressing capability.

In the following sections we present various aspects of the CG, with the emphasis on the points mentioned above, ending with a complete code generation example. Appendix 6 is the PROLOG program for the CG.

5.1 The Intermediate Language (IL)

The most important factors that have to be taken into consideration in designing an IL used in a CGWS are:

- form: tuples / prefix form / postfix form / trees / ... ;
  symbolic / binary.
- level: the position of IL between the source and target languages.
- flexibility: how to overcome the possible source / target / implementation dependencies.

In this section, we introduce the IL used in this CGWS, showing the design decisions about these factors. The complete definition of the IL is in Appendix 1.

5.1.1 The form of IL programs

Because both the CGG and the CG in this CGWS take a general tree pattern matching approach, the IL also takes the tree form. We can consider the whole program as one big tree or each statement as a subtree. The tree form provides the freedom of processing order, which is vital for the CGG as was noticed in the previous chapter. In the CG, because of the fixed evaluation order, the tree form is processed as the linear prefix form. Still, this tree form leaves
the room for future improvement of the CG, adding an optimization phase to it, for example.

In the implementation, a prefix parenthesized notation is used for trees, and PROLOG lists provide an ideal data structure for it. The whole IL program is a sequence of PROLOG terms, which in most cases are lists. An IL program is in a source text file, from which the PROLOG program CG can read PROLOG terms one after another.

The following is a BNF-like syntax of the top level structure of the IL:

```
<IL-program>::=MAIN <body> MAINEND
<body>::=<var-def-part> <proc-def-part> <act-part>
<var-def-part>::=DEFINITION { <def> }* DEFEND
<proc-def-part>::={ <proc-def> }*
<act-part>::=ACTION { <act> }* ACTEND
<proc-def>::=<proc-head> <body> <proc-tail>
```

The undefined non-terminals <def>, <act>, <proc-head>, <proc-tail> will be described in the following subsections.

5.1.2 The level of the IL

The IL is machine independent and on a high level characterized by three features:

1. The definition part is included

   After the translation from source text to IL text, the names used as operands remain unchanged except disambiguating locations and their contents (if an occurrence of a name represents the value, it is prefixed by the dereference operator <>). This means that the storage allocation is regarded as part of the code generation task, and the declaration part of the source program must be included in the IL program.
(def)::={id> (datatype> (amount> (allocation-type>))
<datatype>::=INT | BOOL | BYTE | POINTER | ...
<amount>::=integer
<allocation-type>::=<empty> | S | ...

Here the vertical bar means alternatives, and ... is an informal notation for "more to be implemented". The above BNF says that each <def> is specified as a PROLOG list. <id> gives the the name of the defined identifier; <datatype> is the simple datatype (INT, BYTE etc. are the ones supported by the present prototype CG); <amount> is 1 for simple variables and the number of elements for arrays; and <allocation-type> is empty for stack allocation and S for static allocation.

For example,

(i INT 1) defines i as an integer variable on the run-time stack;
(a BYTE 10 S) defines a as an array of 10 bytes in the static area.

2. High level operations are included:
The action-part consists of a sequence of statements, the non-terminal <act> describes the statements:

<act>::={op> { <operand> }*}
<operand>::=id> | constant> | <act>

The above BNF says that an IL statement is described by a PROLOG list. Here the <op> can be any source operator, no decomposition has been done at this stage. The design principle of our CGWS is that, the CGG can receive very high level operators as input and derive the "best" code sequences for them.

For example, an array reference a[j] is translated into the IL expression (INDEX a (<>j)) only syntactically. The same is for other high level operators: just translating the infix source structures into the prefix ones in the IL and adapting them to the PROLOG syntax. For example, the conditional statement in a source language may look like IF <bool> THEN <st> ELSE <sf> FI, and the IL version would be
(IF_THEN_ELSE bool' st' sf'). Other high-level control operators are
(IF_THEN bool' st'), (WHILE bool' s') or (REPEAT s' bool'). Here
bool', s', st', sf' are PROLOG variables. bool' can be any Boolean
expressions, while s', st' and sf' can be any statements.

As we can see from the BNF, the IL allows the nesting of expressions
and statements. In the examples above, bool', st', sf', s' can be any
<act>, provided the semantics are correct.

3. The IL is a structured language:

Besides the nesting of expressions and statements mentioned above,
there are two more structural features. First, the <act> can be a
sequence of statements, which are processed one after another by the
CG:

<act>::=({<act> ; <act> })*
++ meaning repetition
++ one or more times.

The other structure is the procedure definition. An IL program
keeps the textual structure of the source program including procedure
nesting. Each procedure definition consists of three parts: the head,
the body and the tail. The body is the same as in the main program.
The head gives the procedure name, the type of return value, and the
description of virtual parameters. The tail is just a tag: PROCEND.

<proc-head>::= (PROC <id> <return-type> <parameter-list>)
<return-type>::= VOID | INT | BYTE | BOOL | POINTER | ...
<parameter-list>::= <para> | (<para> ; <para> )+
<para>::= <def> | NIL
<proc-tail>::= PROCEND

The VOID stands for "no value returned", and the NIL for
non-parameter procedures.
The following is an example of a procedure definition:

(PROC  p  INT  ((i INT 1) ; (j INT 1)))

DEFINITION
  (k INT 1)
DEFEND
ACTION
  (<- k (+ (<> i) (<> j)))
  (<- p (<> k))
ACTEND
PROCEND

5.1.3 The flexibility of the IL

While it is machine-independent, the IL cannot get rid of source language dependency and implementation dependency entirely. There are two ways to eliminate (or reduce) this dependency.

If the issue concerned has a very small set of alternative solutions (e.g., some implementation decisions such as the way to access non-local data), the CG can try to provide all of them as options, and the IL will not depend on any particular solution. However, some issues involve a great deal of possibilities, such as the variety of operators in various source languages. In these cases, the IL can be extended to various dialects suitable to various source languages.

The design of this IL provides means to reduce source dependency in both ways.

5.1.4 An example

The IL program below is the binary search problem, showing many features of the IL. This program will serve as the main example in the following sections about various code generation tasks. An equivalent MARY program is given before the IL version for reference.
A MARY program for binary search

BEGIN

(:10) INT item;
INT subs,
result;

PROC b_s=\%RECURS(INT numb) INT :
BEGIN

INT b_s,
  low,
  high,
  middle;

1:=low;
10:=high;
DO
  (low+high)//2:=middle;
  IF item(middle)<=numb
  THEN middle+1:=low;
  ELSE middle-1:=high;
  FI;
OD
UNTIL low>high;
IF (log+1)>high
THEN middle:=b_s;
ELSE 0:=b_s;
FI;
b_s \% in register A

END; \% of PROC b_s

1:=subs;
DO
  subs:=item(subs);
  subs+1:=subs;
OD
UNTIL subs>10;
b_s(5):=result;
END FINISH
The hand-translated IL program for binary search
(the identifiers in upper case are the reserved words)

(1) MAIN
(2) DEFINITION
(3) (item INT 10)
(4) (subs INT 1)
(5) (result INT 1)
(6) DEFEND
(7) (PROC b_s INT (numb INT 1))
(8) DEFINITION
(9) (low INT 1)
(10) (high INT 1)
(11) (middle INT 1)
(12) DEFEND
(13) ACTION
(14) (<= low 1)
(15) (<= high 10)
(16) (REPEAT { (<= middle ( ( <= low ( <= high) ) ? )
(17)         (IF_THEN_ELSE
(18)         (LEQ (<= (INDEX item (<= middle))) (<= numb) )
(19)         (<= low (+ (<= middle) 1))
(20)         (<= high (- (<= middle) 1)) )
(21)         (GTR (<= low (<= high))))
(22)         (IF_THEN_ELSE (GTR (+ (<= low) 1) (<= high))
(23)             (<= b_s (<= middle))
(24)             (<= b_s 0))
(25) ACTEND
(26) PROCEND
(27) ACTION
(28) (<= subs 1)
(29) (REPEAT { (<= (INDEX item (<= subs)) (<= subs) )
(30)          (<= subs (+ (<= subs) 1)) )
(31)          (GTR (<= subs) 10))
(32) (PARA 5) % these three lines
(33) (CALL b_s) % are equal to
(34) (<= result RETURN) % result := b_s(5)
(35) ACTEND
(36) MAINEND
5.2 Machine-independent storage allocation

The storage allocation can be performed in a machine-independent way. For example in /Holt 1981/, it is parameterized by machine-dependent data. In our CGWS, all machine-dependent and implementation-dependent information is isolated in the MT.

The present CG supports static allocation and stack allocation. In the case of stack allocation, the task is to decide the offset of the location allocated to each name from the base of the activation record. In the MT, the following implementation-dependent information is included for storage allocation: (see sections 3.2.1 and 3.2.3.2)

1. the layout of the activation record, specially the relative position of data and the base: DATASTART(sign', abs').
2. the frame direction: FRAME_DIRECTION(d'). (d' is UP or DOWN).
3. the positioning of the parameters.
4. the size and alignment of each simple IL datatype: DATA_TYPE_CONVERSION(ILtype', machinetype', size', align').

From this information we can decide the absolute value and the sign of the offset for each name declared in <def> or <proc-head>. Algorithm 5.1 describes the machine-independent stack allocation for the local data of a procedure.

ALGORITHM 5.1 Machine-independent stack allocation

INPUT <var-def-part> of an IL procedure with static-level 1

OUTPUT the offset of each data definition in <var-def-part>

METHOD 1. Call FRAME_DIRECTION(d') to get the value of d' (UP or DOWN), call DATASTART(sign', abs') to get the values of sign' (+ or -) and the value of abs' (the absolute value of the offset of the first local data position). Set the Currently Available Position CAP=abs'.

2. Read the next <def>. Suppose it is (name ILtype amount). Call DATA_TYPE_CONVERSION(ILtype,machinetype',size',align') to get the machinetype' and its size' and align'. Here ILtype is the input of DATA_TYPE_CONVERSION(,_,_,_), while the other three terms are the output represented by logic
variables.

3. Round up CAP for align' and change CAP to the result. This is the absolute offset of name.

4. Insert the following information as the result of the allocation of name (remember PROLOG allows the dynamic modification of its programs):

OFFSET(name, l, CAP, sign', ILtype).

Here l and CAP are integers, sign' has been bounded to + or -. This result will be used in the data accessing phase.

5. Decide whether CAP should be increased or decreased according to the following decision table:

<table>
<thead>
<tr>
<th>d'</th>
<th>sign'</th>
<th>sign''</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOWN</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>DOWN</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>UP</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UP</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Set CAP=CAP sign'' (size' * amount).

If CAP turns out negative, take the absolute value as CAP and change sign' to the opposite sign.

If there are more <def>, goto 2.

For example, in the b_s program, after line (11) is read and processed, the results of storage allocation on different (virtual) machines are as follows:
Target machine: PDP-11

Virtual machine: (the relevant implementation decisions in MT)
1. the base points to the beginning of the activation record,
2. the first location for data is 6 bytes away from the base,
3. the stack grows downwards,
4. INT takes 2 bytes,
5. the parameters are pushed on the stack before the linkage-locations,
6. the return value (if any) occupies the first location of the local data.

The output of the allocation: inserting the predicate OFFSET mentioned above for each data definition and parameter.

```
<table>
<thead>
<tr>
<th>000000 (down)</th>
</tr>
</thead>
<tbody>
<tr>
<td>head of item</td>
</tr>
<tr>
<td>item's elements</td>
</tr>
<tr>
<td>subs</td>
</tr>
<tr>
<td>result</td>
</tr>
<tr>
<td>param:numb</td>
</tr>
<tr>
<td>STATIC LINK</td>
</tr>
<tr>
<td>DYNAMIC LINK</td>
</tr>
<tr>
<td>RETURN ADDR</td>
</tr>
<tr>
<td>b_s</td>
</tr>
<tr>
<td>low</td>
</tr>
<tr>
<td>high</td>
</tr>
<tr>
<td>middle</td>
</tr>
</tbody>
</table>

0  <-- base of main  (level = 0)
-2
-4
-6  first location for local data

:  :
:  :
-20

+2  push parameters before linkage-locations

0  <-- base of b_s procedure (level = 1)
-2
-4
-6  location for the return value

:  :
-8
-10
-12
```
Target machine: NO-100

Virtual machine:
1. the base points to the middle of the record (of 256 words),
2. the first location for data is 125 words away from base,
3. the stack grows upwards,
4. INT takes 1 word,
5. the parameters are after linkage-locations,
6. the return value is on the first data location.

