



**UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO**

Object- and Pattern-Oriented Compiler Construction in C++

A hands-on approach to modular compiler construction
using GNU flex, Berkeley yacc and standard C++

David Martins de Matos

January 2006

Foreword

Acknowledgements

Lisboa, May 4, 2007
David Martins de Matos

Contents

I	Introduction	1
1	Introduction	3
1.1	Introduction	3
1.2	Who Should Read This Document?	3
1.3	Organization	4
2	Using C++ and the CDK Library	5
2.1	Introduction	5
2.2	Regarding C++	5
2.3	The CDK Library	6
2.3.1	The abstract compiler factory	7
2.3.2	The abstract scanner class	9
2.3.3	The abstract compiler class	10
2.3.4	The parsing function	11
2.3.5	The node set	12
2.3.6	The abstract evaluator	13
2.3.7	The abstract semantic processor	15
2.3.8	The code generators	15
2.3.9	Putting it all together: the main function	16
2.4	Summary	17
II	Lexical Analysis	19
3	Theoretical Aspects of Lexical Analysis	21
3.1	What is Lexical Analysis?	21

3.1.1	Language	21
3.1.2	Regular Language	21
3.1.3	Regular Expressions	21
3.2	Finite State Acceptors	21
3.2.1	Building the NFA	22
3.2.2	Determinization: Building the DFA	22
3.2.3	Compacting the DFA	24
3.3	Analysing a Input String	26
3.4	Building Lexical Analysers	27
3.4.1	The construction process	27
3.4.1.1	The NFA	27
3.4.1.2	The DFA and the minimized DFA	27
3.4.2	The Analysis Process and Backtracking	29
3.5	Summary	29
4	The GNU flex Lexical Analyser	31
4.1	Introduction	31
4.1.1	The lex family of lexical analysers	31
4.2	The GNU flex analyser	31
4.2.1	Syntax of a flex analyser definition	31
4.2.2	GNU flex and C++	31
4.2.3	The FlexLexer class	31
4.2.4	Extending the base class	31
4.3	Summary	31
5	Lexical Analysis Case	33
5.1	Introduction	33
5.2	Identifying the Language	33
5.2.1	Coding strategies	33
5.2.2	Actual analyser definition	33
5.3	Summary	33

III	Syntactic Analysis	35
6	Theoretical Aspects of Syntax	37
6.1	Introduction	37
6.2	Grammars	37
6.2.1	Formal definition	37
6.2.2	Example grammar	37
6.2.3	FIRST and FOLLOWS	38
6.3	LR Parsers	38
6.3.1	LR(0) items and the parser automaton	39
6.3.1.1	Augmented grammars	39
6.3.1.2	The closure function	41
6.3.1.3	The “goto” function	41
6.3.1.4	The parser’s DFA	42
6.3.2	Parse tables	43
6.3.3	LR(0) parsers	45
6.3.4	SLR(1) parsers	45
6.3.5	Handling conflicts	45
6.4	LALR(1) Parsers	45
6.4.1	LR(1) items	45
6.4.2	Building the parse table	45
6.4.3	Handling conflicts	45
6.4.4	How do parsers parse?	45
6.5	Compressing parse tables	45
6.6	Summary	46
7	Using Berkeley YACC	47
7.1	Introduction	47
7.1.1	AT&T YACC	47
7.1.2	Berkeley YACC	48
7.1.3	GNU Bison	48
7.1.4	LALR(1) parser generator tools and C++	48
7.2	Syntax of a Grammar Definition	48

7.2.1	The first part: definitions	49
7.2.1.1	External definitions and code blocks	49
7.2.1.2	Internal definitions	49
7.2.2	The second part: rules	53
7.2.2.1	Shifts and reduces	53
7.2.2.2	Structure of a rule	53
7.2.2.3	The grammar's start symbol	55
7.2.3	The third part: code	55
7.3	Handling Conflicts	56
7.4	Pitfalls	56
7.5	Summary	57
8	Syntactic Analysis Case	59
8.1	Introduction	59
8.1.1	Chapter structure	59
8.2	Actual grammar definition	59
8.2.1	Interpreting human definitions	59
8.2.2	Avoiding common pitfalls	59
8.3	Writing the Berkeley yacc file	59
8.3.1	Selectiong the scanner object	60
8.3.2	Grammar item types	60
8.3.3	Grammar items	60
8.3.4	The rules	60
8.4	Building the Syntax Tree	61
8.5	Summary	61
IV	Semantic Analysis	63
9	The Syntax-Semantics Interface	65
9.1	Introduction	65
9.1.1	The structure of the Visitor design pattern	65
9.1.2	Considerations and nomenclature	65
9.2	Tree Processing Context	65

9.3	Visitors and Trees	67
9.3.1	Basic interface	67
9.3.2	Processing interface	67
9.4	Summary	67
10	Semantic Analysis and Code Generation	69
10.1	Introduction	69
10.2	Code Generation	69
10.3	Summary	69
11	Semantic Analysis Case	71
11.1	Introduction	71
11.2	Summary	71
V	Appendices	73
A	The CDK Library	75
A.1	The Symbol Table	75
A.2	The Node Hierarchy	75
A.2.1	Interface	75
A.2.2	Interface	75
A.2.3	Interface	75
A.3	The Semantic Processors	76
A.3.1	Cápsula	76
A.3.2	Cápsula	76
A.4	The Driver Code	76
A.4.1	Construtor	76
B	Postfix Code Generator	77
B.1	Introduction	77
B.2	The Interface	78
B.2.1	Introduction	78
B.2.2	Output stream	78
B.2.3	Simple instructions	78

B.2.4	Arithmetic instructions	79
B.2.5	Rotation and shift instructions	80
B.2.6	Logical instructions	80
B.2.7	Integer comparison instructions	80
B.2.8	Other comparison instructions	81
B.2.9	Type conversion instructions	81
B.2.10	Function definition instructions	82
B.2.10.1	Function definitions	82
B.2.10.2	Function calls	83
B.2.11	Addressing instructions	83
B.2.11.1	Absolute and relative addressing	83
B.2.11.2	Quick opcodes for addressing	84
B.2.11.3	Load instructions	84
B.2.11.4	Store instructions	85
B.2.12	Segments, values, and labels	85
B.2.12.1	Segments	85
B.2.12.2	Values	85
B.2.12.3	Labels	86
B.2.12.4	Types of global names	87
B.2.13	Jump instructions	87
B.2.13.1	Conditional jump instructions	87
B.2.13.2	Other jump instructions	88
B.2.14	Other instructions	88
B.3	Implementations	88
B.3.1	NASM code generator	89
B.3.2	Debug-only “code” generator	89
B.3.3	Developing new generators	89
B.4	Summary	89
C	The Runtime Library	91
C.1	Introduction	91
C.2	Support Functions	91
C.3	Summary	91

List of Figures

2.1	CDK library's class diagram	6
2.2	CDK library's main function sequence diagram	7
2.3	Abstract compiler factory base class.	8
2.4	Concrete compiler factory for the Compact compiler	9
2.5	Concrete compiler factory for the Compact compiler	9
2.6	Compact's lexical analyser header	10
2.7	Abstract CDK compiler class	12
2.8	Partial syntax specification for the Compact compiler	13
2.9	CDK node hierarchy class diagram	14
2.10	Partial specification of the abstract semantic processor	15
2.11	CDK library's sequence diagram for syntax evaluation	16
2.12	CDK library's main function (simplified code)	17
3.1	Thompson's algorithm example for $a(a b) * c$	22
3.2	Determinization table example for $a(a b) * c$	25
3.3	DFA graph for $a(a b) * c$: full configuration and simplified view (right).	25
3.4	Minimal DFA graph for $a(a b) * c$: original DFA, minimized DFA, and minimization tree.	26
3.5	NFA for a lexical analyser for $G = \{a * b, a b*, a*\}$	28
3.6	Determinization table example for the lexical analyser	28
3.7	DFA for a lexical analyser for $G = \{a * b, a b*, a*\}$: original (top left), minimized (bottom left), and minimization tree (right). Note that states 2 and 4 cannot be merged since they recognize different tokens.	29
3.8	Processing an input string and token identification	29
6.1	LR parser model.	38

6.2	Graphical representation of the DFA showing each state's item set. Reduces are possible in states $I_1, I_2, I_3, I_5, I_9, I_{10}$, and I_{11} : it will depend on the actual parser whether reduces actually occur.	44
6.3	Example of a parser table. Note the column for the end-of-phrase symbol.	44
6.4	exemplo de acções L unitárias	45
6.5	exemplo de acções L quase unitárias	46
6.6	exemplo de conflitos e compressão	46
7.1	General structure of a grammar definition file for a YACC-like tool.	48
7.2	Various code blocks like the one shown here may be defined in the definitions part of a grammar file: they are copied verbatim to the output file in the order they appear.	50
7.3	The <code>%union</code> directive defines types for both terminal and non-terminal symbols.	50
7.4	Symbol definitions for terminals (<code>%token</code>) and non-terminals (<code>%type</code>).	51
7.5	YACC-generated parser C/C++ header file: note especially the special type for symbol values, <code>YYSTYPE</code> , and the automatic declaration of the global variable <code>yyval</code> . The code shown in the figure corresponds to actual YACC output.	52
7.6	Precedence and associativity in ambiguous grammars (see also §7.2.2).	52
7.7	Examples of rules and corresponding semantic blocks. The first part of the figure shows a collection of statements; the second part shows an example of recursive definition of a rule. Note that, contrary to what happens in LL(1) grammars, there is no problem with left recursion in LALR(1) parsers.	54
7.8	Example of a program accepted by the grammar defined in figure 7.7.	54
7.9	Start symbols and grammars. Assuming the two tokens 'a' and 'b', the same rules recognize different syntactic constructions depending on the selection of the top symbol. Note that the non-terminal symbols a and b are different from the tokens (terminals) 'a' and 'b'.	55
9.1	Macro structure of the main function. Note especially the syntax and semantic processing phases (respectively, <code>yyparse</code> and <code>evaluate</code>).	66

List of Tables

3.1	Primitive constructions in Thompson's algorithm.	23
-----	--	----

I Introduction

1 Introduction

1.1 *Introduction*

(Aho et al., 1986)

A compiler is a program that takes as input a program written in the source language and translates it into another the target language, typically (although not necessarily) one understandable by a machine.

Various types: translator (between two high-level languages), compiler (from high- to low-level language), decompiler (from low- to high-level language), rewriter (within the same language).

Modern compilers usually take source code and produce object code, in a form usable by other programs, such as a linker or a virtual machine. Examples:

- C/C++ are typically compiled into object code for later linking (.o files);
- Java is typically compiled (Sun compiler) into binary class files (.class), used by the virtual machine. The GCC Java compiler can, besides the class files, also produce object files (like for C/C++).

Properties and Problems?

Compiler or interpreter: although both types of language analysis tools share some properties, they differ in how they handle the analyzed code. Compilers will simply produce an equivalent version of the original language in a target language; interpreters will, like compilers, translate the input language, not into a target version, but rather directly into the execution of the actions described in the source language.

How to build one?

C++? Why? Other approaches using C++ as a better C but without advanced design methodologies.

1.2 *Who Should Read This Document?*

This document is for those who seek to use the flex and yacc tools beyond the C programming language and apply object-oriented (OO) programming techniques to compiler construction. In the following text, the C++ programming

language is used, but the rationale is valid for other OO languages as well. Note, however, that C++ works with C tools almost without change, something that may not be true of other languages (although there may exist tools similar to flex and yacc that support them).

The use of C++ is not motivated only by a “better” C, a claim some would deny. Rather, it is motivated by the advantages that can be gained from bringing OO design principles in contact with compiler construction problems. In this regard, C++ is a more obvious choice than C¹, and is not so far removed that traditional compiler development techniques and tools have to be abandoned.

Going beyond basic OO principles into the world of design patterns is just a small step, but one that contributes much of the overall gains in this change: indeed, effective use of a few choice design patterns – especially, but not necessarily limited to, the *composite* and *visitor* design patterns – contributes to a much more robust compiler and a much easier development process.

The document assumes basic knowledge of object-oriented design as well as abstract data type definition. Knowledge about design patterns is desirable, but not necessary: the patterns used in the text will be briefly presented. Nevertheless, useful insights can be gained from reading a patterns book such as [citeNbook:gof](#).

1.3 Organization

This text parallels both the structure and development process of a compiler. Thus, the first part deals with lexical analysis, or by a different name, with the morphological analysis of the language being recognized. The second part presents syntax analysis in general and LALR(1) parsers in particular. The fourth part is dedicated to semantic analysis and the deep structure of a program as represented by a linguistic structure. Semantic processing also covers code generation, translation, interpretation, as well as the other processes that use similar development processes.

Regarding the appendices, they present the code used throughout the document. In particular, detailed descriptions of each hierarchy are presented. Also presented is the structure of the final compiler, in terms of code: both the code developed by the compiler developer, and the support code for compiler development and final program execution.

¹Even though one could say that if you have mastered OO design, then you can do it in almost any language, C++ continues to be a better choice than C, simply because it offers direct support for those principles and a strict type system.

Using C++ and the CDK Library

2.1 Introduction

The Compiler Development Kit (CDK) is a library for building compilers by allowing the combination of advanced OO programming techniques and traditional compiler building tools such as Lex () and YACC (). The resulting compiler is clearly structured and easy to maintain.

The CDK version described here uses the GNU Flex () lexical analyser and the Berkeley YACC () LALR(1) parser generator. In this chapter the description will focus on OO aspects and not in compiler construction details. These will be covered in detail in later chapters.

The reader is encouraged to review object-oriented and design pattern concepts, especially, but without limitation, the ones used by the CDK and the compilers based on it: Abstract Factory, Composite, Strategy, Visitor (Gamma et al., 1995, respectively, pp. 87, 163, 315, 331).

2.2 Regarding C++

Using C++ is not only a way of ensuring a “better C”, but also a way of being able to use OO architecture principles in a native environment (the same principles could have been applied to C development, at the cost of increased development difficulties). Thus, we are not interested only in taking a C++ compiler, our old C code and “hope for the best”. Rather, using C++ is intended to impact every step of compiler development, from the organization of the compiler as a whole to the makeup of each component.

Using C++ is not only a decision of what language to use to write the code: it is also a matter of who or what writes the compiler code. If for a human programmer using C++ is just a matter of competence, tools that generate some of the compiler’s code must be chosen carefully so that the code they generate works as expected. Some of the most common compiler development support tools already support C++ natively. This is the case of the GNU Flex lexical analyser or the GNU Bison () parser generator. Other tools, such as Berkeley YACC () (BYACC) support only C. In the former case, the generated code and the objects it supports have only to be integrated into the architecture; in the latter case, further adaptation may be needed, either by the programmer or through specialized wrappers. BYACC-generated parsers, in particular, as will be seen, although they are C code, are simple to adapt to C++.

2.3 The CDK Library

The CDK library is intended to provide the compiler developer with a simple, yet flexible, framework for building a compiler. To that end, the library tries to guide every step of the compilation tasks. Only the abstract parts need to be defined for each language.

