**Design and Development of Compiler**

**For C- Language**

**(Phase 4: Design and Implementation of Code Generator)**

1. **각 단계별 제안서**
2. **각 단계별 결과 보고서**

**(Phase 4: Design and Implementation of Code Generator)**

**과목명: [CSE4120] 기초 컴파일러 구성**

**담당교수: 서강대학교 컴퓨터공학과 정 성 원**

**개발자: 6조 정지원(팀장, 20081638)**

**박경민(20121590)**

**개발기간: 2017. 6. 05 ~ 2017. 6. 28**

**Project3 결과 보 고 서**

**프로젝트 제목: Design and Development of Compiler for C-Language:**

**Phase 4: Design and Implementation of Code Generator**

**제출일: 2017. 6. 28**

**개발자: 정지원(팀장, 20081638), 박경민(20121590)**

1. **개발 목표**

본 프로젝트의 목표는 Compile 과정의 4번째 과정인 Code Generator를 구현하는 것입니다. Code Generator는 다음을 수행합니다. 첫째, 전역 변수에 대한 공간 할당 코드를 생성합니다. 둘째, 각 함수에 대한 코드를 생성합니다.

1. **개발 범위 및 내용**
2. **개발 범위**

Code Generator를 구현하기 위하여, 다음을 개발하여야 한다.

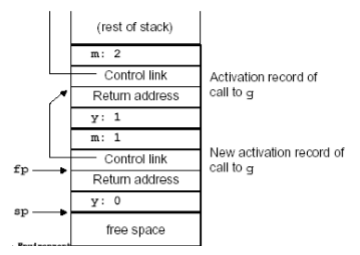
1. Phase 3(Building Symbol Table)에 Location 계산, 각 함수 별 local variable을 위한 크기 계산, 전역 변수 리스트 생성, 전역 변수 flag 설정 기능을 추가
2. 함수의 Runtime Stack Environment 구조 결정 및 함수 호출 시 해당 구조 생성
3. Static Scoping Rule을 고려한 Statement와 Expression에 해당하는 코드 생성
4. Input, Output 기능 구현
5. **개발 내용**
6. Phase 3(Building Symbol Table)에 Location 계산, 각 함수 별 local variable을 위한 크기 계산, 전역 변수 리스트 생성, 전역 변수 flag 설정 기능을 추가

Project3에서는 Location값을 변수의 선언 순서를 저장하였는데, Project4에서는 변수에 접근하기 위하여 (FP 레지스터 값 + Location)의 메모리주소에 변수에 대한 공간을 할당하고, 해당 공간에 값을 store, load함으로써 변수에 대한 접근을 처리한다. 이를 위해서 Location값을 “FP에서 변수 공간까지의 거리”의 정의에 맞게 계산하여야 한다.

또한, Run Time 시에 함수가 호출되면 Compile 시 계산된 각 함수에 정의되어 있는 지역변수를 위한 공간 크기만큼 메모리 공간을 할당하여야 한다. 이를 위해서 Symbol Table을 생성할 때 각 함수 정의 노드에 각 함수에서 필요한 지역변수 공간을 계산하여 저장하여야 한다.

Project 3에서 Abstract Syntax Tree의 모든 Identifier 노드에 각 Identifier가 정의된 변수 선언문의 노드 위치를 저장하였다. 이때 각 Identifier가 전역 변수인지, 지역 변수인지를 확인할 필요가 있다. 왜냐하면, 전역 변수의 경우에는 전역 공간 시작 주소를 가지는 GP 레지스터값에서 Location 만큼 떨어진 공간을 접근해야 하고, 지역 변수의 경우 FP 레지스터값에서 Location 만큼 떨어진 공간을 접근해야 한다. 이를 위해 Phase 3에서 Symbol Table을 만들 때, 각 선언문에 전역/지역 변수를 표시하는 값을 저장해야 한다.

1. 함수의 Runtime Stack Environment 구조 결정 및 함수 호출 시 해당 구조 생성

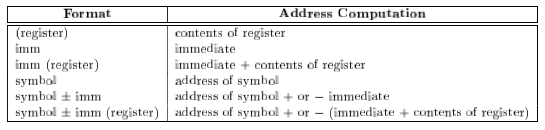


위의 그림은 강의 자료에 있는 Runtime Stack Environment을 설명하는 그림이다. C –는 함수 호출 기능을 지원하기 때문에, 위와 같이 Stack에 각 함수의 Activation record를 쌓아야 한다.

