A PRACTICAL INTERPROCEDURAL ALIAS ANALYSIS FOR AN OPTIMIZING/PARALLELIZING C COMPILER

by
Maryam Emami

School of Computer Science
McGill University, Montreal

July 1993

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Accurate alias analysis is a crucial phase of optimizing/parallelizing compilers for languages which support pointer data structures. The result of the alias analysis is used by most of data-flow analyses and optimization phases in a compiler to produce efficient code.

In this thesis, we introduce a new approach for interprocedural alias analysis which determines the explicit *points-to* relationships between locations in an abstract stack at each program point. Two variables have a points-to relationship if one of them points, or may point, to the other one.

To perform our interprocedural analysis, we have designed a new representation to capture the call-structure of the program, called the *invocation graph*. This is similar to the traditional *call graph*, with some additional properties for the recursive function calls. The invocation graph is used to follow the exact sequence of function calls and returns in the program. In this manner, the precise information is propagated to/from a function.

This work has been integrated in the McCAT optimizing/parallelizing compiler for the C-language. The analysis handles both scalar variables and aggregate data structures (records and arrays).

The results of the points-to analysis provides a base upon which other interprocedural analyses are built. This thesis provides both experimental results and examples to demonstrate the usefulness of the points-to analysis.

Résumé

L'analyse d'alias est une phase extrêmement importante de tout compilateur de langages supportant les pointeurs de données. Le résultat de cette analyse est en effet utilisé par le plupart des méthodes basées sur l'analyse de flots de données ainsi que de nombreuses techniques d'optimisations afin de générer du code efficace.

Dans ce rapport, on introduit une nouvelle approche pour l'analyse inter-procédurale d'alias permettant de déterminer les relations explicites de type *pointe sur* entre les éléments d'une pile abstraite à tout instant d'un programme. Deux variables sont liées par une relation de type *pointe sur* si l'une d'elles pointe, ou peut pointer, sur l'autre.

Notre analyse est basée sur une nouvelle structure de données appelée *graphe d'invocation* permettant de représenter précisemment la hiérarchie d'appels et de retours de fonctions d'un programme. Cette structure est similaire aux traditionnels *graphes d'appels* tout en possédant des propriétés additionnelles pour le traitement des appels récursifs de fonctions. Le graphe d'invocation permet de déterminer la séquence exacte d'appels/retours de fonctions dans un programme. Il autorise la propagation d'une information précise vers (ou à l'extérieur) des fonctions.

Ce travail a été intégré dans le compilateur C McCAT. La méthode présentée traite de façon détaillée aussi bien les variables scalaires que les structures de données plus complexes de C, telles les tableaux ou les enregistrements.

Le contexte du travail présenté dans cette thèse est général et peut servir de base pour l'élaboration d'autres méthodes d'analyse inter-procédurales. Ce manuscrit présente à la fois des travaux expérimentaux ainsi que des exemples décrivant l'analyse de type *pointe sur* développée.

Acknowledgments

I am thankful to Professor Laurie Hendren, my supervisor, for her technical and financial support throughout the course of my study at McGill. I am grateful for her understanding, patience, and kindness. She was always there to listen and to help. THANKS a lot Laurie.

I would like to thank W. A. Landi (currently at Siemens Corporate Research) for all the technical discussions that we had through email and for sending me the results of his method for comparisons. Further, I would like to thank Bhama Sridharan, one of my best friends, for her presence during all the stages of my study and for her technical support. I am thankful to V.C. Sreedhar for his constant support. He is the one who will never say no, the one who has an answer for every question. I am grateful for Ravi Shanker who was there to cheer up people with his warm smile. I am thankful to Chris Donawa, our McCAT and ACAPS administrator, for his great administrative work. I would like to thank Rakesh Ghiya for his friendly attitude and all the technical discussions we had. Further, I am thankful to Clark Verbrugge who provided the table of his results for the general constant propagation. I am grateful for all the people who commented on my thesis and also who provided me with buggy test programs to test my software, especially V.C. Sreedhar. I would like to say a special thanks to all the ACPAS members who made such a friendly environment to work together, and, in particular Justiani, Michel Gagne, Chandrika Mukerji, Cecile Moura, Dhrubajyoti Goswami, Ana Erosa, and Luis Lozano. Special thanks are also due to dear Lise Minogue, the secretary of our department, for her kindness.

Further, I would like to thank Dr. Mehdi Mohageg, Dr. Mahmood Nourbakhsh, and Mr. Eskandar Ilkhan without whose support and help I could never have continued my studies. I am thankful to my dear parents, and to my dear sister, Mardjan, for their understanding. I am grateful for Amir Keyvan Khandani, one of my best friends, for being so patient and supportive.

I thank God who has surrounded me with such nice people.

Dedicated to my mother, Irandokht Yadollahi

Contents

List of Figures

List of Tables

Chapter 1

Introduction

New and innovative compiling techniques as well as novel architecture design philosophies are required to exploit the ample hardware resources provided by the recent advances in VLSI technology and architecture design. To provide both high performance and cost effectiveness, it is essential that compilation techniques and architecture models are developed together, so that the effects of one on the other can be studied.

Optimizing and/or parallelizing compilers play important role in the design of high-performance systems. An optimizing compiler requires detailed and accurate data-flow analysis in order to perform code-improving transformations and to generate efficient code. Data-flow analysis is the process of collecting information such as the definitions and uses of variables in a program. At the core of accurate data-flow analysis, lie effective dependence and alias analysis mechanisms. The ability to accurately disambiguate memory references is critical in determining precise data-dependence between variables used and defined in a program.

An important thrust in architectural improvements in modern architectures involves increasing parallelism. The availability of parallel hardware is of no use without the appropriate software to exploit parallelism. In response to this requirement, great strides have been made in compiler technology to automatically detect parallelism and perform transformations to enhance parallelism and minimize dependence in the code. A major part of such dependency minimization and parallelism enhancement is based on using **alias** information.

1.1 Alias Analysis

An *alias* occurs at some program point when two or more names exist for the same memory location. The aliases of a particular name at a program point are all other names that refer to the same memory location at that point. *Alias analysis* is a technique that collects the information required to determine which names are aliased at each program point.

Traditionally, aliases have been represented as sets of alias pairs. Figure 1.1 demonstrates this, using some simple examples. At program point `p1` in Figure 1.1(a),`*a` and `b` refers to the same memory location and therefore `*a` and `b` are aliased. This aliasing is usually represented by the pair <*a,b>. Figure 1.1(b) illustrates another example of aliasing where at program point `p2`, `*a`, `*c` and `b` are sharing the same memory location. This results in the following alias pairs: {<*a,b>, <*c,b>, <*a,* c>}.

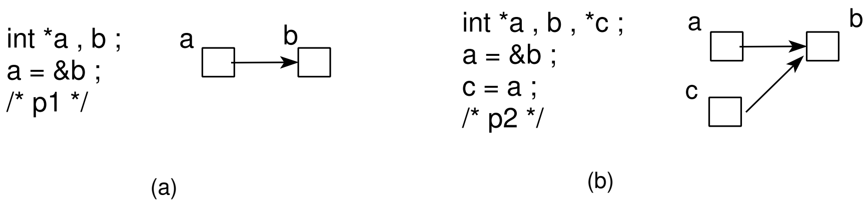


Figure 1.1: An example of C program.

The appropriate alias analysis techniques to detect the aliases that occur in Fortran, due to call-by-reference parameters have been studied and developed [Bar78, Ban79, Mye81, Coo85, CK89]. Unlike the problem of aliasing due only to call-by-reference, where aliases are **only** created by the association of actual arguments with formal parameters due to procedure call invocations, the aliasing problem in C is much more complex. In C, one must consider multi-level reference types such as `int*` and `int**`, and one must support the `&` and `*` operators that allow new aliases to be created at any point in a program (and not just at procedure call time) in addition to call-by-reference parameter passing (call by reference in C is achieved by passing the address of the parameter). The problem of alias analysis in C can be separated into the following three sub-problems:

(1) analysis for call-by-reference parameters,

(2) analysis for stack-allocated, non-recursive data structures, and

(3) analysis for heap-allocated, recursive data structures.

