The Object-Oriented Generation of LR Parsing Tables

By

Minchuan Huang

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba
Canada

© July 1996

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-16163-3

Canada

Name ____ ______________________________

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four digit code in the spaces provided.

Computer Science

SUBJECT TERM

0	9	8	4

SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

- Architecture 0729
- Art History 0377
- Cinema 0900
- Dance 0378
- Fine Arts 0357
- Information Science 0723
- Journalism 0391
- Library Science 0399
- Mass Communications 0708
- Music 0413
- Speech Communication 0459
- Theater 0465

EDUCATION

- General 0515
- Administration 0514
- Adult and Continuing 0516
- Agricultural 0517
- Art 0273
- Bilingual and Multicultural 0282
- Business 0688
- Community College 0275
- Curriculum and Instruction 0727
- Early Childhood 0518
- Elementary 0524
- Finance 0277
- Guidance and Counseling 0519
- Health 0680
- Higher 0745
- History of 0520
- Home Economics 0278
- Industrial 0521
- Language and Literature 0279
- Mathematics 0280
- Music 0522
- Philosophy of 0998
- Physical 0523
- Psychology 0525
- Reading 0535
- Religious 0527
- Sciences 0714
- Secondary 0533
- Social Sciences 0534
- Sociology of 0340
- Special 0529
- Teacher Training 0530
- Technology 0710
- Tests and Measurements 0288
- Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

- Language
  - General 0679
  - Ancient 0289
  - Linguistics 0290
  - Modern 0291
- Literature
  - General 0401
  - Classical 0294
  - Comparative 0295
  - Medieval 0297
  - Modern 0298
  - African 0316
  - American 0591
  - Asian 0305
  - Canadian (English) 0352
  - Canadian (French) 0355
  - English 0593
  - Germanic 0311
  - Latin American 0312
  - Middle Eastern 0315
  - Romance 0313
  - Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

- Philosophy 0422
- Religion
  - General 0318
  - Biblical Studies 0321
  - Clergy 0319
  - History of 0320
  - Philosophy of 0322
- Theology 0469

SOCIAL SCIENCES

- American Studies 0323
- Anthropology
  - Archaeology 0324
  - Cultural 0326
  - Physical 0327
- Business Administration
  - General 0310
  - Accounting 0272
  - Banking 0770
  - Management 0454
  - Marketing 0338
- Canadian Studies 0385
- Economics
  - General 0501
  - Agricultural 0503
  - Commerce-Business 0505
  - Finance 0508
  - History 0509
  - Labor 0510
  - Theory 0511
- Folklore 0358
- Geography 0366
- Gerontology 0351
- History
  - General 0578
  - Ancient 0579
  - Medieval 0581
  - Modern 0582
  - Black 0328
  - African 0331
  - Asia, Australia and Oceania 0332
  - Canadian 0334
  - European 0335
  - Latin American 0336
  - Middle Eastern 0333
  - United States 0337
- History of Science 0585
- Law 0398
- Political Science
  - General 0615
  - International Law and Relations 0616
  - Public Administration 0617
- Recreation 0814
- Social Work 0452
- Sociology
  - General 0626
  - Criminology and Penology 0627
  - Demography 0938
  - Ethnic and Racial Studies 0631
  - Individual and Family Studies 0628
  - Industrial and Labor Relations 0629
  - Public and Social Welfare 0630
  - Social Structure and Development 0700
  - Theory and Methods 0344
- Transportation 0709
- Urban and Regional Planning 0999
- Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

- Agriculture
  - General 0473
  - Agronomy 0285
  - Animal Culture and Nutrition 0475
  - Animal Pathology 0476
  - Food Science and Technology 0359
  - Forestry and Wildlife 0478
  - Plant Culture 0479
  - Plant Pathology 0480
  - Plant Physiology 0817
  - Range Management 0777
  - Wood Technology 0746
- Biology
  - General 0306
  - Anatomy 0287
  - Biostatistics 0308
  - Botany 0309
  - Cell 0379
  - Ecology 0329
  - Entomology 0353
  - Genetics 0369
  - Limnology 0793
  - Microbiology 0410
  - Molecular 0307
  - Neuroscience 0317
  - Oceanography 0416
  - Physiology 0433
  - Radiation 0821
  - Veterinary Science 0778
  - Zoology 0472
- Biophysics
  - General 0786
  - Medical 0760

EARTH SCIENCES

- Biogeochemistry 0425
- Geochemistry 0996
- Geodesy 0370
- Geology 0372
- Geophysics 0373
- Hydrology 0388
- Mineralogy 0411
- Paleobotany 0345
- Paleoecology 0426
- Paleontology 0418
- Paleozoology 0985
- Palynology 0427
- Physical Geography 0368
- Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

- Environmental Sciences 0768
- Health Sciences
  - General 0566
  - Audiology 0300
  - Chemotherapy 0992
  - Dentistry 0567
  - Education 0350
  - Hospital Management 0769
  - Human Development 0758
  - Immunology 0982
  - Medicine and Surgery 0564
  - Mental Health 0347
  - Nursing 0569
  - Nutrition 0570
  - Obstetrics and Gynecology 0380
  - Occupational Health and Therapy 0354
  - Ophthalmology 0381
  - Pathology 0571
  - Pharmacology 0419
  - Pharmacy 0572
  - Physical Therapy 0382
  - Public Health 0573
  - Radiology 0574
  - Recreation 0575
  - Speech Pathology 0460
  - Toxicology 0383
- Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

- Chemistry
  - General 0485
  - Agricultural 0749
  - Analytical 0486
  - Biochemistry 0487
  - Inorganic 0488
  - Nuclear 0738
  - Organic 0490
  - Pharmaceutical 0491
  - Physical 0494
  - Polymer 0495
  - Radiation 0754
- Mathematics 0405
- Physics
  - General 0605
  - Acoustics 0986
  - Astronomy and Astrophysics 0606
  - Atmospheric Science 0608
  - Atomic 0748
  - Electronics and Electricity 0607
  - Elementary Particles and High Energy 0798
  - Fluid and Plasma 0759
  - Molecular 0609
  - Nuclear 0610
  - Optics 0752
  - Radiation 0756
  - Solid State 0611
- Statistics 0463

Applied Sciences

- Applied Mechanics 0346
- Computer Science 0984
- Engineering
  - General 0537
  - Aerospace 0538
  - Agricultural 0539
  - Automotive 0540
  - Biomedical 0541
  - Chemical 0542
  - Civil 0543
  - Electronics and Electrical 0544
  - Heat and Thermodynamics 0348
  - Hydraulic 0545
  - Industrial 0546
  - Marine 0547
  - Materials Science 0794
  - Mechanical 0548
  - Metallurgy 0743
  - Mining 0551
  - Nuclear 0552
  - Packaging 0549
  - Petroleum 0765
  - Sanitary and Municipal 0554
  - System Science 0790
- Geotechnology 0428
- Operations Research 0796
- Plastics Technology 0795
- Textile Technology 0994

PSYCHOLOGY

- General 0621
- Behavioral 0384
- Clinical 0622
- Developmental 0620
- Experimental 0623
- Industrial 0624
- Personality 0625
- Physiological 0989
- Psychobiology 0349
- Psychometrics 0632
- Social 0451

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

THE OBJECT-ORIENTED GENERATION OF LR PARSING TABLES

BY

MINCHUAN HUANG

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Minchuan Huang © 1996

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis/practicum, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis/practicum and to lend or sell copies of the film, and to UNIVERSITY MICROFILMS INC. to publish an abstract of this thesis/practicum..

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

Contents

Abstract iii

List of Figures iv

List of Tables v

1 Introduction 1

2 Object-Oriented Analysis and Design 5

2.1 Overview of Object-Oriented Methods 5

2.1.1 Coad & Yourdon's Object-Oriented Analysis Method 6

2.1.2 Booch's Object-Oriented Analysis and Design Method 9

2.2 Rumbaugh's Object-Oriented Modeling Method 13

2.2.1 Object Modeling 14

2.2.1.1 Objects and Classes 15

2.2.1.2 Links and Associations 16

2.2.1.3 Aggregation 17

2.2.1.4 Generation and Inheritance 18

2.2.2 Dynamic Modeling 19

2.2.3 Functional Modeling 21

3 LR Parsing and Parser Generation 24

3.1 LR Parsing Terminology & LR(0) Parser 26

3.2 LR(1) Parser Construction 30

3.2.1 First, Last, Follow Sets 30

3.2.2 SLR(1) Parser 30

3.2.3 LR(1) Parser 31

3.3 LALR(1) Parser Construction 34

4 Object-Oriented Parser Generation 38

4.1 Object Model 39

4.1.1 The Classes "Symbol" and "SymbolSet" 42

4.1.2 The Classes "Rule" and "RuleSet" 43

4.1.3 The Classes "Item" and "ItemSet" 43

4.1.4 The Class "Transition" 44

4.1.5 The Classes "State" and "StateSet" 45

4.2 Functional Model 45

4.3 Implementation Considerations 48

4.3.1 Implementation of SymbolSet, RuleSet, ItemSet & StateSet 48

4.3.2 BitSet Implementation 49

4.3.3 The Hash Function for States 51

4.3.4 The Parser Construction Algorithm 51

4.3.5 Combination of Shift and Reduce 52

4.3.6 Handling of Object Groups 52

4.3.7 Pass by Reference vs. Pass by Value 53

5 Comparison with A Traditional Parser Generator 55

5.1 Running Time on Pascal and Modula-2 Grammars 56

5.2 Function Calls on Pascal and Modula-2 Grammars 57

5.3 Object Size and Executable File Size 58

5.4 Source Size 59

6 Conclusions 60

Appendix A. System User's Guide 62

Appendix B. Test on an ISO Pascal Grammar 66

References 82

Abstract

Object-oriented technology is no longer an obscure research topic but has moved into the mainstream of software development. Many projects in a variety of areas have applied object-oriented technology. In particular, object-oriented compilers have been developed. We have found with interest that object-oriented technology has not yet been applied to parser *generator*s. This thesis experiments with building an LALR(1) parser generator using object-oriented analysis, design, and C++ programming techniques. The work consists of surveying Object-Orientation and LR Parsing and the design and implementation of appropriate classes for such components as symbols, items, item sets, kernel sets, and state graphs, so that they can be conveniently used in the generator implementation. A comparison between this experimental object-oriented parser generator and a traditional parser generator is also made.

List of Figures

Fig. 2.1 Coad&Yourdon notation for a hypothetical sensor system 9
Fig. 2.2 Documentation aspects in Booch's OOD 13
Fig. 2.3 Notation for classes 15
Fig. 2.4 One-to-one association and links 16
Fig. 2.5 Many-to-many association 16
Fig. 2.6 Multiplicity of Associations 17
Fig. 2.7 Notation for aggregation 17
Fig. 2.8 Notation for inheritance 18
Fig. 2.9 Notation for unstructured state diagrams 19
Fig. 2.10 Notation for structured state diagrams 21
Fig. 2.11 Notation for process 22
Fig. 2.12 Notation for data flow 23
Fig. 2.13 Notation for actor & data store 23
Fig. 3.1 Model of an LR parser 25
Fig. 3.2 LR(1) Parsing states & transitions for G' 36
Fig. 3.3 LALR(1) Parsing states & transitions for G' 37
Fig. 4.1 Object Model of Parser Generator 40
Fig. 4.2 Functional model for the parser generator 46
Fig. 4.3 Expansion of *build LR states* process 47

List of Tables

Table 5.1 Running time 56

Table 5.2 Function calls 57

Table 5.3 Object size 58

Table 5.4 Executable file size 58

Table 5.5 Source size 59

Chapter 1

Introduction

Object-oriented technology is no longer an obscure research topic but has moved into the mainstream of software development. Many projects in a variety of areas have applied object-oriented technology. In particular, objected-oriented compilers have been developed [6,15,16]. I have found with interest that object-oriented technology seems, however, not yet to have been applied to parser generators. The objective of this thesis is to build an LALR(1) parser generator using object-oriented analysis, design, and C++ programming techniques. The focus is on the design of appropriate classes for such components as symbols, items, item sets, kernel sets and state graphs, so that they can be conveniently used in the generator implementation.

Several popular object-oriented analysis and design methodologies are surveyed and one of them is chosen to be applied to the construction of the parser generator. A selected LALR(1) parser construction algorithm is also reviewed.

The object-oriented methodologies which will be investigated in chapter 2 are the most popular ones. They include:

1. James Rumbaugh's Object-Oriented Modeling [2]:

   Rumbaugh's Object-Oriented Modeling Techniques(OMT) is model-driven. It uses a declarative model represented by extended ER diagrams, a behavioral model represented by state diagrams, and a process-interaction model represented by data-flow diagrams. The technique also suggests a minimal object-interaction model.

2. Coad & Yourdon's Object-Oriented Analysis [1]:

   Coad & Yourdon view their Object-Oriented Analysis (OOA) methodology as building "upon the best concepts from information modeling, object-oriented programming languages, and knowledge-based systems." Their OOA technique results in a five-layer model of the problem domain, where each layer builds on the previous layer. The five layers are the object layer, structure layer, subject layer, attribute layer, and service layer respectively.

3. Grady Booch's Object-Oriented Analysis & Design Technique [4]:

   Grady Booch presents a method for the development of complex systems based on an object model. The method includes a graphical notation for

object-oriented analysis and design, followed by its process. Booch also examines the pragmatics of object-oriented development, in particular, its place in the software development life cycle and its implications for project management.

LALR(1) parser construction has long been an interesting topic in the parser generation area. The main issue in this sub-area is how to compute LALR(1) look-ahead sets. Several methods of computing LALR(1) lookahead sets have been presented. They inclue:

1. Aho, Sethi, and Ullman's "full LR(1) compression" method [6].
2. Aho, Sethi, and Ullman's "Spontaneous and Propagated lookahead" method [6].
3. DeRemer and Pennello's method presented in "Efficient Computation of LALR(1) Lookahead Sets" [7].
4. Park, Choe and Chang's method presented in "A New Analysis of LALR Formalism" [8].
5. Fred Ives's method presented in "Unifying View of Recent LALR(1) Lookahead Set Algorithms" [9].

The purpose of the thesis is to perform experiments in the application of object-oriented technology to parser construction. Thus it is not necessary to

choose the most efficient algorithm. Futhermore, the most efficient methods are so obscure and hard to implement that they would be too complex and too hard to maintain. As a result they are rarely used. In this thesis, the first method will be reviewed in detail in Chapter 3 and its use in implementing the parser generator is described in Chapter 4.

