International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 3 Issue: 2 449 - 453

Parsing and Compiler design Techniques for Compiler Applications

Prof. Rajesh Babu
Assistant Professor
Department of CSE
TGPCOE, Nagpur, India

Prof. Vishal Tiwari
Assistant Professor
Department of CSE
TGPCOE, Nagpur, India

Prof. Jiwan Dehakar
Assistant Professor
Department of CSE
TGPCOE, Nagpur, India

Abstract: - Parsing Techniquesare used in software engineering, but most students will not be compiler Techniques. The course is suitable for undergraduate and starting time in graduate students. Any program written in a high level programming language must be translated to object code before going to be executed. Compiler is the need to design and connectivity between the hardware and software process. Compiler design grammar is having set of rule that specify how sentences can be structured with the terminals, non-terminals and the set of Productions. Code generation translated to machine language or assembly language which translated to machine language. For target machines with several registers the code generator is responsible for register allocation. Compiler not only translates the information can be used for the processor design.

Index Terms: - *Compiler Techniques, ParsingTechniques,Code Optimization*

*****

I. INTRODUCTION

Compiler Design Techniques

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers.

Structure of a Compiler

A compiler is a program that can read a program in one language, the source language and translate it into an equivalent program in another language, the target language. A compiler is a box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

Below figure shows the structure of a compiler:

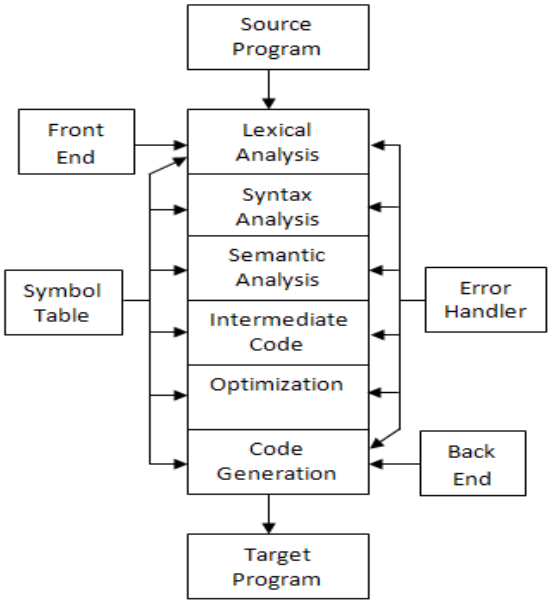

Fig 1:-The structure of a Compiler

Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form:

In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the

tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.

II. Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the inttofloat operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform into the shorter sequence.

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables. For example, using registers Rl and R2, the intermediate code in might get translated into the machine code

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers. The code in loads the contents of address id3 into register R2, and then multiplies it with floating-point constant 60.0.[7] The # signifies that 60.0 are to be treated as an immediate constant. The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2.

Finally, the value in register Rl is stored into the address of i d l, so the code correctly implements the assignment statement.

III. Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument, and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

The Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

IV PARSING APPLICATION

Parsing can be defined as a process of analysing a text which contains a sequence of tokens to determine its grammatical structure in given grammar. One the source code is syntactically valid the compiler has generated into abstract syntax tree or syntax directed translation of the source code.

Classical parsers conventionally accept a context free language defined by a context free, for each program

parser does produce a phrase structure referred to as an abstract syntax tree. Parse including error stabilization and AST constructors can be generated from context free grammars for parsers. Classical parsing techniques may be applied as long as program conforms to the syntax of a programming language however can be assumed in general as programs to analyze can be incomplete erroneous or conform to a dialect of the language.

Fuzzy parsing was designed to efficiently develop parsers by performing the analysis on selected parts of the source instead of the whole input. It is specified by a set of fuzzy context free sub grammars each with their own axioms does not require strict adherence to a language grammar. It scans for instances of the axioms and then parses according to the grammar makes parsing more robust since it ignores source fragments including missing parts errors and deviations that subsequent analyses abstract from anyway. Fuzzy parsers DelphiXPG utilise a context fuzzy parsing technology for building a cross identifiers, OPARI is source to source translation tool which automatically adds all necessary calls to the pomp runtime measurement which allows to collect runtime performance data of fortran, C, C++ applications.