In this thesis, we present a novel technique for precise analysis of the first two subproblems. Alias analysis for heap-allocated objects is more complicated and has been the subject of some interesting studies in the past [JM81, JM82, LH88, HN90, CWZ90, HHN92a]. The analysis presented in this thesis has been designed to complement the path matrix analysis [Hen90, HN90].

1.2 Applications of Alias Analysis

The presence of pointers and reference parameters makes data-flow analysis more complicated. Without accurate alias information, one needs to make the worst case conservative assumptions regarding pointers – any pointer variable can refer to any other variable in the program. Some of the typical analyses that will benefit from alias analysis are: live variable analysis, data dependence analysis, common subexpression elimination, copy propagation, array range checking, constant propagation, dead-code elimination, induction-variable elimination, reaching definitions, use-definition chains, and so on.

In the following subsections, we present some examples which benefit from accurate alias analysis.

1.2.1 Constant Propagation

Constant propagation is a process of collecting constants at each program point, and propagating the constants along control flow paths. As an example, consider the following program:

```
                                        Constants              Constants
   int  *a, b, c, d ;               without alias info.   with alias info.

   c = 3 ;      /* stmt 1 */         {c=3}                 {c=3}
   a = &c ;     /* stmt 2 */         {c=3}                 {c=3}
   read(d) ;    /* stmt 3 */         {c=3, d=?}            {c=3, d=?}
   b = *a ;     /* stmt 4 */         {c=3, d=?, b=?}       {c=3, d=?, b=3}
```

At statement 1, the variable `c` gets the value of 3. At statement 2, `*a` and `c` become aliased. At statement 3, the variable `d` gets an arbitrary value. Now let us look at statement 4 more carefully. Without the alias information, the worst case assumption says that *a can refer to any of the variables b, c, or d. However, in the case that we have access to the alias information, by using the fact that *a is identical to c, we can say that b is equal to the constant 3. This allows the optimizer to replace *a with the constant 3.

1.2.2 Reaching Definitions

A *definition* of a variable x is a statement that assigns, or may assign, a value to x. A definition d is said to *reach* a point p if there is a path from the point immediately following d to p. Consider the following example:

```
    int *a, b, c ;                Reaching Def.       Reaching Def.
                                without alias info.  with alias info.
d1: c = 3 ;     /* stmt 1 */    {d1}                 {d1}
d2: a = &c ;    /* stmt 2 */    {d1, d2}             {d1, d2}
d3: *a = 5 ;    /* stmt 3 */    {d1, d2, d3}         {d2, d3}
d4: b = c ;     /* stmt 4 */    {d1, d2, d3, d4}     {d2, d3, d4}
```

At statement 2, we get the alias pair <*a,c>, and the definition of c at statement 1 reaches this point. If we do not have the alias information at statement 4, we should assume that the reaching definition of c can be either statement 1 or statement 3 (because *a at statement 3 could be either c or d). But, if we know that *a is aliased to c, we can definitely say that only statement 3 reaches statement 4. Furthermore, using Constant Propagation at statement 4, we obtain: b $= 5$.

1.2.3 Replacement of Aliased Variables

If two variables are aliased through all of the execution paths of the program, they can be replaced by one another. This replacement is a useful tool in most of the analyses. In the following, we will briefly explain the effect of such replacement on **live-variable analysis**.

In live-variable analysis, we want to know for a given variable x and program point p, whether the value of x could be used in some execution path starting at p. One important use of live-variable analysis is in register allocation. The live-variable information allows the register allocator to efficiently use the registers for storing the variables. Consider the following example:

```
int *a, b, c, d ;                                   Live-variables
                                                without alias replacement
b = 2;                 /* stmt 1 */                   {}
d = 1 ;                /* stmt 2 */                   {b}
c = 4 ;                /* stmt 3 */                   {d, b}
printf ("%d", d, b); /* stmt 4 */                     {d, c, b}
a = &c ;               /* stmt 5 */                   {d, c, b}
printf ("%d ", *a);  /* stmt 6 */                     {}
```

Live-variable is a backward analysis in the sense that the analysis starts from the last statement of the program and continued to the first statement. Due to this fact, we start from statement 6 of the given example. Without having the alias information, the worst case assumption at statement 6 says that $*$`a` can be any of the variables `b, c,` and `d`. Consequently, at statement 5, all of these variables are live and only `a` gets killed. The same set of variables are live at statement 3 while variable `c` gets killed. Considering these facts, variables `b` and `d` are live after statement 3. Now let us consider the same example after the replacement of the alias information. This information results in the replacement of $*$`a` by `c` in statement 6. The same program after this replacement is shown below:

```
int *a, b, c, d ;                                   Live-variables
                                                with alias replacement
b = 2;                 /* stmt 1 */                 {}
d = 1 ;                /* stmt 2 */                 {b}
c = 4 ;                /* stmt 3 */                 {d, b}
printf ("%d", d, b); /* stmt 4 */                   {d, c, b}
a = &c ;               /* stmt 5 */                 {c}
printf ("%d ", c);   /* stmt 6 */                   {}
```

Since `c` is the unique variable used at statement 6, `c` is the only live variable at statement 5. Thus, using alias replacements, the live ranges of variables `d` and `b` are considerably reduced.

1.2.4 Alias Information in Building the Call-graph

The alias information has a special use for function pointers. Instead of a direct call to a function, the address of the function can be assigned to a variable of type function pointer which will be called later. Without having the alias information for function pointer variables, it is impossible to know which function is being called at a particular call-site. Due to this fact, the construction of the complete call-graph is not accurate in presence of function pointers. In order to complete the call-graph, the worst case assumption has to be taken which in this case says that each call to a function pointer is equivalent to calling all the possible functions in the program. This leads to a very inaccurate result, whereas by using alias analysis a far more precise result will be obtained.

```
f ( ) {
. . .
}
g ( ) {
. . .
}
h ( ) {
. . .
}
main ( ) {
int       (*func_ptr) ( ) ;

func_ptr = f ;     /* stmt 1 */
func_ptr ( ) ;     /* stmt 2 */
}
```

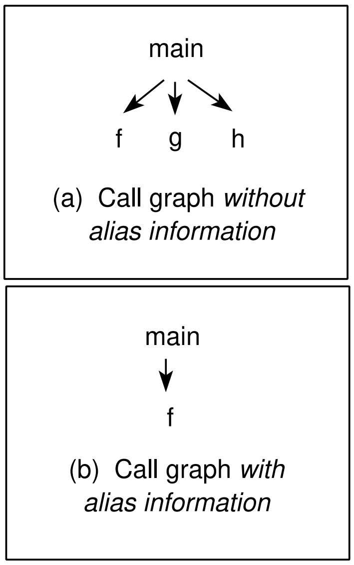


Figure 1.2: An example of the use of alias in building call-graph.

Consider the example shown in Figure 1.2. At statement 1, $*$`func_ptr` and `f` become aliased. At statement 2, without the alias information, one must assume that the call to `func_ptr` can be a call to any of the functions `f`, `g`, and `h`. However, with the alias information in hand, we can restrict this to the set of all functions aliased to the function pointer (in the given example, only `f` is aliased to the function pointer `func_ptr`.

1.3 McGill Compiler Architecture Testbed (McCAT)

In order to study quantitatively the effect of various compilation techniques on sophisticated architectures, it is necessary to develop a complete compiler-architecture testbed. Our alias analysis method has been developed hand-in-hand with the development of the McCAT [Sri92, HDE+92] compiler. Figure 1.3 gives a general view of

McCAT system.