하지만, SPIM Machine에서는 10개의 temporary register($t0 ~ $t9)를 지원하는데, 이를 프로그램 수행 도중 발생하는 많은 temporary 값(예를 들어, a+b+c 를 계산하는데 a+b의 값을 임시 저장)을 저장하는데 사용할 수 있다. 하지만, 이와 같이 temporary register를 사용하면, 함수를 호출할 때 caller 에서 사용하던 temporary register의 값을 메모리에 임시 저장하고, 함수가 종료될 때 메모리에 저장되어 있던 temporary register 값을 register에 다시 대입하는 과정이 필요하다. 이를 지원하기 위해서, 이번 프로젝트에서는 다음과 같은 activation record를 설계, 구현하였다. 아래의 예시는 함수 g에서 함수 f를 호출하는 상황에서 stack이다.

|  |
| --- |
| (rest of stack) |
| g’s parameter |
| $t0 ~ $t9 value |
| control link |
| return address |
| g’s local variable |
| f’s parameter |
| $t0 ~ $t9 value |
| Control link |
| Return address |
| f’s local variable |
| (free space) |

1. Static Scoping Rule을 고려한 각 Statement와 Expression에 해당하는 코드 생성



위는 SPIM Machine에서 Static Scoping Rule을 지원하기 위해서 제공하는 Addressing Mode들이다. 이 중 imm (register)를 보면, 이는 immediate + contents of register로 계산되는데, 변수의 location + fp의 content가 지역변수, 파라미터의 메모리 주소이므로, 위 기능을 이용하면 접근이 용이하다.

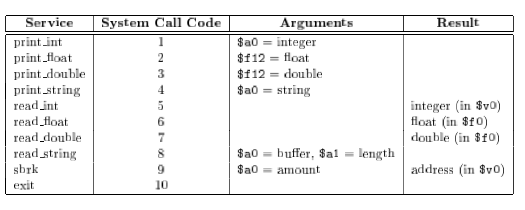
Project 4에서 사용하는 SPIM machine 명령어의 형태는 다음 중 하나이다.

* 명령어 reg1, immediate(reg2)
* 명령어 reg1
* 명령어 reg1, reg2
* 명령어 reg1, reg2, reg3
* 명령어 reg1, immediate
* 명령어 reg1, reg2, immediate
* 명령어 label
* 명령어 reg1, label

위의 형태의 명령어를 출력 파일에 출력하는 함수를 code.c와 code.h에 정의하여야 한다. 이를 cgen.c에서 abstract syntax tree를 순회하며 각 노드 종류에 적절한 명령어들을 출력할 때 사용하여야 한다.

1. Input, Output 기능 구현

Console에서 Integer를 입력 받는 Input 기능과 Console에 Integer를 출력 하는 Output기능을 구현하여야 한다. 이는 SPIM Machine이 제공하는 OS system call 기능을 이용하여야 한다.



위 표는 SPIM machine이 제공하는 system call 종류를 나타낸다. System call 기능을 이용하기 위해서는 $v0 레지스터에 원하는 system call code를 저장하고, 적절한 argument를 저장하고 syscall을 호출하면 된다.

1. **추진 일정 및 개발 방법**
2. **추진 일정**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 6/5 | 6/6 | 6/7 | 6/8 | 6/9 | 6/10 |
|  | 프로젝트 구상 | 프로젝트 구상 | 프로젝트 구상 | 프로젝트 구상 | 프로젝트 구상 | Phase3에 필요한 기능 추가 |
| 6/11 | 6/12 | 6/13 | 6/14 | 6/15 | 6/16 | 6/17 |
| Phase3에 필요한 기능 추가 | Phase3에 필요한 기능 추가 | Phase3에 필요한 기능 추가 | Phase3에 필요한 기능 추가 | 함수의 Runtime Stack Environment 구조 결정 및 함수 호출 시 해당 구조 생성 | 함수의 Runtime Stack Environment 구조 결정 및 함수 호출 시 해당 구조 생성 | Statement와 Expression에 해당하는 코드 생성 |
| 6/18 | 6/19 | 6/20 | 6/21 | 6/22 | 6/23 | 6/24 |
| Statement와 Expression에 해당하는 코드 생성 | Statement와 Expression에 해당하는 코드 생성 | Statement와 Expression에 해당하는 코드 생성 | Input, Output 기능 구현 | Input, Output 기능 구현 | Input, Output 기능 구현 | 테스트 및 수정 |
| 6/25 | 6/26 | 6/27 | 6/28 |  |  |  |
| 테스트 및 수정 | 결과 보고서 작성 | 결과 보고서 작성 | 결과 보고서 작성 |  |  |  |

1. **개발 방법**

먼저, TextBook의 Code의 분석을 통하여, 해당 코드의 부족한 부분이 무엇인지 확인한다. 수업 시간에 배운 내용과 Lecture Node, Textbook을 통하여 구현할 Code Generation의 이론을 확인한다.

