



JOHANNES KEPLER
UNIVERSITÄT LINZ

Netzwerk für Forschung, Lehre und Praxis



Automatic Code Generation using Dynamic Programming Techniques

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

INFORMATIK

Eingereicht von:

Igor Böhm, 0155477

Angefertigt am:

Institut für System Software

Betreuung:

o.Univ.-Prof.Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Linz, Oktober 2007

Abstract

Building compiler back ends from declarative specifications that map tree structured intermediate representations onto target machine code is the topic of this thesis. Although many tools and approaches have been devised to tackle the problem of automated code generation, there is still room for improvement. In this context we present **HBURG**, an implementation of a code generator generator that emits compiler back ends from concise tree pattern specifications written in our code generator description language. The language features attribute grammar style specifications and allows for great flexibility with respect to the placement of semantic actions. Our main contribution is to show that these language features can be integrated into automatically generated code generators that perform optimal instruction selection based on tree pattern matching combined with dynamic programming. In order to substantiate claims about the usefulness of our language we provide two complete examples that demonstrate how to specify code generators for **RISC** and **CISC** architectures.

Kurzfassung

Diese Diplomarbeit beschreibt **HBURG**, ein Werkzeug das aus einer Spezifikation des abstrakten Syntaxbaums eines Programms und der Spezifikation der gewünschten Zielmaschine automatisch einen Codegenerator für diese Maschine erzeugt. Abbildungen zwischen abstrakten Syntaxbäumen und einer Zielmaschine werden durch Baummuster definiert. Für diesen Zweck haben wir eine deklarative Beschreibungssprache entwickelt, die es ermöglicht den Baummustern Attribute beizugeben, wodurch diese gleichsam parametrisiert werden können. Darüber hinaus hat man die Möglichkeit Baummuster an beliebigen Stellen mit semantischen Aktionen zu versehen. Wir zeigen, dass diese Spracheigenschaften in solchen Codegeneratoren anwendbar sind, die auf der Technik der Baummustererkennung und dynamischer Programmierung basieren. Zwei Beispiele der Codegenerator Spezifikationen für **RISC** und **CISC** Zielmaschinen sollen die Mächtigkeit der entwickelten Sprache aufzeigen.

Table of Contents

1	Introduction	1
1.1	Structure of this Master Thesis	2
2	Background Information	3
2.1	Overview of Compilation	3
2.2	Transformation of Representations	4
2.2.1	From Input Language to Intermediate Representation	5
2.2.2	Intermediate Representation Transformations	6
2.2.3	From Intermediate Representation to Target Code	6
2.3	Summary	7
3	State of the Art in Code Generation	9
3.1	Overview	9
3.2	Syntax Directed Code Generation	11
3.3	Tree Pattern Matching with Dynamic Programming	11
3.3.1	Dynamic Programming at Compile Time	12
3.3.2	Dynamic Programming at Compile-Compile Time	13
3.3.3	Single-Pass Optimal Tree Pattern Matching	13
3.4	Finite-State Code Generation	14
3.5	Summary	15
4	Tree Pattern Matching with Dynamic Programming	17
4.1	Motivation	18
4.2	Fundamental Algorithms	20
4.2.1	Calculating Optimal Instruction Sequence	20
4.2.2	Emitting Optimal Instruction Sequence	23
4.3	Code Generation Description Language	24
4.3.1	General Structure	25
4.3.2	Rewrite Rules	26
4.3.3	Tree Pattern Costs	27
4.3.4	Semantic Actions	27
4.3.5	Attributes	28

4.4	Summary	30
5	HURG - Haskell Bottom Up Rewrite Generator	31
5.1	Architecture	32
5.1.1	Intermediate Representation	33
5.1.2	Context Sensitive Analysis	36
5.1.3	Optimal Instruction Sequence Calculation	37
5.1.4	Optimal Instruction Sequence Code Generation	40
5.2	Invoking HURG	42
5.3	Integrating HURG Generated Code	43
5.4	Summary	45
6	Conclusions	47
6.1	Summary of Contributions	48
6.2	Future Work	49
	Appendices	51
A	Language Syntax	53
B	Code Generation Examples	55
B.1	RISC Code Generator	60
B.1.1	RISC Architecture Overview	60
B.1.2	RISC Code Generator Implementation	62
B.1.3	RISC Assembly Output	70
B.2	CISC Code Generator	72
B.2.1	CISC Architecture Overview	72
B.2.2	CISC Code Generator Implementation	74
B.2.3	CISC Assembly Output	83
C	Automatically Generated Code	85
C.1	IR Tree Tiling	85
C.2	Semantic Action Execution	87
	List of Figures	91
	List of Listings	93
	Bibliography	95

— *If we are certain that our job is to produce compilers for a single architecture, there may be no advantage in using automatic methods to generate a code generator from a machine description. [...] If, on the other hand, we expect to be producing compilers for several architectures, generating code generators automatically from machine descriptions may be of great value.*

Steven S. Muchnick [20]

1

Introduction

This chapter gives a general introduction to the goals and main contributions of this master thesis. Finally, the structure of the thesis is outlined.

Tools automating the construction of code generators such as **JBURG** [16], **BEG** [5], **Twig** [1], **GBURG** [7], and **BURG** [2] exist, but there is still room for advances. The goal of this master thesis is to improve the state of the art of automatic code generation by devising a powerful code generator description language together with a reference implementation of a code generator generator. Our language consists of rewrite rules augmented with costs and semantic actions. The resulting code generators perform optimal instruction selection based on tree pattern matching and dynamic programming.

In this master thesis, we make the following contributions:

- We demonstrate that it is possible to use an attribute grammar formalism together with code generators based on tree pattern matching and dynamic programming (Section 4.3.5).
 - We show that our code generator description language offers more flexibility with respect to the placement of semantic actions than the languages accepted by the previously mentioned code generator generators (Section 4.2.2 and 4.3.4).
 - We introduce a special language construct that captures the notion of linked subtrees. It turns out that this construct is quite useful since
-

linked subtrees modeling sequential statements can easily be processed by our code generator description language (Section 4.1).

When compared to other code generator description languages, the overall structure of our language is similar. However, we believe that no other language that yields code generators based on tree pattern matching and dynamic programming offers (1) support for attribute grammars, (2) patterns that capture linked subtrees, (3) and such a degree of flexibility with respect to the placement of semantic actions.

1.1 Structure of this Master Thesis

Chapter 2 of this thesis outlines the structure of a conventional compiler and introduces the process of compilation as a sequence of transformations. It establishes common terminology and introduces the problem of automated code generation. The following Chapter 3 reviews the state of the art in automated code generation, outlining various approaches that have been taken so far.

Our code generator description language together with fundamental algorithms implemented in our code generator are described in Chapter 4. Finally, Chapter 5 outlines the architecture of **HBURG**¹, our code generator generator implementation, followed by a description of how to integrate an automatically constructed code generator into a compiler.

For compiler developers interested in using our code generator description language together with its reference implementation it should suffice to consult Chapter 4 together with Appendix A as a language reference, and sections 5.2 and 5.3 as a reference to our code generator generator implementation. Appendix B offers complete examples of code generator descriptions for **RISC** and **CISC** target architectures.

¹**HBURG** - Haskell Bottom Up Rewrite Generator

2

Background Information

This chapter presents the process of compilation as a sequence of transformations. First, the structure of a typical compiler is outlined, establishing common terminology. Then, the main transformations a compiler performs in order to translate a source language program into a target language program, are identified. After demonstrating how some transformations occurring in the compiler front end can be automated, the focus is directed to the compiler back end, namely the automation of instruction selection.

2.1 Overview of Compilation

To the end user a compiler should behave like a black box taking a computer program in one language as its input, and producing a translated computer program as its output. This chapter deals with the structure of the compiler black box, its internals, and their interaction. Figure 2.1 depicts a typical infrastructure found in most modern compilers:

- The *front end* of a compiler is responsible for understanding the syntax and semantics of the source program. During this phase of compilation it verifies the lexical and grammatical structure of the input. Context sensitive analysis (CSA) statically verifies the correctness of the
-

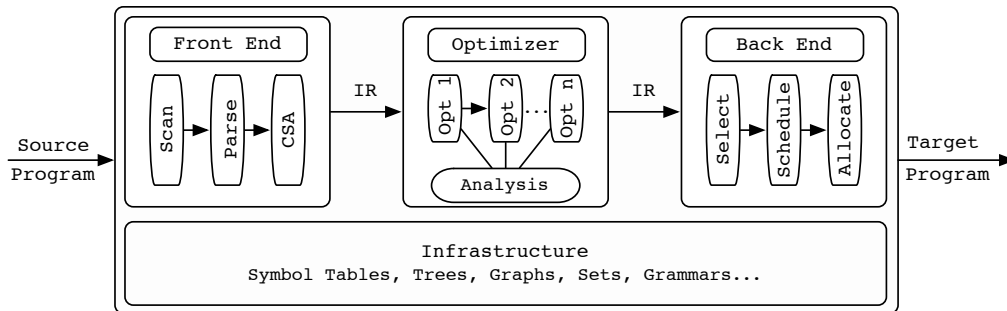


Figure 2.1: Structure of a typical compiler [3].

source program semantics (e.g. type inference, scope analysis). Ideally this phase also aids the programmer with helpful error messages during program development.

- The *middle part* consists of a sequence of optimization transformations. Optimization methods such as constant folding, strictness analysis, redundancy elimination, scalar optimization and strength reduction can be used.
- Finally the *back end* takes the optimized source program and produces a target program. A target program can be anything between a high level language and low level machine code, but since conventional compilers target machine code, we will focus on the latter form.

The “Infrastructure” box at the bottom of Figure 2.1 highlights the importance of choosing efficient data structures and automation tools since those choices greatly impact the performance, resource usage, and complexity of the compiler.

2.2 Transformation of Representations

In order to produce machine code from a high level program, each part of the compiler operates and possibly modifies an intermediate representation (IR) of the source program. Both, transformations that occur during compilation, and the representations they operate on, are the subject of the following sections.

2.2.1 From Input Language to Intermediate Representation

The front end maps a high level source language onto an IR (sometimes called an abstract syntax tree) by scanning input tokens and then parsing them in order to recognize the language syntax. This transformation also records information for later use during context sensitive analysis.

Input Language	Parser Generator Description Language	Intermediate Representation
5 - 2 * 3	<pre> Exp : Exp '+' Term { Add \$1 \$3 } Exp '-' Term { Sub \$1 \$3 } Term { Term \$1 } Term : Term '*' num { Mul \$1 \$3 } Term '/' num { Div \$1 \$3 } num { Num \$1 } </pre>	<pre> graph TD Sub((Sub)) --- Num5[Num 5] Sub --- Mul((Mul)) Mul --- Num2[Num 2] Mul --- Num3[Num 3] </pre>

Figure 2.2: Example demonstrating a transformation from input language to intermediate representation using the **HAPPY**¹ parser generator.

Extensive study and research in the areas of scanning and parsing has lead to the availability of efficient tools that automate most tasks of scanner and parser construction for a wide variety of input grammars. So the compiler writer only needs to specify the syntax using a convenient specification language, and provide semantic actions to be emitted upon the recognition of syntactic phrases.

The example in Figure 2.2 demonstrates how an input language of arithmetic expressions can be transformed into an abstract syntax tree intermediate representation. The example utilizes the **HAPPY**¹ parser generator description language. Non terminals are specified with an initial upper case letter, terminals are lowercase, and semantic actions are placed between curly braces at the end of each production. Productions of the same non terminal are separated by the pipe “|” symbol.

Type checking and context sensitive analysis is also performed during the first transformation of representations. Some compilers preserve and maintain types throughout compilation (see Shao and Appel [26], and Tarditi et al. [27]), thus later compilation phases like optimization and code generation can benefit from the additional information. Maintaining type information

¹**HAPPY** - a parser generator for Haskell <http://www.haskell.org/happy/>

throughout each program transformation is also helpful during compiler development. Peyton Jones and Santos [14] argue that checking types after each transformation is “an outstandingly good way to detect incorrect transformations”, making it easier to detect and pin point errors.

2.2.2 Intermediate Representation Transformations

The compiler optimization phase takes an IR produced by the front end as its input, and performs IR to IR transformations yielding an *improved* IR after each transformation. Improved in this context means that the transformed IR results in faster execution time, more compact code, or more power-efficient code.

An optimizer can make several passes over an IR transforming it into more efficient data structures and gathering enough information in order to perform certain types of optimizations.

Since we do not focus on compiler optimization this section is kept rather short, and included for the sake of completeness. The interested reader may like to consult Muchnick [20], Cooper and Torczon [3], as well as Peyton Jones and Santos [14], for more information on compiler optimization techniques.

2.2.3 From Intermediate Representation to Target Code

Finally the compiler back end produces machine specific target code from an optimized intermediate representation. Emitting target-specific machine code implies selecting the right instructions implementing the given IR operations, choosing an execution order, as well as deciding which values can safely reside in registers and which have to be put into memory locations. Regarding aspects of optimization, Peyton Jones and Santos [14] argue that this last transformation of representations should only “include optimizations ... if they cannot be done by a [intermediate to intermediate] transformation”.

Again there is room for automation during the instruction selection phase denoted by the “Select” box in Figure 2.1. Given a tree-structured IR, the compiler writer needs to specify all possible IR tree patterns using a convenient specification language, and provide semantic actions to be emitted upon the recognition of those patterns. A code generator generator then yields code that performs instruction selection given such a specification.

Suppose we want to generate code for a contrived and simplified RISC-

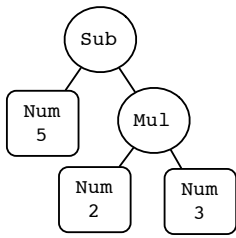
Intermediate Representation	Code Generator Description Language	Target Language
 <pre> graph TD Sub((Sub)) --- Num5[Num 5] Sub --- Mul((Mul)) Mul --- Num2[Num 2] Mul --- Num3[Num 3] </pre>	<pre> reg<.out Reg a.> (. a = getReg(); .) = Mul (reg<.out b.>, reg<.out c.>) (. PutMul(a,b,c); .) :2 Sub (reg<.out b.>, reg<.out c.>) (. PutSub(a,b,c); .) :2 Num v (. PutLoad(a,v); .) :1 . </pre>	<pre> loadi r1,5 loadi r2,2 loadi r3,3 mul r4,r2,r3 sub r5,r1,r4 </pre>

Figure 2.3: Example demonstrating a transformation from intermediate representation to target code using the **HBURG**² code generator.

like architecture only capable of performing basic arithmetic operations with values residing in registers, and the ability to load immediate values into registers. The example in Figure 2.3 should give a taste of how a tree-structured IR can be transformed into such an architecture using **HBURG**'s² code generator description language. IR nodes are specified with upper case letters, the values they produce are all lower case. Semantic actions are placed between “(.” and “.)”, input and output attributes are specified between “<.” and “.>”. Several IR tree patterns may produce the same value and the pipe “|” symbol separates such production patterns. Full details of **HBURG**'s code generator description language and its semantics are given in section 4.3.

2.3 Summary

A typical compiler consists of a front end recognizing the input language, an optimizing middle part, and a code generating back end. Developing a good optimizing compiler is a complex task but fortunately some compilation phases, the most prominent being scanning and parsing, can be automated.

The focus of this thesis is the automation of instruction selection in the compiler back end. For the compiler writer there are several benefits of automating the generation of code which performs instruction selection. Highly

²**HBURG** - Haskell Bottom Up Rewrite Generator

repetitive code sequences and algorithms must not be implemented by hand but are automatically generated. This leaves less room for errors and gives the chance to focus on the essential parts of instruction selection, namely the production of good target code.

3

State of the Art in Code Generation

This chapter gives an account of the state of the art in automated code generation. First, the genesis of automated code generation based on LR parsing techniques is introduced. Then, more recent approaches to optimal code generation based on tree pattern matching combined with dynamic programming are outlined and contrasted with each other. Finally, a code generation approach based on finite-state machine theory is presented. It is very efficient with respect to runtime and code size but does not guarantee to produce optimal code due to its greedy matching algorithm.

3.1 Overview

Easing the task of building a retargetable compiler by automating the generation of code, has been a goal since the early history of compilers. Proebsting [24], and Aho, Ghanapathi and Tjiang [1] give a historical account about the various approaches that have been devised in order to deal with the problem of automatic code generation.

While code generators use different instruction selection algorithms, their specifications are quite similar and typically consist of tree rewrite rules that associate a tree pattern with every instruction of the target machine. The patterns consist of operators denoting the operators of the IR nodes, and

```

 $stmt =$  ASGN ( $mem$   $m$ ,  $reg$   $a$ ) ( $\text{.put}(\text{MOV}, \underline{m}, \underline{a}); \text{.}$ ) : 3
      | ASGN ( $mem$   $m$ ,  $imm$   $i$ ) ( $\text{.put}(\text{MOV}, \underline{m}, \underline{i}); \text{.}$ ) : 2.
 $imm =$  CNST  $c$  ( $\text{.newImm}(\underline{c}); \text{.}$ ) : 0.
 $reg =$  ADD  $op$  ( $reg$   $a$ ,  $reg$   $b$ ) ( $\text{.put}(\underline{op}, \underline{a}, \underline{b}); \text{.}$ ) : 2
      | ADD  $op$  ( $reg$   $a$ ,  $imm$   $i$ ) ( $\text{.put}(\underline{op}, \underline{a}, \underline{i}); \text{.}$ ) : 1
      | ADD  $op$  ( $reg$   $a$ ,  $mem$   $m$ ) ( $\text{.put}(\underline{op}, \underline{a}, \underline{m}); \text{.}$ ) : 3
      | DIV  $op$  ( $reg$   $a$ ,  $imm$   $i$ ) ( $\text{.put}(\underline{op}, \underline{a}, \underline{i}); \text{.}$ ) : 1
      | DIV  $op$  ( $reg$   $a$ ,  $reg$   $b$ ) ( $\text{.put}(\underline{op}, \underline{a}, \underline{b}); \text{.}$ ) : 2.
 $mem =$  VAR  $v$  ( $\text{.newVar}(\underline{v}); \text{.}$ ) : 2
      | ADD  $op$  ( $mem$   $m$ ,  $imm$   $i$ ) ( $\text{.put}(\underline{op}, \underline{m}, \underline{i}); \text{.}$ ) : 2
      | ADD  $op$  ( $mem$   $m$ ,  $reg$   $a$ ) ( $\text{.put}(\underline{op}, \underline{m}, \underline{a}); \text{.}$ ) : 3.

```

Listing 3.1: CISC Code generator specification example.

operands denoting the storage classes of operands on the target machine (e.g. immediate, register, memory). Every pattern is associated with a semantic action that is executed when this pattern is selected as well as with costs that this pattern contributes to the overall costs of the generated code.

The goal is to cover the IR tree with the available patterns in such a way that the overall costs become minimal. When the IR tree has been covered, the semantic actions of the selected patterns are executed in a top-down way generating the corresponding instructions.

Listing 3.1 provides a sample grammar specification with 11 rewrite rules (two for *stmt*, one for *imm*, five for *reg*, and three for *mem*). *Non terminals* denote storage classes. They are written in lower case and appear on the left-hand side of rules. *Terminals* denote operators. They are written in upper case and represent IR nodes, also referred to as IR language operators. There are two types of rewrite rules:

- *Base rules* include a terminal symbol on the right hand side of rules.
- *Chain rules* derive one non terminal from another.

Costs associated with rewrite rules appear after a colon although not all code generator systems require cost annotations. *Semantic actions* are placed between (. and .). *Bindings* to terminals and non terminals that can be referred to within semantic actions are underlined.

The following sections highlight the key ideas and insights behind each code generator approach. We will focus on *optimal* code generation for *irregular* architectures using non-exhaustive algorithms.

3.2 Syntax Directed Code Generation

One of the earliest code generation methods based on LR(1) (Left-to-right Right-canonical-derivation) parsing was developed by Glanville [10], and Graham and Glanville [11]. This approach operates on a low-level linearized IR in polish-prefix form. Possible IR patterns are captured by rules similar to those in a context-free grammar. When a rule matches, its corresponding semantic action implementing the appropriate machine code sequence is executed. So the IR is covered by target code instructions once a parse of its linearized polish-prefix form has been found.

According to Graham and Glanville [11] machine description grammars tend to be ambiguous since most operators can access their operands in a variety of ways. An example are operators that accept their operands only from a specific subrange of registers (e.g. IA-32 DIV instruction). Such semantic information needs to be encoded syntactically and may result in ambiguities leading to shift-reduce and reduce-reduce conflicts during LR parsing. Shift-reduce conflicts are resolved in favor of a shift action, and reduce-reduce conflicts are resolved in favor of a reduction by the rule with the longest right hand side. This conflict resolution strategy has the effect of choosing production rules with long right-hand sides, thus reducing the amount of grammatical reductions, and consequently the amount of emitted machine instructions.