Using Natural Language processing technique can be applied to formal languagedependency structure is one way of representing the syntax of natural language. Thistechnique automatically generates the language specific information extractor using machinelearning and training of a generic parsing instead of explicitly specifying the information
extractor using grammar and transformation rules.

Android Application Development web service request and response uses three types of parsing xml DOM Parser PULL Parser SAX Parser. Android provides a library that contains classes used to parse xml by constructing a document and matching each node to parse the info to parse with DOM parser as shown below. Void parse By DOM (String response) throws Parser Configuration Exception, SAX Exception,
IOException

```
{
Person person=new Person ();
DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance ();
DocumentBuilder db = dbf.newDocumentBuilder ();
Document doc = db.parse(new Input Source (new String
Reader (response)));
doc.getDocumentElement ().normalize();
NodeList nodeList = doc.getElementsByTagName
("person");
Node node=nodeList. Item(0);
for (int i = 0; i <node.getChildNodes().getLength(); i++)
{
Node temp=node.getChildNodes().item (i);
if(temp.getNodeName().equalsIgnoreCase("firstname"))
{
person.firstName=temp.getTextContent ();
}
else if(temp.getNodeName().equalsIgnoreCase("lastname"))
{
person.lastName=temp.getTextContent ();
}
else if(temp.getNodeName().equalsIgnoreCase("age"))
{
person.age=Integer.parseInt (temp.getTextContent ());
}
}
Log.e ("person", person.firstName+"
"+person.lastName+"
"+String.valueOf(person.age));
}
```

It retrieves the correct information but the user needs to familiar with the xml structure so thatwe know the order of each xml. Android provides org.xml.sax package that has provide the event driven SAX parser to parse the previous response with SAX parser have to create a class extending Default Handler.

Start Document () invoked when the xml document is open there we can initialize any member variables.

Start Element () invokes when the parser encounters a xml node here we can initializespecific instances of our object.

EndElement () invoked when the parser eaches the closing of a xml tag here theelement value would have been completely.Characters () this method is called when the parser reads characters of a node value.

The method optJSONArray, optString, optInt instead of using getString, getInt because theopt methods return empty strings or zero integers if no elements are found

V GRAMMARS IN COMPILER DESIGN

Equivalent grammars

As we shall see, not all grammars are suitable as the starting point for developing practical parsing algorithms, and an important part of compiler theory is concerned with the ability to find equivalent grammars. Two grammars are said to be equivalent if they describe the same language, that is, can generate exactly the same set of sentences (not necessarily using the same set of sentential forms or parse trees).

In general we may be able to find several equivalent grammars for any language. A distinct problem in this regard is a tendency to introduce far too few non-terminals, or alternatively, far too many. Choosing too few non-terminals means that semantic implications are very awkward to discern at all, too many means that one runs the risk of ambiguity, and of hiding the semantic implications in a mass of hard to follow alternatives.

It may be of some interest to give an approximate count of the numbers of non-terminals and productions that have been used in the definition of a few languages:

Language	Non-terminals	Productions
Pascal	110	180
Pascal (ISO std)	160	300

 ISSN: 2321-8169
449 - 453

Edison	45	90
C	75	220
C++	110	270
ADA	180	320
Modula-2 (Wirth)	74	135
Modula-2 (ISO std)	225	306

Ambiguous grammars

An important property which one looks for in programming languages is that every sentence that can be generated by the language should have a unique parse tree, or, equivalently, a unique left (or right) canonical parse. If a sentence produced by a grammar has two or more parse trees then the grammar is said to be ambiguous. An example of ambiguity is provided by another attempt at writing a grammar for simple algebraic expressions - this time apparently simpler than before:

(E6)
Goal = *Expression*. (1)
Expression = *Expression* "-" *Expression* (2)
Expression "*" *Expression* (3)
Factor. (4)
Factor = "*a*" | "*b*" | "*c*". (5, 6, 7)

With this grammar the sentence *a - b * c* has two distinct parse trees and two canonical derivations. We refer to the numbers to show the derivation steps.

