Compiler Project

2016312021 문태의

2016313856 성아영

# 입력 언어의 특성

입력 언어가 다음의 특성을 가진다고 간주하고 컴파일러를 제작하였다.

* 정적 유효 범위 (Static Scope)
* 정적 타입 언어 (Static Type Language)

# Scanner

어휘 분석 단계를 담당한다. 소스 코드를 여러 개의 Token으로 쪼개어 리스트에 저장한 후, 이 데이터를 Parser에게 넘겨준다.

## Token

각 토큰은 다음의 정보를 저장한다.

* 토큰 종류
* 토큰에 대응하는 소스 코드 상의 문자열
* 토큰이 위치한 소스 코드 상의 줄 번호
* 토큰이 속한 가장 가까운 블록 번호

### 토큰 종류

토큰 종류는 prog, decl, word 등의 문법 상의 기호를 뜻한다.

### 토큰에 대응하는 소스 코드 상의 문자열

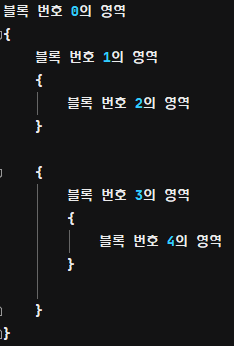
코드와 직접 대응되는 토큰들만 갖는 정보이다. 예를 들어, word의 경우 변수의 식별자를 갖는다.

### 토큰이 위치한 소스 코드 상의 줄 번호

코드와 직접 대응되는 토큰들만 갖는 정보이다. 줄 번호는 컴파일 에러 발생 시 해당 위치를 표시하기 위해 사용된다.

### 토큰이 속한 가장 가까운 블록 번호

블록 번호란, 중괄호 영역이 생길 때 마다 0부터 순서대로 번호를 지정한 것이다. 그림으로 설명하면 다음과 같다.



이를 이용하여 식별자들에게 블록 번호를 저장하여 기호표에서 대응되는 변수를 찾을 수 있다.

## 에러 처리

Scanner에서 검사하는 컴파일 에러는 다음과 같다.

* 소스 코드에 인식할 수 없는 문자열이 있을 때
* 좌측, 우측 중괄호 개수가 일치하지 않을 때

### 소스 코드에 인식할 수 없는 문자열이 있을 때

문법에 정의되지 않은 문자열 패턴이 있을 때, 컴파일 에러를 발생시키고 해당 문자열의 위치를 표시한다.

### 좌측, 우측 중괄호 개수가 일치하지 않을 때

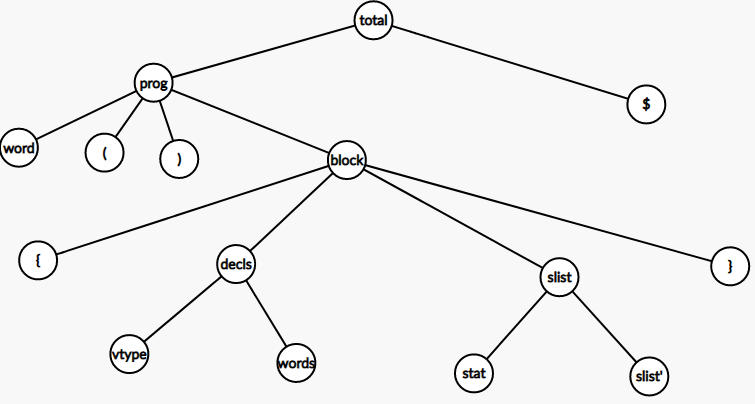
좌측과 우측 중괄호 개수가 일치하지 않을 때 컴파일 에러를 발생시키고 해당 에러 내용을 출력한다.

# Parser

구문 분석 단계를 담당한다. Scanner로부터 받은 Token 리스트를 바탕으로 소스 코드가 Parsing Table의 문법에 대응하는지 확인하고, Syntax Tree와 Symbol Table을 만든다. 분석 방법은 LL Parser의 원리를 사용한다.

## Syntax Tree

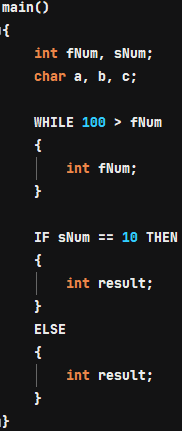
Syntax Tree는 Parsing Table를 바탕으로 작성된 트리이다. 생성된 트리 예시는 다음과 같다.