The first component of such a testbed is a compiler which supports both high-level and low-level compiler transformations which translate high-level programs to low-level programs suitable for a variety of architecture simulators and real machines. For example a precise alias analysis is more effective at the higher-level representation, while instruction scheduling needs to be done on a lower-level representation. On the other hand, the information collected in the high-level intermediate representation should be passed to the lower level, to be used by low-level analysis. The second component consists of architecture simulation tools that process the output of the compiler to produce a variety of performance results. McCAT was designed and developed with the consideration of the above objectives.

The McCAT compiler compiles C programs. C is chosen as the source language, because of: (i) its wide range of usage, (ii) having a variety of programming languages features, and (iii) being powerful because of the flexibility of its usage.

To avoid the redundant work of building an efficient front-end, the front-end of GNU-C compiler has been used and modified. The source code of GNU-C compiler is freely available.

The McCAT compiler first translates the C program into FIRST which is a high level representation of the program. Next, a series of tree transformations are performed on FIRST to create SIMPLE which is a simplified Abstract Syntax Tree (AST). SIMPLE forms the intermediate tree representation for analyses and high-level optimizing transformations.

A more detailed description of McCAT will be presented in Chapter 2. The readers are referred to [Sri92, HDE+92] for a complete discussion.

1.4 Thesis Contributions

This thesis reports on a new, general, and accurate alias processing technique that estimates the alias information at each point of a given program. This analysis is performed on the SIMPLE representation of C programs.

Our approach differs from other approaches in that we do not compute the alias pairs explicitly. We have chosen an approximation that computes the relationships between abstract stack locations. The central idea is that since we are dealing with data-structures that are stack-allocated, we can model the real stack with an abstraction of the stack. Using these abstract stack location, we estimate which stack

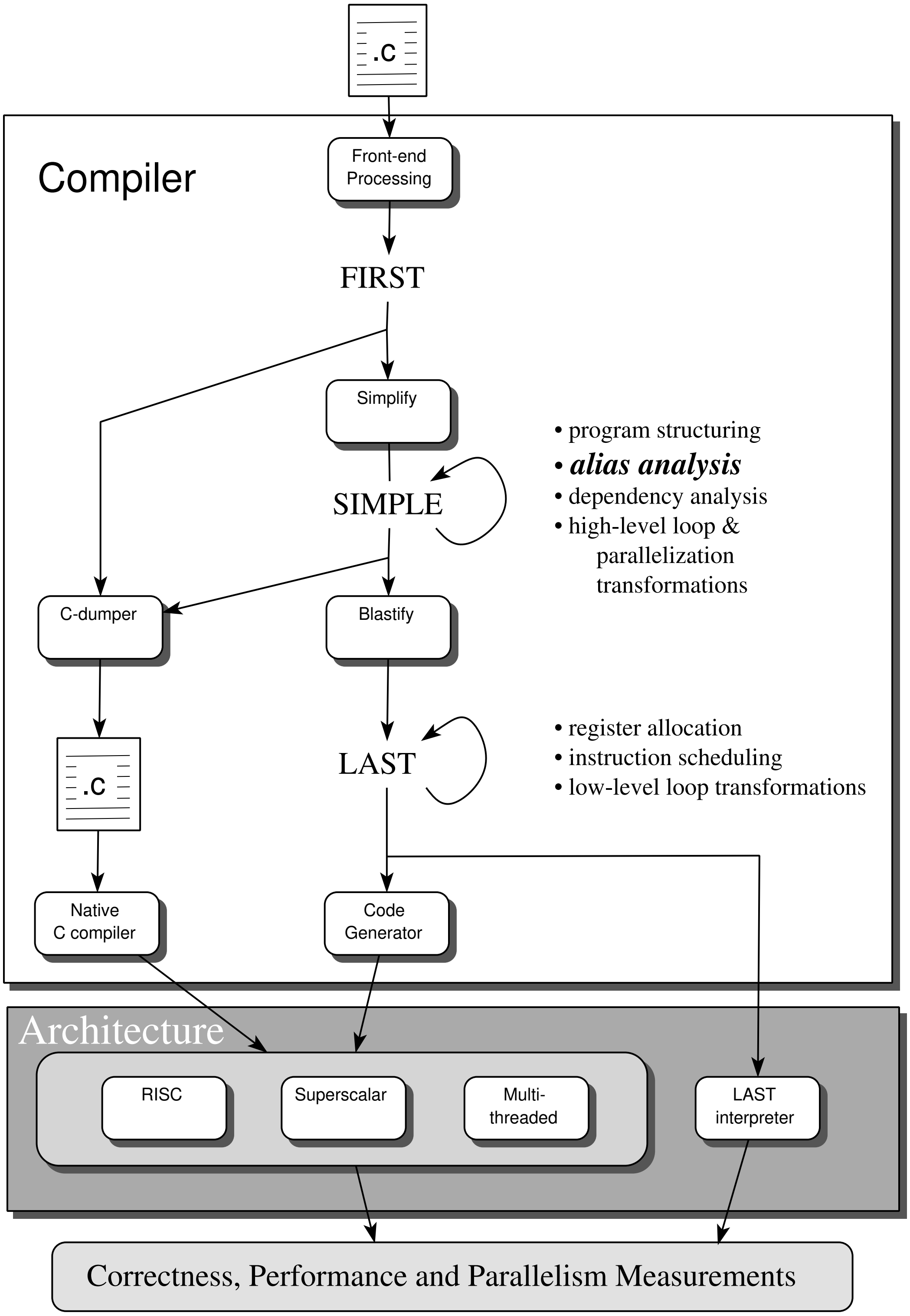


Figure 1.3: A general view of McCAT.

locations point-to other stack locations. Thus, we call our analysis, points-to analysis rather than alias analysis. This is a direct and simple approach that gives accurate results with which we can perform a straight-forward dependence analysis based on the *reads* and *writes* to the abstract stack locations. Furthermore, this information about relationships is sufficient to compute the conventional alias information.

In summary, the main contributions of this thesis are:

- Design of an accurate interprocedural points-to analysis technique for stack-allocated data-structures (including records and arrays) which forms the basis for the design criteria of the intermediate representation SIMPLE.
- Development and implementation of the analysis on the compositional control statements, like `if`, `while`, `switch`, `for`, `repeat`, `break`, and `continue`.
- Design and implementation of a general framework to perform interprocedural points-to analysis in the presence of both recursive and non-recursive procedure calls.
- Presentation of experimental results indicating the effectiveness of the analysis.
- Discussion of how the points-to analysis has been used to improve many other analyses in the McCAT compiler.

It should be mentioned that since we are focusing on structured and structurable programs, any program using `gotos` in an unrestricted manner must be first converted into an equivalent structured program [Ero93].

1.5 Thesis Organization

Chapter 2 discusses the required background and the McCAT testbed. Chapter 3 provides detailed information about the intraprocedural points-to analysis on scalar variables. Chapter 4 is focused on the design of invocation graph and the details of interprocedural points-to analysis (including recursive function calls). Chapter 5 discusses aggregate structures (record and array types). Chapter 6 discusses other features of the C-language like dynamic data allocation and type casting. Chapter 7 is devoted to some implementation details. Chapter 8 is concerned with the practical aspects of our analysis where we give numerical results using standard benchmarks and explain other analyses which are using our result. Chapter 9 presents a review of related work. Finally, Chapter 10 contains some concluding remarks and directions for the future work.

In most parts of the thesis, we assume that the reader has some understanding of the C-language.

Chapter 2

Background

In this chapter, we provide the relevant background material required for the understanding of the following chapters. More specifically, we focus on the SIMPLE intermediate representation. SIMPLE is designed to effectively support various analyses and transformations, in particular alias analysis. This chapter is a summary of the work done by Bhama Sridharan. Readers are referred to [Sri92, HS92, HDE$^+$92] for a complete description.