A comparison between an object-oriented parser generator and traditional parser generation is made in Chapter 5, and important differences are examined. Metrics used for the comparison include:

1. Running time on ISO Pascal and Modula-2 grammars.
2. Number of function calls (direct & indirect calls).
3. Object sizes in bytes and task image size in bytes.
4. Source code size:
    a) Number of lines including and excluding comments.
    b) Number of characters excluding comments.
    c) Number of tokens

Any special benefits or disadvantages of applying object-oriented technology to parser generations are discussed. Every effort is made to minimize the effects of individual programming styles and compilers used.

Chapter 2

Object-Oriented Analysis and Design

Analysis is the study and modeling of a given problem domain, within the context of stated goals and objectives. Analysis focuses on what a system is supposed to do, rather than how it is supposed to do it. An analysis methodology is a set of concepts, techniques, notations, and tools that help guide an analysis process [13]. While the primary goal of object-oriented analysis and design is similar to the traditional software engineering methods, the fundamental distinction is that the traditional methods decompose a system into procedures, whereas OOA/OOD decomposes a system into objects.

2.1 Overview of Object-Oriented Methods

In recent years, numerous object-oriented analysis methods have merged. Among

those methods, three have high popularity. They are Coad & Yourdon's Objected-Oriented Analysis [1], Booch's Object-Oriented Analysis and Design [4], and James Rumbaugh's Object-Oriented Modeling [2]. Rumbaugh's method will be discussed in more detail.

2.1.1 Coad & Yourdon's Object-Oriented Analysis Method

Coad & Yourdon view their OOA methodology as building "upon the best concepts from information modeling, object-oriented programming languages, and knowledge-based systems." Coad & Yourdon's OOA results in a five-layer model of the problem domain, where each layer builds on the previous layers. The layered model is constructed using a five-step procedure[1]:

1. *Identifying objects and class*es. Guidelines for identifying objects and classes are developed. The approach starts by examining the application space to identify the classes and objects forming the basis of the entire application, such as other systems and devices, etc. In the light of this, the system's responsibilities in this domain are analyzed. This includes things such as events remembered, roles played, etc. Investigating the system environment may produce further classes and objects that the system should know about, such as locations and organizational units,

etc. Design notes are made of information that needs to be saved about each object, and what behaviors each object must provide.

2. *Defining structures* is principally done in two different ways. The first, the *general-to-specific* structure (e.g. employee-to-sales manager), is used to capture the inheritance hierachy among the identified classes. The other structure, the *whole-to-part* structure (e.g. car-to-engine), is used to model how an object is part of another object and how objects are grouped into larger categories.

3. *Defining subjects* is done by reorganizing the objects and class into larger units. Subjects are groups of objects and classes. The size of each subject is selected to help readers understand the system through the model. Structures defined earlier can also be used to define subjects. For example, a *general-to-specific* structure can be grouped into one subject.

4. *Defining attributes* is done to identify information and relationships that should be associated with each instance. For each object, the attributes needed to characterize it are identified. The identified attributes are placed at the correct level of the inheritance hierarchy. For example, if a generalization/specialization exists, the common attributes are placed at the high level and the specialization specific attributes are place below.

Any instance connections are also identified by checking previous OOA results or by mapping problem domain relationships.

5. *Defining services* means defining the operations (methods) of each class. This process identifies the object states and defines services such as occur, calculate, monitor and so on that operate on the state. Message connections are used to identify how the objects communicate with messages. A message connection maps one object to another, in which a "sender" sends a message to a "receiver", to get some processing done.

The outcome of Coad&Youron's OOA is documented using a graphical notation (see Figure 2.1) and special templates for the textual description of classes and objects [1]. The whole model is presented in the following five layers:

1. Subject layer - only subjects are presented.
2. Class & Object layer - includes subjects, classes, and objects.
3. Structure layer - structures are added to the Class & Object layer.
4. Attribute layer - attributes are added to the Structure layer.
5. Service layer - services are added to the Structure layer.

Fig. 2.1 combines the object layer, the structure layer, the attribute layer and the service layer. Because the sensor system is so simple, the subject layer, a higher-level overview, is not needed to help the reader understand the system. Note

that only some basic notations are shown in this figure.

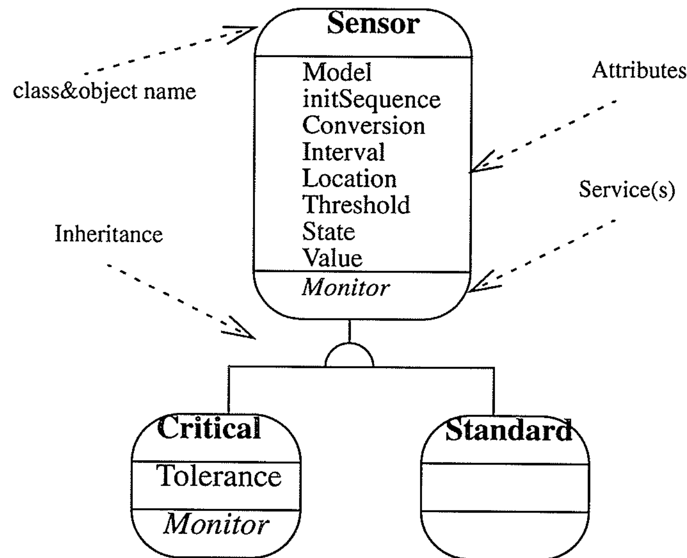


Fig. 2.1 Coad&Yourdon notation for a hypothetical sensor system

2.1.2 Booch's Object-Oriented Analysis & Design Method

Booch views all successful projects as characterized by a strong architectural vision and a well-managed iterative and incremental development life cycle. The development process, according to Booch, should consist of two kinds of elements, *micro* elements and *macro* elements [4]. The *micro process* serves as the framework for an iterative and incremental approach to development. The *macro process* is more closely related to the traditional waterfall lifecycle and is used to control the micro process.

The *micro process* is driven by the stream of scenarios and architectural

products that emerge from the macro process; it presents daily activities of the development team. There are four steps in the *micro process* [4]:

1. *Identifying the classes and objects at a given level of abstraction.* This step discovers key abstractions in the problem domain, finds meaningful classes and objects, and crafts new classes and objects that derive from the solution domain. Key abstractions are usually found by learning the terminology of the problem domain and working in conjunction with domain experts. The data dictionary is the primary product of this step.
2. *Identifying the semantics of classes and objects.* This phase establishes the meanings of the classes and objects already identified. The developer views the objects from the outside and defines the object protocols and investigates how each object may be used by other objects. This part may be highly iterative. Products of this step include data dictionary refinement, a specification for each abstraction, and an interface for each class.
3. *Identifying the relationships among classes and objects.* The previous activities are extended to include the relationships between classes and objects and to identify how these interact with each other. Different types of associations are used, such as inheritance, instantiation, and uses

between the classes. This step also defines static and dynamic semantics of the mechanisms between the objects. Class diagrams, object diagrams, and module diagrams are the main products of this step.

4. *Implementation of classes and objects*. Finally, the classes and objects are examined in detail to determine how to implement them. This includes the selection of data structures and algorithms. Concurrently, a decision should be made on how to use a particular programming language to implement the classes. This step will eventually have products of pseudo or executable code.

The basic philosophy of the *macro process* is that of incremental development. There are five steps in the *macro process* [4]:

1. *Conceptualization* establishes the core requirements for the system; its activity serves as a proof of concept, and so is largely uncontrolled to allow unrestrained innovation. Prototypes are the primary products of this step.

2. *Analysis* provides a model of the system's behavior; primary activities include domain analysis and scenario planning. The formal requirements analysis document and the risk assessment are the main products of this step.

3. *Design* creates an architecture for the implementation and establishes common tactical policies; primary activities include architectural planning, tactical design, and release planning. Products of this step include a description of the architecture and descriptions of common policies for error detection and handling, memory management, data storage management, etc.
4. *Evolution* uses successive refinement to ultimately lead to the production system; primary activities include application of the micro process and change management. The product of this step is a stream of further releases which have successive refinements and enhancements to the preceeding releases.
5. *Maintenance* is the management of post-delivery evolution; primary activities are similar to those of the fourth step, with the addition of managing a list of new tasks which serves as the vehicle for collecting bugs and enhancement requirements.

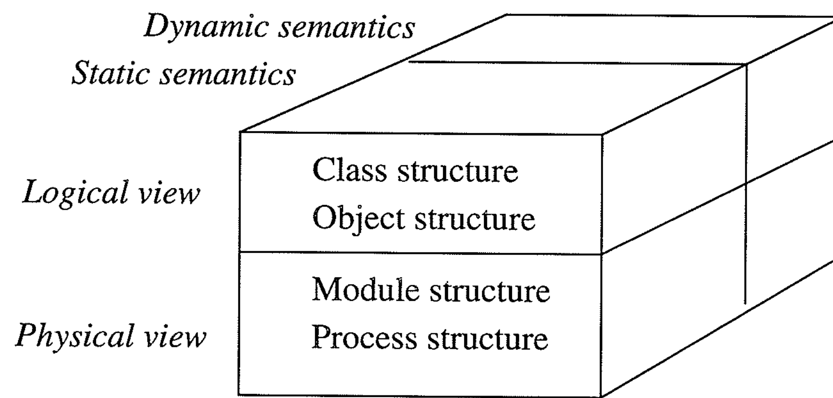


Fig. 2.2 Documentation aspects in Booch's OOD

Booch OOD provides rich diagramming techniques allowing viewing of the model developed from different views (Figure 2.2) [4]. The logical view consists of the *class structure* and the *object structure.* The physical view consists of *module structure* and *process structure.* All these diagrams are formed by the OOD basic notation, which is a static description of the system. In addition, two dynamic diagrams, the *state transition diagram* and the *timing diagram*, are used to describe event occurrences between objects.

2.2 Rumbaugh's Object-Oriented Modeling Method

James Rumbaugh, et al. [2] present an object-oriented approach to software development based on *modeling objects* from the real world and then using the model to build a language-independent *design* organized around those objects. The approach includes a set of object-oriented concepts and a language-independent graphical

notation, the Object Modeling Technique (OMT), that can be used to analyze problem requirements, design a solution to the problem, and then implement the solution using a programming language and/or database.

The Object Modeling Technique combines three views of the modeling system. The *object model* describes the static, structural (data) aspects of a system with classes and their relationships. The *dynamic model* represents the temporal, behavioral, (control) aspects of a system with events and states of the objects. The *functional model* represents the computational (functional) aspects of a system with data flow diagrams. Rumbaugh states "A typical software procedure incorporates all three aspects: It uses data structures (object model), it sequences operations in time (dynamic model), and it transforms values (functional model). Each model contains references to entities in other models. For example, operations are attached to objects in the object model, but are more fully expanded in the functional model."[2]

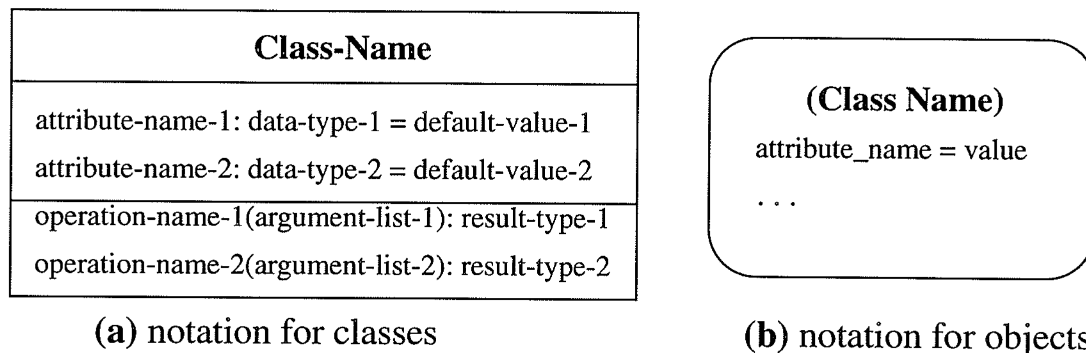
2.2.1 Object Modeling

An object model captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects [2]. Only the most important concepts including

class, link, association, generalization, and inheritance are reviewed.

2.2.1.1 Objects and Classes

Objects and classes are the basic concepts in object-oriented applications. OMT uses the following notation for classes and objects.



(a) notation for classes **(b)** notation for objects

Fig. 2.3 Notations for classes and objects

Fig 2.1(a) illustrates object modeling notation for classes. A class is represented by a box which may have as many as three regions. The regions contain, from top to bottom: a class name, a list of attributes, and a list of operations. Each attribute name may be followed by optional details such as type and default value. Each operation name may be followed by optional details such as an argument list and a result type. Attributes and operations are optional in the diagrams depending on the level of detail desired. Fig 2.1(b) is the general notation for objects. It is a rounded box including the name of the class from which the object is instantiated.

Attribute names and the values assigned to them may or may not be included depending on the level of detail desired.

2.2.1.2 Links and Associations

Links and associations are used to establish relationships among objects and classes. A link connects two or more objects to show the relationships between those objects. An association describes a group of links with common structure and common semantics.

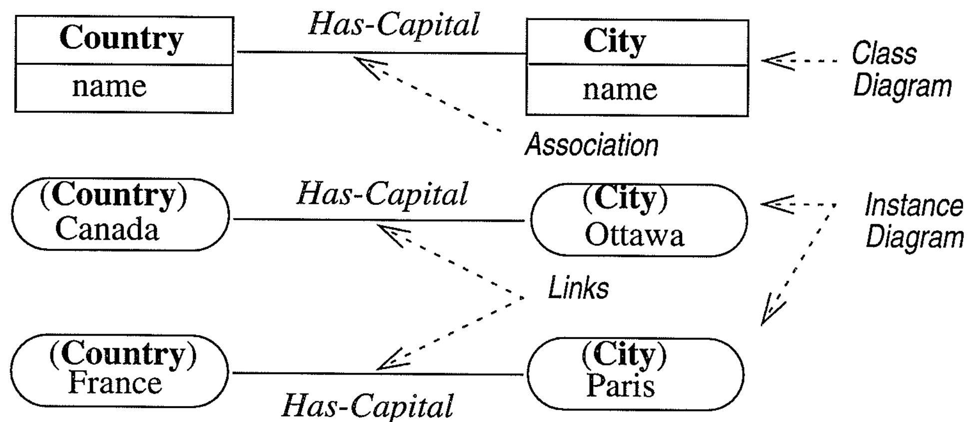


Fig. 2.4 One-to-one association and links

Fig. 2.2 shows a one-to-one association and corresponding links. Each association in the class diagram corresponds to a set of links in the instance diagrams, just as each class corresponds to a set of objects. Each country has a capital city. *Has-Capital* is the name of the association. The OMT notation for an association is a line between classes. A link is drawn as a line between objects.