The output of the allocation: inserting the predicate OFFSET mentioned above for each data definition.

```
<table>
<thead>
<tr>
<th>177777 (up)</th>
<th>-128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-127</td>
</tr>
<tr>
<td></td>
<td>-126</td>
</tr>
<tr>
<td>head of item</td>
<td>-125 first location for local data</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td>item's</td>
<td>:</td>
</tr>
<tr>
<td>elements</td>
<td>:</td>
</tr>
<tr>
<td>subs</td>
<td>-114</td>
</tr>
<tr>
<td>result</td>
<td>-113</td>
</tr>
<tr>
<td>STATIC LINK</td>
<td>-128</td>
</tr>
<tr>
<td>DYNAMIC LINK</td>
<td>-127</td>
</tr>
<tr>
<td>RETURN ADDR</td>
<td>-126</td>
</tr>
<tr>
<td>param: numb</td>
<td>-125 parameters after linkage-locations</td>
</tr>
<tr>
<td></td>
<td>-124 return value</td>
</tr>
<tr>
<td>b_s</td>
<td>-123</td>
</tr>
<tr>
<td>low</td>
<td>-122</td>
</tr>
<tr>
<td>high</td>
<td>-121</td>
</tr>
<tr>
<td>middle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 &lt;--base of main (level = 0)</td>
</tr>
<tr>
<td></td>
<td>0 &lt;-- base of b_s procedure (level = 1)</td>
</tr>
</tbody>
</table>
```
The static allocation of an object involves only inserting an assertion \texttt{STATIC\_DATA(name, ...)}, telling the data accessing phase to use an absolute access mode for this name. The constant allocation can also be done in a machine-independent way. In the MT, there is an assertion, telling the size of constants which can be used as the intermediate access mode. For example,

\begin{verbatim}
INTERMEDIATE\_MODE(16\_bits).     \%on PDP-11
INTERMEDIATE\_MODE(8\_bits).     \%on ND-100
\end{verbatim}

If a constant is too big for that, such as 1000 in ND-100, we allocate space in the end of code area to hold the constant, and use the relative (to PC, the program counter) mode to access it.

5.3 Data accessing

As mentioned in the beginning of this chapter, data accessing consists of two steps: names $\Rightarrow$ virtual access modes $\Rightarrow$ actual access modes (plus necessary code to achieve the latter mapping). In this section we mainly consider the first mapping, leaving the second one to section 5.4 about instruction selection.

Aside the results of the storage allocation phase, the first mapping needs to know how to access non-local data. For example, if it is done by descending the chain of static-links, we must know the frame-pointer and the relative position of the static-link to the base.

This mapping is called the name-binding, and is performed on the action part of IL programs. After a statement is read, the name-binding is done on its operands. Then all operands are replaced by their virtual access modes and the generic IL operators are disambiguated according to the data types of their operands (see Appendix 1 for details). The result is submitted to the instruction selection phase. \textsc{Algorithm 5.2} is the name-binding method by static-links.
ALGORITHM 5.2 Machine-independent name-binding by static-links

INPUT A name used in static-level 1

OUTPUT (1) A virtual access mode for the name,
(2) the IL data type of the name

METHOD
1. Call FRAME_POINTER(base') to get the base register;
   Call STATIC_LINK(sign', abs') to get the relative position
   of the static-link with respect to the base.
2. Call OFFSET(name, 1, abs'', sign'', ILtype').
   If it succeeds, the virtual access mode is
   (sign'' (base' (abs'' #C))),
   and the IL data type is ILtype'.
   Go to 4.
3. Otherwise set l:=l-1, change base' to the base of the
   activation record of the new static-level:
   If abs' # 0 then
   base'=(<> (sign' (base' (abs' #C))),
   else
   base'=(<> base').
   Go to 2.
4. Return the derived machine-independent access mode.

In the following examples the virtual machines are PDP-11 and ND-100
described in section 5.2, and the results are abbreviated for
readability.

The name low in line (14) of the b_s program, will be translated into

(- R5 8) for PDP-11,
(- B 123) for ND-100.

The array name item in line (18) will be translated into

(- (<> R5) 6) for PDP-11,
(- (<> (- B 120)) 125) for ND-100.

For array indexing, we translate the array name and the index name
separately, and replace the whole indexing expression by a virtual access mode. For example, the indexing expression (INDEX item (<> middle)) in line (18) will be translated into

\[- (- (<> R5) 6) (* (<> (- R5 12)) 2))\]  for PDP-11.
\[+ (- (<> (- B 128)) 125) (<> (- B 121)))\]  for ND-100.

Some constant folding can be done here. For example, if the subscript is a constant, the displacement from the array head can be decided at this time.

Very often the mapping from these virtual access modes to the actual ones involves generating instructions for address calculation. The approach to this problem taken in this work is to regard it as normal instruction selection or to do it in the same uniform way as for the explicit operations in IL programs.

This approach has two main advantages:
1. It makes the code generation tasks more uniform, easier to understand, and easier to automate;
2. It facilitates the exploitation of hardware accessing capability without complex algorithms at instruction selection time.

5.4 Instruction selection

The instruction selection algorithm is the same general tree pattern matching, but much easier (therefore more practical for the CG) than that used in the CGG. The following is the comparison with ALGORITHM 4.1:

1. The evaluation order is fixed as top-down, left-to-right; no attempt at eliminating common sub-expressions has been made. So the EVO and CSE rules are removed.
2. Every IL operator (except the high level control operators) has a template with the "best" code sequence in the MT, no AXI rules are needed.
3. Fetch / store subtargeting (the SUB rules) remains and works as in the CGG.

4. The MT contains only templates of conditional and unconditional jumps. The high level control operations will be decomposed into these primitive ones at code generation time. This decomposition (the DEC rules) is machine-independent, having a very small set of alternatives, and is therefore included in the skeleton CG.

5. No backtracking. Only one solution is obtained. However, some simple heuristics are used to solve the multiple matching problem. Tests have shown them to work well:

5.1. When subtargeting occurs, there is usually more than one usable access mode for the temporary location. In the MT we arrange the access mode descriptions by decreasing complexity in order to try the most complex mode first. The net effect is that the hardware addressing features are fully taken advantage of.

For example, in the MT for NO-100, the first access mode described is the "pre- and post-indexing" mode (,BI,X d'). Its PROLOG description is given as the first line below:

(+ X ((<< (- B d')))), % X,B are registers
(+ (- ((<< (- B 128) 125)) ((<< (- B 121))));

The second line is deliberately appended after the access mode expression. We recognize that it is the indexing expression (INDEX item ((<) middle)) in section 5.3. If we first load the subexpression ((<< (- B 128) 125)) to the X register by the instructions

LDX B (-128),
AAX (-125),
then the indexing expression can be used as an operand in any proper instructions with the access mode (,BI,X 121). Here, very simple heuristics give quite satisfactory result.

5.2 Some operators have more than one template because of special cases of operands. In the MT, the templates for special cases are always put before the general cases. For
example, on PDP-11, the template for $x' := x' + 1$ giving the faster code (INC $x'$) is put before the general template $x' := x' + y'$ which gives the code (ADD $y'$ $x'$). For the IL statement \(<-\) \text{subs} (+ (<c> \text{subs} 1)))\) on line (30) of the b_s program the faster code (INC -28(R5)) will be generated instead of the general (ADD #1 -28(R5)).

5.3 For the operation (IF\_THEN bool\_st\_), there are two decomposition alternatives: the first tries to utilize the cheaper SKIP instructions and the second is for general JUMP instructions (see section 4.2.3.2.4). Our CG provides both possibilities and tries the former one first.

Because of these simple heuristics, and because the MT contains any necessary IL patterns (as complex as basic blocks) and the "best" code sequences for them, the simple instruction selection strategy employed in our CG produces code with the quality comparable with some production compilers. (See section 5.7).

5.5 Register allocation

The register allocation is performed "on the fly" during the instruction selection. Simple as this, it works properly, and is machine-independent, driven by the MT.

In the MT, there is a description of user allocatable registers. The information given in the description includes:

- The type (data register or location register, in the former case what types of data the register can hold)
- The status (free or occupied, initially all user allocatable registers are set free).

ALGORITHM 5.3 Register allocation

METHOD
1. When the fetch / store subtargeting chooses an access mode for a
temporary location, and that mode demands allocation of a register of a certain type, we check the register description to find the first free register of that type, set the status of it to occupied.

2. If there is no such register available, the CG tries the next usable access mode, hoping that the register involved this time can be allocated successfully.

3. If no access mode can be applied because of the lack of proper registers, we allocate a memory location (on the run-time stack, for example) for a temporary variable.

4. If the allocation of a busy register is compulsory, we allocate a memory location to store the old contents of that register, and as soon as this new use of the register is over, we reload the old contents to the register.

5. As soon as a bottom assignment (<- x' y') has been processed, the user allocated registers involved in the destination x' will be set free. The bottom assignment is derived from the previous subtargeting, and can be solved without any more subtargeting.

For example, on ND-100, the statement

\[
\langle- a (+ (<> b) (<> c)) \rangle \quad \text{isz a, b, c are memory locations.}
\]

will be processed in the following top-down, left-to-right order:

<table>
<thead>
<tr>
<th>remaining IL text</th>
<th>CG function</th>
<th>code</th>
<th>reg allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle- a (+ (&lt;&gt; b) (&lt;&gt; c))\rangle</td>
<td>store subtarget</td>
<td></td>
<td>alloc A reg</td>
</tr>
<tr>
<td>\langle- a (&lt;&gt; A) \rangle</td>
<td>template match</td>
<td>STA</td>
<td>a</td>
</tr>
<tr>
<td>\langle- A (+ (&lt;&gt; b) (&lt;&gt; c))\rangle</td>
<td>fetch subtarget</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\langle- A (+ A (&lt;&gt; c))\rangle</td>
<td>template match</td>
<td>ADD</td>
<td>c</td>
</tr>
<tr>
<td>\langle- A (&lt;&gt; b)\rangle</td>
<td>bottom assign</td>
<td>LDA</td>
<td>b</td>
</tr>
</tbody>
</table>

As we can see from the simple example above, the code is produced in reverse execution order, the order of "alloc" and "free" is also reversed.
5.6 Procedure-linkage and parameter-passing

This part of the CG is also machine-independent and implementation-independent. But the method to achieve this independency is different from that used for instruction selection. It is more ad hoc, and has effects in the MT, the CG and the IL designs. We will address them separately.

1. In the MT there are templates giving the code sequences for parameter passing and procedure calling. But they are not automatically derived from the machine description. Rather, they are hand-coded and inserted to the MT afterwards. For example,

on PDP-11, \( \text{PARA } x' \) \( \Longrightarrow \) MOV \( x' - (SP) \) ZZpush on stack
\( \text{(CALL } p' \text{)} \) \( \Longrightarrow \) JRS PC \( p' \)

The implementation decision is, first to push parameters onto the stack, then to jump to the callee. And the code for the callee always starts with calling the run-time-routine-enter-procedure to build a new activation record.

on ND-100, \( \text{PARA } x' \) \( \Longrightarrow \) LDA \( x' \)
\( \text{(CALL } p' \text{)} \) \( \Longrightarrow \) JPL (I,p')

Here \( (X, d') \) represents the location for the coming parameter on the activation record of the called procedure \( p' \). The implementation decision is, first, calling the run-time-routine-enter-procedure to build the new record, using the \( X \) register as its frame-pointer temporarily, moving the parameters to their locations (relative to the base \( X \) register), jumping to the callee. At the beginning of the callee, there are instructions changing the frame-pointer from \( X \) to the normal B register, among other things.
2. In the CG, the processing of (PARA x') and (CALL p') is also by
   template matching, which is implementation-independent. However, some
   implementation-dependent issues exist. The most important ones are:
   - interface with the run-time system (procedure entering and
     exiting);
   - parameter positioning as the example above shows;
   - passing of the return value.
   The approach taken to get rid of this dependency is: try to provide
   common alternatives as options in the CG. The present implementation
   of the CG does provide some options like the different positioning of
   parameters.

   The present CG always puts the return value (if any) in the first
   register of the proper type when leaving a procedure. And after (CALL
   p') is processed, the CG uses that register (R1 for PDP-11, A register
   for NO-100, for example) to access the return value.

3. In the IL, we decompose a procedure-calling like result := b_s(5)
   into a sequence of IL statements:

   (PARA 5)
   (CALL b_s)
   (<- result RETURN)  ; RETURN is a reserved word in IL

   This facilitates the parameter-passing and procedure-calling strategy
   described above. And within an IL procedure with return value, we also
   use explicit assignment to the name of the procedure to indicate the
   return value.

5.7 An example of code generation

   After these explanations of various aspects of the CG, we are in the
   position to present a complete example of code generation. The example
   is the binary search program introduced in section 5.1.4. In this
   section we will present the code produced by our CG for PDP-11 and
   NO-100 computers.