In order to separate the front-end processing from other analyses and optimizations, the McCAT compiler is designed to support a family of intermediate representation, namely: FIRST, SIMPLE, and LAST (refer to Figure 1.3). These three representations can be summarized as follows:

1. FIRST: a high-level Abstract Syntax Tree (AST) representation of the original program. FIRST retains program and data declarations as written by the programmer.
2. SIMPLE: an intermediate AST representation based on FIRST. In SIMPLE, complex program statements and expressions are presented in a simplified form. SIMPLE is designed for high-level compiler transformations, analyses and optimizations.
3. LAST: a low-level AST representation. LAST is designed for register allocation, instruction scheduling, code generation and low-level transformations [Don93].

Since the analysis reported in this thesis is performed at the SIMPLE level, we focus on the SIMPLE representation in this chapter. We first give an introduction to the FIRST AST in Section 2.1 and then we explain the SIMPLE AST in Section 2.2.

2.1 An Introduction to FIRST

FIRST is built to separate the front-end processing (e.g. scanning, parsing, and type checking) from high-level analysis, optimization and code generation.

The intermediate representation of the original GNU C compiler [Sta90] is based on the Register Transfer Language (RTL). In the GNU C compiler, the RTL representation is produced one statement at a time. For each statement, the parser builds an abstract syntax tree, converts it to RTL, and then frees the corresponding abstract syntax tree. In this manner, the RTL intermediate code for an entire function is generated. After optimization of the RTL, assembly code for a program is generated, one function at a time. For several reasons which are explained in the next section, FIRST, SIMPLE, and LAST have been designed to replace RTL in McCAT compiler.

In Section 2.1.1, the motivation for building FIRST is explained. In Section 2.1.2, the modifications that have been done to the GNU C compiler in order to build FIRST are given.

2.1.1 Motivations for Building FIRST

- As mentioned before, the GNU C compiler generates the RTL for an entire function. Then, several intraprocedural optimizations are performed on RTL to produce the target code for that function. After generating code for a function or a top-level declaration, all storage used by the function definition is freed unless the function is "inlined". As a result, interprocedural optimization can not be performed by the original GNU C. Furthermore, as the intermediate code for the entire program is unavailable, high-level interprocedural analysis and detailed alias analysis techniques can not be applied.
- High-level compiler optimizations such as loop transformations and array optimizations are extremely difficult to perform on the RTL code. This is due to the fact that the identity of loop structures and array references are completely lost at this low level. For example, it is difficult to perform the array dependence analysis because the array references are broken down to lower-level statements. Similarly, as loops are transformed into blocks with goto's and labels, high-level loop transformations are difficult to perform. Retaining the identity of loop structures and array references enables us to perform a number of high-level loop transformations which are otherwise far more difficult to perform.

2.1.2 Modifications to GNU C to Build FIRST

In the original GNU C compiler, the syntax tree is created up to the expression level. Once the statement has been parsed, the tree nodes representing these expressions are freed. Since we are interested in building the syntax tree for the entire program, the parser was modified to retain the tree nodes. The parser is further modified to continue building the AST for the complete program. The parser now constructs the FIRST tree for the entire program.

In RTL, different kinds of loops are not differentiated from each other. New nodes to construct different kinds of loop structures, such as: while-loop, for-loop, and do-loop constructs, are added to FIRST.

An additional field, the TREE_INFO field, has been incorporated in the basic tree_node structure to store the data-flow analysis information.

2.2 SIMPLE Intermediate Representation

The SIMPLE intermediate representation was specifically designed to handle high-level data-flow analyses and transformations, and in particular alias analysis.

In the rest of this section, we give the important characteristics of SIMPLE (Section 2.2.1), the transformation from FIRST to SIMPLE (Section 2.2.2), and a general overview of the SIMPLE AST (Section 2.2.3).

2.2.1 Important Characteristics of SIMPLE

Compositional Representation: The SIMPLE representation is a compositional representation of the program where the control flow is explicit. For example, it is possible to analyze a while loop by analyzing only its components, namely, the conditional expression and the body. This gives the opportunity for alias analysis to take a compositional approach in the implementation. This kind of compositional representation has three advantages: (i) the flow of control is structured and is explicit in the program representation, (ii) structured analyses tools supporting such techniques can be used to analyze all the control-flow constructs, and (iii) it is simple to find and transform groups of nested loops. It should be noted that in addition to ordinary compositional constructs such as conditionals and loops, the compositional approach directly support the commonly used `break` and `continue` statements for loops, and the `return` statement for procedures and functions. However, unrestricted use of `goto` is not compositional and cannot be supported directly. Any program with unstructured control flow must be converted into an equivalent program with structured control flow [WO75, Bak77, Amm92]. The McCAT compiler overcomes the problem associated with `goto` statements by transforming unstructured programs to structured program through a process called *restructuring* [Ero93].

Explicit Array and Structure References: The identity of array and structure references is retained, i.e., the array and structure references are not broken down into a series of lower-level statements that perform address computations. This is required to make full use of high-level information such as array dimension, array size, pointer types, and recursive structure types. Without this information, alias analysis could not collect the alias information related to arrays and structures. This information is essential in array dependence analysis [Jus93].

Types and Typecasting: The exact type information and type casting is also retained. Often alias analysis takes advantage of type information to provide more accurate results. For example, if there are no type casts available, it can be inferred that a variable of one type cannot be aliased to a variable of another type using type information. A more advanced example is the use of recursive types for dynamically-allocated pointer structures [HN89, Deu92].

Pervasive Data-flow Information: It is possible to transmit important data-flow information collected at a higher-level intermediate representation to a lower-level, and thus improve the effectiveness of the low-level transformations. For example, alias analysis information collected at a high-level can be used to perform better dependence analysis, and therefore better instruction scheduling, at a lower-level.

Simple to Analyze: The intermediate representation is simple enough so that it could be analyzed in a straightforward manner. To simplify accurate analysis of such a program, one needs to break down complex structures and statements into simpler cases. In SIMPLE, there are a small number of basic statements all at a level at which structured analyses rules can be easily specified, i.e. at a level most suitable to perform alias analysis. Further, one is able to represent any complicated C statement or expression as a sequence of these basic statements. Similarly, the conditional parts of while-loops, for-loops, do-loops and if-statements are simple expressions. Any complicated expression is simplified when represented in the intermediate form. This results in a reduced number of possible cases and allows the alias analysis to define a general rule for each case. Furthermore, in C, side-effects can occur in many places where one expects an expression. In this simpler form, statements that can have side-effects are clearly separated. This assure the generality of rules that we provide for alias analysis.

Clear Semantics: SIMPLE has clear and obvious semantics. One part of this process is clarifying some of the implicit meanings in C programs. Consider the example shown in Figure 2.1. The statement `b = a` means "assign the address

```
main(){
    int *b, a[7] ;
    b = a ;
}
```

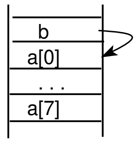


Figure 2.1: An example of pointer to array in the C-language.

of the first item of `a` to `b`", and not "assign the value of `a` to `b`" as one would expect for a scalar assignment. These two interpretations of the same statement can make a difference in alias analysis. For example, in the first case `*a` would be alias to `b`, while in the second case, the alias information would not change. These implicit semantic rules in C are made explicit in SIMPLE.

Interprocedural Analysis: SIMPLE retains all the information about the complete program, so that interprocedural data-flow and alias analysis can be performed. This is particularly important when we have non-scientific code composed of many small and possibly recursive procedures.

2.2.2 Transformation of FIRST to SIMPLE

The following are some of the transformations that have been done to transform FIRST to SIMPLE. The detailed explanation is given in Section 2.2.3.

- Complex variable names are split up whenever it is possible.
- Loops, switches and conditionals are transformed into SIMPLE format.
- Assignment statements and expression statements are broken down into 3-address code format.