Ganapathi [8], and Ganapathi and Fischer [9] extended the Graham-Glanville approach by using an attribute grammar that allows to specify predicates to be used in ambiguous situations. This results in grammar productions that specify the general form of machine instructions, and semantic attributes and predicates that specify architectural restrictions.

3.3 Tree Pattern Matching Combined with Dynamic Programming

Another approach to optimal code generation combines tree pattern matching with dynamic programming. This strategy implies a tree-structured IR which is mapped onto target machine code by specifying tree patterns as rewrite rules annotated with costs. The ultimate goal of the tree pattern matcher is to cover the IR tree with tree patterns in such a way that the overall costs become minimal. In other words, a minimum-cost rewrite rule

cover for the IR tree must be identified by the matcher.

What we have not discussed yet is the strategy used by the matcher to find rewrite sequences for a given IR. Horspool and Scheunemann [12] give examples demonstrating the size of the search space a matching algorithm has to consider in order to select an optimal rewrite sequence, rendering an exhaustive approach unpractical. Instead, the dynamic programming algorithm outlined by Aho, Ganapathi and Tjiang [1] describes a non-exhaustive approach to find an optimal rewrite sequence for a given IR. The basic idea is to recursively partition the problem of generating an optimal rewrite sequence for an expression tree T , into subproblems of finding optimal rewrite sequences for the subtrees of T . This dynamic programming algorithm is integrated with the tree-matching process.

The following sections describe three approaches to tree pattern matching combined with dynamic programming. The first delays dynamic programming until the code generation phase during compiling, whereas the second moves the dynamic programming part to code generator generator construction time by building a bottom up rewrite system automaton. Both, the first and the second approach need two passes over the IR, one bottom-up pass to identify the optimal rewrite sequence by labeling the tree with dynamic programming information, and a top down traversal that emits optimal target code based on the decisions made during the previous pass. The third approach uses only a single pass and does not need an explicit IR to emit target code at the expense of only being able to process a proper subset of grammars that two-pass systems can handle.

3.3.1 Dynamic Programming at Compile Time

This section describes tree pattern matching code generator generators which apply dynamic programming at compile time, during instruction selection. Fraser, Hanson and Proebsting [6] argue that this approach produces code generators that “are fast, compact, and easy to understand”.

Given a cost-augmented tree pattern specification with semantic actions implementing the tree patterns, two passes over the IR are needed to produce optimal target code. The first bottom-up left-to-right pass labels the IR tree with tree patterns that cover it with minimum cost. The minimum-cost cover is calculated *explicitly* during instruction selection using dynamic programming. The second top-down pass executes the semantic actions that are associated with the selected tree patterns.

Since our prototype **HBURG** code generator implementation is based on this approach, a detailed description of the fundamental concepts and algorithms is described in chapters 4 and 5. Horspool and Scheunemann [12] provide a good overview of the dynamic programming approach used at compile time and give a comparison to the Graham-Glanville code generation approach. Fraser, Hanson and Proebsting [6] give a good outline on how to construct a code generator which uses dynamic programming at compile time.

3.3.2 Dynamic Programming at Compile-Compile Time

Pelegri-Llopart [21], the originator of bottom-up rewrite system (BURS) theory, was the first to recognize that dynamic programming could be done prior to instruction selection at compile-compile time, namely during code generator construction. Code generators based on BURS theory are significantly faster than other approaches that manipulate costs explicitly during the instruction selection phase of a compiler. This is possible because all dynamic programming is done when the BURS automaton is built.

The BURS automaton which finds the optimal rewrite sequence for an IR is a simple state-transition machine that given the operator at a node and the states of its children, determines the optimal rewrite sequence via a table lookup. So the first bottom-up traversal encodes all optimal rewrite sequences based on the BURS automaton built during compile-compile time. The second top-down pass uses the selected rewrite sequences to emit target code.

Generating an efficient BURS automaton is the main difficulty which arises when trying to implement a BURS code generator. Since all dynamic programming decisions are done at compile-compile time, state transitions and table lookups must be fast. For typical machine architectures it is inefficient to naively generate BURS states and state transition tables due to large table sizes. Proebsting [24] [23] demonstrates how to reduce table sizes by applying various optimizations, thus increasing the efficiency of BURS-style code generator generators.

3.3.3 Single-Pass Optimal Tree Pattern Matching

The previous two approaches defer emission of optimal target code until a minimum-cost cover of the *complete* IR has been found. Proebsting [25] devises a code generation system which is able to parse an IR and emit

optimal code in a single bottom-up pass. The tree pattern grammars that can be processed by this approach are a proper subset of those that the previously described two pass systems can handle. The most striking aspect of this approach is that it obviates the need for an explicit IR altogether because there is no reason to retain a complete IR if the code generator can emit code in a single pass.

The key idea behind this approach is that for some IR nodes there exists only one rewrite rule, also referred to as a base rule. Such base rules can be applied right away. For all other IR nodes where the matcher cannot determine the optimal rewrite rule immediately, it consults a small buffering stack that is used to record previously seen operations for such deferred matches. So there is no need to explicitly allocate an IR but some amount of bookkeeping must be performed in case an optimal decision at an IR node can not be made and must be delayed.

Despite the fact that a single-pass system can only parse a proper subset of two-pass system grammars, Proebsting [25] argues that “most useful grammars, including those describing the SPARC, the MIPS R3000, and the x86 architectures, fall within this subset”.

3.4 Finite-State Code Generation

For compilation environments where code generator speed and size are of utmost importance, Fraser [7] devised a finite-state machine pattern matching approach yielding tiny and fast code generators.

This approach works on IR's in postfix representation and uses *greedy* pattern matching instead of dynamic programming and unrestricted tree-matching. The matching algorithm makes a *locally* optimum choice at each stage with the hope of finding the global optimum. Base rules are matched immediately (greedy matching) and the application of chain rules is deferred until the next instruction is examined. This strategy can be encoded as a finite-state machine where the current operator is the input symbol and the last non terminal is the state. As a result this method does not guarantee to find an optimal global solution because that would imply looking arbitrarily far ahead.

Because the finite-state approach has to make decisions without complete information, this can lead to the generation of sub-optimal code for target architectures with redundant operations and complex addressing modes (e.g.

IA-32). Given a target architecture that eliminates redundant operations and provides a uniform addressing mode, the finite-state code generation approach fares quite well. The Lean Virtual Machine (LVM) devised by Fraser [7] is a stack based machine designed for efficient translation of machine code by eliminating redundant operations, and providing only one *indirect* addressing mode where load and store operations find their target addresses on the evaluation stack. Experimental results showed that a finite-state code generator produced from a specification that maps stack-based virtual machine IR to x86 code, outperforms the previously mentioned approaches with respect to code generation throughput and code generator size.

3.5 Summary

Automatic code generation had its genesis based on ideas taken from LR parsing (Graham and Glanville [11]). More sophisticated subsequent approaches used tree pattern matching and dynamic programming to produce optimal code generators (Horspool and Scheunemann [12], Fraser, Hanson and Proebsting [6], Proebsting [25] [24] [23]). For environments where code generator speed and size are of greater importance than code quality, a finite-state code generation approach was shown to yield promising results (Fraser [7]).

While the specification languages accepted by code generator generators are very similar from a semantic point of view, differing mostly in the choice of syntax, the strategies used for matching instructions and emitting code are quite different. Solutions utilizing dynamic programming guarantee to emit optimal instructions at the expense of producing fast but large, or small but slower code generators. The finite-state approach produces tiny and fast code generators from restricted tree grammars but does not guarantee to produce optimal instruction sequences due to greedy instruction matching.

We implemented our prototype based on ideas presented by Fraser [6], applying tree pattern matching combined with dynamic programming at compile time. The produced code generator is smaller, easier to debug, but slower than a table driven BURS approach.

4

The Problem of Effective Tree Pattern Matching Combined with Dynamic Programming

This chapter describes fundamental algorithms implemented in our code generator generator and introduces our code generator description language. First, limitations in other code generator description languages are identified followed by outlining our solutions to those limitations. Next, an optimal instruction selection algorithm based on tree pattern matching and dynamic programming is illustrated. Finally, our code generator description language is introduced together with several concise examples.

Advances in compiler construction have shown that many stages can be automated during the design and implementation process of a compiler. The most prominent areas being lexical analysis and parsing where a great deal of research resulted in the availability of tools which generate lexical analyzers and parsers automatically from concise grammar specifications.

We focus on the task of automatically generating code generators from a specification that maps an IR tree onto some target machine instruction set. Figure 4.1 outlines how a code generator is produced from such a specification. The emitted code generator is plugged into a compiler back end and

performs *optimal* instruction selection by covering an IR tree with tree patterns that denote target instructions. Each tree pattern is associated with a cost that contributes to the overall costs of generated code. Optimality is achieved by selecting patterns in such way that the overall costs become minimal.

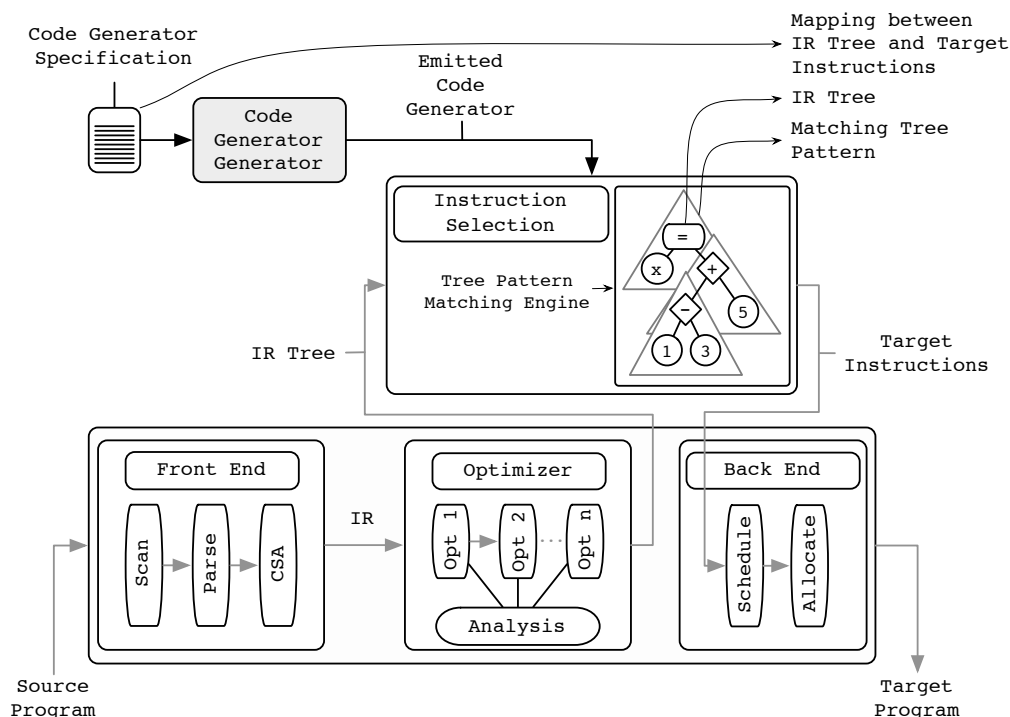


Figure 4.1: Automatic generation of a compiler back end.

Research efforts in the area of automating the construction of code generators is outlined in Chapter 3. Although many novel ideas and approaches have been devised during those efforts, the grammar specification languages and code generator generators are not yet as sophisticated as their lexical analyzers and parser generator counterparts.

4.1 Motivation

Code generator specification languages for tree pattern matching systems with dynamic programming can certainly be improved with respect to the following limitations:

1. **Limitation:** All of the languages we have encountered so far only provide means to emit semantic actions *after* a complete tree pattern has been processed. Given the rewrite rule “ASGN (*mem* *m*, *reg* *a*)” as illustrated in Listing 3.1, it is not possible to add a semantic action before or indeed between the “(*mem* *m*, *reg* *a*)” sub-pattern. This would be useful in situations where jump labels need to be computed, or a register allocator must be consulted before sub-patterns are processed.
2. **Limitation:** There is also no easy way to pass values to and from non terminals produced by rewrite rules (see chapter 3 for a description of basic tree pattern matching grammar components). Again, this would for example be useful to pass jump labels or register ranges to productions.
3. **Limitation:** Translating high level imperative languages to tree-structured intermediate representations gives rise to the question of how to model sequential statements efficiently in a tree. In his technical report about OP2, a portable Oberon compiler, Crelier [4] argues that it would be expensive to insert dummy nodes linking subtrees in order to represent sequential constructs. Therefore he suggests the usage of an additional *link* field in the node data structure as outlined in Figure 4.2. Again, we have not encountered a code generator generator which would support specifications of such intermediate representations on a syntactic and semantic level.

Our aim was to design a powerful tree pattern matching specification language with simplicity and easy readability in mind. We also wanted to address the previously mentioned limitations by allowing arbitrary placement of semantic actions within productions, and by supporting intermediate representations linking sequences of subtrees as illustrated in Figure 4.2. Thus solving the shortcomings mentioned in limitation 1 and 3.

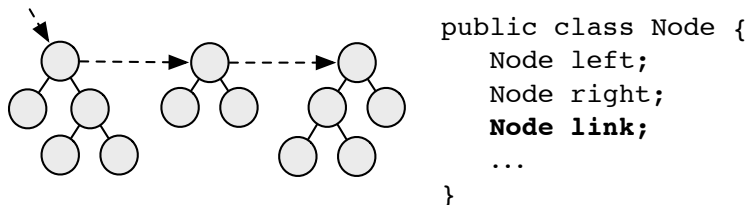


Figure 4.2: Tree-structured intermediate representation with *links* to subtrees.

We have designed our language to accept attribute grammar style specifications since they provide an elegant solution to the problem of passing

values to and from productions as mentioned in limitation 2. The syntax used for specifying formal attributes is based on Mössenböck’s [19] `Coco1/R`¹ parser generator specification language. Attribute grammars, first invented by Knuth [18], have been previously applied to the code generator generator domain. Ganapathi and Fischer [9] used the formalism to improve the Graham-Glanville code generator generator based on `LR` parsing (see 3.2). Our specification language and prototype implementation successfully demonstrates that the attribute grammar formalism can also be applied to code generator generators based on tree pattern matching and dynamic programming. Before presenting our code generator description language, we focus on fundamental algorithms for optimal instruction selection that are applied in our code generator.

4.2 Fundamental Algorithms

Our code generation approach is based on tree pattern matching with dynamic programming applied at compile time as outlined in Section 3.3.1. Essentially it resembles a two-pass system that optimally covers an IR tree with cost-augmented tree patterns in the first bottom-up pass, and executes semantic actions associated with the selected patterns in the second top-down pass. Optimally covering an IR with tree patterns corresponds to finding an optimal instruction sequence that implements it. The task of *efficiently* calculating an optimal cover given ambiguous machine grammar definitions is the topic of the next section.

4.2.1 Calculating Optimal Instruction Sequence

An example inspired by Cooper and Torczon [3] should help to understand the problem of efficient optimal instruction selection. Figure 4.3 illustrates an intermediate representation for the assignment statement $x := y - 2 * 3$. Subtrees that calculate variable locations are highlighted and point to the corresponding variable. A table listing a subset of rewrite rules and their costs for the given IR is depicted to the right-hand side of Figure 4.3. A rewrite rule consists of a non terminal derived by a tree pattern (non terminals denote *immediate*, *register*, and *memory* addressing modes in our example). Note that the specification of semantic actions has been omitted since they are not needed to illustrate the problem.

¹`Coco1/R` - Compiler compiler language generating recursive descent parsers.

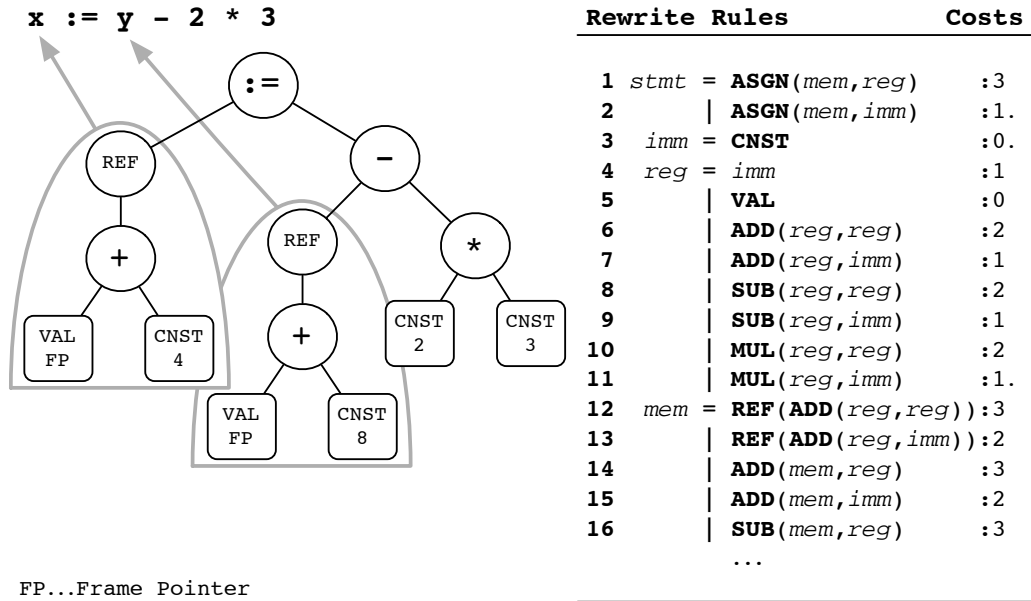


Figure 4.3: Tree-structured intermediate representation for an assignment instruction with corresponding rewrite rules.

The subtree $\text{REG}(\text{ADD}(\text{VAL FP}, \text{CNST 4}))$ that calculates the address of variable x in Figure 4.3 is the one we are concerned about. By looking at the rewrite rules we see that it is possible to cover the **VAL FP** node with rule number 5, namely $\text{reg} = \text{VAL}$. Then we can apply rewrite rule 3 to cover the node **CNST 4**. As a consequence of applying the previous rewrite rules, rule number 13, namely $\text{reg} = \text{REF}(\text{ADD}(\text{reg}, \text{imm}))$, matches the complete resulting subtree.

The diligent reader might have noticed that this is not the only sequence of rewrite rules that can be applied. Figure 4.4 shows that there exists another rewrite rule sequence that covers the subtree. The first sequence $\langle 5, 3, 13 \rangle$ has a cost of two, and the second sequence $\langle 5, 3, 4, 12 \rangle$ has a cost of four. Real world grammar specifications usually include many more potential rewrite rule sequences than this small example. Thus an efficient algorithm for finding optimal sequences of rewrite rules is needed.

Our solution to this problem is based on dynamic programming. Dynamic programming recursively subdivides an overall problem into a number of subproblems that can be solved individually. The solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation. In our case the overall problem of generating an optimal sequence of rewrite rules for a tree-structured intermediate

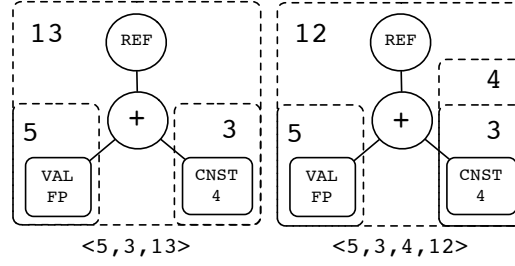


Figure 4.4: Possible rewrite rule matches covering an intermediate representation.

representation T , is subdivided into subproblems of finding optimal rewrite rules for the subtrees of T . A rewrite rule R is recorded at the root node of a subtree T if and only if no cheaper rule that derives the same non terminal exists.

Figure 4.5 demonstrates how an IR tree is optimally covered with tree patterns using dynamic programming. The labeling process works bottom-up. In the first step nodes **VAL** and **CNST** are labeled. Rewrite rule 5 matches node **VAL** and is recorded in a table that encodes all components of a rule, namely its number, its cost, and the non terminal it derives (i.e. the addressing mode). Pattern number 3 matches node **CNST** and is recorded as well. Note that since **CNST** derives an *imm* non terminal now, rule number 4 also matches. So a matching pattern can trigger a chain of subsequent matches.