   First, here is the code for PDP-11:
The IL program

MAIN
DEFINITION
(item INT 10)
(subs INT 1)
(result INT 1)
DEFEND
(PROC b_s INT
 (numb INT 1))

The symbolic assembly code for POP-11

(MAIN : LEVEL IS 0 )

(0 : - 6 = item)
(0 : - 28 = subs)
(0 : - 30 = result)

(B_S : LEVEL IS 0 )

run-time-routine-enter-procedure
(1 : + 2 = numb)
(1 : - 6 = b_s)

DEFINITION
(low INT 1)
(high INT 1)
(middle INT 1)
DEFEND
ACTION
(<= low 1)
(<= high 10)
(REPEAT
  (<= middle
   (/ (+ (<= low) (<= high))
    2)));

(IF_THEN_ELSE
 (LEQ (<= (INDEX item (<= middle)))
  (<= numb))

(<= low (+ (<= middle) 1))
(<= high (- (<= middle) 1))

MOV (# 1) (- 8 ( R5 ))
MOV (# 10) (- 10 ( R5 ))

V682 :

MOV (- 8 ( R5 )) R0
ADD (- 10 ( R5 )) R0
ASR R0
MOV R0 (- 12 ( R5 ))
MOV (- 12 ( R5 )) R1

MOV (⇒ R5 ) R0
SUB (# 6) R0
SUB R1 R0
CMP (⇒ R0) (2 ( R5 ))
BLE V921:

MOV (- 12(R5)) (- 10(R5))
DEC (- 10 ( R5 ))
BR V923:
V921' :
    MOV (- 12(R5)) (- 8(R5))
    INC (- 8 ( R5 ))

V923' :
    CMP (- 8(R5)) (- 10(R5))
    BLE V682'
    MOV (- 8 ( R5 )) R0
    INC R0
    CMP R0 (- 10 ( R5 ))
    BGT V257'
    CLR (- 6 ( R5 ))
    BR V259'

V257' :
    MOV (- 12(R5)) (- 6(R5))

V259' :
    MOV (- 6 ( R5 )) R0
    run-time-routine-exit-procedure

ACTION

<- subs 1)
(REPEAT

((<- (INDEX item (<< subs))

(<< subs)) ;
(<< subs (+ (<< subs) 1)))

(GTR (<< subs) 10)
)

(PARA 5)
(CALL b_s)
(<- result RETURN)
ACTEND
MAINEND
The number of instructions is 41 and the number of bytes taken by the code is 76. The well-known C compiler on PDP-11 produced the code of 46 instructions and of 74 bytes (see /Ganapathi 1980/). So the code quality of our simple CG is as good as a production compiler’s. Considering that, our CG does not try any optimizations except those mentioned in the previous sections, the result is satisfactory. As for the execution time, under the present interpretive implementation of PROLOG, it takes 13 seconds. We will discuss these questions in more detail in the conclusion chapter.

The following is the ND-100 code produced by the CG from the same IL program.

The symbolic code for ND-100 produced by CG from the IL program

```plaintext
(MAIN : LEVEL IS 0 )
(0 : - 125 = item)
(0 : - 114 = subs)
(0 : - 113 = result)
(b_s: LEVEL IS 0 )
COPY X B
COPY L A
STA (,B -126)
(1 : - 125 = numb)
(1 : - 124 = b_s)
(1 : - 123 = low)
(1 : - 122 = high)
(1 : - 121 = middle)
SAA 1
STA (,B - 123)
SAA 10
```
STA (,B - 122)

V688':
LDA (,B - 123)
ADD (,B - 122)
SHA -1
STA (,B - 121)
LDA (,B - 128)

SUB (# 125)
STA X
LDA (,B - 125)
SUB (,BI,X - 121)
JAP V910'
LDA (,B - 121)
AAA (-1)
STA (,B - 122)
JMP (V912')

V910':
LDA (,B - 121)
AAA 1
STA (,B - 123)

V912':
LDA (,B - 122)
SUB (,B - 123)
JAP V688'
LDA (,B - 123)
AAA 1
STA (B - 120)
LDA (,B - 122)
SUB (B - 120)
JAP V259'
STZ (,B - 124)
JMP (V261')

V259':
LDA (,B - 121)
STA (,B - 124)

V2G1’:

LDA (,B - 124)

run-time-routine-exit-procedure

SAA 1
STA (,B - 114)

V403’:

COPY B X
AAX (- 125)
LDA (,B - 114)
STA (,BI,X - 114)
MIN (,B - 114)
NO-OP -
SAA 10
SUB (,B - 114)
JAP V403’

run-time-routine-enter-procedure

SAA 5
STA (,X - 125)
JPL (I B_S)
STA (,B - 113)

B_S: (b_s 0)

The ND-100 code produced by the production MARY compiler (which is of high quality, used as the main system programming tool at RUNIT), from the equivalent MARY program (see section 5.1.4) has the same number of instructions. Again, the code quality of our simple CG is proved to be as good as another production compiler. The execution time of the CG for this example is about 14 seconds.
6. RESULTS, CONTRIBUTIONS AND FUTURE RESEARCH

The design of the Code Generator Writing System (CGWS) based on the predicate logic language PROLOG has been presented. The CGWS accepts the PROLOG description of the target machine architecture and the compiler implementation decisions, and creates a code generator that generates assembly code for the target machine from IL programs. The IL is a high level intermediate language and the created code generator covers a substantial part of the synthesis phase of a practical compiler: storage allocation, data accessing, instruction selection, register allocation and procedure handling.

The Code Generator Generator (CGG) generates for each target machine the Mapping Table (MT), which describes the mapping from IL patterns to code sequences for the target machine. The machine-independent skeleton code generator (CG) is written once for all, valid for all target machines. A CGWS created code generator is in fact the joining of the CG and the MT for a particular target machine.

The CGG can accept IL patterns potentially as complicated as basic blocks, and produce near-optimal code sequences. A pattern selection strategy based on statistics and tuning has been proposed.

6.1 Results

A prototype CGWS has been developed on a ND-100 computer, using NTH PROLOG. In this section we evaluate the results of the current experimental implementation of the CGWS. The prototype system contains the following parts:

1. The Target Machine Description (TMD)

PDP-11, ND-100 and MOTOROLA-68000 have been described in PROLOG
with limited data types (complete production versions containing all data types are straightforward extensions, but not necessary for the purpose of this thesis). Various compiler implementation decisions are also described, each of the three TMDs has a different set of such decisions.

PROLOG has proved to be a very good tool for specifying TMDs. The description is compact. For example, 150 lines of PROLOG are enough to describe PDP-11. It is both human and machine readable. It is easy for the user to write this description. Based on the target machine manual, the task is simply to write a PROLOG clause for each access mode and for each instruction. Also a clause is written for each compiler implementation decision. The whole thing is a matter of hours for a programmer familiar with the target machine.

2. The Code Generator Generator (CGG) and the Mapping Table (MT)

The PROLOG program for the CGG has been completed (about 80) lines). Three MTs for PDP-11, ND-100 and MOTOROLA-68000 have been generated, from the necessary minimum set of IL patterns. The execution time to build an MT ranges from 2 to 4 minutes, and the MT size is from 300 to 600 PROLOG lines.

For more complicated IL patterns, the benchmark in /Kozlak 1981/ has been tested for PDP-11, ND-100 and the machine models A, B, C used in that work. The code sequences of our CGWS are better. The search time for each pattern ranges from few seconds to one minute, the same as the time reported in the former work. This is good considering that Kozlack's system is implemented in an ordinary programming language.

The implementation of the pattern selection based on statistics and tuning proposed in chapter 4 is left for future work.

3. The Intermediate Language (IL) and the IL Pattern File (PF)

The design of the IL is extensible. The current definition of the IL has been given in chapter 5. To extend it we need only add new operators to it and design the representation of more complicated data structures like e.g., "records". The IL is a high level intermediate
language. The translation from source languages to the IL is therefore relatively easy. This is, however, outside the scope of this thesis. The IL program example in chapter 5 is a hand-translation from PASCAL and MARY.

The PF contains only the minimum set of IL patterns and is also hand-written, but the automatic construction of it from the definition of the IL is not difficult.

4. The Code Generator (CG)

The PROLOG program for the CG has also been completed (700 lines). As the example in chapter 5 shows, merging the MT for a particular target machine into the CG makes a complete code generator which can produce good assembly code of that machine for IL programs.

The present CG in PROLOG is mainly for the purpose of demonstration. The speed (4-5 instructions per second) is not very satisfactory for a production compiler. To put the created code generator into practical use, we can use program transformation techniques to remove inefficiencies (see /Hansson & Tärrlund 1982/ for example). We can also improve the PROLOG interpreter itself. Another possibility is to make a PROLOG compiler instead of the present interpreter. This can improve the execution time by a factor of 10-20, which brings the CG up to the speed of production code generators.

We can also translate the PROLOG CG into an ordinary system programming language like MARY or CHILL. In that case the MT must be also translated into tables that can be read as input data by the program in that language, or, translated into program modules in that language and loaded with the CG module. The hand-written translation requires some efforts, but much less than writing the code generator in that language directly. Automatic translation is of course more attractive, but it also presents a bigger challenge. All of these are left for future work.
6.2 The major contributions

The present work tries to combine two useful tools together to facilitate code generator construction: CGWSs and logic programming. The main ideas are:

1. The code derivation algorithm described in chapter 4 produces near-optimal code sequences for IL patterns which can be as complex as basic blocks. The space requirement of this near-optimal code derivation is no more than what is needed for finding only one solution. The algorithm gains a good trade-off between code optimization and time efficiency, and can be conveniently implemented in PROLOG.

2. The idea to do more optimization at code generator generation time is very advantageous. It can improve the overall code quality without time-consuming optimization at code generation time, resulting in a faster and better code generator. The only serious problem involved here is to select the basic blocks for the IL patterns. The proposed strategy based on statistics and tuning is the first step in this direction.

3. This work has shown how all machine dependent issues in the code synthesis phase of compilation can be included into a single package, and to deal with them in the table-driven way. The net effect is that other modules of a compiler need not consider the target machine, making the interfaces between modules much easier.

4. The logic programming language has been introduced into the area of CGWSs for the first time. The experiment shows:

- PROLOG is a excellent tool for the specification of CGWSs;
- PROLOG is a good tool for the implementation of CGGs (here a natural specification of a problem presents at the same time a reasonably good implementation of that problem);
- For CGs, the present PROLOG implementation is not good enough to be a production code generator. Still it is useful for rapid prototyping of experimental code generators (for example to test various compiler implementation decisions).
This use of PROLOG has also important perspectives for the future. Logic programming languages are gaining more and more importance in computer science as a whole. One significant factor is that the Japanese have chosen the logic programming language as the kernel language for their fifth generation computer systems (the FGCS project, see for example /ICOT 1982/). The kernel language will be based on PROLOG, and will be implemented as machine language by hardware. At present, many research activities are going on for enhancing the PROLOG definition, for improving its implementation and for extending its application area and accumulating experience in its application to various problems. For example, the FGCS project has set the aim to improve the performance of PROLOG up to 100,000 LIPS (Logical Inferences Per Second) by the year 1985, 200 times faster than our present PROLOG interpreter. Within this context we can see more significance of this experimental work trying to apply PROLOG to CGWSs other than the possible immediate benefits.

6.3 Future research

Here is the summary of the issues left for future research, some of them have been mentioned before in separate places:

1. To make a PROLOG compiler or improve the present interpreter. The CG implemented in such a new PROLOG will probably be efficient enough as a production code generator.

2. To transform the PROLOG specification of the CG into a more efficient PROLOG implementation.

3. To rewrite the CG in MARY or CHILL or PASCAL to make a production code generator.

4. To implement the pattern selection strategy proposed in chapter 4 and investigate the effect on the overall code quality.

5. To extend the IL definition to include other data types and operators appearing in source languages like CHILL and MARY.

6. To extend the TMD model to include other machine operations such
as operating system interfaces and exotic instructions and investigate how the CGG can be extended to deal with them.

Finally, it should be noted that although this work is a study to test the proposed ideas (not a production implementation), a real CGWS based on it would be a promising project.
REFERENCES

/Aho & Johnson 1976/ A.V.Aho, S.C.Johnson,
"Optimal code generation for expression trees",

/Aho & Ullman 1977/ A.V.Aho, J.D.Ullman,
"Principles of Compiler Design",

/Aho et al. 1977/ A.V.Aho, S.C. Johnson, J.D.Ullman,
"Code generation for expressions with common subexpressions",

/Amble 1983/ T.Amble,
"NTH PROLOG reference manual",
RUNIT, Trondheim, Norway, 1983.

/Ammann 1977/ U.Ammann,
"On code generation in a PASCAL compiler",
Software Practice and Experience 7,3, (June/July 1977),
PP.391-423.

/Bruno & Sethi 1976/ J.Bruno, R.Sethi,
"Code generation for a one-register machine",

/Cattell 1977/ R.G.G.Cattell,
"A survey and critique of some models of code generation",

/Cattell 1978/ R.G.G.Cattell,
"Formalization and automatic derivation of code generators",
1978.

/Cattell et al. 1979/ R.G.G.Cattell, J.M.Newcomer, B.W.Leverett,
"Code generation in a machine-independent compiler",

/Cattell 1980/ R.G.G.Cattell,
"Automatic derivation of code generators from machine description",

/Cater 1982/ L.R.Cater,
"Further analysis of code generation for a single register
machine",

/Conradi & Holager 1979/ R.Conradi, P.Holager,
"Mary Textbook",
RUNIT, Apr. 1979.

/Digital 1971/ Digital Equipment Corporation,

/Elson & Rake 1970/ M.Elson, S.T.Rake,
"Code generation techniques for large language compilers",

/Fraser 1977a/ C.W.Fraser,
"Automatic generation of code generators",

/Fraser 1977b/ C.W.Fraser,
"A knowledge-based code generator generator",

/Ganapathi 1980/ M.Ganapathi,
"Retargetable code generation and optimization using attribute
grammars",
Ph.D. dissertation, Computer Science Dep., Univ. of

/Ganapathi & Fischer 1981/ M.Ganapathi, C.N.Fischer,
"Bibliography on automated retargetable code generation"

/Ganapathi et al. 1982/ M.Ganapathi, C.N.Fischer, J.L.Hennessy,
"Retargetable code generation",

/Glanville 1977/ R.S.Glanville,
"A machine independent algorithm for code generation and its use
in retargetable compilers",

/Glanville & Graham 1978/ R.S.Glanville, S.L.Graham,
"A new method for compiler code generation",

/Graham 1980/ S.L.Graham,
"Table-driven code generation",

/Gries 1971/ D.Gries,
"Compiler Construction for Digital Computers",

/Hansson & Tärnlund 1982/ Å.Hansson, S-Å.Tärnlund,
"Program transformation by data structure mapping",

/Heensaasen et al. 1983/ O.Heensaasen, K.J.Wedde,
O.Solberg, K.H.Sundnes,
"CHILL implementation on NO-100. Machine independent part"
SINTEF report No. STF14 F83014, 1983.