A complete grammar for SIMPLE is given in Appendix A. In the next section, a brief overview of SIMPLE is given.

2.2.3 An Overview of SIMPLE

SIMPLE is basically a simplified form of FIRST. It is built of a tree_node structure. Tree_nodes are varied according to the data type and expression. They are built of the following two major parts:

1. The first part is common to all the tree_nodes. Following is the list of the corresponding major fields:
   - Tree_uid : contains a unique identification integer for the corresponding tree_node.
   - Tree_type: points to the type of the tree_node.
   - Tree_code: contains an enumerated type integer which gives the name of the tree_node.
   - Tree_info: points to an array of pointers which holds data-flow information.
2. The second part of tree_nodes is dependent on the corresponding tree_code. For example, a node with tree_code MODIFY_EXPR (modify expression) has two fields to hold the left and right hand sides of an assignment, while IF_STMT (if statement) has three nodes to hold condition expression, then-body, and else-body.

Tree_nodes are classified into seven basic types. Different cases together with some examples for each case are shown in Figure 2.2. In the rest of this chapter, we explain these different basic cases.

Constant Nodes

These nodes represent constants in C-language. They have the common suffix _CST. Following are the examples given in Figure 2.2:

- INTEGER_CST: its value can be any integer constant (e.g. 7).
- REAL_CST: its value can be any float number (e.g. 5.7).
- STRING_CST: its value can be any string constant (e.g. "abc").

Type Nodes

As mentioned before, tree_type is one of the fields in the common part of each node. This field points to a node of type "type node" which contains the type of the corresponding node. Type nodes have the common suffix _TYPE. Some of the different kinds of type nodes are listed in the following:

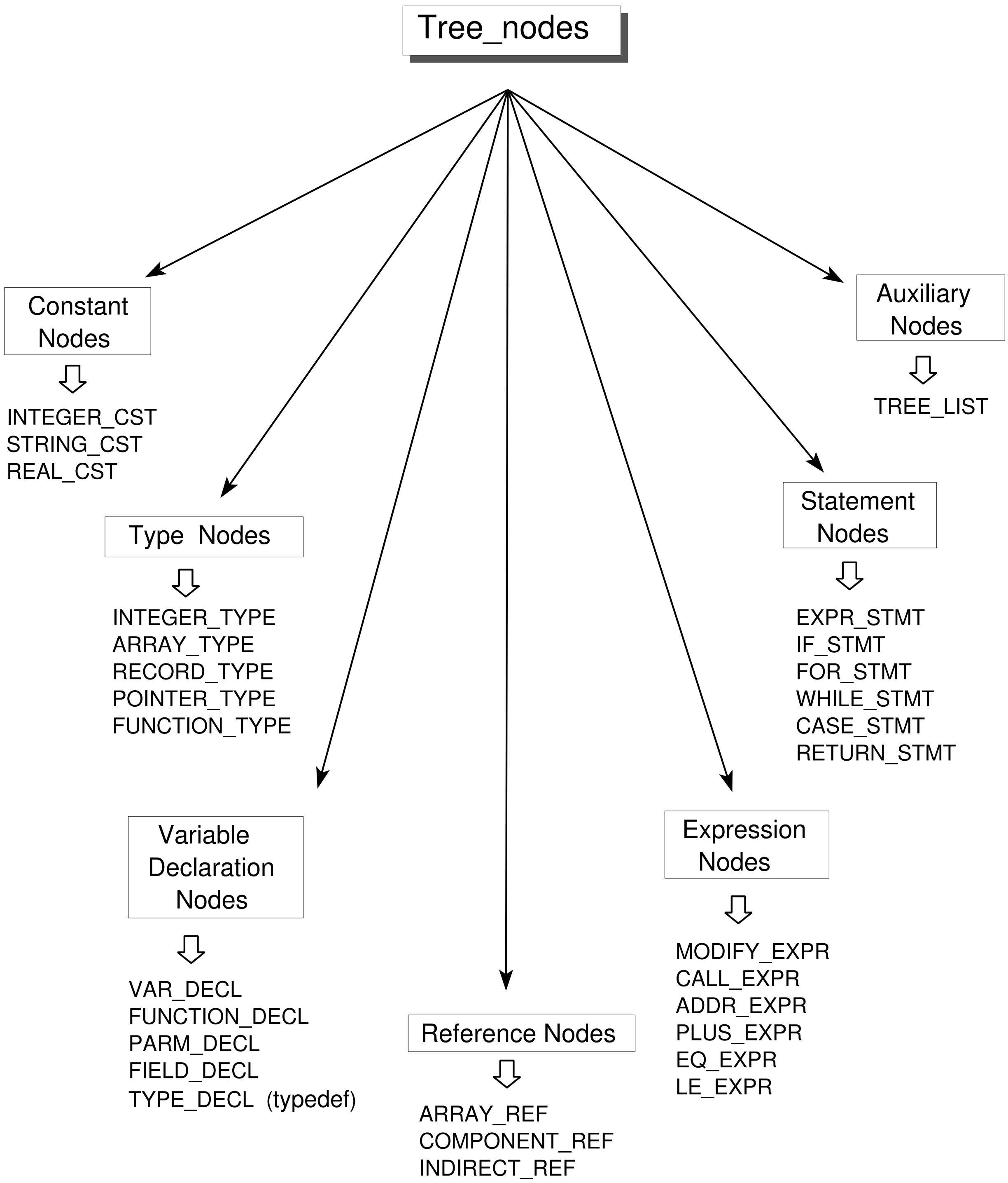


Figure 2.2: Different cases of tree_nodes with some examples of each case.

- INTEGER_TYPE: used for integer basic type. VOID_TYPE, REAL_TYPE, and CHAR_TYPE are other examples of basic type.
- ARRAY_TYPE: used for specifying the array type. Among other fields, this node contains the dimension of the array. The tree_type field of this node points to the type of the array which can be any valid type.
- RECORD_TYPE: used as the type of a structure variable. It has a pointer to the fields of the corresponding structure. Each of the fields is built of a FIELD_DECL node. This will be explained shortly.
- POINTER_TYPE: used for the pointer type. The tree_type field of this node is another "type node". For example, if variable `a` is of type `int*`, the type of `a` would be POINTER_TYPE of type INTEGER_TYPE.
- FUNCTION_TYPE: used for functions. The tree_type field of this node is another "type node" which contains the type of the corresponding function. For example, if a function is of type `void`, the type of the related FUNCTION_TYPE would be VOID_TYPE.

Variable Declaration Nodes

These types of nodes which have the common suffix _DECL, represent variables declaration. All of these nodes have a field "name" which contains the name of the corresponding variable. These nodes also contain some other information which are beyond our discussion. The tree_type field (a field of the common part) contains the type of corresponding variable. Following are some examples of the declaration nodes:

- VAR_DECL: used for any global or local variable.
- FUNCTION_DECL: used for the function declaration. One of its fields is a pointer to the list of formal parameters.
- PARAM_DECL: used for formal parameters of a function declaration.
- FIELD_DECL: used for the fields of a structure.
- TYPE_DECL: used to represent a type defined by the user, using the `typedef` command.

Reference Nodes

The more common cases of these nodes are listed in the following. They have the common suffix _REF. Readers are referred to the subsequent part for some examples.

- ARRAY_REF: used for representing arrays (e.g. a[i][5]).

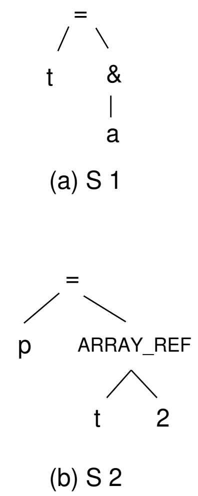
- COMPONENT_REF: used for representing structures (e.g. a.b.c).
- INDIRECT_REF: used for representing the indirect access to a variable (e.g. *a).