In the second step node **ADD** is labeled. The first matching rewrite rule for **ADD** is rule 6. It demands that the left and right child nodes of **ADD** derive *reg* non terminals. The costs of the required non terminals derived by the left and right subtrees and the cost of rule 6, contribute to the *overall* costs resulting from matching rule 6 for **ADD**. Thus rule 6 together with its overall costs is recorded for **ADD**. But in the third step it turns out that another rule, namely rule 7 also matches **ADD**. Rule 7 now overwrites rule 6 in the table for **ADD** since its overall costs are lower than those for rule 6. In other words matching rule 7 is *cheaper* than matching rule 6. Since no further rules match **ADD**, we can proceed to node **REF**. The procedure for selecting an optimal pattern for **REF** is not outlined any further because it is the same as for steps two and three.

The procedure of finding a minimum-cost IR tree cover was given informally. Section 5.1.3 presents a formal algorithm for this procedure as it is implemented in our code generator generator.

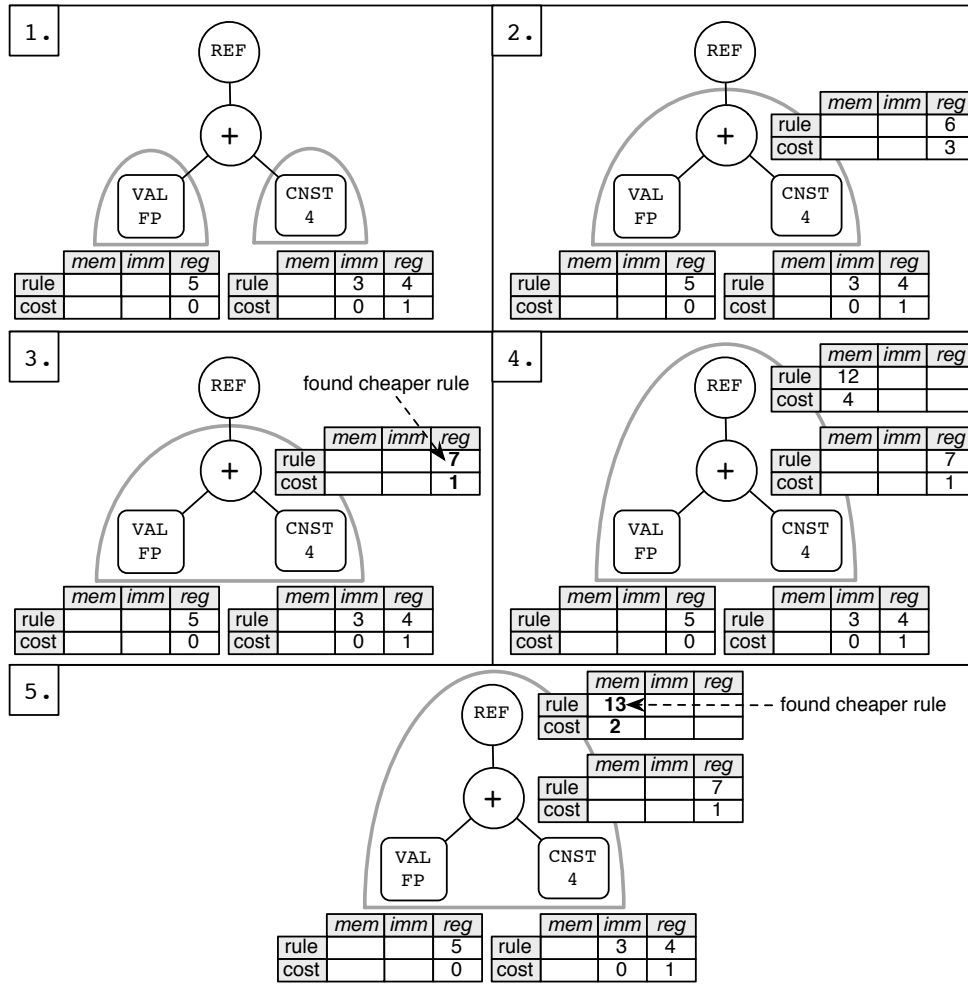


Figure 4.5: Bottom-up optimal instruction selection using dynamic programming.

4.2.2 Emitting Optimal Instruction Sequence

Once the first bottom-up pass of the intermediate representation is complete, yielding an optimal rule rewrite sequence as its result, semantic actions associated with rewrite rules can be emitted. Because the tree is traversed top-down during this phase, semantic actions denoted by s_n , may be included

anywhere within rewrite rules as the following example demonstrates:

$$goal = \underline{s_1} \text{ assign } \underline{s_2} \quad (4.1)$$

$$assign = \underline{s_1} \text{ \textbf{ASGN} } \underline{s_2} (\underline{s_3} \text{ reg } \underline{s_4}, \underline{s_5} \text{ reg } \underline{s_6}) \quad (4.2)$$

$$reg = \underline{s_1} \text{ \textbf{REF} } \underline{s_2} (\underline{s_3} \text{ \textbf{ADD} } \underline{s_4} (\underline{s_5} \text{ reg } \underline{s_6}, \underline{s_7} \text{ reg } \underline{s_8}) \underline{s_9}) \underline{s_{10}} \quad (4.3)$$

The semantics of the previous example with respect to emitting semantic actions, resemble those of the recursive descent parser devised by Mössenböck [19]. As soon as a subtree that matches a pattern given in the previous example is reached, the processing proceeds as follows depending on the rewrite rule type:

1. *Base Rule:* Rules 4.2 and 4.3 are base rules. Base rules consist of operators (terminals) and operands (non terminals). First the semantic actions just before and after the operator are emitted. In our case that would be $\underline{s_1}$ for the **REF** operator. Parenthesis indicate that an operator has child nodes (operands), as is the case for **REF**, **ASGN**, and **ADD**. Prior to recursing down to process a child node terminal or non terminal, semantic actions defined before the respective terminal or non terminal are emitted (i.e. before processing **ADD**s second child node *reg*, $\underline{s_7}$ is emitted). After returning from the recursive call that was responsible for processing **ADD**s second child node *reg*, the following semantic action $\underline{s_8}$ is emitted.
2. *Chain Rule:* Rule 4.1 represents a chain rule. A chain rule is a rule whose pattern is another non terminal (operand). Again the semantic action $\underline{s_1}$ in rule 4.1 is emitted before processing the non terminal *assign*, followed by the emission of the subsequent semantic action $\underline{s_2}$.

4.3 Code Generation Description Language

The following sections describe the language we have devised in order to specify mappings from tree-structured intermediate representations to target machine code. Many syntactic constructs were inspired by Mössenböck's [19] parser generator description language **Cocol/R**. First we outline the overall structure of our language, followed by definitions of core language constructs together with concise examples. Complete examples as well as a precise language specification in **EBNF**² [28] are available in the appendix.

²**EBNF** - Extended Backus Naur Form

4.3.1 General Structure

Before introducing our code generator description language we have to define its basic elements:

- **Terminals**, also referred to as **operators**, are upper case.
- **Non terminals**, also referred to as **operands**, are lower case.
- **Tree patterns** define possible substitutions of a **non terminal** symbol and are separated by a “|” pipe character.
- **Semantic actions** are placed between (. and .), denoting a piece of code written in the target language of the code generator.
- **Costs** are specified after a colon at the end of each tree pattern. Costs can be either constants, or arbitrary expressions written in the target language.
- The combination of a tree pattern, its cost, and the non terminal it derives is called a **rewrite rule**.

```

1  generator          -- include, import statements
2      ( . ... . )
3  declarations      -- global variables and functions
4      ( . ... . )
5  operators          -- also referred to as node kinds
6      NUM(....), ASGN(....), ADD(....), SUB(....)
7  rules              -- rewrite rules
8      stmts = stmt [ stmts ] : 0. -- start rule
9      stmt = ASGN (reg, NUM) (....) : 1
10         | ASGN (reg, reg) (....) : 2.
11      reg = NUM      (....) : 1
12         | ADD (reg, reg) (....) : 2
13         | SUB (reg, reg) (....) : 2.
14  end

```

Listing 4.1: General code generator description language structure.

A code generator specification as outlined in Listing 4.1, has the following structure:

1. **generator:** The **generator** section is used to import packages (in Java), namespaces (in C#), modules (in Haskell), or header files (in C), enclosed within (. and .).
-

2. **declarations:** Arbitrary fields, variables, functions, and methods can be declared within (. and .) in the **declarations** section. All declarations are within the scope of semantic actions specified in the **rules** section.
3. **operators:** Operators are capitalized and designate the kinds of valid nodes that make up a tree-structured intermediate representation. All operators specified within this section comprise the set of valid terminals used within tree patterns of rewrite rules.

Operator identifiers must not always correspond with node kind identifiers used in the target compiler. In such a case it is possible to specify a mapping from operator identifiers used in the code generator specification to identifiers used by the target compiler (i.e. `ADD(.109.)` maps `ADD` to its numeric representation 109).

4. **rules:** The **rules** section contains rewrite rules capturing the structure of possible tree patterns. A detailed description of rewrite rules is given in the following section.

4.3.2 Rewrite Rules

A rewrite rule specifies one or more tree patterns of an intermediate representation. It consists of a left-hand side and a right-hand side which are separated by an equal sign. The right-hand side contains tree patterns in fully parenthesized prefix form, and the left-hand side specifies a non terminal that represents the subtree matched by the right-hand side. Non terminals can in turn be used as operands in tree patterns, and usually represent storage classes or addressing modes provided by the target architecture.

The *first* rewrite rule defined in the **rules** section is the start rule. It must match the root node of a tree-structured intermediate representation. All subsequent rewrite rules can be specified in any order. Each rewrite rule is defined in terms of at least one tree pattern. Tree patterns may be given in any order, and every tree pattern must be augmented with its cost.

Tree patterns can also match *linked* chains of subtrees as Line 8 in Listing 4.1 demonstrates. The example shows how a chain of **stmts** subtrees can be matched by the rule `stmts = stmt [stmts]`, where [and] denote that the non terminal **stmts** is optional. Section 4.1 and Figure 4.2 introduce the concept of linked subtrees in tree-structured intermediate representations.

4.3.3 Tree Pattern Costs

Costs are evaluated during the first bottom-up pass of the intermediate representation and every tree pattern must be associated with its cost.

Costs need not always be constant. It is possible to define costs in terms of arbitrary expressions written in the target language wrapped with `(.` and `.)`. Such expressions *must* evaluate to integer numbers. Furthermore only variables and functions defined in the **declarations** and **generator** section, as well as the root node of the corresponding tree pattern, available via an *implicitly* defined variable, are within the scope of such expressions.

4.3.4 Semantic Actions

A semantic action is specified between `(.` and `.)`. It comprises a piece of code written in the target language of the code generator. Semantic actions are executed by the generated code generator at the position where they have been specified during the second top-down pass of the intermediate representation.

Nodes matched by tree patterns are accessible through **bindings** in semantic actions. A binding can be defined for each terminal and non terminal in a tree pattern, and its scope ranges from its definition until the end of the tree pattern. Code within each semantic action can also access variables and methods defined in the **declarations** section of the specification. Listing 4.2 shows an example code generator specification demonstrating how to build a list of machine code instructions for linked expression subtrees. References to bindings and bindings themselves are underlined in order to highlight them.

```

public class Node {
    Kind    kind,
    String val,
    String result,
    Node    left;
    Node    right;
}

public enum Kind {
    E_CONST,
    E_ADD,
    E_SUB,
    E_MULT
}

```

Figure 4.6: Data Structure for intermediate representation nodes.

The language used within semantic actions is **Java**. Matching operators and non terminals are of type **Node** and the corresponding data structure is depicted in Figure 4.6. Lines 15 and 20 show how the global variable `code` of

```

1 generator      -- import statements
2   (. import java.util.LinkedList;
3     import java.util.List;
4     import code.codegen.Code; .)
5 declarations   -- general declarations
6   (. List<String> code = new LinkedList(); .)
7 operators
8   NUM(.E_CNST.), ASGN(.E_ASGN.), ADD(.E_ADD.),
9   SUB(.E_SUB.), MUL(.E_MUL.)
10 rules         -- rewrite rules
11   stmts = stmt [stmts] : 0. -- start rule
12   stmt = ASGN (reg,NUM) (. ... .) : 1
13         | ASGN (reg,reg) (. ... .) : 2.
14   reg = NUM a      (.a.result = Code.getReg();
15                     code.add("loadI "+
16                             a.val +", "+
17                             a.result);.) : 1
18         | ADD a (reg b, reg c)
19             (.a.result = Code.getReg();
20             code.add("add "+
21                     b.result +", "+
22                     c.result +", "+
23                     a.result);
24             Code.freeReg(b.result);
25             Code.freeReg(c.result);.): 2
26         | SUB a (reg c, reg c) (. ... .) : 2
27         | MUL a (reg b, reg c) (. ... .) : 4.
28 end

```

Listing 4.2: Referring to bindings and declarations in semantic actions.

type `List<String>`, defined in the `declarations` section, can be referenced from semantic actions (i.e. instructions are inserted into `code`).

4.3.5 Attributes

Essentially the rewrite rule `reg = ADD(reg,reg)` produces a *reg* non terminal by matching binary subtrees having an `ADD` root node that expects its child nodes to produce *reg* non terminals. It is still unclear though how non terminals can *produce* values and how to *access* them in semantic actions. Listing 4.2 demonstrates a solution to this problem by storing the result (i.e. the name of the register containing the result value) produced by the `ADD` node in the `result` field of the `Node` data structure depicted in Figure 4.6.

```

1 generator          -- import statements
2   (. import java.util.LinkedList;
3     import java.util.List;
4     import code.codegen.Code;
5     import code.codegen.Reg; .)
6 declarations      -- general declarations
7   (. List<String> code = new LinkedList(); .)
8 operators
9   NUM(.E_CNST.), ASGN(.E_ASGN.), ADD(.E_ADD.),
10  SUB(.E_SUB.), MUL(.E_MUL.)
11 rules             -- rewrite rules
12  stmts = stmt [stmts]          : 0. -- start rule
13  stmt = ASGN (reg<.out Reg b.>, NUM) (. ... .) : 1
14        | ASGN (reg<.out Reg b.>, reg) (. ... .) : 2.
15  reg<.out Reg r.>
16    (. r = Code.getReg(); .) -- result register
17    = NUM a
18    (.code.add("loadI "+ a.val +", "+ r); .) : 1
19    | ADD (reg<.out Reg b.>, reg<.out Reg c.>)
20    (.code.add("add "+ r +", "+ b +", "+ c);
21      Code.freeReg(b); Code.freeReg(c); .) : 2
22    -- SUB and MUL patterns omitted
23    .
24 end

```

Listing 4.3: Defining attributes and referring to them in semantic actions.

While the previous ad-hoc approach works, it is rather unsatisfactory because in order to know the type and value that is produced by a rewrite rule, one has to look into the implementation supplied within semantic actions. Fortunately the well established attribute grammar formalism [18] provides just the right solution for our problem. We simply annotate non terminals with attributes denoting the values they produce. By doing so, it is possible to explicitly encode information about resulting values of non terminals into a grammar specification. Rewriting our previous example to include *output* attributes yields the following:

- $reg<.out\ Reg\ \underline{r0}.> = ADD\ (reg<.out\ Reg\ \underline{r1}.>, reg<.out\ Reg\ \underline{r2}.>)$

We distinguish between *formal* attributes, defined at the non terminals declaration on the left hand side of a tree pattern, and *actual* attributes, specified at the non terminals occurrence within tree patterns. While the scope of a formal attribute ranges over all tree patterns that define a non terminal, the scope of an actual attribute ranges from its definition until the

end of a tree pattern. So the previous rewrite rule defines a non terminal *reg* that *produces* a value of type **Reg** stored into the variable **r0**, by specifying the formal attribute `<.out Reg r0.>`. So far we only mentioned the possibility to define *output* attributes for non terminals, but it is also possible to define *input* attributes (see Appendix B for examples of input attributes).

The example in Listing 4.3 performs the same task as Listing 4.2 in the previous section, but does so by using output attributes. It also demonstrates how to define output attributes, and how to access them from within semantic actions.

4.4 Summary

In this chapter we have dealt with the problem of effective tree pattern matching using dynamic programming. We started by introducing fundamental algorithms (see Section 4.2) used to calculate optimal instruction sequences given a tree-structured intermediate representation, followed by a description of how semantic actions are emitted for matching rewrite rules.

Before introducing our code generator description language we have outlined several limitations discovered in other languages based on our approach. Support for attribute grammars (see Section 4.3.5), linked sequences of tree-structured intermediate representations (see Section 4.1), and the ability to place semantic actions almost anywhere within rewrite rules are our solutions to the outlined limitations. Appendix A gives a precise syntax specification for our code generator description language supplementing our descriptions in Section 4.3.

5

HBURG - Haskell Bottom Up Rewrite Generator

*This chapter starts with an overview of **HBURG**'s architecture and module structure. Then, its intermediate representation is outlined followed by a description of the context-sensitive analysis that is performed in order to statically verify the correctness of grammar specifications. Next, **HBURG** generated code for optimal instruction selection and code generation is described. Finally, the integration of **HBURG** generated code into a compiler is outlined.*

HBURG is a code generator generator, which takes a cost-augmented tree grammar and generates a code generator for this tree grammar. A tree grammar defines a mapping from a tree-structured intermediate representation onto target machine instructions. The resulting code generator can be plugged into the instruction selection phase of a compiler. **HBURG** produces code generators written in **Java**, but its back end can be easily extended to support other languages.

The acronym **HBURG** stands for Haskell Bottom Up Rewrite Generator and denotes the fact that **HBURG** itself is implemented in Haskell, and produces code generators that perform optimal instruction selection based on bottom up rewrite system theory. The following sections outline **HBURG**'s architecture, implementation, and the most important data structures. The integration of

HBURG generated code generators into a compiler infrastructure is covered as well.

5.1 Architecture

HBURG has a rather conventional compiler architecture consisting of a *front end* and a *back end*. Its overall structure is depicted in Figure 5.1. The front end builds a target-language-independent intermediate representation given a code generator specification. The back end produces the final code generator. The following paragraphs outline the main compiler phases implemented in the front and back end:

- *Front End*: Lexical and syntactical analysis are the first two phases in the front end. Both, the lexical analyzer as well as the parser are generated via tools, namely **Alex**¹ and **Happy**². Context-sensitive analysis is performed during and at the end of parsing.
- *Back End*: The back end is separated into two phases. The **Tile** phase emits code for optimal instruction selection, and the **Evaluate** phase generates code that emits semantic actions for matching tree patterns.

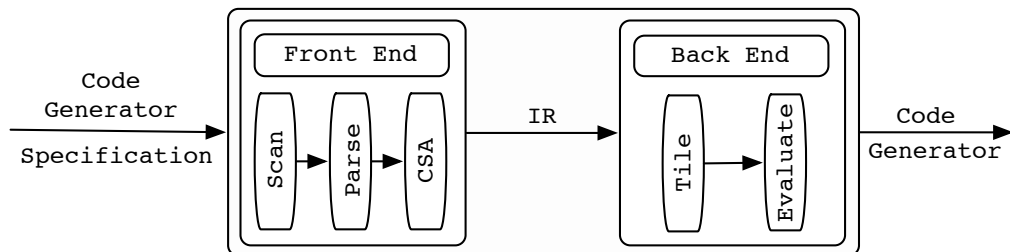


Figure 5.1: HBURG compiler infrastructure.

In order to get a better understanding of HBURG’s implementation, it helps to take a look at its organization at the *module* level. Haskell programs consist of a collection of modules. Modules serve the dual purpose of controlling name-spaces and creating abstract data types [13]. HBURG makes heavy use of a syntactic extension of Haskell’s module system referred to as the “hierarchical module namespace”. This extension enhances the *flat* Haskell module

¹Alex - A lexical analyzer generator for Haskell

²Happy - A dyslexic acronym for “A Yacc-like Haskell Parser generator”

name-space as defined in *The Haskell 98 Report* [15], into a hierarchy of modules, and is understood by most³ Haskell compilers. Figure 5.2 gives an overview of **HBURG**'s most important modules and abstract data types as well as their use. It also indicates which modules belong to the front end and back end of the compiler. The **Main** module contains the **main** function which is evaluated when **HBURG** is executed, and thus serves as the entry point.

	Module Name	Description
	Main	Main module
Front End	Parser.Lexer	Modules for scanning and parsing
	.Parser	
	...	
	Csa.Csa	Modules containing abstract data types and functions for context sensitive analysis
	.Ctx	
	.Elem	
Back End	Ast.Node	Abstract data types used for constructing tree structured intermediate representation
	.Def	
	.Prod	
	...	Abstract data types and modules for code generation
	Gen.Backend	
	.Emit	
	.Emit.Tile	
	.Emit.Eval	
	...	