1. **연구원 역할 분담**

정지원: Phase3에 기능 추가, code.c 구현

박경민: cgen.c 구현, input & output 구현

1. **연구 결과**
2. 합성 내용

* 전체 소프트웨어 구성도



* 각 부분의 역할 및 구현 방법

1. Main

역할: 사용자가 프로그램의 사용법에 맞게 사용하였는지 확인한다.

구현: 프로그램의 사용법인 “{프로그램 명} {C-Language 소스 코드}”에 맞게 프로그램을 실행하였는지, 실행 시 명시한 C-Language 소스 코드가 존재하는지 확인한 후, 파일 스트림을 열고 parse() 함수를 호출한다. 이를 통해, 소스 코드에 대한 abstract syntax tree를 생성한다. 그 후, buildSymtab\_pass1()을 이용하여 scope check를 수행하고 typeCheck()에서 사용할 정보(id와 해당 선언부 연결)를 생성한다. 그 후, typeCheck()를 이용하여 semantic error를 check한다. 그 후, Error가 없으면 spim machine code file을 생성하고 codeGen()에 abstract syntax tree를 전달하며 호출한다.

1. Code.c의 SPIM instruction 생성 함수

개발 내용 4)에서 설명한 SPIM instruction을 생성하기 위한 함수들이다.

* 1. emitComment

역할: 출력 파일에 comment를 출력한다

구현: SPIM machine에서는 #으로 시작하고 Enter로 끝나는 문장을 comment로 처리한다. emitComment는 string을 입력받아, #을 앞에 추가하여 출력 파일에 출력한다.

* 1. emitRO\_3

역할: 명령어 reg1, reg2, reg3 형태의 instruction을 출력한다.

구현: 명령어, register name 3개를 입력으로 받아, “명령어 reg1, reg2, reg3”형태로 출력 파일에 출력한다.

* 1. emitRO\_2

역할: 명령어 reg1, reg2 형태와 명령어 reg1, label 형태의 instruction을 출력한다.

구현: 명령어와 register name 2개 혹은 명령어와 register name, label name을 입력으로 받아 “명령어 reg1, reg2” 혹은 “명령어 reg1, label” 형태로 출력 파일에 출력한다.

* 1. cmitRO\_1

역할: 명령어 reg1 형태와 명령어 label 형태의 instruction을 출력한다.

구현: 명령어와 register name 혹은 명령어와 label name을 입력으로 받아서 “명령어 reg1” 혹은 “명령어 label” 형태로 출력 파일에 출력한다.

* 1. emitRI\_3

역할: 명령어 reg1, reg2, immediate 형태의 instruction을 출력한다.

구현: 명령어와 register name 2개, immediate integer를 입력으로 받아 “명령어 reg1, reg2, immediate” 형태로 출력 파일에 출력한다.

* 1. emitRI\_2

역할: 명령어 reg1, immediate 형태의 instruction을 출력한다.

구현: 명령어와 register name, immediate integer를 입력으로 받아 “명령어 reg1, immediate” 형태로 출력 파일에 출력한다.

* 1. emitRM

역할: 명령어 reg1, immediate(reg2) 형태의 instruction을 출력한다.

구현: 명령어와 register name 2개와 immediate integer를 입력으로 받아 “명령어 reg1, immediate(reg2)” 형태로 출력 파일에 출력한다.

* 1. emitLabel

역할: 라벨을 출력한다.

구현: label name을 입력으로 받아서, “label:” 형태로 출력 파일에 출력한다.

1. codeGen

역할: 출력 파일 설명 출력과 Data 영역 할당 코드를 출력한다.

구현: 출력 파일명과 “C—Compilation to SPIM code”라는 설명문을 주석의 형태로 출력 파일에 출력하고, makeDataArea함수를 호출하여 전역변수를 위한 공간을 할당하는 코드를 출력한다. 그 후, input과 output에 사용할 string을 Data영역에 할당하는 코드를 생성하고, cGen 함수를 호출하여 Text영역에 저장될 SPIM code 생성 및 출력을 시작한다.

