分类号：_____ 密级：_____

U D C：_____ 编号：_____

工学硕士学位论文

面向 VHDL 算法级行为描述的程序语言编译方法研究

硕士研究生 ：毕美娜

指导教师 ：顾国昌 教授

学位级别 ：工学硕士

学科、专业 ：计算机软件与理论

所在单位 ：计算机科学与技术学院

论文提交日期 ：2008 年 1 月 5 日

论文答辩日期 ：2008 年 3 月 11 日

学位授予单位 ：哈尔滨工程大学

摘　要

VHDL 是描述数字系统的硬件描述语言，C 是编写顺序语句程序的高级编程语言。由于 C 语言结构清晰，可扩充性强，因此将 C 语言描述的源代码编译为 VHDL 描述的代码，用具有顺序特征的 C 语句描述的算法表征具有并发特征的 VHDL 电路设计，可以大大缩短进行数字系统设计的时间，降低系统设计的复杂度。本文对面向 VHDL 的算法级行为描述的程序语言编译方法进行了深入的研究。提出了将 CDFG 作为中间表示的方法，然后对本文提出的方法进行了实验，对生成的 VHDL 代码进行了仿真实验，并对实验结果进行了分析。

首先，简要介绍了 VHDL，并分析了 C 与 VHDL 的区别，还阐述了基本的编译技术及其发展概况。给出了面向 VHDL 算法级行为描述的程序语言编译方法的整体设计方案。

然后，详细阐述了本文提出的编译方法的具体编译过程。为了清楚的说明编译的过程，给出了一个循序渐进的阐述过程：先介绍了词法分析和语法分析部分，然后在词法分析的基础上给出了由抽象语法树到 CDFG 再到 VHDL 代码的转化过程。详细研究了基于抽象语法树生成 CDFG 的过程。

最后，将 C 代码作为输入数据，应用本文的编译方法实现了从 C 代码到 VHDL 代码的编译，并对输出的 VHDL 代码在仿真软件中进行了硬件功能的仿真。实验表明，该编译方法能够有效地实现由 C 语言描述的源代码到 VHDL 代码的编译。

关键词：VHDL；词法分析；语法分析；编译

ABSTRACT

VHDL is a language for the description of digital hardware system, and C is a high level programming language for coding sequential statements. Because C is well-structured and easily-extended, when C code being compiled to VHDL code, using sequential C language to model circuits in VHDL with concurrent characterization, can reduce time and complexity of designing on digital system. In this thesis, the approach to compile programming language into VHDL behavioral description is studied. The method of making CDFG to be intermediate representation is presented, then the experiment on the method is carried out, the translated VHDL code is performed by simulation, and the result of experiment is analyzed.

Firstly, VHDL and the difference of C and VHDL is introduced, compile techniques and its status are given. The overall design plan of this thesis is presented briefly.

Secondly, the compiling process of the compiling method in this thesis is discussed in detail. In order to explain the process of the compile techniques, a gradual study process is given: first, lexical analysis and grammar analysis are discussed; second, based on grammar analysis, the processes from AST to CDFG and from CDFG to VHDL code are given. The method of generating CDFG based on AST is studied in detail.

Finally, input the C code, the compiling C to VHDL is carried out by using the presented method of this thesis successfully, and the translated VHDL code is evaluated in simulation software. The result of experiment show that the compile method is effective in C to VHDL.

Keywords: VHDL; lexical analysis; grammar analysis; compile

哈尔滨工程大学
学位论文原创性声明

本人郑重声明：本论文的所有工作，是在导师的指导下，由作者本人独立完成的。有关观点、方法、数据和文献的引用已在文中指出，并与参考文献相对应。除了文中已注明引用的内容外，本论文不包换任何其它个人或者集体已经公开发表的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确的方式注明。本人完全意识到本声明的法律结果由本人承担。

作者（签字）：毕美娜

日期：2008 年 3 月 13 日

第1章 绪论

1.1 本课题研究的意义概述

VHDL 语言是一种主流的硬件描述语言，具有很强的描述和建模能力，能从多个层次对数字系统进行建模和描述，可以大大简化硬件设计任务，提高设计的可靠性。基于 VHDL 语言的设计方法得到了广泛的应用，VHDL 语言已成为硬件描述语言的工业标准。

近年来随着微电子技术和超大规模集成电路技术的高速发展，进行电子系统设计时的系统复杂度不断加大。一方面，系统复杂度不断增加；另一方面，系统软件硬件的异构度提高，软件在系统中所占的比例也获得了极大的增加，但是系统的上市时间却要求更短。在早期的系统设计中，往往只是着眼于硬件设计，因而出现了硬件设计语言，随着系统复杂度的增加，软硬件协同的概念被引入诸如功能-结构方法等设计方法学中。

更短的系统上市时间 TTM(Time to Market)和更复杂的系统设计要求更高的设计、描述效率和工程管理技术。一个解决方法就是提高设计和描述的抽象等级，在算法级对目标系统进行设计，进一步发展到对系统功能进行描述。对于一个系统的功能和行为，在设计最初阶段，描述时尚无硬件的概念，经过软硬件划分之后，将系统规范描述分成软件实现和硬件实现两部分。虽然VHDL等语言也支持算法级描述，然而大部分硬件描述语言HDL(Hardware Description Language)，如 VHDL，Verilog 等，基本上还是面向硬件的描述，面向较低的硬件抽象等级。所以要寻找更适合于描述算法的语言描述系统。C 语言或其扩展成为当前系统设计中最主要的系统级规范语言。

首先，用 C 作为系统规范描述语言，解决了前面提到的硬件描述语言的缺点。

其次，由于软件设计在系统设计中的比例越来越大，需要更多的软件系统工程师参加系统设计。设计自动化系统的实现使得这些设计工具的用户逐

渐扩大到非计算机技术人员，由于他们对C语言远比硬件描述语言熟悉，所以采用C语言来描述系统可以提高工作效率。

另外，许多要用硬件实现的功能，例如快速傅立叶变换FFT、扫描算法、消隐算法等这些成熟的常用的算法都有现成的C语言的描述，如果可以复用这些资源将节省更多的设计时间[1]。

对很多EDA工具只能接受HDL语言的描述，它是设计者和EDA之间的界面，设计者通过HDL语言将自己的设计方案输入给EDA工具，在EDA工具的帮助下进行模拟、综合和验证。

所以用C语言的描述要转化为HDL语言才能被EDA工具接受[2]。在众多HDL语言中VHDL语言是大家广泛使用的，它也具有一些描述算法的能力，但远不及C语言描述算法的能力。所以实现C语言到VHDL的转换具有重要意义和很大的实用价值。

1.2 编译程序的发展历程

第一个编译程序的出现大约在20世纪50年代早期，此时的编译工作是将算术公式翻译成机器代码。用现在的标准来衡量，当时的编译程序能完成的工作十分初步，如：只允许简单的单目运算，数据元素的命名方式有很多的限制。然而它们奠定了对高级语言编译系统的研究和开发的基础。20世纪50年代中期出现了FORTRAN等一批高级语言，相应的一批编译系统开发成功。随着编译技术的发展和社会对编译程序需求的不断增长，20世纪50年代末有人开始研究编译程序的自动生成工具，提出并研制编译程序的编译程序[3]。它的功能是以任意语言的词法规则、语法规则和语义解释出发，自动产生该语言的编译程序。目前很多自动生成工具已广泛使用，如词法分析程序的生成系统flex++，语法分析程序的生成系统bison等[4]。20世纪60年代起，不断有人使用自展技术来构造编译程序。自展的主要特征是用被编译的语言来书写该语言自身的编译程序[5]。1971年，PASCAL的编译程序用自展技术生成后，其影响就越来越大。

随着编译器的不断发展，编译器的使用越来越广泛，并呈现新的特点。

首先，编译器包括了更为复杂的算法，它用于推断或简化程序中的信息；这又与更为复杂的程序设计语言的发展结合在一起。其次，编译器已越来越成为基于窗口的交互开发环境 IDE 的一部分，它包括了编辑器、链接程序、调试程序以及项目管理程序。这样的 IDE 的标准并没有多少，但是已沿着这一方向对标准的窗口环境进行开发了。

随着并行技术和并行语言的发展，处理并行语言的并行编译技术正在深入研究之中，将串行程序转换成并行程序的自动并行编译技术也正在深入研究之中。

1.3 课题的主要研究内容

本论文主要的研究内容由以下几个部分组成：

(1) 词法分析

为源语言设计正则表达式，并为词法分析部分设计主要的数据结构，通过词法分析器的生成工具构建词法分析程序。通过词法分析程序对源程序进行词法分析。

(2) 语法分析

为源语言设计文法产生式。并为语法分析的输出结果设计相应的数据结构，以便后续编译工作能在此基础上进行。通过语法分析器的生成工具构建语法分析程序。通过语法分析程序对词法分析的输出结果进行语法分析，生成抽象语法树。

(3) 将抽象语法树转换生成目标代码

提出编译器的内部表示模型 CDFG，并设计相应的算法对抽象语法树进行转换，从而生成源程序对应的内部表示模型 CDFG，并对 CDFG 进行转换生成目标代码。

1.4 本论文的组织结构

本论文分为 6 章，组织方式如下：

第 1 章介绍编译器的国内外发展现状和研究动态，简述本课题的研究背

景和意义，并给出论文的主要工作和组织结构。

第 2 章介绍 VHDL 语言及相关特点，分析了 C 语言与 VHDL 语言的区别，并介绍了编译的作用及编译的步骤，并简单分析了编译器中的主要数据结构。

第 3 章首先根据课题的需要给出了面向 VHDL 算法级行为描述的编译方法总体概述，接下来分别概述了编译器前端的词法分析模块的设计和语法分析模块的总体设计，最后介绍了编译器的后端的总体设计。

第 4 章详细阐述了由源程序到抽象语法树的生成过程。首先给出了词法分析模块的构建过程，介绍了词法分析器的生成器 flex++，并阐述了如何通过 flex++生成词法分析器，即正则表达式的设计，以及词法分析程序采用的主要数据结构和词法分析程序的主要算法。然后阐述了语法分析模块的构建过程，介绍了语法分析器的生成器 bison，阐述了通过 bison 生成语法分析器的方法，即产生式的设计，以及语法分析程序采用的主要数据结构和语法分析程序的主要算法。最后对抽象语法树进行了简要概述。

第 5 章对阐述了基于抽象语法树生成目标代码的过程。首先给出了编译器的内部表示模型 CDFG，然后阐述了生成 CDFG 的方法，最后介绍了如何将 CDFG 进行转化生成目标代码的方法。

第 6 章通过对具体的程序进行试验及仿真验证了编译过程的正确性。

第2章 VHDL及相关技术

2.1 VHDL语言简介

2.1.1 VHDL 概念及特点

VHDL 是 VHSIC(very high speed integrated circuit) Hardware Description Language (超高速集成电路硬件描述语言)的缩写，诞生于 1982 年。1987 年底，VHDL 被 IEEE(The Institute of Electrical and Electronics Engineers)和美国国防部确认为标准硬件描述语言。自 IEEE 公布了 VHDL 的标准版本（IEEE-1076）以后，各 EDA 公司相继推出了自己的 VHDL 设计环境，或宣布自己的设计工具可以和 VHDL 接口。此后 VHDL 在电子设计领域得到了广泛的接受，并逐步取代了原有的非标准硬件描述语言。1993 年，IEEE 对 VHDL 进行了修改，从更高的抽象层次和系统描述能力上扩展了 VHDL 的内容，公布了新版本的 VHDL，即 IEEE 标准的工业标准硬件描述语言，又得到了众多 EDA 公司的支持，在电子工程领域，已经成为事实上的通用硬件描述语言。

VHDL 语言是一种功能全面的硬件描述语言，支持数字系统设计的设计、验证、综合、测试，以及硬件设计数据的交换、维护和修改，它可以从行为、功能或结构等不同的层次描述硬件电路[6]。由于 VHDL 语言可以用前后一致的语法和语义描述数字系统的多个层次，并且可以在同一设计中混合各种层次的描述，所以它特别适合于对同一系统的不同层次的设计实现进行模拟验证，以支持系统的维护和重新设计[7]。在 Top-Down 设计的全过程中，设计者均可方便地使用同一种语言。VHDL 设计并不十分关心一个具体逻辑是靠何种方式实现的，设计人员不需通过门级原理图描述电路，而是针对目标进行功能描述，把开发者的精力集中到逻辑所实现的功能上，将设计人员的工作重心提高到了系统功能的实现与调试上，只需花较少的精力用于物理实现。由于摆脱了电路细节的束缚，设计人员可以专心于设计方案和构思，使得设

计工作省时省力，加快了设计周期。

VHDL 语言的主要特点：

(1) 与其他的硬件描述语言相比，VHDL 具有更强的行为描述能力。强大的行为描述能力避开了具体的器件结构，是在逻辑行为上描述和设计大规模电子系统的重要保证[8]。VHDL 的宽范围描述能力使它成为高层次设计的核心，从而决定了它成为系统设计领域最佳的硬件描述语言，并可进行系统的早期仿真以保证设计的正确性。

(2) VHDL 丰富的仿真语句和库函数，使得在任何大规模系统的设计早期就能查验设计系统功能的可行性，随时可对设计进行仿真模拟[9]。

(3) VHDL 语句的行为描述能力和程序结构，决定了它具有支持大规模设计的分解和对已有设计的复用功能[10]。

(4) 对于用 VHDL 完成的一个确定的设计，可以利用 EDA 工具进行逻辑综合和优化，并自动地把 VHDL 描述设计转变成门级网表[11]。

(5) VHDL 对设计的描述具有相对独立性，设计者可以不懂硬件的结构，也不必管最终设计实现的目标器件是什么，而进行独立的设计。

(6) VHDL 的设计不依赖于特定的器件，方便了工艺的转换。

(7) VHDL 是一个标准语言，为众多的 EDA 厂商支持，移植性好。

VHDL 描述系统时，一般有三种描述层次：行为描述、数据流描述、结构描述。行为级描述注重描述对象的功能，表示输入与输出之间的转换行为，不包含结构信息；数据流描述用于描述对象的关系；结构描述直接给出实体实现的逻辑网表。

2.1.2 VHDL 的能力范围

VHDL 既可以被计算机阅读又可以被人阅读，它支持硬件的设计、验证、综合和测试。此外，它还支持硬件设计数据的交换、维护、修改和硬件的实现。

VHDL 支持行为领域和结构领域的硬件描述，并且可以从最抽象的系统级一直到最精确的逻辑级。VHDL 主要优点之一是：在描述数字系统时，可以使用前后一致的语义和语法跨越多个层次，并且使用跨越多个级别的混合描述模拟该系统。因此，可以对由高层次行为描述子系统及低层次详细实现

子系统所组成的系统进行模拟。目前 VHDL 还不具有描述模拟电路的能力。

2.1.3 VHDL 的三种描述风格

VHDL 之所以支持多种建模方法，是因为它支持三种描述风格。三种描述风格是：行为描述、数据流描述、结构描述[2]。有时可采用由它们组合成的混合描述。

2.1.4 C 与 VHDL 的语言的区别

C 语言是程序员熟悉的高级语言，适合于描述过程和算法，运行环境是图灵机模型。它有着很多灵活的描述方式。这些描述方式有的是为了书写程序时方便，C 语言更适合描述串行的程序。

VHDL 语言是硬件描述语言，可以结构化的分层次的描述电路结构特性，可以表达电路块输入输出信号的相互关系的行为特性，还可以描述和芯片布图有关的几何特性。适合对并发程序的描述。

2.2 编译技术

2.2.1 编译的作用

编译器是将一种语言翻译为另一种语言的计算机程序。编译器将源语言编写的程序作为输入，而产生用目标语言编写的等价程序。这一过程可以用图 2.1 表示：

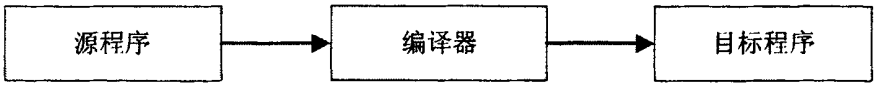


图 2.1 编译器的功能

现代编译器常常由多个阶段组成，在下文中给出了通常的编译步骤。

2.2.2 编译步骤

编译器内部包括了许多步骤或称为阶段(phase)，它们执行不同的逻辑操作。将这些阶段设想为编译器中一个个单独的片断是很有用的，尽管在应用中它们是经常组合在一起的，但它们确实是作为单独的代码操作来编写的。图 2.2 描述了编译器中的阶段和与以下阶段(文字表、符号表和错误处理器)

