MiniC 컴파일러 만들기(4): Code Generation: From AST to IR

컴파일러, 10번째 시간

경민기, 2025-11-18

오늘 수업 내용

미니 컴파일러가 3단계로 확장되는 과정을 설명함

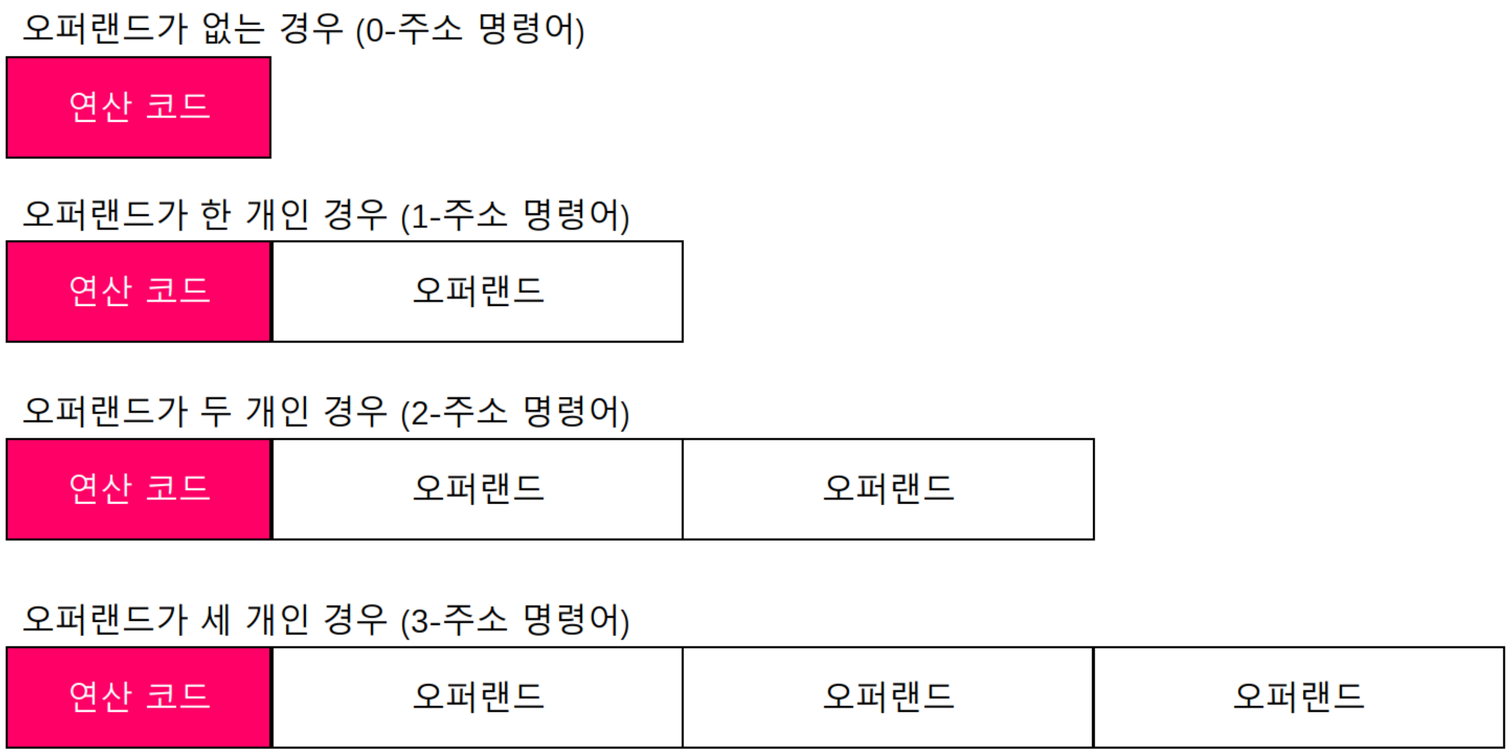
- 1단계: 계산기(AST) → IR (3주소 코드)
- 2단계: MiniC (변수/대입/print) → IR
- 3단계: MiniC + Evaluator → IR + 실제 실행 결과

중간코드 (intermediate code)

