Interfacing Syntax and Semantics*

by

Cynthia A. Brown and Paul W. Purdom, Jr.

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 113

INTERFACING SYNTAX AND SEMANTICS

Cynthia A. Brown and Paul W. Purdom, Jr.

Revised July 1982

*Research reported herein was supported in part by the National Science Foundation under grant number MCS 7906110.

Interfacing Syntax and Semantics

by

Cynthia A. Brown and Paul W. Purdom, Jr.

Abstract. A method for specifying the first pass of bottom-up compilers using a specialized form of attribute grammars is given. The notation specifies both the syntax of the language being accepted and the detailed flow of semantic information in the compiler. It is a powerful aid in developing a structured, well-documented detailed design.

1. Introduction

The design of a compiler is usually broken into two major facets: a parsing, or syntactic, component, which analyzes the structure of the input, and a semantic component, which generates code. Control, at least for the first pass of compilation, is in the parser: it supervises the collection of information, and calls the appropriate semantic routines as each phrase of the input is recognized.

The design of the parsing part of a compiler using context-free grammars and syntax-directed translation is well understood (see [1], for example). Recently, formal notations for the specification of semantics have been developed. These include denotational semantics [11] and attribute grammars (see [9] for a bibliography). Attribute grammar notation provides a convenient way to specify the overall flow of information in a compiler. Information originates in the scanner and in semantic routines, and is used by semantic routines under the direction of the parser. Since no one routine is responsible for the overall information flow, obtaining a correct, smoothly functioning design is made much easier by the use of an appropriate design tool, such as

attribute grammars.

Attribute grammars typically show where information originates and where it is used, but the details of how it gets from one point to another are omitted from the specification. It is left to the implementor to determine what changes each routine should make to the stack on which information is passed, and which quantities, if any, to store in global variables. Algorithms have been proposed for automatically checking certain classes of attribute grammar specifications and generating detailed implementations from them [2,3,6,10]. Such algorithms, while they do much of the compiler writer's work, do not necessarily lead to programs of the same efficiency as those designed by hand.

In this paper we present a notation based on attribute grammars for specifying the detailed flow of information during the first pass of compilation: the phase during which an intermediate representation is built in an optimizing compiler or code is generated in a simple one. This notation, which we call attribute flow notation, is meant to facilitate hand design of compilers. Because our attention is focussed on one left-to-right pass, the notation is closely related to classes of attribute grammars that can be evaluated in one left-to-right pass, such as the one described by Watt [12]. Our approach differs from previous authors' in its goal of producing a detailed specification of an implementation, rather than a high-level framework for one; accordingly, many details that are suppressed in other notations are made explicit in ours. Attribute flow notation is primarily a tool for the implementor, who is interested in these details, rather than the language designer, for whom they represent an unnecessary distraction.