Variables are either scalar or aggregate (array and structure). Since scalar variables are straight forward (they are of either VAR_DECL or PARM_DECL tree_node type), we explain the aggregate variables. We first show how variable references are represented and then we show how the complicated cases are broken down into simpler ones.

The example presented in Figure 2.3 is devised to illustrate array references in SIMPLE. Assignment statement 1 (S1) is equivalent to `t = &a[0]`. This statement

```
main(){
    int   a[10], *t, p, q ;
    char *str ;
    int   b[5][7] ;

    t = a ;         /* S1 */
    p = t[2] ;      /* S2 */
    q = a[1] ;      /* S3 */
    str = "abc" ; /* S4 */
    b[2][i] = 6 ; /* S5 */
}
```



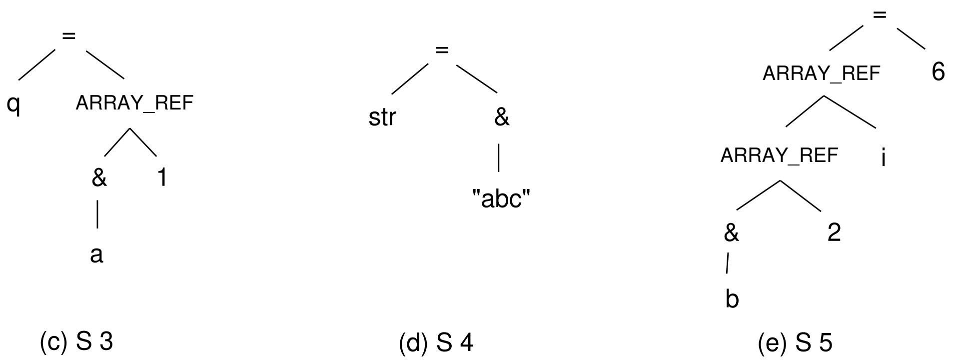


Figure 2.3: An example for the representation of arrays in SIMPLE.

implicitly assigns the address of `a` to `t`. Since one of the goals in the design of SIMPLE is to have clear semantics, the address operator is made explicit in the SIMPLE AST.

This is represented in Figure 2.3(a). As it is shown in statements 2 and 3, the syntax of accessing an array through a pointer to an array (`t[2]`) and the actual array (`a[1]`) look exactly the same, but, semantically they are different. The address of `t[2]` is computed by increasing the contents of `t` by 2, while the address of `a[1]` is computed by increasing the address of `a` by 1. SIMPLE clarifies this difference by adding an address operator before the array reference. This is shown in Figures 2.3(b) and (c). At statement 4, a string constant `"abc"` is assigned to variable `str`. Since a string is actually treated as an array, it will also have an address operator as shown in Figure 2.3(d). Finally, Figure 2.3(e) shows the representation of a two-dimensional array.

In SIMPLE, different fields of a structure are connected through a COMPONENT_REF node. An example of the SIMPLE tree for variable `d.c.n` is shown in Figure 2.4.

```
struct cat{
    int         a ;
    struct cat *next ;
} ;
struct foo{
    int        b ;
    struct cat c ;
} d ;

d.c.a = 1 ;
```

⇒

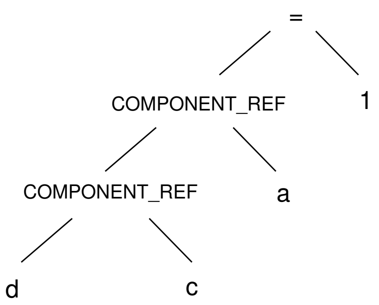


Figure 2.4: An example for the representation of structures in SIMPLE.

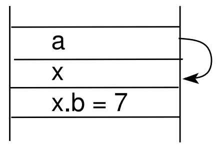
Now that the representation of arrays and structures is clarified, we show how the complicated data structures, which are a combination of arrays and structures (e.g. a.b[4].c), are broken down into simpler cases. This is done as long as the meaning of the variable reference is not lost. For example: `a.b.c, a, a[5][i], (*a).b, (*a).b.c` are some basic cases which can not be broken down.

The example in Figure 2.5 is devised to show why cases like `(*a).b` can not be further broken down. The replacement of statement 1 by statement 2 and 3 does not give the same result. At statement 1, variable `(*a).b` modifies the location `x.b` while at statement 3, `temp0.b` is the variable which is modified. Since these two cases are not identical, `(*a).b` or `(*a).b.c` can not be broken down.

```
struct foo{
    int b ;
} x, *a ;

main()
{
    x.b = 5 ;
    a = &x ;
    (*a).b = 7 ; /* S1 */
}
```

(a) The original C program.

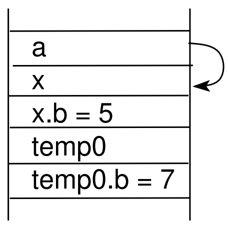


(b) The stack representation of the original program.

```
struct foo{
    int b ;
} x, *a ;

main()
{
    x.b = 5 ;
    a = &x ;
    {
        struct foo temp0 ;
        temp0 = *a ;  /* S2 */
        temp0.b = 7 ; /* S3 */
    }
}
```

(c) The modified C program.



(d) The stack representation of the modified program.

Figure 2.5: An example to show the limitation in breaking down complicated data structures.

In the following, we represent some simple examples of valid conversions in SIMPLE (the left sample is a C language statement and the right one is the same statement as represented in SIMPLE).

Example 1:

```
x = a[5].b.c          ⇒     temp0 = &a[5]
                            x = (*temp0).b.c
```

Example 2:

```
x = a->b->c           ⇒     temp0 = (*a).b
                            x = (*temp0).c
```

Example 3:

```
x = a.b[i]            ⇒     temp0 = &a.b
                            x = temp0[5]
```

Example 4:

```
                                  temp0 = &a.b
                                  temp1 = &temp0[i]
x = a.b[i].c.d[2][j].e     ⇒      temp2 = &(*temp1).c.d
                                  temp3 = &temp2[2][j]
                                  x = (*temp3).e
```

Expression Nodes

Referring to the SIMPLE grammar shown in Figure 2.6, an expression is built of `rhs` when `rhs` is either `unary_expr` or `binary_expr`. We first consider `unary_expr` and then `binary_expr`. Among `unary_expr`, we are particularly interested in the $*$, `&` operators, and `call_expr` because of their effect in alias analysis. Some examples are given in the following to clarify the grammar for `unary_expr`.

Example 5 (on $*$ operator):

```
a = **b               ⇒     temp0 = *b
                            a = *temp0
```

Example 6 (on $*$ operator):

```
a = *(b.c)            ⇒     temp0 = b.c
                            a = *temp0
```

The address operator is also one of the cases that can not always be simplified (because the meaning of the expression might change). Therefore, a complex variable name may appear after '&' operator. As an example, `&(a.b)` or `&(*a.b)` must remain unchanged.

Example 7 (on `&` operator):

```
a = &(b->c->d)        ⇒     temp0 = (*b).c
                            a = &((*temp0).d)
```

```
expr :  modify_expr
     | rhs

modify_expr : varname '=' rhs
        | '*' ID '=' rhs

rhs : unary_expr
    | binary_expr

unary_expr : simp_expr
           | '*' ID
           | '&' varname
           | call_expr
           | unop val
           | '(' cast ')' varname
/* 'cast' stands for valid C typecasts */

simp_expr : varname | CONST

call_expr : ID '(' arglist ')'
```

```
arglist : arglist ',' val | val

unop : '+' | '-' | '!' | '~'

binary_expr : val binop val

binop :  relop
      | '-' | '+' | '/' | '*'
      | '%' | '&' | '|' | '<<'
      | '>>' | '^'

relop : '<' | '<=' | '>' | '>='
      | '==' | '!='

val : ID | CONST

varname : arrayref /*ARRAY_REF*/
        | compref  /*COMPONENT_REF*/
        | ID
```

Figure 2.6: SIMPLE grammar for expressions.