Figure 5.2: **HBURG** modules overview.

The initial version of **HBURG** provides a back end that generates code generators written in the **Java** programming language. In order to add support for a new target language, only the back end must be extended while the front end can remain unchanged.

HBURG's intermediate representation is used to introduce its most important data structures in the following sections. Next, context sensitive analysis consisting of scope and type analysis is outlined. Finally, the code generation phases of the compiler, denoted as **Tile** and **Evaluate** in Figure 5.1, are introduced in greater detail.

5.1.1 Intermediate Representation

A code generator specification is mapped onto an intermediate representation from which target code is generated. The intermediate representation

³GHC, NHC, and Hugs support the hierarchical module namespace extension to Haskell 98

resembles a list of rewrite rules. Rewrite rules define non terminals in terms of tree patterns. Thus a rewrite rule consists of a *forest* of tree patterns that are made up of terminals, non terminals, bindings, semantic actions, and costs. Every part of a rewrite rule is represented by an abstract data type. Figure 5.3 depicts abstract data types that represent a rewrite rule.

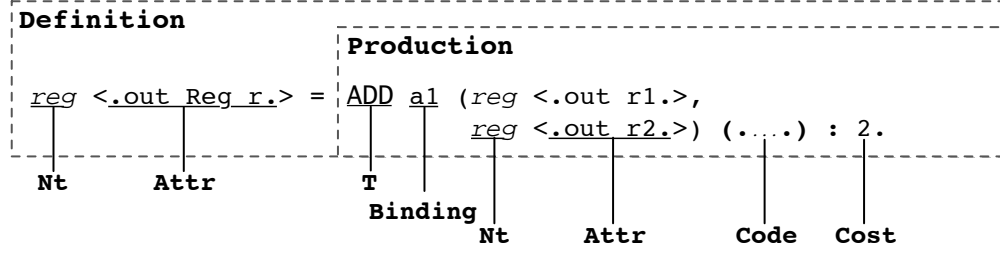


Figure 5.3: Mapping a rewrite rule onto abstract data types defined in HBURG.

The corresponding Haskell data type definitions are specified in Listing 5.1. The first line declares a terminal `T` to be a new *data type*, with a single data constructor `T`. The constructor has two *fields*: an `Ident` giving the name of the terminal, and a `Binding` representing an identifier that can be used to refer to a terminal within a semantic action. The non terminal data type `Nt` has the same fields as `T`, with the addition of a list of attributes denoted by `[Attr]`.

```

1  data T  = T Ident Binding
2  data Nt = Nt Ident Binding [Attr]
3  data Term = Terminal T
4             | NonTerminal Nt
5  data Node = Nil
6             | N { term      :: Term
7                   , child   :: Node
8                   , sibling  :: Node
9                   , link    :: Node
10                  , code    :: Map Position Code }
11  data Production = Prod { pattern :: Node
12                           , cost   :: Cost }
13  data Definition = Def  { nt      :: Nt
14                           , prods  :: [Production] }

```

Listing 5.1: HBURG's intermediate representation defined in terms of algebraic Haskell data types.

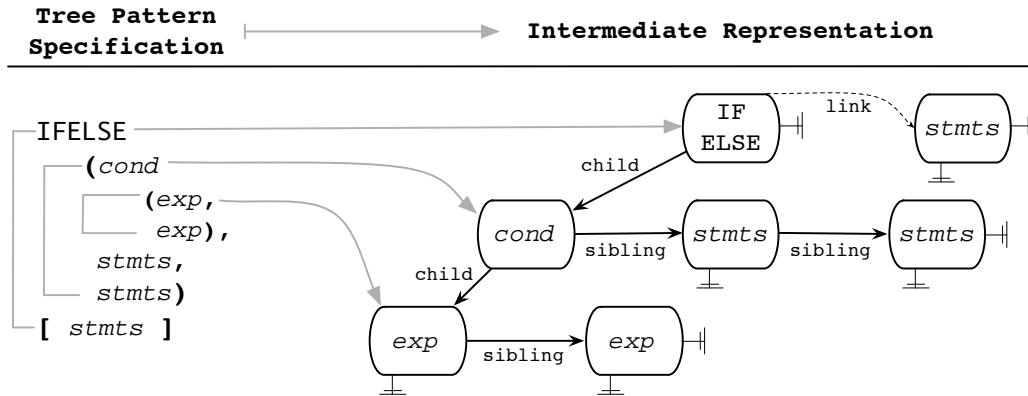


Figure 5.4: Mapping a tree pattern onto **HBURG**'s intermediate representation.

In general, a Haskell data type comprises one or more constructors (i.e. the data types `Term` and `Node` both have two constructors), and each constructor can have zero or more fields (i.e. the constructor `Nil` has no fields). Furthermore the definitions of the `Node`, `Production`, and `Definition` data types make use of Haskell's *record syntax*.

A tree data structure is used to represent tree patterns. Trees are defined in terms of a `Node` data type (see line 5 in Listing 5.1). This type is able to capture tree patterns that contain subtrees with varying *degrees* as the example outlined in Figure 5.4 demonstrates. The remaining data types in Listing 5.1 have the following semantics:

- **Production:** The `Production` type resembles a tree pattern with its associated cost.
- **Definition:** A rewrite rule consists of a non terminal `Nt` defined in terms of a list of `Productions` and has the type `Definition`.
- **Term:** Often it is necessary to deal with terminals and non terminals in a uniform way. This is possible by wrapping them into a `Term` data type.

The previously introduced data types denote essential parts of **HBURG**'s intermediate representation. One question we have yet to answer is where and how semantic actions are stored. Our code generator language allows definitions of semantic actions at various positions within tree patterns (see Section 4.2.2). Thus each node has to store semantic actions defined at its tree *level*. A node stores this information in its `code` field (see line 10 in

Listing 5.1). The data type of this field denotes a Map^4 of *keys* to *values*, where keys denote the relative position of a semantic action within a tree pattern, and values denote the corresponding semantic actions.

5.1.2 Context Sensitive Analysis

Our code generator language should be a *safe language*, thus **HBURG** performs a range of checks in order to statically verify the *absence* of certain kinds of runtime errors. Pierce [22] gives a good intuition of language safety by stating that “a safe language is one that protects its own abstractions”. In the context of our code generator language such *abstractions* resemble the definition of rewrite rules in terms of terminals, non terminals, attributes, bindings, semantic actions, and costs. So **HBURG** checks that all abstractions specified in our language are well-defined, and used in the correct context by performing *scope* and *type* analysis:

- *Scope Analysis*
 - Within the scope of a tree pattern a binding identifier for a terminal or non terminal may only be specified once. Thus the tree pattern “ADD (*reg* *r1*, *reg* *r1*) : 0” results in a *duplicate binding* error since the identifier *r1* is used twice within its scope.
 - Each terminal must be defined in the **operators** section before it can be used in the **rules** section.
 - Non terminals used within tree patterns must be defined at some point within the **rules** section, and the amount and order of *in* and *out* attributes must conform to their definition.

- *Type Analysis*

A **Definition** defines a non terminal in terms of **Productions**. Productions are defined in terms of terminals and non terminals, and can be categorized into *chain rules* and *base rules*. Chain rules derive one non terminal from another, and base rules define a concrete tree pattern that derives a non terminal. For each concrete tree pattern captured by a base rule we check that all of its child nodes have the correct *types* with respect to the non terminals they derive. The following example demonstrates a flawed specification that fails with a *type* error at the

⁴**Map** - Haskell library implementation of maps from keys to values.

underlined non terminal:

1	$reg = \text{CONST}$: 1
2	$\text{ADD } (reg, reg)$: 2.
3	$val = \text{OP } (\text{ADD } (reg, \underline{val}), reg)$: 3.

The production “ $\text{ADD } (reg, reg)$ ” at line 2, defines that tree patterns having **ADD** as their root node, expect their left and right children to produce nodes of type *reg*. Furthermore, it defines that “ $\text{ADD } (reg, reg)$ ” derives the type *reg*. Now let’s try to type check the definition on line 3. Obviously the right child node of **OP** is of type *reg*. But what type does its left child node “ $\text{ADD } (reg, \underline{val})$ ” derive? There is no rewrite rule that would allow us to infer the type of “ $\text{ADD } (reg, \underline{val})$ ”.

Fortunately, the problem can be fixed easily by extending the definition of *reg* with a *chain rule* that derives a *reg* from a *val* as defined at line 1:

1	$reg = val$: 1
2	CONST	: 1
3	$\text{ADD } (reg, reg)$: 2.
4	$val = \text{OP } (\text{ADD } (reg, val), reg)$: 3.

With the additional chain rule **HBURG** is able to infer that “ $\text{ADD } (reg, val)$ ” can be transformed into “ $\text{ADD } (reg, reg)$ ”, and for this pattern the type it derives is known.

Semantic actions are not analyzed by **HBURG**, they are simply copied to the generated code generator without being checked. Thus the target language compiler has the task to detect syntactic and semantic errors in semantic actions.

5.1.3 Optimal Instruction Sequence Calculation

Given a set of cost-augmented rewrite rules, a tree-structured intermediate representation must be optimally *tilled* with matching rules. Thus for each node we must record a set of lowest-cost matching tree patterns together with the non terminals they produce.

The procedure *Tile* outlines an algorithm that finds an *optimal* tiling for a tree rooted at node *n*. In order to simplify the algorithm, two assumptions about the form of rewrite rules have been made. First, each operation has,

Procedure Tile(Node n)

```

if n is a binary node then
  Tile(n.left)
  Tile(n.right)
  foreach pattern p that matches n's operation do
    if  $\text{Nt}(\mathbf{p}.\text{left}) \in \text{NtSet}(\mathbf{n}.\text{left}) \wedge$ 
       $\text{Nt}(\mathbf{p}.\text{right}) \in \text{NtSet}(\mathbf{n}.\text{right})$  then
       $\mathbf{c} \leftarrow \text{Cost}(\mathbf{n}.\text{left}, \text{Nt}(\mathbf{p}.\text{left})) +$ 
         $\text{Cost}(\mathbf{n}.\text{right}, \text{Nt}(\mathbf{p}.\text{right})) + \mathbf{p}.\text{cost}$ 
      if  $\text{Nt}(\mathbf{p}) \notin \text{NtSet}(\mathbf{n}) \vee \mathbf{c} < \text{Cost}(\mathbf{n}, \text{Nt}(\mathbf{p}))$  then
        record(n, Nt(p), c)
        if Nt(p) triggers chain rules then
          Closure(n, Nt(p), c)
    end
  else if n is unary node then
    Tile(n.left)
16   foreach pattern p that matches n's operation do
17     if  $\text{Nt}(\mathbf{p}.\text{left}) \in \text{NtSet}(\mathbf{n}.\text{left})$  then
18        $\mathbf{c} \leftarrow \text{Cost}(\mathbf{n}.\text{left}, \text{Nt}(\mathbf{p}.\text{left})) + \mathbf{p}.\text{cost}$ 
19       if  $\text{Nt}(\mathbf{p}) \notin \text{NtSet}(\mathbf{n}) \vee \mathbf{c} < \text{Cost}(\mathbf{n}, \text{Nt}(\mathbf{p}))$  then
20         record(n, Nt(p), c)
21         if Nt(p) triggers chain rules then
22           Closure(n, Nt(p), c)
    end
  else
    /* n is a leaf node */
    foreach pattern p that matches n's operation do
      record(n, Nt(p), p.cost)
    end

```

at most, two operands. Second, a rule's right-hand side contains at most one operation. The algorithm can be extended to handle the general case, but that would only complicate its explanation. Cooper and Torczon [3] give a similar algorithm that finds *all* matches in a pattern set for each node. We have extended their algorithm so that it finds the lowest-cost match given a pattern set.

The problem of finding an optimal tiling for a tree is subdivided into subproblems of finding optimal tilings for its subtrees. A postorder traversal ensures that the solutions to subproblems are constructed incrementally from those of smaller subproblems, and by recording each solution of a subproblem we avoid recomputation.

Procedure Closure(Node **n**, NonTerminal **nt**, Cost **c**)

```

foreach non terminal x derived by nt do
  if  $\mathbf{x} \notin \text{NtSet}(\mathbf{n}) \vee \mathbf{x}.\text{cost} + \mathbf{c} < \text{Cost}(\mathbf{n}, \mathbf{x})$  then
    record(n, x, x.cost + c)
    if x triggers chain rules then
      Closure(n, x, x.cost + c)
end

```

Consider the **foreach** loop at line 16 in our *Tile* procedure. It examines each pattern p that implements the operation specified by n . The function $\text{NtSet}(\text{Node } n)$ returns the set of non terminals node n can derive, similarly, the function $\text{Nt}(\text{Pattern } p)$ returns the non terminal derived by tree pattern p . If the condition at line 17 holds, then *Tile* has already discovered that n 's left subtree generates a non terminal that is expected by pattern p . Next, n 's cost c is calculated at line 18, by adding the cost of its left subtree to the cost of the selected tree pattern p . The function $\text{Cost}(\text{Node } n, \text{NonTerminal } nt)$ returns the overall cost of deriving nt from node n . The matching pattern together with its cost c is recorded for the current node n (see line 20), if the node does not derive the non terminal demanded by $\text{Nt}(p)$, or if c is lower than the cost for deriving a non terminal $\text{Nt}(p)$ at the current node n . Finally, if the non terminal derived by pattern p triggers chain rules, the procedure *Closure* records all triggered chain rules for node n .

HBURG emits an implementation of the *Tile* algorithm that can process unrestricted tree patterns with arbitrary numbers of subtrees. While the description of the *Tile* algorithm is language independent, its implementation depends on the target language of the resulting code generator. Furthermore, it is important to efficiently store and retrieve information necessary to calculate an optimal rewrite rule cover.

```

public class Node {
    Kind    kind,
    Node    left;
    Node    right;
    EnumMap<Nt, Entry> map;
    ...
}

public class Entry {
    int     cost,
    Pattern pat;
}

```

Figure 5.7: Node data structure for optimal instruction selection.

```

1 stmt  $s_1 = s_2$  ASGN1  $s_3$  (  $s_4$  disp  $s_5$  ,  $s_6$  reg<.out r.>  $s_7$  )  $s_8$       ( L0 )
2   |  $s_9$  reg<.out r.>  $s_{10}$  .      ( L1 )
3   reg<.out Reg r.>  $s_{11}$ 
4   = ADDI (  $s_{12}$  reg<.out r1.> ,  $s_{13}$  reg<.out r2.> )  $s_{14}$       ( L2 )
5   | IOI rc  $s_{15}$       ( L3 )
6   | CNSTI  $s_{16}$       ( L4 )
7   | disp  $s_{17}$  .      ( L5 )
8   disp = ADDI ( reg<.out r.> , CNSTI )  $s_{18}$       ( L6 )
9   | ADDRLP  $s_{19}$  .      ( L7 )

```

Listing 5.2: Examples of rewrite rules without cost annotations and with labels in parentheses. Semantic actions are denoted by s_n .

A **Java** datastructure that stores such information for each **Node** in a map of type **EnumMap** is outlined in Figure 5.7. An **EnumMap** is a specialized map implementation for use with enumeration type keys. It maps non terminals encoded by the enumeration **Nt** to entries of type **Entry**, where an **Entry** consists of a **Pattern** that was chosen to derive the **Nt**, and its cost. By using an **EnumMap**, the implementation of *Tile* is both, straightforward and efficient with respect to access time and memory size.

5.1.4 Optimal Instruction Sequence Code Generation

Once an optimal rewrite rule cover of an intermediate representation has been found, tree patterns that have been recorded for each node can be processed. Processing a tree pattern corresponds to traversing a subtree of the intermediate representation, and emitting semantic actions at the positions where they have been defined. This occurs during the second top-down traversal of the intermediate representation that is designated as the **Evaluate** phase in Figure 5.1.

Listing 5.2 shows a partial set of rewrite rules for an example intermediate representation where semantic actions are denoted by s_n . Costs have been omitted since they are not needed during the second top-down pass over the intermediate representation. Rule labels are enclosed in parentheses at the end of tree patterns. They are only used internally to refer to each pattern and must not be specified explicitly.

Listing 5.3 shows **Java** code that is emitted for the first two rewrite rules defined in Listing 5.2. Each derivation of a non terminal in terms of tree patterns corresponds to a method having the current **Node** as a parameter. Thus each rewrite rule is considered as a tree pattern parsing method. The

```

1      void stmt(Node n) {
2          s1
3          switch (label(n, "stmt")) {
4              case L0: {
5                  s2 s3
6                  s4 disp(left(n)); s5
7                  s6 Reg r = reg(right(n)); s7
8                  s8 break;
9              }
10             case L1: { s9 reg(n); s10 break; }
11         }
12     }
13     Reg reg(Node n) {
14         Reg r; s11
15         switch (label(n, "reg")) {
16             case L2: {
17                 s12 Reg r1 = reg(left(n));
18                 s13 Reg r2 = reg(right(n));
19                 s14 break;
20             }
21             case L3: { rc = n; s15 break; } /* binding */
22             case L4: { s16 break; }
23             case L5: { disp(n); s17 break; }
24         }
25         return r; /* return output attribute */
26     }

```

Listing 5.3: Code generator implementation of the first two rewrite rules specified in Listing 5.2. Semantic actions are denoted by s_n , output attribute types are bold face, and method calls corresponding to non terminals are italicized.

occurrence of a non terminal within a tree pattern can be viewed as a call of the non terminal's parsing method. Semantic actions, denoted by s_n in Listing 5.3, are copied in verbatim to their respective position in the resulting code generator.

Non terminals can have attributes as is the case for *reg* at line 3 in Listing 5.2. Such attributes correspond to parameters of the non terminal's tree pattern parsing method. For our Java back end this means that *output* attributes are translated to method definitions with the corresponding return type and return statement, whereas *input* attributes are translated to formal parameters of methods. Line 15 in Listing 5.2 demonstrates how *reg*'s output

attribute is translated into a **Java** method definition.

Node bindings as defined on line 5 in Listing 5.2 are translated into corresponding assignments in the target language as line 22 in Listing 5.3 demonstrates. The node matching the operator **IOI** is assigned to the variable **rc** which in turn can be accessed in the subsequent semantic action s_{15} .

The previously mentioned rule labels are encoded as enumerations in **Java**, and for each node the label denoting the non terminal that the rule derives is inspected in a **switch** statement as line 3 in Listing 5.3 demonstrates.

The top-down pass over an intermediate representation performed by the resulting code generator is very similar to the way a top-down **LL(k)** recursive descent parser works. We have exploited this resemblance by defining a language that incorporates two powerful features available in recursive descent parsers, namely (1) the possibility to define semantic actions almost anywhere within tree patterns, not just at the end, and (2) the possibility to define attributes for non terminals in our code generation language.

5.2 Invoking HBURG

The invocation of **HBURG** has the following syntax:

```
$ hburg [options] filename [options]
```

All command line options are optional and may occur either before or after the input filename. Options that take multiple arguments may be given multiple times, but the last occurrence will be the value used. The flags accepted by **HBURG** are as follows:

-c *Class*, **--classname=***Class*

Specifies the class name of the generated code generator. If omitted, **HBURG** assigns the name **Codegen**.

-p *Package*, **--package=***Package*

Instructs **HBURG** to place the resulting code generator into the specified **Java** package (e.g. **comp.gen**). If omitted, all files are placed into the current working directory.

-t *Type*, **--type=***Type*

This option specifies the data type that discriminates intermediate representation nodes. If omitted, this option defaults to **NodeKind**.

-d, **--debug**

Directs **HBURG** to output the result of parsing a code generator description file to standard output.

Given a code generator specification file **gen.tpg** that resides in the current directory, a type for discriminating intermediate nodes of **Kind**, and the package **comp.gen** that should include the resulting code generator, one invokes **HBURG** as follows:

```
$ hburg -t comp.ir.Kind -p comp.gen gen.tpg
```

5.3 Integrating **HBURG** Generated Code

After defining a code generator specification and running it through **HBURG** as depicted in Figure 5.8, the resulting code generator must be integrated into an existing compiler. Two interfaces, namely the **Code Generator** and the **Node** interface, are visible to the compiler writer when using **HBURG**'s code generator. The enumeration classes **Nt** and **Rule** as well as the **Entry** class need not concern the compiler writer.