或其中的一部分交互的 3 个辅助部件。

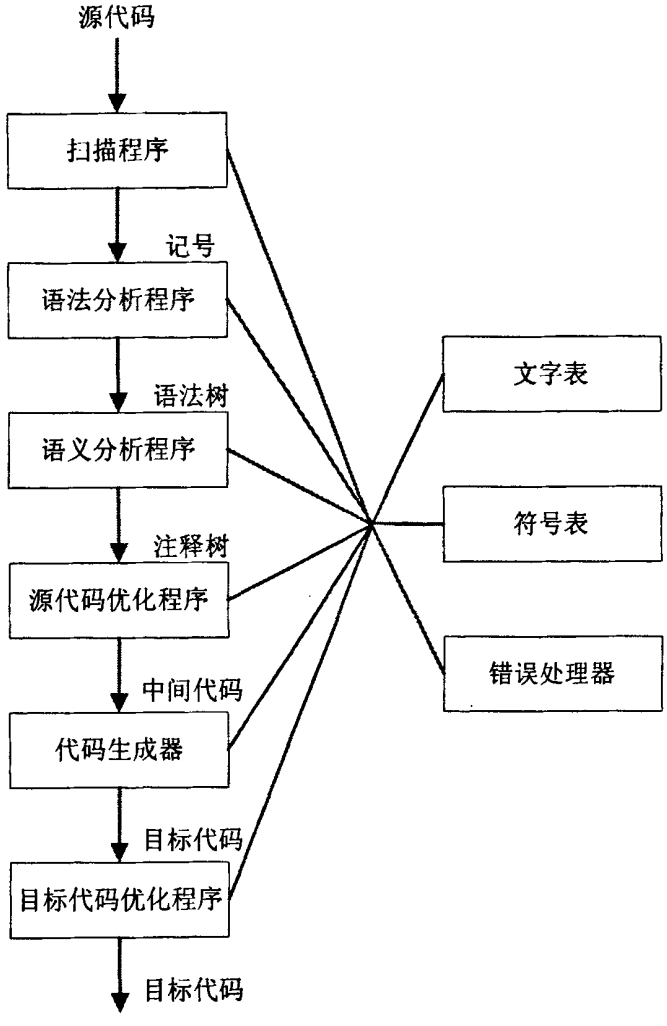


图 2.2 编译器的阶段

2.2.3 编译器中的主要数据结构

当然，由编译器的各个阶段所使用的算法与支持这些阶段的数据结构之间的交互是非常强大的。本节将讲述一些主要的数据结构，它们是编译过程所需要的，并用来在各段中交流信息。下面主要介绍三种编译过程中的三种数据结构：

(1) 记号(token)

当扫描程序将字符收集到一个记号中时，它通常是以符号表示这个记号；

这也就是说，作为一个枚举数据类型的值来表示源程序的记号集。有时还必须保留字符串本身或由此派生出的其他信息(例如：与标识符记号相关的名字或数字记号值)。在大多数语言中，扫描程序一次只需要生成一个记号(这称为单符号先行(single symbol look ahead))。在这种情况下，可以用全程变量放置记号信息；而在别的情况下，则可能会需要一个记号数组。

(2) 语法树(syntax tree)

如果分析程序确实生成了语法树，它的构造通常为基于指针的标准结构，在进行分析时动态分配该结构，则整棵树可作为一个指向根结点的单个变量保存。结构中的每一个结点都是一个记录，它的域表示由分析程序和之后的语义分析程序收集的信息。例如，一个表达式的数据类型可作为表达式的语法树结点中的域。有时为了节省空间，这些域也是动态分配或存放在诸如符号表的其他数据结构中，这样就可以有选择地进行分配和释放。实际上，根据它所表示的语言结构的类型(例如：表达式结点对于语句结点或声明结点都有不同的要求)，每一个语法树结点本身都可能要求存储不同的属性。在这种情况下，可由不同的记录表示语法树中的每个结点，每个结点类型只包含与本身相关的信息。

(3) 中间代码(intermediate code)

根据中间代码的类型和优化的类型，该代码可以是文本串的数组、临时文本文件或是结构的连接列表。对于进行复杂优化的编译器，应特别注意选择允许简单重组的表示。

2.3 本章小结

VHDL 语言的产生反映了硬件描述语言向标准化发展的必然趋势，VHDL 语言高级综合技术的研究是当今逻辑设计自动化的活跃领域。作为整个高级综合系统前端的 VHDL 语言编译系统在 VHDL 支撑系统中具有重要作用与地位。本章简要介绍了 VHDL 语言，分析了 VHDL 与 C 语言的区别，并对编译技术做了简要的概述。

第3章　编译方法整体设计

前面已经介绍了编译方法的基本理论知识及相关技术，为了实现本课题的要求，在这一章中，根据本课题的特点，对面向 VHDL 算法级行为描述的编译方法进行了整体设计。

3.1　面向VHDL算法级行为描述的编译方法概述

进行编译的源语言是C语言，目标语言是VHDL。可以说这是两种面向不同领域，各有所长的语言。要把C语言描述的算法用硬件来实现，就是要把它翻译成VHDL语言描述的算法。为了将一个程序从一种语言翻译成另一种语言，编译器必须首先把程序的各种成分拆开，并清楚其结构和含义，然后再用另一种方式把这些成分组合出来，编译器的前端执行分析，后端进行合成。

本课题实现的是一个由一种高级语言向另一种高级语言翻译的编译器：首先由编译器的前端完成词法分析、语法分析，生成抽象语法树；然后由编译器的后端对抽象语法树进行转换生成内部表示模型CDFG，之后基于CDFG生成目标代码。将C和VHDL的规约规则一一对应是很难的，毕竟这是两种差别很多的语言。编译流程见图3.1：

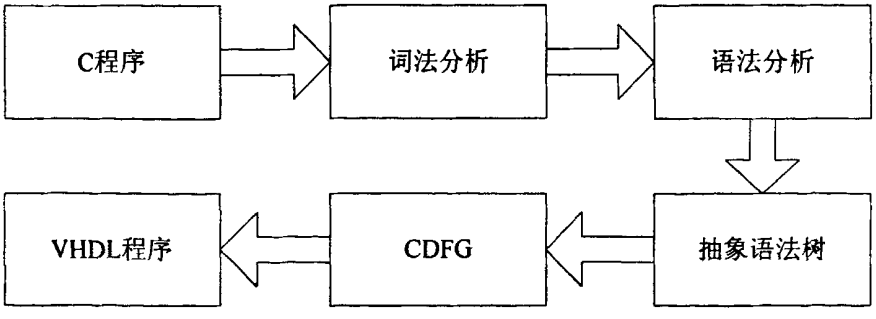


图3.1　C到VHDL转换过程

本文将编译程序划分编译前端和编译后端。前端包括词法分析、语法分析，源程序经过前端的处理被转换为抽象语法树；后端包括两个阶段，即内

部表示模型的生成和目标代码的生成。下面分别简要介绍了编译器的前端和后端。

3.2 编译器前端

编译的前端包括两个阶段：词法分析和语法分析。词法分析的任务是分析和识别记号。源程序是由字符序列构成的，词法分析程序依次扫描源程序中的每个字符，根据语言的词法规则识别出一个一个具有独立意义的最小语法单位，即记号(token)。语法分析的任务是分析和识别各种语法成分。在词法分析的基础上，根据语言的语法规则从单词符号串中识别出各种语法单位。本文将语法分析的结果表示为抽象语法树。下面分别简要介绍了本文的词法分析的设计和语法分析的设计。

3.2.1 词法分析

任何合理的程序设计语言都可以用来实现特定的词法分析器。通常用正则表达式的形式语言来指明词法记号，用有限自动机来实现词法分析器，并用数学的方法将两者联系起来[10,11]。构造有限自动机是一种机械性的工作，很容易由计算机来实现，因此本文利用词法分析器的自动生成器来将正则表达式转换为有限自动机，即构造出词法分析器。本文的词法分析的过程可由图 3.2 表示。

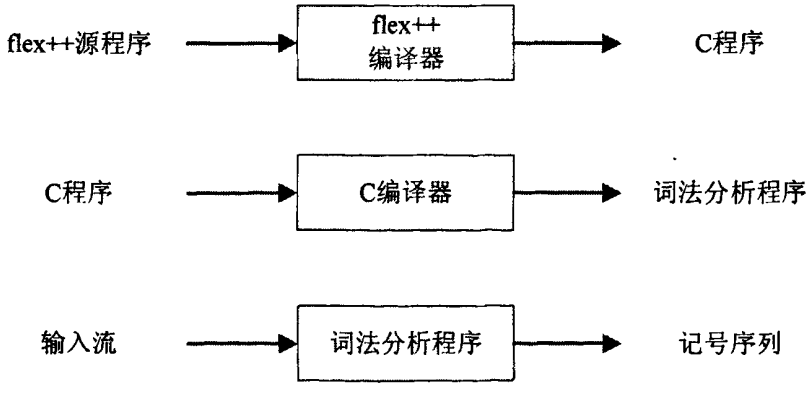


图 3.2 用 flex++建立词法分析程序

本文的词法分析程序的构造是通过词法分析器的生成工具 flex++来完成

的。给定一组描述记号词法结构的正则表达式，以及识别出各类记号时词法分析程序应采取的语义动作，通过 flex++自动地构造出相应的词法分析程序。因此，构造词法分析模块的主要工作即为设计源语言的正则表达式，并设计相应的数据结构以提高词法分析的效率。利用 flex++构造词法分析器的具体构造过程会在地第四章中详细阐述。

3.2.2 语法分析

语法分析程序以记号形式的源程序作为输入或分析的对象。其基本任务是根据语言的语法规则，分析源程序的语法结构(即分析如何将这些单词组成各种语法成分，如各种表达式、语句、函数等)。其输出形式有多种，本文采用的是抽象语法树(AST)[12]。图 3.3 给出了语法分析程序的作用及其在编译程序中的位置。

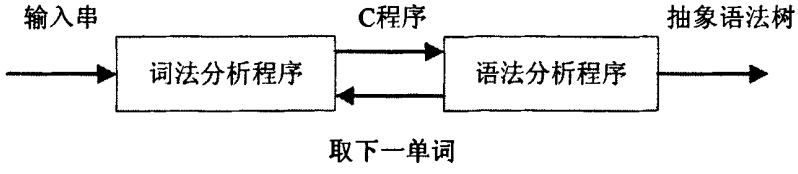


图 3.3 语法分析程序的作用及其在编译程序中的位置

语法分析方法分为自顶向下分析法和自底向上分析法，其中每一类又包括不同的分析方法，本文采用的是自底向上的分析法 LALR(1)[13]。本文的语法分析程序的构造是通过语法分析器的生成工具 bison 来完成的。这一过程可由图 3.4 表示。

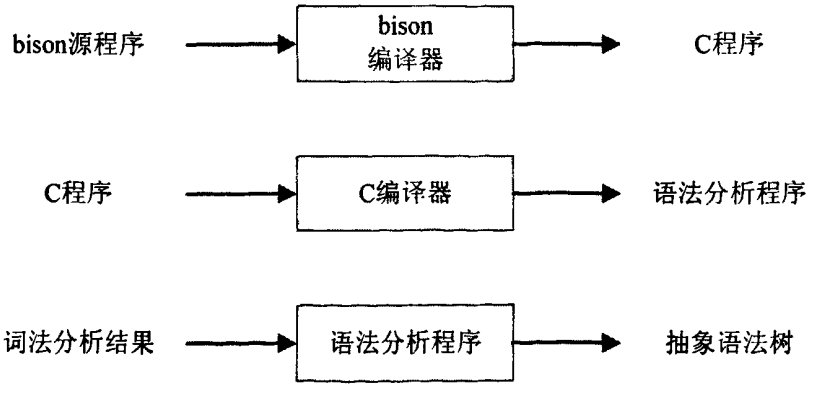


图 3.4 用 bison 建立词法分析程序

给定一组描述文法的文法产生式，以及每个产生式对应的语义动作，即可通过 bison 自动地构造出相应的语法分析程序。因此，构造语法分析模块的主要工作即为设计源语言的文法产生式，并设计相应的数据结构用以表示输出结果。在第四章中将给出语法分析模块的具体构造方法。

3.3 编译器后端

编译器的后端是在编译器的前端生成的抽象语法树的基础上进行的，为了将 C 语言描述的源程序转换成 VHDL 代码，本文设计了一种中间形式-控制数据流图(CDFG)，首先将由前端生成的抽象语法树转换为中间形式 CDFG，然后，对 CDFG 进行转换生成 VHDL 代码，从而完成编译的全部工作。

3.3.1 内部表示模型

本文的编译器的目标代码是 VHDL 代码，首先将源语言转换为一种内部表示形式，这种内部表示形式要方便最后生成目标代码，考虑目标语言 VHDL 的特点，本文设计了一种便于生成目标代码的内部形式 CDFG。内部表示是编译系统的机内表示形式，是进行编译的基础。通过对词法分析器的输出结果抽象语法树的分析将其转换生成对应的 CDFG。

3.3.2 目标代码生成

编译器的后端采用相应的算法，首先将抽象语法树转换成 CDFG，再利用已有的软件将 CDFG 转换为目标代码。

3.4 本章小结

本章介绍了面向 VHDL 算法级行为描述的编译方法的整体设计，并简要介绍了编译器的前端的设计，包括构造词法分析模块和语法分析模块的设计；编译器的后端的设计，包括内部表示模型的设计和目标代码生成模块的设计。

第4章　从源程序到抽象语法树的转换

编译器的前端工作是将源程序转换成抽象语法树，即对源程序进行词法分析和语法分析，本章介绍了本文的编译方法中的词法分析和语法分析部分。词法分析是编译的第一个阶段，它的主要任务是从左至右逐个字符地对 C 语言源程序进行扫描，产生一个个单词序列，也可称为 Token 记号。语法分析是编译过程的第二个阶段。语法分析的任务是在词法分析的基础上，根据语言的语法规则，把单词符号分解成各类语法单位，如“短语”、“句子”、“子句”、“程序段”等等。经过前端的词法分析和语法分析输入的源程序被转换成抽象语法树。

4.1　词法分析

词法分析是通过词法分析程序来进行的，本文采用的词法分析程序是借助于词法分析程序的自动生成工具 flex++构建的，通过将正则表达式描述的构词规则写入 flex++源程序，利用词法分析程序的生成工具 flex++生成词法分析程序，生成流程如图 4.1 所示。

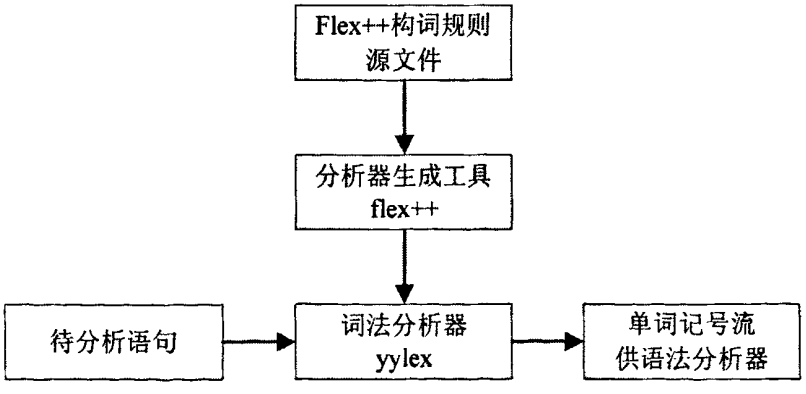


图 4.1 词法分析器生成流程图

在 flex++构词规则源文件中写入 C 语言的正则表达式，该源文件经过词法分析程序生成工具 flex++转换为词法分析程序 lex.yy.c。待分析的 C 语言语句，经过 lex.yy.c 文件扫描后，转化为单词记号流。因此词法分析器部分的

设计主要是正则表达式设计和数据结构的设计。正则表达式的设计主要是针对源语言的，本文将 C 语言作为源程序的描述语言，所以本文是对 C 语言进行正则表达式的设计，具体的设计方法将在下面详细阐述。数据结构部分的设计主要针对关键字的存放，具体实现的数据结构可以是数组，链表，哈希表等，它的主要作用是提高词法分析器的效率，本文采用的数据结构是哈希表。在构造词法分析程序的过程中，主要的工作是设计源语言的正则表达式，通过词法分析程序的生成工具 flex++生成词法分析程序，并为词法分析程序设计相应的数据结构。下面将分别介绍词法分析程序的生成工具 flex++，本文给出了 C 语言中记号的正则表达式的设计，以及词法分析程序采用的数据结构，并对词法分析程序的算法做了简要的介绍。

4.1.1 词法分析程序的生成器 flex++

本文的词法分析程序是通过词法分析程序的生成器 flex++生成的，所以下面我们简要介绍一下 flex++[16]。

(1) flex++原理

flex++是词法分析程序的生成器。用它可以很方便的描述词法分析器，并自动产生出相应的词法分析程序。

flex++源程序是用一种面向问题的语言写成的。这个语言的核心是正则表达式，用它描述输入串的词法结构。在这个语言中用户还可以描述当某一个词形被识别出来时要完成的动作。

flex++自动地表示把输入串词法结构的正则表达式及相应的动作转换成一个宿主语言的程序，即词法分析程序，它有一个固定的名字 yylex，在本文中 yylex 是一个 C 语言的程序。yylex 将识别出输入串中的词形，并且在识别出某词形时完成指定的动作。

flex++的工作原理是将源程序中的正则表达式转换成相应的确定有限自动机，而相应的动作则插入到 yylex 中适当的地方，控制流由该确定有限自动机的解释器掌握，不同的源程序，这个解释器是相同的。

由词法自动生成系统工作原理可知，在词法部分的主要工作是书写 C 语言的 flex++源程序，主要构造 C 中所有单词字符的正则表达式以及单词被识别后所采取的相应的动作。