The actual parameters of a function call (call expression) are simplified to simple variables (ID) or constants. Some examples of CALL_EXPR are given in the following:

Example 8 (on function call):

```
f( 3, &a, *b)          ⇒       temp0 = &a
                               temp1 = *b
                               f( 3, temp0, temp1)
```

Example 9 (on function call):

```
f( a.b, c[7], "abc")   ⇒       temp0 = a.b
                               temp1 = c[7]
                               temp2 = "abc"
                               f( temp0, temp1, temp2)
```

Note that since string constant `"abc"` is considered as an array, it is replaced by a temporary variable.

As it is shown in the grammar represented in Figure 2.6, the binary expressions (`binary_expr`) are in 3-address code format. This is shown in the following example:

Example 10 (on 3-address code format):

```
a = (*b) + c / d        ⇒        temp0 = *b
                                 temp1 = c/d
                                 a = temp0 + temp1
```

One of the objectives of SIMPLE is to produce a clear definition for each expression or statement. Therefore, SIMPLE transforms the conditional, compound and logical expressions into a series of simpler statements. Following are some examples to show this transformation.

Example 11:

```
(a>b)?  a : b = c       ⇒        if ( a > b)
                                     a = c
                                 else
                                     b = c
```

Example 12:

```
a = b = c               ⇒        b = c
                                 a = b
```

Example 13:

```
a = b && c + d          ⇒        temp0 = (b != 0);
                                 if (temp0)
                                     temp0 = (c != 0);

                                 a = temp0 + d ;
```

Statement Nodes

In the following, we study different types of statements in SIMPLE. These are basically divided into two major groups:

- Basic statements: are the expressions followed by ';' (e.g. modify expressions and function calls).
- Compositional control statement: are all the conditional statements (e.g. `if` statement), and loop statements (e.g. `while` statement).

Basic Statements:

Different types of basic statements in a C-program can be replaced by one or more of the 15 basic statements given in Figure 2.7. The node related to basic statements in SIMPLE is EXPR_STMT.

```
 1.   x  =  a binop b   where binop is any binary operation
 2.  *p  =  a binop b
 3.   x  =  unop a      where unop is any unary operation
 4.  *p  =  unop a
 5.   x  =  y
 6.  *p  =  y
 7.   x  =  f(args)     where args is a possibly empty list of arguments
 8.  *p  =  f(args)
 9.   x  =  (cast)b     where cast is any typecast
10.  *p  =  (cast)b
11.   x  =  &y
12.  *p  =  &y
13.   x  =  *q
14.  *p  =  *q
15.  f(args)
```

Figure 2.7: List of the 15 basic statements. Variables `x` and `y` denote `varname`. Variables `a`, `b`, and `c` denote `val`. Variables `p` and `q` denote `ID`.

Compositional Control Statements:

In SIMPLE, compositional control statements consist of: the `if` statement (IF_STMT), `for` loop (FOR_STMT), `while` loop (WHILE_STMT), `do` loop (DO_STMT), `switch` (SWITCH_STMT), `case` (CASE_STMT), `default` (DEFAULT_STMT), `break` (BREAK_STMT), `continue` (CONTINUE_STMT), and `return` (RETURN_STMT) statements. The flexibility of C-language allows the programmers to write the compositional control statements in a complicated manner. SIMPLE changes their syntax to a restricted format. In the following, we explain each of the above cases:

<u>`if` statement:</u> The grammar of `if` statement is as follows:

stmt: IF '(' condexpr ')' stmt
| IF '(' condexpr ')' stmt ELSE stmt

The graph representation of `if` statement in SIMPLE AST is shown in Figure 2.8 (the else-body will be NULL if the else part does not exist). Clearly the 'condexpr' is already being simplified. Following is an example of `if` statement:

Example 14:

```
                                          b = c + d
if ( a = b = c + d, d > a)                a = b
{...}                        ⇒            if ( d > a)
                                          {...}
```

<u>`for` loop statement:</u> The grammar of a `for` loop is as follows:

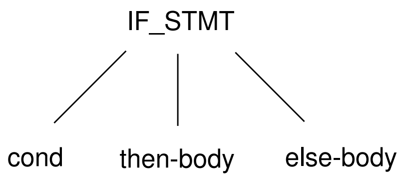


Figure 2.8: The SIMPLE AST for IF_STMT.

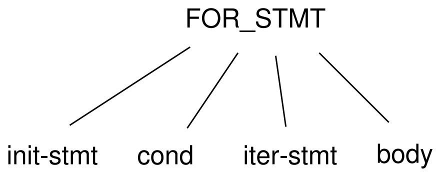


Figure 2.9: The SIMPLE AST for FOR_STMT.

FOR '(' expr ';' condexpr ';' expr ')' stmt

The graph representation of `for` statement in SIMPLE AST is shown in Figure 2.9. Only one initial statement and iteration statement is allowed. If more than one is used, it is moved outside of the loop. This is shown in the following example:

Example 15:

```
for (i=1,j=1 ; j<10 ; i++,j++)
      {
          stmts ;
      }
```

$\Rightarrow$

```
i = 1
for (j=1 ; j<10 ; j=j+1)
{
    stmts ;
    i = i + 1 ;
}
```

If there is more than one iteration statement, in order to get the correct answer, the iteration statement(s) (in this case `i=i+1`) should be moved to the end of the for loop.

The conditional expression should be also in 3-address code format. Following example is devised to show this fact.

Example 16:

```
for (i=1,j=1 ; i+j<k ; i++,j++)        i = 1
{                                      j = 1
    stmts ;                            for (temp0=i+j ; temp0<k ;
}                                           temp0=i+j)
                                 ⇒     {
                                           stmts ;
                                           i = i + 1 ;
                                           j = j + 1 ;
                                       }
```

For all loop constructs, the `continue` statement indicates that flow of control moves to the beginning of the loop. Therefore, the newly added statement(s) to the end of the loop will not be executed. To solve this problem, the newly added statement(s) should be also added before each `continue` statement. An example of this case is given in the following:

Example 17:

```
for (i=1 ; i<10 ; j++,i++)           for (i=1 ; i<10 ; i=i+1)
{                                    {
    if (cond)                            if (cond)
        continue                         {
    stmts ;                      ⇒           j = j + 1 ;
}                                            continue
                                         }
                                         stmts ;
                                         j = j + 1 ;
                                     }
```

The `break` statement moves the flow of control to the end of the loop. In this case, the iteration statement(s) is not executed and, therefore, nothing should be done for the `break` statement.

<u>`while` and `do-while` statements:</u> The grammar of `while` and `do` loop statements is as follows:

stmt: WHILE '(' condexpr ')' stmt
stmt: DO stmt WHILE '(' condexpr ')'

The graph representation of `while` and `do-while` statements are shown in Figure 2.10. As `while` and `do-while` statements are similar to `for` statement, we do not give any further examples for them.

Figure 2.10: The SIMPLE AST for WHILE_STMT and DO_STMT.

<u>`switch` and `case` statements:</u> These statements can be more complicated than other ones. This happens because the body of different `case` statements can be partially shared. This results in the following two problems: (i) the statement sequence may be labeled with more than one case label, (ii) the flow of control into and out of the `switch` statement may not be regular (for example, one may have a case label in the middle of statement sequence). In SIMPLE, each statement sequence begins with one more case labels and ends with either a `break`, `continue`, or `return` statement. Readers are referred to Appendix A for the corresponding grammar. An example of simplifying a `switch` statement is given in example 18 of Figure 2.11.