IR

3-어드레스 코드(Three-address code)는 컴파일러에서 사용되는 중간 언어

- 3-어드레스 코드(Three-address code)는 컴파일러에서 사용되는 중간 언어의 한 종류로, 컴파일러 최적화를 실현하는데 사용됨



IR 코드 형식

인텔 양식은 Windows에서, AT&T 양식은 리눅스와 macOS에서 씀

문법	Intel	AT&T
파라미터 순서	instr dest, src mov eax, 42 destination을 좌측, source를 우측에 표기한다.	instr src, dest movl $42, %eax source를 좌측, destination을 우측에 표기한다.
파라미터 표기	add eax, 1 어셈블러가 자동으로 타입을 파악한다.	addl $1, %eax immediate는 $, register는 % 기호를 앞에 붙여 표기한다.
피연산자 크기	mov eax, 1234 레지스터명으로부터 자동으로 결정된다.	movl $1234, %eax 크기를 나타내는 접미사를 사용하여 명시한다.
메모리 주소 표기	lea esi, [ebx+eax*8+4]	lea 4(%ebx,%eax,8), %esi

명령어 (1)

1. 데이터 전송
2. 산술/논리 연산
3. 제어 흐름 변경
4. 입출력 제어

데이터 전송

- MOVE : 데이터를 옮긴다.
- STORE : 메모리에 저장하라.
- LOAD(FETCH) : 메모리에서 CPU로 데이터를 가져와라
- PUSH : 스택에 데이터를 저장하라
- POP : 스택의 최상단 데이터를 가져와라

산술/논리 연산

- ADD / SUBTRACT / MULTIPLY / DIVIDE : 덧셈 / 뺄셈 / 곱셈 / 나눗셈을 수행하라
- INCREMENT / DECREMENT : 오퍼랜드에 1을 더하라 / 오퍼랜드에 1을 빼라
- AND / OR / NOT : AND/OR/NOT 연산을 수행하라
- COMPARE : 두 개의 숫자 또는 TRUE/FALSE 값을 비교하라

명령어 (2)

1. 데이터 전송
2. 산술/논리 연산
3. 제어 흐름 변경
4. 입출력 제어

제어 흐름 변경

- JUMP : 특정 주소로 실행 순서를 옮겨라
- CONDITIONAL JUMP : 조건에 부함할 때 특정 주소로 실행 순서를 옮겨라
- HALT : 프로그램의 실행을 멈춰라
- CALL : 되돌아올 주소를 저장한 채 특정 주소로 실행 순서를 옮겨라
- RETURN : CALL을 호출할 때 저장했던 주소로 돌아가라

입출력 제어

- READ(INPUT) : 특정 입출력 장치로부터 데이터를 읽어라
- WRITE(OUTPUT) : 특정 입출력 장치로 데이터를 써라
- START IO : 입출력 장치를 시작하라
- TEST IO : 입출력 장치의 상태를 확인하라