/Holt 1981/ R.C.Holt,
"Data descriptors: A compile-time model of data and addressing",
Computer Systems Research Group, Univ. of Toronto, 1981.

/ICOT 1982/ ICOT,
"Outline of research and development plans for fifth generation
computer systems",

/Jensen et al. 1982/ P.Jensen, et.al.
"Definition of S-code",
SIMULA INFORMATION, No 672, 1982.

/Johnson 1978/ S.C.Johnson,
"A portable compiler: Theory and practice",

/Kessler 1981/ R.R.Kessler,
"COG: An architectural description driven compiler generator",

/Kowalski 1974/ R.A.Kowalski,
"Predicate logic as programming language",
Proc. IFIP-74 Congress, pp.569-574.

/Kozlak 1981/ R.H.Kozlak,
"Machine-independent code generation",

/Lunell 1983/ H.Lunell,
"Code generator writing systems",
Ph.D. dissertation, Linkoping Software Systems Research Center,
Sweden, 1983.

/Miller 1971/ P.L.Miller,
"Automatic creation of a code generator from a machine
description",

/Morgan 1982/ T.M.Morgan,
"Analysis techniques for exotic machine instructions and their
use in retargetable compilers",

/Newcomer 1975/ J.M. Newcomer,


Appendix 1.

The definition of the IL

In Chapter 5 we have described the Intermediate Language (IL) used in our CGWS. Here is the complete definition of the IL. The syntax is described in BNF. The semantics of operators is described in PROLOG clauses as a part of the CG program.

The BNF Syntax

Notes of BNF:  { }* stands for repetition zero or more times;
                { }+ stands for repetition one or more times;
                | stands for alternatives;
                ... stands for "more to be implemented"
                objects in < > are non-terminals;
                other objects are terminals;
                % stands for comments.

<IL-program>::=MAIN <body> MAINEND
<body>::=<var-def-part> <proc-def-part> <act-part>
<var-def-part>::=DEFINITION { <def> }* DEFEND
<proc-def-part>::= { <proc-def> }*
<act-part>::=ACTION { <act> }* ACTEND
<proc-def>::=<proc-head> <body> <proc-tail>
<def>::={<id> <datatype> <amount> <allocation-type>}
<datatype>::=INT | BOOL | BYTE | POINTER | ...
<amount>::=integer
<allocation-type>::=<empty> | S | ...
%<empty> for stack allocation, S for static allocation.
<act>::={<op> { <operand> }*}
<operand>::=:<id> | <constant> | <act>
<act>::={<act> ; <act> }+
<proc-head>::=(PROC <id> <return-type> <parameter-list>)
<return-type>::=VOID | INT | BYTE | BOOL | POINTER | ...
%VOID for "no value returned"
<parameter-list>::=<para> | {<para> ; <para> }+
<para>::=<def> | NIL %NIL for non-parameter procedure
<proc-tail>::=PROCEND

The description of operators

There are two kinds of operators: generic and unambiguous. The operators appearing in IL programs are generic, the same as in source programs. However, they are disambiguated by the CG before instruction selection.

The operator description is a part of the CG: For each generic operator we use a set of PROLOG unconditional clauses to describe its n-ary, the data types of its operands and of the returned value, and the name of the unambiguous operator for each legal operand datatype.

1. The monadic operators.
   The clauses describing a monadic operator are like
   MA_OP(generic-op', type-operand', type-returned', unamb-op').
MA_OP(MINUS, BYTE, BYTE, B_MINUS).
MA_OP(MINUS, INT, INT, MINUS).

MA_OP(NOT#, BYTE, BYTE, B_NOT#).  \(\text{\#bit-wise}\)
MA_OP(NOT#, INT, INT, NOT#).  \(\text{\#bit-wise}\)

MA_OP(NOT, BOOL, BOOL, NOT).

MA_OP(<>, BYTE, BYTE, <->B).  \(\text{\# The type-operand' is POINTER}\)
MA_OP(<>, INT, INT, <->).  \(\text{\# to BYTE, INT... We use BYTE,}\)
MA_OP(<>, POINTER, POINTER, <->).  \(\text{\# INT... to simplify the CG code.}\)

MA_OP(PARA BYTE BYTE B_PARA).
MA_OP(PARA INT INT PARA).

2. The dyadic operators
The clauses describing a dyadic operator are like
DY_OP(generic-op', typel', typer', type-returned', unamb-oo').

DY_OP(+, BYTE, BYTE, BYTE, +B).
DY_OP(+, INT, INT, INT, +{}).
DY_OP(+, POINTER, POINTER, POINTER, +{}).

DY_OP(-, BYTE, BYTE, BYTE, -B).
DY_OP(-, INT, INT, INT, -{}).
DY_OP(-, POINTER, POINTER, POINTER, -{}).

DY_OP(*, BYTE, BYTE, BYTE, *B).
DY_OP(*, INT, INT, INT, *{}).

DY_OP(/, BYTE, BYTE, BYTE, /B).
DY_OP(/, INT, INT, INT, /{}).

DY_OP(SHIFT, BYTE, BYTE, BYTE, B_SHIFT).
DY_OP(SHIFT, INT, BYTE, INT, SHIFT).

DY_OP(AND#, BYTE, BYTE, BYTE, B_AND#).  \(\text{\#bit-wise}\)
DY_OP(AND#, INT, INT, INT, AND#).  \(\text{\#bit-wise}\)

DY_OP(OR#, BYTE, BYTE, BYTE, B_OR#).  \(\text{\#bit-wise}\)
DY_OP(OR#, INT, INT, INT, OR#).  \(\text{\#bit-wise}\)

DY_OP(AND, BOOL, BOOL, BOOL, AND).

DY_OP(OR, BOOL, BOOL, BOOL, OR).

DY_OP(LSS, INT, INT, BOOL, LSS).
DY_OP(EQL, INT, INT, BOOL, EQL).
DY_OP(GEQ, INT, INT, BOOL, GEO).
DY_OP(LEQ, INT, INT, BOOL, LEQ).
DY_OP(NEQ, INT, INT, BOOL, NEQ).
DY_OP(GTR, INT, INT, BOOL, GTR).
DY_OP(LSS, BYTE, BYTE, BOOL, B_LSS).
DY_OP(EQL, BYTE, BYTE, BOOL, B_EQL).
DY_OP(LEQ, BYTE, BYTE, BOOL, B_LEQ).
DY_OP(NEQ, BYTE, BYTE, BOOL, B_NEO).
DY_OP(GTR, BYTE, BYTE, BOOL, B_GTR).
DY_OP(GEO, BYTE, BYTE, BOOL, B_GEO).

DY_OP(<-, POINTER', BYTE, VOID, <B>).
DY_OP(<-, POINTER', INT, VOID, <-).
DY_OP(<-, POINTER', POINTER, VOID, <-).

DY_OP(->, BOOL, POINTER, VOID, ->). ZZ if BOOL if TRUE, jump ZZ to POINTER.

DY_OP(IF_THEN, BOOL, STATEMENT, VOID, IF_THEN).
DY_OP(WHILE, BOOL, STATEMENT, VOID, WHILE).
DY_OP(REPEAT, STATEMENT, BOOL, VOID, REPEAT).

3. The trinary operators

TRI_OP(IF_THEN_ELSE, BOOL, STATEMENT, STATEMENT, VOID, IF_THEN_ELSE).

These clauses are part of the CG, used for disambiguating generic operators. For example, suppose the CG reads a statement containing an expression (+ a b). The CG does name-binding on a and b (see section 5.3). If the results of the name-binding are:

a1 is the machine-independent (virtual) access mode of a,
b1 is the machine-independent (virtual) access mode of b,
a is a BYTE,
b is a BYTE,

then the CG calls DY_OP(+,BYTE,BYTE,x',y'). Matching one of the above clauses (that is DY_OP(+,BYTE,BYTE,BYTE,+B).), y' is bound to +B and the CG replaces (+ a b) by (+B a1 b1), which is the proper form for instruction selection. Besides, x' is bound to BYTE, which is the data type of the returned value of (+ a b). This information is passed bottom-up on each IL statement (an action tree). See appendix 5, the CG code, in the portion for name-binding.
Appendix 2

The Pattern File (PF)

This PF contains the minimum necessary set of IL patterns. The CGG builds kernel MTs from this PF.

The access modes of the operands in the patterns are parameterized. For example, DST1 in the first pattern represents any access modes which can be used as a destination of assignment.

The operators are unambiguous. For simplicity, here we only list the patterns of the integer operations. (The patterns for byte operations are similar to those of integers, but use operators for byte operations, such as +B instead of + and so on).

```lisp
(- DST1 (MINUS (-> DST1)))
(- DST1 (MINUS SRC1))
(- DST1 (NOT# (-> DST1)))
(- DST1 (NOT# SRC1))
(- DST1 (+ (-> DST1) (1 #C)))
(- DST1 (+ (-> DST1) SRC1))
(- DST1 (+ SRC1 (1 #C)))
(- DST1 (+ SRC1 SRC2))
(- DST1 (- (-> DST1) (1 #C)))
(- DST1 (- (-> DST1) SRC1))
(- DST1 (- SRC1 (1 #C)))
(- DST1 (- SRC1 SRC2))
(- DST1 (* (-> DST1) (2 #C)))
(- DST1 (* SRC1 (2 #C)))
(- DST1 (/ (-> DST1) (2 #C)))
(- DST1 (/ SRC1 (2 #C)))
(- DST1 (SHIFT (-> DST1) (1 #C)))
(- DST1 (SHIFT (-> DST1) (-1 #C)))
(- DST1 (SHIFT SRC1 (1 #C)))
(- DST1 (SHIFT SRC1 (-1 #C)))
(- DST1 (AND# (-> DST1) SRC1))
(- DST1 (AND# SRC1 SRC2))
(- DST1 (OR# (-> DST1) SRC1))
(- DST1 (OR# SRC1 SRC2))
(- DST1 ("C" #C))
(- DST1 SRC1)
(+ (NEQ (A T(DATA WORD)) SRC2) ("LABEL" #RM))
(+ (EQL (A T(DATA WORD)) SRC2) ("LABEL" #RM))
(+ (LSS (A T(DATA WORD)) SRC2) ("LABEL" #RM))
(+ (GTR (A T(DATA WORD)) SRC2) ("LABEL" #RM))
(+ (LEQ (A T(DATA WORD)) SRC2) ("LABEL" #RM))
(- (NEQ (A T(DATA WORD)) SRC2) (+ PC 1))
(- (EQL (A T(DATA WORD)) SRC2) (+ PC 1))
(- (LSS (A T(DATA WORD)) SRC2) (+ PC 1))
(- (GTR (A T(DATA WORD)) SRC2) (+ PC 1))
(- (LEQ (A T(DATA WORD)) SRC2) (+ PC 1))
(+ (GEO (A T(DATA WORD)) SRC2) (+ PC 1))
(+ (NEQ SRC1 (A T(DATA WORD))) ("LABEL" #RM))
(+ (EQL SRC1 (A T(DATA WORD))) ("LABEL" #RM))
(+ (LSS SRC1 (A T(DATA WORD))) ("LABEL" #RM))
(+ (GTR SRC1 (A T(DATA WORD))) ("LABEL" #RM))
(+ (LEQ SRC1 (A T(DATA WORD))) ("LABEL" #RM))
```
(-) (GEQ SRC1 (A T(DATA WORD)))
(-) (NEQ SRC1 (A T(DATA WORD)))
(-) (EQL SRC1 (A T(DATA WORD)))
(-) (LSS SRC1 (A T(DATA WORD)))
(-) (GTR SRC1 (A T(DATA WORD)))
(-) (LEQ SRC1 (A T(DATA WORD)))
(-) (GEQ SRC1 (A T(DATA WORD)))
(-) (NEQ SRC1 ("D" #C))
(-) (EQL SRC1 ("D" #C))
(-) (LSS SRC1 ("D" #C))
(-) (GTR SRC1 ("D" #C))
(-) (LEQ SRC1 ("D" #C))
(-) (GEQ SRC1 ("D" #C))
(-) (NEQ SRC1 SRC2)
(-) (EQL SRC1 SRC2)
(-) (LSS SRC1 SRC2)
(-) (GTR SRC1 SRC2)
(-) (LEQ SRC1 SRC2)
(-) (GEQ SRC1 SRC2)
(-) (NEQ SRC1 SRC2)
(-) (EQL SRC1 SRC2)
(-) (LSS SRC1 SRC2)
(-) (GTR SRC1 SRC2)
(-) (LEQ SRC1 SRC2)
(-) (GEQ SRC1 SRC2)

("LABEL" #RM))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
(+ PC 1))
Appendix 3

The TMDs for PDP-11 and ND-100

In this appendix we present the Target Machine Descriptions (TMDs) for PDP-11 and ND-100. For simplicity, only integer instructions are included.