Multiplicity of the association specifies how many instances of one class may relate to each instance of another class. Associations may be binary, ternary or

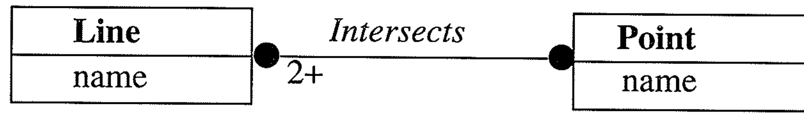


Fig. 2.5 Many-to-many association

higher order and has multiplicity.

The association in figure 2.3 exhibits many-to-many multiplicity. A line may have zero or more intersection points. An intersection point may be associated with two or more lines. Generally speaking, OMT uses the following notation for multiplicity of association.

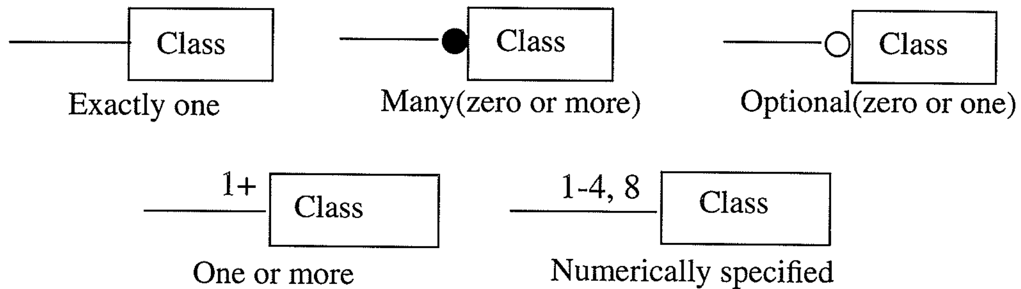


Fig. 2.6 Multiplicity of Associations

2.2.1.3 Aggregation

Aggregation is the "part-whole" or "a-part-of" relationship. The notation of aggregation is similar to association, except for a small diamond indicating the assembly

end of the relationship. Figure 2.5 represents the general notation for aggregation.

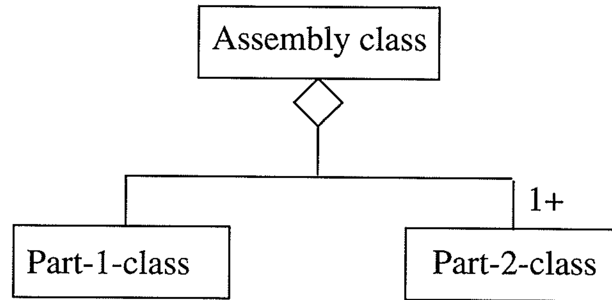


Fig. 2.7 Notation for aggregation

2.2.1.4 Generalization and Inheritance

Generalization and inheritance are fundamental concepts in object-oriented languages. Generalization is a useful construct for both conceptual modeling and implementation. During conceptual modeling, generalization can be used to organize classes into a hierarchical structure based on similarities and differences. During implementation, inheritance facilitates the sharing of code or behaviors common to a collection of classes. In OMT, the term *generalization* refers to the relationship between a class (the superclass) and one or more refined versions of it (the subclass). The term *inheritance* refers to the mechanism of obtaining attributes and operations from the generalized structure[2].

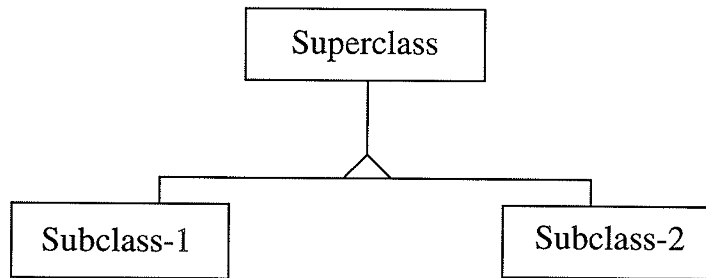


Fig. 2.8 Notation for inheritance

The notation for generalization and inheritance is a triangle connecting a superclass to its subclasses. As figure 2.6 shows, the superclass is connected by a line to the apex of the triangle and the subclasses are connected by lines to a horizontal bar attached to the base of the triangle.

2.2.2 Dynamic Modeling

The dynamic model represents control information: the sequences of events, states, and operations that occur within a system of objects[2]. The major dynamic modeling concepts are *events*, which represent external stimuli, and *states*, which represent values of objects. The *state diagram* is a graphical representation of finite state machines. OMT emphasizes the use of events and states to specify control, rather than as algebraic constructs.

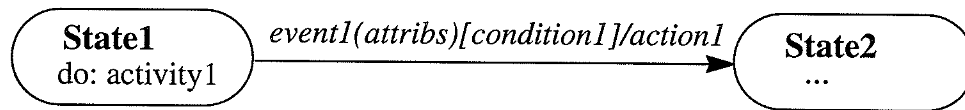


Fig. 2.9 Notation for unstructured state diagrams

Figure 2.7 illustrates the notation for unstructured state diagrams. A state name is written in boldface within a rounded box. An event name is written on a transition arrow and may optionally be followed by one or more attributes within parentheses. A condition may be listed within square brackets after an event name. An activity is indicated within a state box by the keyword "*do:*" followed by the name or description of the activity. An action is indicated on a transition following the event named by a "/" character followed by the event name. All these constructs are optional in state diagrams [2].

State diagrams can be structured to permit concise descriptions of complex systems. There are two ways of structuring state diagrams: generalization and aggregation. Generalization is equivalent to expanding nested activities. It allows an activity to be described at a high level, then expanded at a lower level by adding details, similar to a nested procedure call. In addition, generalization allows states and events to be arranged into generalization hierarchies with inheritance of common structure and behavior, in a fashion similar to inheritance of attributes and

operations in classes. Aggregation allows a state to be broken into orthogonal components, with limited interactions among them, in a fashion similar to an object aggregation hierarchy. Aggregation is equivalent to concurrency of states. Concurrent states generally correspond to object aggregations, possibly an entire system, that have interacting parts. Figure 2.8 is a general structured state diagram notation.

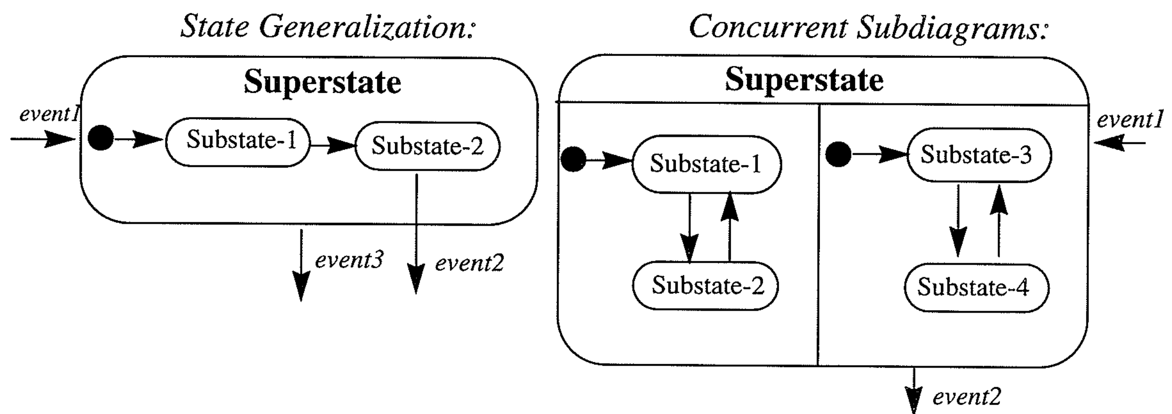


Fig. 2.10 Notation for structured state diagrams

2.2.3 Functional Modeling

The functional model describes computations within a system. The functional model shows how output values in a computation are derived from input values, i.e. the operations of the objects, independent of the order in which the values are computed.

The functional model consists of multiple data flow diagrams showing the

flow of values from external inputs through operations and internal data stores, to external outputs. Data-flow diagrams do not show control or object-structure information; these belong to the dynamic and object model. A data-flow diagram is a graph of *processes,* which transform data, *data flows*, which move data, *actor* objects, which produce and consume data, and *data-store* objects, which store data passively[2].

The notation for a process is an ellipse containing a description of the transformation, usually its name. Each process has a fixed number of input and output data arrows, each arrow carries a value of a given type [2] (see Figure 2.9).

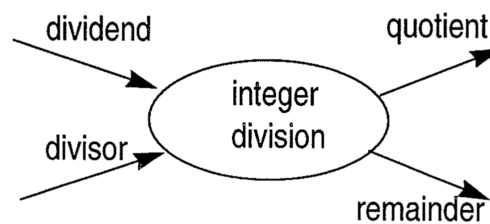


Fig. 2.11 Notation for process

The notation for a data flow is an arrow between the producer and the consumer of the data value. The arrow is labeled with a description of the data, usually its name or type. As figure 2.10 shows, there are three forms of data flows, composition of data values (a), decomposition of data values (b), and duplication of data values (c)[2].

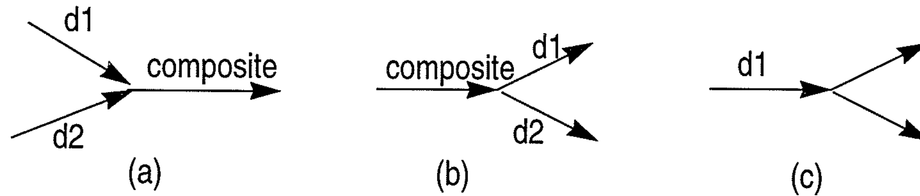


Fig. 2.12 Notation for data flow

There are two kinds of objects in the data flow, called *actor* objects and *data store* objects.

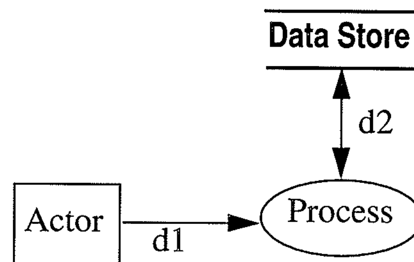


Fig. 2.13 Notation for actor & data store

An *actor* is an active object which produces or consumes data and is drawn as a rectangle. Arrows between the actor and the diagram are inputs and outputs of the diagram.

A *data store* is a passive object which only stores data for later access and is drawn as a pair of parallel lines containing the name of the store. Input arrows indicate information or operations that modify the stored data; this includes adding elements, modifying values, or deleting elements. Output arrows indicate information retrieved from the store. This includes retrieving the entire value or some component of it. Figure 2.11 shows the notations for actor and data store concepts [2].

Chapter 3

LR Parsing and Parser Generation

In this chapter, an overview of LR parsers and how to build them is given. LR(k) parsers are so called because they operate in the manner of bottom up, scanning the input from ***L***eft to right, producing the reverse of a ***R***ightmost derivation, and use ***k*** characters of unscanned input to produce deterministic behavior [5].

An LR parser is generally composed of an input, an output, a driver program, and a parsing table that has two parts, *action* and *goto.* All LR parsers have two basic actions, *shift* and *reduce*, and two kinds of final state, either *accept* or *error*, although the algorithms used to generate parsers can be quite different. The driver program is identical for all LR parsers; what is different are the parsing tables which change from grammar to grammar. Aho, Sethi, and Ullman [7] have explained the typical operation of an LR parser (Figure 3.1). The parsing program reads characters from an input buffer one at a time. The program uses a stack to

store a string of the form $s_0X_1s_1X_2s_2... X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a parser *state*. Each parser state summarizes the information contained in the stack below it, and the combination of the parser state on top of the stack and the current input symbol are used to index the parsing table and determine the shift/reduce parsing decision.

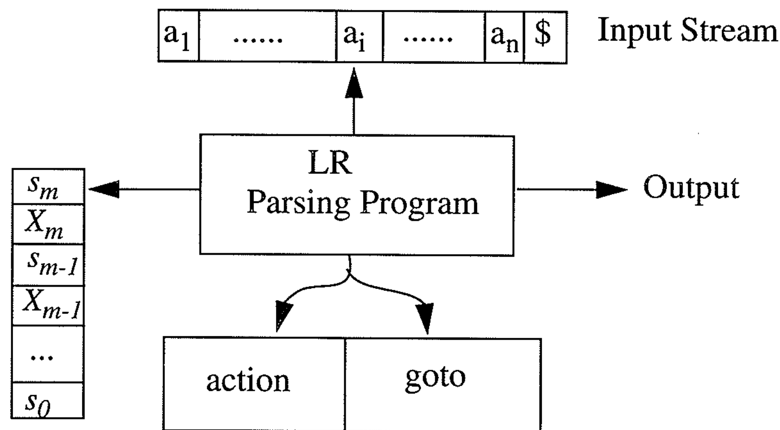


Fig. 3.1 Model of an LR parser

The parsing table consists of two parts, a parsing action function *action* and a goto function *goto*. The program determines s_m, the state currently on the top of the stack, and a_i, the current input symbol, and then consults *action*$[s_m, a_i]$, the parsing action table entry for state s_m and input a_i, which can have one of the following four values:

1. "Shift s". Shift an input symbol and push it onto the stack and go to state s. This action will continue until the right part of a production appears on

the stack.

2. "Reduce R". Reduce by a grammar production. i.e. pop symbols from the stack matching the right part of a production and push the left part symbol of that rule onto the stack.

3. "Accept" or "Error". Accept or reject the input.

The *goto* function takes a state and grammar symbol as arguments and produces a state. Therefore, the *goto* function is a state transition function.

Parsing table construction is crucial to LR parser generation. To review LR parsing table construction methods, it is first necessary to introduce LR parsing terminology.

3.1 LR Parsing Terminology & LR(0) Parser

The following notation and terminology will be used in this chapter and following chapters:

- A,B,C, ... — Upper-case letters early in the alphabet represent a single nonterminal.
- ..., X, Y, Z — Upper-case letters late in the alphabet represent a single terminal or nonterminal.
- a,b,c, ... — Lower-case letters early in the alphabet represent a single termi-

nal.

- ..., x,y,z — Lower case letters late in the alphabet represent strings of terminals, or the empty string.
- $\alpha,\beta,\gamma, \ldots$ — Lower case greek letters represent strings of terminals or non-terminals or the empty string.
- ε — epsilon represents the empty string, the string with no symbols.
- α_i — subscripted letter means the *i*th symbol in α.
- $\Rightarrow$ — directly derives.
- $\Rightarrow^*$ — derives in zero or more steps.
- An *LR Item* is a production with an item dot somewhere in the right hand side. For example, $[\ A \rightarrow \alpha \bullet \beta\ ]$. Either α or β may be ε..
- A *State* is a set of LR items.
- *Kernel Items* are items generated by advancing the dot in the items of predecessor states.
- *Item-Set Closure* is the generation of the full item set from the kernel set, as presented below.
- *Closure Items* are items generated by item-set closure.
- *Complete Items* are items with the item dot at the end of the right part. (e.g. $[\ B \rightarrow \beta \bullet]$).

