Design and Development of Compiler for C- Language

4. Design and Implementation of Code Generator

과목명 : [CSE4120] 기초 컴파일러 구성

담당교수 : 서강대학교 컴퓨터공학과 정성원

개발자 : 17 조 엄태경 (20120085)

개발기간 : 2019. 5. 28. ~ 2019. 6. 25.

현 개발 단계 결과 보고서[1]

프로젝트 제목: Design and Development of Compiler for C-Language:
Phase 4: Design and Implementation of Code Generator
제출일: 2019. 6. 25.
개발자: 17 조 엄태경(20120085)

I. 개발 목표

본 프로젝트는 C-minus 프로그래밍 언어를 처리할 수 있는 컴파일러 구현의 네 번째 단계로, 코드 생성 기능을 구현한다. 이 단계에서 SPIM 시뮬레이터에서 작동하는 MIPS 어셈블리 코드가 구현된다.

II. 개발 내용

본 프로젝트에서는 앞서 구현된 추상문법트리를 이용해 SPIM 시뮬레이터에서 작동하는 MIPS 어셈블리 코드를 생성한다. 이에 앞서, 타겟인 SPIM 시뮬레이터의 아키텍처에 적합하도록 추상문법트리와 심볼 테이블을 수정하는 작업이 필요하다. 또한, C-minus 프로그래밍 언어의 스펙에 정의된 입출력 내장 함수를 구현하여 C-minus 프로그램에서 정상적으로 이용할 수 있어야 한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

5 월 28 일: 과제 명세서 수령 및 요구 사항 분석
5 월 29-6 월 9 일: 기존 구현물 보완 및 구현 계획 발표 준비
6 월 10 일-21 일: 코드 생성 기능 구현
6 월 22 일-23 일: 테스트 케이스 작성 및 디버깅
6 월 24 일-25 일: 보고서 작성

나. 개발 방법

1. 개발 환경

로컬 환경에서 개발과 테스트를 마친 뒤 학교에서 제공되는 cspro 서버에 파일을 업로드해 정상 작동을 다시 확인한다. 소프트웨어 버전에 다소 차이가 있으나, 각 프로그램의 업데이트 로그 등을 통해 버전에 따른 기능 차이가 본 프로젝트 수행에 영향을 주지 않음을 확인하고 진행한다.

로컬 환경: Ubuntu 18.0.4, gcc 7.4.0, flex 2.6.4, bison 3.0.4, SPIM 8.0, QtSpim 9.1.20
서버 환경: Ubuntu 16.0.4, gcc 5.4.0, flex 2.6.0, bison 3.0.4, SPIM 8.0

1 본 프로젝트의 구현물, 테스트용 C-Minus 소스코드와 보고서는 github 에 비공개 저장소로 업로드되어 있으며, 제출 마감 시한인 2019 년 6 월 25 일 17:00 이후에 공개 저장소로 전환될 예정이다. https://github.com/thebarbershop/sogang-cse4120

2. 개발 절차

(1) C-Minus 프로그래밍 언어와 SPIM 시뮬레이터 각각의 특성을 고려하여 활성 레코드의 구조를 결정한다.

(2) 결정한 활성 레코드 구조에 부합하도록 각 심볼의 메모리 위치를 적절히 계산하도록 심볼 테이블 구성을 수정한다.

(2) main 함수를 선언하는 코드, 상수를 계산하는 코드와 내장 출력 함수를 호출하는 코드를 구현한다.

(3) 전역변수의 선언, 대입 및 참조를 구현한다. 전역변수에 이용할 수 있도록 내장 입력 함수를 구현한다.

(4) 각 이진연산자를 처리하는 코드를 구현한다.

(5) 레이블을 생성하는 기능을 구현하고, 이를 이용해 반복 및 조건 제어문을 처리하는 코드를 구현한다.

(6) 지역변수, 전역배열, 지역배열의 선언, 대입 및 참조를 구현한다.

(7) 함수 호출 및 반환 시퀀스의 코드 생성을 구현한다. 레지스터를 이용한 인자 전달을 처리한다.

(8) C- 프로그래밍 언어 문법에 따라 테스트 프로그램을 작성하고 완성된 코드생성기에 입력하여 결과를 확인한다.

