MiniC 컴파일러 만들기(6): Code Generation: x86 & function call

컴파일러, 12번째 시간

경민기, 2025-12-02

함수 호출 (function call)

- C / MiniC: result = add(x, y); 처럼 '값을 넘기고 결과를 받는 것'으로 보임

- 실제 동작
  - CPU 입장
    - 함수라는 개념이 없다. 단지 점프(jump)와 스택 조작만 있을 뿐
  - 컴파일러의 역할:
    - 고수준 함수 호출을 스택/레지스터 조작 + call/ret 명령으로 바꿈

- 이때의 약속이 바로 호출 규약(calling convention)

POSIX

- POSIX는 '이식 가능한 운영 체제 인터페이스(Portable Operating System Interface)'의 약자로, 서로 다른 유닉스 계열 운영체제 간의 호환성과 이식성을 높이기 위해 IEEE에서 제정한 표준

- Windows에서도 제한적으로 준수하고 있으며, WSL(Windows Subsystem for Linux)을 통해 완전한 POSIX 환경을 지원함

호출 규약 (SysV x86-64, 정수 기준)

- 인자 전달 순서:
  - 1번째 rdi, 2번째 rsi, 3번째 rdx, 4번째 rcx, 5번째 r8, 6번째 r9
  - 이후 스택
- 반환값: rax 레지스터에 넣어서 돌려줌
- 기타
  - 어떤 레지스터는 'caller가 보존', 어떤 레지스터는 'callee가 보존'해야 함
- 모든 컴파일러가 이 규약을 지켜서 코드를 만들기 때문에, 서로 다른 컴파일러로 만든 코드도 서로 호출 가능.

스택 프레임(Stack Frame) 개념

스택 프레임 기본 형태 (함수 f 안에서)

call f ; 스택에 return address push
pushq %rbp
movq %rsp, %rbp
subq $N, %rsp ; 지역 변수/임시값을 위한 공간 확보

[높은 주소]
...
(caller의 스택)
return address ← call이 push한 복귀 주소
old rbp ← push %rbp
local var1 (rbp-8)
local var2 (rbp-16)
...
[낮은 주소]

leave ; mov %rbp,%rsp; pop %rbp
ret ; return address로 점프

컴파일러 파이프라인과 함수 호출

컴파일러 파이프라인에서 함수 호출 처리

1) 프론트엔드 (lexer.l, parser.y)
- 소스 코드: result = add(x, y);
- 토큰화: IDENT(result), '=', IDENT(add), '(', IDENT(x), ',', IDENT(y), ')', ';'
- 구문 분석 후 AST 생성: Assign(Var "result", Call "add" [Var "x", Var "y"])

2) (선택) 중간 표현(IR)
- t1 = x
- t2 = y
- t3 = call add, 2
- result = t3

3) 백엔드 (codegen_x86.c)
- 인자 t1, t2를 호출 규약에 따라 rdi, rsi에 배치
- call add 생성, 반환값은 rax에 들어온다고 가정
- rax를 result가 저장된 스택 위치로 옮김

컴파일러에서의 함수 호출

- 고수준 함수 호출은 결국 "인자 전달 + 복귀 주소 저장 + 점프 + 결과 회수”
- 호출 규약(calling convention)은 이 과정을 표준화한 계약서
- 컴파일러는 AST/IR에서 Call 노드를 만나면, 이 계약을 지키는 기계어를 만들어냄
- MiniC 예제를 통해 ① AST, ② IR, ③ x86-64 코드 세 관점에서 함수 호출을 살펴볼 수 있음
- 이를 확장하여 다중 인자, 재귀, 중첩 호출, 라이브러리 함수 호출까지 구현할 수 있음