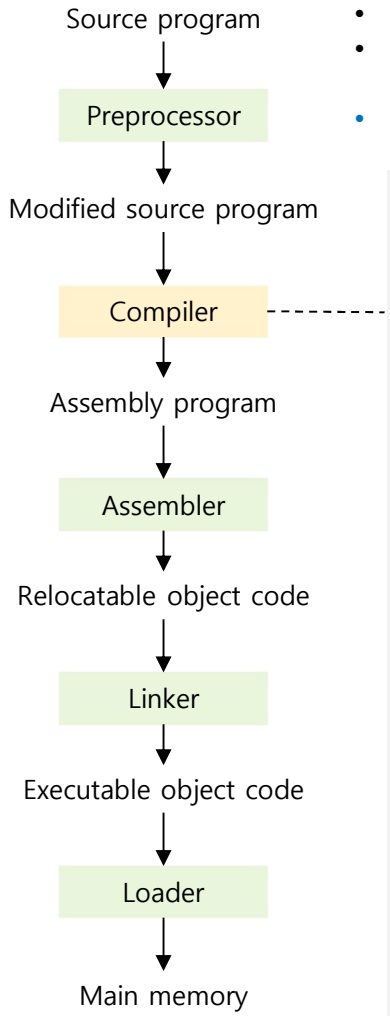
중간표현생성

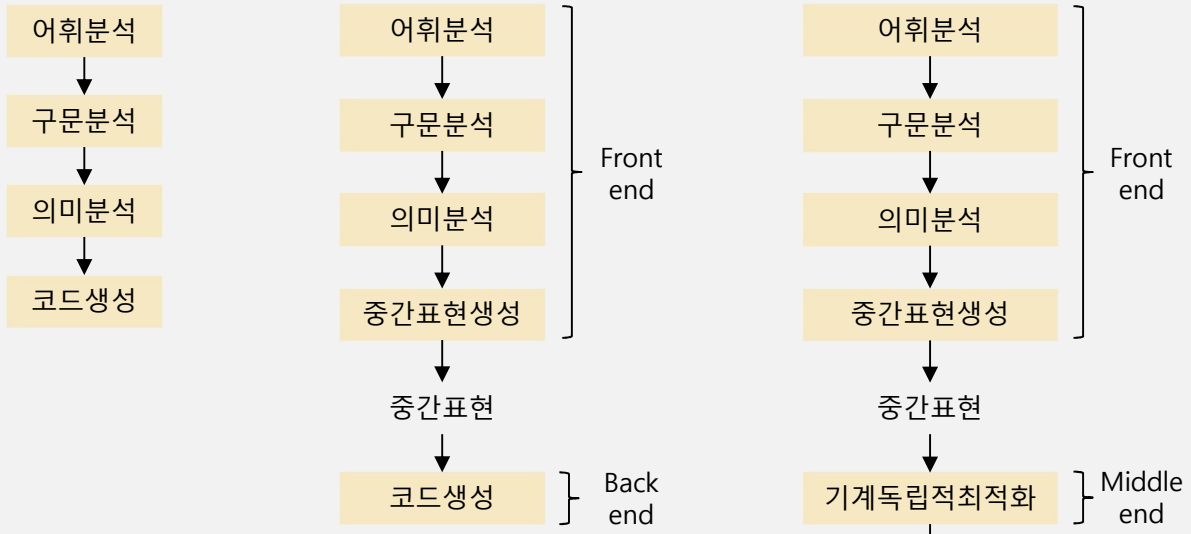
목차

- 중간표현
- TAC(three address code)
- Three address instructions
- TAC 예제

Compiler 개요

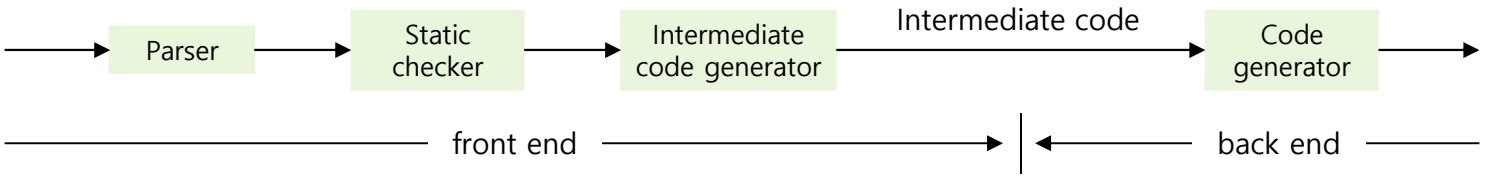


- 어휘분석 → 소스프로그램을 문자 나열로 읽어 들여 의미 있는 어휘 단위인 토큰들로 분할
- 구문분석 → 소스프로그램이 프로그래밍언어의 문법에 맞게 작성되었는지 검사(소스프로그램이 언어의 문법에서 유도되는지 검사하고, 소스프로그램의 유도 트리를 추상구문트리로 변환)
- **의미분석 → 소스프로그램이 의미적으로 오류가 없는지 검사(type checking, name binding 등)**