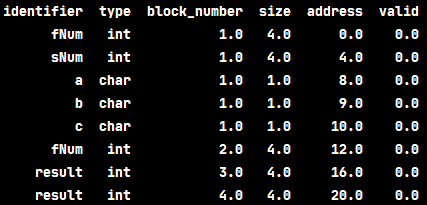
## Symbol Table

Symbol Table은 식별자, 타입, 블록 번호, 크기, 주소, valid를 저장하며, Parser에 의해 Symbol이 추가될 때 자동으로 주소를 할당한다. 주소를 할당하는 원리는 0x00000000부터 시작하여 순서대로 할당한다. int의 경우 4씩 띄어서, char의 경우 1씩 띄어서 할당하되, int는 반드시 4의 배수의 주소에 할당되도록 하였다. 블록 번호가 다른 곳에서 선언된 변수는 다른 변수로 간주한다.

valid는 변수에 값이 할당되었는지 확인하는 값이다. valid가 1이면 변수가 초기화되었다는 뜻으로, 변수의 값을 가져올 수 있다. Symbol Table 작성 예시는 다음과 같다.



<소스 코드 예시>



<Symbol Table 예시>

## Parsing Table

일부 문법이 수정되었다.

### 결과는 같으나 LL Parser에 적용하기 위해 추가 및 수정된 문법들

* (수정) decls → decls decl |
* decls → decl decls |
* (수정) words → words “,” word
* words → word words’
* (추가) words’ → “,” word words’ |
* (변경) slist → slist stat | stat
* slist → stat slist |
* (변경) stat → IF cond THEN block ELSE block

| WHILE cond block

| word “=” expr “;”

| RETURN expr “;”

|

* stat → IF cond THEN block ELSE block

| WHILE cond block

| word “=” expr “;”

| RETURN expr “;”

* (변경) cond → expr “>” expr | expr “==” expr
* cond → expr expr’
* (추가) expr’ → “==” expr | “>” expr
* (변경) expr → term | term “+” term
* expr → term term’
* (추가) term’ → “+” term |
* (변경) term → fact | fact “\*” fact
* term → fact fact’
* (추가) fact’ → “\*” fact |

### 결과가 수정된 문법

block → “{“ decls slist “}”

해당 문법에서 input으로 word를 받았을 때, decl에서 vtype이 없는 변수 정의인지,

stat에서 word “=” expr “;” 형태의 문법인지 구분할 수 없어서 다음의 문법을 수정하였다.

* (변경) vtype → int | char |
* vtype → int | char

따라서 데이터 타입이 명시되지 않은 변수 선언은 불가능하다.

### 최종 문법

prog → word “(“ “)” block

decls → decl decls |

decl → vtype words “;”

words → word words’

words’ → “,” word words’ |

vtype → int | char

block → { decls slist }

slist → stat slist |

stat → IF cond THEN block ELSE block

| WHILE cond block

| word “=” expr “;”

| RETURN expr “;”

cond → expr expr’

expr’ → “==” expr | “>” expr

expr → term term’

term’ → “+” term |

term → fact fact’

fact’ → “\*” fact |

fact → num | word

num → [0-9][0-9]\*

word → ([a-z] | [A-Z])([a-z] | [A-Z])\*

## 에러 처리

Parser에서 검사하는 컴파일 에러는 다음과 같다.

* 세미 콜론이 있어야할 위치에 없을 때
* 변수 중복 선언

### 세미 콜론이 있어야할 위치에 없을 때

세미 콜론이 있어야할 위치에 없는 경우 컴파일 에러를 발생시키고 세미 콜론이 있어야 할 위치를 표시한다.

### 변수 중복 선언

같은 블록 안에 같은 식별자의 변수가 선언된 경우 컴파일 에러를 발생시키고 중복된 변수가 선언된 위치를 표시한다.

# Code Generator

최적화된 Syntax Tree로부터 Code Table을 작성하고, 해당 테이블을 Target 파일에 작성한다.