The TMD for PDP-11

% The Compiler Implementation Decisions.

ALLOCATABLE_REG(((R0 T((DATA LOCA) (BYTE WORD)) FREE),
(R1 T((DATA LOCA) (BYTE WORD)) FREE),
(R2 T((DATA LOCA) (BYTE WORD)) FREE),
(R3 T((DATA LOCA) (BYTE WORD)) FREE),
(R4 T((DATA LOCA) (BYTE WORD)) FREE))).

DATA_TYPE_CONVERSION(BOOL BYTE 1 1).
DATA_TYPE_CONVERSION(CHAR BYTE 1 1).
DATA_TYPE_CONVERSION(INT WORD 2 2).
DATA_TYPE_CONVERSION(POINTER WORD 2 2).
DATA_TYPE_CONVERSION(Void VOID 0 1).

FRAME_POINTER((R5 T(LOCA WORD))).
STATIC_LINK(-, 0).
FRAME_DIRECTION(DOWN).
DATASTART(-, 6).
PARAMETER(BEFORE).
INTERMEDIATE_MODE(16_bits).

% The access modes and operand classes.

AM1((<> [+ (R' T(LOCA WORD)), (C' #C)]), (C' [R'])).
AM1((<> [+ (C' #C), (R' T(LOCA WORD))]), (C' [R'])).
AM1((<> [- (R' T(LOCA WORD)), (C' #C)]), (C' [R'])).
AM2(+[ (C' #C), (R' T(LOCA WORD))], (C' [R'])).
AM2(+[ (R' T(LOCA WORD)), (C' #C)], (C' [R'])).
AM2(+[ (R' T(LOCA WORD)), (C' #C)], (C' [R'])).
AM3((R' T(LOCA WORD)), (C' [R'])).
AM4((R' T(DATA WORD)), (R')).
AM4((R' T(LOCA WORD)), (R')).
AM5((H #AM), (@# H')).
AM6(<> [L' #RM], (@ L')).
AM7(<> [L' #RM], L').
AM0((C' #C), (# C')).

DST OC((TREE', DST' 0): AM4((TREE', DST'), ACCEPT.
DST OC((TREE', DST' 1): AM1((TREE', DST'),
DST OC((TREE', DST' 1): AM2((TREE', DST'),
DST OC((TREE', DST' 0): AM3((TREE', DST'),
DST OC((TREE', DST' 1): AM5((TREE', DST'),
DST OC((TREE', DST' 1): AM6((TREE', DST'),
DST OC((TREE', DST' 1): AM7((TREE', DST'),
SRC OC((TREE SRC' 0): AM4((TREE SRC'), ACCEPT.
SRC OC((<> TREE', SRC' 1): AM1((TREE', SRC'),
SRC OC((<> TREE', SRC' 1): AM2((TREE', SRC'),
SRC OC((<> TREE', SRC' 0): AM3((TREE', SRC').
% The instructions.

CODE(\(<- \) $1'$ (0 \#C)), INS(CLR DST '_1'), C'):
   DST_OCR($1',DST',C'''), SRC_OCR($1',DST',C''), PLUS(1 C' C''').
CODE(\(<- \) $1'$ (NOT#, $1'')) INS(COM DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C''').
CODE(\(<- \) $1'$ (+ $1'' (1 \#C))), INS(INC DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C'').
CODE(\(<- \) $1'$ (+ (1 \#C) $1''')) INS(INC DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C'').
CODE(\(<- \) $1'$ (- $1'' (1 \#C))), INS(DEC DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C'').
CODE(\(<- \) $1'$ (MINUS, $1''')) INS(NEG DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C'').
CODE(\(<- (N \#N), LSS $1'$ (0 \#C))), INS(TST DST '_1'), C'):
   SRC_OCR($1',DST',C''), PLUS(1 C' C'').
CODE(\(<- (Z \#2), EQL $1'$ (0 \#C))), INS(TST DST '_1'), C'):
   SRC_OCR($1',DST',C''), PLUS(1 C' C'').
CODE(\(<- $1'$ (SHIFT $1'$ (-1 \#C))), INS(ASR DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C'').
CODE(\(<- $1'$ (SHIFT $1'$ (1 \#C))), INS(ASL DST '_1'), C'):
   DST_OCR($1',DST',C'''), CONTENTS($1' $1'''), PLUS(1 C' C'').

CODE(\(<- \) $1'$ (+ $1'',$2'')) INS(ADD SRC' DST'1'), C'):
   DST_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- \) $1'$ (+ $2', $1'')) INS(ADD SRC' DST'1'), C'):
   DST_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- \) $1'$ (- $1'',$2'')) INS(SUB SRC DST'1'), C'):
   DST_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- (N \#N) (LSS $2'$,$1'')) INS(CMP SRC DST'1'), C'):
   SRC_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- (Z \#2), EQL $2'$,$1'')) INS(CMP SRC DST'1'), C'):
   SRC_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- (Z \#2) (EQL (OR# $2'$,$1'') (0 \#C))), INS(BIT SRC',DST'1'), C'):
   SRC_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- (N \#N) (LSS (OR# $2'$,$1'') (0 \#C))), INS(BIT SRC',DST'1'), C'):
   SRC_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- \) $1'$ (AND# (NOT# $2'$) $1'')) INS(BIC SRC',DST'1'), C'):
   DST_OCR($1',DST',C''), SRC_OCR($2',SRC',C''), CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C''''').
CODE(\(<- \) $1'$ (AND# $1'' (NOT# $2''))) INS(BIC SRC',DST'1'), C'):
   DST_OCR($1',DST',C''), SRC_OCR($2',SRC',C''),
CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C'' C''').
CODE({< - $1' (OR# $2' $2''), INS(BIS SRC', DST'), C''):
  DST_OC($1', DST', C'''), SRC_OC($2', SRC', C'''),
  CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C' C''').
CODE({< - $1' $2' (OR# $2' $2''), INS(BIS SRC', DST'), C''):
  DST_OC($1', DST', C'''), SRC_OC($2', SRC', C'''),
  CONTENTS($1' $1''), PLUS(1 C' C'''), PLUS(C' C' C''').
CODE({< - $1' $2') INS(MOV SRC' DST''), C''):
  DST_OC($1', DST', C''), SRC_OC($2', SRC', C'''),
  PLUS(1 C' C'''), PLUS(C' C' C''').

CODE((GOTO (LA' #RM)) INS(BR LA' _) 1).
CODE((GOTO $1') INS(JMP DST' _ C''):
  DST_OC($1' DST' C''),
  PLUS(C' C').
CODE({< - (Z #Z) ($1' #RM)) INS(BEQ $1' _ 1).
CODE({< - (NOT (Z #Z)) ($1' #RM)) INS(BNE $1' _ 1).
CODE({< - (NOT (N #N)) ($1' #RM)) INS(BGE $1' _ 1).
CODE({< - (N #N) ($1' #RM)) INS(BLT $1' _ 1).
CODE({< - (OR (Z #Z) (N #N)) ($1' #RM)) INS(BLE $1' _ 1).
CODE({< - (AND (NOT (N #N)) (NOT (Z #Z))) ($1' #RM)) INS(BGT $1' _ 1).
CODE({< - (AND (NOT (Z #Z)) (NOT (N #N))) ($1' #RM)) INS(BGT $1' _ 1).

## The ESTIMATE of IL patterns.

ESTIMATE(1; S' S''' E''): ESTIMATE(S' E1'), ESTIMATE(S''' E2'),
PLUS(E1' E2' E').
ESTIMATE(X' E''): DST_OC(X' Y' E').
ESTIMATE(X' E''): SRC_OC(X' Y' E').
ESTIMATE({< - X' Y' E''): ESTIMATE(X' E1''), ESTIMATE(Y' E2''),
PLUS(E1' E2' E'''), PLUS(E' 1 E').
ESTIMATE((OP' Y') E''): ESTIMATE(Y' E').
ESTIMATE((OP' X' Y') E''): ESTIMATE(X' E1''), ESTIMATE(Y' E2''),
PLUS(E1' E2' E').
The TMD for ND-100

The Compiler Implementation Decisions.

ALLOCATABLE_REG((A T((DATA) (BYTE WORD)) FREE),
                 (X T((DATA LOCA) (BYTE WORD)) FREE)).
DATA_TYPE_CONVERSION(BOOL WORD 1 1).
DATA_TYPE_CONVERSION(CHAR WORD 1 1).
DATA_TYPE_CONVERSION(INT WORD 1 1).
DATA_TYPE_CONVERSION(POINTER WORD 1 1).
DATA_TYPE_CONVERSION(VOID VOID 0 1).
FRAME_POINTER((B T(LOCA WORD))).
STATIC_LINK(-,128).
FRAME_DIRECTION(UP).
DATASTART(-,125).
PARAMETER(AFTER).
INTERMEDIATE_MODE(8_bits).

The access modes and operand classes.

AM1(+ (<> (+ B T(LOCA WORD)) (D' #C)) (X T(LOCA WORD))) (E, I, X C').
AM1(+ (X T(LOCA WORD)) (<> (+ B T(LOCA WORD)) (D' #C)) (E, X C')).
AM1(+ (<> (- B T(LOCA WORD)) (D' #C)) (X T(LOCA WORD))) (E, I, X - C').
AM1(+ (X T(LOCA WORD)) (<> (- B T(LOCA WORD)) (D' #C))) (E, I, X - C').
AM2(+ (+ B T(LOCA WORD)) (D' #C)) (X T(LOCA WORD))) (E, X C').
AM2(+ (X T(LOCA WORD)) (+ B T(LOCA WORD)) (D' #C))) (E, X C').
AM2(+ (- B T(LOCA WORD)) (D' #C)) (X T(LOCA WORD))) (E, X - D').
AM2(+ (X T(LOCA WORD)) (- B T(LOCA WORD)) (D' #C))) (E, X - D').
AM3(+ (X T(LOCA WORD)) (D' #C)) (X T(LOCA WORD)) (X D').
AM3(- (X T(LOCA WORD)) (D' #C)) (X T(LOCA WORD)) (X - D').
AM4(+ (<> (L' #RM)) (X T(LOCA WORD))) (I, X L').
AM5((X T(LOCA WORD))) (X 0).
AM6((A T(DATA WORD))) (NIL).
AM7((M' #AM)) (I M').
AM8((L' #RM)) (L').
-AM0((C' #C)) (# C').

On ND-100, operands do not require extra locations, so the cost for all operand classes are 0 and we omit it.

EL(IN' EL'):AM1(IN' EL').
EL(IN' EL'):AM2(IN' EL').
EL(IN' EL'):AM3(IN' EL').
EL(IN' EL'):AM4(IN' EL').
EL(IN' EL'):AM5(IN' EL').
EL(IN' EL'):AM7(IN' EL').
EL(IN' EL'):AM8(IN' EL').

DST_OC(TREE',DST'):EL(TREE',DST').
DST_OC((X T(DATA WORD)) X).
SRC_OC((<> TREE'),SRC'):EL(TREE',SRC').
SRC_OC((A T(DATA WORD)) A).
SRC_OC(TREE',C'):AM4(TREE',C').
SRC_OC((X T(DATA WORD)) X).

CONTENTS(S' S'): AM6(S' W').
% The instructions.

CODE((- (A T(DATA WORD)) (B T(LOCA WORD))) INS(COPY B A) 1).
CODE((- (A T(DATA WORD)) (X T(T1' WORD))) INS(COPY X A) 1).
CODE((- (X T(DATA WORD)) (B T(LOCA WORD))) INS(COPY B X) 1).
CODE((- (X T(DATA WORD)) (A T(T1' WORD))) INS(COPY A X) 1).

CODE((- (A T(DATA WORD)) (MINUS (A T(DATA WORD)))) INS(CPYC A A) 1).
CODE((- (A T(DATA WORD)) (NOT# (A T(DATA WORD)))) INS(CPYN A A) 1).
CODE((- $1' (O #C))
      DST_OC($1' EL').
      INS(STZ EL' _) 1):
      DST_OC($1' EL').
      INS(STA EL' _) 1):
      DST_OC($1' EL').
      INS(STX EL' _) 1):
      DST_OC($1' EL').
      INS(SAA C' _) 1):
      SOURCE_OC($1' EL').
      INS(LDA EL' _) 1):
      SOURCE_OC($1' EL').

CODE((- (A T(DATA WORD)) (C' #C))
      SOURCE_OC($1' EL').
      INS(SAX C' _) 1).
CODE((- (X T(DATA WORD)) (+ (X T(DATA WORD))) (C' #C)) INS(AAX C' _) 1).
CODE((- (X T(DATA WORD)) $1')
      SOURCE_OC($1' EL').
      INS(LDX EL' _) 1):
      SOURCE_OC($1' EL').