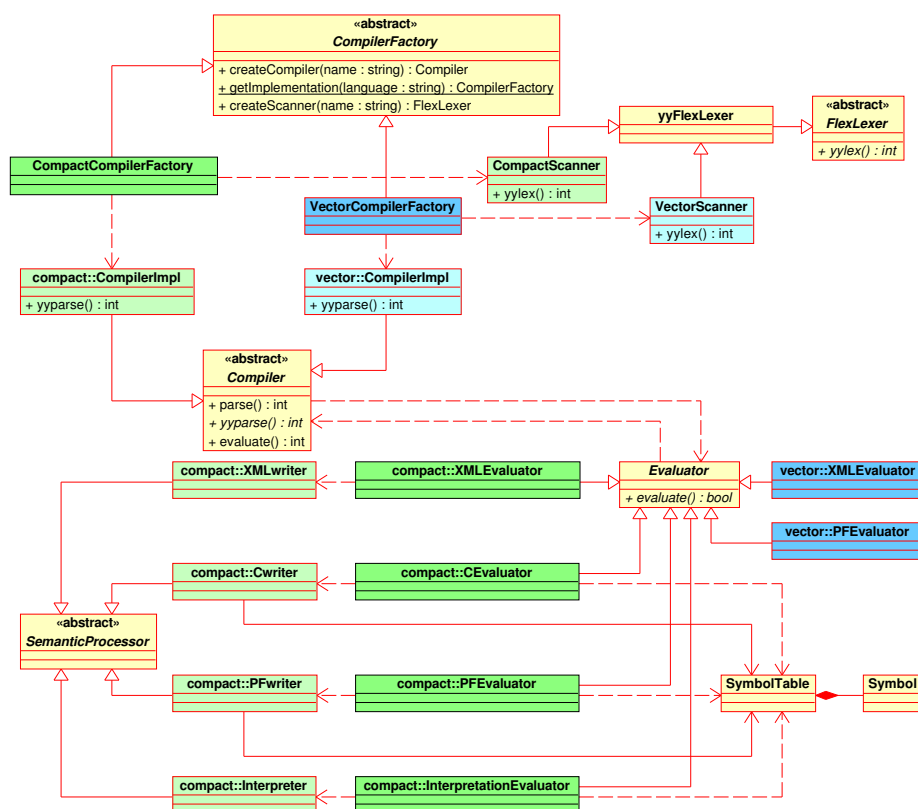


Figure 2.1: CDK library's class diagram: Compact and Vector are two examples of concrete languages.

The library defines the following items, each of which will be the subject of one of the sections below.

- The abstract compiler factory: transforming language names into specialized objects (§2.3.1);
- The abstract compiler class: performing abstract compilation tasks (§2.3.3);
- The abstract scanner: lexical analysis (§2.3.2);
- The parsing function: syntactic analysis and tree building (§2.3.4);
- The node set: syntax tree representation (§2.3.5);

- The abstract evaluator: evaluating semantics from the syntax tree (§2.3.6);
- The abstract semantic processor: syntax tree node visiting (§2.3.7);
- The code generators: production of final code (§2.3.8);
- The main function (§2.3.9);

These topics are arranged more or less in the order they are needed to produce a full-fledged compiler: everything starts in the main function with the creation of the compiler for the work language and helper objects (e.g., the scanner); then nodes are created and passed to a specific evaluator that, in turn, creates the appropriate visitors to handle tasks such as code generation or interpretation. Figure 2.2 presents the top-level interactions (main function – see §2.3.9).

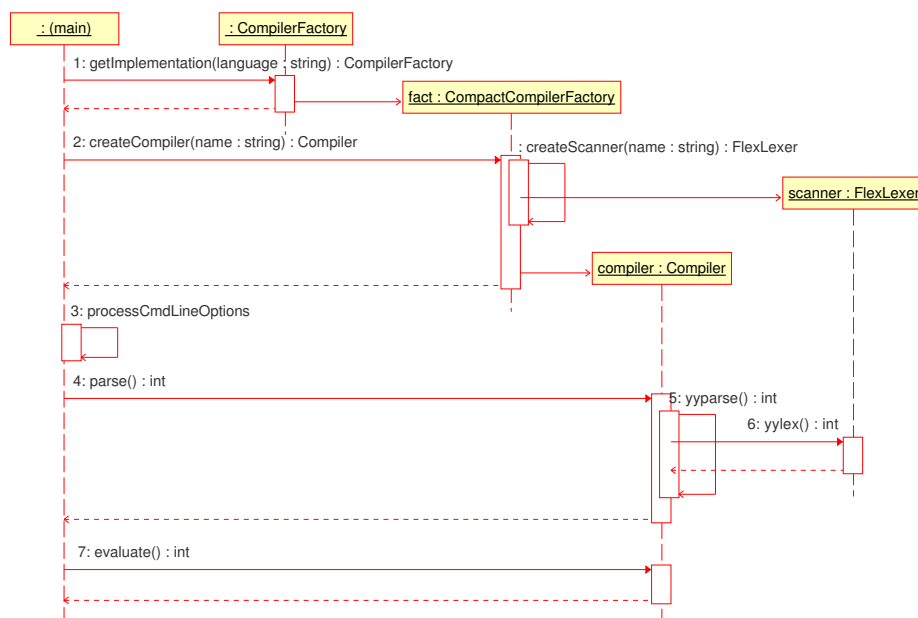


Figure 2.2: CDK library’s main function sequence diagram: details from `yyparse`, `yylex`, and `evaluate` have been omitted in this diagram.

2.3.1 The abstract compiler factory

As shown in figure 2.2, the abstract factory is responsible for creating both the scanner and the compiler itself. The parser is not an object itself, but rather a compiler method. This arrangement is due to the initial choice of support tools: the parser generator, BYACC, is (currently) only capable of creating a C function (`yyparse`). It is easier to transform this function into a method than creating a class just for encapsulating it. Note that these choices (BYACC and method vs. class) may change and so it may happen with all the classes involved (both compiler and factory).

The process of creating a compiler is a simple one: the factory method for creating the compiler object first creates a scanner that is, later, passed as an argument of the compiler's constructor. Client code may rely on the factory for creating the appropriate scanner and needs only to ask for the creation of a compiler.

Figure 2.3 presents the superclass definition. This is the main factory abstract class: it provides methods for creating the lexical analyser and the compiler itself. Instances of concrete subclasses will be obtained by the `main` function to provide instances of the scanner and compiler objects for a concrete language. The factory provides a registry (`_factories`) of all the known instances of its subclasses. Each instance is automatically registered by the corresponding constructor. Also, notice that scanner creation is not available outside the factory: this is to avoid mismatches between the scanner and the compiler object (which contains the parser code that uses the scanner).

```
class FlexLexer;
namespace cdk {

    class Compiler;
    class CompilerFactory {
        static std::map<std::string, CompilerFactory *> _factories;

    protected:
        CompilerFactory(const char *lang) {
            _factories[lang] = this;
        }

    public:
        static CompilerFactory *getImplementation(const char *lang) {
            return _factories[lang];
        }

    public:
        virtual ~CompilerFactory();

    protected:
        virtual FlexLexer *createScanner(const char *name) = 0;

    public:
        virtual Compiler *createCompiler(const char *name) = 0;

    }; // class CompilerFactory
} // namespace cdk
```

Figure 2.3: Abstract compiler factory base class.

The programmer **must** provide a concrete subclass for each new compiler/language. Figure 2.4 shows the class definition for the concrete factory, part of the Compact compiler. Note that this class is a singleton object that automatically registers itself with the abstract factory superclass. Classes `CompactScanner` and `CompilerImpl` are implementations for `Compact` of the abstract concepts, `FlexLexer` and `Compiler`, used in the CDK (see, respectively, §2.3.2 and §2.3.3).

Figure 2.5 presents an example implementation of such a subclass for the Compact compiler (this code is not part of the CDK). Notice how the static variable is initialized in this subclass, inserting the instance in the superclass' registry for the given language ("compact", in this example).

```
class CompactCompilerFactory : public cdk::CompilerFactory {
    static CompactCompilerFactory _thisFactory;

protected:
    CompactCompilerFactory(const char *language)
        : cdk::CompilerFactory(language) {}

    FlexLexer *
    createScanner(const char *name) {
        return new CompactScanner(name, NULL, NULL);
    }

    cdk::Compiler *
    createCompiler(const char *name) {
        FlexLexer *scanner = createScanner(name);
        return new CompilerImpl(name, scanner);
    }
}; // class CompactCompilerFactory
```

Figure 2.4: Concrete compiler factory class definition for the Compact compiler (this code is not part of the CDK).

```
CompactCompilerFactory
CompactCompilerFactory::_thisFactory("compact");
```

Figure 2.5: Implementation of the concrete compiler factory for the Compact compiler (this code is not part of the CDK).

2.3.2 The abstract scanner class

The abstract scanner class, `FlexLexer`, is provided by the GNU Flex tool and is not part of the CDK proper. We include it here because the CDK depends on it and the compiler developer **must** provide a concrete subclass for each new compiler/language. Essentially, this class is a wrapper for the code implementing the automaton which will recognize the input language. The most relevant methods are `lineno` (for providing information in source line numbers) and `yyllex` (the lexical analyser itself).

The Compact compiler defines the concrete class `CompactScanner`, for implementing the lexical analyser. Figure 2.6 shows the header file. The rest of the class is implemented automatically from the lexical analyser's specification (§3.4). Note that we also defined `yyerror` as a method, ensuring source code compatibility with traditional C-based approaches.

```
class CompactScanner : public yyFlexLexer {
    const char *_filename;

public: // constructors
    CompactScanner(const char *filename,
                  std::istream *yyin = NULL,
                  std::ostream *yyout = NULL)
        : yyFlexLexer(yyin, yyout), _filename(filename) {
        set_debug(1);
    }

    int yylex(); // automatically generated by flex
    void yyerror(char *s);
};
```

Figure 2.6: Implementation of the concrete lexical analyser class for the Compact compiler (this code is not part of the CDK).

The concrete class will be used by the concrete compiler factory (see §2.3.1) to initialize the compiler (see §2.3.3). There is, in principle, no limit to the number of concrete lexical analyser classes that may be defined for a given compiler. Normally, though, one should be enough to account for the whole lexicon.

2.3.3 The abstract compiler class

The abstract compiler class, `Compiler`, represents the compiler as a single entity and is responsible for performing all high-level compiler actions, namely lexical, syntactic, semantic analysis, and actions deriving from those analyses: e.g., interpretation or code generation.

To carry out those tasks, the compiler class depends on other classes and tools to perform specific actions. Thus, it relies on the scanner class hierarchy (see §2.3.2) to execute the lexical analysis phase and recognize the tokens corresponding to the input program text. As we saw, that code was generated by a specialized tool for creating implementations for regular expression processors. Likewise, the compiler relies on another specialized tool, YACC, to create from a grammar specification, an LALR(1) parser. Currently, this parser is not encapsulated as an object (as was the case with the Flex-created code), but is simply a method of the compiler class itself: `yyparse`.

Besides the compilation-specific parts of the class, it defines a series of flags for controlling how to execute the compilation process. These flags include behaviour flags and input and output processing variables. The following table describes the compiler's instance variables as well as their uses.

Variable	Description
<code>_errors</code>	Counts compilation errors.
<code>_extension</code>	Output file extension (defined by the target and output file options).
<code>_ifile</code>	Input file name.
<code>_istream</code>	Input file stream (default <code>std::cin</code>).
<code>_name</code>	Language name.
<code>_optimize</code>	Controls whether optimization should be performed.
<code>_ofile</code>	Output file name.
<code>_ostream</code>	Output file stream (default <code>std::cout</code>).
<code>_scanner</code>	Pointer to the scanner object to be used by the compiler.
<code>_syntax</code>	Syntax tree representation (nodes).
<code>_trace</code>	Controls compiler execution trace level.
<code>_tree</code>	Create only the syntax tree: this is the same as specifying an XML target (see <code>_extension</code> above).

Note that, although the scanner is passed to the compiler constructor as an initialization parameter, the programmer is free to change an existing compiler's scanner at any time¹. If this is the case, then the scanner's input and output streams will be reset with the compiler's.

Regarding the syntactic tree, it starts as a null pointer and is initialized as a consequence of running the parser (call to `parse` / `yyparse`) (see §2.3.4). This implies that, even though any action could be possible in YACC actions, a node structure must be created to represent the input program and serve as input for the semantic processing phase executed through a call to the `evaluate` method (see §2.3.6). Figure 2.7 presents these methods.

Since `yyparse` is pure virtual in the `Compiler` class, the programmer **must** provide a concrete subclass for each new compiler/language. In the Compact compiler, this concrete class is called simply `CompilerImpl` and has as its single method `yyparse`. The concrete subclass works simply as a wrapper for the parsing function, while the rest of the compilation process is non-language-specific and is handled by the general code in the superclass defined by the CDK.

2.3.4 The parsing function

As mentioned in the previous section, the parsing function is simply the result of processing an LALR(1) parser specification with a tool such as Berkeley YACC or GNU Bison. Such a function is usually called `yyparse` and is, in the case of BYACC and the default action for Bison, written in C. Bison can, however produce C++ code and sophisticated reentrant parsers. In the current version of the CDK, it is assumed that the tool used to process the syntactic specification is BYACC and that the native code is C.

¹The capability to change the scanner does not mean that the change is in any way a good idea. It may, indeed, be inadvisable in general. You have been warned.

```

namespace cdk {
  namespace node { class Node; }
  class Compiler {
    typedef cdk::semantics::Evaluator evaluator_type;

  protected:
    virtual int yyparse() = 0;

  public:
    inline int parse() { return yyparse(); }

  public:
    virtual bool evaluate() {
      evaluator_type *evaluator =
        evaluator_type::getEvaluatorFor(_extension);
      if (evaluator) return evaluator->evaluate(this);
      else exit(1); // no evaluator defined for target
    }

  }; // class Compiler
} // namespace cdk

```

Figure 2.7: Abstract CDK compiler class (simplified view): note that `yyparse` is pure virtual.

We have thus a compatibility problem: C is similar to C++, but it not C++. Fortunately, the cases in which C and C++ disagree do not manifest themselves in the generated code and the only care to be taken is that the function created by the tool can be converted into a method. This procedure will allow calling the parser as if it were an object. This “fiction” (for it is a fiction) should not be confused with the “real thing” if the CDK were to be reimplemented².

Grammar specifications will be presented in detail in chapters 6 (syntactic theory), 7 (the Berkeley YACC tool), and 8 (syntax in the Compact compiler). For a given language/compiler, the programmer **must** provide a new grammar: in the Compact compiler, this is done in file `CompactParser.y`. Again, note that this does not mean that there will be a `CompactParser` class: the code is simply incorporated in the concrete compiler class `CompilerImpl`. Figure 2.8 shows part of the syntactic specification and the macros that ensure that the `yyparse` function is indeed included in the `CompilerImpl` class. Analogously, any calls to `yylex` (in C) must be replaced by method invocations on the scanner object.

2.3.5 The node set

Besides the classes defining compiler architecture, the CDK framework also includes classes for representing language concepts, i.e., they represent com-

²Note that although the code is contained in a single file, there is no guarantee that the global variables it contains do not cause problems: for instance, if multiple parsers were to be present at a given moment.

```
%{  
#define LINE scanner()->lineno()  
#define yylex scanner()->yylex  
#define yyparse CompilerImpl::yyparse  
%}  
%%  
    // ... rules...  
%%
```

Figure 2.8: Partial syntax specification for the Compact compiler: note the macros used to control code binding to the `CompilerImpl` class.

pilation results. The classes that represent syntactic concepts are called nodes and they form tree structures that represent whole or part of programs: the syntax tree.

Although it is difficult, if not outright impossible, to predict what concepts are defined by a given programming language, the CDK, nevertheless, tries to provide a small set of basic nodes for simple, potentially useful, concepts. The reason is twofold: on the one hand, the nodes provide built-in support for recurrent concepts; on the other hand, they are useful examples for extending the CDK framework.

Figure 2.9 presents the UML diagram for the CDK node hierarchy. The nodes are fairly general in nature: general concepts for unary and binary operators, as well as particularizations for commonly used arithmetics and logical operations. In addition terminal nodes for storing primitive types are also provided: a template class for storing any atomic type (`Simple`) and its instantiations for integers, doubles, strings, and identifiers (a special case of string). Other special nodes are also provided: `Data`, for opaque data types; `Sequence`, for representing data collections organized as vectors; `Composite`, for organizing data collections as linearized trees; `Nil`, for representing empty nodes (null object).

When developing a new compiler, the programmer has to provide new concrete subclasses, according to the concepts to be supported. In the Compact compiler, for instance, nodes were defined for concepts such as while loops, if-then-else instructions, and so on. See chapters 8 and 11 for detailed information.

2.3.6 The abstract evaluator

After the parser has performed the syntactic analysis, we have a syntax tree representing the structure of the input program. This structured is formed by instances of the node set described in §2.3.5. Representing the program, though, is simply a step towards computing its true meaning or semantics. This is the evaluator's task: to take the syntax tree and extract from it the semantics corresponding to the concepts modeled by the input programming language.