1. cGen

역할: 노드의 종류에 따라 적절한 코드 출력 함수를 호출한다.

구현: ADT 노드의 종류에 따라 expression이면 genExp 함수를 호출하고, statement이면 genStmt 함수를 호출한다. 그 후, 노드의 sibling에 대해서 cGen 함수를 호출한다.

1. genExp

역할: expression의 종류에 따라 적절한 코드를 출력한다.

구현: 노드의 kind.exp를 확인하여, 어떤 expression의 노드인지 확인한다. 그 후, 각 종류에 따라 적절한 코드를 출력한다. 각 종류에 따라 다음과 같은 기능을 수행하는 코드를 생성하여야 한다.

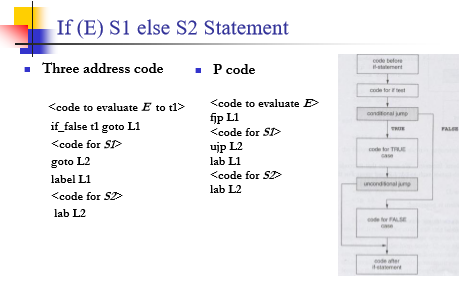
* 1. Constant: constant immediate value를 temporary register에 load한다.
  2. Identifier: identifier에 해당하는 변수의 memory address에서 temporary register로 content를 load한다.
  3. Operator: 각 operand에 대한 코드를 생성하고, 각 코드의 결과가 저장된 temporary register 2개를 operator 종류에 따라 계산하여 결과를 temporary register에 저장하는 코드를 생성한다.
  4. Assignment: R-value에 대한 코드를 생성하고, 해당 결과를 l-value의 주소를 계산하고 해당 주소의 memory에 store하는 코드를 생성한다.
  5. Array accessing: array[expression]의 형태라 할 때, expression의 결과를 계산하는 코드를 출력하고, 해당 array element의 주소인 “array의 주소 + WORDSIZE \* expression 결과”의 내용을 temporary register에 load하는 코드를 출력한다.
  6. Function call: 함수 호출은 다음의 기능을 수행하는 코드를 출력하여야 한다.
     1. Argument를 계산하고, 해당 결과를 stack에 push 한다.
     2. Temporary register를 stack에 push한다.
     3. 함수 body로 jump한다.
     4. 함수에서 return되면 저장한 temporary register를 stack에서 pop한다.
     5. Argument를 저장한 stack 공간을 제거한다.
     6. 함수의 반환 값을 temporary register에 저장한다.
  7. Input: read integer system call을 호출하고, console에서 사용자로부터 입력 받은 integer를 temporary register에 저장하는 코드를 출력한다.
  8. Output: print integer system call을 호출한다.

1. genStmt

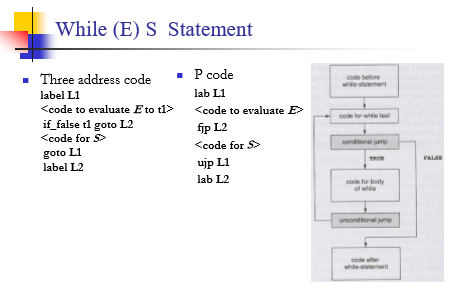
역할: statement의 종류에 따라 적절한 코드를 출력한다.

구현: 노드의 kind.stmt를 확인하여, 어떤 expression의 노드인지 확인한다. 그 후, 각 종류에 따라 적절한 코드를 출력한다. 각 종류에 따라 다음과 같은 기능을 수행하는 코드를 생성하여야 한다.

* 1. Function Declaration: 함수 선언은 다음과 같은 기능을 수행하는 코드를 출력하여야 한다.
     1. 함수 이름에 해당하는 label을 출력한다
     2. FP와 RA register를 stack에 push한다.
     3. FP를 위에서 RA register를 저장한 주소 바로 위를 가리키게 한다.
     4. 지역 변수를 위한 공간을 할당한다.
     5. 함수 body를 수행한다.
     6. 함수 종료 sequence를 가리키는 label을 출력한다.
     7. 지역 변수 공간을 제거한다.
     8. 저장한 FP와 RA를 stack에서 pop한다.
     9. 해당 함수를 호출한 instruction 다음으로 jump한다.
  2. Compound Statement: variable declaration 부분을 제외한 statement에 대한 코드를 출력한다.
  3. Selection Statement: 다음과 같은 형태의 코드를 출력한다.



* 1. Iteration Statement: 다음과 같은 형태의 코드를 출력한다.