- n개 서로 다른 CPU(예: x86, ARM, SPARK)에서 동작하는 m개 서로 다른 프로그래밍언어의 컴파일러들을 모두 제작한다고 가정할 때,
- 일체형 컴파일러로 제작한다면 $m \times n$개 컴파일러 제작이 필요하지만,
- 중간표현을 도입하여 front-end와 back-end를 분리하면 총 m개 front-end와 n개 back-end 제작으로 충분

중간표현



Compiler(컴파일러)

- 컴파일러의 analysis-synthesis 모델에서, front end는 소스 프로그램을 분석하여 중간표현을 만들고, back end에서는 중간표현으로부터 목표 코드를 생성 → 이러한 접근법을 통해 m개의 front end와 n개의 back end로 $m \times n$개의 컴파일러 제작이 가능하다

Static checking

- Type checking
- Syntactic check → 예) C에서 break문이 while, for, switch문 내부에 사용되었는가

중간코드

중간코드는 AST, 언어, ByteCode, TAC 등등 다양한 것이 될 수 있다.

Intermediate representation

- 중간표현(예: syntax tree, three-address code, programming language)의 선택은 컴파일러마다 다를 수 있음
- 초기 C++ compiler의 front end는 C 코드를 생성했고, back end로 C compiler 사용
- Three-address code → 예) $x = y\ op\ z$

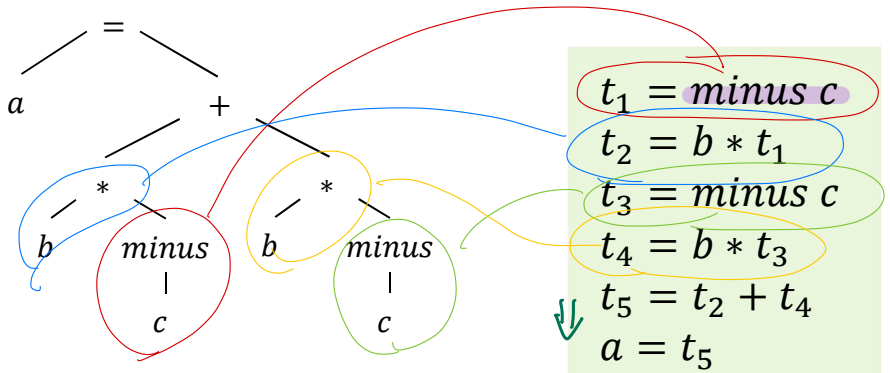
Three-address instructions, quadruples

Three-address instructions

- 컴파일러에서 사용되는 중간언어의 한 종류
- 각 three-address instruction은 최대 3개 피연산자(address)를 갖는 연산에 대응
- Address는 name, constant, compiler-generated temporary 중 하나임. Name은 소스 프로그램 내 name이며, 구현 시 심볼테이블 내 대응하는 entry로의 pointer로 대체

Quadruples

- Three-address instruction은 컴파일러 내부 자료구조에서의 표현을 명시하지 않음
- Quadruple은 three-address instruction에 대응하는 컴파일러 내부 자료구조 표현의 하나임
- 하나의 quadruple은 $op,\ arg_1, arg_2,\ result$의 4개 필드로 구성됨
- $x = y + z$의 경우 op는 $+$이며 assignment(=)는 묵시적 연산임
- Copy statemen인 $x = y$의 경우 op가 $=$이며 arg_2는 미사용됨
- $param\ x$의 경우 $arg_2,\ result$ 미사용됨.
- Conditional jump(조건 분기) 및 unconditional jump(무조건 분기)의 경우 목표 label은 $result$에 저장



$a = b * -c + b * -c$

$$t_1 = minus\ c$$
$$t_2 = b * t_1$$
$$t_3 = minus\ c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