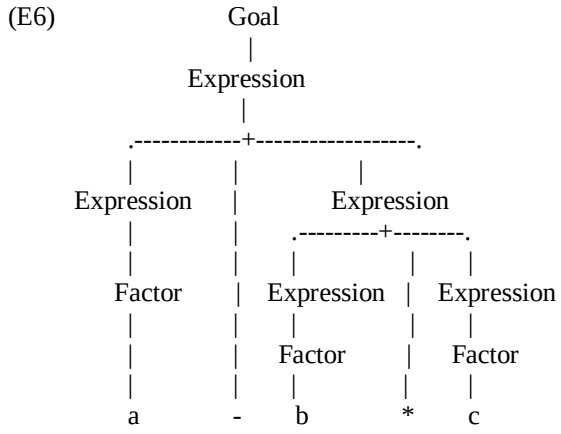

Fig 2: One parse tree for the expression a - b * c using grammar E6

Context sensitivity

Some potential ambiguities belong to a class which is usually termed context-sensitive. Spoken and written language is full of such examples, which the average person parses with ease, albeit usually within a particular cultural context or idiom. For example, the sentences

Time flies like an arrowand
Fruit flies like a banana

in one sense have identical construction
Noun Verb Adverbial phrase

but, unless we were preoccupied with aerodynamics, in listening to them we would probably subconsciously parse the second along the lines of

Adjective Noun Verb Noun phrase

Examples like this can be found in programming languages too. In Fortran a statement of the form

A = B (J)

(when taken out of context) could imply a reference either to the Jth element of array B, or to the evaluation of a function B with integer argument J. Mathematically there is little difference - an array can be thought of as a mapping, just as a function can, although programmers may not often think that way.

VI APPLICATIONS OF COMPILER DESIGN

Compiler Network Processor*:*

Network processor instruction set allows avoiding costly shift operations special instructions for packet level addressing, compiler bitpacket manipulation is made visible to the programmer by means of compiler knownfunctions and maps call to compiler known functions not the regular function calls intoinstruction sequences. Using compiler known functions the developers have detail knowledgeabout the underlying processor readability of the code is improved significantly.

Collaboration of Compiler & Operating System:
System designer must define how toinitiate a speed change and how to select a speed level to automatically deciding on theproper locations to insert PMPs by the compiler in an application code, how frequently thespeed should be changed. The solution over here that determines how far apart any two PMPsshould be placed with sequential code and an estimate of instruction latency the code isinserted. In real time the problem occurs due to the presence of branches, loops andprocedure calls that eliminate the determinism of the executed path compared to sequentialcode.Benefits to use collaborative scheme that includes both PMHs and PMPs that timinginformation can be inexpensively collected without actually doing a speed change andincurring its high overhead.

Twosystem calls are needed that are called only once at the start of an application one is a systemcall that gives the address of the buffer that holds timing information collected by hints.Executes the power management hints at some point before a PMP to update the WCRbased on the path of execution. Compiler role is to support and inserts PMHs in theapplication code. Dynamically the compiler instruments the PMHs in a way to collect theapplication dynamic behavior information.

VII CONCLUSION

In this analysis describes the implementation of compiler for compiler application in order addressing the accessible at the high level language which uses the compiler knowledgefunction and register allocation technique. Outlines the techniques of compiler parsing
application and types the grammars which can be used in computer application in designprocess, aim are to model the compiler not only used for translation.

REFERENCE

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*

[2] J.H Edmondson P. Rubinfeld R. Preston and Rajagopalan, Superscalar instructionexecution in the 21164 Alpha microprocessor "IEEE micro pp. 33-43 April 1995.

[3] Jens Nilsson, Proceedings of the 11th International Conference on Parsing Technologies(IWPT), Paris, October 2009.

[4] elphixpg.com/docs/contextparsing.html

[5] A.V. Aho M.Ganapathi and S.W.K. Tjiang "code generation using tree matching anddynamic programming". ACM Trans Program Lang. Syste. Vol 11, no. 4, pp, 4911-516 2009.

[6] Azevedo A Issenin I, Cornea R, Gupta R, Dutt N, Veidenbaum A, Nicolau A dynamicvoltage scheduling using program Design automation and test in Europe 2005.

[7] William A. Barrett , San Jose State UniversityCompiler DesignCmpE 152,FALL Version, 2005.

[8] Alfred V.Aho, Ravi Sethi, Jeffery D. Ullman, AddisonWesley, Compilers Principles, Techniques, and Tools,2007.