(2) flex++源程序的结构

flex++源程序的一般格式是:

{辅助定义的部分}

%%

{识别规则部分}

%%

{用户子程序部分}

其中用花括号括起来的各部分都是可选的。%%把整个脚本分成 3 节，当没有"用户子程序部分"时，第三节与第二节的%%也可以省略。但第一节和第二节之间的%%是必须的，因为它标志着识别规则部分的开始，最短的合法的 flex++源程序是：

%%

它的作用是将输入串原样抄到输出文件中。识别规则部分是 flex++源程序的核心。它是一张表，左边一列是正则表达式，右边一列是相应的动作。

通过对词法分析程序的生成工具 flex++的介绍，可以知道借助于 flex++生成词法分析程序过程中主要的工作是为源语言设计正则表达式，即为 C 语言设计正则表达式，下面给出本文设计的 C 语言的 flex++源程序中的正则表达式。

4.1.2 C 语言中记号的正则表达式的结构

由上文可以知道通过 flex++生成词法分析程序的主要工作是在 flex++的源程序中写入识别规则，即正则表达式，本节给出本文设计的正则表达式。由于进行词法分析时扫描程序要按照输入语言的格式说明来进行，本文的输入语言是C语言,所以下面给出本文采用的C语言中记号的正则表达式结构。本文将 C 语言中的记号分为这样几类：关键字、标识符、文字和常量、特殊符号、预处理程序命令、注释、定界符。

(1) 关键字和标示符

C 语言中有 32 个关键字，由于关键字和标识符的结构是相同的，所以考虑到构造关键字表，将关键字和标识符一起作为单词来处理，识别出单词后，再查关键字表，如果其在关键字表中，则返回关键字，否则返回标识符的

token。

通常，标识符必须由一个字母或者下划线开头且只包含字母、下划线和数字。可用以下的正则定义表示：

letter = [a-zA-Z]

digit = [0-9]

得到关键字和标识符的正则表达式：

ID = (letter|"_") (letter | digit|"_") *

(2) 常量

C 语言中的常量包括数字常量、字符串及字符等。在 C 语言中字符包含在单引号内部，其可以是字母、数字、单个的特殊符号及转义字符。转义字符是以一个"\"开头的字符序列。因此其正则表达式可以写成：

ZF =

字符串包含在双引号内部。其可以是字母、数字、单个的特殊符号的组合。其正则表达式是：

ZFC = "\""([0-9A-Fa-f+-*!@#$%^&<>?{}]+)

数字常量可以分为整数、浮点数、二进制串、十六进制串、八进制串几类，正则表达式分别是：

DIGIT0 = [0-9]

DIGIT1 = [1-9]

ZNUMBER = [+\-]?{DIGIT1}{DIGIT0}*|0

FLOAT = ({ZNUMBER} (\.{DIGIT0}+)?)(E[+\-]?{ZNUMBER}+)?

Bitstring = "'" (0|1) *"'" "B"

Hexstring = "'" ([0-9A-Fa-f]) *"'" "H"

Octetstring = "'"(([0-9A-Fa-f]) +)"'" "O"

(3) 特殊符号

对于这些特殊符号一般直接返回即可，因为本设计只是将扫描到的记号返回到一个文件中，因此不需要对一些特殊的字符作出处理，比如"{"和"}"等。在这就不对特殊字符一一列出。

(4) 预处理程序命令

C 程序的源代码中可包括各种编译指令，这些指令称为预处理命令。虽

然它们实际上不是C语言的一部分，但却扩展了C程序设计的环境。所有预处理命令均以符号＃开头。这就涉及到flex++提供的处理上下文相关的措施。

flex++提供的处理上下文相关的措施要处理的问题是对某些规则在不同的上下文中采取不同的动作，或者说同样的字符串在不同的上下文中有不同的解释。flex++提供了两种主要的方法：第一种方法是使用标志来区分不同的上下文。标志是用户定义的变量，用户在不同的上下文中为它指定不同的值，以区分它在哪个上下文中，这样识别规则就可以根据标志当前值决定在哪个上下文中并采取相应的动作。第二种方法是使用开始条件来区分不同上下文。在flex++源程序中用户可以用名字定义不同的开始条件。当把某个开始条件置于某条识别规则之前时，只有在flex++处于这个开始条件下这条规则才起使用，否则等于没有这条规则。flex++当前所处的开始条件可以随时由用户程序改变。开始条件由用户在flex++源程序的“辅助定义部分”定义，语法是%Start name1 name2 name3...。其中Start可以缩写成S或s。开始条件名字的顺序可以任意给出，有很多开始条件时也可以由多个%Start行来定义它们。开始条件在识别规则中的使用方法是把它用尖括号括起来放在识别规则的正规式左边：＜name1＞expression。要进入开始条件如name1，在动作中用语句BEGIN name1。它将flex++所处的当前开始条件改成name1要恢复正常状态，用语句BEGIN 0。一条规则也可以在几个开始条件下都起作用。要使一条规则在所有开始条件下都起作用，就不在它前面附加任何开始条件。

(5) 注释

注释在编译过程中并无实际的意义。在词法分析部分我们用正则表达式将其识别出来，然后过滤。在C语言中，有单行注释“//”和多行注释“/*.....*/”两种。对于单行注释我们可以简单的写出其正则表达式，但是对“/*.....*/”其中的分隔符多于一个字符，而且我们还要保留注释中的行号，以便输出正确的错误信息。因此编写正则表达式时就要困难很多。因此，实际上，这种情况在真正的扫描程序的过程中，经常是通过特殊办法解决的，我们连着用多条规则将注释的各种情况表示出来，并且提供搜索“\n”的行为代码及一些额外处理的代码。

(6) 定界符

对于定界符主要有制表符、空格符、换行符。因为C是自由格式的，所以空格和制表符在编译时是没有实际意义的。所以在识别过程中将他们过滤掉。换行符本身没有什么意义，但是对于错误处理有重要作用，因此换行符不能删掉。

以上是本文的编译器进行词法分析时采用的正则表达式，在进行词法分析的过程中，要对识别出的记号进行区分，当扫描器扫描到一个有数字和字母组成的记号时，首先要区分该记号是表达式还是关键字，这样就要把C语言的关键字预先存放起来，进行区分时到关键字中去查找该记号，从而确定记号的属性。对于关键字要采用一定的数据结构使得进行查找时方便，快速，下面介绍本文存放关键字所采用的数据结构。

4.1.3 数据结构的设计与实现

在本文的词法分析部分用到的主要数据结构是哈希表。利用哈希表来存放C语言中的关键字，当扫描器扫描到一个有数字和字母组成的记号时，首先查询此表，如果查找到就返回：此记号是关键字，否则返回标识符。因为哈希表的效率很高，并且在哈希表中的结果就30几个，查找的时间几乎可以忽略不计，使用哈希表，可以节省大量的正则表达式，大大提高了扫描的效率。

可如下描述哈希表：根据设定的哈希函数H(key)和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集(区间)上并以关键字在地址集中的“象”作为相应记录在表中的存储位置，这种表被称为哈希表。本设计解决冲突的方法是将冲突的关键字放在一个链表中，然后对其进行依次比较。

本文词法分析部分的的数据结构也可以叫做关键字表，就是将C语言中的关键字放在哈希表里面， 为了让扫描的效率做高，完成的哈希函数是：

```
unsigned int hash(char *str){
    unsigned int h;
    unsigned char *p;
    h = 0;
    for(p=(unsigned char*)str; *p!='\0'; p++){
```

```
        h = MULTIPLIER*h + (*p);
    }
    return h%NHASH;
}
```

当我们获得一个符号的时候，首先建立哈希表，程序如下：

```
void createHashTable(){
    Nameval *temp;
    int i = 0;
    unsigned int h;
    for(i=0; i<NHASH; i++){
        symtab[i] = NULL;
    }
    for(i=0; i<KEYWORDNUMBER; i++){
        h = hash(keyWord[i]);
        temp = new Nameval;
        temp->name = keyWord[i];
        temp->next = symtab[h];
        symtab[h] = temp;
    }
}
```

然后执行哈希函数，查找关键字表，然后执行相应的动作。查询函数是

```
int   lookup_word(char *word){
    Nameval *p;
    unsigned int h = hash(word);
    for(p=symtab[h]; p!=NULL; p=p->next)
   {
        if(strcmp(word, p->name)==0)
            {return 1;}
    }
   return 0;
```

```
}
```

如果此符号存在关键字表中，将此符号返回为关键字，否则返回一个标识符，程序如下：

```
switch(lookup_word(yytext))
        {
            case 1:   i++;
                      strcpy(out[i][0],"KEYWORD");
                      strcpy(out[i][1],yytext);
                      break;
            case 0:   i++;
                      strcpy(out[i][0],"ID");
                      strcpy(out[i][1],yytext);
                      break;
```

以上是本文为存放关键字所设计的数据结构，采用哈希表来存放关键字提高了查找效率，从而也提高了词法分析的效率。生成词法分析程序的主要工作即通过将正则表达式写入 flex++，并为词法分析程序设计相应的数据结构，从而利用词法分析程序的生成工具 flex++得到词法分析程序，通过词法分析程序对源程序进行词法分析。下面简要介绍一下词法分析程序的算法。

4.1.4 词法分析程序的算法及其说明

词法分析程序对源程序进行词法分析。在本文的编译系统中，词法分析程序是一个子程序，每当语法分析程序需要一个单词时，则调用该子程序。词法分析程序每被调用一次，便从源程序文件读入一些字符，直至识别出一个单词，或说是直至下一个单词的第一个字符为止。词法分析程序是通过调用 C 的标准库函数 getc (finput), ungetc (c, finput)来实现从源程序文件读取字符到单词的缓冲区及将字符放回到输入源程序文件的。图 4.2 是本文所采用的词法分析程序的流程图。

本文的词法分析程序的功能主要通过函数 yylex 来实现，yylex 调用标准库函数 getc 从源程序文件中逐个读取字符，该源程序文件是预处理的输出文件，并对该字符进行分析处理。

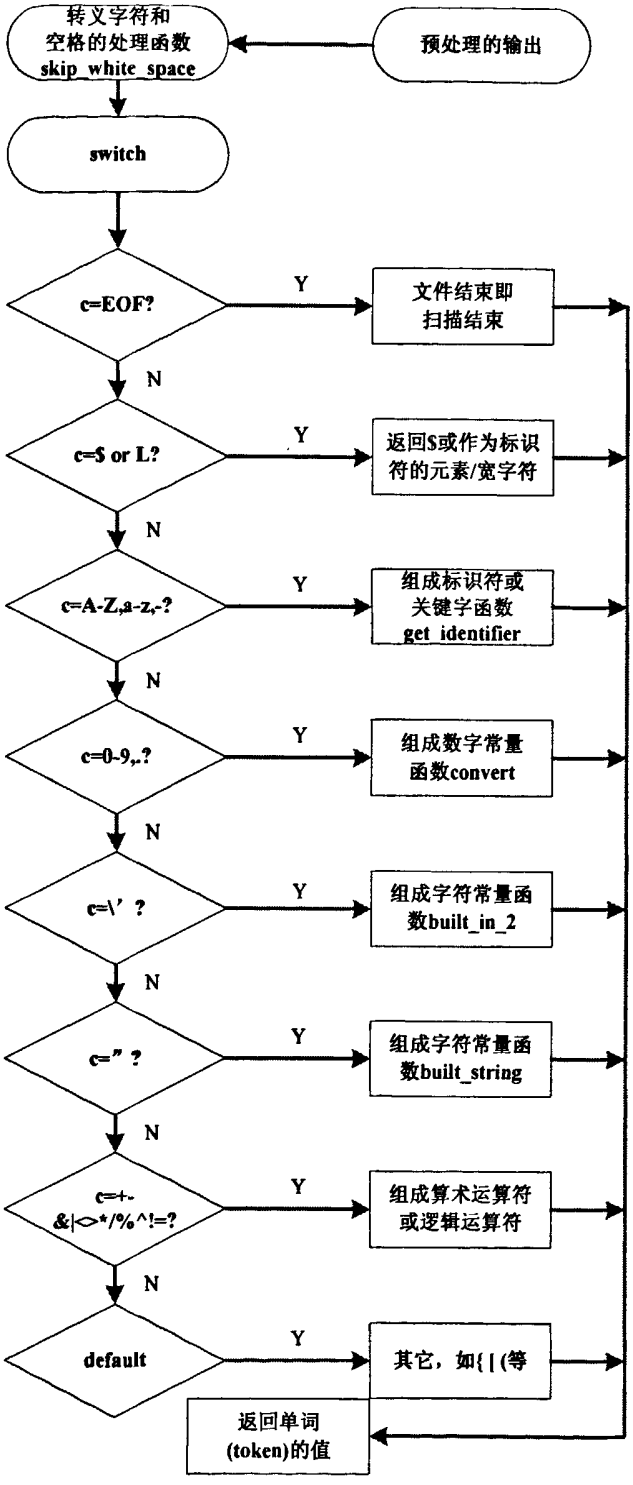


图 4.2 词法分析程序流程图

首先函数 yylex 判别读入的字符是否为转义字符，如： '\t',, 'T',, '\v',

'\b'等，若是转义字符，则跳过转义字符，然后再从源程序文件中读取字符。接着判别该字符是否为'\n'，'/'，'\\'等，若是则调用函数 skip_white_space 取得新字符。将读入的字符暂时放在临时变量中，并对该字符进行分析处理。

本文的词法分析程序中，将可能组成单词的字符分成八种情况分析处理，以下分别对每种情况作简要的说明。

EOF 是源程序文件的结束符号。当词法分析程序读到该字符时，就认为对源程序文件的扫描工作结束了，也可以说是词法分析结束了。

'L'是宽字符常量或是宽字符串常量的起始标志。如果 L 后是'\'，则是宽字符常量。如果 L 后是'"'，则是宽字符串常量。

'$'是美元符号。如果标志位被置 1，则它作为组成标识符的字符元素，否则返回'$'。

由字符'A-Z'，'a-z'，'-'('$')，开始的单词是标识符。调用函数 is_ reserved_ word 在关键字(或保留字)表中查找此标识符是否存在，若找到则该标识符为保留字。否则为户自定义的标识符，调用函数 get_identifier 在 hash table 中查找该标识符，如果找到则返回指向此节点的指针，否则调用函数 make_node 建立一个节点，返回此节点的指针。

由数字'0-9'，'.'开始的单词构成各种数，包括整数，实数，指数，浮点数，复数等，有八进制，十进制和十六进制以及由'u'，'l'，'U'，'L'修饰的数。

由'\'"开始的单词是字符常量，即由单引号包含的字符是字符常量。

由'"'"开始的单词是字符串常量，即由双引号包含的单词是字符串常量。

由"+"、"-"、"&"、"|"、"<"、">"、"*"、"/"、"%"、"^"、"!"、"="构成算术运算符和逻辑运算符。

经过分析处理的字符将被放入单词的缓冲区内，函数 yylex 将此单词的值返回给调用它的语法分析程序，其属性值通过结构变量 yylval.ttype 等返回的。

通过以上的介绍可以清楚的了解词法分析程序的构造过程，以及词法分析的原理，源程序经过词法分析后要进行语法分析，下面我们介绍一下语法分析程序的构造以及语法分析程序的算法。

4.2 语法分析

在上一节中，介绍了如何构造词法分析程序等问题。一个字符串形式的源程序经过词法分析，即被转换为一串记号。编译程序在完成词法分析之后，就进入语法分析阶段[31]。

语法分析是编译过程的第二个阶段。语法分析的任务是在词法分析的基础上，根据语言的语法规则，把单词符号分解成各类语法单位，如“短语”、“句子”、“子句”、“程序段”等等[17]。一般这种语法短语，也称为语法单位，本文将其表示成抽象语法树。本文中对 C 语言进行语法分析的程序是在 bison 工具的辅助下生成的，C 语言描述的源程序经过语法分析后生成抽象语法树。

这一部分主要是利用 LALR(1)文法，编写 C 语言所涉及到的大量的语法产生式，再通过 bison 生成语法分析程序。完成的程序要达到：利用词法分析部分所产生的词法分析函数，扫描 C 语言程序，得到 token，再根据已编写的语法产生式，完成语法分析，同时得到该源程序的抽象语法树结构，作为后续编译工作的输入[36]。

下面将分别介绍主要使用的工具 bison，以及本文设计的 C 语言的产正式，以及设计中定义的关键的数据结构 TreeNode，最后对抽象语法树做一简要说明。

4.2.1 语法分析器的生成器 bison

由于本文的语法分析程序是借助于语法分析器的生成工具 bison 来生成的，所以首先介绍语法分析器的生成工具 bison[18]。

(1) bison 简介

bison 是一种通用目的的语法分析器生成器。它将 LALR(1)上下文无关文法的描述转化成分析该文法的 C 程序。bison 使用简单，功能强大。首先将文法写成 LALR(1)上下文无关文法，按 bison 脚本文件的书写要求输入文法，编译后即可生成可运行的语法分析程序。下面将详细介绍 bison 的特点及使用方法。