Three address code

TAC 표현

순서

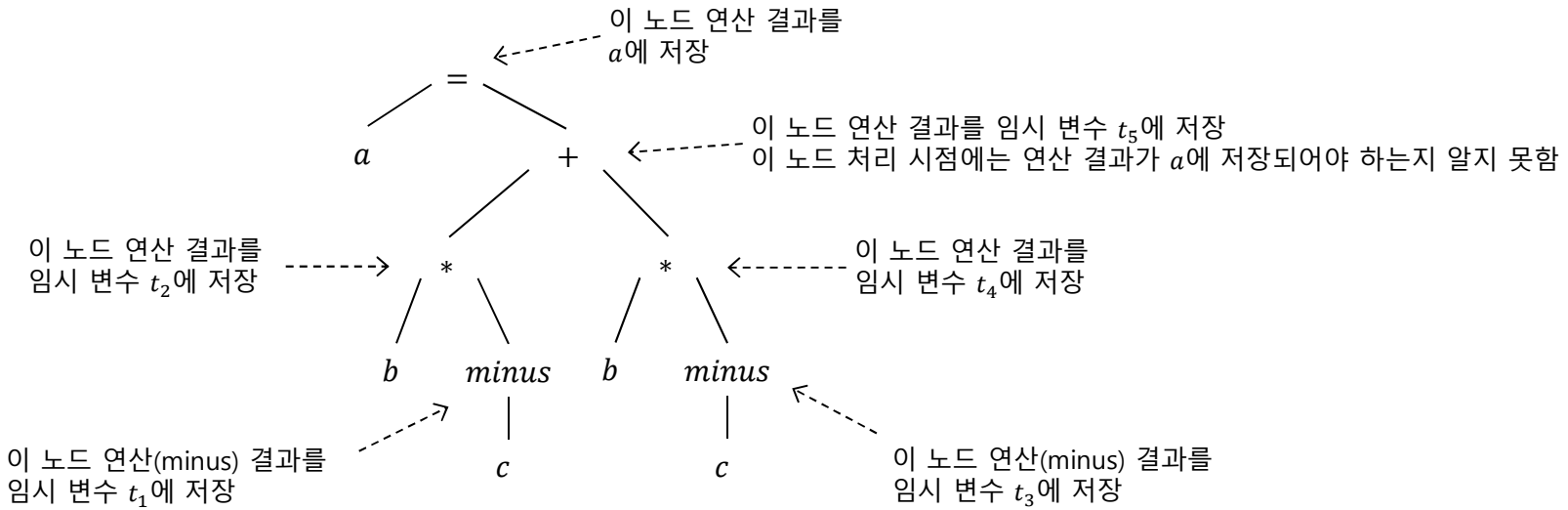
	op	arg_1	arg_2	$result$
0	$minus$	c		t_1
1	$*$	b	t_1	t_2
2	$minus$	c		t_3
3	$*$	b	t_3	t_4
4	$+$	t_2	t_4	t_5
5	$=$	t_5		a

Quadruples

항 네개로 표현하기

Three-address code (TAC)

$$a = b * -c + b * -c$$



$$t_1 = minus\ c$$
$$t_2 = b * t_1$$
$$t_3 = minus\ c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

Three-address code

$a = b * -c + b * -c$

$t_1 = minus\ c$
$t_2 = b * t_1$
$t_3 = minus\ c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

$do\ \ i = i + 1;\ \ while(a[i] < v)\ ;$

$L:\ t_1 = i + 1$
$i = t_1$
$t_2 = i * 8$
$t_3 = a\ [\ t_2\ ]$
$if\ t_3 < v\ goto\ L$