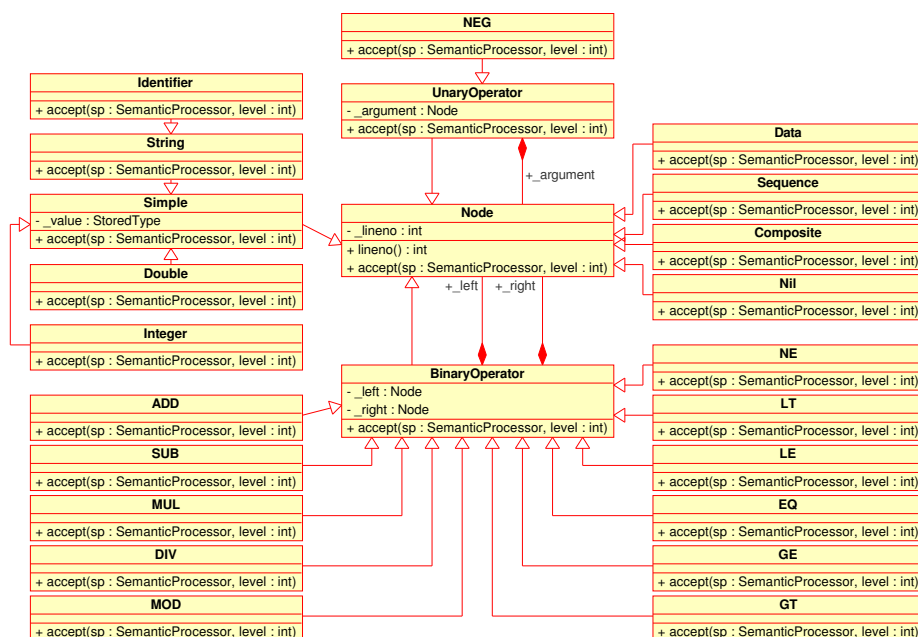


Figure 2.9: CDK node hierarchy class diagram.

The CDK provides an abstract evaluator class for defining the interface its subclasses must implement. The compiler class, when asked to evaluate the program, creates a concrete evaluator for the selected target (see figure 2.7). The programmer **must** provide a concrete subclasses for each new compiler/language/target. Each such class will automatically register its instance with the superclass' registry for a given target. In the Compact compiler, four concrete subclasses are provided: for generating XML trees (XMLevaluator); for generating assembly code (PFEvaluator); for generating C code (Cevaluator); and for interpreting the program, i.e., directly executing the program from the syntax tree (InterpretationEvaluator).

To do its task, the evaluator needs two entities: the syntax tree to be analysed and the code for deciding on the meaning of each node or set of nodes. It would be possible to write this code as a set of classes, global functions, or even as methods in the node classes. Nevertheless, all these solutions present disadvantages: using multiple classes or multiple functions would mean that some kind of selection would have to be made in the evaluation code, making it more complex than necessary; using methods in the node classes would solve the former problem, but would make it difficult, or even impossible, to reuse the node classes for multiple purposes (such as generating code for different targets).

The selected solution is to use the Visitor design pattern (described in §2.3.7).

2.3.7 The abstract semantic processor

Figure 2.10 shows a partial specification of the abstract semantic processor. Note that the interface cannot be defined independently from the node set used by a specific compiler: the visitor class, by the very nature of the Visitor design pattern must provide a visiting method (`process nodes`) for each and every node present in the compiler implementation. The minimum interface is that used for handling the node set already present in the CDK (see §2.3.5).

```
class SemanticProcessor {
    std::ostream &_os; // output stream

protected:
    SemanticProcessor(std::ostream &os = std::cout) : _os(os) {}
    inline std::ostream &os() { return _os; }

public:
    virtual ~SemanticProcessor() {}

public: // CDK nodes
    virtual void processNode(cdk::node::Node *const, int) = 0;
    virtual void processNil(cdk::node::Nil *const, int) = 0;
    virtual void processSequence(cdk::node::Sequence *const, int) = 0;
    virtual void processInteger(cdk::node::Integer *const, int) = 0;
    virtual void processString(cdk::node::String *const, int) = 0;
    //...

public: // Compact nodes
    virtual void processWhileNode(WhileNode *const node, int lvl) = 0;
    virtual void processIfNode(IfElseNode *const node, int lvl) = 0;
    //...
};
```

Figure 2.10: Partial specification of the abstract semantic processor. Note that the interface cannot be defined independently from the node set used by a specific compiler.

Each language implementation must provide a new class containing methods for both the CDK node set and for the node set in that language. For instance, the Compact compiler defines such nodes as `WhileNode` and `IfElseNode`. Thus, the corresponding abstract semantic processor must define methods `processWhileNode` and `processIfNode` (among others).

2.3.8 The code generators

Currently, the CDK provides an abstract stack machine for generating the final code. Visitors for final code production will call the stack machine's pseudo-instructions while performing the evaluation of the syntax tree. The pseudo-instructions will produce the final machine code. The stack machine is encapsulated by the `Postfix` abstract class. Figure 2.11 shows the sequence diagram

for the syntax tree evaluation process: this includes tree traversal and including code generation.

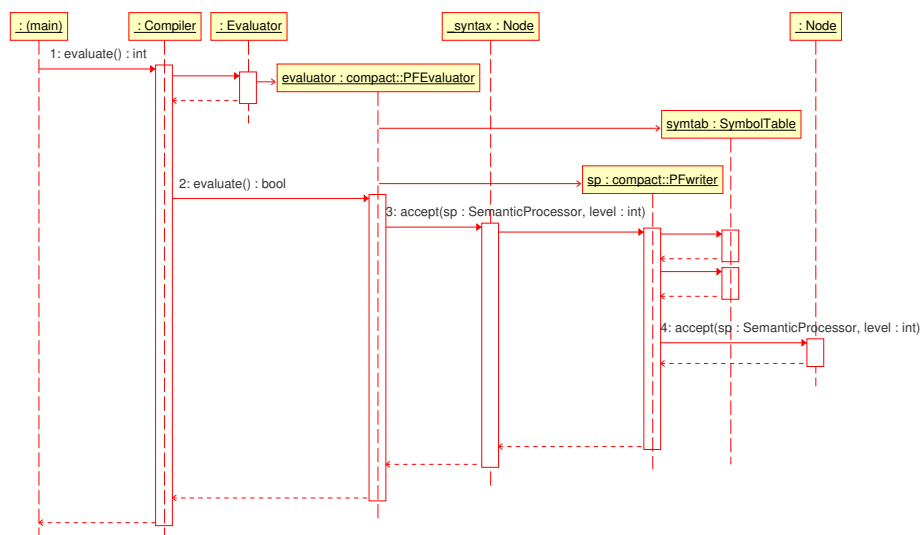


Figure 2.11: CDK library's sequence diagram for syntax evaluation.

Two concrete generator classes are provided: `DebugOnly`, providing abstract output, i.e., which pseudo-instructions were generated; and `ix86`, a code generator compatible with NASM (). The latter class is also capable of generating pseudo-code debugging information. Currently, the code produced by the `ix86` class is only guaranteed to run when compiled with NASM in a 32-bit environment.

Appendix B presented an in-depth description of the postfix interface for code generation.

2.3.9 Putting it all together: the main function

The main function is where every part, both defined on the CDK or redefined/implemented in each compiler, comes together (see figure 2.2): the first step is to determine which language the compiler is for (step 1) and getting the corresponding factory, to create all the necessary objects (step 2). Steps 3 through 7 correspond to parsing the input to produce final code (currently, assembly, by default).

Note that the main function is already part of the CDK and there is, thus, no need for the programmer to provide it.

The code corresponding to the above diagram (simplified version) is depicted in figure 2.12.

```
int main(int argc, char *argv[]) {
    std::string language; // the language to compile

    // ... determine language...

    cdK::CompilerFactory *fact =
        cdK::CompilerFactory::getImplementation(language.c_str());
    if (!fact) {
        // fatal error: no factory available for language
        return 1; // failure
    }

    cdK::Compiler *compiler = fact->createCompiler(language.c_str());
    // ... process command line arguments...

    if (compiler->parse() != 0 || compiler->errors() > 0) {
        // ... report syntax errors...
        return 1; // failure
    }

    if (!compiler->evaluate()) {
        // ... report semantic errors...
        return 1; // failure
    }

    return 0; // success
}
```

Figure 2.12: CDK library’s main function code sample corresponding to the sequence diagram above.

2.4 Summary

In this chapter we presented the CDK library and how it can be used to build compilers for specific languages. The CDK contains classes for representing all concepts involved in a simple compiler: the scanner or lexical analyser, the parser, and the semantic processor (including code generation and interpretation). Besides the simple classes, the library also includes factories for abstracting compiler creation as well as creation of the corresponding evaluators for specific targets. Evaluation is based on the Visitor design pattern: it allows for specific functionality to be decoupled from the syntax tree, making it easy to add or modify the functionality of the evaluation process.

The next chapters will cover some of the topics approached here concerning lexical and syntactic analysis, as well as semantic processing and code generation. The theoretical aspects, covered in chapters 3 and 6, will be supplemented with support for specific tools, namely GNU Flex (chapter 4) and Berkeley YACC (chapter 7). This does not mean that other similar tools cannot be used: it means simply that the current implementation directly supports those two. In addition to the description of each tool, the corresponding code in use in the Compact compiler will also be presented, thus illustrating the full process.

II

Lexical Analysis

Theoretical Aspects of Lexical Analysis

3.1 *What is Lexical Analysis?*

Lexical analysis is the process of analysing the input text and recognizing elements of the language being processed. These elements are called tokens and are associated with lexemes (bits of text associated with each token).

There are several forms of performing lexical analysis, one of the most common being finite state-based approaches, i.e., those using a finite state machine to recognize valid language elements.

This chapter describes Thompson's algorithm for building finite-state automata for recognizing/accepting regular expressions.

3.1.1 **Language**

Formally, a language (more precisely, a lexicon) is defined as ():

Note that this definition is not that of a grammar (covered in §6.2). In particular, lexical analysers do not usually concern themselves with structure above that of individual language items.

3.1.2 **Regular Language**

One particular type of language

Formally, a regular language is defined as ():

3.1.3 **Regular Expressions**

3.2 *Finite State Acceptors*

Since we are going to use sets of regular expressions for recognizing input strings, we need a way of implementing that functionality. The recognition process can be efficiently carried out by finite state automata that either accept or reject a given string.

Ken Thompson, the creator of the B language (one of the predecessors of C) and one of the creators of the UNIX operating system, devised the algorithm

that carries his name and describes how to build an acceptor for a given regular expression.

Created for Thompson's implementation of the grep UNIX command, the algorithm creates an NFA from a regular expression specification that can then be converted into a DFA. It is this DFA that after minimization yields an automaton that is an acceptor for the original expression.

The following sections cover the algorithm's construction primitives and how to recognize a simple expression. Lexical analysis such as performed by flex is presented in §3.4. In this case, several expressions may be watched for, each one corresponding to a token. Such automata feature multiple final states, one or more for each recognized expression.

3.2.1 Building the NFA

Thompson's algorithm is based on a few primitives, as show in table 3.1.

Other expressions can be obtained by simply combining the above primitives, as illustrated by the following example, corresponding to the expression $a(a|b)^*|c$ (see figure 3.1).

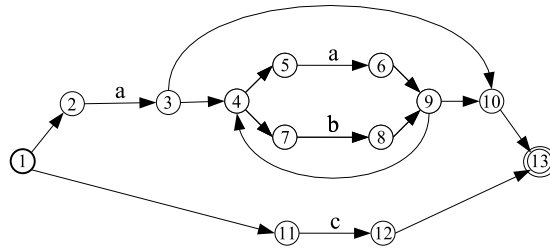


Figure 3.1: Thompson's algorithm example for $a(a|b)^*|c$.

3.2.2 Determinization: Building the DFA

NFAs are not well suited for computers to work with, since each state may have multiple acceptable conditions for transitioning to another state. Thus, it is necessary to transform the automaton so that each state has a single transition for each possible condition. This process is called determination. The algorithm for transforming an NFA into a DFA is a simple one and relies on two primitive functions, *move* and ϵ -closure.

The *move* function is defined over a set of NFA states and input symbol pairs and a set of NFA states sets: for each state and input symbol, it computes the set of reachable states. As an example consider, for the NFA in figure 3.1:

$$\text{move}(\{2\}, a) = \{3\} \quad (3.1)$$

$$\text{move}(\{5\}, a) = \{6\} \quad (3.2)$$

$$\text{move}(\{11\}, a) = \{\} \quad (3.3)$$

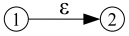
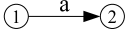
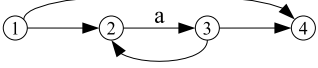
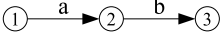
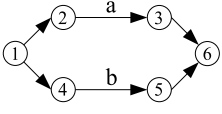
Example	Diagram	Meaning
ϵ		Empty expression.
a		One occurrence of an expression.
a^*		Zero or more occurrences of an expression: this case may be generalized for more complex expression. In this case, the complex expression will simply take the place of a arc in the diagram.
ab		Concatenation of two or more expressions: the first expression's final state coincides with the second's. This case, like the previous one, may be generalized to describe more complex concatenations.
$a b$		Alternative expressions: the to initial states and the final states of each expression are connected to two new states. Both expressions may be replaced by more general cases.

Table 3.1: Primitive constructions in Thompson's algorithm.

The ϵ -closure function is defined for sets of states: the function computes a new set of states reachable from the initial set by using only all the possible ϵ transitions to other states (including the each state itself), as well as the states reachable through ϵ transitions from those states. Thus, considering the NFA in figure 3.1, we could write:

$$\epsilon - \text{closure}(\{1\}) = \{1, 2, 11\} \quad (3.4)$$

$$\epsilon - \text{closure}(\text{move}(\{2\}, a)) = \epsilon - \text{closure}(\{3\}) = \{3, 4, 5, 7, 10, 13\} \quad (3.5)$$

With the two above functions we can describe a determinization algorithm. The input for the determinization algorithm is a set of NFA states and their corresponding transitions; a distinguished start state and a set of final states. The output is a set of DFA states (as well as the configuration of NFA states corresponding to each DFA state); a distinguished start state and a set of final states.

The algorithm considers an agenda containing pairs of DFA states and input symbols. Each pair corresponds to a possible transition in the DFA (possible in the sense that it may not exist). Each new state, obtained from considering successful transitions from agenda pairs, must be considered as well with each input symbol. The algorithm ends when no more pairs exist in the agenda and no more can be added.

DFA states containing in their configurations final NFA states are also final.

Step 1: Compute the ϵ -closure of the NFA's start state. The resulting set will be the DFA's start state, I_0 . Add all pairs (I_0, α) ($\forall \alpha \in \Sigma$, with Σ the input alphabet) to the agenda.

Step 2: For each unprocessed pair in the agenda (I_n, α) , remove it from the agenda and compute $\epsilon - \text{closure}(\text{move}(I_n, \alpha))$: if the resulting configuration, I_{n+1} , is not a known one (i.e., it is different from all $I_k, \forall k < n+1$), add the corresponding pairs to the agenda.

Step 3: Repeat 2 until the agenda is empty.

The algorithm's steps can be tabled (see fig. 3.2): $\Sigma = \{a, b, c\}$ is the input alphabet; $\alpha \in \Sigma$ is an input symbol; and $I_{n+1} = \epsilon - \text{closure}(\text{move}(I_n, \alpha))$.

Figure 3.3 presents a graph representation of the DFA computed in accordance with the determinization algorithm. The numbers correspond to DFA states whose NFA state configurations are presented in figure 3.2.

3.2.3 Compacting the DFA

The compaction process is simply a way of eliminating DFA states that are unnecessary. This may happen because one or more states are indistinguishable from each other, given the input symbols.