IV. 연구 결과

*** 활성레코드의 구조와 프로시저 호출 루틴**

각 프로시저의 활성레코드에 필수적인 요소는 해당 프로시저가 종료된 뒤 실행할 명령어의 주소 (반환주소; Return Address)와 해당 함수의 지역변수들이다. 반환 주소는 프로시저의 프레임 포인터가 가리키는 주소인 0($fp)에 저장되어있고, 지역변수는 -4($fp)[2]부터 시작하여 선언된 순서대로 스택에 자리를 차지한다.

(양의 방향)		→스택 진행 방향			(음의 방향)
호출 이후에도 유지해야 할 레지스터들	호출 인자들 (0 번~3 번 인자는 레지스터에 저장)	호출하는 프로시저의 프레임 포인터 주소 (Control Link)	반환 후 실행할 명령어 주소 (Return Address)	호출된 프로시저의 지역변수들	
			`$fp`	`$sp`	

도표 1: 프로시저 호출 직후의 스택 상태

프로시저를 호출할 때에는, 호출 후에 유지해야 할 레지스터 등이 우선 스택이 삽입되고, 전달할 인자가 4 개보다 많을 경우 4 번 인자부터는 스택에 역순으로, 즉 4 번 인자가 가장 마지막에 들어가도록 삽입된다. 그 다음, 호출된 프로시저가 종료된 뒤 기존 프레임 포인터를 복구할 수 있도록 프레임 포인터를 스택에 삽입한다. 따라서, 4($fp)는 컨트롤 링크의 주소이고, 4 번 인자는 8($fp), 5 번 인자는 12($fp) 등으로 접근할 수 있다.

프로시저 호출 루틴을 올바르게 구현하기 위해서는 심볼 테이블 구성 기능을 수정해야 한다. 함수의 첫 네 인자에는 메모리 오프셋 대신 레지스터의 번호가 부여되어야 한다. 각 함수마다 파라미터와 지역변수에 필요한 메모리가 각각 몇 바이트 씩인지를 알아야 프로시저 호출 과정에서 스택에 공간을 올바르게 확보할 수 있다. 이 부분 또한 심볼테이블을 구성할 때 계산하여 함수 정의 노드에 저장한다. 파라미터에 필요한 메모리는 파라미터 개수가 4 개 이하일 경우에는 0, 그 이상일 경우에는 ((파라미터의 수) − 4) × 4 이다. 지역변수에 필요한 메모리는 심볼테이블을 구성하면서, 그 함수에 속한 각 지역변수의 메모리 오프셋 중 절대값이 가장 큰 값이다. 이 부분은 심볼테이블이 구성되는 `symtab.c:insertNode` 함수에 구현되어있다. 그 결과 수정된 심볼 테이블의 구조는 다음과 같다.

2 SPIM 시뮬레이터는 MIPS32 아키텍처를 구현하므로 메모리 주소는 워드 크기인 4 바이트를 차지한다.

```
** Symbol table for scope of function g declared at at line 3
Symbol Name  Scope  Offset  Stack  Class     Array Size  Param.  Type  Line Numbers
-------------------------------------------------------------------------------
a               1  $a0         -  Parameter          -       -  int      3    5    5
b               1  $a1         -  Parameter          -       -  int      3
c               1  $a2         -  Parameter          -       -  int      3
d               1  $a3         -  Parameter          -       -  int      3
e               1       4      -  Parameter          -       -  int      3
f               1       8      -  Parameter          -       -  int      3

** Symbol table for scope of function main declared at at line 7
Symbol Name  Scope  Offset  Stack  Class     Array Size  Param.  Type  Line Numbers
-------------------------------------------------------------------------------
x               1     -24      -  Variable           5       -  int      9   10   11

** Symbol table for global scope
Symbol Name  Scope  Offset  Stack  Class     Array Size  Param.  Type  Line Numbers
-------------------------------------------------------------------------------
main            0       -    -24  Function           -       0  void     7
input           0       -      0  Function           -       0  int     -1
g               0       -     -4  Function           -       1  void     3   11
output          0       -      0  Function           -       1  void    -1
x               0       -      -  Variable           -       -  int      1
```