Let us first consider LR(0) parser generation. The steps for constructing an LR(0) parsing automaton are as follows:

1. Augment the grammar by adding a rule 0: $S' \rightarrow \$S\$$ where \$ is the begin and the end of input marker and S' is a new start symbol that appears in no other rules.
2. Create state 0 with the kernel item set of [$S' \rightarrow \$ \bullet S\$, \emptyset$]
3. Maintain a list of unprocessed states called USL, and add state 0 to USL.
4. Select next state R from USL while USL is not empty.
   - 4.0. Select next state R from USL
   - 4.1. Perform Item-Set closure on R
     - 4.1.1. Maintain a list UIL (unprocessed item list) of unprocessed items. Initialize UIL with R's kernel item set.
     - 4.1.2. Remove items from UIL until encountering the next item of the form [$B \rightarrow \beta \bullet C\gamma$] (i.e. a nonterminal C follows the item dot).
     - 4.1.3. For every rule $C \rightarrow \delta$ in the grammar, if the item [$C \rightarrow \bullet \delta$] is not already in R, add it to R and to UIL.
     - 4.1.4. Remove [$B \rightarrow \beta \bullet C\gamma$] from UIL, if UIL is not empty. Go back to step 4.1.2.
   - 4.2. Compute the GOTO set (the set of destination states)

4.2.1. For each symbol X such that R contains an item $[A \rightarrow \alpha \bullet X\beta]$ create a new state T = goto(R, X). For each item $[B \rightarrow \alpha \bullet X\delta]$ in R (i.e. an X follows the dot), add a kernel item $[B \rightarrow \alpha X \bullet \delta]$ to T (i.e. move the dot one position right.)

4.3. Remove R from USL.

When a state in an LR(0) parser built for a grammar G contains more than one item, and one of those items is a complete item then the state is not LR(0) consistent and the Grammar G is not an LR(0) grammar. The reason is that there could be two kinds of LR conflicts in the state:

1. Reduce-Reduce Conflicts:

The state has at least two complete items, $[A \rightarrow \alpha \bullet]$ and $[B \rightarrow \beta \bullet]$ and we cannot decide which one should be reduced.

2. Shift-Reduce Conflicts:

The state has at least one complete item, $[A \rightarrow \alpha \bullet]$ and at least one other incomplete item $[B \rightarrow \beta \bullet \gamma]$ for some γ which is not ε. We cannot decide if we should reduce α to A or shift γ_1 to try to recognize B.

A solution to resolving these conflicts that often works is to look ahead at the next input symbol to make a decision. One such parser that uses lookahead is the LR(1) parser.

3.2 LR(1) Parser Construction

In order to discuss LR(1) parsing, it is useful to define the functions FIRST, LAST, and FOLLOW which are now described:

3.2.1 FIRST, LAST, and FOLLOW Sets

FIRST(X) is defined as the set of all terminals and nonterminals that can be the first symbol of any sentential form derivable from X:

$$FIRST(X) = \{ Y \mid X \Rightarrow^* Y\alpha \}$$

LAST(X) is the set of all terminals and nonterminals that can be the last symbol of any sentential form derivable from X:

$$LAST(X) = \{ Y \mid X \Rightarrow^* \alpha Y \}$$

Notice that the symbol itself belongs to its own FIRST set and LAST set. The FIRST sets and LAST set are used to efficiently compute FOLLOW sets. For each nonterminal A, FOLLOW(A) is the set of symbols that can follow A in some derivation from S':

$$FOLLOW(A) = \{ Y \mid S' \Rightarrow^* \alpha A Y \beta \}$$

3.2.2 SLR(1) Parser

Among LR(1) parser families, there is a so-called *simple LR(1)*, i.e. SLR(1), parser

which attaches FOLLOW(B), as a lookahead set, to each complete item $[B \rightarrow \beta \cdot]$ and if the next input symbol is not in the lookahead set, the corresponding reduce action is not allowed. For a shift-reduce conflict $[B \rightarrow \beta \cdot]$ and $[C \rightarrow \gamma \cdot X\delta]$, if X does not belong to FOLLOW(B) then the conflict is resolved. For the reduce-reduce conflict $[B \rightarrow \beta \cdot]$ and $[C \rightarrow \gamma \cdot]$, if the intersection of FOLLOW(B) and FOLLOW(C) is empty, then the conflict is resolved. If all conflicts in the LR(0) automaton can be resolved in this way, then the grammar is SLR(1). Essentially, SLR(1) parser generators analyze the grammar and individual states, but not the paths between the states, to determine the lookahead sets. Therefore, SLR is the weakest member of the LR(1) family in terms of the number of grammars for which it succeeds. The most general form of LR(1) parser is presented next.

3.2.3 LR(1) Parser

Aho, Sethi, and Ullman [7] present the following LR(1) construction algorithm. Note that the general form of an LR(1) item is $[A \rightarrow \alpha \bullet \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is the lookahead symbol, and assume that G' is an augmented grammar ("augment" is defined on page 28).

function *closure*(I)

begin

```
    repeat
        for  each item [A→α•Bβ, a] in I,
             each production B→γ in G'
             and each terminal b in FIRST(βa)
              such that [B→•γ , b] is not already in I do
                   add [B→•γ , b] to I;
    until no more items can be added to I;
    return I
end;
function goto (I, X)
begin
    let J be the set of items [A→αX•β, a] such that [A→α•Xβ, a],
     for any A, α,  β, and a, is in I;
    return closure(J)
end;
procedure items(G')                          -- Top level procedure
begin
    C := { closure({[S' →$•S$,  Ø ]}) };
    repeat
         for each set of items I in C and each grammar symbol X
             such that goto(I, X) is not empty and not in C do
                   add goto(I, X) to C
    until no more sets of items can be added to C
end
```

With these routines, the LR(1) parsing table can be constructed by the following steps [7]:

1. Construct C = $\{ I_0, I_1, \ldots, I_n \}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from item set I_i. The parsing actions for state i are determined as follows:
   a) If [A→α•aβ, b] is in I_i and $goto(I_i, a) = I_j$, then set *action[i, a]* to "*shift j*". Here, a is required to be a terminal.
   b) If [A→α•, a] is in I_i, A is not S', then set *action[i, a]* to "reduce A→α"
   c) If [S'→$S•$] is in I_i, then set *action*[i, $] to "accept".

If a conflict results from the above rules, the grammar is said not to be LR(1), and the algorithm is said to fail.

3. The *goto* transitions for state i are determined as follows: If $goto(I_i, A) = I_j$, then $goto[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error" entries.
5. The initial state of the parser is the one constructed from the set containing item [S' → $•S$, Ø].

Essentially, this LR(1) parser generator analyzes paths between the states. So, the lookahead set is not simply the FOLLOW set. Thus, LR(1) parsers are more

powerful than SLR(1) parsers, but they are also much larger. A compromise can be made by constructing an LALR(1) parser, which is discussed in the following section.

3.3 LALR(1) Parser Construction

LALR stands for *lookahead*-LR. The LALR(1) method is often used in practice because the tables generated by it are considerably smaller than LR(1) tables and most common syntactic constructs of programming languages can be expressed conveniently by an LALR(1) grammar. The same is almost true for SLR(1) grammars, but there are a few constructs that cannot be conveniently handled by SLR(1) techniques.

An LALR(1) parsing table is constructed by merging those LR(1) sets having identical "core" items. Core items are items without any lookahead information attached. The detailed algorithm as described by Aho, Sethi, and Ullman [7] is as follows:

1. Construct $C = \{ I_0, I_1, \dots , I_n \}$, the collection of sets of LR(1) items.
2. For each core item set present among the set of LR(1) items, find all sets having the same core items, and replace these sets by the set with common core items and the union of lookahead sets.

3. Let $C' = \{ J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as constructing LR(1) parse tables. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The *goto* table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the core items of $goto(I_1,X)$, $goto(I_2,X), \ldots, goto(I_k,X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core items. Let K be the union of all sets of items having the same core items as $goto(I_1,X)$. Then $goto(I_1,X) = K$.

Note that, when constructing LALR(1) parsers, the merging of states with common core items can never produce a shift/reduce conflict that was not present in one of the original states, since shift actions depend only on the core item, not the lookahead. Nevertheless, it is possible that a merging will produce a reduce/reduce conflict. The following simple example shows sample LR(1) states and how they are combined into LALR(1) states.

Let an augmented grammar G' be:

0. S' → $ S $
1. S → F F
2. F → f F
3. F → g

The LR(1) states would be as shown in Fig 3.2. Where I_i are closure item sets generated from the following corresponding kernel sets:

K_0: $S' \rightarrow \$ \bullet S \$, \emptyset$
K_1: $S' \rightarrow \$ S \bullet \$, \emptyset$
K_2: $S \rightarrow F \bullet F, \{ \$ \}$
K_3: $F \rightarrow f \bullet F, \{ f, g \}$
K_4: $F \rightarrow g \bullet, \{ f, g \}$
K_5: $S \rightarrow F F \bullet, \{ \$ \}$
K_6: $F \rightarrow f \bullet F, \{ \$ \}$
K_7: $F \rightarrow g \bullet, \{ \$ \}$
K_8: $F \rightarrow f F \bullet, \{ f, g \}$
K_9: $F \rightarrow f F \bullet, \{ \$ \}$

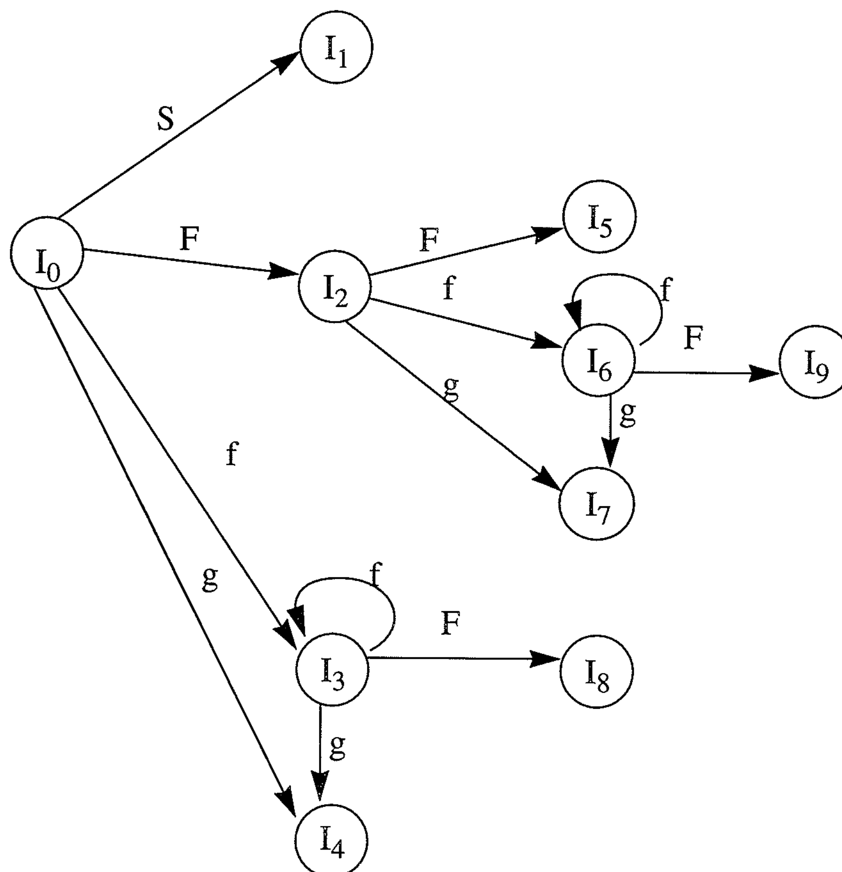


Fig. 3.2 LR(1) Parsing states & transitions for G'

Notice that K_3 has the same core items as state K_6, state K_4 has the same core items as state K_7, and state K_8 has the same core items as state K_9.

By applying the LALR(1) construction algorithm, we can merge K_3 with K_6, K_4 with K_7, and K_8 with K_9 respectively and get the following LALR(1) kernel item sets:

K_0: S' →$ • S $, Ø
K_1: S' → $ S • $, Ø
K_2: S → F • F, { $ }
K_3: S → F F• , { $ }
K_4: F → f • F , { $, f, g }
K_5: F → g• , { $, f, g }
K_6: F → f F• , { $, f, g }

Therefore, the corresponding LALR(1) states would be as follows:

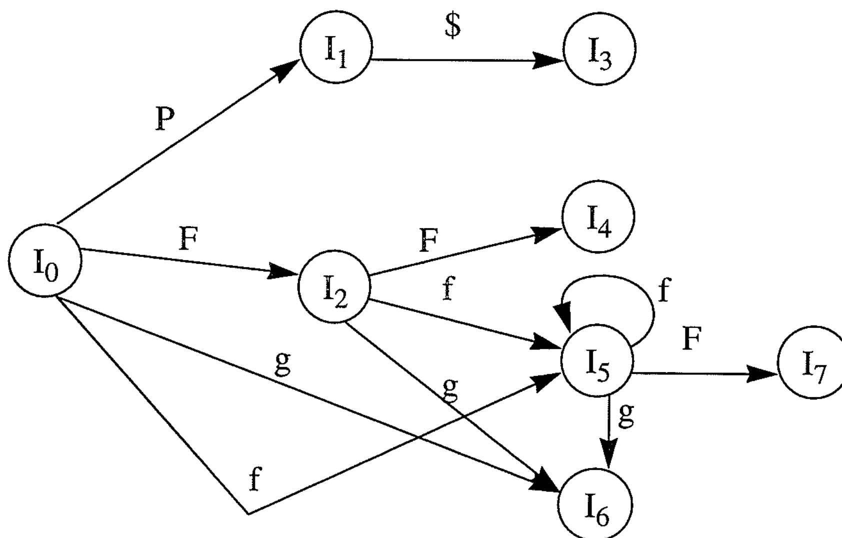


Fig. 3.3 LALR(1) Parsing states & transitions for G'

Chapter 4

Object-Oriented Parser Generation

The notations and algorithms of object-oriented analysis & design and LR parsing just reviewed will now be applied. The object-oriented development of an LALR(1) parser generator is discussed and the concepts and techniques of Rumbaugh's Object-Oriented Modeling Technique (OMT) are used to depict the structure of the resulting system.

While OMT is composed of three kinds of models, the object model, the dynamic model, and the functional model, only two of those models, the object model and the functional model, will be presented because non-interactive programs such as compilers and parser generators, have a trivial dynamic model. Their purpose is simply to compute a function. Thus the functional model is the main model for such programs. The object model is also important for any problem with nontrivial data structures.