* 1. Return Statement: “return expression;”의 형태라 할 경우 expression를 계산하는 코드를 출력한다. 그 후, $v0 레지스터에 해당 결과를 저장하고, 해당 함수의 종료 sequence를 가리키는 label로 jump하는 코드를 생성한다.

1. getLocation

역할: 선언문의 node를 입력으로 받아, 해당 선언문의 변수의 location을 반환한다.

구현: 선언문의 종류를 확인하여, parameter일 경우에는 location값에 temporary register를 저장하는 공간의 크기 40byte를 더하여 반환하고, 전역/지역 변수일 경우에는 location 값을 반환한다.

1. temporary Register 관리 함수

SPIM에서 사용 가능한 temporary register $t0 ~ $t9을 관리하는 함수들이다. 이는 temporary register가 필요할 때마다 0->9순으로 저장하고, 더 이상 필요하지 않으면 다른 값을 저장하기 위해서 아래와 같은 함수를 정의하였다. tempReg라는 변수를 공유하는데, 이는 사용되지 않은 tempReg 중 가장 작은 idx를 저장하고 있다.

* 1. nextTempReg

역할: temporary register 하나를 사용한다고 표시한다.

* 1. currentTempReg

역할: 현재 사용중인 가장 높은 temporary register가 무엇인지 반환한다.

* 1. removeTempReg

역할: 사용중인 temporary register에서 높은 register부터 입력 받은 개수만큼 사용하지 않는다고 표시한다.

* 1. removeAllTempReg

역할: 모든 temporary register를 사용하지 않는다고 표시한다.

1. 함수 호출 관련 함수

함수 호출(Function Call Expression)을 구현하는데 필요한 함수를 정의하였다.

* 1. argumentPush

역할: argument를 stack에 push한다.

구현: argument를 저장하는데 필요한 만큼 stack에 공간을 할당하고, 해당 공간에 argument를 store하는 코드를 출력한다.

* 1. save\_FP\_RA, retrieve\_FP\_RA

역할: FP와 RA 레지스터를 stack에 push, pop한다.

구현: FP와 RA 레지스터를 위한 공간을 할당하고, 해당 공간에 FP와 RA의 값을 store하는 코드를 출력한다. Retrive의 경우에는 반대로, FP와 RA의 값을 memory에서 load하고 해당 공간을 삭제하는 코드를 출력한다.

* 1. saveTempReg, retriveTempReg

역할: $t0 ~ $t9을 stack에 push, pop한다.

구현: $t0 ~ $t9을 위한 40byte를 stack에 할당하고, 해당 공간에 $t0 ~ $t9의 값을 store하는 코드를 출력한다. Retrieve의 경우에는 반대로 해당 공간에서 $t0 ~ $t9로 load하고, 40byte 공간을 stack에서 삭제하는 코드를 출력한다.

1. getLabel

역할: 새로운 label을 생성한다.

구현: \_label\_0부터 \_label\_1, \_label\_2 순으로 함수가 호출될 때마다 해당 label name string을 반환한다.

* 사용자 매뉴얼

Usage: ./program\_name <file\_name>

Ex) ./20121590 test1.tny

프로그램의 사용법은 C-Language로 작성된 소스 코드의 파일명을 <file\_name>자리에 입력하고, 프로그램의 이름을 program\_name자리에 입력하고, Enter를 누릅니다.

1. 분석 내용
2. Main

Main함수의 input인 argc의 값이 2가 아닐 경우, argv[1]의 파일명의 파일이 존재하지 않을 경우를 각각 비교연산자와 fopen함수의 반환값(파일이 존재하지 않을 경우 NULL을 반환)으로 확인한다. 그리고 parse 함수를 호출하여, abstract syntax tree를 생성한다. 이를 매개변수로 하여 buildSymtab\_pass1을 호출하여 pass1을 수행하고, typeCheck를 호출하여 pass2를 수행한다. 그 후, Error 변수를 확인하여, 0일 경우 spim code 출력 파일을 생성하고 codeGen에 adt의 노드와 spim code 출력 파일명을 매개변수로 전달하며 호출한다. 이를 통해, code generation을 수행한다.

1. Code.c의 SPIM instruction 생성 함수
   1. emitComment

char\* c에 출력하고 하는 comment를 입력으로 받는다. 이를 fprintf를 이용하여 “#c”의 형태로 파일에 출력한다.