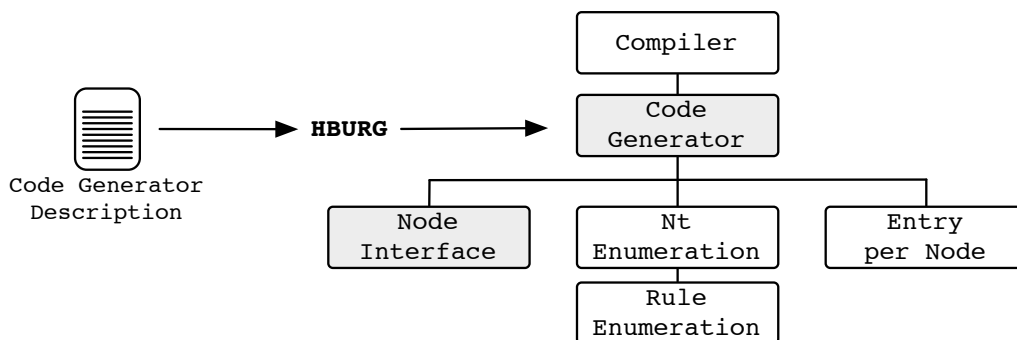


Figure 5.8: Input and Output of **HBURG**.

1. Node Interface

In order to traverse a tree-structured intermediate representation, **HBURG** requires an implementation of a **Node** interface that provides all methods necessary for tree traversal. Given a code generator specification, it infers how many children an intermediate representation node has, and if link nodes are present. That information is then used to emit the appropriate **Node** interface.

During the calculation of an optimal instruction sequence as outlined in Section 5.1.3, it is necessary to store selected rewrite rules, the non terminals they derive, as well as their cost, for each node. Figure 5.9 shows a **Node** interface emitted by **HBURG** for binary intermediate representation nodes with links to subtrees. The contract defined by the interface guarantees **HBURG** that methods to traverse and query intermediate representation nodes, and methods to store and retrieve information for optimal instruction sequence calculation, are implemented.

```

public interface Node {
    traverse — [ Node    child1();
                  Node    child2();
                  Node    link();
    query — [ Kind     kind();
              boolean  is(Nt nt);
              int     cost(Nt nt);
              Rule    rule(Nt nt);
    store and — [ Entry   put(Nt nt, Entry e);
    retrieve — [ Entry   get(Nt nt);
    }

```

Figure 5.9: Java **Node** interface emitted by **HBURG**.

2. Code Generator Invocation Interface

Invoking the code generator is as simple as calling its **emit()** method, passing it a reference to the root node of a tree-structured intermediate representation. Additionally, any parameters defined as input attributes for the *start rule* of a code generator specification must be passed to the **emit()** method. The return value of **emit()** is either **void**, or the output attribute specified by the *start rule*.

Appendix B provides a concrete example demonstrating how to implement the **Node** interface and how to invoke an **HBURG** generated code generator.

5.4 Summary

In this chapter we have outlined **HBURG**'s architecture, data structures, algorithms, and interfaces. We started by conveying **HBURG**'s module structure with respect to its compilation phases. We also described **HBURG**'s internal representation of code generator description grammars that is capable of capturing rewrite rules consisting of arbitrary tree patterns augmented with semantic actions and costs.

Scope and type analysis of grammar specifications was the topic of Section 5.1.2, indicating the types of errors that can be statically detected. Next, Section 5.1.3 introduced an algorithm for calculating an optimal instruction sequence given cost-augmented rewrite rules for a tree-structured intermediate representation. Having determined how to select optimal instructions, we looked at the code generation phase of the resulting code generator in Section 5.1.4. Finally, we described the integration and invocation of an **HBURG** generated code generator into a compiler infrastructure.

6

Conclusions

*This final chapter summarizes our approach to the field of automatic code generation as well as our contributions. We also suggest further improvements of our code generator description language and **HBURG**, our implementation of a code generator generator.*

In this master thesis we have designed a code generator description language together with a reference implementation of a code generator generator called **HBURG**. It generates code generators from a set of cost-augmented rewrite rules, that produce optimal code for tree-structured intermediate representations.

The principal mechanism for finding optimal instruction sequences is based on tree pattern matching combined with dynamic programming. Dynamic programming can either be delayed until the code generation phase of a compiler, or moved to code generator generator construction time by building a bottom up rewrite system automaton. We have decided in favor of the first approach despite the fact that it is necessarily slower than the latter table-driven approach. The reasons are as follows:

- Debugging code generators based on the first approach is simpler because information is recorded explicitly for each node, making it easier to compare the code generators' actual operations with expected results. The latter table-driven approach makes it very difficult to find
-

errors because the only output in such a case usually consists of impossible to understand numbers encoding some state in a table.

- The close resemblance of the first approach to recursive-descent $LL(k)$ parsers allows for great flexibility with respect to the placement of semantic actions. A table-driven approach only allows the definition of semantic actions at the end of a pattern.
- The resulting code generators are much smaller in size and easier to implement than their table-driven counterparts.

HBURG performs many static checks to verify the integrity of grammar specifications. Additionally, it tries hard to provide useful error messages in case mistakes happen.

We also focused on simplifying the integration of code generators into a compiler. In order to integrate a code generator produced by **HBURG**'s **Java** back end, only one simple interface must be implemented besides calling the respective code generation method.

6.1 Summary of Contributions

Given the current state of the art of code generators and code generator description languages, the following items list our main contributions in this field:

1. Ganapathi [8] was the first to suggest the use of attribute grammars for code generators based on **LR** parsing techniques. We have successfully shown that the attribute grammar formalism can also be integrated into code generators based on tree pattern matching combined with dynamic programming at compile time.
2. Other code generator description languages allow the definition of semantic actions only at the end of tree patterns. The language we devised allows semantic actions to be defined almost anywhere within tree patterns. This works because the second phase of the resulting code generator performs a top-down pass over a tree-structured intermediate representation that is already labeled with matching tree patterns.
3. Our tree pattern matching language also supports intermediate representations that consist of linked subtrees as suggested by Crelier [4] and outlined in Section 4.1.

6.2 Future Work

In future work we would like to improve the speed of the first bottom-up pass over the intermediate representation. Our current implementation uses recursive function calls where the compiler inserts code that manages the function stack upon each function invocation. But the overhead of a function call is expensive when all we need in order to traverse tree data structures is a simple stack that stores unprocessed nodes. Such an algorithm is outlined by Knuth [17] and we conjecture that compared with our current approach based on recursive function calls, it improves the performance of code generators, especially for target languages that do not support tail call optimizations. It is questionable though if the added complexity of manual stack management is worth the resulting speed improvement.

We would also like to improve **HBURG**'s back end. The process of implementing support for another target language should reveal opportunities where our current back end implementation can be improved with respect to modularity and extensibility.

EBNF has constructs that capture optional patterns '[...]' and sequences '{...}'. Our code generator language supports linked subtrees as optional patterns defined at the end of a rewrite rule. We would like to extend our code generator description language with full support for optional patterns and pattern sequences similar to **EBNF**.

Appendices



Language Syntax

$$\begin{aligned}
 digit &= '0' \mid \dots \mid '9'. \\
 alpha &= digit \mid 'a' \mid \dots \mid 'z' \mid 'A' \mid \dots \mid 'Z'. \\
 ident &= ('a' \mid \dots \mid 'z' \mid 'A' \mid \dots \mid 'Z') \{ alpha \mid '-' \mid '._' \}.
 \end{aligned}$$

G \rightarrow *Code Generator*

$G =$ **generator** *Sem* **declarations** *Sem*
operators *Op* **rules** *D* $\{ D \}$ **end**.

Op \rightarrow *Operator*

$Op = ident_u Sem \{ ', ' ident_u Sem \}.$

D \rightarrow *Definition*

$D = ident_l ['<.' Ad '>'] Sem '=' Prod '.'$

C \rightarrow *Cost*

$C = digit \{ digit \}$
 $\mid Sem.$

Prod \rightarrow *Production*

$$\begin{aligned} \text{Prod} = & \text{Prod } ']' \text{ Prod} \\ & | \text{Sem } T \text{ Sem} [\text{Pat Sem}] ['[' \text{ Sem Nt Sem } ']' \text{ Sem }] ':' C \\ & | \text{Sem Nt Sem} ['[' \text{ Sem Nt Sem } ']' \text{ Sem }] ':' C. \end{aligned}$$

Pat \rightarrow *Pattern*

$$\text{Pat} = '(' \text{ Elem } \{ ', ' \text{ Elem } \}.$$

Elem \rightarrow *Element*

$$\text{Elem} = \text{Sem} (\text{Nt Sem} \mid T \text{ Sem} [\text{Pat Sem}]).$$

Nt \rightarrow *Non Terminal*

$$\text{Nt} = \text{ident}_l ['<.' A '.' >'] [\text{ident}].$$

T \rightarrow *Terminal*

$$T = \text{ident}_u [\text{ident}].$$

Ad \rightarrow *Attribute Definition*

$$\begin{aligned} \text{Ad} = & \text{Ad } ', ' \text{ Ad} \\ & | [\text{out}] \text{Type ident}. \end{aligned}$$

A \rightarrow *Attribute*

$$\begin{aligned} A = & A ', ' A \\ & | [\text{out}] \text{ident} \\ & | \text{Ad}. \end{aligned}$$

Sem \rightarrow *Semantic Action*

$$\text{Sem} = ['(.' \dots '.')'].$$

Type \rightarrow *Data Type*

$$\text{Type} = \text{ident}.$$

EBNF [28] is used to describe the syntax of our code generator description language. There are two slight deviations from **EBNF** which require further explanation:

- Identifiers denoted by the non terminal *ident* include upper and lower case alphabetic characters. The subscripts *u* and *l* denote that an *ident* may *only* contain upper or lower case alphabetic characters.
- The dots in the definition of semantic actions $(. \dots .)$ denote that any character sequence except $.$ may occur between $(.$ and $.)$.

B

Code Generation Examples

*This Appendix presents two complete code generation examples. First, a high level example program and its intermediate representation are given. Next, two code generator specifications for **RISC** and **CISC** target architectures are presented. Finally, target machine code that is produced for the introductory example program is analyzed for each code generator.*

The following high level example program calculates and prints the greatest common divisor of two integral values:

```

1  PROGRAM
2  VAR r:INTEGER;
3  PROCEDURE GCD(x,y:INTEGER):INTEGER;
4  VAR rest,t:INTEGER;
5  BEGIN
6  IF y > x THEN t:=x; x:=y; y:=t; END;
7  rest:=x % y;
8  WHILE rest > 0 Do
9  x:=y; y:=rest; rest:=x % y;
10 END;
11 RETURN y;
12 END GCD;
13 BEGIN (* Main Program *)
14 r:=GCD(24,18); PUTINT(r);
15 END.
```

Listing B.1: Program calculating Greatest Common Divisor.

The program in Listing B.1 is written in a subset of the **Oberon-0** language (see Wirth [29]). Our goal is to write a compiler in **Java** that produces assembler code for programs written in this language. We will not concern ourselves with lexical analysis, parsing, type checking, or optimization. Instead, we focus on the compiler back end. First, we start by defining an intermediate representation. Then, we develop two code generators that map this representation onto **RISC** and **CISC** assembler code.

The back end of our compiler is generated automatically by **HBURG** from a code generator specification. But before we can write a code generator specification for a target architecture, we have to define our intermediate representation. The **Oberon-2** compiler uses a binary tree intermediate representation with extra pointers that link sequences of subtrees (see Figure 4.2 and Crelier [4]). Since our input language is a subset of **Oberon-0**, it makes sense to adopt a similar representation and define a binary **Node** as follows:

```
class Node {
    Kind      kind; // var,plus,assign...
    Node      left; // left son
    Node      right; // right son
    Node      link; // next tree
    Obj       obj;  // symbol table entry
    Constant  val;  // leaves that are constants
}
```

The following enumeration defines possible **Kind**'s of **Node**'s:

```
enum Kind {
    ADD,SUB,MUL,DIV,MOD, // integer arithmetic
    EQ,LEQ,GEQ,LTH,GTH, // comparison operators
    IF,IFE,EIF,WHILE,    // control flow
    CNST,VAR,            // constant,variable
    ASGN,                // assignment
    ENTER,RET,           // proc definition, return statement
    PROCP,PROC,          // proc call with or without parameters
    FUNP,FUN,            // fun call with or without parameters
    ROOTD,ROOT           // program with or without procedure definitions
}
```

Nodes representing variables, constants, and procedures, store references to elements of type **Obj** within a symbol table:

```
class Obj {
    String name; // symbol name
    Type type;   // Int,Void
    Kind kind;   // Cnst,Var,Param,Proc
    int  adr;    // Var,Param
    int  level;  // Var: 0 = global, 1 = local
}
```

```

    int varSize, parSize; // for procedures
    ...
}

```

For the sake of simplicity we only allow for `INTEGER` and `VOID` data Types in our source language:

```

enum Type {
    VOID(0); // 0 bytes
    INT(4);  // 4 bytes
    private int size; // in bytes
    Type(int s) { size = s; }
    public int size() { return size; }
}

```

With the previously defined data structures, we can now outline tree-structured intermediate representations for our language constructs. A language construct falls in one of two categories depending on the type of value it produces:

1. Constructs that return a value of type `VOID` belong to the category of *statements*. Assignments, control flow constructs, variable and procedure definitions, and calls to procedures returning values of type `VOID` belong to this category.
2. Any construct yielding a value that is not of type `VOID` belongs to the category of *expressions*. Integer arithmetic, comparison operators, and function calls are examples of this category.

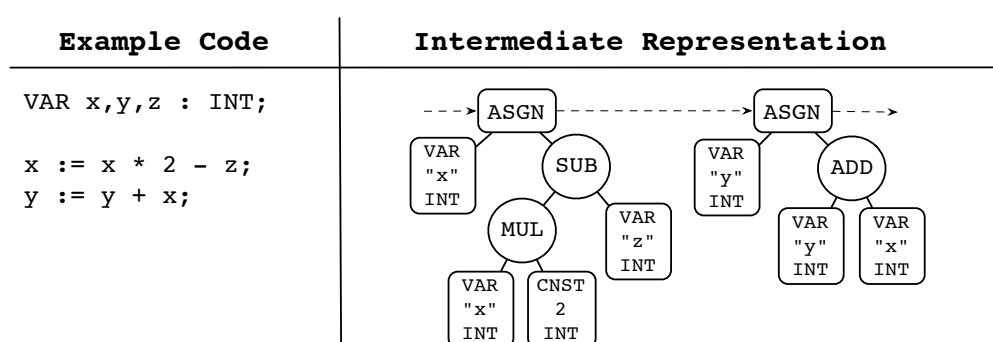


Figure B.1: Tree-structured IR for expressions and assignments.

Figure B.1 shows a tree-structured intermediate representation for two sequential assignment statements. Each expression and statement is mapped

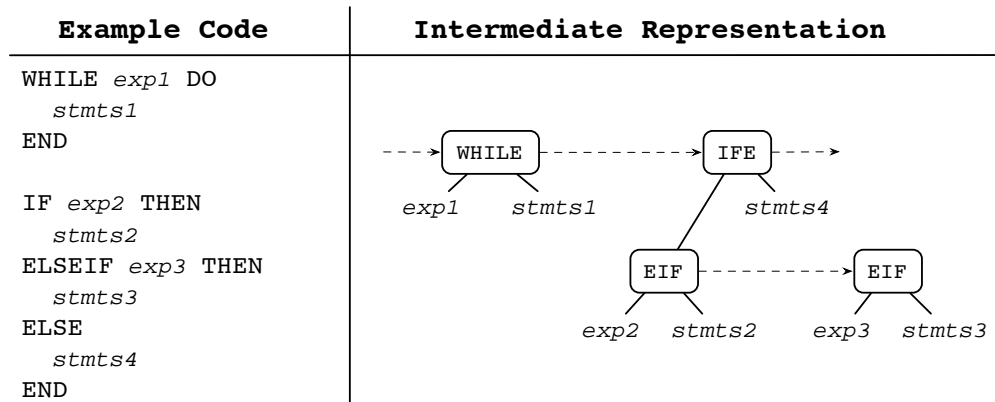


Figure B.2: Tree-structured IR for control flow statements.

onto a binary tree, and sequences of statements are connected via *link* references (see Node data type definition).

Figures B.2 and B.3 demonstrate intermediate representations for control flow statements and procedure calls together with their definitions. Figure B.4 depicts the intermediate representation for our introductory example program given in Listing B.1. After having defined a tree-structured representation for our input language, the next step consists of analyzing the target architecture by studying its instruction set, addressing modes, and memory layout.

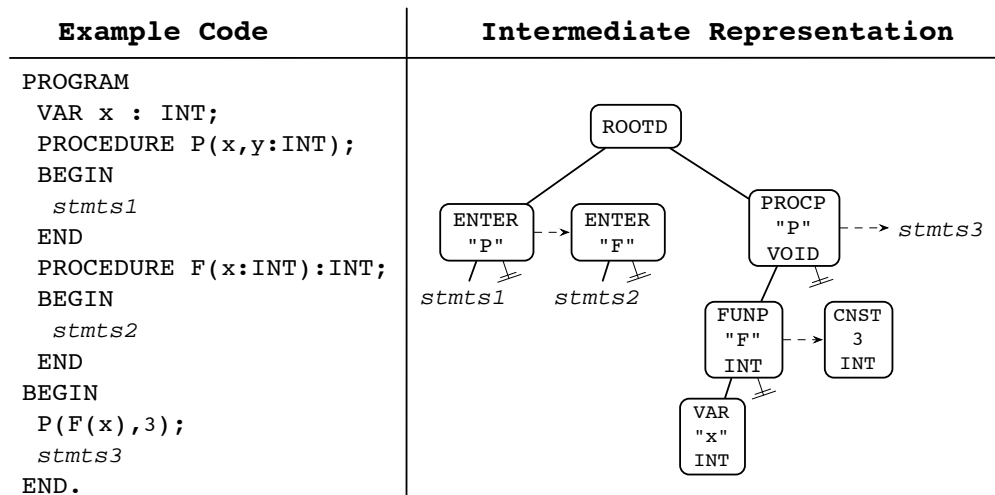


Figure B.3: Tree-structured IR for procedure definition and application.

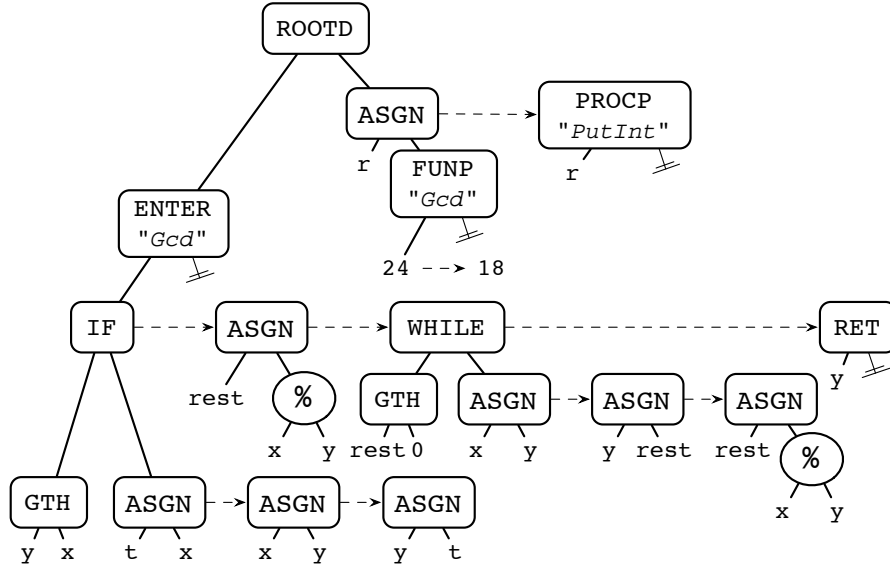


Figure B.4: Tree-structured IR for greatest common divisor example in Listing B.1.

Instruction sets for **RISC** architectures are highly regular because they store their result and expect their operands in registers. **CISC** architectures on the other hand provide instructions with irregular operand access patterns where each instruction may pose different restrictions on the location of its operands. Some instructions expect both of their operands to reside in registers, others can access one operand directly from memory while the other has to reside in a register, and there are instructions that restrict their operands to specific registers (e.g. IA-32 multiplication instruction **DIV**).

Given an instruction set, we have to tile our intermediate representation in such a way that each possible subtree is covered by its corresponding instruction sequence. A subtree cover is specified via a tree pattern and the value it derives depends on the result produced by the target instructions that implement it. Defining tree patterns for **RISC** architectures is straight forward since each instruction expects its operands in registers and produces a result in a register. Tree patterns for **CISC** architectures must capture various operand access patterns (e.g. register, immediate, memory).