배열의 각 원소가 8개 공간단위(unit of space)를 차지한다고 가정

TAC 명령어 종류

Three-address instructions

Three-address instructions	
$x = y\ op\ z$	op는 이항 산술/논리 연산자이며 x, y, z는 address
$x = op\ y$	op는 단항 연산자(예: unary minus, logical negation, type conversion)
$x = y$	x에 y의 값 대입
$goto\ L$	레이블(label) L에 있는 three-address 명령문이 다음에 실행됨
$if\ x\ goto\ L$	x가 참이면 label L의 명령문 실행하고 거짓이면 순서상 다음 명령문 실행 $ifFalse\ x\ goto\ L$도 가능하며 $if\ x\ goto\ L$의 반대 동작
$if\ x\ relop\ y\ goto\ L$	$x\ relop\ y$가 참이면 label L의 명령문 실행하고 거짓이면 순서상 다음 명령문 실행 $relop$는 관계연산자(예: <, <=, ==)
$param\ x_1$ $param\ x_2$ ... $param\ x_n$ $call\ p, n$	n개 파라미터의 함수 p 호출 $p(x_1, x_1, \ldots, x_n)$에 해당(n은 파라미터 개수로 정수) $y = call\ p, n$도 가능 $return$는 callee에서 caller로 되돌아 감 $return\ y$는 callee에서 caller로 y를 반환
$x = y[i]$ $x[i] = y$	$x = y[i]$ → x에 $y[i]$의 값을 대입 $x[i] = y$ → $x[i]$에 y의 값을 대입
$x = \&y$ $x = *y$ $*x = y$	$x = \&y$ → x의 $r-value$가 y의 $l-value$가 되도록 함 $x = *y$ → x에, y가 point하는 주소에 저장된 값을 대입 $*x = y$ → x가 point하는 주소에 y의 값을 대입

"call func, 3" 이면 파라미터 수가 3개

Three-address code 생성 예시 1

```
{
  int x;
  float y;
  x = 95;
  y = 2.3;
  x = x + 1;
  y = y / 3;
}
```

```
L1:     x = 95
L3:     y = 2.3
L4:     x = x + 1
L5:     y = y / 3
L2:
```

```
{
  int x;
  int y;
  int z;
  int v; int w;
  x = 1; y = 2; z = 3;
  v = x - y * -z + 5 / x;
  w = (x - y) * (-z + 5) / x;
}
```

```
L1:     x = 1
L3:     y = 2
L4:     z = 3
L5:     t1 = minus z
        t2 = y * t1
        t3 = x - t2
        t4 = 5 / x
        v = t3 + t4
L6:     t5 = x - y
        t6 = minus z
        t7 = t6 + 5
        t8 = t5 * t7
        w = t8 / x
L2:
```

```
{
  int x; int y; int z;
  int i; int j; int k;
  int v; int w;
  x = 1; y = 2; z = 3;
  i = 4; j = 5; k = 6;
  v = x - y - z + i / j / k;
  w = x - (y - z) + i / (j / k);
}
```

```
L1:     x = 1
L3:     y = 2
L4:     z = 3
L5:     i = 4
L6:     j = 5
L7:     k = 6
L8:     t1 = x - y
        t2 = t1 - z
        t3 = i / j
        t4 = t3 / k
        v = t2 + t4
L9:     t5 = y - z
        t6 = x - t5
        t7 = j / k
        t8 = i / t7
        w = t6 + t8
L2:
```

Three-address code 생성 예시 2

```
{
  int n; int m;
  n = -5;
  if (n < 0) n = -n;
  m = 2 * n;
}
```

if 문의 예시

```
L1:     n = minus 5
L3:     iffalse n < 0 goto L4
L5:     n = minus n
L4:     m = 2 * n
L2:
```

```
{
  int x; int y;
  x = 0;
  if (x == 10) {
    x = 0;
  }
  y = x;
}
```

```
L1:     x = 0
L3:     iffalse x == 10 goto L4
L5:     x = 0
L4:     y = x
L2:
```

값가 100 하고 같지 않으면 L4로 이동

```
{
  int x; int y; int z;
  x = 1; y = 2;
  x = y - z;
  if (x != 0) {
    z = 3;
    x = x + y * z;
  }
  else {
    z = 5;
    x = x - y - z;
  }
  x = -x;
}
```

```
L1:     x = 1
L3:     y = 2
L4:     x = y - z
L5:     iffalse x != 0 goto L8
L7:     z = 3
L9:     t1 = y * z
        x = x + t1
        goto L6
L8:     z = 5
L10:    t2 = x - y
        x = t2 - z
L6:     x = minus x
L2:
```