CODE((- $1' (+ $2' (O #C)) 1):SOURCE_OC($2' W'), CONTENTS($1' $2').
CODE((- $1' (- $2' (O #C)) 1):SOURCE_OC($2' W'), CONTENTS($1' $2').
CODE((- (A T(DATA WORD)) (+ (A T(DATA WORD)) (C' #C)) INS(AAA C' _) 1).
CODE((- (A T(DATA WORD)) (- (A T(DATA WORD)) (C' #C))
      INS(ATA (- C') _) 1).
      SOURCE_OC($1' EL').
      INS(ADD NO-OP _) 1):
      SOURCE_OC($1' EL').
      INS(ADD NO-OP _) 1):
      SOURCE_OC($1' EL').

CODE((- (A T(DATA WORD)) (+ (A T(DATA WORD)) $1'))
      INS(ADD EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (+ $1' (A T(DATA WORD))))
      INS(ADD EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (- (A T(DATA WORD)) $1'))
      INS(SUB EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (AND# (A T(DATA WORD)) $1'))
      INS(AND EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (AND# $1' (A T(DATA WORD))))
      INS(AND EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (OR# (A T(DATA WORD)) $1'))
      INS(OR # EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (OR# $1' (A T(DATA WORD))))
      INS(OR # EL' _) 1):
      SOURCE_OC($1' EL').

CODE((- (A T(DATA WORD)) (* (A T(DATA WORD)) (1 #C)) 1).
CODE((- (A T(DATA WORD)) ( [A T(DATA WORD)] (1 #C)) 1).
CODE((- (A T(DATA WORD)) (* (A T(DATA WORD)) (2 #C)) INS(SHA 1) 1).
CODE((- (A T(DATA WORD)) (* (A T(DATA WORD)) (2 #C)) INS(SHA -1) 1).
CODE((- (A T(DATA WORD)) (* (A T(DATA WORD)) $1')) INS(MPY EL' _) 1):
      SOURCE_OC($1' EL').
CODE((- (A T(DATA WORD)) (* $1' (A T(DATA WORD)))) INS(MPY EL' _) 1):
      SOURCE_OC($1' EL').
CODE((GOTO $1')
    INS(JMP EL' _),1):EL($1' EL').
CODE(-> (GEO (A T(DATA WORD)) (0 #C)) (L' #RM))
    INS(JAP L' _),1).
CODE(-> (LSS (A T(DATA WORD)) (0 #C)) (L' #RM))
    INS(JAN L' _),1).
CODE(-> (EQL (A T(DATA WORD)) (0 #C)) (L' #RM))
    INS(JAZ L' _),1).
CODE(-> (NEQ (A T(DATA WORD)) (0 #C)) (L' #RM))
    INS(JAF L' _),1).
CODE(-> (LSS (X T(DATA WORD)) (0 #C)) (L' #RM))
    INS(JXN L' _),1).
CODE(-> (EQL (X T(DATA WORD)) (0 #C)) (L' #RM))
    INS(JXZ L' _),1).
CODE(-> (A T(DATA WORD)) (SHIFT (A T(DATA WORD)) (N' #C)))
    INS(SHA N' _),1).
CODE(-> (EQL (A T(DATA WORD)) (0 #C)) (+ PC 1))
    INS(SKPEQL A 0),1).
CODE(-> (NEQ (A T(DATA WORD)) (0 #C)) (+ PC 1))
    INS(SKPNEQ A 0),1).
CODE(-> (GEQ (A T(DATA WORD)) (0 #C)) (+ PC 1))
    INS(SKPGEQ A 0),1).
CODE(-> (LSS (A T(DATA WORD)) (0 #C)) (+ PC 1))
    INS(SKPLST A 0),1).

II the ESTIMATE of IL patterns.

ESTIMATE((; S' S'' E') : ESTIMATE(S' E1'), ESTIMATE(S'' E2'),
    PLUS(E1' E2' E')).
ESTIMATE(X' 0): DST_OC(X' Y').
ESTIMATE(X' 0): SRC_OC(X' Y').
ESTIMATE((< X' Y') E') : ESTIMATE(X' E1'), ESTIMATE(Y' E2'),
    PLUS(E1' E2' E''), PLUS(E' 1 E').
ESTIMATE((OP' Y' E') : ESTIMATE(Y' E').
ESTIMATE((OP' X' Y') E') : ESTIMATE(X' E1'), ESTIMATE(Y' E2'),
    PLUS(E1' E2' E').
Appendix 4

Extracts of the MTs for PDP-11 and ND-100

The Mapping Tables (MTs) for PDP-11 and ND-100 are generated by the CGG from the TMDs for PDP-11 and ND-100 respectively. In this appendix we list only extracts of these MTs to show the working style of the CGG and the general shape of the MTs.

Extract of PDP-11 MT

### Part 1. The Compiler Implementation Decisions copied from the PDP-11 TMD.

```
ALLOCATABLE_REG(((R0 T((DATA LOCA) (BYTE WORD)) FREE),
(R1 T((DATA LOCA) (BYTE WORD)) FREE),
(R2 T((DATA LOCA) (BYTE WORD)) FREE),
(R3 T((DATA LOCA) (BYTE WORD)) FREE),
(R4 T((DATA LOCA) (BYTE WORD)) FREE))).
```

### Part 2. The machine operations copied from PDP-11 TMD, with some modifications.

#### Part 2.1. Access Modes, the same as in the TMD.

```
AM1({<} `{+(R'T(LOCA WORD)),(C' #C)},{@ C' ( R' ) }).
```

#### Part 2.2. Operand-classes. Cost information is omitted.

```
DST_OC(TREE',DST'):AM1{TREE',DST'}.  
```

#### Part 2.3. Instructions. They are regarded now as templates:

```
TEMPLATE(pattern',code-seq',R',R'',L'). Here the information about cost is omitted, but the register description R' (before the call) and R'' (after the call) and the static level L' are added. They are used for TN (registers or locations on run-time stack) allocation.
```

```
TEMPLATE({<- $1' (0 #C)},{INS CLR DST' _),R' R' L'}):DST_OC($1',DST').
```

#### Part 3. The new templates generated by the CGG from IL patterns which do not exist in the instruction description.

```
TEMPLATE({<- DST1' (* SRC1' (2 #C))),INS(ASL,DST1A',_),R',R', V27'):
DST_OC(DST1',DST1A'),
CONTENTS(DST1',SRC1').
```

```
TEMPLATE({<- DST1' (* SRC1' (2 #C)),
(INS(MOV,SRC1A',DST1A')) . INS(ASL, DST1A',_),R',R', V27'):
DST_OC(DST1',DST1A'),
SRC_OC(SRC1',SRC1A').
```

TEMPLATE({"<" DST1' (AND# SRC1' SRC2')),
(IN{MOV, SRC2A', DST1A'') INS{MOV, SRC1A', R0'}
INS{COrO, R0', _}. INS{BIC, R0', DST1A'), R', V26', V27') :
DST_OC(DST1', DST1A'),
SRC_OC(SRC1', SRC1A'),
SRC_OC(SRC2', SRC2A'),
ALLO_REG((R0' T{DATA, WORD}), R', V592'),
FREEREG((R0' T{DATA, WORD}), V592', V26').

TEMPLATE({"-> (NEQ SRC1' SRC2'') (LABEL' #RM)),
(IN{CMP, SRC1A', SRC2A'}), INS{BNE, LABEL', _}), R', R', V27') :
SRC_OC(SRC1', SRC1A'), SRC_OC(SRC2', SRC2A').
TEMPLATE({"-> (EQL SRC1' SRC2'') (LABEL' #RM)),
(IN{CMP, SRC1A', SRC2A'}) INS{BEQ, LABEL', _}), R', R', V27') :
SRC_OC(SRC1', SRC1A'), SRC_OC(SRC2', SRC2A').
TEMPLATE({"-> (LEQ SRC1' SRC2'') (LABEL' #RM)),
(IN{CMP, SRC1A', SRC2A'}), INS{BLE, LABEL', _}), R', R', V27') :
SRC_OC(SRC1', SRC1A'), SRC_OC(SRC2', SRC2A').

;; Part 4. Hand-coded templates

PARAMTEMPLATE({PARAM X'} INS{MOV SRC' "-(SP)"') R' R' DIR' C'):SRC_OC(X' SRC').
TEMPLATE({CALL NAME'} INS{JSR PC NAME'} R' R' L').

Extract of NO-100 MT


ALLOCATABLE_REG({{A T({DATA} {BYTE WORD}) FREE},
{X T({DATA LOCA} {BYTE WORD}) FREE}}).

;; Part 2. The machine operations.

;; Part 2.1. The access modes.
AM1{(+ (<) (+ (B T({LOCA WORD}) D' #C)))} (X T({LOCA WORD}))
{,,B,X D').

;; Part 2.2. The operand classes.
EL(IN' EL'); AM1{IN' EL'}. 

;; Part 2.3. The instructions.

;;
TEMPLATE((-> $(SRC1 SRC2) [+ PC 1]),
        (INS(LDA,SRC1A,_)) INS(SUB,SRC2A,_)) INS(SKPNQ,A,0)),
        R', V26', V27') :
        SRC_OC(SRC1',SRC1A'),
        SRC_OC(SRC2',SRC2A'),
        ALLO_REG((A T(DATA,WORD)),R', V517'),
        FREEREG((A T(DATA,WORD)),V517', V26').

TEMPLATE((-> $(SRC1 SRC2) [LABEL #RM]),
        (INS(LDA,SRC1A,_)) INS(SUB, SRC2A,_))
        INS(JAF,LABEL',_)) R', V26', V27') :
        SRC_OC(SRC1',SRC1A'),
        SRC_OC(SRC2',SRC2A'),
        ALLO_REG((A T(DATA,WORD)),R', V505'),
        FREEREG((A T(DATA,WORD)),V505', V26').

% Part 4. The hand-coded templates.

PARA_TEMPLATE((PARA (A T(DATA WORD))
        INS(STA (" X" DIR' C') _) R' R' DIR' C').
PARA_TEMPLATE((PARA (D' #C))
        (INS(SAA D' _)).INS(STA (" X" DIR' C') _) )
        R' R' DIR' C').

PARA_TEMPLATE((PARA $1')
        (INS(LDA EL' _)).INS(STA (" X" DIR' C') _) )
        R' R' DIR' C').

"TEMPLATE((CALL NAME')
        INS(JPL (" I " ABSOLUTE_ADDR(NAME')) _) R' R' L').
Appendix 5.

Examples of code derivation for complex IL patterns

In this appendix three IL patterns representing non-trivial basic blocks are submitted to the CGG and the optimal or near-optimal code sequences for PDP-11 and ND-100 are derived. The COST is defined as the number of words (16 bits in each word) taken by a code sequence. The ND-100 code sequence for Pattern 2 is 25% shorter than that in /Kozlak 1981/, while others are the same.

{ α β } means that, α and β are parallel trees.

Pattern 1.

\[
\{ (- DST1 (+ (+ DST2 SRC1)) (- DST2 (+ (+ DST1 SRC1))) \}
\]

**code sequence for PDP-11**

- ALLOREG((RO' T(DATA,WORD)))
- MOV DST2 RO'
- MOV DST1 DST2
- ADD SRC1 DST2
- MOV RO' DST1
- FREER((RO' T(DATA,WORD)))
- ADD SRC1 DST1

**code sequence for ND-100**

- LDA DST2
- ALLOCTEMP(M1')
- STA M1'
- LDA DST1
- ADD SRC1
- STA DST2
- LDA M1'
- ADD SRC1
- STA DST1

(COST = 13) (COST = 8)

Pattern 2.

\[
\{ (- DST1 (+ (+ (+ DST2 SRC1) (A T(DATA,WORD))))
\{ (- DST2 (+ (+ (+ DST2 SRC1) (( DST1))) \}
\]

**code sequence for PDP-11**

- ALLOREG((RO' T(DATA,WORD)))
- MOV DST1 RO'
- ADD SRC1 DST2
- MOV DST2 DST1
- ADD RO' DST2
- FREER((RO' T(DATA,WORD)))
- ADD A DST1

**code sequence for ND-100**

- ALLOCTEMP(M2'')
- STA M2'
- LDA DST2
- ADD SRC1
- ALLOCTEMP(M1')
- STA M1'
- ADD DST1
- STA DST2
- LDA M2'
- ADD M1'
- STA DST1

(COST = 12) (COST = 9)
Pattern 3.

{ (- DST1 (+ (- DST5) (- DST4)))
 (- DST2 (+ (- DST5) (- DST4)))
 (- DST3 (- DST5))
 (- DST4 (+ (- DST3) (- DST1)))
 (- DST5 (- DST1))
}

code sequences for PDP-11
code sequences for NO-100

ALLOREG((R1', T(DATA,WORD)))
MOV DST5 R1'
ALLOREG((RO', T(DATA,WORD)))
MOV DST4 RO'
MOV DST1 DST5
MOV DST3 DST4
ADD DST1 DST4
MOV R1' DST1
FREER((R1', T(DATA,WORD)))
MOV DST1 DST3
ADD RO' DST1
FREER((RO', T(DATA,WORD)))
MOV DST1 DST2

(COST = 23)

ALLOREG((RO', T(DATA,WORD)))
MOV DST4 RO'
MOV DST1 DST4
MOV DST5 DST1
MOV DST4 DST5
ADD DST3 DST4
MOV DST1 DST3
ADD RO' DST1
FREER((RO', T(DATA,WORD)))
MOV DST1 DST2

(COST = 22)

ALLOREG((RO', T(DATA,WORD)))
MOV DST5 RO'
MOV DST1 DST5
MOV DST4 DST1
MOV DST5 DST4
ADD DST3 DST4
MOV RO' DST3
ADD RO' DST1
FREER((RO', T(DATA,WORD)))
MOV DST1 DST2

(COST = 21)
Appendix 6
The CG in PROLOG

PROBLEM: REWRITE(LIUL), INCLUDE(MT),
GET_SIGN(SIGN' SIGN'), INSERT(SIGN(SIGN' SIGN')),
ALLOCATABLE_REG(REGDES'),
QQQ, LEVEL(L'), CODE_GENERATOR(L' L' REGDES' REGDES'),
RETRACT(LEVEL(LE')), INSERT(LEVEL(L'')), FAIL.