Design and implementation considerations will also be discussed in this chapter.

4.1 Object Model

The purpose of the object model is to describe the structure of objects in the system. Because object-oriented programming emphasizes building a system around objects rather than procedures, the object model is the most important one in object-oriented analysis. Figure 4.1 gives a simplified description of the object model for the OO parser generator created. More detail is given in later diagrams.

As Figure 4.1 shows, the system has a *symbol-set,* a *rule-set,* and a *state-set.* The *symbol-set* consists of one or more *symbol*s. Each *symbol* is associated with the first *rule* which has that symbol as the left part. The *rule-set* consists of one or more *rule*s which have two kinds of information, one left *symbol* and zero or more right *symbol*s. The *state-set* consists of one or more parser *state*s. The number of parser states increases during the execution of the system. Every *state* has an *item set* and some *transition* information which includes the reduce information or the goto information used by the parser based on the lookahead symbol. So, the *transition* information contains the input symbol information and either the rule applied to reduce an item or the state to be shifted to. An *item set* is composed of *items*

which includes information about a *Rule*. Worthy of mention is that the boxes in the diagram are classes rather than objects even though the model name is called the

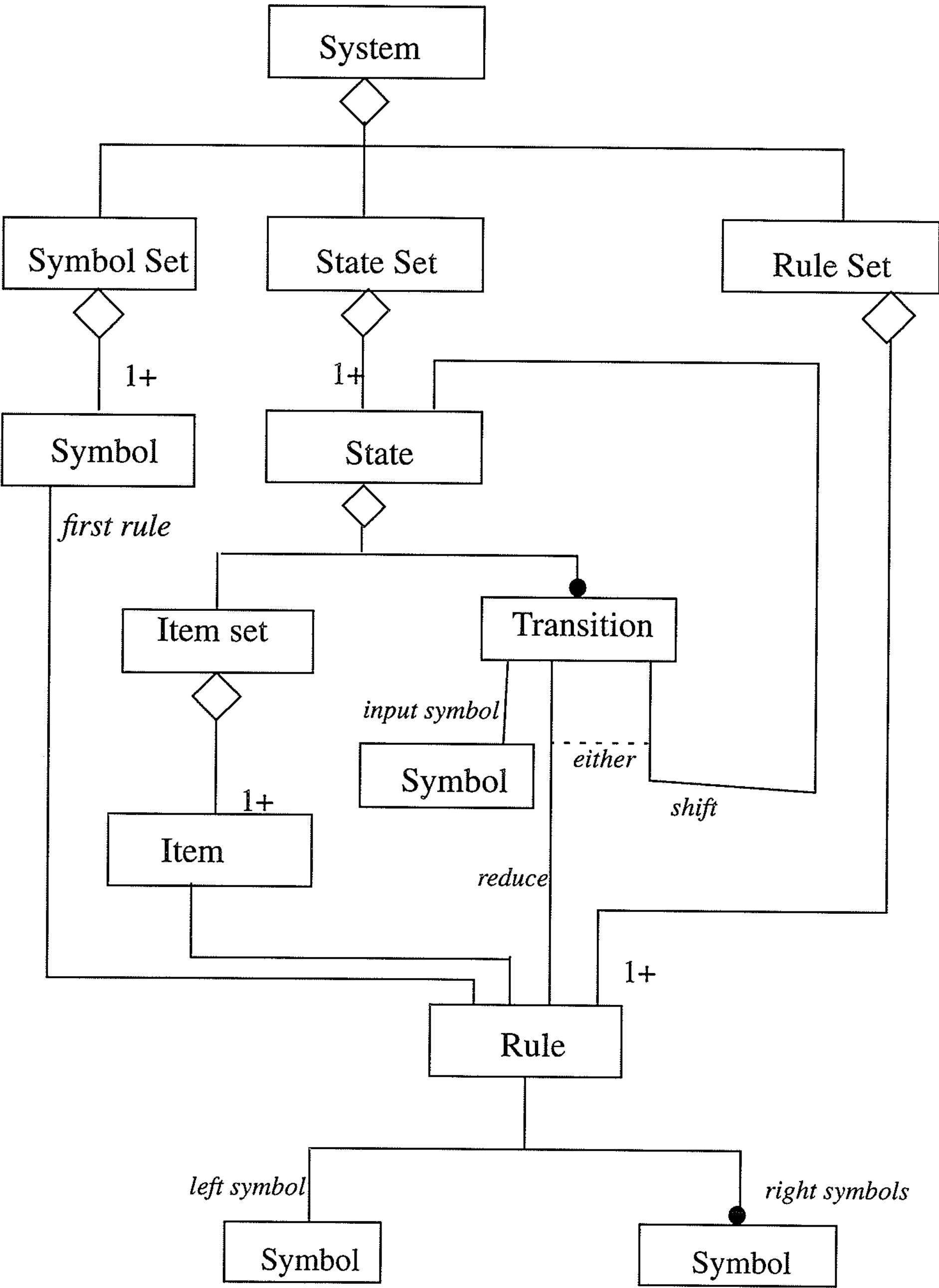


Fig. 4.1 Object Model of Parser Generator

object model. Although this diagram shows only the top-level classes, it does present the overall structure of parser construction.

Symbol
symText : String tnt : Bool nullable : Bool firstSet : BitSet lastSet : BitSet followSet : BitSet
GetFirstAlt IncludeFirst

Fig 4.1.1 Class Symbol

SymbolSet
numSym : integer
Look DoEmpty DoFirst DoLast DoFollow

Fig 4.1.2 Class SymbolSet

Rule
numRHS : integer
DoString

Fig 4.1.3 Class Rule

RuleSet
numRules : integer

Fig 4.1.4 Class RuleSet

Item
pos : integer fSymLr : BitSet fSymLalr : BitSet
SameCoreItem IncludeLookahead operator = operator = = PrintItem

Fig 4.1.5 Class Item

ItemSet
numItems : integer maxItems : integer stateHash : integer
operator = operator = = AddItem DoClosure SameCoreItemSet MergeCoreItemSet

Fig 4.1.6 Class ItemSet

Transition
action : char
AddTrans InsertLrReduces InsertLalrReduces

Fig 4.1.7 Class Transition

State
stateNum : integer
SameCoreState PrintState

Fig 4.1.8 Class State

StateSet
numLrStates : integer numLalrStates: integer lastState : integer stateTbl : array of int hashTbl : array
DoLrStates DoLalrStates PrintStates . . .

Fig 4.1.9 Class StateSet

4.1.1 The Classes "Symbol" and "SymbolSet"

Symbol is an essential class (Fig 4.1.1) in the system. Symbol objects contain information about a particular symbol in the grammar, including the attributes: `tnt`, a boolean which indicates whether it is a terminal or nonterminal; `nullable`, which indicates whether the symbol is nullable ($C \Rightarrow+ \varepsilon$) or not; `firstSet`, `lastSet` and `followSet` store the FIRST set, LAST set, and FOLLOW set of the symbol respectively; and `symText` which stores the name of the symbol. The methods in the class "Symbol" are `GetFirstAlt`, which gets the first rule with this symbol as the left part, and `IncludeFirst`, which is used for lookahead computation.

The class "SymbolSet" (Fig 4.1.2) serves as a symbol table that holds all symbols of the input grammar. It includes a data member `numSym` which keeps track of the total number of input symbols. This class does not have much specific data-member information, but is used to manage symbols. The methods in this class include: `Look`, which either finds a particular symbol in the symbol table, if it exists, or inserts it into the symbol table; `DoEmpty`, which computes whether the symbol is nullable or not; `DoFirst`, which computes the first set for every symbol; `DoLast`, which computes the last set for every symbol, and `DoFollow`, which computes the follow set for every symbol.

4.1.2 The Classes "Rule" and "RuleSet"

The class "Rule" (Fig 4.1.3) represents the rules in the grammar. The characteristics of the grammar rules are defined by the attribute `numRHS`, the number of symbols on the right hand side of the rule; and by the associations that every *rule* has, such as a left part *symbol* and a number of right part *symbol*s as indicated in Fig. 4.1. Method `DoString` is used for lookahead computation.

The class "RuleSet" (Fig 4.1.4) holds all the *rule*s of the input grammar. Attribute `numRules` indicates the total number of production rules. The RuleSet object is a object which has no operations of its own but merely stores data. This kind of object is called a *data store* in OMT.

4.1.3 The Classes "Item" and "ItemSet"

The class "Item" (Figure 4.1.5) represents the notion of item in LR parser theory. An item has a related rule (as indicated in Fig 4.1) with associated dot position information (attribute `pos`), and two lookahead sets — the `fSym` attribute is for LR parsing and the `fSymLalr` attribute is for LALR parsing. The methods in the class "Item" include `PrintItem`, which reports the information in item objects; the `operator =`, which assign one item to another; the `operator ==`, which

compares two items; `SameCoreItem`, which compares two items disregarding lookahead sets; and `InlcudeLookahead`, which is used for lookahead computation.

The class "ItemSet" (Figure 4.1.6) is a collection of Item objects. The member `stateHash` stores the hash value of the state in which this ItemSet object is contained. The data member `numItems` stores the number of items in a specific ItemSet object. The data member `maxItems` holds the maximum possible number of items, and is a static member. The value of `maxItems` depends on the particular grammar. The top-level methods are the `operator =`, which copies an item set; the `operator ==`, which does item set comparison; and `SameCoreSet`, which compares two item sets disregarding lookahead information. In addition, `AddItem` adds an item to an item set, and `DoClosure` computes the closure set for the given item set.

4.1.4 The Class "Transition"

The Transition class (Figure 4.1.7) stores state transition information as the attribute `action`, which can be either reduce or shift, and has associations with *symbol*s, *rule*s, and *state*s as indicated in Fig 4.1.

Method `AddTrans` is defined to store shift transition information, and

method InsertReduces is defined to add reduce information to transition objects.

4.1.5 The Classes "State" and "StateSet"

The class "State" (Figure 4.1.8) defines state (or kernel) information. The attribute stateNum stores the state number assigned by the parser generator. The method SameCoreState is used to check if two states are mergeable or not during LALR computation, and PrintState prints essential state information.

The class "StateSet" is defined to manage dynamically created state objects during parser construction. The top-level methods of the StateSet class include: DoLrStates, which computes LR states; DoLalrStates, which constructs LALR(1) states based on the LR(1) states; and printStates, which prints essential state information. Note that in the actual implementation, there are more private methods and sub-methods which are not presented here because they are not needed to understand the system.

4.2 Functional Model

The functional model shows the computation and the functional derivation of the

data values in it without indicating how, when, or why the values are computed.

This section describes the high-level functional model for parser generation. Figure 4.2 is the top level functional model and Figure 4.3 is the expansion of the *build-LR-states* process.

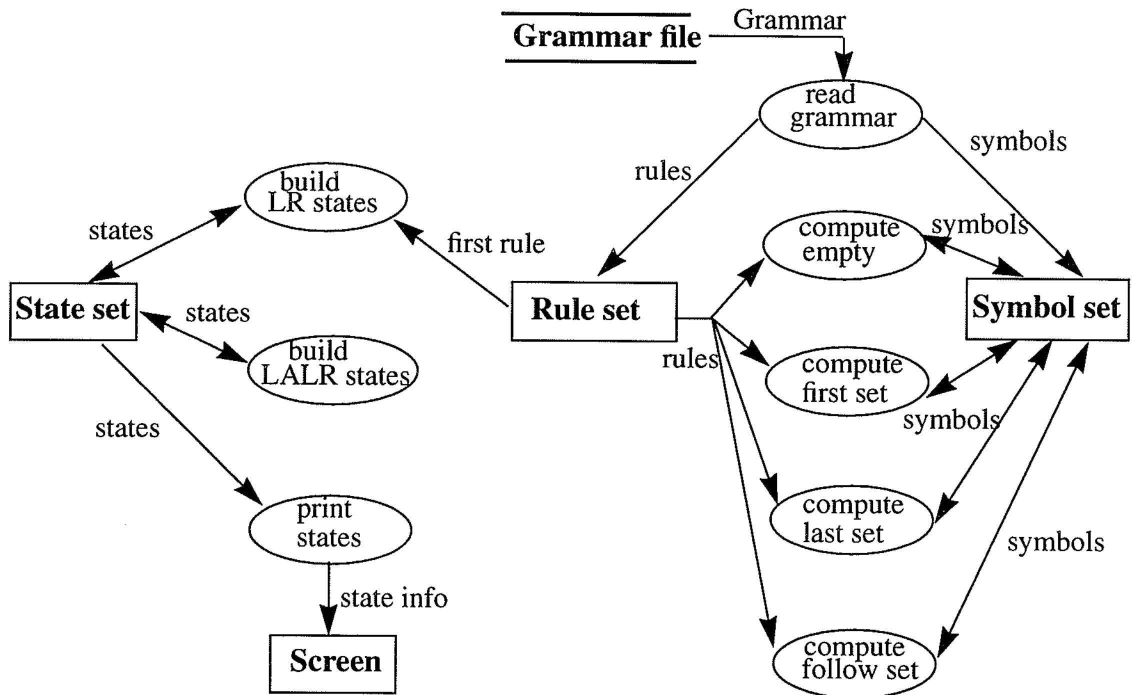


Fig 4.2 Functional model for the parser generator

Figure 4.2 shows how the *read grammar* process reads a grammar from an external file and then generates a set of *rule* objects (RuleSet) and *symbol* objects (SymbolSet). The *compute-empty* process computes nullability for every symbol based on existing rules. After this process, every symbol has information about whether it is nullable or not. The *compute-first-set*, *compute-last-set*, and *compute-*

follow-set processes update every symbol with FIRST set, LAST set, and FOLLOW set information, respectively, based on existing rules and symbol-nullability information. Note that the *follow set* is not really needed for LR(1) parser construction, but is implemented here for the completeness of set computation and for the -d option of the system (see Appendix A) which dumps FIRST, LAST, and FOLLOW set information. The *build LR states* process generates all LR(1) parsing states and the *build LALR states* merges LR(1) states into LALR(1) states. The *print-states* process displays LALR(1) states to the standard output.

