**컴파일러의 기초: Project #4 for Code Generation**

**Due date: 12월 20일 월요일 23:59**

이번 프로젝트의 목적은 lex와 yacc utility를 이용하여 (C의 feature를 가감한, subc라 부른다)에 대한 프로젝트 3에서 구현한 문법을 바탕으로 Code Generator 를 만드는 것이다.

프로젝트의 시작에 앞서 **강의 교재를 철저하게 공부하여** 프로젝트를 준비한 다음, 문서에 언급된 Code Generator 를 구현하도록 한다. 구현 방법과 제한사항, 문법 등은 다음과 같다.

* **How to do this project4?**

Environment: 이전 환경과 동일; flex, bison 사용

Makefile: 이전에 주어진 형태에서 바꿔서 사용

Simulator: 생성된 코드를 수행하기 위한 스택 기반의 시뮬레이터이며, 시뮬레이션을 통해 생성된 코드의 정확성을 검증한다.

* **What is the stack simulator?**

본 수업을 위해 연구실에서 개발한 스택 기반 어셈블리 시뮬레이터로, 사용방법은 다음과 같다.

1. 제공된 스택 시뮬레이터의 압축을 푼 뒤 make를 하여 sim 바이너리 파일을 만든다.

**( unzip sim, make, ls (sim 바이너리 파일이 만들어졌음을 확인) )**

1. 수업시간에 배운 rule을 기반으로 code generator 를 완성한다. (subc)
2. Subc code generator는 **./subc (source .c file\_input) (assembly .s file\_output)** 의 형식으로 구현한다.

ex) ./subc test.c output.s

1. Output file 인 assembly (result.s라고 가정) 을 스택 기반 시뮬레이터에 돌려 결과를 확인한다.

ex) ./sim result.s

* **Stack simulator assembly code**

각 instruction의 실제 동작을 확인해보려면 gram.y파일의 357번째 줄에 정의된 함수인simulate\_stack\_machine를 확인해 보면 된다.

* **Registers**

|  |
| --- |
| 1. sp: stack top을 가리키는 포인터 2. fp: stack frame을 가리키는 포인터 3. pc: 현재 수행중인 프로그램의 program counter |

레지스터는 모두 값이 커지는 방향으로 진행한다.

* **Start up code**

shift\_sp 1

push\_const EXIT

push\_reg fp

push\_reg sp

pop\_reg fp

jump main

EXIT:

exit

실제 함수의 시작은 main이라는 이름을 가지는 label을 만들어서 사용한다.

* **allocate global data area**
  + Lglob. data <uninitialized global data size>

전역 변수를 저장하기 위한 공간의 크기를 지정한다.

* **Global Data**
  + data area

|  |
| --- |
| 사용   1. <label>. data <size> |

전역변수를 저장하기 위해 사용되는 공간으로 gp는 data영역의 시작을 가리키고 있다. 전역변수의 값을 사용하기 위해서 (gp + offset)을 이용해서 접근한다.

* + string area

|  |
| --- |
| 사용   1. <label>. string <string> |

문자열을 사용하기 위해 사용되는 공간

문자열 HelloWorld!!! 저장하기 위해서는 Str0. string HelloWorld!!!\n 로 선언

* **Arithmetic/Logic Instruction:**
  + **Unary operation**

**[negate, not, abs]**

|  |
| --- |
| 동작   1. pop top element of stack 2. apply operation 3. push result onto stack |

* + **Binary operation:**

**[add, sub, mul, div, mod, and, or, equal, not\_equal]**

**[greater, greater\_equal, less, less\_equal]**

|  |
| --- |
| 동작   1. pop two top elements of stack 2. apply operation as top element on right hand, second element of left hand 3. push result onto stack |

* **Control**
  + **unconditional jump**

**[jump]**

|  |
| --- |
| 동작   1. jump [label][+/-offset] |

label이나 offset을 둘 다 사용하거나 하나만 사용할 수 있다.

예를 들어서 jump L1 6은 PC값을 L1+6의 위치로 이동시키고, jump L1은 L1의 위치로, jump 6은 PC+6의 위치로 이동한다.

* + **conditional jump**

**[branch\_true, branch\_false]**

|  |
| --- |
| 동작   1. pop top element of stack 2. branch\_true [label][+/- offset] 3. branch\_false [label][+/- offset] |

unconditional jump와 마찬가지 방법으로 사용된다.

* + **terminate program**

**[exit]**

|  |
| --- |
| 동작   1. terminate program |

프로그램을 종료한다.