QQQ.QQQ: QQQ. LEVEL(0). ZZ using backtrack instead of recursive calling
CODE_GENERATOR(L' L' R' R' '):
READ(TERMINAL, SOURCE'),
NO(EQUAL(SOURCE END-OF-FILE)),
CASE(SOURCE L' L' R' R' '), ACCEPT.
CODE_GENERATOR(L' L' R' R' ').

ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZ MACHINE-INDEPENDENT STORAGE ALLOCATION. ZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
CASE(MAIN 0 0 R' R'):
READ(TERMINAL DEFINITION),
DATASTART(SIGN' C'), SPACEALLOC(0 C' C' ').
SPACEALLOC(L' C' C' '):
READ(TERMINAL DEF'),
INSERT(LAST(L' C'')), INSERT(TEMPSTART(L' C' ')), ACCEPT.
SPACEALLOC(L' C' C' '):
READ(TERMINAL DEF'),
SPACE_ALLOC(L' L' C' C' '), SPACEALLOC(L' C' C' ').
SPACE_ALLOC((NAME TYPE AMOUNT) L' C' C' '):
DATA_TYPE_CONVERSION(TYPE TYPE' SIZE ALIGN'),
ROUND_UP(C' ALIGN' CAP' SIGN'),
INSERT(OFFSET(NAME L' CAP' SIGN' TYPE')),
MULT(SIZE' AMOUNT SUM'),
STACK_GROW(CAP' C' SUM').

ROUND_UP(C' ALIGN' CAP' SIGN'):
TRY(X'), STACK_GROW(C' CAP' X'), DIV(CAP' ALIGN' W'),
SIGN(SIGN' SIGN'), ACCEPT.
TRY(0). TRY(1). TRY(2). TRY(3).

STACK_GROW(CAP' C' SUM '):
SIGN(SIGN' +),
PLUS(CAP' SUM' C' '), ACCEPT.
STACK_GROW(CAP' C' SUM '):
PLUS(C' SUM' CAP'), ACCEPT.
STACK_GROW(CAP' C' SUM '):
PLUS(C' CAP' SUM'), OPPOSITE(SIGN' SIGN'),
RETRACT(SIGN(SIGN' -')), INSERT(SIGN(SIGN' -')).

GET_SIGN(SIGN' SIGN '):
FRAME_DIRECTION(D'), DATASTART(SIGN', ABS'),
DECISION_TABLE(D' SIGN' SIGN'),
DECISION_TABLE(DOWN + -). DECISION_TABLE(UP '),
DECISION_TABLE(DOWN - +). DECISION_TABLE(UP + +).
ACTION(A’ L’ R’ R’’ BASE’ S’):
BIND(A’ A’’ L’ BASE’ S’ TYPE’), \[\text{II} \text{name-binding}\]
SEARCH(A’ CODE’ R’ R’’ L’), \[\text{II} \text{instruction selection}\]
OUTPUT(CODE’),
LAST(L’ C’),
TEMPSTART(L’ C’’),
RETRACT(TEMPSTART(L’ C’’)),
ASSERT(TEMPSTART(L’ C’’)),
ACCEPT.

MACHINE-INDEPENDENT DATA ACCESSING (NAME-BINDING)

BOTTOM-UP WALKING ON THE ACTION TREE

BIND((OP’ SON’) (OP’ SON’’) L’ BASE’ S’ TYPE-RETURN’):
BIND(SON’ SON’’ L’ BASE’ S’ TYPE-OPERAND’),
MA_OP(OP’ TYPE-OPERAND’ TYPE-RETURN’ OP’’).ACCEPT.

BIND((OP’ S1’ S2’’) (OP’’ S1’’ S2’’) L’ BASE’ S’ TYPE-RETURN’):
BIND(S1’ S1’’ L’ BASE’ S’ TYPE’),
BIND(S2’ S2’’ L’ BASE’ S’ TYPE’),
DY_OP(OP’ TYPE’ TYPE’ TYPE-RETURN’ OP’’).ACCEPT.

BIND((X’’ ; Y’’) (X’’ ; Y’’) L’ BASE’ S’ TYPE’):
BIND(X’ X’’ L’ BASE’ S’ TYPE’),
BIND(Y’ Y’’ L’ BASE’ S’ TYPE’’).ACCEPT.

BIND(NIL NIL L’ BASE’ S’ TYPE’).ACCEPT.

BIND CONSTANTS

BIND(NUMB’ (NUMB’ #C) L’ BASE’ S’ INT):
NUMERIC(NUMB’), \[\text{II} \text{predefined in PROLOG}\]
INTERMEDIATE_MODE(T’),
WITHIN(NUMB’ T’), ACCEPT,
WITHIN(N’ 16_bits):PLUS(N’ X’ 32768),
WITHIN(N’ 8_bits):PLUS(N’ X’ 128).

BIND(NUMB’ (BIGINT(NUMB’) #RM) L’ BASE’ S’ INT):
NUMERIC(NUMB’),
ASSERT(BIGINT(NUMB’)).

BIND VARIABLES

BIND(NAME’ NAME’’ L’ BASE’ S’ TYPE’):
ATOMIC(NAME’), \[\text{II} \text{predefined in PROLOG}\]
BINDING(NAME’ NAME’’ L’ BASE’ S’ TYPE’).
BINDING(NAME’ (SIGN’ BASE’ (C’ #C)) L’ BASE’ S’ TYPE’):
OFFSET(NAME’ L’ C’ SIGN’ TYPE’).ACCEPT.
BINDING(NAME’ NAME’’ L’ BASE’ (O’ DIR’) TYPE’):
PLUS(L’’ 1 L’’).
BINDING(NAME' NAME'' L'' BASE'' (O' DIR') TYPE').
  IFF(EQUAL(O' 0) EQUAL(BASE'' (<> BASE')),
      EQUAL(BASE'' (<> (DIR' BASE' (O' #C)))).

BIND ARRAY ELEMENTS

BIND((INDEX ARRAY' SUBS') (DIR' ARRAY'' SU'),
      L' BASE' S' TYPE'):
  IFF(FRAME_DIRECTION(DOWN) EQUAL(DIR' ->) EQUAL(DIR' *)),
      BIND((ARRAY' ARRAY'' L' BASE' S' TYPE'),
            BIND((SUBS' SUBS'' L' BASE' S' INT),
                 DATA_TYPE_CONVERSION(TYPE' TYPE'' SIZE', ALIGN'),
                 EQUAL(SU' (* (- SUBS'' (1 #C)) (SIZE' #C)))).

BIND LABELS

BIND((GOTO LABEL') (GOTO (LABEL' #RM)) L' BASE' S' TYPE').
BIND((LABEL LA') NIL L' BASE' S' TYPE'):
  WRITELN(LIUL (LA' ":")),ACCEPT.

INSTRTUCTION SELECTION (FETCH/STORE SUBTARGETTING)

-- Only for <-. The version for <= is similar --

The general scheme is that, if the destination is a memory mode, use STORE1 to get a register to perform the calculation. If the destination is a virtual mode, use STORE2 to calculate the address. FETCH always performs on register destination.

SEARCH(NIL _ R' R'' L''):ACCEPT.
SEARCH((<- X' Y') CODE' R' R'' L'):
  ZZdirect matching
      TEMPLATE((<- X' Y') CODE' R' R1' L'), ACCEPT,
      FREERE(X' R1' R'').
SEARCH((<- (REG' T(DATA T2')) Y') CODE' R' R'' L'):
      TEMPLATE((<- (REG' T(DATA T2')) Y') CODE' R' R'' L'),ACCEPT.
SEARCH((<- X' Y') CODE' R' R'' L'):
      DST_OC(X' X'),
      STORE1((<- X' Y') CODE' R' R'' L'),
SEARCH((<- X' Y') CODE' R' R'' L'):
      STORE2((<- X' Y') CODE' R' R'' L'). ZZaddress calculation

FETCH((<- X' Y') CODE' R' R'' L'):
  TEMPLATE((<- X' Y') CODE' R' R1' L'),
  FREERE(X' R1' R''),ACCEPT.
FETCH((<- (REG' T(DATA T2')) (<> SON')) (CODE''.CODE') R' R'' L'):
  TEMPLATE((<- (REG' T(DATA T2'))
                  (<> (REG' T(LOCA T2')))) CODE' R' R1' L'),
  FETCH((<- (REG' T(DATA T2')) (<> SON')) CODE''.CODE' R' R'' L'),ACCEPT.
FETCH((<- DST' (OP' SON')) (CODE''.CODE') R' R'' L'):
  MA_OP(_, _, _, OP'),
  TEMPLATE((<- DST' (OP' DST')) CODE' R' R1' L'),
  FETCH((<- DST' SON') CODE''.CODE' R1' R'' L'),ACCEPT.
FETCH((<- DST' (OP' S1' S2')) (CODE''.CODE') R' R'' L'):
  DY_OP(_, _, _, _, OP'),
  SRC_OC(S2' W'),
  TEMPLATE((<- DST' (OP' DST' S2')) CODE' R' R1' L'),
  FETCH((<- DST' S1') CODE' R1' R'' L'),ACCEPT.
FETCH((<- DST' (OP' S1' S2')) (CODE''.CODE') R' R'' L'):
  SRC_OC(S1' W'),
  TEMPLATE((<- DST' (OP' S1' DST')) CODE' R' R1' L'),
  FETCH((<- DST' S2') CODE' R1' R'' L'),ACCEPT.
FETCH((<- DST' (OP' S1' S2')) (C2'.C1'.CODE') R' R'' L'):
  OROR SRC_OC(S2' W'), MEMORYCONTENTS(S''),
  MISMATCH(S2' S2'' MIS' R' R1' L' INT'), ZZTN allocation
  TEMPLATE((<- DST' (OP' DST' S2')) CODE' R1' R'' L'),
FETCH((< DST' S1'>) C1' R'' R''' L'),
SEARCH(MIS' C2' R'' R'' L').

STORE1((< DST' SRC') (CODE'' CODE') R' R'' L'),
ALLO_REG((REG' T(DATA WORD)) R' R1'),
TEMPLATE((< DST' (REG' T(DATA WORD))) CODE' R1' R'' R''' L'),
FETCH((< (REG' T(DATA WORD)) SRC') CODE'' R'' R''' L'),
FREE_REG(DST' R'' R'' R').

STORE1((< DST' SRC') C1' R' R'' L'),
GET_REG((REG' T(DATA WORD)) R' R'),
SWAP((REG' T(DATA WORD)) C1' R1' R'' L' (< DST' SRC')).

STORE2((< DST' SRC') (CODE'' CODE') R' R'' L'),
OROR(DST_QC(DST'' X'), MEMORYLOCATION(DST'')),
MISMATCH(DST' DST'' MIS' R' R1' L' INT),
SEARCH(MIS' CODE'' R1' R'' L'),
SEARCH((< DST' SRC') CODE' R'' R' L').

MISMATCH(X' X'' MIS' R' R' R'' L' TYPE'): %register allocation
PEEPOP(X' X'' MIS' R' R'' L' TYPE').

PEEPOP((OP' SON') (OP' SON'') M' R' R'' L' TYPE'),
PEEPOP((OP' SON') (OP' SON'') M' R' R'' L' TYPE'),
PEEPOP((OP' S1' S2') (OP' S1'' S2''') (M1'; M2';) R' R'' L' TYPE'),
PEEPOP((OP' S1' S2') (OP' S1'' S2''') (M1'; M2';) R' R'' L' TYPE'),
PEEPOP(X' X' NIL R' R' L' TYPE'): ACCEPT.

MISMATCH(X' X'' (< (SIGN' BASE' (C' #C)) X') R' R' L' TYPE'): %memory alloc
OROR(MEMORYLOCATION(X''), MEMORYLOCATION(X''), FRAME_POINTER(BASE'),
TEMPO(C' TYPE' L' SIGN').

TEMPO(C' TYPE' L' SIGN'),
DATA_TYPE_CONVERSION(TYPE' TYPE'' SIZE' ALIGN'),
TEMPSTART(L' CT'),
ROUD_UP(ct' ALIGN' C' SIGN'),
RETRACT(TEMPSTART(L' CT'),
STACK_GROW(C' C'' SIZE'),
INSERT(TEMPSTART(L' C''')).

MEMORYLOCATION((SIGN' BASE' (C' #C))): FRAME_POINTER(BASE'),
MEMORYCONTENTS(< X>): MEMORYLOCATION(X').