(2) bison 语法文件的整体布局

一个 bison 语法文件有四个主要的部分，由固定的分隔符分隔，如下所示：

```
%{
C  declarations
%}
Bison declarations
%%
Grammar rules
%%
Additional C code
```

其中'%%'，'%{'和'%}'是 bison 在每个 bison 语法文件中用于分隔部分的标点符号。bison 的四个主要部分为：C 语言声明部分、bison 声明部分、语法规则和附加的 C 代码部分。以下介绍各个部分的功能。

- C 语言声明部分

C 语言声明部分，可用来定义在动作中使用的类型和变量。可以使用预处理器命令在本节定义宏，或者使用#include 包含相关的头文件，或者声明全局变量。 同时需要在这里声明词法分析器 yylex 和错误打印程序 yyerror。当脚本文件经 bison 编译后，该部分的内容将全部复制到生成文件中，所以需要使用 C 语句来编写这个部分。

- bison 声明部分

bison 声明部分，声明终结符和非终结符的名字和数据类型，同时可以指出运算符的优先级和各个符号的语义值的数据类型。产生式中出现的所有的终结符都必须在这里声明；非终结符可以不声明，若要指明非终结符的数据类型，则必须在这部分明确写出；同时运算符的声明需要明确理解该运算符的优先级和结合性，声明的结果直接影响着产生式的操作。

- 语法规则部分

语法规则部分运用 LALR(1)文法编写文法规则，定义了如何从每一个非终结符的部分构建其整体的语法规则。这部分是脚本文件的核心，完成主要的语法分析任务。编写这部分时，须注意语法是不是 LALR(1)；所使用的终结符必须是已经声明过的。同时须编写操作，完成抽象语法树的存储。

- 附加的 C 代码部分

附加的 C 代码部分，可以包含一些程序需要的 C 语言程序。通常将词法

分析器 yylex 放在这里，还有一些在操作里用到的子程序都可以放在这里。这部分的内容将全部复制到所产生的文件中，使用 C 语言编写。

(3) bison 文件的设计过程

本节主要介绍如何设计 bison 的脚本文件，首先叙述了 bison 须采用的文法规则；接着介绍了如何按照 bison 的书写要求，将 LALR(1)文法的产生式写成脚本文件；同时描述了语义值和语义动作的特点；经过这些操作后，就可以编译生成语法分析文件了。

- 上下文无关文法

为了使 bison 能分析语言，这种语言必须由上下文无关文法(context-free grammar)描述。例如，在 C 语言中，有一种称之为'表达式(expression)'的语法组。一个生成的表达式的规则可能是'一个表达式由一个减号和另一个表达式构成'。另外一个规则可能是'一个表达式可以是一个整数'。就像看到的一样，规则经常是递归定义的，但是必须有一个结束递归的规则。用于表示这些规则的最普遍的系统是 Backus-Naur 范式（Backus-Naur Form），简写为 BNF。任何用 BNF 表示的文法都是一种上下文无关文法。bison 要求它的输入必须用 BNF 表示。

上下文无关文法有许多重要的子集。尽管 bison 可以处理几乎所有的上下文无关文法，但 bison 针对 LALR(1)文法做了优化。简而言之，在这些文法中，可以告之如何分析仅带有一个超前扫描记号的输入字符串的任意部分。严格的说，这是一个 LR(1)文法的描述。

- 从正规文法转换到 bison 的输入

正规文法是一种数学结构。为了定义 bison 要分析的语言，必须用 bison 语法编写一个表达该语言的文件，即一个 bison 语法文件。就像 C 语言中的标识符一样，在 bison 的输入中，一个正规文法的非终结符由一个标识符表示。根据惯例，非终结符应该用小写子母表示。

在 bison 中，终结符可以由类似 C 语言标识符来表示。根据惯例，这些标识符应改用大写字母表示以区分它和非终结符。一个表示某种语言的特定关键字的终结符应该由紧随该关键字之后的它的大写表示来命名。终结符 error 保留用作错误恢复之用。

一个终结符也可以由一个像 C 语言中的字符常量一样的一个字符来表示。

当一个记号就是一个字符（括号，加号等等）的时候，可以这样做：使用同一个字符作为该记号的终结符。

第三种表示终结符的方法是使用包含一些字符的 C 语言字符串常量。

- 语义值

正规文法仅仅靠类别来选择记号。例如，如果一个规则提到了终结符“integer constant”，这就意味着任何整数常量在那个位置上都是语法有效的。常量的精确值与如何分析输入不相关，如果“x+4”符合语法，那么“x+1”或者“x+3989”也符合语法。

但是，当对输入进行分析时，它的精确值是非常重要的，通过精确值可以了解输入的含义。如果一个编译器不能区别程序中的 4，1 和 3989 等常量，那么这个编译器是没有用的。因此，每一个 bison 语法的记号既含有一个符号类别也有一个语义值(semantic value)。

符号类型是在语法中定义的终结符，例如 INTEGER，IDENTIFIRE 或者‘，’。它决定记号可能有效出现的位置以及如何将它组合成其它记号的信息。语法规则只识别符号的类型。

语义值包括了记号的所有剩余信息。例如整数的数值，标识符的名称。

- 语义动作

为了更加实用，一个程序不仅仅要分析输入而且必须处理输入。在 bison 语法中，一个语法规则可以有一个包括多个 C 语句的动作。分析器每次识别一个规则的匹配，相应的动作就会被执行。

大多数时候，动作的目的是从部分结构的语义值计算整个结构的语义值。例如，有一个规则表明一个表达式可以由两个表达式相加而成。当分析器识别了一个加法和，每一个子表达式都有一个描述其如何构建的语义值。这个规则的动做就是为了新识别的大表达式建立一个类似的语义值。

- 分析器文件

当运行 bison 的时候，需要给 bison 一个语法文件作为其输入。bison 的输出是一个分析这个语法文件描述的语言的 C 源代码文件。这个文件叫做 bison 分析器。此处的 bison 工具和 bison 分析器是两个明显不同的程序：bison 工具是一个以 bison 分析器为输出的程序。这个 bison 分析器应是程序的一部分。

bison 分析器的工作是依照语法规则组合记号。例如，将标识符和操作符

构建成表达式。在组合的过程中它还要执行相应的语法规定的动作。

记号是来源于词法分析器，所以必须以某种形式提供词法分析器。bison 分析器每当需要一个新的记号的时候就会调用词法分析器，bison 分析器并不知道记号中有什么东西。

bison 分析器文件是实现了语法分析功能的函数，默认名称为 yyparse。这个函数并不能成为一个完成的 C 程序，必须提供额外的一些函数。其中之一是词法分析器，另一个是错误报告函数。另外，一个完整的 C 程序必须以名为 main 的函数开头；必须提供这个函数，并且安排它调用 yyparse，否则分析器永远都不会运行。

除了编写的动作中的记号类型名称和符号以外，所有 bison 分析器文件自己定义的符号都以'yy'或者'YY'开头。这些符号包括了接口函数，例如词法分析函数 yylex，错误报告函数 yyerror 和分析器函数 yyparse。

通过对 bison 的介绍，可以了解到构造语法分析程序的过程，而其中的主要工作就是对源语言文法产生式的设计，下面给出本文对 C 语言文法产生式的设计。

4.2.2 C 语言文法产生式的设计

根据 bison 的文法规则定义终结符、非终结符和运算符；通过 C 语言语句的结构，将终结符、非终结符和运算符组织编写成产生式。以下分别介绍终结符、非终结符的定义和运算符的性质是如何通过脚本文件体现出来的，然后再介绍 bison 如何运用这些定义编写产生式。其中终结符、非终结符的定义主要是根据 C 语言中词汇的分类和特点，主要分成五大类，同时产生式的规则也是基于这些定义写出的；运算符的性质是通过声明运算符时体现的。

(1) 终结符、非终结符的定义

bison 中通过在 bison 声明模块中声明终结符、非终结符和运算符，同时指明终结符和非终结符的数据类型，指明运算符的结合顺序和优先级。

终结符的定义形式如下：

%token <ival> identifier　string

其中%token 用来表明这里定义的是终结符；<ival>表明该终结符的数据类型；identifier 和 string 是终结符的名称。当有很多个终结符，可以写在一行，也可以另起一行。定义的终结符没有先后顺序。

非终结符的定义形式如下：

%type <tnp> input　expression

其中%type 表明这里定义的是非终结符；<tnp>表明该非终结符的数据类型，tnp 是已经定义的为 TreeNode 的数据指针类型名；input 和 expression 是非终结符的名称。当有很多个非终结符，可以写在一行，也可以另起一行。定义的非终结符没有先后顺序。

脚本中定义了很多终结符和非终结符，在文法规则中要使用的终结符必须在这里先声明，但是非终结符可以不用声明，声明只是为了指出该非终结符的数据类型。

(2) 运算符的定义

为了在产生式中实现正确的计算顺序，运算符的结合顺序和优先级必须提前定义。该定义写在 bison 声明模块中，当产生式匹配时会根据运算符的声明，进行归约。运算符的声明形式如下：

%left　<ival> COMMA

%right <ival> ASG PLUSEASG MINUSASG MULTASG DIVASG RESIASG LSHIFTASG RSHIFTASG BANDASG BORASG BNORASG

%left　<ival> COLON　QUESTIONM

%left　<ival> OR

%left　<ival> AND

%left　<ival> BOR

%left　<ival> BNOR

%left　<ival> BAND

%left　<ival> COMP　NEQUAL

%left　<ival> SMALL BIG NBIG NSMALL

%left <ival> RSHIFT LSHIFT

%left <ival> PLUSE MINUS

%left <ival> MULT DIV RESI

%right <ival> NOT BNOT DOUBPLUSE DOUBMINUS SIZEOF GETADDR NEG GETCONTANT

%left <ival> MEMBER PMEMBER LSBRAKET RSBRAKET LMBRAKET RMBRAKET

其中%left 和%right 指出该运算符的结合顺序，%left 表示该运算符为左结合，%right 表示该运算符是右结合；<ival>用来指出该运算符的数据类型，ival 是之前定义的 int 数据类型名；COMMA 等是运算符的名称，为了不产生冲突，将运算符都定义成英文形式。

运算符的优先顺序是通过声明的顺序来体现的，先声明的优先级低，即上面的运算符的优先级低于下面声明的。同一行的运算符优先级相同。

对于双义的运算符，如：&即表示取地址运算符又表示按位与运算符。此时需在运算符定义部分分别定义，名称不同 BAND 和 GETADDR，产生式中通过%precGETADDR 来表明&的优先级和结合性与 GETADDR 相同。同理'-'运算符，由于'-'既是减运算符又是取负运算符，脚本中定义了 NEG 和 MINUS 分别标明取负运算符和减运算符的优先级和结合性，产生式中通过%precNEG 来标明'-'的优先级和结合性与 NEG 相同。

(3) 产生式的设计

有了终结符、非终结符的定义和运算符的定义，下面通过这些定义来表达 C 语言的语句结构，完成产生式的编写。产生式中所用到的所有终结符和运算符必须是声明过的，其中终结符可以不用声明。文法后部须写出操作。总体把握 C 语言后，将语法分成以下几大类：

- Expression

主要包含基本的表达式，如：标识符、常量、字符串、单元运算、双目运算、逗号表达式、条件运算等。

- Lvalue

主要包含那些可以出现在运算符旁边的运算整体。

- Declarations

主要是对类型的定义，包含数据类型、函数类型及结构体的定义。

- Statements

主要是定义一些系统中固有的方法，如：if 语句、while 语句、for 语句、switch 语句、case 语句、default 语句、break 语句、continue 语句、return 语句、goto 语句等。

- External definitions

主要定义程序及函数的整体。

下面以 Lvalue 为例来说明实现的细节。Lvalue 的实现如下所示：

```
lvalue:identifier
|expression LMBRAKET expression RMBRAKET
|expression MEMBER identifier
|expression PMEMBER identifier
|MULT expression %prec GETCONTANT
;
```

上面是 Lvalue 去掉了操作后的代码。该实现表明 Lvalue 可以归约成终结符 identifier，或者归约成下面四个产生式。文法最后以分号结束，用冒号表明匹配关系，多个匹配项通过竖线连接起来。

bison 通过|来表示或的关系，可以将具有相同归约的产生式通过'|'连接起来，也可以分开来写。如：

```
expression:lvalue
|constant
;
```

等价于：

```
expression:lvalue ;
```

expression:constant;

并且产生式结尾用分号，表示一个产生式结束。

bison 文档内可以加注释，通过"/*　*/"包围。

以上给出了本文设计的 C 语言的文法产生式，通过语法分析，将源程序转换为抽象语法树，对于语法分析的输出结果要采用一定数据结构，下面给出抽象语法树的数据结构。

4.2.3 数据结构的设计与实现

本节描述了语法分析中定义的关键的数据结构 TreeNode，该结构是一个类，包含两个 TreeNode 型指针、一个共用体、一个结点类型变量和八个成员函数；该类用来存储结点特性值，连接结点确定结点关系，实现了抽象语法树的构造；同时在脚本文件中操作的实现大都通过调用该类的成员函数来实现，极大地方便了设计。

(1) 数据类型的定义

首先给出数据类型的定义，本文中将结点类型定义成可连接父子结点的数据类型，并且需设定存储单元记录结点的类型和特性值。操作中要频繁的建立结点间的关系：插入父结点、插入结节点，所以定义成员函数 insertchild 和 insertsibling。结点类型不同，特性值也不同，针对这种情况，运用函数重载的方法定义多个构造函数。

首先介绍数据类型的设计细节。bison 通过 union 来定义操作中和产生式中用到的数据类型。为了表示每个 token 的属性，定义了一个类：

```
class TreeNode
{
public: TreeNode *child,*sibling;
        NodeKind kind;
        union{
        TokenType op;
        double val;
        char *value;
        char *name;
```

```
    }attr;
    int i;
    TreeNode(NodeKind temp);
    TreeNode(NodeKind temp,TokenType op);
    TreeNode(NodeKind temp,double d);
    TreeNode(NodeKind temp,char* c);
    TreeNode(const TreeNode &temp);
    void insertchild(TreeNode * temp);
    void insertsibling(TreeNode * temp);
    ~TreeNode();
};
```

这个数据类型在存储结点信息起到了很重要的作用，以下将详细介绍各个成员变量和成员函数的作用。

Child 指针是用来保存该结点的第一个孩子结点，其他的孩子结点都连接在第一个孩子结点的兄弟结点上。

Sibling 指针是用来保存该结点的兄弟结点。

共用体 attr 是用来保存不同结点的值，如：运算符的类型就保存在 op 中；数字就保存在 val 中；字符串就保存在 value 中；标识符就保存在 name 中。通过共用体将这些类型都保存起来，节省系统开销。

类中有构造函数重载，其中 TreeNode(NodeKind temp)是针对于那些没有 attr 值得节点，这些结点一般都是产生式中的非终结符，用于指明下面结点之间的关系；TreeNode(NodeKind temp,TokenType op)用于构造那些运算符结点；TreeNode(NodeKind temp,double d)用于构造那些数字结点；TreeNode(NodeKind temp,char* c)用于构造字符串和标识符结点，该函数通过 NodeKind 来判断是不是字符串结点，是则将特性值保存在 value 中，不是则当作标识符来处理，将特性值保存在 name 中；TreeNode(const TreeNode &temp)为拷贝构造函数，将 temp 的所有信息都传递给当前结点。

void insertchild(TreeNode * temp)函数用来将 temp 节点插入到当前结点，确定俩结点的关系为父子节点。函数设计如下：

```
void TreeNode::insertchild(TreeNode * temp)
```

```
{
    TreeNode *t;
    t=this;
    if (t->child==NULL){
        t->child=temp;
    }
    else{
        t=t->child;
        while(t->sibling!=NULL)
            t=t->sibling;
            t->sibling=temp;
        }
    }
}
```

函数首先判断该节点是否已经有孩子结点了，如果没有将 temp 树结点赋给 child；否则，说明 temp 结点是该结点孩子的兄弟，将 temp 赋到兄弟链的最后。

void insertsibling(TreeNode * temp)是将 temp 结点插到当前结点兄弟链的最后，并确定二者的关系是兄弟。函数细节如下：

```
void TreeNode::insertsibling(TreeNode * temp)
{
    TreeNode *t;
    t=this;
      while(t->sibling!=NULL)
          t=t->sibling;
          t->sibling=temp;
      }
}
```

实现中还定义了两个枚举类型，分别用来标识节点的类型和运算符的类型，它们分别是 NodeKind 和 TokenType。

(2) 操作的编写

有了 TreeNode 类，就可以将需要保存的结点都通过相应的构造函数生成树结点，再通过结点之间的父子或兄弟关系连接起来。如：

```
|value:identifier
{
    $$=&TreeNode(IdK,(char*)$1);
}
|expression LMBRAKET expression RMBRAKET
{
    $$=&TreeNode(ArrStmtK);
    $$->insertchild($1);
    $$->insertchild($3);
}
|expression MEMBER identifier
{
    $$=&TreeNode(OpK,MEMBER);
    $$->insertchild($1);
    $$->insertchild($3);
}
 |expression PMEMBER identifier
{
    $$=&TreeNode(OpK,PMEMBER);
    $$->insertchild($1);
    $$->insertchild($3);
}
    |MULT expression %prec GETCONTANT
    {
     $$=&TreeNode(OpK,GETCONTANT);
    $$->insertchild($2);
}
```