값가 0이 아닌지 아닐때 L8로이동

else 문이 있어서 L6로 점프함

반복문의 예시

Three-address code 생성 예시 3

```
{
  int n;
  int sum;
  n = 1;
  sum = 0;
  while ( n <= 10 ) {
    sum = sum + n;
    n = n + 1;
  }
}
```

```
L1:     n = 1
L3:     sum = 0
L4:     iffalse n <= 10 goto L2
L5:     sum = sum + n
L6:     n = n + 1
        goto L4
L2:
```

```
{
  int n;
  int sum;
  n = 1;
  sum = 0;
  while ( n <= 10 ) {
    if ( sum + n > 20 ) break;
    sum = sum + n;
    n = n + 1;
  }
}
```

```
L1:     n = 1
L3:     sum = 0
L4:     iffalse n <= 10 goto L2
L5:     t1 = sum + n
        iffalse t1 > 20 goto L6
L7:     goto L2          — break;
L6:     sum = sum + n
L8:     n = n + 1
        goto L4
L2:
```

```
{
  int n;
  int sum;
  n = 1;
  sum = 0;
  do {
    sum = sum + n;
    n = n + 1;
  } while ( n < 10 );
}
```

```
L1:     n = 1
L3:     sum = 0
L4:     sum = sum + n
L6:     n = n + 1
L5:     if n < 10 goto L4
L2:
```

do - while 은
조건식 검사가 맨에 있음

Three-address code 생성 예시 4

```
{
  int x; int y; int z;
  x = 90; y = 1;
  if ( x >= 90 && y == 1 ) {
    z = 1;
  }
  z = 0;
}
```

AND - OR 이 없는 조건문

```
L1:     x = 90
L3:     y = 1
L4:     iffalse x >= 90 goto L5
        iffalse y == 1 goto L5
L6:     z = 1
L5:     z = 0
L2:
```

```
{
  int x; int y; int z;
  x = 90; y = 1;
  if ( x >= 90 ) {
    if ( y == 1 ) {
      z = 1;
    }
  }
  z = 0;
}
```

```
L1:     x = 90
L3:     y = 1
L4:     iffalse x >= 90 goto L5
L6:     iffalse y == 1 goto L5
L7:     z = 1
L5:     z = 0
L2:
```

```
{
  int x; int y; int z;
  x = 90; y = 1;
  if ( x >= 90 || y == 1 ) {
    z = 1;
  }
  z = 0;
}
```

OR

```
L1:     x = 90
L3:     y = 1
L4:     if x >= 90 goto L7
        iffalse y == 1 goto L5
L7:L6:  z = 1
L5:     z = 0
L2:
```

Three-address code 생성 예시 5

```
{
  int i ;
  int [10] v ;
  v[5] = 111 ;
  v[9] = v[5] + 1 ;
}
```

```
L1:     t1 = 5 * 4
        v [ t1 ] = 111
L3:     t2 = 9 * 4
        t3 = 5 * 4
        t4 = v [ t3 ]
        t5 = t4 + 1
        v [ t2 ] = t5
L2:
```

```
{
  int i ; int sum ;
  int [10] v ;
  i = 0 ;
  while ( i < 10 ) v[i] = i ;
  i = 0 ; sum = 0 ;
  while ( i < 10 ) {
    sum = sum + v[i] ;
  }
}
```

```
L1:     i = 0
L3:     iffalse i < 10 goto L4
L5:     t1 = i * 4
        v [ t1 ] = i
        goto L3
L4:     i = 0
L6:     sum = 0
L7:     iffalse i < 10 goto L2
L8:     t2 = i * 4
        t3 = v [ t2 ]
        sum = sum + t3
        goto L7
L2:
```

```
{
  int i ; int j ;
  int [2][3] v ;
  v[1][2] = 111 ;
  i = 1 ;
  j = 2 ;
  v[i][j] = 111 ;
}
```

```
L1:     t1 = 1 * 12
        t2 = 2 * 4
        t3 = t1 + t2
        v [ t3 ] = 111
L3:     i = 1
L4:     j = 2
L5:     t4 = i * 12
        t5 = j * 4
        t6 = t4 + t5
        v [ t6 ] = 111
L2:
```

References

- 박두순. (2016). (내공 있는 프로그래머로 길러주는)컴파일러의 이해. 한빛아카데미.
- 창병모. (2021). 프로그래밍 언어론 : 원리와 실제. 인피니티북스.
- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Ed.). Addison Wesley.
- Nystrom, R. Crafting interpreter. https://craftinginterpreters.com/
- Sebesta, R. (2012). Concepts of Programming Languages (10th. ed.). Pearson.
- Thain, D. (2023). Introduction to Compilers and Language Design.
- Paxson, V. (1995). Flex, version 2.5.
- Donnelly, C., Stallman, R. (2008). Bison.
- LexAndYacc.pdf (epaperpress.com)
- Tom Niemann, LEX & YACC. http://epaperpress.com/lexandyacc