I_n	$\alpha \in \Sigma$	$move(I_n, \alpha)$	$I_{n+1} - move(I_n, \alpha)$	I_{n+1}
–	–	1	2, 11	1
1	a	3	4, 5, 7, 10, 13	2
1	b	–	–	–
1	c	12	13	3
2	a	6	4, 5, 7, 9, 10, 13	4
2	b	8	4, 5, 7, 9, 10, 13	5
2	c	–	–	–
3	a	–	–	–
3	b	–	–	–
3	c	–	–	–
4	a	6	4, 5, 7, 9, 10, 13	4
4	b	8	4, 5, 7, 9, 10, 13	5
4	c	–	–	–
5	a	6	4, 5, 7, 9, 10, 13	4
5	b	8	4, 5, 7, 9, 10, 13	5
5	c	–	–	–

Figure 3.2: Determinization table example for $a(a|b)^*|c$. $I_0 = \epsilon - closure(\{1\})$ and $I_{n+1} = \epsilon - closure(move(I_n, \alpha))$. Final states are marked in bold.

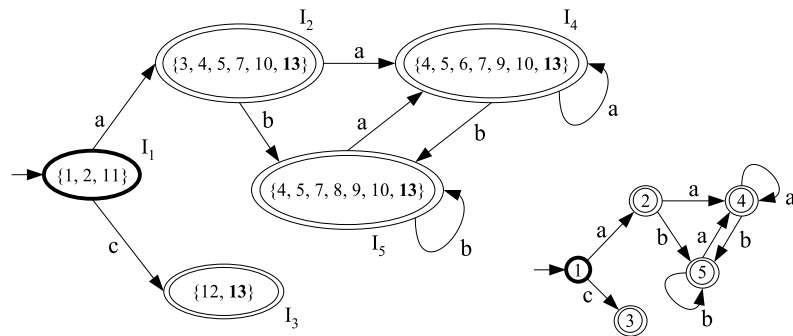


Figure 3.3: DFA graph for $a(a|b)^*|c$: full configuration and simplified view (right).

A simple algorithm consists of starting with a set containing all states and progressively dividing it according to various criteria: final states and non-final states are fundamentally different, so the corresponding sets must be disjoint; states in a set that have transitions to different sets, when considering the same input symbol are also different; states that have transitions on a given input symbol are also different from states that do not have those transitions. The algorithm must be applied until no further tests can be carried out.

Regarding the above example, we would have the following sets:

- All states: $A = \{1, 2, 3, 4, 5\}$; separating final and non-final states we get
- Final states, $F = \{2, 3, 4, 5\}$; and non-final states, $NF = \{1\}$;
- Considering a and F : 2, 4, and 5 present similar behavior (all have transitions ending in states in the same set, i.e., 4); 3 presents a different behavior (i.e., no a transition). Thus, we get two new sets: $\{2, 4, 5\}$ and $\{3\}$;
- Considering b and $\{2, 4, 5\}$ we reach a conclusion similar to the one for a , i.e., all states have transitions to state 5 and cannot, thus, be distinguished from each other;
- Since $\{2, 4, 5\}$ has no c transitions, it remains as is. Since all other sets are singular, the minimization process stops.

Figure 3.4 presents the process of minimizing the DFA (the starting point is the one in figure 3.2), in the form of a minimization tree.

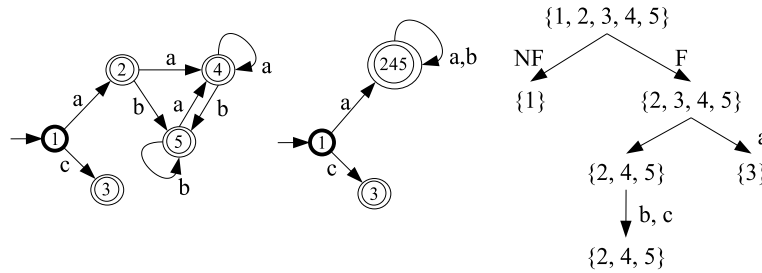


Figure 3.4: Minimal DFA graph for $a(a|b)^*|c$: original DFA, minimized DFA, and minimization tree.

3.3 Analysing a Input String

After producing the minimized DFA, we are ready to process input strings and decide whether or not they accepted by the regular expression. The analysis process uses a table for keeping track of the analyser's current state as well as of the transitions when analysing the input. The analysis process ends when there is no input left to process and the current state is a final state. If the input

is empty and the state is not final, then there was an error and the string is said to be rejected. If there is no possible transition for a given state and the current input symbol, then processing fails and the string is also rejected.

3.4 Building Lexical Analysers

A lexical analyser is an automaton that, in addition to accepting or rejecting input strings, also identifies the expression that matched the input. This identifier is known as token.

Building lexical analysers is a simple matter of compising multiple analysers for the component regular expressions. However, final states corresponding to different expressions must be kept separate. Other than this restriction, the process of building the DFA is the same as before: first the NFA is built according to Thompson's algorithm and the corresponding DFA minimized. The minimization process accounts for another slight difference: after separating states according to whether they are final or non-final, final states must be divided into sets according to the expressions they recognize.

3.4.1 The construction process

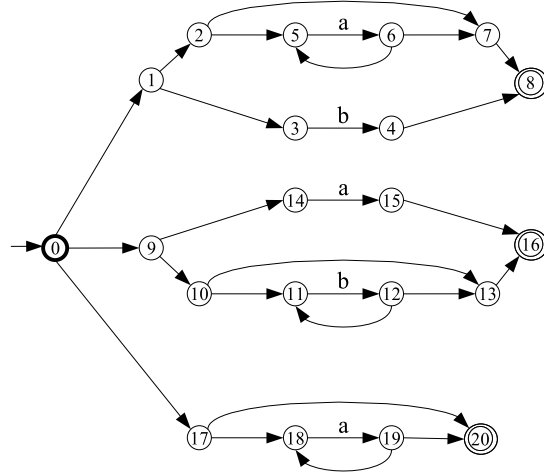
The following example illustrates the construction process for a lexical analyser that identifies three expressions: $G = \{a * |b, a|b*, a*\}$. Thus, the recognized tokens are $TOK1 = a * |b$, $TOK2 = a|b*$, and $TOK3 = a*$. Note that the construction process handles ambiguity by selecting the token that consumes the most input characters and, if two or more tokens match, by selecting the first. It may possible that the lexical analyser never signals one of the expressions: in an actual situations, this may be undesirable, but may be unavoidable. For instance, when recognizing identifiers and keywords, care must be exercised so as not to select an identifier when a keyword is desired.

3.4.1.1 The NFA

As figure 3.6 clearly illustrates, all DFA states are final: each of them contains, at least, one final NFA state. When several final NFA states are present, the first is the one considered. In this way, we are able to select the first expression in the list, when multiple matches would be possible. Note also that the third expression is never matched. This expression corresponds to state 20 in the NFA: in the DFA this state never occurs by itself, meaning that the first expression is always preferred (as expected).

3.4.1.2 The DFA and the minimized DFA

The minimization process is as before, but now we have to take into account that states may differ only with respect to the expression they recognize. Thus, after splitting states sets into final and non-final, the set of final states should be

Figure 3.5: NFA for a lexical analyser for $G = \{a * | b, a | b *, a *\}$.

I_n	α	$move(I_n, \alpha)$	$I_{n+1} - move(I_n, \alpha)$	I_{n+1}	Token
–	–	0	1, 2, 3, 5, 7, 8, 9, 10, 11, 13, 14, 16 , 17, 18, 20	0	TOK1
0	a	6, 15, 19	5, 7, 8 , 16 , 18, 20	1	TOK1
0	b	4, 12	8 , 11, 13, 16	2	TOK1
1	a	6, 19	5, 7, 8 , 18, 20	3	TOK1
1	b	–	–	–	–
2	a	–	–	–	–
2	b	12	11, 13, 16	4	TOK2
3	a	6, 19	5, 7, 8 , 18, 20	3	TOK1
3	b	–	–	–	–
4	a	–	–	–	–
4	b	12	11, 13, 16	4	TOK2

Figure 3.6: Determinization table for $G = \{a * | b, a | b *, a *\}$. $I_0 = \epsilon - closure(\{0\})$ and, as before, $I_{n+1} = \epsilon - closure(move(I_n, \alpha))$, $\alpha \in \Sigma$. Final states are marked in bold.

split according to the recognized expression. From this point on, the procedure is as before.

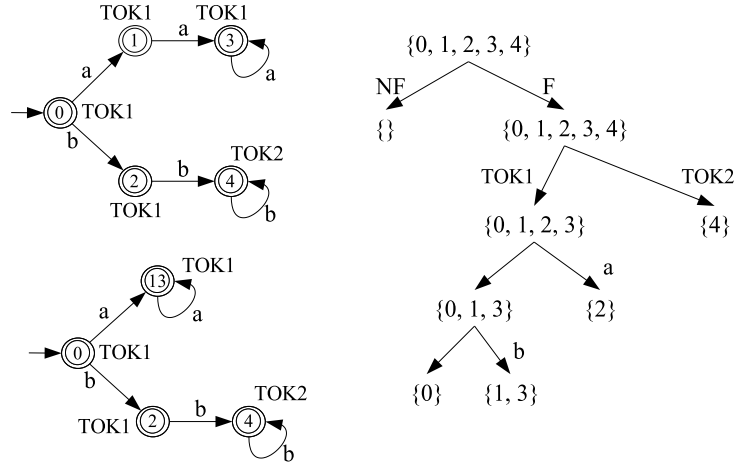


Figure 3.7: DFA for a lexical analyser for $G = \{a * | b, a|b*, a*\}$: original (top left), minimized (bottom left), and minimization tree (right). Note that states 2 and 4 cannot be merged since they recognize different tokens.

3.4.2 The Analysis Process and Backtracking

Figure 3.8 shows the process of analysing the input string *aababb*. As can be seen from the table, several tokens are recognized and, for each one, the analyser returns to the initial state to process the remainder of the input.

I_n	Input	I_{n+1}
0	aababb\$	13
13	ababb\$	13
13	babb\$	TOK1
0	babb\$	2
2	abb\$	TOK1
0	abb\$	13
13	bb\$	TOK1
0	bb\$	2
2	b\$	4
4	\$	TOK2

Figure 3.8: Processing an input string and token identification. The input string *aababb* is split into *aa* (TOK1), *b* (TOK1), *a* (TOK1), and *bb* (TOK2).

3.5 Summary

4

The GNU flex Lexical Analyser

4.1 Introduction

4.1.1 The lex family of lexical analysers

4.2 The GNU flex analyser

4.2.1 Syntax of a flex analyser definition

4.2.2 GNU flex and C++

4.2.3 The FlexLexer class

4.2.4 Extending the base class

4.3 Summary

5

Lexical Analysis Case

This chapter describes the application of the lexical processing theory and tools to our test case, the *compact* programming language.

5.1 *Introduction*

5.2 *Identifying the Language*

5.2.1 **Coding strategies**

5.2.2 **Actual analyser definition**

5.3 *Summary*

III Syntactic Analysis

Theoretical Aspects of Syntax

Syntactic analysis can be carried out by a variety of methods, depending on the desired result and the type of grammar and analysis.

falar dos diferentes métodos?

6.1 *Introduction*

This chapter is centered around the LR family of parsers: these are bottom-up parsers that shift items from the input to the stack and reduce symbols on the stack in accordance with available grammar rules.

6.2 *Grammars*

6.2.1 **Formal definition**

6.2.2 **Example grammar**

In the current chapter, we will use a simple grammar for illustrating our explanations with clear examples. We will choose a simple grammar but one which will allow us to exercise a wide range of processing decisions. This grammar is presented in 6.1: E (the start symbol) and F are non-terminals and id is a terminal (token) that represents arbitrary identifiers (variables).

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow T * F | F \\ F &\rightarrow (E) | id \end{aligned} \tag{6.1}$$

In addition to the non-terminals and the token described above, four other tokens, whose value is also the same as their corresponding lexemes, exist: $($, $)$, $+$, and $*$.

6.2.3 FIRST and FOLLOWS

6.3 LR Parsers

As mentioned in the introduction, this chapter is centered around the LR family of parsers: these are bottom-up parsers that shift items from the input to the stack and reduce symbols on the stack in accordance with available grammar rules.

An LR parser has the following structure (Aho et al., 1986, fig. 4.29):

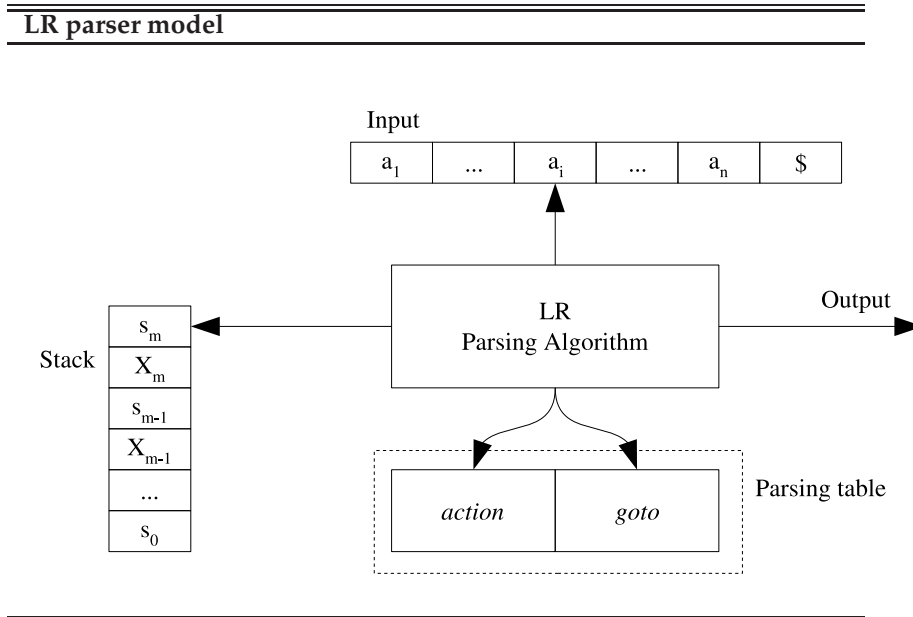


Figure 6.1: LR parser model.

The stack starts with no symbols, containing only the initial state (typically 0). Parsing consists of pushing new symbols and states to the stack (shift operation) or in removing groups of symbols from the stack – corresponding to the right hand side of a production and pushing back the left hand side of that production (reduce operation). Parsing ends when the end of phrase is seen in the appropriate state (see also §6.4.4).

The parse table is built according to the following algorithm: it takes as input an augmented grammar (§6.3.1.1), and produces as output the parser table (actions and gotos).

1. The first step is to build $C = \{I_0, \dots, I_n\}$, a collection of items corresponding to the augmented grammar. If we consider LR(0) items (§6.3.1), then the parse table will produce either a LR(0) parser (§6.3.3) or a SLR(1) parser (§6.3.4). If LR(1) items (§6.4.1) are considered, then a LALR(1) parser can be built. In this case, the other parser types can also be

built (although the effort of computing the LR(1) items would be partially wasted).

2. Each state i is built from the DFA's I_i state. Actions in state i are built according to the following method (with terminal a):
 - a) If $[A \rightarrow \alpha \bullet a\beta]$ is in I_i and $\text{goto}(I_i, a) = I_j$, then $\text{action}[i, a] = \text{shift } j$;
 - b) If $[A \rightarrow \alpha \bullet]$ is in I_i , then $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha, \forall a \in \text{FOLLOW}(A)$ (with A distinct from S');
 - c) If $[S' \rightarrow S \bullet]$ is in I_i , then $\text{action}[i, a] = \text{accept}$.
3. Gotos to state i (with non-terminal A): if $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$.
4. The parser table cells not filled by the second and third steps correspond to parse errors;
5. The parser's initial state corresponds to set of items containing item $[S' \rightarrow \bullet S]$ (§6.3.1.4).

6.3.1 LR(0) items and the parser automaton

Informally, an LR(0) item is a “dotted rule”, i.e., a grammar rule and a dot indicating which parts of the rule have been seen/recognized/accepted so far. As an example, consider rule 6.2: the LR(0) items for this rule are presented in 6.3.

$$E \rightarrow ABC \quad (6.2)$$

$$E \rightarrow \bullet ABC \quad E \rightarrow A \bullet BC \quad E \rightarrow AB \bullet C \quad E \rightarrow ABC \bullet \quad (6.3)$$

Only one LR(0) item (6.5) exists for empty rules (6.4).

$$E \rightarrow \epsilon \quad (6.4)$$

$$E \rightarrow \bullet \quad (6.5)$$

These dotted rules can be efficiently implemented as a pair of integer numbers: the first represents the grammar rule and the second the dot's position within the rule.