* **stack manipulation**
  + **push values to stack**

**[push\_const, push\_reg]**

|  |
| --- |
| 동작:   1. push\_const <constant> 2. push\_reg <reg> |

push\_const는 숫자 값 혹은 label의 이름(ex. Str0)이 될 수 있다.

* + **pop values from stack**

**[pop]**

|  |
| --- |
| 동작   1. pop\_reg <reg> |

* + **shift stack pointer**

**[shift\_sp]**

|  |
| --- |
| 동작   1. shift\_sp <integer constant> |

sp를 지정된 숫자만큼 이동한다.

* **assign/fetch**
  + **assign values into specified address**

**[assign]**

|  |
| --- |
| 동작   1. pop two elements from stack 2. assign value of the top element into address specified by second element |

* + **fetch value from specified address**

**[fetch]**

|  |
| --- |
| 동작   1. pop top element from stack 2. get value from the address specified by top element 3. push read value onto stack |

* **I/O**
  + **input**

**[read\_int, read\_char]**

|  |
| --- |
| 동작   1. read\_int: 숫자(integer)를 입력받는다. 2. read\_char: character를 입력받는다. |

입력받은 integer/character는 stack top에 저장된다.

* + **output**

**[write\_int, write\_string]**

|  |
| --- |
| 동작   1. write\_int: 화면에 숫자를 출력한다. 2. write\_string: 화면에 문자열을 출력한다. |

데이터 영역(ex. Str0. string HelloWorld!!!)에 있는 문자열을 출력할 경우

1. push\_const Str0

2. write\_string

의 순서로 실행한다.

스택에 있는 문자열을 출력할 경우

1. push\_reg sp (sp는 문자열의 시작을 가리키도록)

2. write\_string

문자열을 데이터영역에 저장하고 출력하는 방법을 권장

* **Intermediate Code Representation**

"negate", "not", "abs",

"add", "sub", "mul", "div", "mod", "and", "or", "equal", "not\_equal",

"greater", "greater\_equal", "less", "less\_equal",

"jump", "branch\_true", "branch\_false", "exit",

"push\_const", "push\_reg", "pop\_reg",

"shift\_sp",

"assign", "fetch",

"read\_int", "read\_char",

"write\_int", "write\_char", "write\_string",

"sp", "fp", "pc",

"data", "string",

* **TODO**

프로젝트 3과 마찬가지로 이번 프로젝트 역시 많은 부분의 해법이 강의 교재에 있으므로 이를 바탕으로 진행한다.

프로젝트 4의 입력이 되는 소스코드는 문법적으로 아무 문제가 없는 코드이다. 즉 syntax error와 semantic error가 발생하지 않는 코드들이 수행 대상이다. 따라서 프로젝트 3을 완벽하게 구현하지 못했더라도 프로젝트 4를 진행할 수 있다. 다만, 프로젝트 3에서 symbol table 구조가 잘못되었거나 변수, 함수, 구조체의 선언조차 처리하지 못한다면 프로젝트의 진행이 어려우므로 먼저 수정하도록 한다.

이번 프로젝트를 다음 순서로 진행하는 것을 권장한다.

1. 강의 교재를 꼼꼼하게 공부해서 각 상황에 맞는 코드 생성에 대한 개념을 익힌다.
2. 본 문서와 gram.y의 357번째 줄에 정의된 simulate\_stack\_machine 함수를 분석해서 시뮬레이터의 정확한 동작을 이해한다.
3. 프로젝트 4의 문법 제한 사항에 맞게 문법을 수정한다.
4. 코드 생성을 위해서 프로젝트 3에서 디자인한 symbol table에 몇 가지 필드를 추가하고, 각 non-terminal에 대응하는 data structure를 검토한다.
5. subc.y의 적절한 위치에서 코드를 생성하도록 파일을 수정한다.
6. 비교적 간단한 코드를 먼저 생성해보고, 시뮬레이터를 이용해 테스트하면서 모든 상황을 처리할 수 있도록 프로그램을 점차 확장시킨다.

* **Submission**

source code, Makefile, README file(이름, 학번, 이메일, 심화구현 여부 등), **결과 보고서**

**결과 보고서**

* 전체적인 디자인에 대해 설명
* 완성도에 대해서 설명 (구현한 것과 구현하지 못한 것을 구분)
* 심화 구현 사항에 대해서는 보다 상세히 작성 (언급이 없다면 평가 X)

**project4\_학번.zip**으로 압축하여 eTL을 통해 제출한다.

**TA Contact**

* + 조중하 (301동 819호)
  + E-mail: zoonghi@snu.ac.kr