同理 insertsibling；

采用以上的数据结构，便可以构造出抽象语法树。下面给出词法分析程序的算法。

4.2.4 语法分析程序的算法及其说明

(1) 语法分析程序的主要工作原理[19]

语法分析程序是由放在栈中的有限状态机构成的，它同时能够读取并记住下一个输入的单词(token)，有时也叫这样的单词为向前看字符(lookahead token)。有限状态机的状态是用一个整数表示的，与词法、语法分析程序中的单词是相似的。在初始时，机器处在 0 状态，此时仅仅栈中包含了状态 0，而没有向前看字符被读入。

有限自动机在语法分析程序中可执行相应的动作，以决定下一步的动作。语法分析可执行的动作(action)共有 4 个，它们分别是：移进(shift)、归约(reduce)、接受(accept)和错误(error)。语法分析程序的运行如下：

- 根据当前状态，语法分析程序决定是否需要一个向前看字符以决定将要执行什么动作。如果需要一个单词但还没有读入时，语法分析程序将调用 yylex 获得下一个单词。
- 用当前状态和向前看字符(如果需要的话)，语法分析程序决定它的下一个动作(action)，并且执行这个动作。这样将导致状态被压入或被弹出栈，以及向前看字符是被处理了还是放在一边待用的不同变化。

以下分别说明 4 种语法动作：

移进动作(shift action)是语法分析程序采用的最平常的动作。根据当前栈的状态和向前看字符，语法分析程序查找动作表(action 表)，以决定将采取什么动作。对于移进动作，不论何时采用移进动作，都有一个向前看字符，不然移进动作将无法执行对单词的清除处理。而对应的状态却被压入了状态栈，从而造成错误。

归约(reduce action)是用语法规则的左边代替其右边的过程。当语法分析程序已经看到某语法规则的右边，并且确定右边是语法规则的一个实例时，归约动作将被采用。通常语法分析程序在采用归约动作时是不需要查看向前

看字符的。

归约动作能防止栈无限增大，因为大多数情况下归约动作是将状态弹出栈，减少占用的栈空间。归约动作取决于语法规则左边的符号(非终结符)和右边的符号(终结符和非终结符)数。归约动作首先根据语法规则右边的符号数决定将被弹出栈的状态数，然后将状态从栈中弹出，使栈中的其它的状态被显露出来。事实上，被弹出的状态就是在识别此规则右边单词时压入栈的状态，规则一旦被识别，这些状态将不再有用。将这些状态弹出之后，在状态栈中处在栈顶的状态是语法分析程序开始处理此规则之前的状态。使用弹出之后的状态和规则左边的符号，将获得一个新状态。将此新状态压入栈，语法分析继续。

处理左边符号和普通的单词移进是有明显的不同的。把对左边符号的移进动作叫做状态转换动作(goto action)，尤其不同的是，当执行移进时，向前看字符是被清除的，但在执行状态转换动作时，向前看字符是不受影响的。

如果规则的右边是空的，没有状态被弹出栈，也就是栈中被显露出来的状态就是栈中的当前状态。

归约在处理用户提供的动作和值时也是重要的。当一条规则被识别并被归约时，在栈被调整以前，由规则提供的代码(code)将被执行。语法分析程序使用两个栈：一个是拥有状态的栈，即上文提到的栈；另一个是与之并行运行的值栈(value stack)，这个栈拥有从词法分析程序和动作返回的值。当移进时，外部变量 yylavl 被拷贝到值栈。从用户码(user code)返回后，归约被执行。当状态转换动作被执行后，外部变量 yylavl 被拷贝到值栈。在 bison 中，用伪变量$l, $2 等来代表值栈的值，而$$代表左边非终结符的值。

另外两个语法动作相对来说比较简单。

接受动作(accept action)表明整个输入已经看到并且匹配语法规则，这个动作只有当向前看单词是 endmarker(文件结束符，本文中是以 EOF 为文件结束符的)才被采用，表明语法分析程序已经成功地完成语法分析工作。

报错动作(error action)说明根据规则在某处语法分析程序不能再继续语法分析了。当输入的单词已经被看到，同时还有向前看字符，但后面不能跟随可导致合法输入的任何东西，此时语法分析程序报告一个错误，并且试图恢复状况(situation)，试图接着进行语法分析。

(2) 语法分析程序

本文的语法分析程序 yyparse 的主要流程说明，如图 4.3

在进行语法分析时，语法分析器主要使用两个栈：一个是状态栈，一个是语义值栈。两套指针:short *yyss, yyssa； YYSTYPE *yyvs， yyvsa。

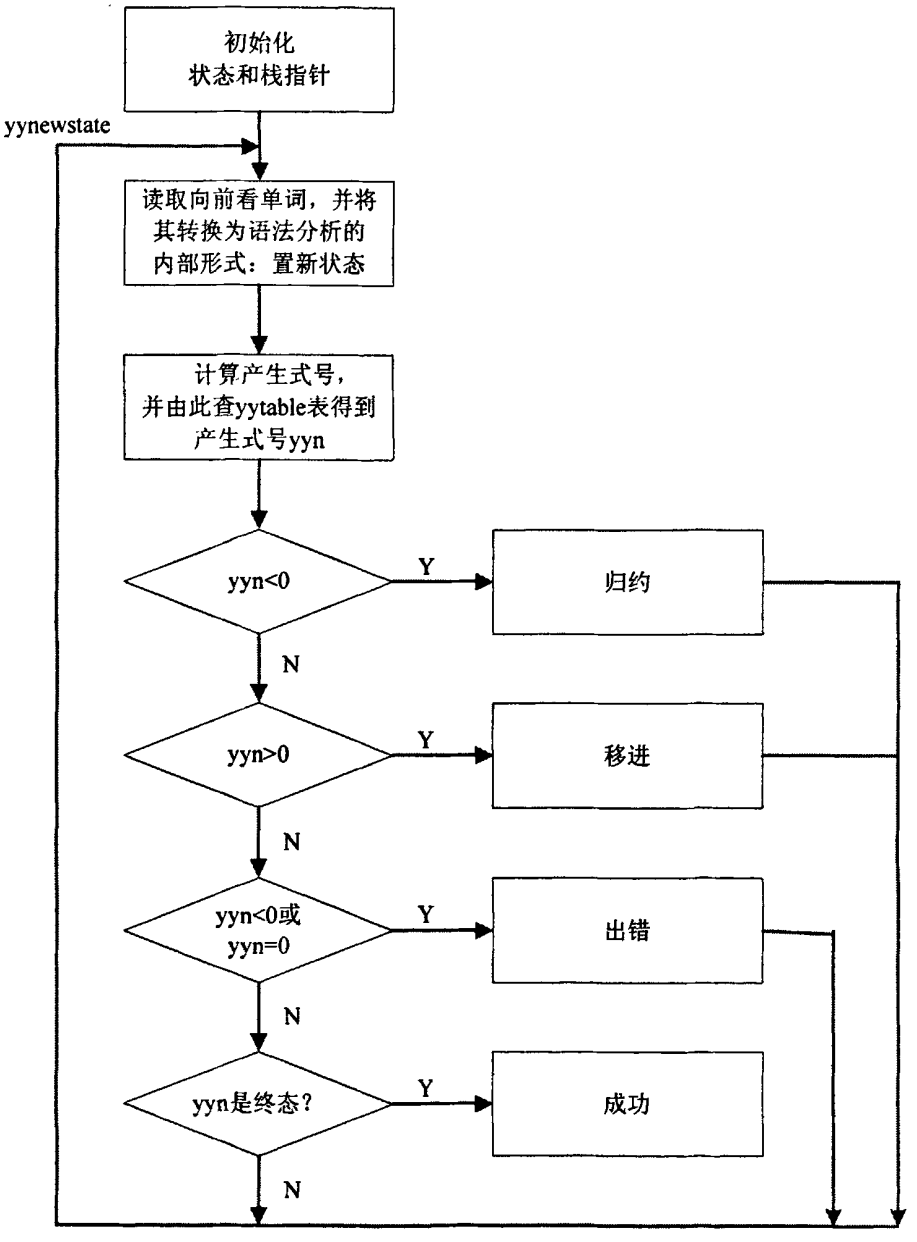


图 4.3 语法分析程序 yyparse 的主要流程

- 初始化状态栈的状态和值栈的单词值。yystate=0； yyerrstatus=0；

yynerrs=0；yychar-YYEMPTY；初始化栈指针：yyssp=yyss-1；yyvsp=yyvs；对于其中主要变量的说明：yychar 是 lookahead symbol；yycharl 是 yychar 的语法分析的内部形式；yylval 是 lookahead symbol 的语义值：yyval 是用于从执行程序中返回语义值；yystate 是从执行程序中返回的状态；

- yynewstate：将返回的状态 yystate 压入状态栈，成为新状态。*++yyssp=yystate ；取出产生式号 yyn=yypact[yystate]；如果产生式号 yyn 为 YYFLAG，则转到 4 (yydefault)。如果 yychar 为 YYEMPTY，则调用词法分析函数得到一个单词(token),即 yychar-YYLEX。索引表是用单词(token)的内部形式索引的，故将单词转换为内部形式。当 yychar<=0 时：yycharl=0； yychar=YYEOF；当 yychar>0 时：yycharl= YYTRANSLATE (yychar)；计算 yycharl 的产生式号：yyn=yytable[yyn]。 yyn 是这个 token 类型在此状态下要做的动作：如果 yyn 是正数，则移进，而 yyn 是新状态。如果 yychar 不是 YYEOF，则放弃被移进 token，置 yychar= YYEMPTY；*++yyvsp=yylval； yystate=yyn；转到 2 (yynewstate)；如果新状态是终态，则返回成功而不必移进。
- yydefault：执行当前状态的默认产生式。yyn=yydefact [yystate] 。
- yyreduce：如果 yyn 是负数，则规约，而_yyn 是语法规则号，yyn=_yyn;yylen=yyr2[yyn]：如果 yylen>0，则 yyval= yyvsp[1-yylen]；由此实现执行的默认值。
- 如果 yyn 是 0 或者大多数的负数时，则出错。
- 根据 yyn 的值确定产生式，并执行相应的动作。
- 规约后，yyvsp-=yylen；yyssp-=yylen；将单词(token)的语义值赋给语义栈，*++yyvsp=yyval。
- 根据弹出后的状态和被规约的产生式号决定转换到什么状态。yyn=yyrl[yyn]； yystate=yypgoto[yyn-YYNTBASE] + *yyssp。
- yyerrhandle：出错处理。yystate=yyn；转到 2 (yynewstate)。或返回 1。
- yyacceptlab：接受状态，释放栈空间:free(yyss)； free(yyvs)；并返回 0。

通过以上的过程便可以完成对源程序的语法分析，源程序经过语法分析后得到抽象语法树，由于后续的编译工作是建立在抽象语法树之上的，所以

有必要对抽象语法树做一简单介绍。

4.3 抽象语法树AST

经过语法分析后会得到一棵语法树，也称抽象语法树(Abstract Syntax Tree，AST)[20]。抽象语法树是程序的一种中间表示形式。能够包含整个编译单元的完整表示，比较直观地表示出源程序的语法结构，并具有较高的存储效率。对遍历和操作十分方便。因此，AST 是编译过程中最合适的数据结构。

在编译过程中，首先要把源程序转化为抽象语法树，后续的编译工作要建立在抽象语法树之上，所以这里说明一下 C 语言中主要语句对应的抽象语法树。

(1) 赋值语句

对于赋值语句，抽象语法树中每个叶结点代表运算分量，而其他的各个内结点则代表运算符。例如，对于赋值语句："x=a+2"的抽象语法树的如图 4.4 所示。

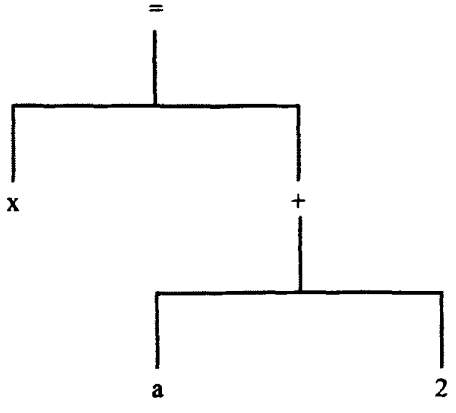


图 4.4 赋值语句"x=a+2"的抽象语法树

其中 x,a 与 2 等是标识符或无正负号整数，这样的终结符号称为运算分量终结符号。"="与 "+"是运算符，这样的终结符号称运算符终结符号。

(2) 循环语句

对于循环语句，对应的抽象语法树分成两个子树，第一个子树对应循环条件；第二个子树对应循环体，如果循环体包含多条语句，则每条语句对应循环体的一个子树。例如，对于语句：

```
while (q>0) {
a=a+1;
q=q-1;
}
```

对应的抽象语法树如图 4.5 所示。

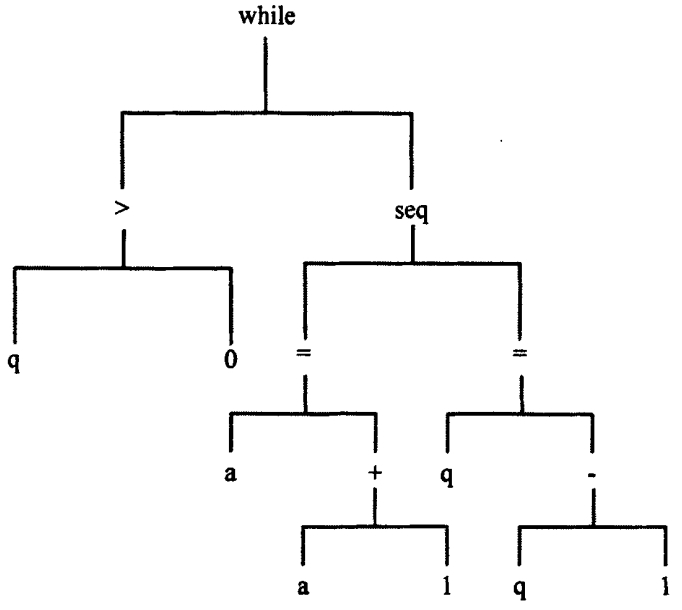


图 4.5 while 语句对应的抽象语法树

其中，循环条件(q>0)对应的抽象语法树作为循环语句抽象语法树的第一子树，循环体对应循环语句的第二子树，由于循环体是复合语句，所以每条语句对应对应循环体抽象语法树的一个子树。

(3) if 语句

对于 if 语句，同循环语句类似，if 语句的抽象语法树第一子树对应条件表达式，第二子树对应 if 语句体，该子树又有两个子树，第一子树为条件表达式为真时执行的语句体，第二子树为条件表达式为假时执行的语句体。例如，对于语句：

```
if(a>0)
  b=b+e;
else
  c=g+f;
```

对应的抽象语法树如图 4.6 所示。

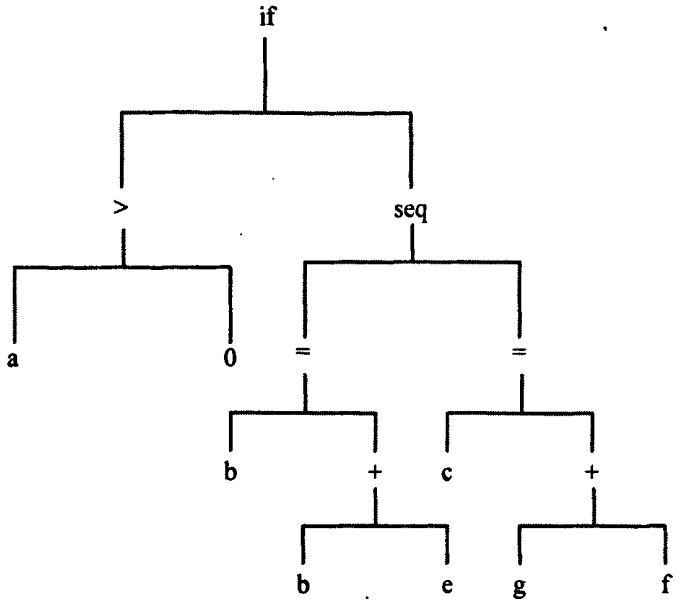


图 4.6 if 语句对应的抽象语法树

其中，条件表达式(a>0)对应 if 语句抽象语法树的第一子树，if 语句抽象语法树的第二子树对应 if 语句的语句体，因为这个 if 语句包含 else 语句，所以语句体对应的抽象语法树又包含两个子树，第一子树对应语句“b=b+e”，第二子树对应语句“c=g+f”。

以上给出了 C 语言中主要语句对应的抽象语法树，源程序经过编译器前端的处理，就得到相应的抽象语法树，后续的编译工作将在抽象语法树之上进行。

4.4 本章小结

本章介绍了编译方法的词法分析部分和语法分析部分，介绍了词法分析器的生成器 flex++，本文设计的 C 语言中记号的正则表达式，然后阐述了词法分析程序的算法以及采用的相应的数据结构；语法分析部分介绍了语法分析器的生成器 bison，本文设计的相应的产生式，以及语法分析程序的算法及采用的数据结构，最后介绍了经过语法分析后得到的抽象语法树。