The idea behind the LR parsing algorithm is to build an automaton for recognizing viable prefixes. The automaton may be built by computing all LR(0) items and grouping them. For this task three additional concepts are needed: an augmented grammar (§6.3.1.1); a closure function (§6.3.1.2); and a goto function (§6.3.1.3).

6.3.1.1 Augmented grammars

If S is the start symbol for a given grammar, then the corresponding augmented grammar is defined by adding an additional rule $S' \rightarrow S$ and defining S' as

the new start symbol. The idea behind the concept of augmented grammar is to make it simple to decide when to stop processing input data. With the extra production added to the augmented grammar, it is a simple matter of keeping track of the reduction of the old start symbol when processing the new production. Thus, and now in terms of LR(0) items, the entire processing would correspond to navigating through the automaton, starting in the state with $[S' \rightarrow \bullet S]$ and ending at $[S' \rightarrow S \bullet]$, with no input left to be processed.

YACC parser generators stop processing when the start symbol of the original grammar is reduced. Bison, however, introduces a new transition to an extra state (even when in “YACC compatibility mode”). This transition (the only difference from a YACC parser) corresponds to processing the end-of-phrase symbol (see below). The parser automata are otherwise identical.¹

The augmented grammar corresponding to the grammar to be processed. The augmented grammar for the grammar presented in 6.1 is shown in 6.6. This grammar has a new start symbol: E' .

Example 1

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned} \tag{6.6}$$

For building a parser for this grammar, the NFA automaton would start in state $[E' \rightarrow \bullet E]$ and end in state $[E' \rightarrow E \bullet]$. After determinizing the NFA, the above items would be part of the DFA's initial and final states.

So, all there is to do to build a parser is to compute all possible LR(0) items: this is nothing more than considering all possible positions for a “dot” in all the possible productions and the causes of transition from one item to another: if a terminal is after the dot, then the transition is labeled with that terminal; otherwise, a new ϵ transition to all LR(0) items that can generate (by reduction) the non-terminal after the dot will be produced. By following the procedure until no more transitions or items are added to the set, the parser's NFA is finished. The final state is the one containing the $[E' \rightarrow E \bullet]$ item.

Determinization proceeds as in the case of the lexical automata. When implementing programs for generating these parsers, starting with the NFA may seem like a good idea (after all, the algorithms may be reused), when building the state machines by hand, first building the NFA, and afterwards the DFA, is very time consuming and error prone. Fortunately, it is quite straightforward to build the DFA directly from the LR(0) items. We will see how in a moment: first we will introduce two concepts that will help us do it.

explicar que a closure e o goto permitem não construir o NFA e avançar para o DFA diretamente.

¹In fact, Bison makes a better job at code generation for supporting the parser. In this document, however, we use the YACC tool as originally defined.

6.3.1.2 The closure function

As in the case of the determinization of the automaton associated with lexical analysis, we define a closure operation. The closure function is defined for sets of items (def. 1).

Definition 1 Closure.

Let I be a set of items. Then $\text{closure}(I)$ is a set of items such that:

1. Initially, all elements belonging to I are also in $\text{closure}(I)$;
2. If $[A \rightarrow \alpha \bullet B\beta] \in \text{closure}(I)$ and $B \rightarrow \gamma$ is a production, then add item $[B \rightarrow \bullet\gamma]$ to $\text{closure}(I)$. Repeat until no more items can be added to the closure set.

Example 2 shows the application of the closure function to our example augmented grammar (as defined in 6.6).

Example 2

$$\begin{aligned}
 I &= \{[E \rightarrow \bullet E']\} \\
 \text{closure}(I) &= \{ \\
 &\quad [E \rightarrow \bullet E'], [E \rightarrow \bullet E + T], [E \rightarrow \bullet T], [T \rightarrow \bullet T * F], \\
 &\quad [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet (E)], [F \rightarrow \bullet id] \\
 &\quad \}
 \end{aligned} \tag{6.7}$$

6.3.1.3 The “goto” function

Given a grammar symbol and a set of items, the goto function computes the closure of the set of all possible transitions on the selected symbol (see definition 2).

Definition 2 Goto function.

If I is a set of items and X is a grammar symbol, the the goto function, $\text{goto}(I, X)$, is defined as:

$$\text{goto}(I, X) = \text{closure}(\{[A \rightarrow \alpha X \bullet \beta] : \text{for all items } [A \rightarrow \alpha \bullet X\beta] \in I\}) \tag{6.8}$$

Informally, if γ is a viable prefix for the elements in I , then γX is a viable prefix for the elements in $\text{goto}(I, X)$. In reality, the only symbols that matter are those that follow immediately after the “dot” (see example 3).

Example 3

$$\begin{aligned}
 I &= \{[E \rightarrow E' \bullet], [E \rightarrow E \bullet + T]\} \\
 \text{goto}(I, +) &= \{[E \rightarrow E + \bullet T], [T \rightarrow \bullet T * F], [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet id]\} \\
 &\tag{6.9}
 \end{aligned}$$

Note that, in 6.8, only the second item in I contributes to the set formed by $goto(I, +)$. This is because no other item has $+$ in any viable prefix.

6.3.1.4 The parser's DFA

Now that we have defined the closure and goto functions, we can build the set of states C of the parser's DFA automaton (see def. 3).

Definition 3 DFA set.

Initially, for a grammar with start symbol S , the set of states is (S' is the augmented grammar's start symbol):

$$C = \{closure(\{[S' \rightarrow \bullet S]\})\} \quad (6.10)$$

Then, for each I in C and for each grammar symbol X , add $goto(I, X)$, if not empty, to C . Repeat until no changes occur to the set.

Let us now consider our example and build the sets corresponding to the DFA for the parser: as defined in 3, we will build $C = \{I_0, \dots, I_n\}$, the set of DFA states. Each DFA state will contain a set of items, as defined by the closure and goto functions. The first state in C corresponds to the DFA's initial state, I_0 :

$$\begin{aligned} I_0 &= closure(\{[E' \rightarrow \bullet E]\}) \\ I_0 &= \{ \\ &\quad [E \rightarrow \bullet E'], [E \rightarrow \bullet E + T], [E \rightarrow \bullet T], [T \rightarrow \bullet T * F], \\ &\quad [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet id] \\ &\} \end{aligned} \quad (6.11)$$

After computing I_0 , the next step is to compute $goto(I_0, X)$ for all symbols X for which there will be viable prefixes: by inspection, we can see that these symbols are $E, T, F, ($, and id . Each set resulting from the goto function will be a new DFA state. We will consider them in the following order (but, of course, this is arbitrary):

$$\begin{aligned} I_1 &= goto(I_0, E) = \{[E \rightarrow E' \bullet], [E \rightarrow E \bullet + T]\} \\ I_2 &= goto(I_0, T) = \{[E \rightarrow T \bullet], [T \rightarrow T \bullet * F]\} \\ I_3 &= goto(I_0, F) = \{[T \rightarrow F \bullet]\} \\ I_4 &= goto(I_0, () = \{ \\ &\quad [F \rightarrow (\bullet E)], [E \rightarrow \bullet E + T], [E \rightarrow \bullet T], [T \rightarrow \bullet T * F], \\ &\quad [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet id] \\ &\} \\ I_5 &= goto(I_0, id) = \{[F \rightarrow id \bullet]\} \end{aligned} \quad (6.12)$$

The next step is to compute the goto functions for each of the new states I_1 through I_5 . For instance, from I_1 , only one new state is defined:

$$I_6 = \text{goto}(I_1, +) = \{[E \rightarrow E + \bullet T], [T \rightarrow \bullet T * F], [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet id]\} \quad (6.13)$$

Applying the same method to all possible states and all possible grammar symbols, the other DFA states are:

$$\begin{aligned} I_7 &= \text{goto}(I_2, *) = \{[T \rightarrow T * \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet id]\} \\ I_8 &= \text{goto}(I_4, E) = \{[F \rightarrow (E) \bullet], [E \rightarrow E \bullet + T]\} \\ I_9 &= \text{goto}(I_6, T) = \{[E \rightarrow E + T \bullet], [T \rightarrow T \bullet * F]\} \\ I_{10} &= \text{goto}(I_7, F) = \{[T \rightarrow T * F \bullet]\} \\ I_{11} &= \text{goto}(I_8,) = \{[F \rightarrow (E) \bullet]\} \end{aligned} \quad (6.14)$$

Computing these states is left as an exercise for the reader (tip: use a graphical approach). At first glance, it would seem that more states would be produced when computing the goto functions. This does not necessarily happen because some of the transitions lead to states already seen, i.e., the DFA is not acyclic. Figure 6.2 presents a graphical DFA representation: each state lists its LR(0) items (i.e., NFA states).

If you look carefully at the diagram in figure 6.2, you will notice that in each state some of the items have been “propagated” from other states and others have yet to be processed (having been derived from the former). This implies that what really characterizes each state are its “propagated” items. These are called *nuclear items* and contain the actual information about the state of processing by the parser.

6.3.2 Parse tables

A parse table defines how the parser behaves in the presence of input and symbols at the top of the stack. The parse table decides the action and state of the parser.

exemplo?

A parse table has two main zones: the left one, for dealing with terminal symbols; and the right hand side, for handling non-terminal symbols at the top of the stack. Figure 6.3 shows an example.

Parser DFA and LR(0) items

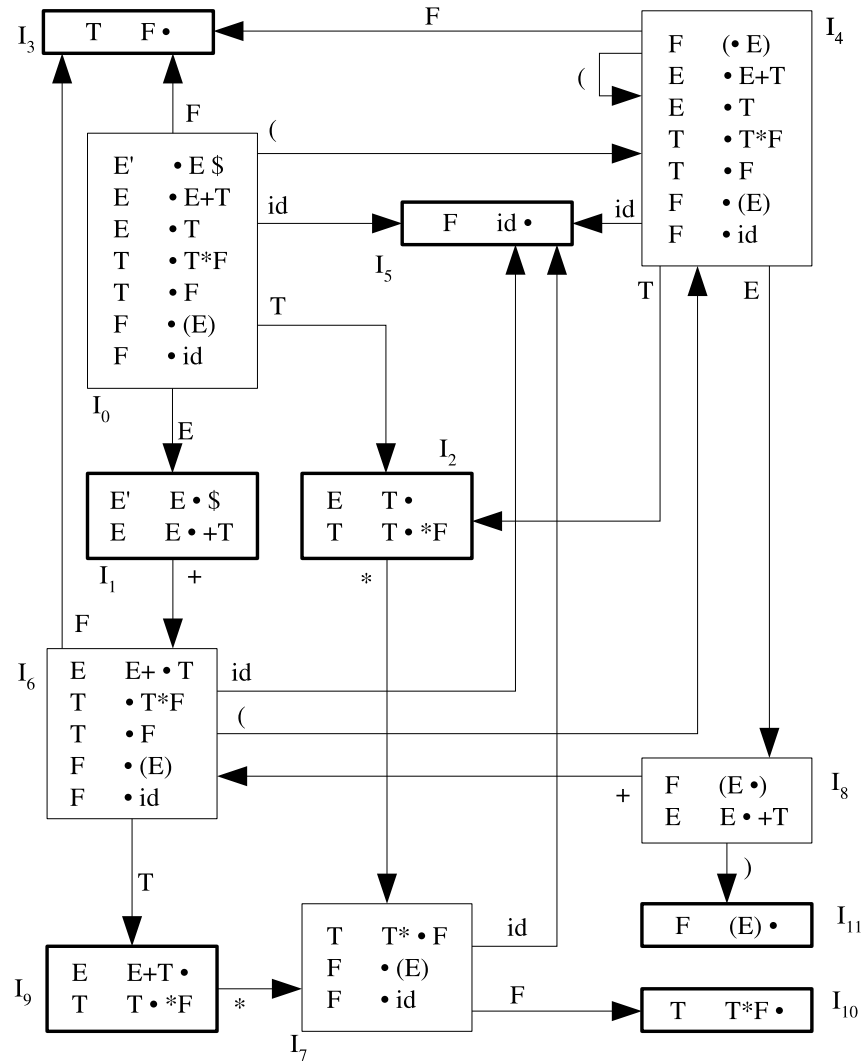


Figure 6.2: Graphical representation of the DFA showing each state's item set. Reduces are possible in states I_1 , I_2 , I_3 , I_5 , I_9 , I_{10} , and I_{11} : it will depend on the actual parser whether reduces actually occur.

Parse table

Figure 6.3: Example of a parser table. Note the column for the end-of-phrase symbol.

6.3.3 LR(0) parsers**6.3.4 SLR(1) parsers****6.3.5 Handling conflicts****6.4 LALR(1) Parsers****6.4.1 LR(1) items****6.4.2 Building the parse table****6.4.3 Handling conflicts****6.4.4 How do parsers parse?**

Once we have a parse table, we can start parsing a given language.

6.5 Compressing parse tables

The total number of steps a parser need to recognize a given phrase of some input language depends on the size of the parse table: the larger the table, the larger the number of steps needed.

Fortunately, parse tables can be compressed. Compression is achieved by noticing that, in some parser states, the parser only reduces a given rule, changing state immediately afterwards. In other cases, the parser does nothing except pushing a state into the stack before doing something meaningful. That is, if in a state the input is not considered, and no two reductions are different, then that state is a good candidate for being eliminated. Once a state is eliminated, the table entries that would cause the state machine to reach it have to be changed, so that the equivalent actions are performed.

Table compression for single-reduction lines

Figure 6.4: exemplo de acções L unitárias

A second type of optimization is possible by considering states where only reductions exist, but considering one to be the default (i.e., the most frequent is chosen). This compression is achieved by adding a new column to the parse table and writing, in the line corresponding to the state being optimized, the default reduction: in all other cells in that line, the reduction is erased (it becomes the default); all other entries in the line stay unchanged.

Table compression for quasi-single-reduction lines

Figure 6.5: exemplo de acções L quase unitárias

Note that if there are any conflicts in the parse table, it should only be compressed **after** the conflicts have been eliminated.

Table compression and conflicts

Figure 6.6: exemplo de conflitos e compressão

6.6 *Summary*

7

Using Berkeley YACC

In the previous chapter we looked at various grammar types and suitable parsing algorithms. The last chapter also dealt with some semantic aspects, such as attributive grammars. While semantics aspects still have to be presented, some of them will be presented here, since they are important in syntax specifications.

In this chapter we consider grammar definitions for automatic parser generation. The parsers we consider here are LALR(1). Several tools are available for creating this type of parser from grammar definitions, e.g. the one presented in this chapter: Berkeley YACC¹.

Parser generator tools do not limit themselves to the grammar itself: they also consider semantic aspects (as mentioned above). It is in this semantic part that the syntactic tree is built. It is also here the place where a particular programming language, for encoding meaning, is relevant.

7.1 *Introduction*

The Berkeley YACC (byacc) tool is one of a family of related tools, some of which differ only in implementation details. It should, however, be noted that while all members of this family generate LALR(1) parsers, they use different strategies, corresponding to different steps in the parsing algorithm. One example is the difference between byacc and GNU bison: the latter specifies a different final parsing state and always considers an extra transition in the DFA. Bison always introduces the extra transition, even when in YACC-compatible mode.

The most significant members of the family are AT&T YACC, Berkeley YACC, and GNU Bison.

7.1.1 AT&T YACC

AT&T's YACC was developed with C as the working programming language: the parser it generates is written in C and the semantic parts are also to be written in C, for seamless integration with the generated parser.

¹YACC means "Yet Another Compiler Compiler".

7.1.2 Berkeley YACC

7.1.3 GNU Bison

GNU bison is widely used and generates powerful parsers. In this document we use byacc due to its simplicity. In future versions, we may yet consider using bison.

