프로젝트 4 보고서

2016-17069 이상민

2017-17088 박찬정

1. 기본 구현

1) I/O 구현

I/O 함수를 지원하기 위해서 wirte\_int, write\_string, write\_char를 subc.l 에서 symboltable에 추가해주었다. 그래서 이를 만났을 때 WRITEINT, WRITECHAR, WRITESTR 토큰을 넘겨주도록 하였으며, subc.y 파일에서는

stmt | WRITEINT '(' unary ')'

stmt | WRITECHAR '(' unary ')'

stmt | WRITESTR '(' unary ')' 문법에 대하여, 괄호 안의 unary가 decl\_class가 \_VAR이면 fetch를 프린트하도록 하였고, 그리고 그 다음 줄에 wirte\_int, write\_char, write\_string를 출력하도록 하였다.

2) 전역변수

파스트리 가장 상위의 program : ext\_def\_list 문법에서 전역변수를 관리하는 ssetop이 가르키는 struct sse의 내부변수인 size를 이용하여, 출력 파일의 마지막에 Lglob. data 가 몇 사이즈인지 출력한다. int, char, pointer 모두 1 word로 계산하며, ssetop->size의 경우는, functions.c 의 declare() 함수와 push\_stelist() 함수에서 ssetop->size를 증가시킨다.

전역변수에 값을 대입하는 것은,

unary | ID 에서 만약 해당 ID 토큰이 global인 경우에 push\_const Lglob+ offset 을 한다. 이를 통해 그 주소 값에다가 값이 들어가도록 한다.

만약 이때 전역변수가 아니라면 fp 레지스터를 기준으로 offset을 더해줘서 해당 변수에 값이 들어가도록 하였다.

3) 함수

함수의 경우, 함수 이름에 대한 라벨을 만들어서, 함수가 호출 될 때 그 라벨로 점프할 수 있도록 한다. function\_compound\_stmt : '{' local\_defs stmt\_list '}'문법에서 함수가 시작하기 전에 local변수를 위한 스택 공간할당을 shift\_sp 를 이용하여서 하였다.

2. 추가 구현

2-1) loop 구현

* While

우선 다른 while문마다 다른 이름의 라벨을 사용하기 위해 global 변수로 loop문 사용 횟수의 카운터와 현재 loop문의 id를 정의하고 라벨의 끝에 붙이도록 했다. while문의 stack machine 코드는 다음과 같은 구조로 구현하였다.

|  |  |
| --- | --- |
| Loop\_start\_%d: | 한 번의 Loop가 시작되는 부분을 의미하는 라벨이다. |
| 조건 확인용 expr | Expr이 내부 연산을 마치고 stack top에 연산 결과를 둔다. |
| Branch\_false loop\_end\_%d | Expr이 두고 간 결과가 false라면 loop를 탈출하는 라벨로 간다. |
| Loop 코드 | 한 번의 loop에서 실행할 코드이다. |
| Loop\_last\_%d: | 한 번의 loop이 끝나면 실행하는 코드이다. |
| Jump loop\_start\_%d | while에서는 그냥 loop의 처음으로 이동하기만 하면 된다. |
| Loop\_end\_%d: | Loop 조건이 해제되어 탈출할 때 오는 곳이다. |

* For

for문도 마찬가지로 각기 다른 for문마다 다른 이름의 라벨을 사용하기 위해 global 변수로 loop문 사용 횟수의 카운터와 현재 loop문의 id를 정의하고 라벨의 끝에 붙이도록 했다. for문의 stack machine 코드는 다음과 같은 구조로 구현하였다.

|  |  |
| --- | --- |
| Expr\_e | For문 진입시에 무조건 실행하는 expression이다. |
| Loop\_start\_%d: | 한 번의 Loop가 시작되는 부분을 의미하는 라벨이다. |
| 조건 확인용 expr | Expr이 내부 연산을 마치고 stack top에 연산 결과를 둔다. |
| Branch\_false loop\_end\_%d | Expr이 두고 간 결과가 false라면 loop를 탈출하는 라벨로 간다. |
| Jump loop\_body\_%d | Loop\_last 범위를 수행하지 않고 body 코드 영역으로 이동한다. |
| Loop\_last\_%d: | 한 번의 Loop이 끝나고 나면 실행하는 코드 부분이다. |
| Loop의 마지막 Expr\_e | 코드에서 요구하는 expr를 수행한다. |
| Jump loop\_start\_%d | Loop의 처음으로 이동한다. |
| Loop\_body\_%d: | 조건 확인을 마치면 이 곳으로 온다. |
| Loop 코드 | 한 번의 loop에서 실행할 코드이다. |
| Jump loop\_last\_%d | 한 번의 loop에서 마지막에 무조건 실행할 코드를 수행하러 간다. |
| Loop\_end\_%d: | Loop 조건이 해제되어 탈출할 때 오는 곳이다. |

특이사항으로는 nested while, for를 지원하기 위하여 yacc에서 WHILE, FOR 토큰을 읽으면 midrule을 정의하여 기존의 while\_now를 저장해두고, while이 끝나면 while\_now 변수를 기존 값으로 다시 복구하는 작업을 수행한다.

* Break

Break의 경우 global 변수로 정의해둔 loop의 id를 사용한다. While과 for 모두 loop의 탈출 부분에는 loop\_end\_%d 형태의 라벨이 존재하므로, jump loop\_end\_%d 코드가 break;의 역할을 해줄 수 있다.