第5章 基于AST生成目标代码

本文的编译方法的后端是在前端生成的抽象语法树的基础上完成的，为了将 C 语言描述的源程序转换生成 VHDL 代码，本文设计了一种中间形式控制数据流图(CDFG)，首先将由前端生成的抽象语法树转换为中间形式 CDFG，然后，对 CDFG 进行转换生成 VHDL 代码，从而完成编译的全部工作。下面分别介绍了编译器的内部表示模型 CDFG，由抽象语法树生成 CDFG 的过程，以及如何将 CDFG 进行转化生成 VHDL 代码。

5.1 编译器内部表示模型CDFG

本文的编译器的目标代码是 VHDL 代码，所以要将源语言转换为一种内部表示，这种内部表示要便于最后生成目标代码，考虑 VHDL 的特点，本文设计了一种便于生成目标代码的内部形式 CDFG，首先介绍一下本文所采用的内部表示模型[37]。编译过程的第一步通常是将语言描述翻译成一种中间表示的格式，大多数方法是用包含数据流和控制流的各种语法分析树或分析图来表示。本章对内部表示模型进行了分析研究，针对本文的 C 子集，给出了中间代码模型，讨论了 C 子集中各种语句的 CDFG 表示，并给出了从 C 源代码描述到 CDFG 的生成方法。

5.1.1 内部表示模型

本文的编译方法的第一步是将用 C 描述的源代码转换生成抽象语法树，然后将抽象语法树编译成中间表示形式，而后其他编译任务在此基础上进行。从目前编译系统采用的内部表示模型来看，有的编译系统采用的有向图表示形式，生成数据流图 DFG[29]、控制流图 CFG 或控制数据流图 CDFG，C/DFG[25,26]。比较而言有向图方式表示方便，所以本文的编译系统均采用这种内部表示形式。

DFG 由一个结点集合和一个有向边的集合构成，一个结点表示源描述中的一个操作。如果两个结点之间有数据依赖关系(即操作 Ni 的结果是操作 Nj

的输入)，就用一条有向边将两个结点 Ni 和 Nj 连接起来，该边表示操作 Nj 不能在操作 Ni 之前执行。DFG 能表示出数据操作的依赖性及操作顺序。一个典型的 DFG 就是将算法描述中的操作映射到结点上，将值映射到结点之间的边上。

CFG 能够表示出操作之间的控制依赖关系，所以它一般用来表示条件结构、循环结构等控制结构，可从行为描述中直接得到。如果在 DFG 之上加上控制流，实际上是加入了控制结点和控制边，就可将 DFG 扩充到 CDFG，显然，CDFG 可表示出条件转移、循环等控制结构[38,39]。

通过对大量的编译系统进行分析之后，本文发现许多早期的编译系统采用分开的数据流图和控制流图表示，这实际上包含着高度的冗余,尽管容易实现，但更难处理，而且这种表示限制了算法的搜索空间，增加了产生次优化解的可能性。因此，本文采用 CDFG 表示。

5.1.2 CDFG 模型

针对本文选取的 C 子集，本章提出的 CDFG 是一个有向图表示 G=(V，E)，其中 V 是结点的集合，E 是边的集合。结点表示操作，边表示从一个结点到另一个结点的值的传输或两个结点之间执行顺序的控制。

在 CDFG 中，需要四种类型的结点和两种类型的边，结点包括操作结点、输入输出结点、控制结点与复合结点。图 5.1 给出了各种类型结点的例子。

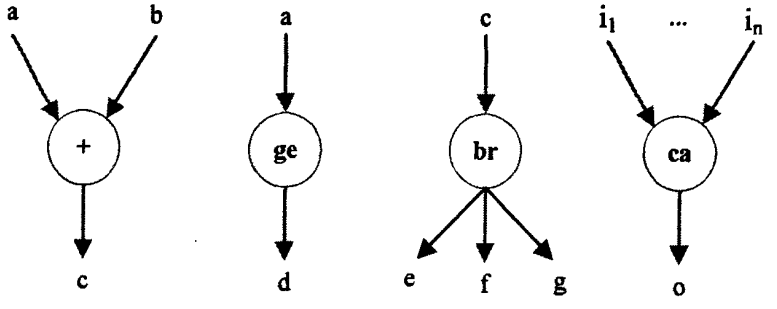
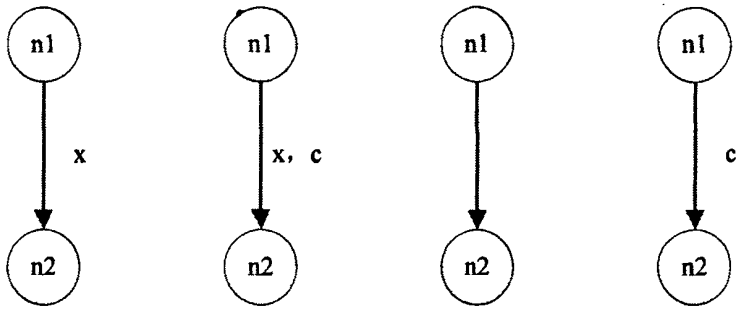


图 5.1 各种类型的结点

操作结点表示的操作包括算术操作(如+，一，*，/，++等)、布尔操作(如 and，or，nor，xor 等)、关系操作(如<，>，<=等)、逻辑操作等等；get 结点为输入结点，接收到的值是输入数据，put 结点完成输出操作；控制结点用来表示条件结构和循环结构的控制流，它包括 loop 结点、endloop 结点、branch

结点和 merge 结点；复合结点表示子程序的调用，它的执行会引起一个子图的执行，它仅包含一个 call 结点。本文中用 ge、pu、br、me、lo、el、ca 和 n 来分别表示 get、put、branch、merge、loop、endloop、call 和 node。

在 CDFG 中，有两种类型的边：数据依赖边和控制依赖边。前者包括数据边和条件数据边；后者包括控制边和条件控制边，图 5.2 给出了各种类型的边。



(a)数据　(b)条件数据　(c)控制　(d)条件控制

图 5.2 各种类型的边

在图 5.2(a)中，n2 使用 n1 的操作结果 x；在图 5.2 (b)中，n2 是 n1 的条件后继节点，仅当条件 c 为真时，n2 才等 n1 的操作结果 x；图 5.2 (c)中，仅当 n1 执行完成之后，n2 才可执行；在图 5.2 (d)中，仅当条件 c 为真时，n2 才等待 n1 的执行。显然，当一个结点要用另一个结点的结果时，后者必须提供一个对前者的控制，所以，一个数据依赖包含着一个隐含的控制依赖，一个控制依赖由一个数据依赖进行支配。

在给出了 CDFG 的定义后，本文将给出针对所选取的 C 子集中一些主要语句的 CDFG 表示。

(1) IF 语句的表示

在 CDFG 中，可采用一个 br 结点和一个 me 结点来表示一个 IF 语句，即条件结构，其中 br 结点是条件结构的入口，me 结点是条件结构的出口，图 5.3(a) 给出了条件结构的通用表示形式。为了说明条件结构的 CDFG 表示，图 5.3(b)给出了一段包含条件结构的 C 描述，图 5.3(c)给出了对应的抽象语法树。图 5.3(d)给出了相应的 CDFG 表示。在图 5.3(d)中，结点(0)与 br 之间的。

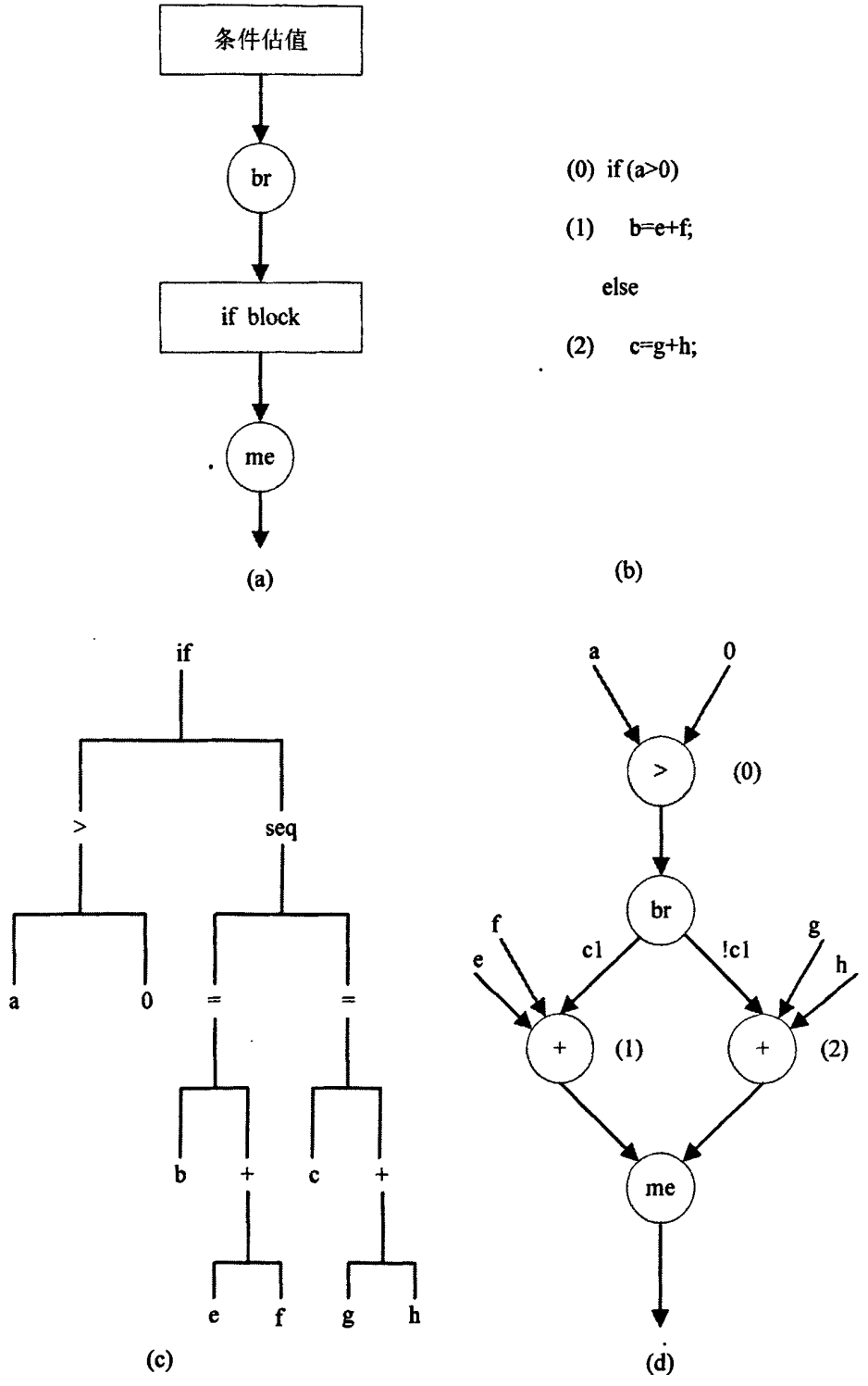


图 5.3 条件结构的表示实例

边为数据边(c1 为数值)；br 结点与结点(1)、结点(2)之间的边为条件控制边(c1，!c1 为条件)；与 me 结点相连的边均为条件控制边

(2) 循环结构的表示

在表示循环结构时，可通过对循环结构中的每个结点引入一个标记，用它来记录循环执行的次数，而将循环条件的判定放在循环结构的入口处，就可完全表示出循环结构的语义。在 CDFG 中，给出了循环结构的一种内部表示形式，用四个控制结点(lo 结点，el 结点，br 结点和 me 结点)来表示一个循坏控制结构，其中 lo 结点是循环结构的入口，me 结点是循环结构的出口，进入入口后，计算循环条件的值，然后用一个 br 结点来选择是否进入循环体。图 5.4 给出了循环结构的通用的表示图。

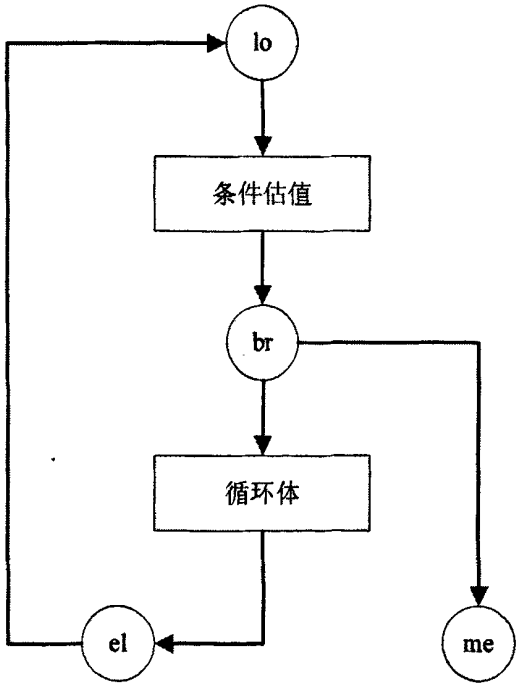
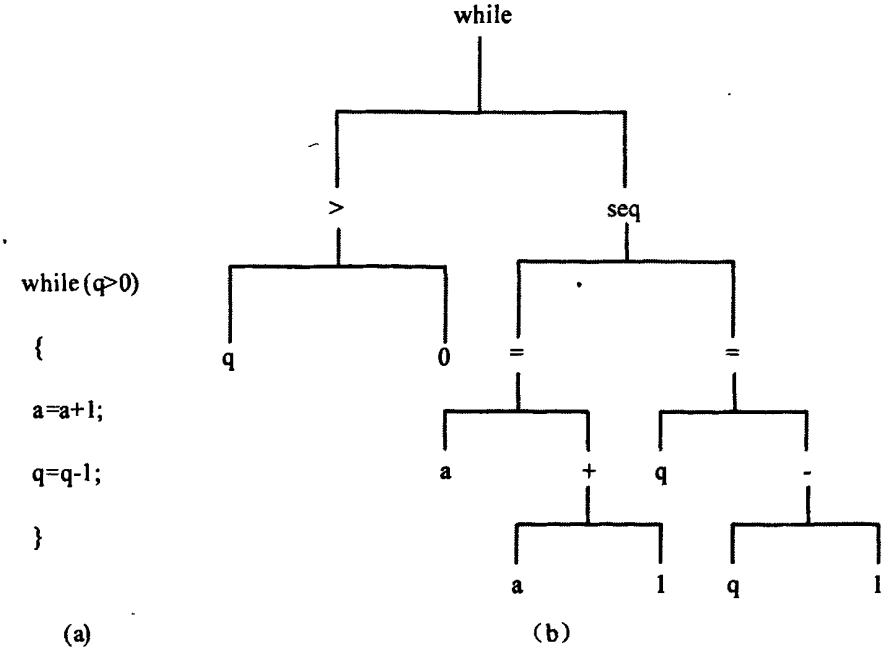


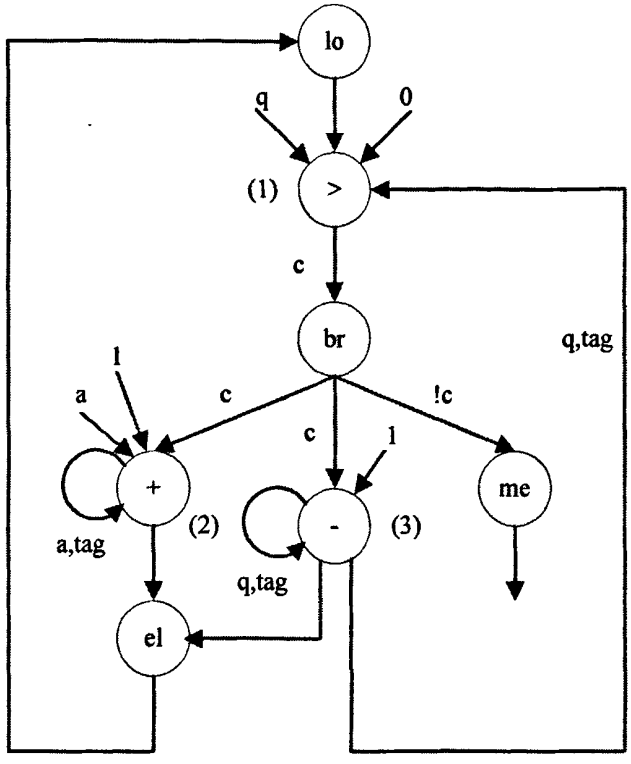
图 5.4 循环结构的通用表示

在一个循环结构中，数据依赖关系与控制依赖关系非常复杂，循环体内的一个结点可被执行多次，操作数据可能来自循环体外，也可能来自循环的上一次迭代。在通常情况下，第一次迭代时，数据总是来自循环体外，在后续的迭代中，一些数据来自上一次的迭代结果，为了表示出这种数据依赖关系，对每个结点引入了一个标记(tag)，它记录了结点被执行的次数。显然，在第一次迭代时，tag 的值为 0，在后续的每一次迭代中，tag 的值将逐步增 1。

可将标记看作是边的权，也可以认为它是一个条件，当 tag 的值为 0 时，它所表示的条件为 FALSE，当 tag 的值大于 0 时，它所表示的条件为 TRUE。下面给出一个例子来说明循环结构的 CDFG 表示。