도표 2: 심볼 테이블의 구조

프로시저 호출 루틴은 `cgen.c:cgenExp` 함수에 구현되어있다. 함수 호출 노드를 만났을 때 해당 함수 명이 `input, output` 이 아니면 [도표 2]와 같이 스택이 형성되도록 코드를 생성한다. 유일하게 호출된 함수의 지역변수를 위해 스택 메모리를 할당하는 코드만 `cgenFunDecl` 함수에서 처리한다. 주의할 점은 인자를 하나 씩 계산하면서 레지스터/스택의 해당 위치에 저장하면 안 되고, 우선 계산 되는 대로 모두 스택에 삽입했다가 하나씩 제거하면서 올바른 위치에 저장해야 한다는 것이다. 이는 각 인자 계산이 이루어지는 과정에서 $a0-3 레지스터에 값이 덮어씌워지는 것을 피하기 위함이다. 다음 인자에서 어떤 레지스터 값을 이용해야 하는데, 이번 인자가 그 값을 덮어쓰면 다음 인자를 올바르게 계산할 수 없다.

*** 내장 입출력 함수**

내장 입출력 함수 `input`, `output` 은 레지스터에 적절한 값을 넣은 뒤 시스템콜을 호출하는 것으로 구현할 수 있다. 따라서 따로 코드 영역에 독립적인 프로시저 코드를 생성할 필요 없이 매 호출이 일어날 때마다 레지스터 할당과 시스템콜 호출 코드만 생성하면 된다.

입력을 받을 때에는 콘솔 화면에 "input: ", 출력을 할 때에는 "output: "이 표시되도록 했다. 이를 위해서 데이터 영역에 다음과 같이 문자열 세 개를 저장한다. C-Minus 프로그래밍 언어 스펙 상 심볼 이름에 언더바(_)를 이용할 수 없기 때문에, 이와 같은 컴파일러 전용 레이블 명에 언더바를 두 개 붙임으로써 C-Minus 소스코드 상의 레이블(함수명, 전역변수명)과 중복되지 않게 할 수 있다. 이렇게 준비된 문자열을 출력 함수와 동일한 절차로 화면에 표시한다.

```
.data
__inputStr:  .asciiz "input: "
__outputStr: .asciiz "output: "
__newline:   .asciiz "\n"
```

도표 3: 입출력을 위한 문자열 리터럴

입력 시스템콜은 $v0 에 시스템콜 번호를 저장하여 호출하면 입력된 숫자가 $v0 에 저장된다. 출력 시스템콜은 레지스터 $a0 에 출력할 값을, $v0 에 시스템콜 번호를 저장하여 호출한다. 따라서 출력 시스템콜을 올바르게 구현하기 위해서는 우선 함수의 인자로 지정된 표현식을 계산하고, 그 값을 $a0 에 저장하여야 한다. 이 과정에서 레지스터 오염을 피하기 위해, 인자를 계산하기 전 두 레지스터 $a0 과 $v0 을 스택에 저장하고, 출력이 모두 끝난 후 다시 값을 되돌려야 한다. 이 부분은 `cgen.c:cgenExp` 함수에 구현되어있다. 함수 호출 노드를 만났을 때 해당 함수 명이 `input, output` 이면 처리하도록 구현되었다.

*** 전역변수와 함수 생성**

지역변수 및 파라미터와는 달리 전역변수와 함수는 각각 힙 영역과 코드 영역에 정적으로 생성된다. 모두 동일하게 전역 스코프에 정의된 심볼들이므로, 중복되는 이름이 없다고 보장된다. 따라서 심볼 이름을 그대로 레이블 이름으로 삼아 힙 영역 및 코드 영역에 접근하게 할 수 있다. 단, SPIM 시뮬레이터 스펙 상 연산자와 레이블의 이름이 같으면 안 되기 때문에, 이런 문제를 피하기 위해 main 함수를 제외한 모든 전역 심볼 이름 앞에 언더바를 붙여서 레이블로 사용한다.