* Continue

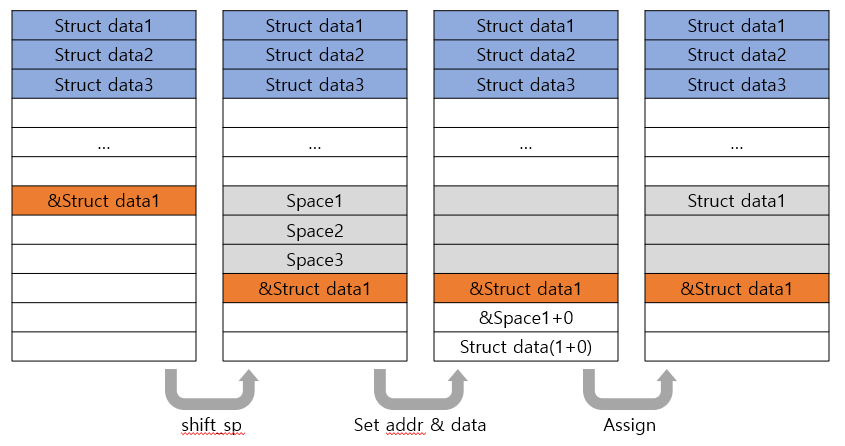
Continue도 마찬가지로 global 변수로 정의해둔 loop의 id를 사용한다. Continue는 loop의 마지막에 실행하는 코드가 수행되어야 하므로 loop\_last\_%d 로 jump시키면 된다.

2-2) Struct 구현

Struct assign, struct return, struct argument 모두 하나 이상의 stack element를 동시에 움직이는 동작을 요구한다. 이에 Struct와 관련된 여러 작업을 수행하기 위해 다음과 같은 함수를 정의하였다.

* Void fetch\_struct\_value(decl\* \_decl)

이 함수는 stack top에 있는 값을 주소로 하여 그 주소부터 \_decl의 size만큼의 데이터를 읽어오는 stack machine 코드를 생성한다. Stack을 다음과 같이 조작하여 stack top의 주소를 없애고 해당 주소에 있는 struct의 값으로 바꾼다.



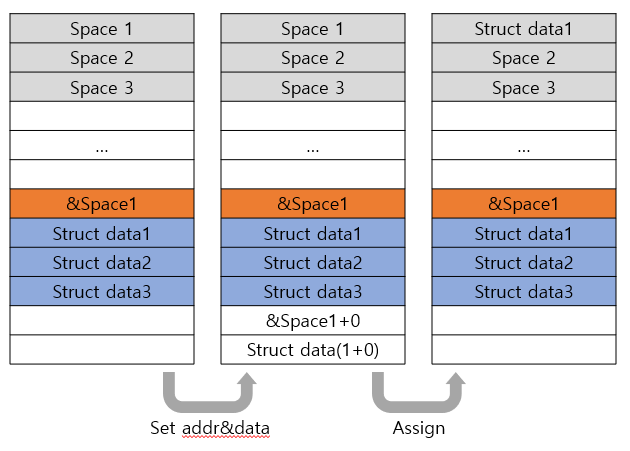
테이블이(가) 표시된 사진

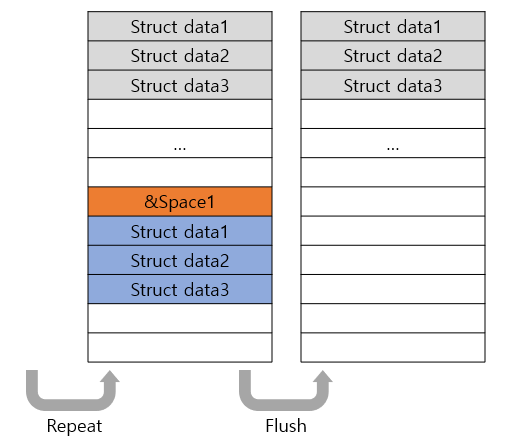
자동 생성된 설명

이 함수는 unary -> binary reduce에서 unary가 struct의 address를 넘겨주는 경우, 함수의 argument로 struct를 제공할 때 사용된다.

* Void assign\_bulk(decl\* \_decl)

이 함수는 address, struct\_value가 stack에 들어있을 때 해당 address에 struct value를 assign하는 stack machine 코드를 생성한다. Stack을 다음과 같이 조작한다.





이 함수는 struct assign의 경우와 function의 return type이 struct인 경우에 사용된다.

* Void pop\_struct\_value(decl\* \_decl)

Stack top에 있는 struct value를 버린다.

테이블이(가) 표시된 사진

자동 생성된 설명

이 함수는 expr ‘;’ -> stmt reduce시에 사용된다.

기존의 assignment와 return value 제공(사실 둘은 매우 비슷하게 동작한다), argument 부여 방식을 struct 경우에 한하여 위 함수들로 코드를 생성하도록 하면 잘 동작한다.

이 함수들의 한 가지 특징은 변수로 들어오는 decl 객체의 size에 따라 code generation을 달리 수행한다는 점인데, 그렇기 때문에 size가 1이어도 잘 작동한다. 현재로서는 decl의 여러 특성을 고려하여 경우에 따라 일반적인 assign, fetch 등의 명령을 사용하거나 struct용으로 만든 이 함수들을 사용하도록 구현하였지만, 위 함수들로 모두 통합하여 구현할 수도 있을 것으로 보인다. 다만 이렇게 구현하면 성능의 저하가 있을 수 있다.