ALLO_REG((REG' T(T1' T2')) R' R'): BOUND(RESULT),
FRAME_POINTER((REG' T(T1' T2'))), REJECT.
ALLO_REG((R' T(T1' T2')) ((R' T(T1'' T2'') FREE).REST'),
((R' T(T1'' T2'') BUSY).REST'): BELO(T1' T1'),
BELO(T2' T2'),
ACCEPT.
ALLO_REG(R' (H'.T') (H'.T''): ALLO_REG(R' T' T').
FREEREG((OP' SON') R' R'''):
  MA_OP(_, _, _, _, OP'), ACCEPT,
  FREEREG(SON' R' R''').
FREEREG((OP' S1' S2') R' R'''):
  DY_OP(_, _, _, _, OP'), ACCEPT,
  FREEREG(S1' R' R''''),
  FREEREG(S2' R'' R''').
FREEREG((REG' T(T1' T2'')) R' R'''):
  NO(FRAME_POINTER((REG' T(T1' T2''))),
    FREEREG((REG' T(T1' T2'')) R' R'''), ACCEPT.
FFREEREG((R' T') ((R' T' '_BUSY).REST'),
  ((R' T' 'FREE).REST'')): ACCEPT.
FFREEREG(X' (H'.T') (H'.T'')):
  FFREEREG(X' T' T''), ACCEPT.
FREEREG(R' R'' R')).

GET_REG((REG' T(T1' T2'')) ((REG' T(T1'' T2''') S') .REST')
  ((REG' T(T1'' T2''') BUSY).REST'')):
  BELONG(T1' T1''),
  BELONG(T2' T2''),
  ACCEPT.
GET_REG(R' (H'.T') (H'.T'')):
  GET_REG(R' T' T'').

SWAP((REG' T(DATA TYPE')) {C1' C3' C2'.C4'') R' R'' L' (<- DST' SRC')):
  FRAME_POINTER(BASE'),
  DATA_TYPE_CONVERSION(TYPE'" TYPE' _ _ ),
  TEMPO(C' TYPE' L' SIGN'),
  TEMPLATE((<- (SIGN' BASE' (C' #C)) (REG T(DATA TYPE'))
    C1' R' R1' L'),
  TEMPLATE((<- DST' (REG' T(DATA TYPE')) C2' R1' R2' L'),
  FETCH((<- (REG' T(DATA TYPE')) SRC') C3' R2' R3' L'),
  TEMPLATE((<- (REG' T(DATA TYPE')) (<> (SIGN' BASE' (C' #C))),
    C4' R3' R4' L'),
  FREEREG(DST' R4' R''').

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Z HIGH-LEVEL FLOW-CONTROL OPERATIONS
Z
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SEARCH((X' ; Y') (C1'.C2') R' R'' L'),
  SEARCH(X' C1' R' R'''' L''),
  SEARCH(Y' C2' R'''''' R'' L').
SEARCH((IF-THEN X' (<- Y' Z')) (V'.U') R' R' LE'): ZZskip
  TEMPLATE((<- Y' Z') U' R' R1' LE'), NO(EQUAL(U' (A'.B'))),
  CONTROL((-> (NOT X') (+ PC 1)) V' R1' R' LE').
SEARCH((IF-THEN X' Y') (V' U'.LABEL(L')) R' R' LE'): ZZjump
  SEARCH(Y' U' R' R''' LE'),
  CONTROL((-> (NOT X') (L' #RM)) V' R' R'' LE').
SEARCH((IF-THEN-ELSE X' Y' Z')
  (C1' C2' C3' LABEL(L') C4'.LABEL(L'')) R' R' LE'):
  SEARCH(Y' C4' R' R1' LE'),
  SEARCH((GOTO (L' #RM)) C3' R1' R2' LE'),
  SEARCH(Z' C2' R2' R3' LE'),
  CONTROL((<- X' (L' #RM)) C1' R3' R' LE').
SEARCH((WHILE X' Y') (LABEL(L') W' V' U'.LABEL(L'')) R' R' LE'):
  SEARCH((GOTO (L' #RM)) U' R' R1' LE'),
  SEARCH(Y' V' R1' R2' LE').
INSERT(PROCEDURE(NAME' L')),
OUTPUT((INS(COPY X,B) INS(COPY L A) .INS(STA ("B " -126) _))),
PLUS(L' 1 L''),
DATASTART(SIGN' C' ),
PA_SPACE_ALLOC(PALIST' L' C' C''),
SPACE_ALLOC(NAME' TYPE' 1) L' C' C' C' ,
OFFSET(NAME' L' CAP' DIR' TYPE'),
INSERT(RETURN_ADDRESS(L' CAP' DIR' TYPE')), READ(TERMIAL DEFINITION),
SPACEALLOC(L' C' C' C' '').

PA_SPACE_ALLOC({X' ; Y' } L' C' C' '):
PA_SPACE_ALLOC(X' L' C' C' ' ),
PA_SPACE_ALLOC(Y' L' C' C' ' ),
PA_SPACE_ALLOC(NIL L' C' C' ),
PA_SPACE_ALLOC(NAME' TYPE' NU') L' C' C' '):
SPACE_ALLOC(NAME' TYPE' NU') L' C' C' ' ).

CASE((PROC NAME' TYPE' PALIST') L' L' R' R' '):
PARAMETER(BEFORE),
INSERT(PROCEDURE(NAME' L' )),
WRITELN(LIUL RUN-TIME-ROUTINE-ENTERPROCEDURE),
PLUS(L' 1 L' ),
PARA_SIZE(PALIST' SUM' ),
PARA_ALLOC(PALIST' L' SUM' SUM' ),
DATASTART(SIGN' C' ),
SPACE_ALLOC(NAME' TYPE' 1) L' C' C' ' ,
OFFSET(NAME' L' CAP' DIR' TYPE'),
INSERT(RETURN_ADDRESS(L' CAP' DIR TYPE')), READ(TERMIAL DEFINITION),
SPACEALLOC(L' C' C' C' ' ).

PARA_SIZE({X' ; Y' } S' ) :
PARA_SIZE(X' S1' ),
PARA_SIZE(Y' S2' ),
PLUS(S1' S2' S' ).

PARA_SIZE(NIL 0).
PARA_SIZE(NAME' TYPE' NUMBER'), S' ) :
DATA_TYPE_CONVERSION(TYPE' TYPE' SIZE' ALIGN'),
MULT(SIZE' NUMBER' S' ).

PARA_ALLOC({X' ; Y' } L' S' S' '):
PARA_ALLOC(X' L' S' S' ' ),
PARA_ALLOC(Y' L' S' ' S' ' ).
PARA_ALLOC(NIL L' S' S' ).
PARA_ALLOC(NAME' TYPE' NUMBER') L' S' S' '):
DATA_TYPE_CONVERSION(TYPE' TYPE' SIZE' ALIGN'),
SIGN(SIG' SIGN' ),
OPPOSITE(SIG' SIGN' '),
INSERT(OFFSET(NAME' L' S' SIGN' ' TYPE')),
MULT(SIZE' NUMBER' ' SUM'),
PLUS(S' ' SUM' S' ).

CASE(PROCED L' L' R' R' '):
CONST_ALLOC,
ABSOLUTE_ALLOC,
PLUS(L' 1 L' ),
RETURN_ADDRESS(L' C' DIR' VOID),
WRITELN(LIUL RUN-TIME-ROUTINE-EXITPROCEDURE).
CASE({PROCEDURE RETURNTYPE'}) L' L'' R' R'':%move return value to register
CONST_ALLOC,
ABSOLUTE_ALLOC,
PLUS(L'' 1 L'),
RETURN_ADDRESS(L' C' DIR' RETURNTYPE'),
DATA_TYPE_CONVERSION(RETURNTYPE' TYPE' SIZE' ALIGN'),
ALLOC_REG({REG' T(DATA TYPE'')} R' R1'),
FRAME_POINTER(BASE'),
RETURN_ADDRESS(L' C' DIR'),
TEMPLATE({<- (REG' T(DATA TYPE')) } (<> (DIR' BASE' (C' #C'))),
CODE' R1' R'" L'),
FREE_REG({REG' T(DATA TYPE'')} R'" R'"),
OUTPUT(CODE'),
INSERT(RETURN_VALUE({REG' T(DATA TYPE')}) ),
WRITELN(LIUL RUN-TIME-Routine-ExitProcedure).

CONST_ALLOC:BIGINT(C'),RETRACT(BIGINT(C')),
WRITELN(LIUL {BIGINT(C') "$" C'}),
CONST_ALLOC.

ABSOLUTE_ALLOC:ABSOLUTE_ADDR(NAME'),RETRACT(ABSOLUTE_ADDR(NAME')),
WRITELN(LIUL {ABSOLUTE_ADDR(NAME') "$" NAME'}),
ABSOLUTE_ALLOC.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XX  PROCEDURE CALL HANDLING  XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXX  BIND PROCEDURE CALL  XXXXXXXXXXXXXXXXXXXX
BIND({CALL NAME'} {CALL NAME''} L' BASE' S' TYPE'): 
INTERMEDIATE_MODE(16_bits),
BINDPROCNAME(NAME' NAME'' L'),ACCEPT.
BIND({CALL NAME'} {CALL ABSOLUTE_ADDR(NAME'')} L' _ _ _):
BINDPROCNAME(NAME' NAME'' L'),
ASSERT(ABSOLUTE_ADDR(NAME'')).
BINDPROCNAME(NAME' NAME'' L'): 
PROCEDURE(NAME' L'),ACCEPT.
BINDPROCNAME(NAME' NAME'' L':
PLUS(L'' 1 L'),
BINDPROCNAME(NAME' NAME'' L'').
BIND(RETURN (REG' T(DATA T2'')) L' BASE' S' TYPE'): 
RETURN_VALUE({REG' T(DATA T2'')}),ACCEPT.

### PARAMETER_PASSING (Only for PARA. The version for B_PARA is similar ###

SEARCH({PARA X'}) CODE' R' R' LE'):PARAMETER(BEFORE),
SEARCHPARA({PARA X'}) CODE' R' R' LE' DIR' C'),ACCEPT.

SEARCHPARA({PARA X'}) CODE' R' R' LE' DIR' C');
PARAEMPLATE({PARA X'}) CODE' R' R' DIR' C').
SEARCHPARA({PARA X'}) (C1'.C2') R' R' LE' DIR' C'); 
CALCULX(X' X' C1' R' R'" LE' INT).
PARAEMPLATE({PARA X'}) C2' R'" R'" DIR' C').
SEARCH({PARA X'}) CODE' R' R' LE'):  PARAMETER(AFTER),
FFS,
RETRACT(FFS),
WRITELN(LIUL RUN-TIME-Routine-EnterProcedure),
DATA_TYPE_CONVERSION(INT TYPEM' SIZE').
DATASTART(SIGN ' C'),
ALLO_REG((X T(LOCA WORD)) R' R1'),
SEARCHPARA((PARA X') CODE ' R' R' ' LE' SIGN ' C'),
STACK_GROW(' C' ' SIZE'),
INSERT(PARAPosition(C'')).
SEARCH((PARA X') CODE ' R' R' ' LE'?):
RETRACT(PARAPosition(C''),
DATA_TYPE_CONVOLUTE(IN'TypeM' 'SIZE' 'ALIGN'),
SIGN(DIR ' DIR'),
SEARCHPARA((PARA X') CODE ' R' R' ' LE' DIR ' C'),
STACK_GROW(' C' ' SIZE'),
INSERT(PARAPosition(C'')).

SEARCH((CALL NAME') CODE ' R' R' ' LE'?):
TEMPLATE((CALL NAME') CODE ' R' R1' LE'),
IFF(PARAMETER(BEFORE) ACCEPT ASSERT(FFS)),
FREE_REF((X T(LOCA WORD)) R1' R'').

ZII ZZ USEFUL PREDICATES ZZ
ZII

OPPOSITE(+ -). OPPOSITE(- +).

FFS.
EQUAL(X' X').
NO(X'):X', REJECT.
NO(X').
OROR(X' Y'):X', ACCEPT.
OROR(X' Y'):Y'.
ANDAND(X' Y'):X', Y'.
BOUND(SOME-STRANGE-AND-NOT-OTHERWISE-USED-CONSTANT):REJECT.
BOUND(X').
FREE?(X'):NO(BOUND(X')).
IFF(X' Y' Z'):X', ACCEPT, Y'.
IFF(X' Y' Z'):Z'.
BELONG(X' (X' Y')):ACCEPT.
BELONG(X' (Z' Y')):BELONG(X' Y').
RETRACT(X'):X', DELETE.
RETRACT(X').
ASSERT(X'):X', ACCEPT.
ASSERT(X'):INSERT(X').

OUTPUT(_):ACCEPT.
OUTPUT((X' Y')):OUTPUT(X'), OUTPUT(Y'), ACCEPT.
OUTPUT(IN(S' Y ' _)): WRITE(LIUL X'),
WRITE(LIUL Y'), ACCEPT.
OUTPUT(IN(S' Y' Z')): WRITE(LIUL X'),
WRITE(LIUL Y'),
WRITE(LIUL Z'), ACCEPT.
OUTPUT(LABEL(X'')):
WRITE(LIUL X'), WRITELN(LIUL ": ").