IR code example

```
# Calculate one solution to the [[Quadratic
equation]].
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

예제 1: 계산식 -> IR

1단계: 계산기 IR 생성기

정수 + - * / ( ) 로 이루어진 수식을 AST로 만든 뒤 IR(3주소 코드)로 변환

- scanner.l – 문자열 입력을 토큰(NUMBER, 연산자, 괄호, 개행)으로 쪼갬
- parser.y – 토큰을 AST(AST_*)로 파싱하고, 한 줄이 끝날 때마다 gen() 호출
- ast.h/ast.c – AST 노드 구조와 생성 함수들
- codegen.c – AST를 후위순회하면서 IR_* 명령들을 emit
- ir.h/ir.c – IR 리스트를 관리하고 출력
- main.c – yyparse() 호출해서 전체 파이프라인 실행
- Makefile – Bison/Flex + C 컴파일/링크 자동화

main.c

프로그램 전체를 시작하는 코드

- main 함수는 프로그램을 시작할 때, 먼저 “표현식을 입력하라”는 안내 문구를 한 줄 출력
- 그 다음 yyparse()를 호출해서 Bison 파서를 실행시킴
- 이때 파서는 내부적으로 Flex 스캐너(yylex)를 호출해서 입력을 토큰으로 읽어오고, AST를 만들고, 코드 생성기를 실행함.
  - 사용자가 EOF(Ctrl+D)를 입력할 때까지,
  - yyparse()는 여러 줄을 계속 읽으며 줄 하나당 “AST 생성 → IR 생성 → IR 출력 → IR 정리” 과정을 반복함
- main 자체는 복잡한 일을 하지 않고, “전체 파이프라인을 한 번 돌려라”라는 진입점 역할

scanner.l

입력을 토큰으로 쪼개는 코드 (Flex)

- 이 파일은 Flex 스캐너 정의로, “문자열을 어떤 단위로 잘라서 Bison에게 넘길지”를 담당합니다. [0-9]+ 패턴은 “숫자로만 이루어진 덩어리”를 의미함
- 이 패턴을 만나면, 그 문자열을 atoi로 정수로 바꾼 다음, yylval.ival에 넣고 NUMBER 토큰으로 반환
  - [\t\r]+ 패턴은 공백, 탭, 캐리지리턴들을 의미하며, 이 경우에는 아무 토큰도 반환하지 않고 그냥 무시
  - \n은 개행 문자를 만나면 '\n' 토큰을 반환해서, Bison의 line 규칙에서 줄 단위 처리를 할 수 있도록 해 줌
  - 나머지 한 글자(. 패턴)는 그대로 해당 문자 하나를 토큰으로 반환함. 덕분에 +, -, *, /, (,) 같이 별도의 토큰 이름을 정의하지 않아도 문자 그대로 쓸 수 있음.
- yywrap 함수는 입력이 끝났을 때 1을 리턴해서 스캐너가 종료되게 하는 기본 구현입니다.

parser.y (1)
문법 규칙 + AST 생성 + 코드 생성 트리거 (Bison)

- 이 파일은 Bison 문법 파일로, “입력 문자열을 어떻게 잘라서 어떤 트리(AST)로 만든 뒤, 언제 코드 생성기를 호출할지”를 정의
- %union 선언으로, 파서가 다룰 수 있는 값의 타입을 정해놓음
  - 여기서는 정수 토큰의 값을 저장하는 int ival과 식 전체를 담는 AST 포인터 AST* node를 사용
- NUMBER 토큰은 ival 타입을 사용한다고 선언하고, expr라는 비단말(symbol)은 node 타입(AST 포인터)을 쓴다고 지정함
- %left '+' '-'와 %left '*' '/' 선언은 연산자 우선순위 및 결합 방향을 알려줘서 자동으로 * 와 /가 +와 -보다 먼저 계산되게 함

parser.y (2)

- input 규칙은 여러 줄을 처리할 수 있도록, “입력이 비어 있을 수도 있고, input line이 반복될 수도 있다”라고 정의
- line 규칙에서 핵심: expr '\n' 을 읽었을 때, 그 expr이 만든 AST에 대해 gen($1)을 호출해서 IR을 만들고, 결과가 들어있는 임시 변수가 무엇인지 출력하고, print_ir()로 IR 전체를 출력한 뒤, free_ir()로 IR을 정리함. 빈 줄 '\n'은 그냥 넘어감
- expr 규칙들은 각각
  - expr '+' expr → ast_bin(AST_ADD, $1, $3)