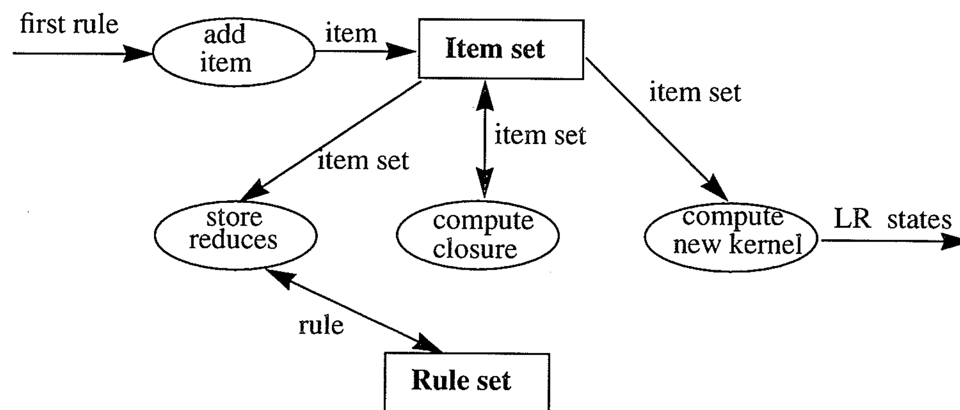


Fig. 4.3 Expansion of *build LR states* process

Figure 4.3 provides a low-level diagram for the *build LR states* process which consists of four processes. The *add item* process is a leaf process, which converts a rule into an item by appending lookahead-set and dot-position information. The *compute-closure* process generates closure items based on the given item set. The generated closure items will be used by the *compute-new-kernel* process which

generates LR states. The *store-reduces* process is also a leaf process which stores states (in a rule) in which the rule is used to reduce an item. Note that the expected process *store-shifts* is actually implemented as a sub-process of *compute-new-kernel.*

4.3 Implementation Considerations

In this section, some design and implementation considerations pursuant to parser generation will be discussed.

4.3.1 Implementation of SymbolSet, RuleSet, StateSet and ItemSet

In parsing theory, although there are concepts (and objects) such as symbols, rules, items, and states, the only space consuming objects are actually symbols. Symbols are ultimately the main components of other objects. For example, an item's main component is a rule which is composed of the left part symbol and a number of right part symbols.

Because of the above fact and the fact that the numbers of symbols, rules, items, and states dynamically increase during computation, and varies from grammar to grammar, SymbolSet, RuleSet, and StateSet are implemented as linked-lists.

Pointers are used instead of keeping multiple physical copies of symbols in rules, items, and states. Therefore, the symbols in a rule are implemented as pointers to the objects corresponding to the left hand symbol and right hand symbols, and the rule attribute in an item is a pointer to the corresponding rule. A similar philosophy applies to states. This method improves both efficiency and space usage. Since the maximum possible size of an item set can be decided when the grammar is entered in the system, ItemSet is implemented as an array of item pointers for easy and quicker handling of item sets during computation.

4.3.2 BitSet Implementation

Sets are used in a parser generator to compute and keep FIRST sets, LAST sets, FOLLOW sets and lookahead sets during parser construction. Since every symbol has a corresponding first set, last set, and follow set, and every item has a lookahead set attached to it, attention must be paid to memory consumption and efficiency.

The class "BitSet" is implemented in this program for this purpose. For reasons of space-saving and program portability, the BitSet class consists of two data members. One is the length of the bit set, which is determined by the actual total number of symbols in the grammar. The other is the actual bit set represented by a

dynamic array of `char`, i.e. a dynamic string (in C++ it's type is `unsigned char*`). Every symbol in the grammar is represented by a bit in the set. There are two reasons why `char` is chosen instead of `int`. First, most current computer systems' space is represented by 8-bit bytes, but depending on the size of the computer, the integer bit length may vary. (where 16-bit, 32-bit, and 64-bit are the common values). Portability is achieved by choosing an 8-bit char as the basic storage unit for set implementation. Second, due to the dynamic length of the string, minimum space usage is achieved since the maximum possible wasted space is 7 bits per set.

The first set, last set, and follow set should be data members of the symbol class, and the lookahead set should be a data member of the item class. An object of Symbol class should be created when a symbol is recognized by the scanner. This creates a potential problem. Because the total number of symbols in the grammar cannot be determined until the end of the scanning, the constructor for BitSet is unable to allocate appropriate space for every set object in the symbol object. The program has to be responsible for setting the `length` data member of the set, and calling another member function of the set to allocate space for every set object right after scanning the whole grammar.

4.3.3 The Hash Function for States

Hash functions are often used for looking up symbol names in a table, but in the parser generator implementation, a hash function is used to find states. The main objective of using a hash table is to make a reasonable trade off between space and speed. Therefore, the design of the hash function is critical to performance. In the implementation, the hash function for each state is the logical OR of each item's sum of rule pointer and "dot" position (`(int)rulePointer + pos`) since an item in the parser generator is implemented as a pointer to the corresponding rule and "dot" position information. An additional benefit of this hash function is that those LR(1) states with the same core items must have the same hash value. This helps compute LALR(1) states which must merge LR(1) states with identical core item sets.

4.3.4 The Parser Construction Algorithm

While several LR parser generation algorithms are available, the algorithm implemented in this thesis is the one described in section 3.3.3. This algorithm was chosen for the following reasons:

1. The algorithm enables the program to generate not only an LALR(1)

parser, but also an LR(1) parser if the grammar is not LALR(1),

2. Although the algorithm is the least efficient one, it is still acceptable in terms of the frequency of use — once per language, and
3. The conventional parser generator to which we will compare ours is also implemented in this way. It would not be fair to make a comparison with a different construction algorithm.

4.3.5 Combination of Shift and Reduce

The option of combining shift-and-reduce operations can sometimes significantly reduce the number of parser states. Combining shift and reduce means that, when there is only one item action in the current state for a particular input, if this item becomes reducable after the shift action, this item will be reduced immediately instead of waiting for the next separate reduce action. In the case of ISO Pascal, the number of LALR(1) states is 365 without shift-and-reduce combination and is 214 with this combination.

4.3.6 Handling of Object Groups

Because of the nature of parser generation, very large groups of objects of the same class have to be handled during computation. For example, hundreds or thou-

sands of states or kernels are involved during the computation. Therefore, the implementation of this kind of association is important to performance. Also, the number of objects in parser generation are dynamically growing during the computation. Due to the nature of parser generation, those object groups such as kernels, rules, and symbols have to be traversed several times. This situation is not favorable from the point of object-oriented programming. Because of the above facts, objects of the same class are implemented as a linked list instead of an array.

4.3.7 Pass by Constant Reference vs. Pass by Value

Thanks to the fact that the implementation is in C++, pass by constant reference is available. This feature is extensively used in function calls instead of pass by value for the following two reasons:

1. Performance. If an object is passed by value, there will be considerable impact on the performance especially when the object is big and the frequency of calls is high (since the copy constructor is involved in pass by value).
2. Safety. While the issue of efficiency still be addressed by passing a pointer, constant reference passing guarantees that the object passed to the function will not be mutated. Any attempts to change the "pass by

constant reference" variable will be flagged as compile-time errors. It is also important to note that a reference can never be NULL whereas a pointer can be NULL.

Chapter 5

Comparison With A Traditional Parser Generator

In this chapter, some comparisons with a traditional parser generator written by Nigel Horspool [14] are made. Although programs can be greatly affected by individual programming styles and compilers used, all possible efforts have been made to minimize these effects. First, the parser construction algorithms are the same and the outputs are in the same format. Second, the GNU C and C++ compiler (`gcc`) has been chosen for the reason that it accepts both C and C++ programs. It is also assumed that the optimizations performed on the C++ program are as good as on the C program.

All data presented here is based on tests using the `gcc` V2.7 compiler running on a SUN SPARC 1+ workstation under the SunOS 4.1.3 operating system. The test cases are ISO Pascal and Modula-2 grammars. In the following comparison tables, OOPG stands for Object-Oriented Parser Generator (my project), and

TPG stands for Traditional Parser Generator (Nigel Horspool's generator).

5.1 Running Time for Pascal & Modula-2 Grammars

The algorithm used for a computation and the quality of the compiler used for translation can have a significant effect on the running time of a program. By holding these two constant, the significance of the programming paradigm used can be measured. Two examples of the ways that the paradigm can influence running speed are the overhead of calling *virtual functions* and the added time needed for invoking *constructors* and *destructors* in the OOPG.

Two sets of results are given according to whether or not the optimizer is used during compilation.

Table 5.1: Running Time

Time (seconds)		OOPG	TPG	Ratio
ISO Pascal Grammar	Optimized	6.1	3.0	2.03
	Un-optimized	12.0	4.0	3.00
Modula-2 Grammar	Optimized	5.6	3.2	1.75
	Un-optimized	11.4	4.0	2.85

Note that without optimization, only those variables declared to be *register* are actually allocated in registers. In the OOPG, the running time difference was 3.5 seconds (on the ISO Pascal grammar) depending on whether variables are

declared *register* or not in the program (compiled without optimization). However, this difference disappears if the optimizer is applied. Because it is believed that this kind of optimization should be left to the compiler, no *registers* have been declared in the program. Meanwhile, the TPG program declares all possible variables as *register.* This explains, to some extent, why the OOPG has a bigger running-time difference between optimized and unoptimized code.

5.2 Function Calls for Pascal & Modula-2 Grammars

Due to the nature of object-oriented programming, more function calls are expected to provide support for information hiding and encapsulation. In addition, macros, which are extensively used in traditional programming, are not common in object-oriented programming. This too contributes to an increse in the number of function calls in an OO program. The following is the total number of function calls when an ISO Pascal parser or Modula-2 parser is constructed. Please note that small percentage of function calls are in-lined in the OOPG.

Table 5.2: Function Calls

Total Calls (times)	OOPG	TPG	Ratio
ISO Pascal Grammar	1,529,964	525,045	2.91
Modula-2 Grammar	1,637,900	555,610	2.94

5.3 Object Size and Executable File Size

While TPG consists of only one C source file, OOPG is composed of six header files and seven C++ source file. So, the object file size discussed below is the total size of the seven corresponding object files. Again, it is more informative to give two sets of results here.

Table 5.3: Executable Size

Executable Size(bytes)	OOPG	TPG	Ratio
Optimized	52,623	37,942	1.39
Un-optimized	69,340	46,134	1.50

Table 5.4: Object Size

Object Size(bytes)	OOPG	TPG	Ratio
Optimized	42,154	26,447	1.59
Un-optimized	59,101	37,647	1.57

On the one hand, the object and image sizes are naturally dependent on the compiler used, on the other hand, the difference between OOPG and TPG also indicates that it is hard to accomplish some object-oriented programming and software engineering ideas, such as information-hiding/encapsulation and good code-readability and maintainability, without the sacrifice of program size and object size in some situations. Of course, some object-oriented programming features such as templates and inheritance could shorten the code if applied properly.

5.4 Source Size

Although the source size does not necessarily indicate the degree of complexity, it is a rough measure of the overall programming style, including naming convention, degree of conciseness, readability, and so on. Four comparisons, regarded as useful, are listed here.

Table 5.5: Source Size

Total Number	OOPG	TPG	Ratio
Lines Excluding Comments	1933	1599	1.21
Lines Including Comments	2212	1626	1.36
Characters Excluding Comments	40916	24932	1.64
Tokens Excluding Operators	433	387	1.12

It is not surprising to discover, from table 5, that the OOPG is bigger than the TPG because object-oriented programming languages such as C++ require more definitions or "protypes" than C. Also, as mentioned in the previous section, there will likely be more functions in C++. The amount of comments usually depends solely on individual programming style.

It is also interesting that while the size of the source code grew by only 21%, the sizes of object file and running time increased by 57% and 185-200% respectively (un-optimized).

Chapter 6

Conclusions

This thesis investigates the current popular object-oriented methods as applied to LR parsing theory. An experiment applying object-oriented techniques to parser generation is presented and a parser generator has been implemented and tested with an ISO Pascal and Modula-2 grammars.

This experiment does not exaggerate the advantages of object-oriented programming in the parser generation area. Instead, an honest comparison between an object-oriented parser generator and a traditional one has been made, and every effort in both design and implementation has been made to minimize the effects of individual programming styles, compilers, and the behavioral differences between C and C++ programs as observed by Calder [11].

Some experience was gained from this experiment including that object-orientedness in parser generation is not as beneficial in some aspects as in other areas of software developmenet. This is due to the following reasons. First of all, because

parser construction computation is not highly interactive, the design and implementation issues are usually relatively simple and straightforward although the algorithms applied could be very complicated. Secondly, parser construction is almost a pure computation and the objects in the system are obvious and limited, just the same as those in the theory, such as symbols, rules, items, kernels or states, etc. Thirdly, as mentioned in chapter 4, the nature of parsing makes such unfavorable situations as walking through objects inevitable.

While the data obtained in this experiment do not favor object-oriented programming in parser generation in some aspects, it is important to note that object-oriented programming never promised faster and smaller programs. Unfortunately, it is hard to compare OOPG and TPG based on what object-oriented programming techniques promise — clearer, easier to debug, easier to maintain and reuse programs without longer term analysis & data collection. Nevertheless, the experience and test data in this experiment can still be used as a reference or start for further object-oriented parser generation research and development.

Appendix A

System User's Guide

1. Introduction

The program reads either an LALR(1) or LR(1) grammar and builds the states and lookahead sets necessary to implement a shift-reduce parser. Two forms of output are available: one is a human-readable listing of terminals, non-terminals, grammar, and state sets, which is sent to standard output; the other is a `.tbl` file which contains similar information that can be read easily by other programs. If the grammar is not an LALR(1) grammar, but is an LR(1) grammar, the program will report that there are LALR(1) conflicts and then produce the corresponding LR(1) state sets and `.tbl` file instead of LALR(1) ones.

2. Program Input

The input to the system is an LALR(1) or LR(1) grammar. Terminals and non-terminals may be any sequence of characters delimited by blanks or end-of-line. One replacement rule appears per line in the input file. The first rule defining a given non-terminal is of the form:

```
non-terminal  right-hand-side-of-rule
```

Subsequent alternatives for the same non-terminal are of the form:

```
| alternative-right-hand-side
```

The generator automatically assigns symbol numbers to each terminal and non-terminal symbol. Any symbol not appearing as the left-hand side of a rule is assumed to be a terminal symbol.

The first rule must have a right-hand side of the form.

```
bof  non-terminal  eof
```

Where `bof` (begin-of-file-mark) and `eof` (end-of-file-mark) are unique terminals.

3. Program Output

The program output produced depends on the program options chosen. The options are as follows:

`-l` List terminals, non-terminals, production rules which are numbered by the program, and state sets. In every state set, there is such information as kernel items, shift transitions, and reduction rules.

`-d` List first sets, last sets, and follow sets for the symbols.

`-t` Do not produce a `.tbl` file.

`-c` Do not combine shift & reduce operations into a single step. Consequently, there will be more states.

`-w<number>` Define the width of standard output. e.g. `-w40`.