## Instruction set

### Instruction SEt 추가

메모리로부터 값을 가져올 때, word단위로 가져올 것인지(int형일 때) byte단위로 가져올 것인지(char형일 때) 구분이 필요하여 다음의 instruction을 추가하였다.

|  |  |
| --- | --- |
| LDB Reg#1, addr | addr 주소에서 byte 데이터를 Reg#1에 불러옵니다 |
| STB Reg#1, addr | Reg#1의 byte 데이터를 addr 주소에 저장합니다. |

## Code Table

three address의 quadruples형식으로, instruction, arg1, arg2, result 순서로 저장한다.

변수의 주소를 파악해야 할 때는 Symbol Table을 사용한다. Syntax Tree로부터 토큰들을 분석하여, 좌측 중괄호를 만날 때 마다 블록 번호를 스택에 쌓고, 우측 중괄호를 만날 때마다 스택에서 하나씩 꺼낸다. 이 스택과 식별자를 Symbol Table에 넘겨주면 해당 변수의 주소를 알려준다.



해당 예시 코드에서 블록 번호 1에 a라는 이름의 변수가 선언되어 있고, 블록 번호 4에 변수 a를 사용하고 있다. 이때 블록 번호 4에서 블록 번호 스택은 0, 1, 3, 4이다. 이 스택을 Symbol Table에 넘겨주면 4→3→1→0순서로 식별자 a에 대한 주소를 찾는다. 블록 번호 3, 4에는 a의 이름으로 정의된 변수가 없으므로, 블록 번호 1에 정의된 변수 a의 주소를 넘겨준다.

### Code 작성 방식

#### WHILE문

loop<number>:

cond 연산 -> 결과 R0에 저장

JUMPF R0 endloop<number>

블록 코드

Jump loop<number>

endloop<number>:

#### IF문

cond 연산 -> 결과 R0에 저장

JUMPF R0 else<number>

True인 경우 실행할 블록 코드

Jump endif<number>

else<number>:

False인 경우 실행할 블록 코드

endif<number>:

#### 대입 연산

expr 연산 -> 결과 R0에 저장

ST R0, addr

#### RETURN 문

expr 연산 -> 결과 R0에 저장

ADD RV, ZERO, R0

#### cond 연산

앞 expr 연산 -> 결과 R0에 저장

ADD R1, ZERO, R0

뒤 expr 연산 -> 결과 R0에 저장

(조건 연산자가 >일 때)

LT R0, R1, R0

(조건 연산자가 ==일 때)

LD R2, -1

MUL R0, R2, R0

ADD R0, R1, R0

LD R2, 1

LT R0, R0, R2

#### expr 연산

예시) 0x00000000 \* 0x00000004 + 0x00000008 \* 0x0000000C

LD R0 0x0000000

LD R1 0x00000004 (앞 곱셈 없는 경우 생략)

LD R2 0x00000008 (덧셈 없는 경우 생략)

LD R3 0x0000000C (뒤 곱셈 없는 경우 생략)

MUL R0, R0, R1 (앞 곱셈 없는 경우 생략)

MUL R2, R2, R3 (뒤 곱셈 없는 경우 생략)

ADD R0, R0, R2 (덧셈 없는 경우 생략)

(타입이 char인 변수는 LD를 LDB로 수정)

### 레지스터 사용

사용하는 레지스터는 연산을 위해 R0, R1, R2, R3 레지스터와return 값 전달을 위해 사용되는 RV 레지스터, 항상 0이 저장되어 있는 ZERO레지스터로 총 5개의 레지스터를 사용한다.

## 에러 처리

Code Geneartor에서 검사하는 컴파일 에러는 다음과 같다.

* 선언되지 않은 변수 사용
* 초기화되지 않은 변수 사용

### 선언되지 않은 변수 사용

static scope를 기준으로 접근할 수 있는 변수가 없으면 변수가 선언되지 않았다고 간주하고 컴파일 에러를 발생시키며 해당 변수가 사용된 위치를 표시한다.

### 초기화되지 않은 변수 사용

접근한 변수에 해당하는 Symbol Table의 valid값이 0(False)일 때, 해당 변수가 초기화되지 않았다고 간주하고 컴파일 에러를 발생시키며, 해당 변수가 사용된 위치를 표시한다.