7.1.4 LALR(1) parser generator tools and C++

Although other parser generators could have been considered we chose the above for their simplicity and ease of use. Furthermore, since they were all initially developed for using C as the underlying programming language, it was a small step to make them work with C++. Note that we are not referring to the usual rhetoric that C++ is just a better C: we are interested in that aspect, but we are not limited to it. In particular, the use of C++ is a bridge for using sophisticated object- and pattern-oriented programming with these tools. As with the lexical analysis step, the current chapter presents the use of the CDK library for elegantly deal with aspects of syntactic processing in a compiler.

Other parser generator tools for C++

Comparisons and why byacc...

7.2 Syntax of a Grammar Definition

A byacc file is divided into three major parts: a definitions part; a part for defining grammar rules; and a code part, containing miscellaneous functions (figure 7.1).

Structure of a YACC grammar file	
<i>Miscellaneous Definitions</i>	← See §7.2.1.
%%	← This line must be exactly as shown.
<i>Grammar Rules</i>	← See §7.2.2.
%%	← This line must be exactly as shown.
<i>Miscellaneous Code</i>	← See §7.2.3.

Figure 7.1: General structure of a grammar definition file for a YACC-like tool.

The first part usually contains local (static) variables, include directives, and other non-function objects. Note that the first part may contain C code blocks. Thus, although functions usually are not present, this does not mean that they cannot be defined there. Having said this, we will assume that all functions are *outside* the grammar definition file and that only their declarations are present and that only if actually needed. Other than C code blocks, the first part contains miscellaneous definitions used throughout the grammar file.

The third part is another code section, one that usually contains only functions. Since we assume that all functions are defined outside the grammar definition file, the third part will be empty.

The second part is the most important: this is the rules section and it will deserve the bulk of our attention, since it is the one that varies most from language to language.

7.2.1 The first part: definitions

Definitions come in two flavours: general external definitions, defined in C, for instance; and grammar-specific definitions, defined using the tool's specific language.

7.2.1.1 External definitions and code blocks

External definitions come between `%{` and `%}` (these special braces must appear at the first column). Several of these blocks may be present. They will be copied verbatim to the output parser file.

7.2.1.2 Internal definitions

Internal definitions use special syntax and cover specific aspects of the grammar's definition: (i) types for terminal (tokens) and non-terminal grammar symbols; (ii) definitions of terminals and non-terminals; and (iii) optionally, symbol precedences and associativities;

YACC grammars are attributive grammars and, thus, specific values can be associated with each symbol. Terminal symbols (tokens) can also have attributes: these are set by the lexical analyser and passed to the parser. These values are also known as lexemes. Types for non-terminal symbols are computed whenever a semantic rule (a code block embedded in a syntactic rule) is reduced. Regardless of their origin, i.e., lexical analyser or parser reductions, all values must have a type. These types are defined using a union (à la C), in which various types and variables are specified. Since this union will be realized as an actual C union, the corresponding restrictions apply: one of the most stringent is that no structured types may be used. Only atomic types (including pointers) are allowed.

Types, variables, and the grammar symbols are associated through the specific YACC directives `%token` (for terminals) and `%type` (for non-terminals).

Example code block in a grammar file definitions part

```
%{
#include <some-decl-header-file.h>
#include "mine/other-decl.h"

// This static (global) variable will only be available in the
// parser definition file.
static int var1;

// This variable, on the other hand, will be available anywhere
// in the application that uses the parser.
const char *var2 = "This is some string we need somewhere...";

// Here we have some type we can use as if defined in a regular
// code file.
struct sometype { /* etc. */ };

// Functions declared or defined here have the same access
// constraints as any regular function defined in other code
// files.
void this_is_a_function(int, char*); /* fwd decl */
int even_a_function_is_possible() { /* with code!! */ }
%}
```

Figure 7.2: Various code blocks like the one shown here may be defined in the definitions part of a grammar file: they are copied verbatim to the output file in the order they appear.

Example definition of types union

```
%union {
    char          *s;          // string values
    int           i;           // integer values
    double        d;           // floating point numbers
    SomeComplexType *complex;   // what this...?
    SomethingElse  *other;      // hmmm...
}
```

Figure 7.3: The %union directive defines types for both terminal and non-terminal symbols.

The token declarations also declare constants for use in the program (as C macros): as such, these constants are also available on the lexical analyser side. Figure 7.4 illustrates these declarations. Regarding constant declarations, `STRING` and `ID` are declared as character strings, i.e., they use the union's `s` entry as their attribute and, likewise, `NUMBER` and `INTEGER` are declared as integers, i.e., they use the union's `i` entry. Regarding non-terminal symbols, they are declared as `complex` and `other`, two structured types.

Example definition of symbol types

```
%token<s> STRING ID
%token<i> NUMBER
%type<complex> mynode anothernode
%type<other>   strangestuff
```

Figure 7.4: Symbol definitions for terminals (`%token`) and non-terminals (`%type`).

After processing by YACC, a header file is produced², containing both the definitions for all declared tokens and a type definition of crucial importance: `YYSTYPE`. This type is associated with a variable shared between the lexical analyser and the parser: `yyval`, which the parser accesses using the special syntax `$1`, `$2`, etc., `$$` (see below).

Assuming the above definitions, the C/C++ header file would be as shown in figure 7.5. This file reveals how YACC works and although these are implementation details, understanding them will provide a better experience when using the tool. The first thing to note is that the first declared token is numbered 257: this happens because the first 256 numbers are reserved for automatic single-character tokens, such as `'+'` or `'='`. Note that only single-character tokens are automated, all other must be explicitly named. Single character tokens are just an optimization and are otherwise normal (e.g., they can have types, as any other token).

Note that YACC processes the input definitions without concern for details such as whether all the types used in the union are in fact defined or accessible. This fact should be remembered whenever the header is included, especially is the headers for the used types are not included as well.

Another definition type corresponds to the associativity and precedence definitions for operators. These definitions are actually only useful when the grammar is ambiguous and the parser cannot decide whether to shift or reduce. In fact, if the grammar is not ambiguous, then these definitions are not taken into account at all.

We used the term *operators* because these ambiguities are usually associated with operators and expressions, but there is no restriction regarding the

²By default, this file is called `y.tab.h`, but can be renamed without harm. In fact, some YACC implementations make renaming the file alarmingly easy.

Automatic header file with YACC definitions

```

#define STRING 257
#define ID 258
#define NUMBER 259
#define mynode 260
#define anothernode 261
#define strangestuff 262
typedef union {
    char          *s;          /* string values*/
    int           i;          /* integer values*/
    double        d;          /* floating point numbers*/
    SomeComplexType *complex;  /* what this...?*/
    SomethingElse  *other;     /* hmmm...*/
} YYSTYPE;
extern YYSTYPE yylval;

```

Figure 7.5: YACC-generated parser C/C++ header file: note especially the special type for symbol values, `YYSTYPE`, and the automatic declaration of the global variable `yylval`. The code shown in the figure corresponds to actual YACC output.

association of explicit precedences and associativities: they can be specified for any symbol.

Three types of associativity exist and are specified by the corresponding keywords: `%left`, for specifying left associativity; `%nonassoc`, for specifying that an operator is non-associative; and `%right`, for specifying right associativity.

Precedence is not explicitly specified: operators derive their precedence from their relative positions, with the operators defined first having the lower precedence than the ones defined last. Each line counts for a precedence increase. Operators in the same line share the precedence level.

Figure 7.6 illustrates the specifications for common unary and binary operators.

Precedence and associativity	Precedence and rules
<pre> %left '+' '-' %left '*' '/' %right '^' %nonassoc UMINUS </pre>	<pre> expr : '-' expr %prec UMINUS expr '+' expr expr '-' expr /* other rules */ ; </pre>

Figure 7.6: Precedence and associativity in ambiguous grammars (see also §7.2.2).

Note that precedence increases as defined by the use of parentheses is not defined in this way and must be explicitly encoded in the grammar. Note also that the tokens specified need not be the ones appearing in the rules: keyword

`%prec` can be used to make the precedence of a rule be the same as that of a given token (see figure ??). The first rule of figure ?? states that, if in doubt, a unary minus interpretation should be preferred to a binary interpretation (we are assuming that precedences have been defined as in figure 7.6).

7.2.2 The second part: rules

This section contains the grammar rules and corresponding semantic code sections. This is the section that will give rise to the parser's main function: `yyparse`.

7.2.2.1 Shifts and reduces

In the LALR(1) parser algorithm we discussed before, the parser carries out two types of action: shift and reduce. Shifts correspond to input consumption, i.e., the parser reads more data from the input; reductions correspond to recognition of a particular pattern through a bottom-up process, i.e., the parser assemble a group of items and is able to tell they are equivalent to a rule's left side (hence, the term reduce).

Regarding reductions: any rule whose head (left hand part) does not appear in any other rule's right hand side means that rule will never be reduced. If a rule is never reduced, then the corresponding semantic block will never be executed.

7.2.2.2 Structure of a rule

A rule is composed by a head (the left hand side), corresponding to the construction to be recognized, and a right hand side corresponding to the items to be matched against the input, before reducing the left hand side.

Reductions may be performed from different right hand side hypotheses, each corresponding to an alternative in the target language. In this case, each hypothesis is considered independent from the others and each will have its own semantic block. Figure 7.7 shows different types of rules: `statement` has several hypothesis, as does `list`. Note that `list` is recursive (a somewhat recurring construction).

The simple grammar in figure 7.7 uses common conventions for naming the symbols: uppercase symbols represent terminals and lowercase ones non-terminals. As any other convention, this one is mostly for human consumption and useless for the parser generator. The grammar shown recognizes several statements and also lists of statements. "Programs" recognized by the grammar will consist of print statements and assignments to variables. An example program is shown in figure 7.8.

Rule definition examples		
<hr/>		
statement :	PRINT NUMBER	{ /* something to print */ }
	PRINT STRING	{ /* something similar */ }
	PRINT ID	{ /* yet another print */ }
	ID '=' NUMBER	{ /* assigning something */ }
	ID '=' STRING	{ /* ah! something else */ }
	;	
list :	statement ';' { /* single statement */ }	
	list statement ';' { /* statement list */ }	
	;	

Figure 7.7: Examples of rules and corresponding semantic blocks. The first part of the figure shows a collection of statements; the second part shows an example of recursive definition of a rule. Note that, contrary to what happens in LL(1) grammars, there is no problem with left recursion in LALR(1) parsers.

Rule definition examples	
<hr/>	
<pre> print l; i = 34; s = "this is a long string in a variable"; print s; print i; print "this string is printed directly, without a variable"; </pre>	
<hr/>	

Figure 7.8: Example of a program accepted by the grammar defined in figure 7.7.

7.2.2.3 The grammar's start symbol

The top grammar symbol is, by default, the symbol on the left hand side of the first rule in the rules section. If this is not the desired behaviour, the programmer is able to select an arbitrary one using the directive `%start`. This directive should appear in the definitions section (first part). Figure 7.9 shows two scenarios: the first will use `x` as the top symbol (it is the head of the first rule); the second case uses `y` as the top symbol (it has been explicitly selected).

Default scenario	Explicit choice
<pre>%{ /* miscellaneous code */ }% %union { /* symbol types */ } %% x : a b y a ; a : 'a' ; y : b a x b ; b : 'b' ; %% /* miscellaneous code */</pre>	<pre>%{ /* miscellaneous code */ }% %union { /* symbol types */ } %start y %% x : a b y a ; a : 'a' ; y : b a x b ; b : 'b' ; %% /* miscellaneous code */</pre>

Figure 7.9: Start symbols and grammars. Assuming the two tokens `'a'` and `'b'`, the same rules recognize different syntactic constructions depending on the selection of the top symbol. Note that the non-terminal symbols `a` and `b` are different from the tokens (terminals) `'a'` and `'b'`.

The first grammar in figure 7.9 recognizes strings like `a b b a a b b . . .`, while the second recognizes strings like `b a a b b a . . .`. Note that the only difference from the first grammar definition to the second is in the selection of the start symbol (i.e., the last symbol to be reduced).

7.2.3 The third part: code

The code part is useful mostly for writing functions dealing directly or exclusively with the parser code or for exclusive use by the parser code. Of course, it is possible to write **any** code in this section. In fact, in some cases, the `main` function for tests is directly coded here.

In this document, however, we will not use this section for any code other than for functions used only by the parser, i.e. static functions. The reason for this procedure is that it is best to organize the code in well-defined blocks, instead of grouping functions by their accidental proximity or joint use. This aspect assumes greater importance in object-oriented programming languages than in C (even though the remark is also valid for this language). Thus, we will try and write classes for all concepts we can identify and the parser will use objects created from those classes, instead of using private code.

In addition to recognizing a language's grammar, the parser also computes several actions whenever a reduce takes place. This is in addition to its internal functions. Thus, whenever a reduce occurs, the corresponding semantic block is executed; this always happens, even if the programmer does not provide a semantic block. In that case, the executed block is a default one that makes the attribute associated with the rule's left symbol equal to the attribute of the first symbol on the rule's right hand side, i.e., $\{ \$\$ = \$1; \}$.

7.3 Handling Conflicts

By their very nature, LALR(1) parsers are vulnerable to conflicts, i.e., while a grammar may be written without problems, there is no guarantee that a special kind of parser exists for that grammar. This is to say that the algorithm for constructing a parser is unable to avoid conflicts posed by the grammar's special circumstances. In the case of LR parsers, these circumstances occur in two cases. The first is when the parser is unable to decide whether to shift, i.e., consume input; or reduce, i.e., with the information it already possesses, build a new non-terminal symbol. The second occurs when the parser is confronted with two possible reductions.

referir a teoria

Theoretically, these situations correspond to cells in the parse table which have more than an entry. Although, in truth, a parser is said not to exist in the presence of ambiguous tables, we saw that certain procedures allow the conflict to be lifted and processing to go on. It should be recalled that this procedure corresponds to using a different grammar (one for which a parser can be built), which may behave differently from the original. For this reason, it is very important that no grammar in which conflicts exist be used for semantic analysis. The results may be unreliable or, the most common case, plainly wrong.

falar de levantamento de conflitos e de regras de boa escrita

Since YACC must always produce a parser, even for unsuitable grammars. If it is the case that conflicts exist, then YACC will still produce the parser, but it will also signal the presence of problems, both in the error log and, if selected, in the state map (`y.output`). What happens, though, to the conflict? If it was a shift/reduce conflict, then YACC will generate a shift and ignore the reduce; if it is a reduce/reduce conflict, then YACC will prefer to reduce the first rule, e.g. if a conflict exists between reductions of rules 7 and 13, YACC will prefer to reduce rule 7 rather than rule 13.

7.4 Pitfalls

Syntactic analysis is like a minefield: on top it is innocently harmless, but a misstep may do great harm. This is true of syntactic rules and their associated meanings, i.e., how semantics is "grafted" onto syntactic tree nodes.

Compilers and programming languages use what is known as compositional semantics, i.e., the meaning of the whole is directly derived from the meaning of the parts. This is not the case, in general, with natural languages, in which fixed phrases may have widely different meanings from what their words would imply: consider, for instance, the first phrase of this section...

That semantics depends on syntax does not mean however that for a given meaning the same structures have to be present. If this were the case, there would be a single compiler per programming language and a single structural representation. What we address here is the problem of representing, in syntax, structures that have no semantic correspondence and are, thus, completely useless (and probably wrong as well).

Consider the following example...

falar do erro da ligação do else com elsif (sintacticamente é aceite; semanticamente é errado: "lógica booleana com 3 valores!"). apontar para a semântica.

não entrar em pormenores

(falar disto só depois, na semântica?)

7.5 Summary

In this chapter we presented the YACC family of LALR(1) parser generators.

Syntax of a Grammar Definition, i.e., structure of a YACC file

Conflicts

pitfalls

Syntactic Analysis

Case

This chapter describes the application of the syntactic processing theory and tools to our test case, the *compact* programming language.

8.1 *Introduction*

8.1.1 Chapter structure

8.2 *Actual grammar definition*

8.2.1 Interpreting human definitions

How to translate the human-written definitions into actual rule.