  - expr '-' expr → ast_bin(AST_SUB, $1, $3)
  - expr '*' expr → ast_bin(AST_MUL, $1, $3)
  - expr '/' expr → ast_bin(AST_DIV, $1, $3)
  - NUMBER → ast_int($1)
  - (expr) → 그냥 안의 expr 트리 그대로 사용
- 이런 식으로 입력 토큰들을 AST 노드들로 조립하는 역할을 함
- yyerror 함수는 파싱 도중 문제가 생겼을 때, 에러 메시지를 출력하는 함수

ast.h / ast.c

식(수식)을 트리로 표현하는 코드

- 이 코드에서는 프로그램 안에서 다루는 식을 추상 구문 트리(AST) 라는 구조로 표현
- ASTType이라는 열거형은 “이 노드가 정수냐, 더하기냐, 빼기냐, 곱하기냐, 나누기냐”를 구분하는 꼬리표 역할을 함
- AST 구조체는 “한 개의 노드”를 나타내는데, 이 노드 안에는 노드 종류(type), 정수일 때의 값(value), 그리고 왼쪽·오른쪽 자식 포인터가 들어 있음
- ast_int(int v) 함수는 “정수 리터럴을 나타내는 AST 노드 하나를 새로 만들어서” 돌려줌
- ast_bin(ASTType t, AST* l, AST* r) 함수는 “왼쪽과 오른쪽에 이미 만들어진 서브트리를 붙인, 이항 연산 노드를” 생성해서 돌려줌
- 파서는 이 함수들을 이용해서, 예를 들어 1 + 2 * 3 같은 입력을 “+ 노드 밑에 1 노드와 * 노드를 두고, 그 * 노드 밑에 2와 3을 두는 트리”로 만들어 둠

codegen.c/codegen.h

AST를 IR로 바꾸는 핵심 재귀 함수

gen(AST* node) 함수는 “이 AST 노드를 계산했더니, 결과가 어느 임시 변수에 들어있다”를 알려주는 역할을 함
즉, 반환값은 **결과를 보관하는 임시 변수 이름 문자열**입니다.

만약 노드 타입이 AST_INT라면,

1. 새 임시 변수 이름을 하나 만들고 (t0 같은),
2. 그 임시 변수에 상수 값을 로드하는 IR_LOADI 명령을 하나 emit
3. 그리고 “결과가 들어있는 변수 이름”으로 그 임시 변수 이름을 반환

반대로, 노드가 +, -, *, / 같은 이항 연산이면,

1. 먼저 왼쪽 자식에 대해 gen을 호출해서, “왼쪽 계산 결과가 들어있는 임시 변수 이름”을 얻고,
2. 오른쪽 자식에 대해서도 gen을 호출해 “오른쪽 결과 변수”를 얻음.
3. 그 다음 새 임시 변수 하나를 만든 뒤,
4. 해당 연산에 맞는 IR 명령(IR_ADD, IR_SUB 등)을 emit 해서 “왼쪽 결과와 오른쪽 결과를 계산한 값을 그 새 임시 변수에 저장해라”라고 기록함
5. 마지막으로, 그 새 임시 변수 이름을 반환해서 “이 노드의 결과는 여기 들어있다”고 알려줌

이렇게 재귀적으로 내려가면서, 트리의 잎(정수)에서부터 루트(마지막 연산)까지 **후위 순회**로 IR을 쌓게 됨

ir.h / ir.c

AST를 3주소 코드 형태의 IR로 바꾼 결과를 저장하는 코드

- “중간 표현(IR, Intermediate Representation)”을 책임짐
- IROp 열거형은 한 줄짜리 IR 명령이 무슨 일을 하는지 나타냄. 예를 들면, IR_LOADI는 “상수 값을 임시 변수에 로드하는 명령”, IR_ADD는 “두 값을 더해서 결과를 어떤 변수에 저장하는 명령”이라는 식으로 저장함
- IR 구조체는 “IR 한 줄”을 표현함. 어떤 연산인지(op), 결과를 저장할 목적지(dst), 첫 번째/두 번째 피연산자(src1, src2), 그리고 다음 IR을 가리키는 next 포인터가 들어 있음. new_temp() 함수는 t0, t1, t2처럼 새로운 임시 변수 이름을 하나씩 만들어 줌. emit(...) 함수는 “지금 설명한 IR 한 줄을 새로 만들어, 리스트의 맨 끝에 붙이는 역할”을 함. 코드 생성기가 이 함수를 계속 호출하면서, 한 줄 한 줄 IR이 쌓임. print_ir() 함수는 리스트에 쌓여 있는 IR들을 사람이 보기 좋은 형태로 예를 들어 t0 = 3 / t1 = t0 * t2 이런 식으로 출력해 줌.
- free_ir() 함수는 IR 리스트를 싹 지우고, 임시 변수 번호도 0으로 초기화해서 “다음 줄(line)을 새로 컴파일할 준비”를 시켜줌

다음 단계:
명령 추가