The following two examples demonstrate how to specify a code generator description that maps the previously introduced intermediate representation onto **RISC** and **CISC** assembler code.

B.1 RISC Code Generator

In this section we first outline a simple RISC target architecture by defining its register set, available instructions, and the layout of the runtime stack. Finally we present a complete code generator specification for this architecture and present the output it produces for our introductory example in Listing B.1.

B.1.1 RISC Architecture Overview

The following Java enumeration outlines the available set of registers:

```
enum Reg {
    R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15,
    R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28,
    R29, // FP...frame pointer
    R30, // SP...stack pointer
    R31, // LNK...return address
    R0} // always zero
```

It is safe to store any values in registers R1 through R28, whereas registers R29, R30, R31, and R0 are special. R29 contains the current frame pointer (FP), R29 the stack pointer (SP), R31 the return address (LNK), and R0 is always zero. The allocation of registers is implemented in its most basic form by allocating registers on demand and freeing them as soon as possible.

Available instructions include integer arithmetic operations, comparison operations, conditional and unconditional branch operations as well as load and store operations:

```
enum Op {
    // Integer arithmetic
    ADD, SUB, MUL, DIV, MOD, // OP a,b,c - R.a := R.b OP R.c
    ADDI, SUBI, MULI, DIVI, MODI, // OP a,b,c - R.a := R.b OP c
    // Comparison instructions
    EQ, LEQ, GEQ, LTH, GTH, // CMP a,b,c - R.a := R.b CMP R.c
    EQI, LEQI, GEQI, LTHI, GTHI, // CMP a,b,c - R.a := R.b CMP c
    // Conditional and Unconditional branch instructions
    CBR, // CBR a,b,c - jump to address R.b if R.a is true, else to R.c
    JSR, // JSR c - save PC in R31, then jump to address in R.c
    JMP, // JMP c - jump to address in R.c
    // Load and Store instructions
    LDW, // LDW a,b,c - R.a := Mem[R.b + c] - load word
    STW} // STW a,b,c - Mem[R.b + c] := R.a - store word
```

Operands can refer to immediate values or values stored in registers and the following `Item` data type is used to encode them:

```
class Item {
    Mode mode; // CNST,VAR,REG
    Type type;
    Constant v; // CNST:value
    int off;    // VAR:offset
    Reg r;     // REG:register

    // create operand
    Item() { }
    Item(Mode m, Type t) { mode = m; type = t; }

    // emit operand
    public String toString() {
        switch (mode) {
            case CNST: { return v.toString(); }
            case VAR: { return Integer.toString(off); }
            case REG: { return r.toString(); }
            default: { return null; }
        }
    }
}
```

The runtime stack layout is depicted in Figure B.5. The stack starts from high addresses and grows downwards. Register R29 always contains the address to the beginning of the current frame, and register R30 always points to the current stack top. We assume that registers holding the stack and frame pointers are initialized appropriately before execution commences.

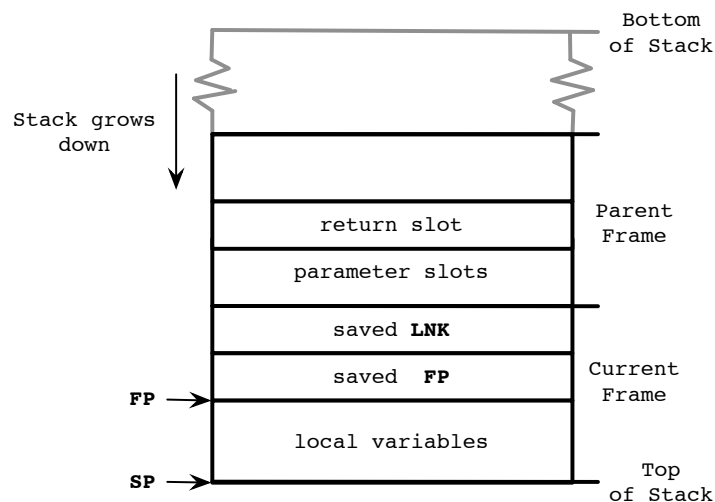


Figure B.5: RISC runtime stack layout.

B.1.2 RISC Code Generator Implementation

This section includes a RISC code generator specification together with examples that demonstrate how to invoke and integrate **HBURG** generated code. Furthermore the implementation of a **Code** manipulation class that provides common code generation functionality used throughout the specification is given as well.

```

generator -- RISC code generator specification
(.import static sl.ir.Kind.*; // node kinds
 import sl.parser.Obj;
 import sl.parser.SymTab;
 import sl.ir.Node;
 import sl.code.Lab;           // assembler labels
 import java.util.List;.)

declarations
(.private static Code c;    // target code manipulation
 private static SymTab t; // symbol table.)

operators
  ADD, SUB, MUL, DIV, MOD, EQ, LEQ, GEQ, LTH, GTH,
  ASGN, CNST, VAR, IF, IFE, EIF, WHILE,
  ENTER, RET, PROCP, PROC, FUNP, FUN, ROOTD, ROOT

rules

-- Start Rule
root <.out List<String> code, SymTab tab.>
  (.c = new Code(); code = c.code; t = tab;.)
  = (.c.put(Lab.MAIN);.)
  ROOT r(.Obj o = ((Node)r).obj; c.init(o.varSize);.) (stmtseq) :0
  | ROOTD r(.Obj o = ((Node)r).obj;.)
    (procdefseq(.c.put(Lab.MAIN); c.init(o.varSize);.) ,stmtseq):0.

-- Procedure Definitions
procdefseq = procdef [ procdefseq ] :0.
procdef
  = ENTER e(.Obj o = ((Node)e).obj;
             c.put(Lab.funLab(o.name,o.next));
             c.prologue(o.varSize);.)
    (stmtseq)
    (.c.epilogue(o.parSize);.):0.

```

```

-- Statements
stmtseq = stmt [ stmtseq ] :0.
stmt
= (.Lab t = new Lab(),f = new Lab();.)
  IF(cond<.out Item x.>
    (.c.put(Op.CBR,x.r,t,f); c.freeReg(x.r); c.put(t);.)
    ,stmtseq(.c.put(f);.))
:1
| (.Lab end = new Lab();.)
  IFE(eifseq<.Lab end.>,stmtseq(.c.put(end);.))
:0
| (.Lab loop = c.putLab(),t = new Lab(),f = new Lab();.)
  WHILE(cond<.out Item x.>
    (.c.put(Op.CBR,x.r,t,f); c.freeReg(x.r); c.put(t);.)
    ,stmtseq (.c.put(Op.JMP,loop); c.put(f);.))
:1
-- Store return value into corresponding memory slot
| RET r(.Obj o = ((Node)r).obj;.) (reg<.out Item x.>)
  (.c.storeRet(x.r,o.parSize,o.type.size());.)
:2
| ASGN(VAR v, reg<.out Item x.>)
  (.Item y = c.newItem((Node)v);
   c.put(Op.STW,x.r,y.r,y.off);
   c.freeReg(x.r);.)
:2
-- Discard return value since function is used in statement context
| FUN p(.Obj o = ((Node)p).obj;
  c.pushRet();
  c.put(Op.JSR,Lab.funLab(o.name,o.next));
  c.discardRet();.)
:2
| FUNP p(.Obj o = ((Node)p).obj;
  c.pushRet(); c.pushParams(o.parSize);.)
  (paramseq<.t.params(o).>) -- argument is a list of parameters
  (.c.put(Op.JSR,Lab.funLab(o.name,o.next));
   c.discardRet();.)
:4
-- Procedure Call
| PROC p(.Obj o = ((Node)p).obj;
  c.put(Op.JSR,Lab.funLab(o.name,o.next));.)
:1
| PROCP p(.Obj o = ((Node)p).obj; c.pushParams(o.parSize);.)
  (paramseq<.t.params(o).>)
  (.c.put(Op.JSR,Lab.funLab(o.name,o.next));.)
:2.

```

```

-- Function Call rewrite rule returns location of return value
fun<.out Item x.> (.x = null;.)
  = FUN p(.Obj o = ((Node)p).obj;
          c.pushRet();
          c.put(Op.JSR,Lab.funLab(o.name,o.next));
          x = c.popRet();.)
:3
  | FUNP p(.Obj o = ((Node)p).obj;
          c.pushRet();
          c.pushParams(o.parSize);.)
  (paramseq<.t.params(o).>)
  (.c.put(Op.JSR,Lab.funLab(o.name,o.next));
   x = c.popRet();.)
:4.

-- Parameters are pushed on stack in reverse order
paramseq<.List<Obj> lst.>
  = param<.lst.get(0).> (.lst.remove(0);.) [ paramseq<.lst.> ] :0.
param<.Obj o.>
  = reg<.out Item x.>
    (.c.put(Op.STW,x.r,c.SP,c.offset(o)); c.freeReg(x.r);.) :1.

-- Sequence of ELSIF statements
elseifseq<.Lab end.>
  = eif<.end.> [ elseifseq<.end.> ] :0.
eif<.Lab end.> (.Lab t = new Lab(),f = new Lab();.)
  = EIF(cond<.out Item x.>
        (.c.put(Op.CBR,x,t,f); c.freeReg(x.r);
         c.put(t);.)
        ,stmtseq
        (.c.put(Op.JMP,end); c.put(f);.) ) :1.

-- Conditionals returning values stored in registers
cond<.out Item x.> (.x = null; Item a,y;.)
  = EQ (reg<.out x.>,reg<.out y.>) (.c.put(Op.EQ,x,y);.) :2
  | EQ (reg<.out x.>,imm<.out a.>) (.c.put(Op.EQI,x,a);.) :1
  | EQ (imm<.out a.>,reg<.out x.>) (.c.put(Op.EQI,x,a);.) :1
  | LEQ (reg<.out x.>,reg<.out y.>) (.c.put(Op.LEQ,x,y);.) :2
  | LEQ (imm<.out a.>,reg<.out x.>) (.c.put(Op.GTHI,x,a);.) :1
  | LEQ (reg<.out x.>,imm<.out a.>) (.c.put(Op.LEQI,x,a);.) :1
  | GEQ (reg<.out x.>,reg<.out y.>) (.c.put(Op.GEQ,x,y);.) :2
  | GEQ (imm<.out a.>,reg<.out x.>) (.c.put(Op.LTHI,x,a);.) :1
  | GEQ (reg<.out x.>,imm<.out a.>) (.c.put(Op.GEQI,x,a);.) :1
  | LTH (reg<.out x.>,reg<.out y.>) (.c.put(Op.LTH,x,y);.) :2
  | LTH (imm<.out a.>,reg<.out x.>) (.c.put(Op.GEQI,x,a);.) :1
  | LTH (reg<.out x.>,imm<.out a.>) (.c.put(Op.LTHI,x,a);.) :1
  | GTH (reg<.out x.>,reg<.out y.>) (.c.put(Op.GTH,x,y);.) :2
  | GTH (imm<.out a.>,reg<.out x.>) (.c.put(Op.LEQI,x,a);.) :1
  | GTH (reg<.out x.>,imm<.out a.>) (.c.put(Op.LTHI,x,a);.) :1.

```

```

-- Patterns returning immediate values
imm<.out Item x.> (.x = null; Item y;.)
  = CNST a(.x = c.newCnstItem((Node)a);.) :0
  -- Simple constant folding
  | ADD (imm<.out x.>,imm<.out y.>) (.c.fold(Op.ADD,x,y);.) :0
  | SUB (imm<.out x.>,imm<.out y.>) (.c.fold(Op.SUB,x,y);.) :0
  | MUL (imm<.out x.>,imm<.out y.>) (.c.fold(Op.MUL,x,y);.) :0
  | DIV (imm<.out x.>,imm<.out y.>) (.c.fold(Op.DIV,x,y);.) :0
  | MOD (imm<.out x.>,imm<.out y.>) (.c.fold(Op.MOD,x,y);.) :0.

-- Patterns returning values stored in registers
reg<.out Item x.> (.x = null; Item a,y;.)
  = cond<.out x.> :0
  | fun <.out x.> :0
  | imm <.out x.> (.c.load(x);.) :1
  | VAR v (.x = c.load(c.newItem((Node)v));.) :2
  | ADD (reg<.out x.>,reg<.out y.>) (.c.put(Op.ADD,x,y);.) :2
  | ADD (imm<.out a.>,reg<.out x.>) (.c.put(Op.ADDI,x,a);.) :1
  | ADD (reg<.out x.>,imm<.out a.>) (.c.put(Op.ADDI,x,a);.) :1
  | SUB (reg<.out x.>,reg<.out y.>) (.c.put(Op.SUB,x,y);.) :2
  | SUB (reg<.out x.>,imm<.out a.>) (.c.put(Op.SUBI,x,a);.) :1
  | MUL (reg<.out x.>,reg<.out y.>) (.c.put(Op.MUL,x,y);.) :2
  | MUL (imm<.out a.>,reg<.out x.>) (.c.put(Op.MULI,x,a);.) :1
  | MUL (reg<.out x.>,imm<.out a.>) (.c.put(Op.MULI,x,a);.) :1
  | DIV (reg<.out x.>,reg<.out y.>) (.c.put(Op.DIV,x,y);.) :2
  | DIV (reg<.out x.>,imm<.out a.>) (.c.put(Op.DIVI,x,a);.) :1
  | MOD (reg<.out x.>,reg<.out y.>) (.c.put(Op.MOD,x,y);.) :2
  | MOD (reg<.out x.>,imm<.out a.>) (.c.put(Op.MODI,x,a);.) :1.

end

```

In order to generate a code generator from the previous specification, **HBURG** is invoked with the following parameters (see Section 5.2):

```
$ hburg -t sl.ir.Kind -p sl.code.risc sl/code/risc/RISC.tpg
```

The following Java statement invokes the generated code generator, where the variable **root** denotes the root of an intermediate representation tree, and **symTab** is a reference to a symbol table (see Section 5.3):

```
List<String> code = sl.code.risc.CodeGen.emit(root,symTab);
```


HBURG also emits a `Node` interface specification as outlined in Figure 5.9. This interface must be implemented by our intermediate representation `Node` data type:

```

package sl.ir; // implementation of HBURG generated Node interface
import sl.parser.Obj;
import java.util.EnumMap;
// HBURG generated enumerations
import sl.code.risc.Nt;
import sl.code.risc.Rule;
import sl.code.risc.Entry;

public class Node implements sl.code.risc.Node {
    public Kind kind; // VAR, PLUS, ASGN, IF, ...
    public Node left, right; // to the sons
    public Node link; // to the next tree
    public Obj obj; // symbol table entry
    public Constant val; // for leaves that are constants
    // Table holding per Node dynamic programming information
    EnumMap<Nt, Entry> tab = new EnumMap<Nt, Entry>(Nt.class);

    // create
    public Node() { }
    public Node(Kind k) { kind = k; }
    public Node(Kind k, Constant v) { kind = k; val = v; }

    // traverse
    public sl.code.risc.Node child1() { return left; }
    public sl.code.risc.Node child2() { return right; }
    public sl.code.risc.Node link() { return link; }

    // query
    public sl.ir.Kind kind() { return kind; }
    public boolean is(Nt nt) { return tab.containsKey(nt); }
    public int cost(Nt nt) {
        Entry e = tab.get(nt);
        return (e != null) ? e.cost : Integer.MAX_VALUE;
    }
    public Rule rule(Nt nt) {
        Entry e = tab.get(nt);
        return (e != null) ? e.rule : null;
    }
}

// store and retrieve dynamic programming information
public Entry put(Nt nt, Entry e) { return tab.put(nt, e); }
public Entry get(Nt nt) { return tab.get(nt); }
}

```

Our code generator specification utilized methods from a **RISC Code** manipulation class that implements common code generation and register allocation functionality, and maintains the actual list of target instructions:

```

package sl.code.risc; // RISC code manipulation class
import static sl.code.risc.Reg.*; // register set
import static sl.code.risc.Op.*; // instruction set
import static sl.code.risc.Mode.*; // addressing modes
import sl.parser.Obj;
import sl.parser.Type;
import sl.parser.Kind;
import sl.ir.Node;
import sl.code.Lab;
import java.util.Set;
import java.util.EnumSet;
import java.util.List;
import java.util.LinkedList;
import java.io.PrintStream;

public class Code {
    static String clazz = Code.class.getName();
    static PrintStream err = System.err;
    static Reg FP = R29, SP = R30, LNK = R31;
    static short SIZ = 4; // 4 Bytes
    Set<Reg> regs; // used registers
    List<String> code; // RISC assembler code

    public Code() {
        regs = EnumSet.noneOf(Reg.class);
        code = new LinkedList<String>();
    }
    // ----- //
    void init(int size) { put(ADDI, SP, FP, -size); }
    int offset(Obj o) { return o.adr * o.type.size(); }
    // ----- //
    Item newCnstItem(Node n) {
        Item i = new Item(CNST, n.obj.type); i.v = n.val;
        return i;
    }
    Item newItem(Node n) {
        Item i = new Item(VAR, n.obj.type);
        if (n.obj.level == 0 // global vars
            && n.obj.kind != Kind.PARAM) {
            i.off = -offset(n.obj);
        } else { // local vars and parameters
            i.off = (n.obj.kind == Kind.PARAM)
                ? (2*SIZ)+offset(n.obj) // param
                : -offset(n.obj); // local var
        }
    }
}

```

```

    }
    i.r = FP; // address is always relative to FP
    return i;
}
// ----- //
void put(Object o, Object a, Object b, Object c) {
    code.add(o + " " + a + ", " + b + ", " + c);
}
void put(Op o, Reg a, Reg b, int c) {
    put(o, a, b, Integer.toString(c));
}
void put(Op o, Item b, Item c) {
    Reg a = getReg(); put(o, a, b, c); freeReg(b.r); freeReg(c.r);
    b.r = a;
}
void put(Op o, Lab l) { code.add(o + " " + l); }
void put(Lab l) { code.add(l + ":"); }
Lab putLab() { Lab l = new Lab(); put(l); return l; }
// ----- //
void fold(Op op, Item x, Item y) {
    int a = x.v.ival, b = y.v.ival;
    switch (op) {
        case ADD: x.v.ival = a + b; break;
        case SUB: x.v.ival = a - b; break;
        case MUL: x.v.ival = a * b; break;
        case MOD: x.v.ival = a % b; break;
        case DIV: x.v.ival = a / b; break; // division by 0 not handled
        default: err.printf("E:%s: Unknown Op '%s'!\n", clazz, op);
    }
}
// ----- //
Reg getReg() {
    for (Reg r : Reg.values())
        if (!regs.contains(r) && (r.ordinal() < R29.ordinal())) {
            regs.add(r);
            return r;
        }
    err.printf("E:%s: Can not allocate register.\n", clazz);
    return null;
}
void freeReg(Reg r) { regs.remove(r); }
// ----- //
Item load(Item x) {
    if (x.mode == VAR || x.mode == CNST) {
        Reg r = getReg();
        if (x.mode == VAR) {
            put(LDW, r, x.r, x.off);
            freeReg(x.r);
        } else

```

```

        put(ADDI,r,R0,x.v);
        x.r = r; x.mode = REG;
    } else err.printf("E:%s: Wrong mode '%s'!\n",clazz,x.mode);
    return x;
}
// ----- //
// PSH memory slots for parameters on stack
void pushParams(int size) { put(ADDI,SP,SP,-size); }
// PSH memory slot for return value on stack
void pushRet() { put(ADDI,SP,SP,-SIZ); }
// POP return value from stack into register
Item popRet() {
    Item i = new Item();
    i.r = getReg();
    put(LDW,i.r,SP,SIZ); // return value is below SP
    discardRet();
    return i;
}
// discard return value
void discardRet() { put(SUBI,SP,SP,SIZ); }
// Store return value into allocated memory slot:
// @param register
// @param param size
// @param return value size
void storeRet(Reg r,int psize,int rsize) {
    put(STW,r,FP,(2*SIZ)+psize+rsize);
    freeReg(r);
}
// procedure prologue:
// @param frame size
void prologue(int fsize) {
    put(STW,LNK,SP,-SIZ); // store LNK
    put(STW,FP,SP,-2*SIZ); // store FP
    put(ADDI,FP,SP,-2*SIZ); // new FP = SP - 8 bytes
    put(ADDI,SP,FP,-fsize); // adjust SP = FP - fsize
}
// procedure epilogue:
// @param param size
void epilogue(int psize) {
    put(ADDI,SP,FP,psize+(2*SIZ)); // restore SP
    put(LDW,LNK,FP,2*SIZ); // restore LNK
    put(LDW,FP,FP,SIZ); // restore FP
    put(JMP,R0,R0,LNK); // return to caller
}
}