Example 18:

```
switch(a)                          switch(a)
{                                  {
    case 1:                            case 1:
    case 2:                            case 2:
        stmt1 ;                            stmt1 ;
    case 3:                                stmt2 ;
        stmt2 ;                            break ;
        break ;          ⇒             case 3:
    default:                               stmt2 ;
        stmt3 ;                            break ;
        break ;                        case 4:
    case 4:                                stmt4 ;
        stmt4 ;                            break ;
}                                      default:
                                           stmt3 ;
                                           break ;
                                   }
```

Figure 2.11: An example of `switch` statement in SIMPLE.

Auxiliary Nodes

Among the auxiliary nodes, we explain the TREE_LIST. This node is used to create lists of different SIMPLE statements. An example is shown in Figure 2.12. Statements are linked through TREE_CHAIN field while the TREE_VALUE field points to each statement.

```
stmt1 ;
stmt2 ;
stmt3 ;
```

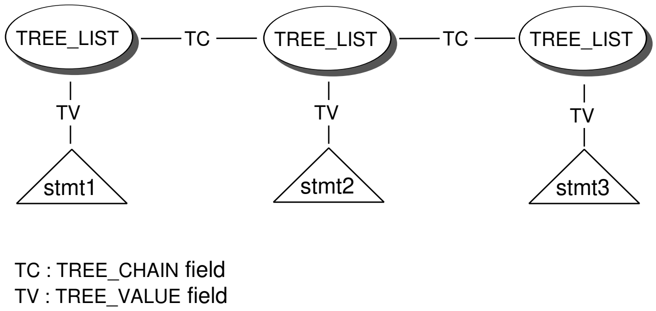


Figure 2.12: An example for the connection between the statements.

A new scope is defined by another TREE_LIST node. If it has some variables, the TREE_PURPOSE field points to the list of variables in that scope. Otherwise, this field would be NULL. TREE_VALUE field points to the list of statements in the scope. Figure 2.13 shows this fact (the newly added parts are shown in bold face).

```
{
    int var1,
        var2 ;

    stmt1 ;
    stmt2 ;
    stmt3 ;
}
```

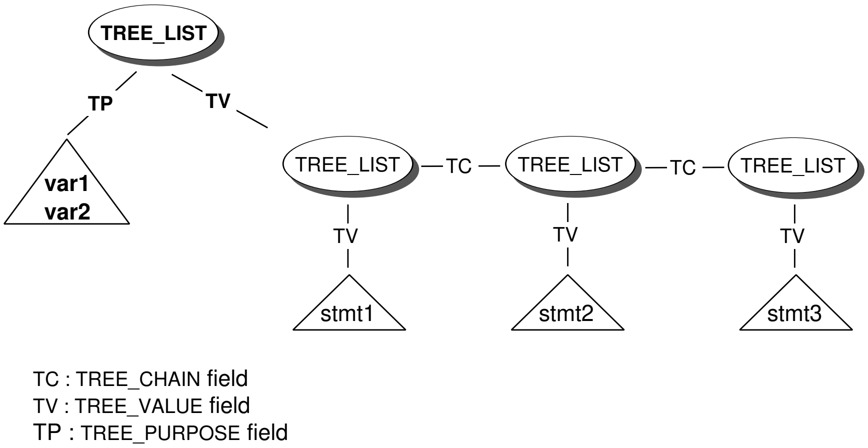


Figure 2.13: An example for the connection between statements and variables.

This construction is done recursively for all statements and functions. It includes the scope appearing in the body of loops and conditions. Consider the example shown in Figure 2.14. The body of the `while` statement is defined in a new scope. Therefore, a new TREE_LIST node is defined for this scope (shown in bold face).

```
while (cond)
{
    stmt1 ;
    stmt2 ;
}
```

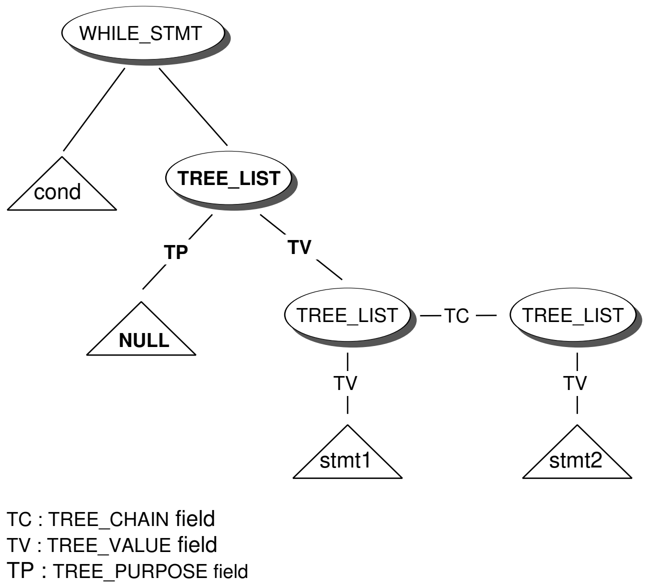


Figure 2.14: An example for the representation of scope in a `while` statement.

The next step is to provide a connection between functions and global variables. A tree node called AST_DECL_NODE is used for this purpose. Each AST_DECL_NODE is related to a function or global variable, its AST_DECL_PTR field points to the function declaration or global variable declaration, its AST_DECL_BODY points to the body of the function in case of function, or to NULL in case of global variable, and its TREE_CHAIN node points to the next AST_DECL_NODE. The example shown in Figure 2.15 clarifies this connection. By having the head of AST_DECL_NODE, one has access to the whole SIMPLE tree. The head of the tree in the example of Figure 2.15 is called 'decl_list'.

```
int   var3
float var4 ;

f()
{ f_body }

char  var5 ;

main() {
    int var1,
        var2 ;

    stmt1 ;
    stmt2 ;
}
```

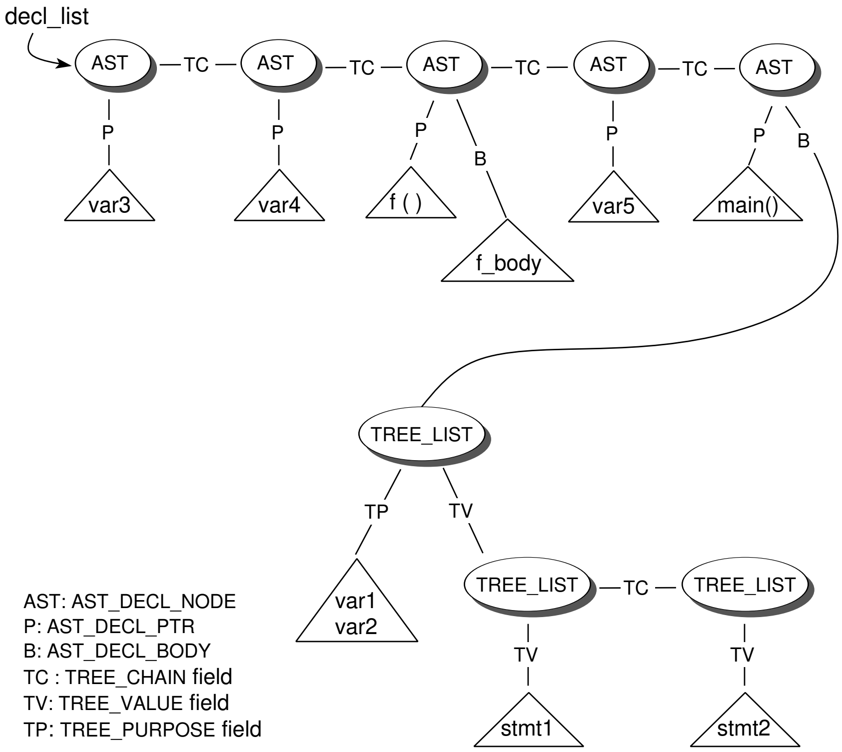


Figure 2.15: An example for the connection between functions and global variables.