*** 변수의 대입과 참조**

변수가 전역변수인지 지역변수인지, 파라미터인지 배열인지 그냥 변수인지에 따라 대입과 참조 방식이 달라질 수 있다. 우선 모든 배열형은 배열의 첫 원소가 할당된 메모리 주소를 기준으로 대입과 참조가 이루어져야 한다. 배열이 전역변수일 때, 지역변수일때, 그리고 파라미터로 전달 받은 주소값일 때, 파라미터라면 레지스터에 있는 파라미터일 때와 스택에 있는 파라미터일 때 각각 필요한 코드가 모두 다르다. 이에 해당하는 코드는 `cgenArrayAddress` 에서 처리하여 $v0 에 저장되도록 한다.

전역변수는 레이블을 그대로 읽어 그 주소에 값을 쓰거나 그 주소로부터 값을 불러올 수 있다. 파라미터라면 해당 파라미터가 레지스터에 저장되었는지 스택에 저장되었는지를 판단해 올바른 위치를 찾아 값을 쓰고 불러와야 한다. 대입은 `cgenOp` 에서 호출하는 `cgenAssign` 에 구현되어있고, 참조는 `cgenExp` 에 구현되어있다.

*** 연산자**

C-Minus 프로그래밍 언어의 모든 연산자는 이진연산이다. 따라서 모든 연산에 앞서 두 피연산자를 계산해 레지스터에 하나씩 저장하고, 연산자에 따라 적절한 MIPS 연산자를 이용해 연산의 결과를 계산할 수 있다. 계산된 결과는 레지스터 $v0 에 저장된다. 각 피연산자를 $t0, $1 에 저장하고 연산을 하려면, 첫 피연산자를 계산하는 코드를 먼저 출력한 뒤, $v0 에 있을 결과물을 스택에 삽입해야 한다. 그 다음 둘째 피연산자를 계산하고, 이를 $v0 에서 $t1 로 복사한 뒤, 스택에서 첫 피연산자의 값을 $t0 으로 복사하여 연산할 수 있다. 연산자 노드에 해당하는 코드 생성은 `cgenOp` 에서 처리한다.

*** 조건문 및 반복문 제어**

제어문을 위해 우선 레이블 관리가 필요하다. 이때 사용할 임시 레이블은 L# 형식으로 하나씩 필요할 때마다 숫자를 늘려가면서 생성한다.(`getLabel` 함수) C-Minus 프로그래밍 언어 스펙 상 변수 이름에 숫자가 들어갈 수

없기 때문에 레이블 충돌의 우려가 없다. 레이블과 조건분기, 무조건분기를 이용한 조건문과 반복문의 구현은 전형적이므로 구체적인 설명은 생략한다. 다만 조건문에 else 절이 있는지에 따라 다른 코드가 생성되어야 한다는 점을 유의해야 한다.

*** 함수 반환문**

함수 반환은 오직 정수형 함수에서만 일어날 수 있다. 함수 반환 노드에서는 반환할 표현식을 계산한 후, $v0 에 저장된 결과값을 그대로 두고 반환 주소로 분기하는 코드를 작성한다. 그 결과 함수의 반환값은 $v0 에 저장된다.

*** 코드 파일 생성**

C-Minus 프로그래밍 언어로 작성된 소스코드를 SPIM 시뮬레이터에서 작동하는 MIPS 어셈블리 코드로 변환한 결과를 파일로 저장한다. 생성하는 파일명은 입력으로 주어진 소스코드의 파일명에서 확장자를 제거하고 .tm 를 붙인다.

참고자료로 삼은 교재의 TINY 컴파일러 코드는 새로운 코드의 파일명을 계산할 때 문자열의 앞에서부터 탐색하여 가장 먼저 발견하는 '.'을 기준으로 파일명을 삼고 거기에 확장자 'tm'을 붙인다. 하지만 이 알고리즘은 ㅇ 입력 파일명을 '../test.c', './test.c' 등과 같이 했을 때 출력 파일을 '.tm'으로 만드는 오류가 있다. 이를 해결하기 위해, 함수 `util.c:getBaseIndex` 를 새로 작성했다. 이 함수는 입력파일명을 뒤에서부터 탐색하여 '.'를 만나면 해당 문자의 인덱스를, '/'를 만나면 해당 문자열의 길이를 반환한다. 이렇게 하면 확장자가 있는 입력파일명은 확장자를 잘라낸 베이스 파일명을 쉽게 얻을 수 있고, 확장자가 없는 입력파일명도 오류없이 베이스 파일명을 구할 수 있다.