* 1. emitRO\_3

입력: op(명령어 string), r(reg1 string), s(reg2 string), t(reg3 string), c(comment)

Fprintf를 이용하여 “op r, s, t #c”의 형태로 파일에 출력한다.

* 1. emitRO\_2

입력: op(명령어 string), r(reg1 string), s(reg2 string), c(comment)

Fprintf를 이용하여 “op r, s #c”의 형태로 파일에 출력한다.

* 1. cmitRO\_1

입력: op(명령어 string), r(reg1 string), c(comment)

Fprintf를 이용하여 “op r #c”의 형태로 파일에 출력한다

* 1. emitRI\_3

입력: op(명령어 string), r(reg1 string), s(reg2 string), t(immediate integer), c(comment)

Fprintf를 이용하여 “op r, s, t #c”의 형태로 파일에 출력한다.

* 1. emitRI\_2

입력: op(명령어 string), r(reg1 string), s(immediate integer), c(comment)

Fprintf를 이용하여 “op r, s #c”의 형태로 파일에 출력한다.

* 1. emitRM

입력: op(명령어 string), r(reg1 string), d(immediate integer), s(reg2 string), c(comment)

Fprintf를 이용하여 “op r, d(s) #c”의 형태로 파일에 출력한다.

* 1. emitLabel

출력하고자 하는 라벨을 label에 입력으로 받아서, fprintf를 이용하여 “label:”의 형태로 파일에 출력한다.

1. codeGen

emitComment함수를 이용하여 “File: 파일명”과 “C—Compilation to SPIM Code”라는 해당 spim 코드 설명을 comment의 형태로 출력한다. 그 후, “.data”를 출력하여 data영역임을 표시하고, makeDataArea를 호출하여 전역 변수에 대한 공간을 “.space 전역변수전체크기”의 형태로 파일에 출력함으로써 할당한다. 그 후, input, output에 사용할 string 3개를 data 영역에 할당하는 코드를 출력한다.

그 후, “.text”를 출력하여 text영역임을 표시하고, cGen()을 호출하여 spim 코드를 생성, 출력하기 시작한다.

1. cGen

tree->nodekind를 확인하여 StmtK이면 genStmt를 호출하고, ExpK이면 genExp를 호출한다. 그 후, tree->sibling에 대해서 cGen을 호출한다.

1. genExp

tree->kind.exp를 확인하여 해당 노드가 어떤 expression인지 확인한다. 그 후, 각 expression 종류에 따라 다음과 같은 기능을 수행한다.

* 1. Constant

nextTempReg를 호출하여 temporary register 하나를 할당받아 tempRegNum에 저장한다. 그 후, emitRI\_2를 호출하여 “li Rtmp constant”를 출력하게 하여, constant 값이 temporary register에 load되도록 한다.

* 1. Identifier

nextTempReg를 호출하여 temporary register 하나를 할당받아 tempRegNum에 저장한다. 그 후, getLocation을 호출하여 해당 변수의 location을 구한다. 그 후, 전역변수 flag를 확인하여 전역변수이면 “lw Rtmp location($gp)”, 지역변수이면 “lw Rtmp location($fp)”를 emitRM 함수를 호출하여 파일에 출력한다. 이를 통해 메모리 내의 변수 값을 temporary register에 load되도록 한다.

* 1. Operator

cGen을 노드의 child[0]과 child[1]에 대해 각각 호출하여 operand 값을 계산한다. 이를 통해, 사용 중인 가장 높은 temporary register 2개에 operand 계산 값이 들어간다. 그 후, tree->attr.op를 확인하여 각 operator 종류를 확인하고, 각 종류에 받는 명령어를 op라고 할 때, emitRO\_3를 이용하여 “op Rtmp1, Rtmp1, Rtmp2”의 형태로 파일에 출력되도록 한다. 이를 통해 operator의 결과가 Rtmp1에 저장되게 한다. 그 후, removeTempReg(1)을 호출하여 Rtmp2를 사용하지 않는다고 표시한다.

* 1. Assignment

cGen을 노드의 child[1]에 대하여 호출하여 r-value 값을 계산하는 코드를 출력한다. 이를 통해서, 계산된 r-value값은 가장 높은 temporary register에 저장되게 된다. 그 후, l-value(child[0])가 variable일 경우에는 “sw Rtemp, location($gp 혹은 $sp)”을 emitRM을 호출함으로써 파일에 출력한다. 만약, l-value가 array일 경우에는 하나의 temporary register를 nextTempReg를 호출함으로써 할당 받아, ($gp 혹은 $sp) + location + expression value \* WORDSIZE를 계산하는 코드를 생성하고, emitRM을 호출하여 “sw Rtemp(r-value), 0(Rtemp(l-value 주소))”의 형태의 코드를 출력하게 한다.