8.2.2 Avoiding common pitfalls

How to write write robust rules and avoid common mistakes.

8.3 *Writing the Berkeley yacc file*

As seen before, a yacc file is divided into three major parts: a definitions part; a part for defining grammar rules; and a code part, containing miscellaneous functions.

For the compact language, we will leave the third and last part empty and will concentrate, rather, on the other two. Of these, the first will be briefly approached, since it is almost the same as the one seen for our small example (§??).

The rules section deserves the bulk of our attention, since it is the one that varies most from language to language. As seen before (§??), this section contains various types of information:

- A union for defining types for terminal (tokens) and non-terminal grammar symbols;

- Definitions of terminals and non-terminals;
- Optionally, symbol precedences and associativities;
- Rules and semantic code sections.

8.3.1 Selectiong the scanner object

```
#define yylex scanner.yylex
CompactScanner scanner(g_istr, NULL);

#define LINE scanner.lineno()
```

8.3.2 Grammar item types

```
%union {
    int                i;          /* integer value */
    std::string        *s;          /* symbol name or string literal */
    cdK::node::Node    *node;       /* node pointer */
    ProgramNode        *program;    /* node pointer */
    cdK::node::Sequence *sequence;
};
```

8.3.3 Grammar items

```
%token <i> CPT_INTEGER
%token <s> CPT_VARIABLE CPT_STRING
%token WHILE IF PRINT READ PROGRAM END

%nonassoc IFX
%nonassoc ELSE
%left CPT_GE CPT_LE CPT_EQ CPT_NE '>' '<'
%left '+' '-'
%left '*' '/' '%'
%nonassoc UMINUS

%type <node> stmt expr
%type <program> program
%type <sequence> list
```

8.3.4 The rules

```
program : PROGRAM list END      { syntax = new ProgramNode(LINE, $2); }
      ;

stmt    : ';'                  { $$ = new cdK::node::Nil(LINE); }
      | PRINT CPT_STRING ';'  { $$ = new cdK::node::String(LINE, $2); }
      | PRINT expr ';'        { $$ = new PrintNode(LINE, $2); }
```

```

| READ CPT_VARIABLE ';'
{
  $$ = new ReadNode(LINE, new cdk::node::Identifier(LINE, $2));
}
| CPT_VARIABLE '=' expr ';'
{
  $$ = new AssignmentNode(LINE, new cdk::node::Identifier(LINE, $1),
                          $3);
}
| WHILE '(' expr ')' stmt { $$ = new WhileNode(LINE, $3, $5); }
| IF '(' expr ')' stmt %prec IFX { $$ = new IfNode(LINE, $3, $5); }
| IF '(' expr ')' stmt ELSE stmt
{
  $$ = new IfElseNode(LINE, $3, $5, $7);
}
| '{' list '}' { $$ = $2; }
;

list : stmt { $$ = new cdk::node::Sequence(LINE, $1); }
| list stmt { $$ = new cdk::node::Sequence(LINE, $2, $1); }
;

expr : CPT_INTEGER { $$ = new cdk::node::Integer(LINE, $1); }
| CPT_VARIABLE { $$ = new cdk::node::Identifier(LINE, $1); }
| '-' expr %prec UMINUS { $$ = new cdk::node::NEG(LINE, $2); }
| expr '+' expr { $$ = new cdk::node::ADD(LINE, $1, $3); }
| expr '-' expr { $$ = new cdk::node::SUB(LINE, $1, $3); }
| expr '*' expr { $$ = new cdk::node::MUL(LINE, $1, $3); }
| expr '/' expr { $$ = new cdk::node::DIV(LINE, $1, $3); }
| expr '%' expr { $$ = new cdk::node::MOD(LINE, $1, $3); }
| expr '<' expr { $$ = new cdk::node::LT(LINE, $1, $3); }
| expr '>' expr { $$ = new cdk::node::GT(LINE, $1, $3); }
| expr CPT_GE expr { $$ = new cdk::node::GE(LINE, $1, $3); }
| expr CPT_LE expr { $$ = new cdk::node::LE(LINE, $1, $3); }
| expr CPT_NE expr { $$ = new cdk::node::NE(LINE, $1, $3); }
| expr CPT_EQ expr { $$ = new cdk::node::EQ(LINE, $1, $3); }
| '(' expr ')' { $$ = $2; }
;

```

8.4 Building the Syntax Tree

The syntax tree uses the CDK classes.

The root of the syntactic tree is represented by the global variable `syntax`. It will be defined when the bottom-up syntactic analysis ends. This variable will be used by the different semantic processors.

```
ProgramNode *syntax = 0;
```

8.5 Summary

IV

Semantic Analysis



The Syntax-Semantics Interface

The Visitor design pattern (Gamma et al., 1995) provides the appropriate framework for processing syntactic trees. Using visitors, the programmer is able to decouple the final code generation decisions from the objects that form the syntactic description of a program.

In view of the above, it comes as no surprise that the bridge between syntax and semantics is formed by the communication between visitors, representing semantic processing, and the node classes, representing syntax structure.

9.1 Introduction

Before going into the details of tree processing we will consider the Visitor pattern and its relation with the collections of objects it can be used to “visit”.

9.1.1 The structure of the Visitor design pattern

9.1.2 Considerations and nomenclature

Function overloading vs. functions with different names

Function overloading may be confusing and prone to erroneous assumptions

Different function names are harder to write and, especially, to automate. In addition, most object-oriented programming languages handle function overloading without problems.

9.2 Tree Processing Context

Main compiler function from libcdk (predefined `main`). It assumes that two global function are defined: `yyparse`, for performing syntactic analysis, and `evaluate`, for evaluating the syntactic tree. Evaluation is an abstract process that starts with the top tree node and progresses down the tree.

Example code block in a grammar file definitions part

```
int main(int argc, char *argv[]) {
    // processing of command line options

    /* ==== [ INITIALIZE SCANNER AND PARSER ] ==== */

    extern int yyparse();

    if (yyparse() != 0 || errors > 0) {
        std::cerr << errors << " syntax errors in " << g_ifile << std::endl;
        return 1;
    }

    /* ==== [ SEMANTIC ANALYSIS ] ==== */

    extern bool evaluate();

    if (!evaluate()) {
        std::cerr << "Semantic errors in " << g_ifile << std::endl;
        return 1;
    }

    return 0;
}
```

Figure 9.1: Macro structure of the main function. Note especially the syntax and semantic processing phases (respectively, `yyparse` and `evaluate`).

9.3 *Visitors and Trees*

The bridge between syntax and semantics is formed by the communication between visitors, representing semantic processing, and the node classes, representing syntax structure.

The CDK library assumes a special class for representic abstract semantic processors: `SemanticProcessor` (note that it does not belong to the `cdk` namespace. This pure virtual abstract class is the root of all visitors. It must be provided by the programmer. Moreover, it must contain a declaration for processing each node type ever to be processed by any of its subclasses. If this aspect is not taken into account, then the process of semantic analysis will most probably fail. Other than the above precautions, it may be redefined as desired or deemed convenient.

9.3.1 Basic interface

```
class SemanticProcessor {
    //! The output stream
    std::ostream &_os;

protected:
    SemanticProcessor(std::ostream &os = std::cout) : _os(os) {}
    inline std::ostream &os() { return _os; }

public:
    virtual ~SemanticProcessor() {}

public:
    // processing interface

};
```

9.3.2 Processing interface

```
virtual void process(cdk::node::Node *const node, int lvl) = 0;
```

9.4 *Summary*

Semantic Analysis and Code Generation

10.1 Introduction

10.2 Code Generation

10.3 Summary

11

Semantic Analysis Case

This chapter describes the application of the semantic processing theory and tools to our test case, the *compact* programming language.

11.1 Introduction

11.2 Summary



Appendices

A

The CDK Library

A.1 *The Symbol Table*

A.2 *The Node Hierarchy*

A.2.1 Interface

```
#ifndef __parole_morphology__class_2176_drv_H__
#define __parole_morphology__class_2176_drv_H__
#include <DTL.h>
#include <date_util.h>
#include <table.h> /* DTL header for macros */
#include <driver/driver.h> /* {lr:db} defs. */
#include <driver/Meta.h> /* access to metadata tables */
namespace cdk {
    //... etc. etc. ...
}
#endif
```

A.2.2 Interface

```
#ifndef __parole_morphology__class_2608_drv_H__
#define __parole_morphology__class_2608_drv_H__
#include <DTL.h>
#include <date_util.h>
#include <table.h> /* DTL header for macros */
#include <driver/driver.h> /* {lr:db} defs. */
#include <driver/Meta.h> /* access to metadata tables */
namespace cdk {
    namespace node {
        //... etc. etc. ...
    }
}
#endif
```

A.2.3 Interface

```
#ifndef __parole_morphology__class_2621_drv_H__
#define __parole_morphology__class_2621_drv_H__
#include <DTL.h>
```

```
#include <date_util.h>
#include <table.h> /* DTL header for macros */
#include <driver/driver.h> /* {lr:db} defs. */
#include <driver/Meta.h> /* access to metadata tables */
namespace parole {
    namespace morphology {

        }; // namespace morphology
    }; // namespace parole
#endif
```

A.3 *The Semantic Processors*

A.3.1 Cápsula

```
#ifndef __parole_morphology__class_2176_H__
#define __parole_morphology__class_2176_H__
#include <driver/auto/dbdrv/parole/morphology/__class_2176_drv.h>
#endif
```

A.3.2 Cápsula

```
#ifndef __parole_morphology__class_2621_H__
#define __parole_morphology__class_2621_H__
#include <driver/auto/dbdrv/parole/morphology/__class_2608.h>
#include <driver/auto/dbdrv/parole/morphology/__class_2621_drv.h>
#endif
```

A.4 *The Driver Code*

A.4.1 Construtor

```
#define __parole_morphology_MorphologicalUnitSimple_CPP__
#include <parole/morphology/MorphologicalUnitSimple.h>
#undef __parole_morphology_MorphologicalUnitSimple_CPP__
```

B Postfix Code Generator

This chapter documents the reimplementaion of the postfix code generation engine. The original was created by Santos (2004). It was composed by a set of macros to be used with `printf` functions. Each macro would “take” as arguments, either a number or a string.

The postfix code generator class maintains the same abstraction, but does not rely on macros. Instead, it defines an interface to be used by semantic analysers, as defined by a strategy pattern (Gamma et al., 1995). Specific implementations will provide the realization of the postfix commands for a particular target machine.

B.1 Introduction

Like the original postfix code generator, the current abstraction uses an architecture based on a stack machine, hence the name “postfix”, and three registers.

- IP – the instruction pointer – indicates the position of the next instruction to be executed;
- SP – the stack pointer – indicates the position of the element currently at the stack top;
- FP – the frame pointer – indicates the position of the activation register of the function currently being executed.

In some of the following tables, the “Stack” column presents the actions on the values at the top of the stack. Note that only elements relevant in a given context, i.e., that of the postfix instruction being executed, are shown. The notation `#length` represents a set of `length` consecutive bytes in the stack, i.e., a vector. Consider the following example:

a #8 b : a b

The stack had at its top `b`, followed by eight bytes, followed by `a`. After executing some postfix instruction using these elements, the stack has at its top `b`, followed by `a`.

B.2 The Interface

B.2.1 Introduction

The generators predefined in the CDK belong to namespace `cdk::generator`.

The interface is called `Postfix`. The various implementations will provide the desired behaviour.

B.2.2 Output stream

The default behaviour is to produce the text of the generated program to an output stream (default is `std::cout`). Implementations may provide alternative output streams.

In C++, the interface is defined as a pure virtual class. This class does not assume any output stream, but the constructor presents `std::cout` as the default value for the stream.

```
class Postfix {
protected:
    std::ostream &_os;
    inline Postfix(std::ostream &os) : _os(os) {}
    inline std::ostream &os() { return _os; }

public:
    virtual ~Postfix();

public: // miscellaneous
    // rest of the class (mostly postfix instructions: see below)
};
```

Postfix instructions in the following tables have `void` return type unless otherwise indicated.

B.2.3 Simple instructions

Method	Stack		Function / Action
<code>DUP()</code>	a	a a	Duplicates the value at the top of the stack.
<code>INT(int value)</code>		value	Pushes a integer value to the stack top.
<code>SP()</code>		sp	Pushes to the stack the value of the stack pointer.
<code>SWAP()</code>	a b	b a	Exchanges the two elements at the top of the stack.

Method	Stack		Function / Action
ALLOC ()	a	#a	Allocates in the stack as many bytes as indicated by the value at the top of the stack.
Dynamic memory allocation in the stack, equivalent to a call to the C language <code>alloca</code> function, changes the offsets of temporary variables that may exist in the stack when the allocation is performed. Thus, it should only be used when no temporary variables exist, or when the full import of its actions is fully understood.			

B.2.4 Arithmetic instructions

The following operations perform arithmetic calculations using the elements at the top of the stack. Arguments are taken from the stack, the result is put on the stack. The arithmetic operations considered here apply to (signed) integer arguments, natural (unsigned) integer arguments, and to double precision floating point arguments.

Method	Stack		Function / Action
NEG ()	a	-a	Negation (symmetric) of integer value.
ADD ()	b a	b+a	Integer sum of two integer values.
SUB ()	b a	b-a	Integer subtraction of two integer values.
MUL ()	b a	b*a	Integer multiplication of two integer values.
DIV ()	b a	b/a	Integer division of two integer values.
MOD ()	b a	b%a	Remainder of the integer division of two integer values.
UDIV ()	b a	b/a	Integer division of two natural (unsigned) integer values.
UMOD ()	b a	b%a	Remainder of the integer division of two natural (unsigned) integer values.

The following instructions take one or two double precision floating point values. The result is also a double precision floating point value.

Method	Stack		Function / Action
DNEG ()	d	-d	Negation (symmetric).
DADD ()	d1 d2	d1+d2	Sum.
DSUB ()	d1 d2	d1-d2	Subtraction.
DMUL ()	d1 d2	d1*d2	Multiplication.
DDIV ()	d1 d2	d1/d2	Division.

B.2.5 Rotation and shift instructions

Shift and rotation operations have as maximum value the number of bits of the underlying processor register (32 bits in a ix86-family processor). Safe operation for values above is not guaranteed.

These operations use two values from the stack: the value at the top specifies the number of bits to rotate/shift; the second from the top is the value to be rotated/shifted, as specified by the following table.

Method	Stack		Function / Action
ROTL ()	a b	$a \gg r1 < b$	Rotate left.
ROTR ()	a b	$a \gg rr < b$	Rotate right.
SHTL ()	a b	$a < < b$	Shift left.
SHTRU ()	a b	$a > > b$	Shift right (unsigned).
SHTRS ()	a b	$a > > > b$	Shift right (signed).

B.2.6 Logical instructions

The following operations perform logical operations using the elements at the top of the stack. Arguments are taken from the stack, the result is put on the stack.

Method	Stack		Function / Action
NOT ()	a	$\sim a$	Logical negation (bitwise), i.e., one's complement.
AND ()	b a	$b \& a$	Logical (bitwise) AND operation.
OR ()	b a	$b a$	Logical (bitwise) OR operation.
XOR ()	b a	$b \wedge a$	Logical (bitwise) XOR (exclusive OR) operation.

B.2.7 Integer comparison instructions

The comparison instructions are binary operations that leave at the top of the stack 0 (zero) or 1 (one), depending on the result of the comparison: respectively, *false* or *true*. The value may be directly used to perform conditional jumps (e.g., JZ, JNZ), that use the value of the top of the stack instead of relying on special processor registers (*flags*).

Method	Stack		Function / Action
GT ()	b	b>a	"greater than".
	a		
GE ()	b	b>=a	"greater than or equal to".
	a		
EQ ()	b	b==a	"equal to".
	a		
LE ()	b	b<=a	"less than or equal to".
	a		
LT ()	b	b<a	"less than".
	a		
NE ()	b	b!=a	"not equal to".
	a		
UGT ()	b	b>a	"greater than" for natural numbers (unsigned integers).
	a		
UGE ()	b	b>=a	"greater than or equal to" for natural numbers (unsigned integers).
	a		
ULE ()	b	b<=a	"less than or equal to" for natural numbers (unsigned integers).
	a		
ULT ()	b	b<a	"less than" for natural numbers (unsigned integers).
	a		

B.2.8 Other comparison instructions

Method	Stack		Function / Action
DCMP ()	d1	i	Compares two double precision floating point values. The result is an integer value: less than 0, if the d1 is less than d2; 0, if they are equal; greater than 0, otherwise.
	d2		

B.2.9 Type conversion instructions

The following instructions perform elementary type conversions. The conversions are from and to integers and simple and double precision floating point values.