图 5.5(a)给出了一个循环结构的 C 描述，图 5.5(b)给出了相应的抽象语法树表示，图 5.5(c)给出了对应的 CDFG 表示。可以看出，为了表示出该描述，需要四个控制结点与三个操作结点。当 el 结点执行后，表示一次迭代的结束，控制将传到 lo 结点，tag 的值将增 1。el 结点与 fo 结点之间的边为控制边。对结点(l)来说，在第一次迭代时，q 的值来自循环体外部，但在后续的迭代中，q 的值来自循环体内(来自结点(3) 的操作结果)。连接结点(1)与结点(3)的边是一个条件数据边，其中 q 是数据，tag 为条件，当 tag 的值为 0 时，q 的值来自循环体外，当 tag 的值大于 0 时，q 的值来自节点(3)的操作结果。对结点(2)，连接它与 br 结点之间的边为条件控制边，当条件 c 为真时，将执行结点(2)，在第一次迭代中，a 的值来自循环体外，但在后续迭代中，a 的值来自它本身，连接结点(2)与它自己的边为条件数据边，连接结点(2)与 el 结点的边为控制边。对结点(3)，第一次迭代时，q 的值来自循环体外部，但在后续的迭代中，q 的值来自它本身，连接结点(3)与它自己的边为条件数据





(c)

图 5.5 一个循环结构表示的例子

边，连接结点(3)与 el 结点的边为控制边。对于 br 结点，连接它与结点(l)的边为数据边(c 是布尔型数据)，连接它与结点(2)、结点(3)及 me 结点的边为条件控制边(c 是条件)，当 c 为真时，将执行结点(2)与结点(3)，否则，控制将传到 me 结点，这表示循环结构的结束。从上述可看出，为了正确执行循环语句，必须先确定数据依赖关系和控制依赖关系。

从图 5.5 的例子中可以看出，在循环结构的 CDFG 表示中，仅用了一个转移结点和一个合并结点，但对每个结点引入了一个标记，这样形成的流图并不复杂，容易理解。

(3) 函数调用语句的表示

在 CDFG 中，我们给出了函数调用语句的内部表示形式，用一个复合结点(即 ca 结点)来表示函数调用语句，这个结点的输入是函数调用的输入参数，输出就是执行过程后的结果。实际上，ca 结点的执行会引起一个子图的执行，这个子图与 CDFG 是独立的。图 5.6 中给出了一个说明性的例子，其中图 5.6

(a)是 C 描述，图 5.6 (b)是图 5.6 (a)中 C 描述中的函数 bb 的行为描述，图 5.6(c)是图 5.6 (a)C 描述对应的抽象语法树，图 5.6(d)是图 5.6 (b)C 描述对应的抽象语法树，图 5.6 (e)是图 5.6 (c)对应的 CDFG 表示，图 5.6 (f)是图 5.6 (d)对应的 CDFG 表示。从图 5.6 (e)中可以看出，在 CDFG 中，函数调用语句是由一个复合结点来表示的，图 5.6 (f)给出了该函数的 CDFG 表示。当控制到达复合结点时，子图将被执行，当子图执行完成后控制将返回到复合结点，再传到其后续结点，这样，就彻底表示出了函数调用语句的语义。

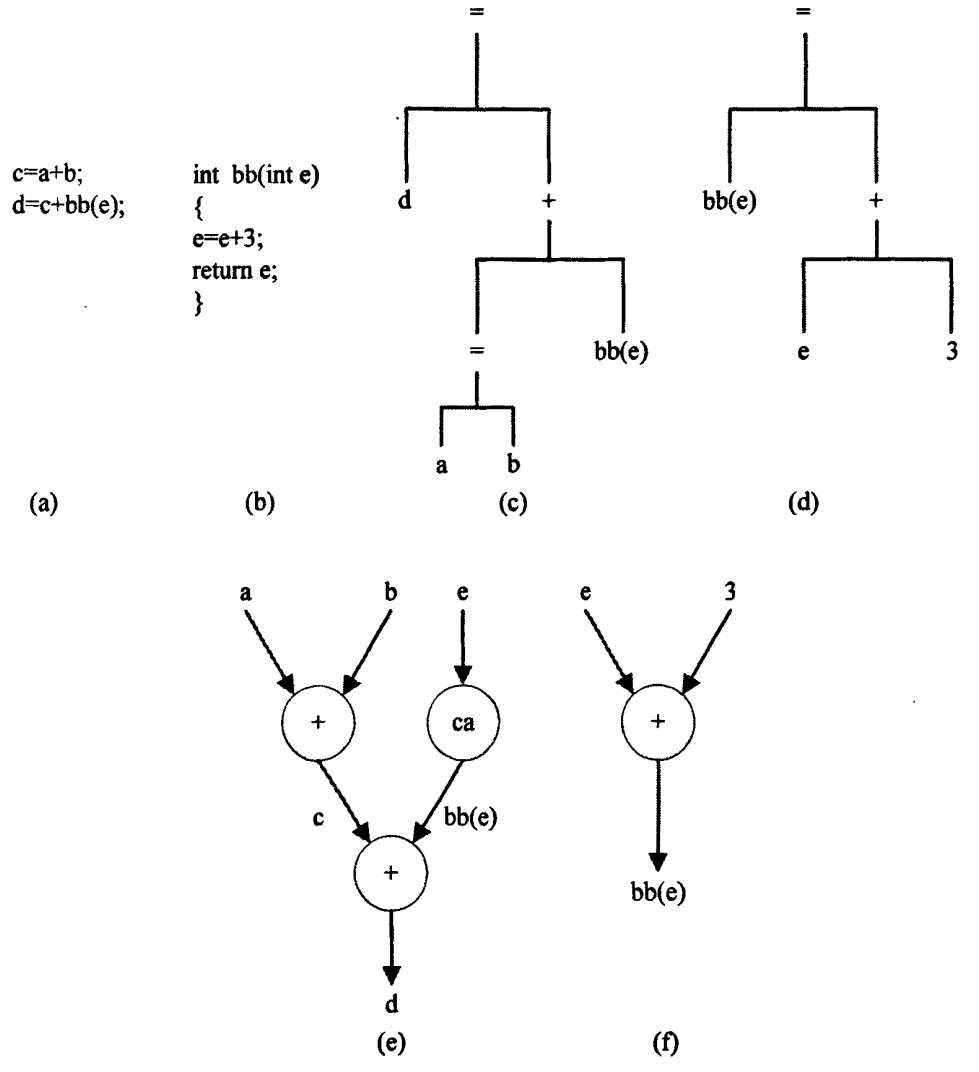


图 5.6 函数调用语句的例子

至此，给出了 C 语言中一些主要语句的 AST 及 CDFG 表示。本文的编

译系统的的后端的任务就是将源代码对应的 AST 转换成相应的 CDFG，最后将 CDFG 转换为目标代码。下面将给出由 AST 到 CDFG 再到目标代码的转换过程。

5.2 生成CDFG

本节针对上文提出的 CDFG 模型，建立中间代码的数据结构，并介绍如何由 AST 生成 CDFG 图。在将一个以抽象语法树给出的行为描述转化为 CDFG 时，可通过以下两步来进行：

(1) 对抽象语法树进行遍历时，产生相应的结点，同时，产生控制边。

(2) 在控制依赖的基础上，通过对抽象语法树结点的分析，产生数据依赖边。

本文建立的 CDFG 图由多个子图构成，每个子图表示一个函数。CDFG 生成后，后续的编译任务就可在 CDFG 之上进行了。首先，给出了 CDFG 的数据结构，然后给出了生成 CDFG 的整体算法。

5.2.1 建立 CDFG 的数据结构

```
Class Midcode
Public:
Midcode();
~Midcode();
bool delete_flag;
castring   *parameter;//函数参数记录
int   paramcount;//参数计数器
int   addr;//产生结点序号
int   node_tag;//结点类型标志
cstring argl，arg2;//第一，第二操作数
int   argltype，arg2type;//操作数类型：0—常量；1—变量；2—临时变量
Midcode*arglPtr;
Midcode*arg2Ptr;
```

```
Cstring oper;//操作符
Midcode*the;//真链
Midcode*feh;//假链
Midcode*next_clause;//下一语句头结点
CAedge*edge;//记录数据边
Merge *merge;//记录复合语句(if、loop)尾结点
void Insertedge(&v1,&v2,&G.arcnum,&incinfo);//插入一条边
int  Insertmerge(floaty，intx);//插入一个尾结点
int  mergecount;//尾结点计数器
};
```

在中间代码的数据结构中结点类型标志的定义如下：

(1) 操作运算的表示

node_tag=l；如果 argltype=0/l，argl 记录第一操作数；如果 argltype=2，arg1Ptr 记录第一操作数中间格式指针；同理对 arg2 也是如此。oper 记录操作符。

(2) 赋值语句的表示

node_tag=2；表示同 1。

(3) if 语句的表示

node_tag=3；tch 连接真分支语句首结点，fch 连接假分支首结点。node_tag=4；表示 ELSEIF 语句，表示方法同 if 语句。

(4) while 语句的表示

node_tag=5；tch 记录真分支语句。

(5) 函数调用语句的表示

node_tag=6;argl--函数名；arg2ptr--函数语句头结点；Parameter 记录函数参数。

5.2.2 基于抽象语法树生成 CDFG 总体算法

上面介绍了中间代码的数据结构，下面将介绍由抽象语法树到 CDFG 的生成算法，将抽象语法树转换生成 CDFG 实质上就是对抽象语法树进行遍历的过程，本文中对抽象语法树进行遍历时采用先序遍历法，具体的算法如下：

(1) 访问 a 抽象语法树的根结点，生成其在 CDFG 中对应的结点；

(2) 先序遍历根结点第一棵子树，并为其生成对应的 CDFG 子图；

(3) 先序遍历除去第一棵子树之后剩余的子树森林，为每棵子树生成对应的 CDFG 子图。

由第四章可知每棵根结点的子树对应一个语句体，所以将每棵子树进行转换生成 CDFG 的过程即为将语句体进行转换生成对应的 CDFG 的过程。具体的转换算法如下：

```
Conversion(treenode *t)
{
    While(还有子树未被转换)
    {
       if(该子树为简单语句体对应的抽象语法树)
          {
          if(该树不为空)
             {
             switch(flag)
             {
             case 赋值语句：生成赋值语句抽象语法树对应 CDFG;
             case if 语句：生成 if 语句抽象语法树对应 CDFG;
             case 循环语句：生成循环语句抽象语法树对应 CDFG;
              }
             }
          }
       else
             {
             为该子树根结点建立相应的 CDFG;
             对该子树的各棵子树递归调用 convertion 算法;
             }
     }
}
```

将赋值语句抽象语法树转换生成 CDFG 算法：

(1) 如果赋值语句右部为表达式，则为相应的表达式的抽象语法树建立相应的 CDFG 子图，否则直接建立输入结点；

(2) 对于各表达式抽象语法树，首先建立操作结点，然后生成指向操作结点的数据边；

将 if 语句抽象语法树转换生成 CDFG 算法：

(1) 遍历 if 抽象语法树条件表达式子树，生成操作结点及相应的数据边；

(2) 生成 br 结点，并建立与操作节点的控制边；

(3) 为条件语句体抽象语法树各个子树分别建立操作结点和相应的数据边和控制边；

(4) 建立 me 结点，并生成合并结点与各操作结点之间的条件控制边；

将循环语句抽象语法树转换生成 CDFG 算法：

(1) 建立循环入口 lo 结点；

(2) 访问抽象语法树第一子树(条件表达式子树)，建立循环表达式操作结点以及指向操作结点的数据边；

(3) 为循环体抽象语法树各个子树分别建立操作结点和相应的数据边和控制边；

(4) 对于改变循环条件变量的操作结点，生成其与循环条件操作结点的条件数据边；

(5) 建立 el 结点，并生成各循环语句分支指向 el 结点的控制边，同时生成 el 结点与 lo 结点的控制边；

(6) 建立 me 结点，生成 br 结点与 me 结点的条件控制边；

通过以上算法可以对抽象语法树进行转换，生成相应的 CDFG，这样就将 C 描述的源程序转换生成了对应的 CDFG。下面的工作就是编译器的最后一部分工作，即将 CDFG 进行转换生成 VHDL 描述的目标代码。

5.3 基于CDFG生成VHDL代码

基于 CDFG 生成 VHDL 代码的工作是通过一个现有的软件 cdfgtool-1.0 来实现的(http://poppy.snu.ac.kr/CDFG/)。该软件是开源的。其输入是

CDFG，输出是对应的 VHDL 代码。cdfgtool 的主要工作是对 CDFG 进行遍历，并对每一个结点和边执行相应的操作从而生成 VHDL 描述的目标代码。CDFG 是一个分层的结构，所以这一操作是对 CDFG 进行并对其子图进行递归操作。下面的算法即为 cdfgtool 的主要算法——遍历图中的所有子图并输出相应的内容。

```
void trav(Subgraph *subg) {
  NodePtrList &nlist = *subg->getNodes();
  EdgePtrList &elist = *subg->getEdges();
   for (Pix pi=elist.first(); pi; elist.next(pi)) {
    Edge *e = elist(pi);
    e->dump();
}
for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
  Node *n = nlist(pi);
  n->dump(0);
}
for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
  Node *n = nlist(pi);
  switch (n->getType()) {
    case N_COND:
    {
      test(n->getCond());
      int num_subg = n->getNumSubg();
      for (int i=0; i < num_subg; i++) {
        test(n->getSubg(i));
}
break;
}
    case N_MOD:
      test(n->getSubg());
```

```
            break;
        case N_ITER:
            test(n->getCond());
            test(n->getSubg());
            break;
    }
  if (n->getCond())   test(n->getCond());
  int num_subg = n->getNumSubg();
  for (int i= -1; i < num_subg; i++) {
    if (n->getSubg(i))   test(n->getSubg(i));
  }
  }
  }
  void genvh(){
  CDFG graph(!test.cdfg!);
  trav(graph.getSubg());
  }
```

这样，通过 cdfgtool 便可将源代码对应的 CDFG 转换生成相应的 VHDL 代码，从而实现了 C 代码到 VHDL 代码的转换，达到了本课题的目标。

5.4 本章小结

本章阐述了基于抽象语法树生成目标代码的方法，并对本文提出的中间表示模型 CDFG 进行了详细阐述，给出了由抽象语法树生成 CDFG 的算法，及采用的主要的数据结构，最后介绍了利用已有软件将 CDFG 转换生成目标代码的方法。

第6章 编译结果及验证

本文的编译系统实现了从C到VHDL的转化，该系统针对C中主要语句进行了大量的试验，实验结果验证达到了设计要求，由于转化后得到的目标代码量较大，限于篇幅本文只以具有代表性的if语句为例，给出由C到VHDL的转化以及验证。

6.1 验证流程

对编译系统的验证由以下步骤完成：

(1) 给出某个包含if语句的C语言实现。

(2) 通过编译系统转化为VHDL描述。

(3) 在Modelsim仿真软件中对该VHDL描述的硬件进行功能仿真。

(4) 对比硬件与软件运行的结果，得出结论。

6.2 C to VHDL的操作语句算法

本设计基于的基本语句原型的C语言实现如下：

```
extern float a, b, c, d;
void run() {
  a = 1.0;
  if( b == 0.0 ) {
    b = 2.0;
    if( c == 0.0 ) {
      c = 3.0;
      if( d == 0.0 ) {
     d = 4.0;
      } else {
```

```
            d = 5.0;
          }
        } else {
          c = 6.0;
        }
      } else {
        b = 7.0;
      }
    }
```

将上述 C 语言描述的算法输入编译系统，得到的对应的 VHDL 描述，由于转换后的代码量较大，所以只给出了转换后目标代码的主体部分，具体转换结果见附录。

6.3 仿真验证

对该算法进行验证，为 a, b, c, d 均赋值为 0，将上面得到的 VHDL 代码在 modelsim 中进行仿真，得到的模拟结果如图 6.1 所示。

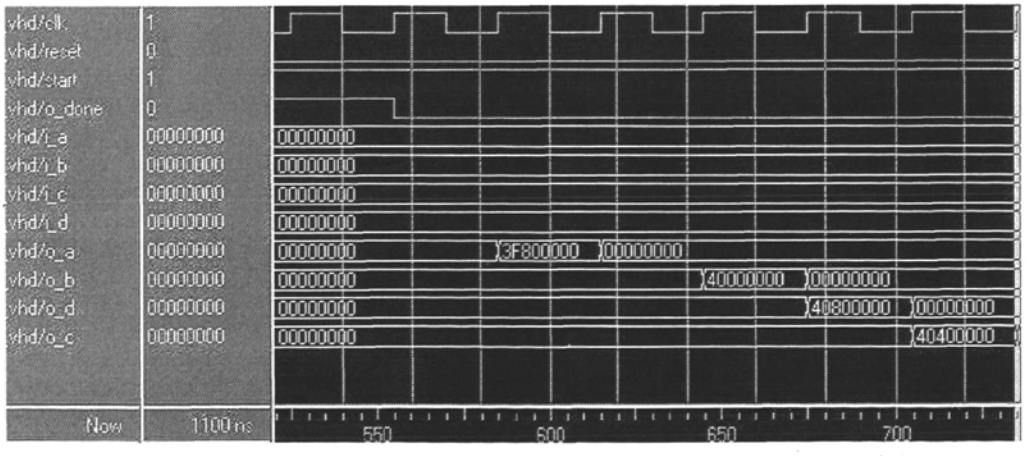


图 6.1 在 modelsim 中对 VHDL 代码仿真结果

图 6.1 中的输出结果为十六进制浮点数，将图 6.1 中的对 VHDL 代码仿真的输出结果，输出的十六进制数对应的十进制数，及 C 代码的输出结果进行比较，得到表 6.1。表 6.1 中第三行为对 VHDL 代码仿真的到的十六进制数对