```

B.1.3 RISC Assembly Output

The goal was to output RISC assembly code for the introductory example in Listing B.1, and the automatically generated RISC code generator outputs the following target code for it:

```

$Ggt INT INT:      ; ----- Ggt(Int x,Int y): -----
STW R31,R30,-4     ; PROLOGUE: PSH LNK,SP,-4...store LNK
STW R29,R30,-8     ; PROLOGUE: PSH FP,SP,-8...store FP
ADDI R29,R30,-8    ; PROLOGUE: FP := SP-8...new FP
ADDI R30,R29,-8    ; PROLOGUE: SP := FP-size(locals)...new SP
LDW R1,R29,16      ; load param y
LDW R2,R29,12      ; load param x
GTH R3,R1,R2       ; compare: R1 := x > y
CBR R3,$L1,$L2     ; if (x > y) then JMP $TRU else JMP $FLS
$L1:              ; $TRU
LDW R1,R29,12      ; load param x
STW R1,R29,-4      ; t := x
LDW R1,R29,16      ; load param y
STW R1,R29,12      ; x := y
LDW R1,R29,-4      ; load local variable t
STW R1,R29,16      ; y := t
$L2:              ; $FLS
LDW R1,R29,12      ; load param x
LDW R2,R29,16      ; load param y
MOD R3,R1,R2       ; calculate x % y
STW R3,R29,0       ; rest := x % y
$L3:              ; $WHILE
LDW R1,R29,0       ; load rest
LTHI R2,R1,0       ; compare: R1 := rest > 0
CBR R2,$L4,$L5     ; if (rest > 0) then JMP $BODY else JMP $END
$L4:              ; $BODY
LDW R1,R29,16      ; load param y
STW R1,R29,12      ; x := y
LDW R1,R29,0       ; load local variable rest
STW R1,R29,16      ; y := rest
LDW R1,R29,12      ; load param x
LDW R2,R29,16      ; load param y
MOD R3,R1,R2       ; x % y
STW R3,R29,0       ; rest := x%y
JMP $L3           ; JMP $WHILE
$L5:              ; $END
LDW R1,R29,16      ; load param y
STW R1,R29,20      ; store return value into return slot
ADDI R30,R29,16    ; EPILOGUE: SP := FP+size(params)...restore SP
LDW R31,R29,8      ; EPILOGUE: LNK := FP+8...restore LNK
LDW R29,R29,4      ; EPILOGUE: FP := FP+4...restore FP

```

```
JMP R0,R0,R31      ; EPILOGUE: Return to caller
$main:             ; ----- ENTRY POINT -----
ADDI R30,R29,-4     ; init SP := FP-size(globals)
ADDI R30,R30,-4     ; PSH return slot for Ggt() on stack
ADDI R30,R30,-8     ; PSH param slots for Ggt() on stack
ADDI R1,R0,24       ; load 24
STW R1,R30,4        ; store 4 in 1. param slot
ADDI R1,R0,18       ; load 18
STW R1,R30,8        ; store 18 in 2. param slot
JSR $Ggt_INT_INT    ; LNK=PC; JMP $Ggt_INT_INT
LDW R1,R30,4        ; load return value
SUBI R30,R30,4      ; POP return value off stack
STW R1,R29,0        ; r := Ggt(24,18)
ADDI R30,R30,-4     ; PSH param slot for PutInt() on stack
LDW R1,R29,0        ; load local variable r
STW R1,R30,4        ; store R1 in 1. param slot
JSR $PUT_INT        ; LNK=PC; JMP $PUT_INT
```

Note that the procedure `PUT_INT` on line 14 in Listing B.1 is a library procedure and outputs an integral value to the console. Its implementation is not presented in this example.

B.2 CISC Code Generator

In this section we outline the IA-32 CISC target architecture by defining its register set, available instructions, and the layout of the runtime stack. Finally we present a complete code generator specification for this architecture and present the output it produces for our introductory example in Listing B.1.

B.2.1 CISC Architecture Overview

The following Java enumeration outlines the available set of registers:

```
enum Reg {
    EAX,EBX,ECX,EDX,ESI,EDI,
    ESP, // stack pointer
    EBP} // base pointer
```

Registers ESP and EBP contain the current stack and base pointer which leaves six general purpose registers for free use. The instruction pointer register EIP contains the address of the next instruction to be executed. It can not be accessed directly, but is modified implicitly by control flow instructions.

Available instructions include integer arithmetic operations, comparison operations, conditional and unconditional jump operations, as well as runtime stack management instructions:

```
enum Op {
    // Integer arithmetic
    ADD,SUB,IMUL,IDIV,MOD,
    // Data instructions
    MOV, // MOV dst,src - dst := src
    LEA, // LEA dst,src - dst := Adr(src)
    CWD, // Convert Word to Double word
    // Comparison instruction
    CMP, // CMP a,b (sets EFLAGS register)
    // Conditional jump (evaluates EFLAGS register)
    JE,JNE,JL,JLE,JG,JGE,JZ,JNZ,
    // Unconditional jump
    JMP, // JMP c - unconditional jump to c
    // Stack management instructions
    PUSH,POP,
    CALL, // CALL P; PSH EIP; EIP := Adr(P);
    ENTER, // PSH EBP; EBP := ESP; ESP := ESP-size - setup stack frame
    LEAVE, // ESP := EBP; EBP := POP - release stack fram
```

```
RET} // EIP := POP; ESP := ESP+size(params)
```

IA-32 instructions operate on zero or more operands. Some instructions operate on explicitly defined operands whereas some access operands implicitly (e.g. IA-32 CWD instruction). Operands can denote immediate values, registers, or memory locations. We use the following `Item` data type to represent operands for the IA-32 architecture:

```
public class Item {
    Mode mode; // CNST,ABS,REG,REGREL
    Type type;
    Constant v; // CNST: constant value, ABS: address
    Reg r;      // REG,REGREL: register
    int off;    // REGREL: offset
    // create operand
    Item() { }
    Item(Mode m, Type t) { mode = m; type = t; }
    // emit operand
    public String toString() {
        switch (mode) {
            case CNST: { return v.toString(); }
            case ABS:  { return "DS:" + off; }
            case REG:  { return r.toString(); }
            case REGREL: { return off + "[" + r + ""]; }
            default:   { return null; }
        }
    }
}
```

Memory operands can either be accessed by an absolute address (ABS) or relative to the address stored in a register (REGREL). The IA-32 architecture offers an additional indexed addressing mode where an index register multiplied by a scale factor is added to a base register (e.g. `base + (index * scale)`) to form a memory address. Since our input language lacks constructs that would benefit from indexed memory addressing, our code generator does not emit such operands.

The runtime stack layout is depicted in Figure B.6 and is similar to the stack layout of the previously defined RISC architecture (see Section B.1.1). Register EBP always contains the address to the beginning of the current frame, and register ESP always points to the current stack top. We assume that registers holding the stack and frame pointers are initialized appropriately before execution starts.

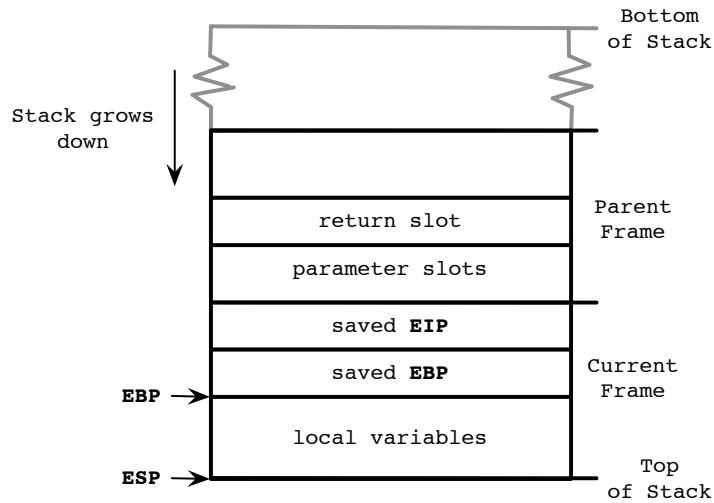


Figure B.6: CISC runtime stack layout.

B.2.2 CISC Code Generator Implementation

This section includes a CISC code generator specification together with an implementation of a `Code` manipulation class that includes common code generation functionality. The implementation of the `Node` interface emitted by `HBURG` is omitted since it is almost identical to the previous RISC example.

```

generator -- CISC code generator specification
(.import static sl.ir.Kind.*; // node kinds
 import sl.parser.Obj;
 import sl.parser.SymTab;
 import sl.ir.Node;
 import sl.code.Lab;          // assembler labels
 import java.util.List;.)

declarations
(.private static Code c;    // target code manipulation
 private static SymTab t; // symbol table.)

operators
  ADD, SUB, MUL, DIV, MOD, EQ, LEQ, GEQ, LTH, GTH,
  ASGN, CNST, VAR, IF, IFE, EIF, WHILE,
  ENTER, RET, PROCP, PROC, FUNP, FUN, ROOTD, ROOT

```

```

rules
-- Start Rule
root <.out List<String> code, SymTab tab .>
    (.c = new Code(); t = tab; code = c.code;.)
    = (.c.put(Lab.MAIN);.)
    ROOT (stmtseq) :0
    | ROOTD (procdefseq (.c.put(Lab.MAIN);.) ,stmtseq) :0.

-- Procedure Definitions
procdefseq = procdef [ procdefseq ] :0.
procdef
    = ENTER e(.Obj o = ((Node)e).obj;
                c.put(Lab.funLab(o.name,o.next));
                c.put(Op.ENTER,o.varSize,0);.)
        (stmtseq)
        (.c.put(Op.LEAVE); c.put(Op.RET,o.parSize);.) :0.

-- Statements
stmtseq = stmt [ stmtseq ] :0.
stmt
    = (.Lab f = new Lab();.)
      IF(cond<.f.>,stmtseq (.c.put(f);.))
      :0
    | (.Lab end = new Lab();.)
      IFE(eifseq<.end.>,stmtseq (.c.put(end);.))
      :0
    | (.Lab f = new Lab(),loop = new Lab(); c.put(loop);.)
      WHILE(cond<.f.>,stmtseq (.c.put(Op.JMP,loop); c.put(f);.))
      :1
    -- Store return value into corresponding memory slot
    | RET r(.Obj o = ((Node)r).obj;.) (reg<.out Item x.>)
      (.c.storeRet(x,o.parSize,o.type.size());.)
      :1
    | RET r(.Obj o = ((Node)r).obj;.) (imm<.out Item x.>)
      (.c.storeRet(x,o.parSize,o.type.size());.)
      :1
    | RET r(.Obj o = ((Node)r).obj;.) (mem<.out Item x.>)
      (.c.storeRet(x,o.parSize,o.type.size());.)
      :2
    -- Free all registers after assignment
    | ASGN(mem<.out Item x.>, reg<.out Item y.>)
      (.c.put(Op.MOV,x,y); c.freeAllRegs();.)
      :1
    | ASGN(mem<.out Item x.>, imm<.out Item y.>)
      (.c.put(Op.MOV,x,y); c.freeAllRegs();.)
      :1

```

```

-- Discard return value since function is used in statement context
| FUN p(.Obj o = ((Node)p).obj;
      c.pushRet();
      c.put(Op.CALL, Lab.funLab(o.name,o.next));
      c.discardRet();.)

:3
| FUNP p(.Obj o = ((Node)p).obj;
      c.pushRet();
      c.pushParams(o.parSize);.)
  (paramseq<t.params(o)>) -- argument is a list of parameters
  (.c.put(Op.CALL,Lab.funLab(o.name,o.next));
   c.discardRet();.)

:3
-- Procedure Call
| PROC p(.Obj o = ((Node)p).obj;
      c.put(Op.CALL,Lab.funLab(o.name,o.next));.)

:1
| PROCP p(.Obj o = ((Node)p).obj; c.pushParams(o.parSize);.)
  (paramseq<t.params(o)>)
  (.c.put(Op.CALL,Lab.funLab(o.name,o.next));.)

:2.

-- Function call rewrite rule returns location of return value
fun<.out Item x.> (.x = null;.)
= FUN p(.Obj o = ((Node)p).obj;
      c.pushRet();
      c.put(Op.CALL,Lab.funLab(o.name,o.next));
      x = c.popRet();.)

:3
| FUNP p(.Obj o = ((Node)p).obj;
      c.pushRet();
      c.pushParams(o.parSize);.)
  (paramseq<t.params(o)>)
  (.c.put(Op.CALL,Lab.funLab(o.name,o.next));
   x = c.popRet();.)

:3.

-- Parameters are pushed on stack in reverse order
paramseq<.List<Obj> l.>
= param<.l.get(0).> (.l.remove(0);.) [ paramseq<.l.> ] :0.
param<.Obj o.>(.Item x = null;.)
= imm<.out x.>(.c.put(Op.MOV,c.offset(o)+"["+ c.SP +"]",x);.) :1
| reg<.out x.>(.c.put(Op.MOV,c.offset(o)+"["+ c.SP +"]",x);
              c.freeReg(x);.) :1.

```

```

-- Sequence of ELSIF statements
elseifseq<.Lab end.> = elseif<.end.> [ elseifseq<.end.> ] :0.
elseif<.Lab end.>(.Lab l = new Lab();.)
    = EIF(cond<.l.>,stmtseq(.c.put(Op.JMP,end); c.put(1);.)) :1.

-- Conditionals use inverted jump operators and set EFLAGS register
cond<.Lab l.> (.Item x,y;.)
    = EQ (reg<.out x.>, reg<.out y.>) (.c.putCond(1,Op.JNE,x,y);.) :2
    | EQ (reg<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JNE,x,y);.) :1
    | EQ (reg<.out x.>, mem<.out y.>) (.c.putCond(1,Op.JNE,x,y);.) :3
    | EQ (mem<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JNE,x,y);.) :2
    | LEQ (reg<.out x.>, reg<.out y.>) (.c.putCond(1,Op.JG,x,y);.) :2
    | LEQ (reg<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JG,x,y);.) :1
    | LEQ (reg<.out x.>, mem<.out y.>) (.c.putCond(1,Op.JG,x,y);.) :3
    | LEQ (mem<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JG,x,y);.) :2
    | GEQ (reg<.out x.>, reg<.out y.>) (.c.putCond(1,Op.JL,x,y);.) :2
    | GEQ (reg<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JL,x,y);.) :1
    | GEQ (reg<.out x.>, mem<.out y.>) (.c.putCond(1,Op.JL,x,y);.) :3
    | GEQ (mem<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JL,x,y);.) :2
    | LTH (reg<.out x.>, reg<.out y.>) (.c.putCond(1,Op.JGE,x,y);.) :2
    | LTH (reg<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JGE,x,y);.) :1
    | LTH (reg<.out x.>, mem<.out y.>) (.c.putCond(1,Op.JGE,x,y);.) :3
    | LTH (mem<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JGE,x,y);.) :2
    | GTH (reg<.out x.>, reg<.out y.>) (.c.putCond(1,Op.JLE,x,y);.) :2
    | GTH (reg<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JLE,x,y);.) :1
    | GTH (reg<.out x.>, mem<.out y.>) (.c.putCond(1,Op.JLE,x,y);.) :3
    | GTH (mem<.out x.>, imm<.out y.>) (.c.putCond(1,Op.JLE,x,y);.) :2.

-- Patterns returning immediate values
imm<.out Item x.> (.Item y;.)
    = CNST a(.x = c.newCnstItem((Node)a);.) :0
    -- Constant folding
    | ADD (imm<.out x.>, imm<.out y.>) (.c.fold(Op.ADD,x,y);.) :0
    | SUB (imm<.out x.>, imm<.out y.>) (.c.fold(Op.SUB,x,y);.) :0
    | MUL (imm<.out x.>, imm<.out y.>) (.c.fold(Op.IMUL,x,y);.) :0
    | DIV (imm<.out x.>, imm<.out y.>) (.c.fold(Op.IDIV,x,y);.) :0
    | MOD (imm<.out x.>, imm<.out y.>) (.c.fold(Op.MOD,x,y);.) :0.

-- Patterns returning values stored in memory
mem<.out Item x.> (.x = null; Item y;.)
    = VAR a(.x = c.newItem((Node)a);.) :0
    | ADD (mem<.out x.>, imm<.out y.>) (.c.put(Op.ADD,x,y);.) :2
    | ADD (mem<.out x.>, reg<.out y.>) (.c.put(Op.ADD,x,y);.) :2
    | SUB (mem<.out x.>, imm<.out y.>) (.c.put(Op.SUB,x,y);.) :2
    | SUB (mem<.out x.>, reg<.out y.>) (.c.put(Op.SUB,x,y);.) :2.

```

```

-- Patterns returning values stored in registers
reg<.out Item x.> (.x = null; Item y;.)
= imm<.out y.> (.x = c.load(y);.) :1
| mem<.out y.> (.x = c.load(y);.) :1
| fun<.out x.> :0
| ADD (reg<.out x.>, imm<.out y.>) (.c.put(Op.ADD,x,y);.) :1
| ADD (reg<.out x.>, reg<.out y.>) (.c.put(Op.ADD,x,y);.) :2
| ADD (reg<.out x.>, mem<.out y.>) (.c.put(Op.ADD,x,y);.) :3
| SUB (reg<.out x.>, imm<.out y.>) (.c.put(Op.SUB,x,y);.) :1
| SUB (reg<.out x.>, reg<.out y.>) (.c.put(Op.SUB,x,y);.) :2
| SUB (reg<.out x.>, mem<.out y.>) (.c.put(Op.SUB,x,y);.) :3
| MUL (reg<.out x.>, imm<.out y.>) (.c.put(Op.IMUL,x,y);.) :1
| MUL (reg<.out x.>, reg<.out y.>) (.c.put(Op.IMUL,x,y);.) :2
| MUL (reg<.out x.>, mem<.out y.>) (.c.put(Op.IMUL,x,y);.) :3
| DIV (reg<.out x.>, reg<.out y.>) (.c.putDiv(x,y);.) :2
| DIV (reg<.out x.>, mem<.out y.>) (.c.putDiv(x,y);.) :3
| MOD (reg<.out x.>, reg<.out y.>) (.c.putMod(x,y);.) :2
| MOD (reg<.out x.>, mem<.out y.>) (.c.putMod(x,y);.) :3.

end

```

In order to generate a code generator from the previous specification, **HBURG** is invoked with the following parameters (see Section 5.2):

```
$ hburg -t sl.ir.Kind -p sl.code.cisc sl/code/cisc/CISC.tpg
```

The following Java statement invokes the generated code generator, where the variable **root** denotes the root of an intermediate representation tree, and **symTab** is a reference to a symbol table (see Section 5.3):

```
List<String> code = sl.code.cisc.CodeGen.emit(root,symTab);
```

The previously defined code generator utilized methods from a **CISC Code** manipulation class that implements common code generation and register allocation functionality, and maintains the actual list of target instructions:

```

package sl.code.cisc; // CISC code manipulation class
import static sl.code.cisc.Reg.*; // register set
import static sl.code.cisc.Op.*; // instruction set
import static sl.code.cisc.Mode.*; // addressing modes
import sl.parser.Obj;
import sl.parser.Type;
import sl.parser.Kind;
import sl.ir.Node;
import sl.code.Lab;
import java.util.EnumSet;
import java.util.LinkedList;
import java.util.Set;
import java.util.List;
import java.util.Stack;
import java.io.PrintStream;

public class Code {
    static String clazz = Code.class.getName();
    static PrintStream err = System.err;
    static Reg FP = EBP, SP = ESP;
    static short SIZ = 4; // 4 Bytes

    Set<Reg> regs; // used registers
    Stack<Reg> spill; // spilled registers
    List<String> code; // CISC ASM code

    public Code() {
        regs = EnumSet.noneOf(Reg.class);
        spill = new Stack<Reg>();
        code = new LinkedList<String>();
    }
    // ----- //
    static int offset(Obj o) { return o.adr * o.type.size(); }
    // ----- //
    static Item newCnstItem(Node n) {
        Item i = new Item(CNST, n.obj.type); i.v = n.val;
        return i;
    }
    static Item newItem(Node n) {
        Item i = new Item();
        if (n.obj.level == 0
            && n.obj.kind != Kind.PARAM) {
            i.mode = ABS; i.off = offset(n.obj); // global var
        } else { // local