Method	Stack		Function / Action
D2F ()	d	f	Converts from double precision floating point to simple precision floating point.
D2I ()	d	i	Converts from double precision floating point to integer.
F2D ()	f	d	Converts from simple precision floating point to double precision floating point.
I2D ()	i	d	Converts from integer to double precision floating point.

B.2.10 Function definition instructions

B.2.10.1 Function definitions

In a stack machine the arguments for a function call are already in the stack. Thus, it is not necessary to put them there (it is enough not to remove them). When building functions that conform to the C calling conventions (?, ?), those arguments are destroyed by the caller, *after* the return of the callee, using TRASH, stating the total size (i.e., for all arguments). Regarding the callee, it must create a distinct activation register (ENTER or START) and, when no longer needed, destroy it (LEAVE). The latter action must be performed immediately before returning control to the caller.

Similarly, to return values from a function, the callee must call POP to store the return value in the accumulator register, so that it survives the destruction of the invocation context. The caller must call PUSH, to put the accumulator in the stack. An analogous procedure is valid for DPOP/DPUSH (for double precision floating point return values).

Method	Stack		Function / Action
ENTER(size_t val)		fp #val	Starts a function: push the frame pointer (activation register) to the stack and allocate space for local variables, according to the size given as argument (in bytes).
START ()		fp	Equivalent to ENTER(0).

Method	Stack		Function / Action
LEAVE ()	fp ...		Ends a function: restores the frame pointer (activation register) and destroys the function-local stack data.

Method	Stack		Function / Action
TRASH(int n)	#n		Removes n bytes from the stack.
RET()	addr		Returns from a function (the stack should contain the return address).
RETN(int n)	#n addr		Returns from a function, but removes n bytes from the stack after removing the return address. More or less the same as RET()+TRASH(n).

Method	Stack		Function / Action
POP()	a		Removes a value from the stack (to the accumulator register).
PUSH()		a	Pushes the value in the accumulator register to the stack.
DPOP()	d		Removes a double precision floating point value from the stack (to a double precision floating point register).
DPUSH()		d	Pushes the value in the double precision floating point register to the stack.

B.2.10.2 Function calls

Method	Stack		Function / Action
CALL(std::string name)		addr	Calls the named function. Stores the return address in the stack.

B.2.11 Addressing instructions

Note [*4*] that these operations (ADDR, LOCAL) put at the top of the stack the symbol's address, independently of its origin. O endereço pode posteriormente ser utilizado como ponteiro, obtido o valor nesse endereço (LOAD) ou guardar um valor nesse endereço (STORE). No entanto, nas duas últimas situações, devido à frequência com que ocorrem e o número de ciclos de relógio que levam a executar, podem ser substituídas com vantagem pela operações descritas em [*10*].

B.2.11.1 Absolute and relative addressing

Absolute addressing (ADDR) is performed using labels. Relative addressing (LOCAL) requires a frame pointer to work: the frame pointer defines an addressing reference.

Method	Stack	Function / Action
ADDR(std::string name)	addr	Puts the address of the name passed as argument at the top of the stack.

Method	Stack	Function / Action
LOCAL(int offset)	fp+offset	Puts at the top of the stack the address of the local variable, obtained by computing the offset relative to the frame pointer.
The value passed as argument is as follows: greater or equal to 8, means function arguments; equal to 4, means the function's return address; equal to 0, means the frame pointer itself; less than -4, means local variables.		

B.2.11.2 Quick opcodes for addressing

“Quick opcodes” are shortcuts for groups of operations commonly used together. These opcodes may be made efficient by implementing them in different ways than the original set of high-level operations would suggest, i.e., the code generated by ADDRv may be more efficient than the code generated by ADDR followed by LOAD. Nevertheless, the outcome is the same.

Method	Stack	Function / Action
ADDRV(std::string name)	[name]	ADDR(name); LOAD();
ADDRA(std::string name)	a	ADDR(name); STORE();
LOCV(int offset)	[fp+offset]	LOCAL(offset); LOAD();
LOCA(int offset)	a	LOCAL(offset); STORE();

B.2.11.3 Load instructions

The load instructions assume that the top of the stack contains an address pointing to the data to be read. Each load instruction will replace the address at the top of the stack with the contents of the position it points to. Load instructions differ only in what they load.

Method	Stack		Function / Action
LDCHR ()	addr	[addr]	Loads 1 byte (char).
ULDCHR ()	addr	[addr]	Loads 1 byte (without sign) (unsigned char).
LD16 ()	addr	[addr]	Loads 2 bytes (short).
ULD16 ()	addr	[addr]	Loads 2 bytes (without sign) (unsigned short).
LOAD ()	addr	[addr]	Loads 4 bytes (integer – rvalue).
LOAD2 ()	addr	[addr]	Loads a double precision floating point value.

B.2.11.4 Store instructions

Store instructions assume the stack contains at the top the address where data is to be stored. That data is in the stack, immediately after (below) the address. Store instructions differ only in what they store.

Method	Stack		Function / Action
STCHR ()	val addr		Stores 1 byte (char).
ST16 ()	val addr		Stores 2 bytes (short).
STORE ()	val addr		Stores 4 bytes (integer).
STORE2 ()	val addr		Stores a double precision floating point value.

B.2.12 Segments, values, and labels

B.2.12.1 Segments

These instructions start various segments. They do not affect the stack, nor are they affected by its contents.

Method	Function / Action
BSS ()	Starts the data segment for uninitialized values.
DATA ()	Starts the data segment for initialized values.
RODATA ()	Starts the data segment for initialized constant values.
TEXT ()	Starts the text (code) segment.

B.2.12.2 Values

These instructions declare values directly in segments. They do not affect the stack, nor are they affected by its contents.

Method	Function / Action
BYTE (int)	Declares an uninitialized vector with the length (in bytes) given as argument.
SHORT (int)	Declares a static 16-bit integer value.
CHAR (char)	Declares a static character.
CONST (int)	Declares a static integer value.
DOUBLE (double)	Declares a static double precision floating point value.
FLOAT (float)	Declares a static simple precision floating point value.
ID (std::string)	Declares a name for an address [*1*]
STR (std::string)	[*1*]

Note [*1*] that literal values, e.g. integers, may be used in their static form, using memory space from a data segment (or text, if it is a constant), using LIT. On the other hand, only integer literals and pointers can be used in the instructions themselves as immediate values (INT, ADDR, etc.).

B.2.12.3 Labels

These instructions operate directly on symbols and their definition within some segment. They do not affect the stack, nor are they affected by its contents.

Method	Function / Action
ALIGN ()	Forces the alignment of code or data.
LABEL (std::string)	Generates a new label, as indicated by the argument.
EXTRN (std::string)	Declares the symbol whose name is passed as argument as being externally defined, i.e., defined in another compilation module.
GLOBL (const char*, std::string)	Declare a name/label (first argument) with a given type (second argument; see below). Declaration of a name must precede its definition.
GLOBL (std::string, std::string)	void GLOBL (const char*, std::string), but with a different interface.
COMMON (int)	Declares that the name is common to other modules.

In a declaration of a symbol common to more than one module, other modules may also contain common or external declarations. Nevertheless, only one initialized declaration is allowed. Declarations need not be associated with any particular segments.

In a declaration common to several modules, any number of modules may contain common or external declarations, but only one of them may contain an initialized declaration. A declaration does not need to be specified in a specific segment.

B.2.12.4 Types of global names

Global names may be of different types. These functions are to be used to generate the types needed for the second argument of `GLOBL`.

Method	Function / Action
<code>std::string NONE()</code>	Unknown type.
<code>std::string FUNC()</code>	Name/label corresponds to a function.
<code>std::string OBJ()</code>	Name/label corresponds to an object (data).

B.2.13 Jump instructions

B.2.13.1 Conditional jump instructions

Method	Stack	Function / Action
<code>JZ(std::string)</code>	a	Jump to the address of the label passed as argument if the value at the top of the stack is 0 (zero).
<code>JNZ(std::string)</code>	a	Jump to the address of the label passed as argument if the value at the top of the stack is non-zero.

Method	Stack	Function / Action
<code>JGT(std::string)</code>		
<code>JGE(std::string)</code>		
<code>JEQ(std::string)</code>		
<code>JLE(std::string)</code>		
<code>JLT(std::string)</code>		
<code>JNE(std::string)</code>		
<code>JUGT(std::string)</code>		
<code>JUGE(std::string)</code>		
<code>JULE(std::string)</code>		
<code>JULT(std::string)</code>		

B.2.13.2 Other jump instructions

Method	Stack		Function / Action
JMP(std::string)			Unconditional jump to the label given as argument.
LEAP()	addr		Unconditional jump to the address indicated by the value at the top of the stack.
BRANCH()	addr	ret	Invokes a function at the address indicated by the value at the top of the stack. The return value is pushed to the stack.

B.2.14 Other instructions

Method	Stack		Function / Action
NIL()			No action is performed.
NOP()			Generates a null operation (consumes time, but does not change the state of the processor).
INCR(int val)	a	a	Adds val to the value at the position defined by the address at the top of the stack, i.e. [a] becomes [a]+val.
DECR(int val)	a	a	Subtracts val from the value at the position defined by the address at the top of the stack, i.e. [a] becomes [a]-val.

B.3 Implementations

interface above

uml diagram

As should be expected, the classes described here provide concrete implementations for the abstract functions declared in the superclass. Although the main objective is to produce the final (machine- and OS-specific) code, the generators are free to go about it as they see fit. In general, though, each instruction of the stack machine (postfix) will produce a set of instructions belonging to the target machine.

Two example generators are presented here, and provided with the CDK library: a nasm code generator (§B.3.1) and a debug-only generator (§B.3.2).

B.3.1 NASM code generator

This code generator implements the postfix instructions for producing code to be processed by NASM¹ (NASM, n.d.), the Netwide Assembler. NASM is an assembler for the x86 family of processors designed for portability and modularity. It supports a range of object file formats including Linux a.out and ELF, COFF, Microsoft 16-bit OBJ and Win32. The NASM processor is designed to be simple and easy to understand, similar to Intel's but less complex.

The NASM code generator can be used in two basic modes: code-only or code and debug. The debug data provided here is different from the produced by the debug-only generator (see §B.3.2) in that it describes groups of target machine code using as labels the names of postfix instructions.

B.3.2 Debug-only “code” generator

The debug-only generator does not produce executable code for any machine. Instead, it provides a trace of the postfix instructions executed by the postfix code generator associated with a particular visitor from the syntax tree. Needless to say, although the code generator does not actually produce any code, it can be used just like any other code generator.

B.3.3 Developing new generators

The development of new generators corresponds to implementing the `Postfix` interface. The CDK will be able to use any of these new implementations, but it is the implementer who decides the true semantics of each of the implemented operations.

For instance, instead of producing final target machine code, a code generator could produce target machine instructions in logical form, more amenable to further processing, e.g. by optimization algorithms.

B.4 Summary

Note that the code provided with the CDK library is written in standard C++ and will compile almost anywhere a C++ compiler is available. However, note that while a working CDK is a guarantee for a working compiler, this does not mean that the final program will run in that particular environment. For final programs to work in a given environment, final code generators must be provided for that environment. Consider the following example: the CDK, and the rest of the development tools exist in a Solaris environment running on a SPARC machine. If we were to use the NASM generator in that environment, it would work, i.e., it would produce the code it was supposed to produce, but

¹Information on NASM and related software packages may be found at <http://freshmeat.net/projects/nasm/>

for a ix86-based machine. Further confusion would ensue because NASM **can** produce code for ix86-based machines from SPARC-based machines, using the same binary format, both Solaris and Linux – just to give an example – use the ELF () binary format.



C.1 Introduction

The runtime support library is a set of functions for use by programs produced by the compiler. The intent is to simplify code generation by providing library functions for commonly used functions, such as complex mathematical functions, or input and output routines.

In principle, there are no restrictions regarding the programming style or language, as long as the code is binary-compatible with the one produced by the code generator used by the compiler. In the examples provided in this document, and included in the CDK library, the function calling convention adheres to the C calling convention. Thus, in principle, any C-based library or compatible, could be used.

C.2 Support Functions

The support functions provided by the RTS library are divided into three groups, some or even all of which may be unnecessary on a given project (either because the project is simple enough or because the developer prefers some other interface). The groups are as follows:

- Input functions.
- Output functions.
- Floating-point functions.
- Operating system-related functions.

C.3 Summary

D

Glossary

Este capítulo apresenta alguma da terminologia utilizada na dissertação. Alguns dos termos apresentados resultam da tradução de termos utilizados na literatura internacional.

DOM Document Object Model (W3C, 2002). O Document Object Model é uma interface neutra relativamente a plataformas ou linguagens particulares. Esta interface permite acesso dinâmico ao conteúdo, estrutura e estilo de documentos que sigam este padrão.

m4 Processador de macros. GNU m4 possui funções internas para inclusão de ficheiros, execução de comandos, aritmética, etc.

XMI XML Metadata Interchange (OMG, 2002). XMI é um enquadramento para a definição, intercâmbio, manipulação e integração de objectos XML. As normas baseadas em XMI permitem a integração de ferramentas e repositórios (OMG, 2002).

XML Extensible Markup Language (W3C, 2001a) é um formato de texto, simples e flexível, derivado de SGML (ISO 8879) (ISO, 2001).

XSD XML Schema Definition (W3C, 2001b). Os esquemas XML expressam vocabulários partilhados e providenciam formas de definir a estrutura, conteúdo e semântica de documentos XML. Ver www.oasis-open.org/cover/schemas.html.

XSLT XSL Transformations (W3C, 1999) é uma linguagem para transformar documentos XML. A transformação XSLT descreve as regras para transformar uma árvore de entrada numa árvore de saída independente da árvore original. A linguagem permite filtrar a árvore original assim como a adição de estruturas arbitrárias.

Bibliography

- Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company. (ISBN 0-201-10194-7)
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. (ISBN 0-201-63361-2)
- ISO. (2001). *Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. ISO – International Organization for Standardization. (ISO 8879:1986 (standard). Technical committee/subcommittee: JTC 1/SC 34; ISO Standards)
- Nasm, the netwide assembler*. (n.d.). (<http://freshmeat.net/projects/nasm/>)
- OMG. (2002, January). *XML Metadata Interchange (xmi) Specification, v1.2*. (<http://www.omg.org/technology/documents/formal/xmi.htm>)
- Santos, P. R. dos. (2004). *postfix.h*.
- W3C. (1999). *XSL Transformations (XSLT), Version 1.0*. (<http://www.w3.org/TR/xslt>)
- W3C. (2001a). *Extensible Markup Language*. (<http://www.w3.org/XML/>)
- W3C. (2001b). *XML Schema*. (<http://www.w3c.org/XML/Schema>)
- W3C. (2002). *Document object model*. (<http://www.w3.org/DOM/>)

Author Index

Aho, A. V., 3, 38, 95	Johnson, R., 95	Ullman, J. D., 95
Gamma, E., 5, 65, 77, 95	OMG, 95	Vlissides, J., 95
Helm, R., 95	Santos, P. R. dos, 77, 95	W3C, 95
ISO, 95	Sethi, R., 95	

Index

GNU m4, <i>see</i> m4	<i>see</i> JDBC	semantic analysis, 65–71
ISO, 93	lexical analysis, 21–33	SGML, 93
8879, <i>see</i> SGML	m4, 93	syntactic analysis, 37–61
TC37, <i>see</i> TC37	Open Database	XMI, 93
Java Database	Connectivity,	XML, 93
Connectivity,	<i>see</i> ODBC	XSL, 93
		XSLT, 93