다음과 같이 컴파일러와 SPIM 시뮬레이터를 실행할 수 있다.

```
예시 1)
$ ./project4_17 sample.c
$ spim -file sample.tm

예시 2)
$ ./project4_17 ../sample.c
$ spim -file ../sample.tm

예시 3)
$ ./project4_17 ../sample
$ spim -file ../sample.tm
```

도표 4: 실행 방법 예시

V. 평가 및 감상

본 프로젝트는 [1]에 명세된 C- 프로그래밍 언어의 컴파일러를 제작하는 네 번째 단계로 MIPS 어셈블리 코드 생성기를 구현하였다. 다양한 테스트 프로그램을 이용해 생성된 코드가 적절하게 동작하는 것을 확인하였다. 교재에 주어진 정렬 프로그램은 배열을 파라미터로 이용하고, 함수 내에서 다른 함수를 호출하는 등 파라미터의 처리가 올바르게 이루어져야 정상적으로 작동한다. 마찬가지로 교재에 주어진 최대공약수 계산 프로그램은 재귀호출로 구현되어 프로시저 호출 시퀀스 및 반환 시퀀스가 올바르게 작성되어야 작동한다. 그 밖에 여러 수를 배열에 입력 받아 최대값과 최소값을 구하는 프로그램, 입력받은 수 중 짝수의 개수를 세는 프로그램, 피보나치 수열을 계산하는 프로그램, 팩토리얼을 계산하는 프로그램 등을 작성해 각종 연산자, 함수 호출, 배열 접근 및 대입 등이 올바르게 작동함을 확인했다. 그외에 실질적 의미는 없지만 문법적으로 복잡한 여러 프로그램을 작성하여

배열 인덱스로 함수호출과 연산을 포함한 복잡한 식이 주어지는 경우, 변수 음영이 일어나는 경우 등 다양한 상황에서 정상적으로 작동하는 것을 확인하였다.

두 가지 부분에서 주요하게 어려움을 겪었다. 첫 번째로 활성 레코드를 생성하여 각종 변수와 파라미터를 이용할 때 메모리 주소 계산에 오류가 생기는 경우가 많았다. 이를 해결하기 위해, 앞서 제시한 것과 같은 활성 레코드의 구조를 확정하고 다시 심볼테이블 구현 단계로 돌아가 각 심볼의 위치를 정확하게 다시 계산하였다.

두 번째로, 함수 인자 계산, 연산식 처리, 시스템콜 호출 등 다양한 시나리오에서 다음 단계에 사용하려고 저장해둔 레지스터가 하위 노드를 처리하는 과정에서 다른 값으로 덮어 씌워져서 값을 잃어버리는 오류가 있었다. 이런 경우를 방지하기 위하여, 레지스터에서 값을 계산한 뒤 곧 사용하고자 남겨둔 채로 하위 노드를 처리할 때에는, 레지스터에만 값을 남겨두지 말고 반드시 스택에 삽입했다가 하위노드 처리가 끝난 뒤 스택에서 삭제하여 레지스터를 복구하도록 하였다.

한 학기에 걸쳐 단계별로 수행한 컴파일러 설계 과제가 드디어 실제 코드를 실행할 수 있는 단계에 이르게 되어 보람이 컸다. 하지만 동시에 상당한 난이도로 구현했음에도 불구하고 C-Minus 프로그래밍 언어의 단순한 스펙 상 실제 작성할 수 있는 프로그램에 한계가 있음이 아쉬웠다. 또, 안전하게 작동하는 코드를 생성하는데만 집중하다보니 어떤 값이 스택에 삽입되자마자 삭제되거나, 레지스터 복사 연산이 불필요하게 연속해서 반복되는 코드가 생성되기도 했다. 코드 최적화에 관한 고민을 거의 하지 못한 아쉬움이 남았다.

VI. 참고 문헌

[1] Kenneth C. Louden. 1997. *Compiler Construction: Principles and Practice*. PWS Publishing Co. Boston, MA, USA. pp. 491-544.

[2] James R. Larus. 1997. *SPIM S20: A MIPS R2000 SIMULATOR*. URL: http://pages.cs.wisc.edu/~larus/SPIM/spim_documentation.pdf Accessed 25 June, 2019.