应的十进制数。

表 6.1 输出结果

输出结果	a	b	c	d
VHDL	3F800000	40000000	40400000	40800000
十进制数	1.0	2.0	3.0	4.0
C	1.0	2.0	3.0	4.0

比较仿真结果与 C 代码执行结果，模拟输出的结果与 C 的运行结果一致。从而验证了本文的编译方法的正确性。

6.4 本章小结

本章通过一个实际的算法，给出软件查找过程及运行结果，并在 Modelsim 仿真软件中对该 VHDL 描述的硬件进行功能仿真，通过比较二者结果，证明了转化过程的正确性。

结论

用C语言描述电子系统，可以提高传统描述的层次和系统设计的效率，并可以复用大量用C描述的算法。由于HDL语言仍是大多EDA工具接受的语言，所以本文的设计实现了C到VHDL的转换。文章分析了C与VHDL的区别，并提出了合理的转化方案，详述了对一些问题的解决方法，最后展示了转换结果。

本文介绍了一个从C语言转换到VHDL语言的编译方法，该编译系统可以作为VHDL高层次综合的前端，接受C语言源描述，编译后生成VHDL语言供后续系统使用。

本文所进行的研究工作如下：

(1) 编译系统前端的设计过程中，使用并改进了许多关键的技术，为源语言设计正则表达式，通过词法分析器的生成工具flex++构建词法分析程序；设计C语言的文法产生式，借助语法分析器的生成工具构建语法分析程序。

(2) 编译系统后端的设计过程中，提出了内部表示模型CDFG作为中间表示，并设计了由抽象语法树到CDFG的转换算法，以及CDFG的数据结构。

(3) 如何将C语言的描述方式翻译成VHDL,本文给出了具体的解决方法。

总之，本文的工作重点在于对面向VHDL算法级行为描述的程序语言编译方法的设计，C语言程序描述经过编译后输出对应的VHDL代码。本编译系统通过了很多实例进行了测试，证明已实现了要求的功能。本文所进行的研究工作在对高级综合技术的研究方面具有一定的参考价值和利用价值。

还需要做的后续工作：

(1) 在编译系统的基础上，进行后续综合系统的设计实现，包括各种分配和调度算法的研究。以便能够实现通过高级语言实现来进行芯片综合这样一个目的。

(2) 作为综合系统的前端，本编译方法还需不断的补充和完善，进一步向VHDL可综合子集的转换，VHDL数据类型的扩充，可转化的C语言子集

的扩充以及转换器的优化。

(3) 完善并添加错误处理程序，增强编译方法的处理能力。

参考文献

[1] 谭浩强. C 语言程序设计. 清华出版社, 1999

[2] 林敏, 方颖立. VHDL 数字系统设计与高层次综合. 电子工业出版社, 2002

[3] 张素芹, 吕映芝等. 编译原理. 清华大学出版社, 2005

[4] 陈火旺, 刘春林等. 程序设计语言编译原理. 国防工业出版社, 2000

[5] 蒋立源, 康慕宁等. 编译原理. 西北工业大学出版社, 2005

[6] 王兼明. VHDL 中间数据格式及其设计库. 计算机学报, 1992, (07)

[7] 陈吉华. VHDL——一种统一的设计描述语言. 微电子学与计算机. 1993, (08)

[8] Kenneth C.Louden. 冯博琴, 冯岚等译. 编译原理及实践. 机械工业出版社, 2000

[9] 王高峰, 赵刚. 独立语法描述的 VHDL 词法和语法分析. 计算机与数字工程, 2007, (05)

[10] 刘明业等编. 数字系统自动设计. 高等教育出版社, 1996

[11] 北京理工大学 ASIC 研究所. VHDL 语言 100 例详解. 清华大学出版社, 1999

[12] Keith D. Cooper, Linda Torczon. 冯速译. 编译器工程. 机械工业出版社, 2006

[13] Charles N. Fischer, Richard J. LeBlanc, Jr. 编译器构造: C 语言描述. 机械工业出版社, 2005

[14] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. 李建中, 姜守旭译. 编译原理: principles, techniques, and tools. 机械工业出版社, 2003

[15] 王兼明, 沈永朝. VHDL 硬件描述语言及其设计环境. 系统工程与电子技术, 1990, (01)

[16] 李原. VHDL 编译系统设计方法及关键技术研究. 四川大学, 2005

[17] Steven S. Muchnick. 赵克佳, 沈志宇译. 高级编译器设计与实现. 机械工

业出版社, 2005

[18] Christopher W. Fraser, David R. Hanson. design and implementation. 电子工业出版社, 2005

[19] Dick Grune. 冯博琴, 傅向华等译. 现代编译程序设计. 人民邮电出版社, 2003

[20] Andrew W. Appel, Maia Ginsburg. 赵克佳, 黄春, 沈志宇译. 现代编译原理：C 语言描述. 人民邮电出版社, 2006

[21] S. S. Munchnick, Advanced Compiler Design and implementation. San Fransisco, California, USA: Morgan Kaufmann Publishers, Inc., third ed., 1997

[22] G. Snider, B. Shackleford, and R. J. Carter, "Attacking the semantic gap between application programming languages and configurable hardware," in FPGA 2001,(Monterey, CA), pp. 115–124, ACM, ACM, February 2001

[23] Celoxica, Handel-C Language Reference Manual. Celoxica Limited, 2001

[24] S. Swan, D. Vermeersch, D. Dumlug¨ol, P. Hardee, T. Hasegawa, A. Rose, M. Coppolla, M. Janssen, T. Gr¨otker, A . Ghosh, and K. Kranen, Functional Specification for SystemC 2.0. Open System C Initiative, 2.0-p ed., October 2001

[25] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren, "Conversion of control dependence to data dependence," in Symposium on Principles of Programming Languages, pp. 177–189, 1983

[26] G.G.deJong.Data flow graph: System specification with the most unrestricted semantics. In Proc. European Desigh Autom. Conf,1991

[27] J. T. J. Van Ei jndhoven and L. Stok. A data flow graph exchange standard. In Proc. European Desigh Autom. Conf, 1992

[28] T.Krol, J. V. Meerbergen, et al. The Sprite input language，an intermediate format for high-level synthesis. In Proc. European Desigh Autom. Conf, 1991

[29] A. L. Davis and R.M. Keller. Data flow program graphs. IEEE Trans. Computers, 1982

[30] Bumbulis, P. and Cowan, D. D. RE2C: A more versatile scanner generator. ACM Letters on Programming Language and Systems. 1993, 2(1-4), 70-84

[31] Burke, M. G. and Fisher, G. A. A practical method for LR and LL syntactic error diagnosis and recovery. ACM Trans. on Programming Language and Systems. 1987, 9(2), 164-167

[32] Cardelli, L. Compiling a functional languge. LISP and Functional Programming. ACM Press.1984, 208-17

[33] Chow, F., Himelstein, M., Killian, E.,and Weber,L. Engineering a RISC compiler system. In proc. COMPCON Spring 86.IEEE, 132-137

[34] Cocke, J. and Schwartz, J. T. Programming language and their compilers: Preliminary notes. Tech. rep., Courant Institute, New York University.1970

[35] Deremer, F. L. Simple LR(k) grammars. Commun.ACM. 1971, 14,453-456

[36] Fraser, C. W. and Hanson, D.R. A Retargetable C Compiler: Design and Implementation. Benjamin Cummings, Redwood City, CA

[37] 刘志鹏, 边计年, 王云峰, 薛宏熙. 面向SOC系统设计的层次化CDFG的扩展. 计算机工程与科学, 2005年27卷4期, 46-48,95

[38] 赵康, 边计年, 吴强, 薛宏熙. C语言系统描述的HCDFG-II实现.计算机工程与科学, 2005年27卷4期, 80-83

[39] 牛亚文, 边计年, 吴强, 薛宏熙. HCDFG-II-面向C语言系统描述的控制/数据流图表示. 计算机辅助设计与图形学学报, 2004年16卷11期, 1547-1552

附录 编译结果

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity tr_design is
    port(
        signal i_a : in std_logic_vector(31 downto 0);
        signal i_a_we : in std_logic;
        signal o_done : out std_logic;
        signal reset : in std_logic;
        signal i_c_we : in std_logic;
        signal i_b : in std_logic_vector(31 downto 0);
        signal i_c : in std_logic_vector(31 downto 0);
        signal clk : in std_logic;
        signal o_d : out std_logic_vector(31 downto 0);
        signal i_d_we : in std_logic;
        signal o_c : out std_logic_vector(31 downto 0);
        signal o_a : out std_logic_vector(31 downto 0);
        signal o_b : out std_logic_vector(31 downto 0);
        signal i_d : in std_logic_vector(31 downto 0);
        signal start : in std_logic;
        signal i_b_we : in std_logic
    );
end entity tr_design;
architecture tr_design_arch of trident_design is
    component regfile is
        port(
            signal i_a : in std_logic_vector(31 downto 0);
            signal i_c_we : in std_logic;
            signal i_c : in std_logic_vector(31 downto 0);
            signal o_d : out std_logic_vector(31 downto 0);
            signal i_clk : in std_logic;
            signal i_b : in std_logic_vector(31 downto 0);
            signal i_reset : in std_logic;
            signal o_b : out std_logic_vector(31 downto 0);
            signal i_d_we : in std_logic;
            signal i_d : in std_logic_vector(31 downto 0);
            signal i_a_we : in std_logic;
            signal o_c : out std_logic_vector(31 downto 0);
            signal o_a : out std_logic_vector(31 downto 0);
            signal i_b_we : in std_logic
        );
    end component regfile;
    component entry is
```

```
    port(
        signal i_b : in std_logic_vector(31 downto 0);
        signal i_c : in std_logic_vector(31 downto 0);
        signal o_b : out std_logic_vector(31 downto 0);
        signal o_a : out std_logic_vector(31 downto 0);
        signal o_d_we : out std_logic;
        signal o_b_we : out std_logic;
        signal i_clk : in std_logic;
        signal o_idle : out std_logic;
        signal i_start_entry : in std_logic;
        signal o_a_we : out std_logic;
        signal i_d : in std_logic_vector(31 downto 0);
        signal o_d : out std_logic_vector(31 downto 0);
        signal o_c_we : out std_logic;
        signal o_c : out std_logic_vector(31 downto 0);
        signal i_reset : in std_logic
    );
end component entry;
signal net_3 : std_logic_vector(31 downto 0);
signal we_b : std_logic;
signal r_b : std_logic_vector(31 downto 0);
signal we_d : std_logic;
signal init_d_we : std_logic;
signal entry_b_we : std_logic;
signal entry_c : std_logic_vector(31 downto 0);
signal init_b : std_logic_vector(31 downto 0);
signal r_d : std_logic_vector(31 downto 0);
signal done_2 : std_logic;
signal entry_b : std_logic_vector(31 downto 0);
signal we_a : std_logic;
signal start_entry : std_logic;
signal m_init_c : std_logic_vector(31 downto 0);
signal reg_done_0 : std_logic;
signal w_a : std_logic_vector(31 downto 0);
signal w_d : std_logic_vector(31 downto 0);
signal entry_d_we : std_logic;
signal r_c : std_logic_vector(31 downto 0);
signal w_c : std_logic_vector(31 downto 0);
signal entry_a : std_logic_vector(31 downto 0);
signal entry_a_we : std_logic;
signal done : std_logic;
signal net_1 : std_logic_vector(31 downto 0);
signal r_a : std_logic_vector(31 downto 0);
signal m_init_b : std_logic_vector(31 downto 0);
signal init_b_we : std_logic;
signal m_init_d : std_logic_vector(31 downto 0);
signal net_2 : std_logic_vector(31 downto 0);
signal idle_entry : std_logic;
signal m_init_a : std_logic_vector(31 downto 0);
signal init_c : std_logic_vector(31 downto 0);
```

```
    signal entry_d : std_logic_vector(31 downto 0);
    signal net_0 : std_logic_vector(31 downto 0);
    signal init_d : std_logic_vector(31 downto 0);
    signal entry_c_we : std_logic;
    signal init_a_we : std_logic;
    signal and_done : std_logic;
    signal w_b : std_logic_vector(31 downto 0);
    signal init_c_we : std_logic;
    signal we_c : std_logic;
    signal init_a : std_logic_vector(31 downto 0);
begin
    init_a <= i_a ;
    net_2 <= (others => '0') ;
    w_c <= m_init_c or entry_c ;
    process ( clk, reset ) is
    begin
        if  (reset = '1')  then
            done <= '1' ;
        elsif  (clk'event and clk = '1')  then
            done <= done_2 ;
        end if;
    end process ;
    net_3 <= (others => '0') ;
    m_init_d <= net_1 when init_d_we = '0' else init_d when init_d_we = '1' ;
    net_0 <= (others => '0') ;
    init_a_we <= i_a_we ;
    o_done <= done ;
    we_b <= init_b_we or entry_b_we ;
    net_1 <= (others => '0') ;
    we_a <= init_a_we or entry_a_we ;
    init_c_we <= i_c_we ;
    and_done <= idle_entry ;
    done_2 <= reg_done_0 and and_done ;
    init_b <= i_b ;
    init_c <= i_c ;
    process ( clk, reset ) is
    begin
        if  (reset = '1')  then
            reg_done_0 <= '1' ;
        elsif  (clk'event and clk = '1')  then
            reg_done_0 <= and_done ;
        end if;
    end process ;
    o_d <= r_d ;
    init_d_we <= i_d_we ;
    we_d <= init_d_we or entry_d_we ;
    o_c <= r_c ;
    m_init_a <= net_2 when init_a_we = '0' else init_a when init_a_we = '1' ;
    m_init_c <= net_0 when init_c_we = '0' else init_c when init_c_we = '1' ;
    m_init_b <= net_3 when init_b_we = '0' else init_b when init_b_we = '1' ;
```

```
    o_a <= r_a ;
    o_b <= r_b ;
    init_d <= i_d ;
    start_entry <= start ;
    w_d <= m_init_d or entry_d ;
    we_c <= entry_c_we or init_c_we ;
    w_a <= m_init_a or entry_a ;
    w_b <= entry_b or m_init_b ;
    init_b_we <= i_b_we ;
    entry_0 : component entry
        port map(
            i_b => r_b,
            i_c => r_c,
            o_b => entry_b,
            o_a => entry_a,
            o_d_we => entry_d_we,
            o_b_we => entry_b_we,
            i_clk => clk,
            o_idle => idle_entry,
            i_start_entry => start_entry,
            o_a_we => entry_a_we,
            i_d => r_d,
            o_d => entry_d,
            o_c_we => entry_c_we,
            o_c => entry_c,
            i_reset => reset
        );
    regfile_0 : component regfile
        port map(
            i_a => w_a,
            i_c_we => we_c,
            i_c => w_c,
            o_d => r_d,
            i_clk => clk,
            i_b => w_b,
            i_reset => reset,
            o_b => r_b,
            i_d_we => we_d,
            i_d => w_d,
            i_a_we => we_a,
            o_c => r_c,
            o_a => r_a,
            i_b_we => we_b
        );
end tr_design_arch;
```

致谢

本文是在我的导师顾国昌教授的悉心指导下完成的。在研究生期间，顾老师无论是在学习、科研还是在生活方面都给予我无私的关怀和精心的教诲。他一丝不苟的工作作风，严谨求实、孜孜不倦的治学态度，稳重踏实的性格都是非常令人敬佩的，始终是我学习的榜样。正是顾老师的言传身教，使我不仅在学术上，而且在对待生活、对待工作的态度等各方面都受益匪浅。能够在顾老师的指导下完成我的研究生学业，是我一生难得的机遇。在此，我要向顾老师表示我最衷心的谢意！

感谢实验室的高忠杰、张浩、张博为、孙霖、孙延腾等同学的的支持与帮助。还要尤其感谢程利新老师在我完成论文的过程中给予的帮助和关怀，在此表示深深的敬意和由衷的感谢。

在哈尔滨工程大学两年多的学习生活中，我度过了最难忘的时光。在此期间，我曾得到了许多老师无私的关怀和热心的帮助与鼓励，以及同学们的关心与帮助，使我在各方面得以成长和成熟起来。在我研究生学习生活即将结束之际，一并表示感谢。

感谢我的亲人朋友这么多年给于我的支持，关心和理解。

最后我要深深的感谢我的父母，正是由于他们无微不至的照顾和悉心的培养以及在精神上和经济上给予的强大支撑，才使得我顺利完成学业！