The format of the `.tbl` file is as follows. The first line in the file contains three numbers: the number-of-symbols, the number-of-grammar-rules, and the number-of-states. For each symbol, rule, and state there is one subsequent line in the file. The first symbol, rule, and state are numbered 0.

For each symbol, there is a bit indicating whether or not the symbol is a terminal (0 means terminal, 1 means non-terminal) followed by the symbol itself.

Each rule appears in the following format:

```
left-side  length-of-right-side  right-side-symbols
```

For each state, a line is produced containing the number of transitions, reductions, and shift & reduce actions, followed by these actions. A transition (i.e. “Shift” action) is represented by:

```
<symbol-number>   S<new-state-number>
```

A reduction is represented by:

```
<symbol-number>   R<rule-number>
```

A shift&reduce action is represented by:

```
<symbol-number>   *<rule-number>
```

Appendix B

Tests on an ISO Pascal Grammar

This appendix contains information about running the parser generator on an ISO Pascal grammar. Because of the large size of the output, it has been shortened to enhance readability. All output are in Courier font and all comments added to help you understand the output are in Italic-Times font.

Program Output

```
Terminal Symbols:

   1: BOF    3: EOF    6: T_DOT    8: T_SEMI    12: T_PROGRAM
   13: T_ID    14: T_LPAR    16: T_RPAR    17: T_COMMA
   23: T_LABEL    25: T_INT    26: T_CONST    29: T_EQUAL
   32: T_PLUS    33: T_MINUS    34: T_STRING    35: T_REAL
   36: T_TYPE    42: T_POINT    44: T_DOTDOT    46: T_PACKED
   47: T_ARRAY    48: T_LSQR    50: T_RECORD    52: T_END
   53: T_SET    54: T_OF    55: T_FILE    56: T_RSQR    60: T_COLON
   61: T_CASE    63: variant_lis    67: T_VAR    78: T_PROCEDURE
   79: T_FUNCTION    84: T_BEGIN    93: T_ASSIGN    96: T_REPEAT
   97: T_UNTIL    99: T_GOTO    105: T_NE    106: T_LE    107: T_LT
   108: T_GE    109: T_GT    110: T_IN    112: T_OR    114: T_MULT
   115: T_RDIV    116: T_DIV    117: T_MOD    118: T_AND
   122: T_NOT    123: T_NIL    131: T_DO    135: T_ELSE
   136: T_IF    137: T_THEN    138: T_WHILE    139: T_FOR
```

141: T_TO   142: T_DOWNTO   143: T_WITH

Non-Terminal Symbols:

0: start   2: program   4: program_decls   5: block
7: program_head   9: decls   10: program_name
11: program_parms   15: file_list   18: label_decl_part
19: const_decl_part   20: type_decl_part   21: var_decl_part
22: proc_decl_part   24: label_decl_list
27: const_decl_list   28: const_decl   30: const
31: unsigned_num   37: type_decl_list   38: type_decl
39: type   40: simple_type   41: structured_type
43: enum_list   45: u_struct_type   49: array_rest
51: field_list   57: fixed_part   58: variant_part
59: fixed_item_list   62: tag_field   64: variant_list
65: variant   66: case_label_list   68: var_decl_list
69: var_decl   70: proc_decl_list   71: proc_decl
72: proc_heading   73: proc_beg   74: f_parm_decl
75: func_beg   76: proc_head_beg   77: func_head_beg
80: f_parm_list   81: f_parm   82: val_fparm_list
83: var_fparm_list   85: stmt_list   86: stmt   87: ul_stmt
88: label   89: simple_stmt   90: struct_stmt   91: beg_stmt
92: var   94: expr   95: proc_invok   98: case_stmt
100: noparms_pinvok   101: plist_pinvok   102: parm
103: subscripted_var   104: simple_expr   111: term
113: factor   119: unsigned_const   120: func_invok
121: set   124: plist_finvok   125: start_finvok
126: element_list   127: element   128: if_then_else
129: if_beg   130: while_beg   132: for_beg   133: with_beg
134: matched_stmt   140: updown   144: with_list
145: case_alt   146: case_beg   147: ul_matched_stmt

Production Rules:

```
 0.start := BOF program EOF
 1.program := program_decls block T_DOT
 2.program_decls := program_head T_SEMI decls
 3.program_head := program_name program_parms
 4.program_name := T_PROGRAM T_ID
 5.program_parms :=
 6.program_parms := T_LPAR file_list T_RPAR
 7.file_list := T_ID
 8.file_list := file_list T_COMMA T_ID
 9.decls := label_decl_part const_decl_part type_decl_part
      var_decl_part proc_decl_part
10.label_decl_part :=
11.label_decl_part := T_LABEL label_decl_list T_SEMI
```

```
 12.label_decl_list := T_INT
 13.label_decl_list := label_decl_list T_COMMA T_INT
 14.const_decl_part :=
 15.const_decl_part := T_CONST const_decl_list T_SEMI
 16.const_decl_list := const_decl
 17.const_decl_list := const_decl_list T_SEMI const_decl
 18.const_decl := T_ID T_EQUAL const
 19.const := unsigned_num
 20.const := T_PLUS unsigned_num
```

.

. *-- Rules 21 to 179 are omitted.*

.

```
180.updown := T_DOWNTO
181.with_beg := T_WITH with_list
182.case_stmt := case_alt stmt T_SEMI T_END
183.case_stmt := case_alt stmt T_END
184.case_beg := T_CASE expr T_OF const
185.case_beg := case_beg T_COMMA const
186.case_beg := case_alt stmt T_SEMI const
187.case_alt := case_beg T_COLON
188.matched_stmt := ul_matched_stmt
189.matched_stmt := label ul_matched_stmt
190.ul_matched_stmt := simple_stmt
191.ul_matched_stmt :=
192.ul_matched_stmt := if_then_else matched_stmt
193.ul_matched_stmt := while_beg T_DO matched_stmt
194.ul_matched_stmt := for_beg T_DO matched_stmt
195.ul_matched_stmt := with_beg T_DO matched_stmt
196.with_list := var
197.with_list := with_list T_COMMA var

THE NUMBER OF LR(1) STATES IS 1248

THE NUMBER OF LALR(1) STATES IS 214

LALR(1) TABLE IS OK


State 0:
    Kernel Items:
     [[ 0.  start  ::=  BOF  ###  program EOF ]]  T_SEMI     T_RPAR
            T_COMMA    T_LABEL    T_EQUAL    T_LSQR    T_SET
            T_OF    T_FILE    T_BEGIN    T_UNTIL
    Shift Transitions:
         to state 5 on T_PROGRAM    to state 4 on program_name
```

```
        to state 3 on program_head    to state 2 on program_decls
        to state 1 on program

State 1:
    Kernel Items:
     [[ 0.  start  ::=  BOF program  ###  EOF ]]  EOF
    Shift-Reduces:
        by rule 0 on EOF

State 2:
    Kernel Items:
     [[ 1.  program  ::=  program_decls  ###  block T_DOT ]]  EOF
    Shift Transitions:
        to state 101 on T_BEGIN    to state 6 on block

State 3:
    Kernel Items:
     [[ 2.  program_decls  ::=  program_head  ###  T_SEMI decls ]]
            T_BEGIN
    Shift Transitions:
        to state 7 on T_SEMI

State 4:
    Kernel Items:
     [[ 3.  program_head ::= program_name ### program_parms ]]
            T_SEMI
    Shift Transitions:
        to state 8 on T_LPAR
    Shift-Reduces:
        by rule 3 on program_parms

State 5:
    Kernel Items:
     [[ 4.  program_name  ::=  T_PROGRAM  ###  T_ID ]]  T_SEMI
            T_LPAR
    Shift-Reduces:
        by rule 4 on T_ID

State 6:
    Kernel Items:
     [[ 1.  program  ::=  program_decls block  ###  T_DOT ]]  EOF
    Shift-Reduces:
        by rule 1 on T_DOT

State 7:
    Kernel Items:
     [[ 2.  program_decls  ::=  program_head T_SEMI  ###  decls ]]
```

```
        T_BEGIN
    Shift Transitions:
        to state 15 on T_LABEL    to state 14 on label_decl_part
    Shift-Reduces:
        by rule 2 on decls

State 8:
    Kernel Items:
    [[ 6.  program_parms  ::=  T_LPAR  ###  file_list T_RPAR ]]
        T_SEMI
    Shift Transitions:
        to state 16 on file_list
    Shift-Reduces:
        by rule 7 on T_ID

State 9:
    Kernel Items:
    [[ 177.  while_beg  ::=  T_WHILE  ###  expr ]]  T_DO
    Shift Transitions:
        to state 159 on start_finvok
        to state 160 on plist_finvok    to state 158 on T_NOT
        to state 155 on term    to state 118 on simple_expr
        to state 151 on subscripted_var    to state 156 on var
        to state 162 on T_LSQR    to state 153 on T_MINUS
        to state 152 on T_PLUS    to state 157 on T_LPAR
        to state 161 on T_ID
    Shift-Reduces:
        by rule 158 on T_NIL
        by rule 154 on set
        by rule 153 on func_invok
        by rule 151 on unsigned_const
        by rule 144 on factor
        by rule 177 on expr
        by rule 27 on T_REAL
        by rule 157 on T_STRING
        by rule 156 on unsigned_num
        by rule 26 on T_INT

State 10:
    Kernel Items:
    [[ 178.  for_beg  ::=  T_FOR  ###  T_ID T_ASSIGN expr updown
        expr ]]  T_DO
    Shift Transitions:
        to state 23 on T_ID

State 11:
    Kernel Items:
```

```
    [[ 181.  with_beg  ::=  T_WITH  ###  with_list ]]  T_DO
    Shift Transitions:
        to state 25 on with_list    to state 151 on
subscripted_var
        to state 24 on var
    Shift-Reduces:
        by rule 121 on T_ID

State 12:
    Kernel Items:
    [[ 184.  case_beg ::=  T_CASE  ###  expr T_OF const ]] T_COMMA
            T_COLON
    Shift Transitions:
        to state 159 on start_finvok
        to state 160 on plist_finvok    to state 158 on T_NOT
        to state 155 on term    to state 118 on simple_expr
        to state 151 on subscripted_var    to state 26 on expr
        to state 156 on var    to state 162 on T_LSQR
        to state 153 on T_MINUS    to state 152 on T_PLUS
        to state 157 on T_LPAR    to state 161 on T_ID
    Shift-Reduces:
        by rule 158 on T_NIL
        by rule 154 on set
        by rule 153 on func_invok
        by rule 151 on unsigned_const
        by rule 144 on factor
        by rule 27 on T_REAL
        by rule 157 on T_STRING
        by rule 156 on unsigned_num
        by rule 26 on T_INT

State 13:
    Kernel Items:
    [[ 187.  case_alt  ::=  case_beg  ###  T_COLON ]]  T_SEMI
            T_ID    T_INT    T_END    T_CASE    T_BEGIN
            T_REPEAT    T_GOTO    T_IF    T_WHILE    T_FOR
            T_WITH
    [[ 185.  case_beg  ::=  case_beg  ###  T_COMMA const ]]
T_COMMA
            T_COLON
    Shift Transitions:
        to state 27 on T_COMMA
    Shift-Reduces:
        by rule 187 on T_COLON

State 14:
    Kernel Items:
```

```
    [[ 9.  decls  ::=  label_decl_part  ###  const_decl_part
          type_decl_part var_decl_part proc_decl_part ]]  T_BEGIN
    Shift Transitions:
         to state 29 on T_CONST    to state 28 on const_decl_part

State 15:
    Kernel Items:
    [[ 11.  label_decl_part  ::=  T_LABEL  ###  label_decl_list
          T_SEMI ]]  T_CONST    T_TYPE    T_VAR    T_PROCEDURE
            T_FUNCTION    T_BEGIN
    Shift Transitions:
         to state 30 on label_decl_list
    Shift-Reduces:
         by rule 12 on T_INT

State 16:
    Kernel Items:
    [[ 8.  file_list ::= file_list  ###  T_COMMA T_ID ]]  T_RPAR
            T_COMMA
    [[ 6.  program_parms  ::=  T_LPAR file_list  ###  T_RPAR ]]
            T_SEMI
    Shift Transitions:
         to state 31 on T_COMMA
    Shift-Reduces:
         by rule 6 on T_RPAR

State 17:
    Kernel Items:
    [[ 103.  stmt  ::=  label  ###  ul_stmt ]]  T_SEMI    T_END
            T_UNTIL
    Shift Transitions:
         to state 13 on case_beg    to state 44 on case_alt
         to state 11 on T_WITH    to state 10 on T_FOR
         to state 9 on T_WHILE    to state 22 on T_IF
         to state 21 on with_beg    to state 20 on for_beg
         to state 19 on while_beg    to state 43 on if_beg
         to state 18 on if_then_else    to state 42 on T_GOTO
         to state 41 on T_REPEAT    to state 40 on beg_stmt
         to state 101 on T_BEGIN    to state 12 on T_CASE
    Reductions:
         by rule 107 on T_SEMI T_END
         by rule 114 on T_ID
    Shift-Reduces:
         by rule 111 on case_stmt
         by rule 106 on struct_stmt
         by rule 105 on simple_stmt
         by rule 103 on ul_stmt
```

```
        by rule 113 on block

State 18:
    Kernel Items:
    [[ 170.  struct_stmt  ::=  if_then_else  ###  stmt ]]  T_SEMI
          T_END    T_UNTIL
    Shift Transitions:
        to state 13 on case_beg    to state 44 on case_alt
        to state 11 on T_WITH    to state 10 on T_FOR
        to state 9 on T_WHILE    to state 22 on T_IF
        to state 21 on with_beg    to state 20 on for_beg
        to state 19 on while_beg    to state 43 on if_beg
        to state 18 on if_then_else    to state 42 on T_GOTO
        to state 41 on T_REPEAT    to state 40 on beg_stmt
        to state 17 on label    to state 101 on T_BEGIN
        to state 12 on T_CASE    to state 39 on T_INT
    Reductions:
        by rule 107 on T_SEMI T_END
        by rule 114 on T_ID
    Shift-Reduces:
        by rule 111 on case_stmt
        by rule 106 on struct_stmt
        by rule 105 on simple_stmt
        by rule 102 on ul_stmt
        by rule 170 on stmt
        by rule 113 on block

State 19:
    Kernel Items:
    [[ 172.  struct_stmt  ::= while_beg  ###  T_DO stmt ]]  T_SEMI
          T_END    T_UNTIL
    Shift Transitions:
        to state 36 on T_DO

State 20:
    Kernel Items:
    [[ 173.  struct_stmt  ::=  for_beg  ###  T_DO stmt ]]  T_SEMI
          T_END    T_UNTIL
    Shift Transitions:
        to state 37 on T_DO
```

.