```

```

        i.mode = REGREL; i.r = FP;
        i.off = (n.obj.kind == Kind.PARAM)
            ? (2*SIZ)+offset(n.obj) // param
            : -offset(n.obj); // local var
    }
    return i;
}
// ----- //
void put(Op o, Object a, Object b) { code.add(o + " "+ a +", "+ b); }
void put(Op o, Object a, int b) { put(o,a,String.valueOf(b)); }
void put(Op o, Object a, Item b) { put(o,a,(Object)b); freeReg(b); }
void put(Op o, Object a) { code.add(o + " "+ a); }
void put(Op o, int a, int b) { code.add(o + " "+ a +", "+ b); }
void put(Op o) { code.add(o.toString()); }
void put(Lab l) { code.add(l+":"); }
void putCond(Lab l, Op o, Item a, Item b) {
    put(Op.CMP,a,b); put(o,l); freeReg(a); freeReg(b);
}
// ----- //
Item load(Item y) {
    Item x = null;
    if (y.mode == REG) x = y;
    if (y.mode == REGREL || y.mode == CONST || y.mode == ABS) {
        x = new Item(REG,y.type);
        x.r = getReg();
        put(MOV,x,y);
        if (y.mode == REGREL) freeReg(y.r);
    } else
        err.printf("E: %s: load() wrong mode.\n",clazz);
    return x;
}
// ----- //
void fold(Op op, Item x, Item y) {
    int a = x.v.ival, b = y.v.ival;
    switch (op) {
        case ADD: x.v.ival = a + b; break;
        case SUB: x.v.ival = a - b; break;
        case IDIV: x.v.ival = a / b; break; // division by zero not handled
        case IMUL: x.v.ival = a * b; break;
        case MOD: x.v.ival = a % b; break;
        default: err.printf("E:%s: Unknown Op '%s'!\n",clazz,op);
    }
}
// ----- //
Reg getReg() {
    for (Reg r : Reg.values())
        if (!regs.contains(r) && (r.ordinal() < ESP.ordinal())) {
            regs.add(r);
            return r;
        }
}

```

```

    }
    err.printf("E:%s: Can not allocate register.\n",clazz);
    return null;
}
boolean getReg(Reg r) {
    if (!regs.contains(r)) {
        regs.add(r);
        return true;
    }
    err.printf("E:%s: Can not allocate register.\n",clazz);
    return false;
}
boolean isFree(Reg r) { return !regs.contains(r); }
void freeAllRegs() { regs.clear(); }
void freeReg(Reg r) { regs.remove(r); }
void freeReg(Item x) { if (x.mode == REG) freeReg(x.r); }
// ----- //
// PSH slots for parameters on stack
void pushParams(int size) { put(ADD,SP,-size); }
// PSH slot for return value on stack
void pushRet() { put(ADD,SP,-SIZ); }
// POP return value from stack into register
Item popRet() {
    Item x = new Item(); x.mode = Mode.REG;
    x.r = getReg();
    put(POP,x.r); // return value is below ESP
    return x;
}
// discard return value below SP
void discardRet() { put(SUB,SP,SIZ); }
// Store return value in allocated slot
void storeRet(Item x,int psize,int rsize) {
    put(MOV,((2*SIZ)+psize+rsize) + "[ "+FP+" ]",x); freeReg(x);
}
// ----- //
boolean spilled() { return !spill.empty(); }

void pshReg(Reg r) { spill.push(r); put(PUSH,r); freeReg(r); }

Reg popReg() {
    Reg r = spill.pop(); getReg(r); put(POP,r); return r;
}

// EDX contains remainder of division
Item putMod(Item x,Item y) { return putDiv(x,y,EDX); }

// EAX contains quotient of division
Item putDiv(Item x,Item y) { return putDiv(x,y,EAX); }

```

```

// Division: (EDX:EAX)/opd ... opd is register or memory location
//      - EAX holds quotient
//      - EDX holds remainder
private Item putDiv(Item x, Item y, Reg R) {
    if (y.mode != REG) { // IDIV m: divisor is in memory
        if (x.r != EAX) {
            if (!isFree(EAX)) pshReg(EAX); // spill EAX
            put(MOV, EAX, x.r);
        }
        if (x.r != EDX && !isFree(EDX)) pshReg(EDX); // spill EDX
        put(CWD); // sign extend EAX to EDX:EAX
        put(IDIV, y); // EAX = IDIV m
        if (x.r != R) put(MOV, x.r, R);
        while (spilled()) popReg(); // restore EAX, EDX
    } else { // IDIV r: divisor in register is put on top of stack
        // and we perform an IDIV m. While this is suboptimal
        // it simplifies preservation of registers.
        if (x.r != EAX && y.r != EAX && !isFree(EAX))
            pshReg(EAX); // spill EAX
        if (x.r != EDX && y.r != EDX && !isFree(EDX))
            pshReg(EDX); // spill EDX
        put(PUSH, y.r); // store y on top of stack
        freeReg(y.r);
        if (x.r != EAX) put(MOV, EAX, x.r);
        put(CWD); // sign extend EAX to EDX:EAX
        put(IDIV, SIZ + "[" + EBP + "]); // y is on top of stack
        put(ADD, EBP, SIZ); // remove y from stack
        if (x.r != R) put(MOV, x.r, R);
        while (spilled()) popReg(); // restore EAX, EDX
    }
    return x;
}
}

```

B.2.3 CISC Assembly Output

The goal was to output CISC assembly code for the introductory example in Listing B.1, and the automatically generated CISC code generator outputs the following target code for it:

```

$Ggt_INT_INT:      ; ----- Ggt(Int x, Int y): -----
ENTER 8,0          ; PROLOGUE: PSH EBP;EBP:=ESP;ESP:=ESP-size(stack frame)
MOV EAX,16[EBP]    ; load param y
CMP EAX,12[EBP]    ; compare x > y where y
JLE $L1            ; if (x <= y) then JMP $FLS
MOV EAX,12[EBP]    ; load param x
MOV -4[EBP],EAX    ; t := x
MOV EAX,16[EBP]    ; load param y
MOV 12[EBP],EAX    ; x := y
MOV EAX,-4[EBP]    ; load local variable t
MOV 16[EBP],EAX    ; y := t
$L1:               ; $FLS
MOV EAX,12[EBP]    ; load param x
CWD                ; double size of operand in register EAX to EDX:EAX
IDIV 16[EBP]       ; x / y: EAX contains quotient, EDX contains remainder
MOV EAX,EDX        ; store remainder in register EAX
MOV 0[EBP],EAX     ; rest := remainder of x / y
$L3:               ; $WHILE
CMP 0[EBP],0       ; compare: rest > 0
JLE $L2            ; if (rest <= 0) then JMP $END
MOV EAX,16[EBP]    ; load param y
MOV 12[EBP],EAX    ; x := y
MOV EAX,0[EBP]     ; load local variable rest
MOV 16[EBP],EAX    ; y := rest
MOV EAX,12[EBP]    ; load param x
CWD                ; double size of operand in register EAX to EDX:EAX
IDIV 16[EBP]       ; x / y
MOV EAX,EDX        ; store remainder in register EAX
MOV 0[EBP],EAX     ; rest := remainder of x / y
JMP $L3            ; JMP $WHILE
$L2:               ; $END
MOV 20[EBP],16[EBP] ; store return value into return slot
LEAVE              ; EPILOGUE: ESP:=EBP; EBP:=POP
RET 8              ; EPILOGUE: EIP:=POP; ESP:=ESP-size(params)
$main:             ; ----- ENTRY POINT -----
ADD ESP,-4         ; PSH return slot for Ggt() on stack
ADD ESP,-8         ; PSH param slots for Ggt() on stack
MOV 4[ESP],24       ; PSH parameter 24 in 1. param slot
MOV 8[ESP],18       ; PSH parameter 18 in 2. param slot
CALL $Ggt_INT_INT  ; PUSH EIP; EIP=Adr($Ggt_INT_INT)
POP EAX             ; POP return value off stack

```

```
MOV DS:0,EAX      ; r := Ggt(24,18)
ADD ESP,-4        ; PSH param slot for PutInt() on stack
MOV EAX,DS:0      ; load local variable r
MOV 4[ESP],EAX    ; PUH parameter r in 1. param slot
CALL $PUT_INT     ; PSH EIP; EIP:=Adr($PUT_INT)
```

Note that the procedure `PUT_INT` on line 14 in Listing B.1 is a library procedure and outputs an integral value to the console. Its implementation is not presented in this example.

Automatically Generated Code

The following code snippets are generated by **HBURG** from the CISC specification defined in Appendix B.2.2.

Operators (terminals) defined in a specification are grouped in sets according to their arity. For example the `EnumSet arity2Set` denotes all operators with two operands (child nodes), and the `EnumSet linkSet` denotes all operators that may have link references (e.g. “`ADD (reg,reg) [stmt]`”). Non terminals (operators) and rewrite rules are encoded as enumerations prefixed with `NT_` and `R_`.

[illegible]

```

static EnumSet linkSet = EnumSet.of(ADD,ASGN,CNST,DIV,EIF,
                                     ENTER,FUN,FUNP,IF,IFE,
                                     MOD,MUL,PROC,PROCP,
                                     RET,SUB,VAR,WHILE);

// tile(): tile IR by traversing it bottom-up
public static void tile(sl.code.cisc.Node _n) {
    assert (_n != null) : "ERROR: tile() - node is null.";
    if (arity0Set.contains(_n.kind())) {
        label_0(_n); // label leaf nodes
    } else if (arity1Set.contains(_n.kind())) {
        tile(_n.child1());
        label_1(_n); // label nodes with one child
    } else if (arity2Set.contains(_n.kind())) {
        tile(_n.child1());
        tile(_n.child2());
        label_2(_n); // label nodes with two children
    } else {
        throw new AssertionError("ERROR: tile() - Encountered "+
                                "undefined node '"+_n.kind()+"'");
    }
    if (linkSet.contains(_n.kind())) {
        sl.code.cisc.Node link = _n.link();
        if (link != null) tile(link); // tile linked nodes
    }
} // END METHOD tile()

// label(): record rule number and non terminal derived
//           by node and handle triggered chain rules
private static void label(sl.code.cisc.Node _n,
                          sl.code.cisc.Nt _nt,
                          int _c,
                          sl.code.cisc.Rule _r) {
    if (_c < _n.cost(_nt)) {
        _n.put(_nt, new sl.code.cisc.Entry(_c, _r));
        switch (_nt) {
            case NT{EIF}: {
                label(_n, NT{EIFSEQ}, _c, R{EIFSEQ{EIF}_0); break;
            }
            case NT{FUN}: {
                label(_n, NT{REG}, _c, R{REG}{FUN}_13); break;
            }
            case NT{IMM}: {
                label(_n, NT{PARAM}, _c+1, R{PARAM}{IMM}_1);
                label(_n, NT{REG}, _c+1, R{REG}{IMM}_15);
                break;
            }
            case NT{MEM}: {
                label(_n, NT{REG}, _c+1, R{REG}{MEM}_14); break;
            }
            case NT{PARAM}: {
                label(_n, NT{PARAMSEQ}, _c, R{PARAMSEQ}{PARAM}_0); break;
            }
            case Nt_procdef: {

```

```

        label(_n, NT_PROCDEFSEQ, _c, R_PROCDEFSEQ_PROCDEF_0); break;
    } case NT_REG: {
        label(_n, NT_PARAM, _c+1, R_PARAM_REG_0); break;
    } case NT_STMT: {
        label(_n, NT_STMTSEQ, _c, R_STMTSEQ_STMT_0); break;
    }
}
}
} // END METHOD label()
...
// label_0(): label leaf nodes
private static void label_0(sl.code.cisc.Node _n) {
    int _c;
    switch (_n.kind()) {
        case CNST: {
            _c = 0; label(_n, NT_IMM, _c, R_IMM_CNST_5); break;
        } case FUN: {
            _c = 3; label(_n, NT_STMT, _c, R_STMT_FUN_3);
            _c = 3; label(_n, NT_FUN, _c, R_FUN_FUN_1);
            break;
        } case PROC: {
            _c = 1; label(_n, NT_STMT, _c, R_STMT_PROC_1); break;
        } case VAR: {
            _c = 0; label(_n, NT_MEM, _c, R_MEM_VAR_4); break;
        } default: {
            throw new AssertionError("ERROR - label_0(): Unhandled "+
                                    "Node kind: " + _n.kind());
        }
    }
} // END SWITCH
} // END METHOD label_0()
...

```

C.2 Semantic Action Execution

Semantic actions are executed after the IR tree has been labeled. Each `eval_` method encodes semantic actions defined in tree patterns that derive a particular non terminal.

```

...
// eval_fun(): out parameter has type Item
private static Item eval_fun(sl.code.cisc.Node _n) {
    sl.code.cisc.Rule _r = _n.rule(NT_FUN); // retrieve stored rule
    Item x; // out parameter

```

```

// (.
    x = null;
// .)
switch (_r) {
    case R_FUN_FUNP_0: {
        sl.code.cisc.Node p = _n;
        // (.
            Obj o = ((Node)p).obj;
            c.pushRet();
            c.pushParams(o.parSize);
        // .)
        eval_paramseq(_n.child1(), t.params(o));
        // (.
            c.put(Op.CALL, Lab.funLab(o.name, o.next));
            x = c.popRet();
        // .)
        break;
    } case R_FUN_FUN_1: {
        sl.code.cisc.Node p = _n;
        // (.
            Obj o = ((Node)p).obj;
            c.pushRet();
            c.put(Op.CALL, Lab.funLab(o.name, o.next));
            x = c.popRet();
        // .)
        break;
    } default: {
        throw new AssertionError("ERROR: Unhanded semantic "+
                                " rule - "+ _r +".");
    }
}
return x; // return out parameter
} // END METHOD eval_fun()
...
// eval_paramseq(): in parameter has type List<Obj>
private static void eval_paramseq(sl.code.cisc.Node _n, List<Obj> lst){
    sl.code.cisc.Rule _r = _n.rule(NT_PARAMSEQ);
    switch (_r) {
        case R_PARAMSEQ_PARAM_0: {
            eval_param(_n, lst.get(0));
            // (.
                lst.remove(0);
            // .)
            if (_n.link() != null) {
                eval_paramseq(_n.link(), lst);
            }
            break;
        } default: {
            throw new AssertionError("ERROR: Unhanded semantic "+

```

```
                                " rule - "+ _r +".");  
                                }  
                                }  
} // END METHOD eval_paramseq()  
...  

```

List of Figures

2.1	Compiler Infrastructure	4
2.2	Transforming Input Language to Intermediate Representation	5
2.3	Transforming Intermediate Representation to Target Code	7
4.1	Automatic Generation of a Compiler Back End	18
4.2	Linked Tree-Structured Intermediate Representation	19
4.3	Example IR with Rewrite Rules	21
4.4	Potential Rewrite Rule Matches	22
4.5	Bottom-up Optimal Instruction Selection	23
4.6	Data Structure for Intermediate Representation Nodes	27
5.1	HBURG Compiler Infrastructure	32
5.2	HBURG Modules Overview	33
5.3	HBURG Abstract Data Types	34
5.4	Mapping a Tree Pattern onto Intermediate Representation	35
5.5	Calculating Optimal Rewrite Rule Cover	38
5.6	Chain Rule Closure Calculation	39
5.7	Node Data Structure for Optimal Instruction Selection	39
5.8	Input and Output of HBURG	43
5.9	Node Interface	44
B.1	IR for Expressions and Assignments	57
B.2	IR for Control Flow Statements	58
B.3	IR for Procedure Definition and Application	58
B.4	IR for Greatest Common Divisor Example	59
B.5	RISC Runtime Stack Layout	61
B.6	CISC Runtime Stack Layout	74

List of Listings

3.1	CISC Code Generator Specification Example	10
4.1	Code Generator Description Language Structure	25
4.2	Referring to Bindings and Declarations in Semantic Actions	28
4.3	Attribute Grammar Example	29
5.1	Data Type Definitions for HBURG Intermediate Representation	34
5.2	Rewrite Rule Example	40
5.3	Code Generator Implementation of Rewrite Rules	41
B.1	Program calculating Greatest Common Divisor	55
B.2	IR Binary Node Data Type	56
B.3	IR Node Kinds	56
B.4	Symbol Table Data Type	56
B.5	RISC Register Set	60
B.6	RISC Instruction Set	60
B.7	RISC Operand Data Type	61
B.8	RISC Code Generator Specification	62
B.9	Node Interface Implementation	66
B.10	RISC Code Manipulation Class	67
B.11	RISC Assembly Output	70
B.12	CISC Register Set	72
B.13	CISC Instruction Set	72
B.14	CISC Operand Data Type	73
B.15	CISC Code Generator Specification	74
B.16	CISC Code Manipulation Class	79
B.17	CISC Assembly Output	83
C.1	Generated Code: IR Tree Tiling	85
C.2	Generated Code: Semantic Action Execution	87

Bibliography

- [1] AHO, A. V., GANAPATHI, M., AND TJANG, S. W. K. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems* 11, 4 (1989), 491–516.
 - [2] CHRISTOPHER W. FRASER, ROBERT R. HENRY, T. A. P. BURG - fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices* 27, 4 (1989), 68–76.
 - [3] COOPER, K. D., AND TORCZON, L. *Engineering a Compiler*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.
 - [4] CRELIER, R. OP2: A portable Oberon compiler. Technical Report 125, Informatics Department, Institute for Computer Systems, ETH Zürich, Switzerland, 1990.
 - [5] EMMELMANN, H., SCHRÖER, F.-W., AND LANDWEHR, R. BEG - a generator for efficient back ends. In *Proceedings of the SIGPLAN symposium on Interpreters and interpretive techniques* (1989), vol. 24, pp. 227–237.
 - [6] FRASER, C. W., AND HANSON, D. R. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems* 1, 3 (1992), 213–226.
 - [7] FRASER, C. W., AND PROEBSTING, T. A. Finite-state code generation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (1999), pp. 270–280.
 - [8] GANAPATHI, M. *Code Generation and Optimization using Attribute Grammars*. PhD thesis, University of Wisconsin, Madison, 1980.
 - [9] GANAPATHI, M., AND FISCHER, C. N. Description-driven code generation using attribute grammars. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1982), ACM Press, pp. 108–119.
-

-
- [10] GLANVILLE, R. S. *A machine independent algorithm for code generation and its use in retargetable compilers*. PhD thesis, University of California, Berkeley, 1977.
 - [11] GLANVILLE, R. S., AND GRAHAM, S. L. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1978), ACM Press, pp. 231–254.
 - [12] HORSPOOL, N. R., AND SCHEUNEMANN, A. Automating the selection of code templates. *Software - Practice and Experience* 15, 5 (1985), 503–514.
 - [13] HUDAK, P., AND FASEL, J. H. A gentle introduction to haskell. *SIGPLAN Not.* 27, 5 (1992), 1–52.
 - [14] JONES, S. L. P., AND SANTOS, A. L. M. A transformation-based optimiser for Haskell. *Sci. Comput. Program.* 32, 1-3 (1998), 3–47.
 - [15] JONES, S. P. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
 - [16] KANG, K. W. A study on generating an efficient bottom-up tree rewrite machine for JBang. In *Computational Science and Its Applications - ICCSA 2004* (2004), vol. 3043 of *Lecture Notes in Computer Science*, Springer, pp. 65–72.
 - [17] KNUTH, D. E. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.
 - [18] KNUTH, D. E. The genesis of attribute grammars. In *Proceedings of the international conference on Attribute grammars and their applications* (1990), pp. 76–90.
 - [19] MÖSSENBOCK, H. Coco/R: A generator for fast compiler front ends. Technical Report 127, Informatics Department, Institute for Computer Systems, ETH Zürich, Switzerland, 1990.
 - [20] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1997.
 - [21] PELEGRI-LLOPART, E. *Rewrite Systems, Pattern Matching, and Code Generation*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1988.
-

-
- [22] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, 2002.
 - [23] PROEBSTING, T. A. Simple and efficient BURS table generation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (1992), pp. 331–340.
 - [24] PROEBSTING, T. A. BURS automata generation. *ACM Transactions on Programming Languages and Systems* 17, 3 (1995), 461–486.
 - [25] PROEBSTING, T. A., AND WHALEY, B. R. One-pass, optimal tree parsing - with or without trees. In *Proceedings of the 6th International Conference on Compiler Construction* (1996), pp. 294–308.
 - [26] SHAO, Z., AND APPEL, A. W. A type-based compiler for standard ML. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (New York, NY, USA, 1995), ACM Press, pp. 116–129.
 - [27] TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. TIL: a type-directed, optimizing compiler for ML. *SIGPLAN Not.* 39, 4 (2004), 554–567.
 - [28] WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM* 20, 11 (1977), 822–823.
 - [29] WIRTH, N. *Compiler Construction*. Addison-Wesley, 1996.
-