* 1. Array accessing

cGen을 노드의 child[1]에 대하여 호출하여 expression의 값을 계산하는 코드를 출력한다. 이를 통해서, 계산된 expression의 값은 가장 높은 temporary register에 저장된다. 그 후, 2개의 temporary register를 할당 받아서 “($gp 혹은 $fp) + location + expression value \* WORDSIZE”를 계산하는 코드를 출력하고, emitRM을 호출하여 “lw Rtemp, 0(Rtemp(array element 주소))”의 형태의 코드를 출력한다. 이를 통해서 array element의 값이 temporary register에 저장되게 된다.

* 1. Function call: 함수 호출은 다음의 기능을 수행하는 코드를 출력하여야 한다.
     1. Argument를 계산하고, 해당 결과를 stack에 push 한다.
        + - argumentPush함수를 호출한다.
     2. Temporary register를 stack에 push한다.
        + - saveTempReg함수를 호출한다.
     3. 함수 body로 jump한다.
        + - emitRO\_1을 호출하여 “jal 함수명”의 코드를 출력한다.
     4. 함수에서 return되면 저장한 temporary register를 stack에서 pop한다.
        + - retriveTempReg함수를 호출한다.
     5. Argument를 저장한 stack 공간을 제거한다.
        + - emitRI\_3을 호출하여 “addi $sp, $sp, argument\_size”의 코드를 출력하게 한다.
     6. 함수의 반환 값을 temporary register에 저장한다.
        + - emitRO\_2를 호출하여 “move Rtemp(결과), $v0”의 코드를 출력하게 한다.
  2. Input: read integer system call을 호출하고, console에서 사용자로부터 입력 받은 integer를 temporary register에 저장하는 코드를 출력한다.
  3. Output: print integer system call을 호출한다.

1. genStmt
2. getLocation
3. temporary Register 관리 함수
   1. nextTempReg
   2. currentTempReg
   3. removeTempReg
   4. removeAllTempReg
4. 함수 호출 관련 함수
   1. argumentPush
   2. save\_FP\_RA, retrieve\_FP\_RA
   3. saveTempReg, retriveTempReg
5. getLabel
6. 제작 내용



Figure 수행 command

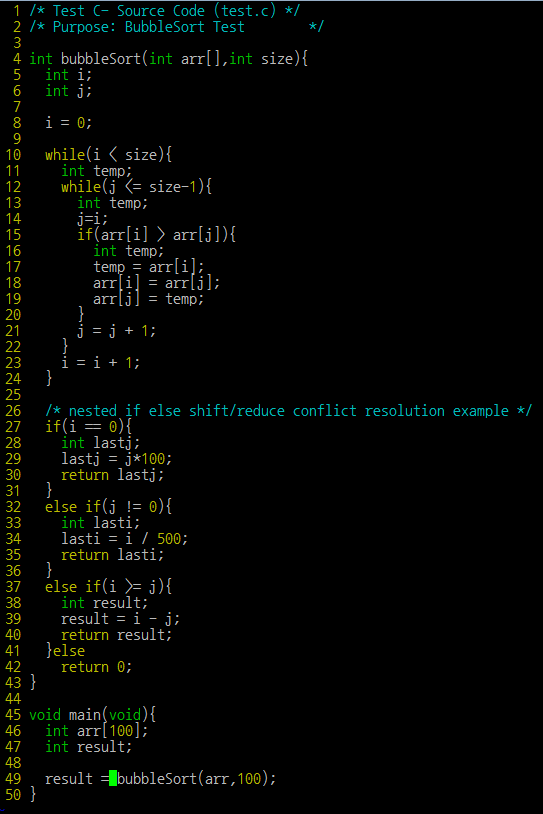


Figure C—language test code(test.c)