. *-- States 21 - 199 are omitted.*

.

```
State 200:
```

```
    Kernel Items:
     [[ 48.  array_rest ::= simple_type T_COMMA  ###  array_rest ]]
          T_SEMI    T_RPAR    T_END
    Shift Transitions:
         to state 193 on simple_type    to state 192 on T_MINUS
         to state 191 on T_PLUS    to state 144 on const
         to state 143 on T_LPAR    to state 142 on T_ID
    Shift-Reduces:
         by rule 48 on array_rest
         by rule 27 on T_REAL
         by rule 25 on T_STRING
         by rule 19 on unsigned_num
         by rule 26 on T_INT

State 201:
    Kernel Items:
     [[ 64.  variant  ::=  case_label_list T_COLON T_LPAR  ###
         field_list T_RPAR ]]  T_SEMI
    Shift Transitions:
         to state 205 on T_CASE    to state 203 on fixed_part
         to state 206 on field_list    to state 204 on T_ID
    Shift-Reduces:
         by rule 54 on fixed_item_list
         by rule 52 on variant_part

State 202:
    Kernel Items:
     [[ 47.  array_rest  ::=  simple_type T_RSQR T_OF  ###  type ]]
          T_SEMI    T_RPAR    T_END
    Shift Transitions:
         to state 149 on T_FILE    to state 148 on T_SET
         to state 147 on T_RECORD    to state 146 on T_ARRAY
         to state 145 on T_PACKED    to state 141 on T_POINT
         to state 192 on T_MINUS    to state 191 on T_PLUS
         to state 144 on const    to state 143 on T_LPAR
         to state 142 on T_ID
    Shift-Reduces:
         by rule 41 on u_struct_type
         by rule 34 on structured_type
         by rule 33 on simple_type
         by rule 47 on type
         by rule 27 on T_REAL
         by rule 25 on T_STRING
         by rule 19 on unsigned_num
         by rule 26 on T_INT

State 203:
```

```
    Kernel Items:
     [[ 55.  fixed_part  ::=  fixed_part  ###  T_SEMI
          fixed_item_list ]]  T_SEMI    T_RPAR    T_END
     [[ 51.  field_list ::= fixed_part  ###  T_SEMI variant_part ]]
           T_RPAR    T_END
     [[ 50.  field_list  ::=  fixed_part  ###  T_SEMI ]]  T_RPAR
           T_END
     [[ 49.  field_list  ::= fixed_part  ###  ]]  T_RPAR    T_END
    Shift Transitions:
        to state 207 on T_SEMI
    Reductions:
        by rule 49 on T_RPAR T_END

State 204:
    Kernel Items:
     [[ 57.  fixed_item_list  ::=  T_ID  ###  T_COMMA
          fixed_item_list ]]  T_SEMI    T_RPAR    T_END
     [[ 56.  fixed_item_list ::= T_ID  ###  T_COLON type ]]  T_SEMI
           T_RPAR    T_END
    Shift Transitions:
        to state 208 on T_COLON    to state 209 on T_COMMA

State 205:
    Kernel Items:
     [[ 59.  variant_part  ::=  T_CASE  ###  tag_field T_OF
          variant_list T_SEMI ]]  T_RPAR    T_END
     [[ 58.  variant_part  ::=  T_CASE  ###  tag_field T_OF
          variant_lis ]]  T_RPAR    T_END
    Shift Transitions:
        to state 210 on tag_field    to state 128 on T_ID

State 206:
    Kernel Items:
     [[ 64.  variant ::= case_label_list T_COLON T_LPAR field_list
            ###  T_RPAR ]]  T_SEMI
    Shift-Reduces:
        by rule 64 on T_RPAR

State 207:
    Kernel Items:
     [[ 55.  fixed_part  ::=  fixed_part T_SEMI  ###
          fixed_item_list ]]  T_SEMI    T_RPAR    T_END
     [[ 51.  field_list  ::= fixed_part T_SEMI  ### variant_part ]]
           T_RPAR    T_END
     [[ 50.  field_list  ::=  fixed_part T_SEMI  ###  ]]  T_RPAR
           T_END
    Shift Transitions:
```

```
        to state 205 on T_CASE    to state 204 on T_ID
    Reductions:
        by rule 50 on T_RPAR T_END
    Shift-Reduces:
        by rule 55 on fixed_item_list
        by rule 51 on variant_part

State 208:
    Kernel Items:
    [[ 56.  fixed_item_list ::= T_ID T_COLON  ###  type ]]  T_SEMI
           T_RPAR    T_END
    Shift Transitions:
        to state 149 on T_FILE    to state 148 on T_SET
        to state 147 on T_RECORD    to state 146 on T_ARRAY
        to state 145 on T_PACKED    to state 141 on T_POINT
        to state 192 on T_MINUS    to state 191 on T_PLUS
        to state 144 on const    to state 143 on T_LPAR
        to state 142 on T_ID
    Shift-Reduces:
        by rule 41 on u_struct_type
        by rule 34 on structured_type
        by rule 33 on simple_type
        by rule 56 on type
        by rule 27 on T_REAL
        by rule 25 on T_STRING
        by rule 19 on unsigned_num
        by rule 26 on T_INT

State 209:
    Kernel Items:
    [[ 57.  fixed_item_list  ::=  T_ID T_COMMA  ###
          fixed_item_list ]]  T_SEMI    T_RPAR    T_END
    Shift Transitions:
        to state 204 on T_ID
    Shift-Reduces:
        by rule 57 on fixed_item_list

State 210:
    Kernel Items:
    [[ 59.  variant_part  ::=  T_CASE tag_field  ###  T_OF
          variant_list T_SEMI ]]  T_RPAR    T_END
    [[ 58.  variant_part  ::=  T_CASE tag_field  ###  T_OF
          variant_lis ]]  T_RPAR    T_END
    Shift Transitions:
        to state 211 on T_OF

State 211:
```

```
    Kernel Items:
     [[ 59.  variant_part  ::=  T_CASE tag_field T_OF  ###
          variant_list T_SEMI ]]  T_RPAR    T_END
     [[ 58.  variant_part  ::=  T_CASE tag_field T_OF  ###
          variant_lis ]]  T_RPAR    T_END
    Shift Transitions:
        to state 176 on case_label_list
        to state 212 on variant_list    to state 192 on T_MINUS
        to state 191 on T_PLUS
    Shift-Reduces:
        by rule 62 on variant
        by rule 58 on variant_lis
        by rule 27 on T_REAL
        by rule 25 on T_STRING
        by rule 19 on unsigned_num
        by rule 65 on const
        by rule 26 on T_INT
        by rule 22 on T_ID

State 212:
    Kernel Items:
     [[ 63.  variant_list  ::= variant_list  ###  T_SEMI variant ]]
           T_SEMI
     [[ 59.  variant_part  ::=  T_CASE tag_field T_OF variant_list
           ###  T_SEMI ]]  T_RPAR    T_END
    Shift Transitions:
        to state 213 on T_SEMI

State 213:
    Kernel Items:
     [[ 63.  variant_list  ::= variant_list T_SEMI  ###  variant ]]
           T_SEMI
     [[ 59.  variant_part  ::=  T_CASE tag_field T_OF variant_list
          T_SEMI  ###  ]]  T_RPAR    T_END
    Shift Transitions:
        to state 176 on case_label_list    to state 192 on T_MINUS
        to state 191 on T_PLUS
    Reductions:
        by rule 59 on T_RPAR T_END
    Shift-Reduces:
        by rule 63 on variant
        by rule 27 on T_REAL
        by rule 25 on T_STRING
        by rule 19 on unsigned_num
        by rule 65 on const
        by rule 26 on T_INT
        by rule 22 on T_ID
```

-- The following is the contents of the .tbl file.

```
148 198 214
```
-- Number of symbols, number of rules, number of states.
-- The following are symbols. 0 = terminal, 1 = nonterminal.

```
1 start
0 BOF
1 program
0 EOF
1 program_decls
1 block
0 T_DOT
1 program_head
0 T_SEMI
1 decls
1 program_name
1 program_parms
0 T_PROGRAM
0 T_ID
0 T_LPAR
1 file_list
0 T_RPAR
0 T_COMMA
1 label_decl_part
1 const_decl_part

  .
  .
  .
```
-- Other symbols are omitted.

```
1 element
1 if_then_else
1 if_beg
1 while_beg
0 T_DO
1 for_beg
1 with_beg
1 matched_stmt
0 T_ELSE
0 T_IF
0 T_THEN
0 T_WHILE
0 T_FOR
1 updown
0 T_TO
```

```
0 T_DOWNTO
0 T_WITH
1 with_list
1 case_alt
1 case_beg
1 ul_matched_stmt
```

-- The following are rules in the .tbl file.

```
0 3 1 2 3
2 3 4 5 6
4 3 7 8 9
7 2 10 11
10 2 12 13
11 0
11 3 14 15 16
15 1 13
15 3 15 17 13
9 5 18 19 20 21 22
18 0
18 3 23 24 8
24 1 25
24 3 24 17 25
19 0
19 3 26 27 8
27 1 28
27 3 27 8 28
28 3 13 29 30
30 1 31

 .
 .
 .
```

-- Rules 21 - 179 are omitted.

```
130 2 138 94
132 6 139 13 93 94 140 94
140 1 141
140 1 142
133 2 143 144
98 4 145 86 8 52
98 3 145 86 52
146 4 61 94 54 30
146 3 146 17 30
146 4 145 86 8 30
145 2 146 60
134 1 147
134 2 88 147
147 1 89
147 0
```

```
147 2 128 134
147 3 130 131 134
147 3 132 131 134
147 3 133 131 134
144 1 92
144 3 144 17 92
```

-- The following are the states in the .tbl file.

```
5 12 S5 10 S4 7 S3 4 S2 2 S1

1 3 *0

2 84 S101 5 S6

1 8 S7

2 14 S8 11 *3

1 13 *4

1 6 *1

3 23 S15 18 S14 9 *2

2 15 S16 13 *7

22 125 S159 124 S160 123 *158 122 S158 121 *154 120 *153 119 *151
113 *144 111 S155 104 S118 103 S151 94 *177 92 S156 48 S162 35 *27
34 *157 33 S153 32 S152 31 *156 25 *26 14 S157 13 S161

1 13 S23

4 144 S25 103 S151 92 S24 13 *121

22 125 S159 124 S160 123 *158 122 S158 121 *154 120 *153 119 *151
113 *144 111 S155 104 S118 103 S151 94 S26 92 S156 48 S162 35 *27
34 *157 33 S153 32 S152 31 *156 25 *26 14 S157 13 S161

2 60 *187 17 S27

2 26 S29 19 S28

2 25 *12 24 S30

2 17 S31 16 *6

24 8 R107 13 R114 52 R107 146 S13 145 S44 143 S11 139 S10 138 S9
136 S22 133 S21 132 S20 130 S19 129 S43 128 S18 99 S42 98 *111 96
S41 91 S40 90 *106 89 *105 87 *103 84 S101 61 S12 5 *113

27 8 R107 13 R114 52 R107 146 S13 145 S44 143 S11 139 S10 138 S9
136 S22 133 S21 132 S20 130 S19 129 S43 128 S18 99 S42 98 *111 96
S41 91 S40 90 *106 89 *105 88 S17 87 *102 86 *170 84 S101 61 S12
25 S39 5 *113

1 131 S36

1 131 S37
```

.

-- States 21 - 199 are omitted.

.

```
11 49 *48 40 S193 35 *27 34 *25 33 S192 32 S191 31 *19 30 S144 25
*26 14 S143 13 S142

6 61 S205 59 *54 58 *52 57 S203 51 S206 13 S204

19 55 S149 53 S148 50 S147 47 S146 46 S145 45 *41 42 S141 41 *34
40 *33 39 *47 35 *27 34 *25 33 S192 32 S191 31 *19 30 S144 25 *26
14 S143 13 S142

3 8 S207 16 R49 52 R49

2 60 S208 17 S209

2 62 S210 13 S128

1 16 *64

6 16 R50 52 R50 61 S205 59 *55 58 *51 13 S204

19 55 S149 53 S148 50 S147 47 S146 46 S145 45 *41 42 S141 41 *34
40 *33 39 *56 35 *27 34 *25 33 S192 32 S191 31 *19 30 S144 25 *26
14 S143 13 S142

2 59 *57 13 S204

1 54 S211

12 66 S176 65 *62 64 S212 63 *58 35 *27 34 *25 33 S192 32 S191 31
*19 30 *65 25 *26 13 *22

1 8 S213

12 16 R59 52 R59 66 S176 65 *63 35 *27 34 *25 33 S192 32 S191 31
*19 30 *65 25 *26 13 *22
```

References

[1] Peter Coad & Edward Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, 1990.

[2] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling & Design, Prentice Hall, 1991.

[3] David W. Embley, *et al* *Object-Oriented Systems Analysis*, Yourdon Press, 1992.

[4] Grady Booch, *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Series, 1994.

[5] Nigel P. Chapman, *LR Parsing - Theory and Practice*, Cambridge University Press, 1987.

[6] Jim Holmes, *Object-Oriented Compiler Construction,* Prentice-Hall inc., 1995.

[7] Alfred V Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compiler Principles, Techniques, and Tools,* Addison Wesley, Reading, Massachusetts, 1986.

[8] Frank DeRemer and Thomas Pennello, "Efficient computation of LALR(1) look-ahead sets" *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 4, October 1982.

[9] Joseph C. H. Park, K. M. Choe, and C.H. Chang, "A new analysis of LALR formalism" *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, January 1985.

[10] Fred Ives, "Unifying view of recent LALR(1) lookahead set algorithms" ACM SIGPLAN Notices, Vol. 21, No. 7 July 1986.

[11] Brad Calder, Dirk Grunwald, and Benjamin Zorn, "Quantifying behavioral differences between C and C++ programs", Journal of Programming Languages, Vol 2, No. 4, 1994.

[12] Ivar Jacobson, *Object-Oriented Software Engineering - A Use Case Driven Approach,* Addison Wesley, 1992.

[13] Grady Booch, Martin Fowler, Adele Goldberg, and Kenneth Rubin, "Object-Oriented Analysis & Design - Finding Your Path", SIGS Publications, 1993.

[14] M. Whitney and Nigel Horspool, "Extremely rapid LR parsing" Proceedings of Workshop on Compiler-Compilers and High-Speed Compilation, Berlin, G.D.R., October 1988.

[15] Jan Bosch, "Delegating Compiler Objects - An object-oriented approach to crafting compilers", *Proceedings of the 6th International Conference on Compiler Construction, CC'96*, Linkoping, Sweden, April 1996.

[16] Damian Conway, "Parsing with C++ classes", *ACM SIGPLAN Notices, Vol 29*, No. 1, January 1994.