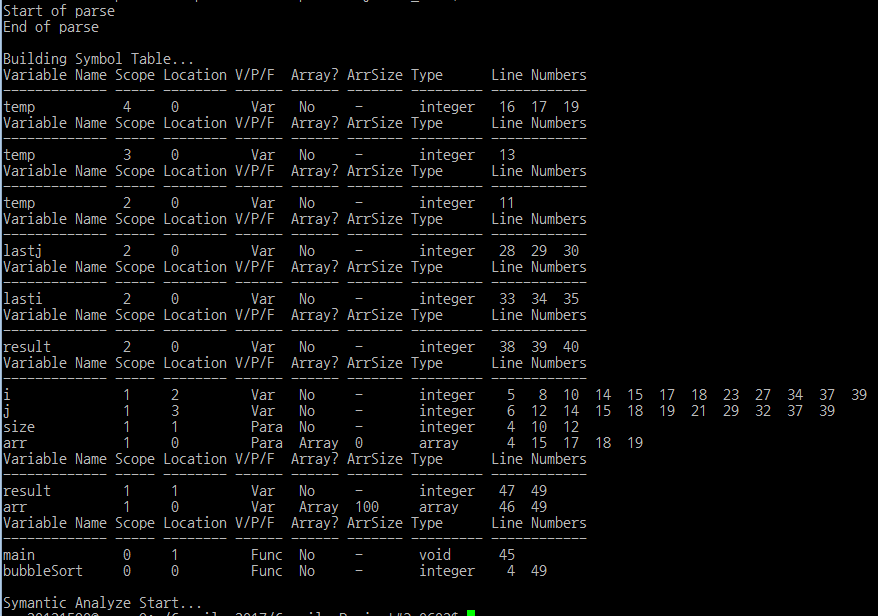


Figure 수행 결과

* 결과 분석

소스코드에 의해 생성된 Symbol Table을 보면, 각 Scope에 대해서 적절한 처리가 되어 있음을 알 수 있다. 또한, nested된 scope에서 temp를 계속 선언하였음에도, 에러 없이 잘 처리 되었고, 또한 가장 안쪽 scope의 temp로 접근하는 것 또한 잘 처리되었음을 scope 4의 temp에 line number가 증가하는 것을 보고 알 수 있다.

1. 시험 내용

- 평가 기준 #1 – 반복된 수행 시 동일한 결과 출력

위의 테스트 파일에 대한 프로그램의 수행을 10번 반복한 결과 매번 동일한 결과를 출력하였다.

- 평가 기준 #2 – PASS1에서 오류 검출 시 PASS2 미 수행

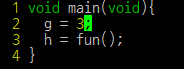


Figure Semantic Error가 있는 C—Code

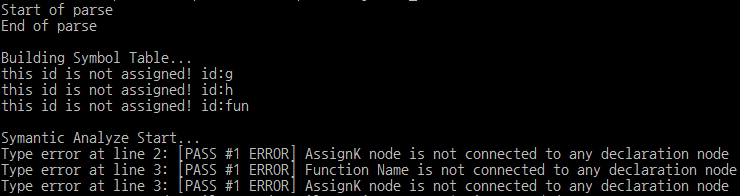


Figure 변경 전 수행 결과

PASS1에서 Scope Error나 Semantic Error가 발견되었을 경우 PASS2가 수행되면, PASS2의 결과가 PASS1의 에러 때문인지, 혹은 다른 이유의 Error때문인지 혼란을 가질 수 있다. 이를 방지하기 위해서 PASS1에서 Error가 발생할 경우 main의 Error Flag를 TRUE로 설정하고, 이를 PASS2를 수행하기 위한 조건으로 설정하였다.

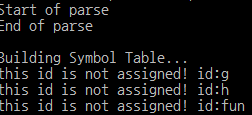


Figure 변경 후 수행 결과

1. 평가 내용

- 안정성

프로그램 실행 시 사용자의 적절한 입력인지를 검사하여 잘못된 입력일 경우 사용법(Usage)를 출력하고 프로그램을 종료한다. 또한, semantic error가 발생할 경우 적절한 오류 메시지를 출력하여, 사용자가 자신의 소스 코드를 고칠 수 있도록 하여, 안정성을 향상시킨다.

- 신뢰성

동일한 입력 파일에 대한 반복수행에도 동일한 결과를 출력한다. 이를 통해 프로그램의 신뢰성이 높음을 알 수 있다.

V. 기타

가. 자체 평가

프로젝트를 진행함에 있어, 제안서에서 기술하였던 프로젝트 추진 일정에 맞게 프로젝트를 단계적으로 진행하였습니다. 또한, textbook에서 제공하는 frame code를 분석하여, 이번 프로젝트에 필요한 부분을 제외한 코드를 제외하여, 간결한 프로젝트 코드를 완성하였습니다.

나. 느낀 점

수업 시간에 배운 Semantic Analyzer를 직접 구현해 봄으로써 컴파일러의 수행 원리를 좀 더 잘 이해할 수 있게 되었습니다.