프로젝트 #2

컴퓨터학부 컴파일러

2024년 10월 21일

문제

파서 자동생성 도구인 Bison을 사용하여 COOL 컴파일러 구성 요소 중 두 번째 단계인 구분분석기 (syntactic analyzer)를 구현한다.

COOL 문법

COOL의 문법은 아래와 같다. 여기서 굵은체는 키워드, 이탤릭체는 논터미널(non-terminal)을 의미한다.

program	::=	⟦*class*;⟧$^+$
class	::=	**class** TYPE [**inherits** TYPE] {⟦*feature*;⟧*}
feafure	::=	ID([*formal*⟦, *formal*⟧*]) : TYPE {*expr*}
	|	ID : TYPE [<- *expr*]
formal	::=	ID : TYPE
expr	::=	ID <- *expr*
	|	*expr*[@TYPE].ID([*expr*⟦, *expr*⟧*])
	|	ID([*expr*⟦, *expr*⟧*])
	|	**if** *expr* **then** *expr* **else** *expr* **fi**
	|	**while** *expr* **loop** *expr* **pool**
	|	{⟦*expr*;⟧$^+$}
	|	**let** ID : TYPE [<- *expr*] ⟦, ID : TYPE [<- *expr*]⟧* **in** *expr*
	|	**case** *expr* **of** ⟦ID : TYPE => *expr*;⟧$^+$ **esac**
	|	**new** TYPE
	|	**isvoid** *expr*
	|	*expr* + *expr*
	|	*expr* - *expr*
	|	*expr* * *expr*
	|	*expr* / *expr*
	|	~*expr*
	|	*expr* < *expr*
	|	*expr* <= *expr*
	|	*expr* = *expr*
	|	**not** *expr*
	|	(*expr*)
	|	ID
	|	integer
	|	string
	|	**true**
	|	**false**

Bison 설치

Bison은 C/C++ 환경에서 가장 많이 사용하는 파서 생성 도구로 주어진 문법(context-free grammar)을 인식하는 LR 파서를 생성한다. 프로젝트를 수행하기 전에 먼저 Bison를 설치한다.

- **Linux** 환경에서는 터미널을 열고 다음 명령어를 실행한다.

```
% sudo apt update
% sudo apt install bison
```

  설치후 `bison` 명령어를 사용하여 COOL언어 토큰, 문법, 의미행위(semantic action)가 정의된 `cool.y`를 컴파일하면 파서 소스파일 `cool.tab.c`와 헤더파일 `cool.tab.h`가 생성된다. 이 헤더 파일은 어휘분석기에 필요하므로 `bison`을 실행하고 난 다음에 `flex`를 실행한다. COOL언어 어휘분석 규칙이 정의된 `cool.l` 파일을 `flex`로 컴파일하면 `lex.yy.c`가 생성된다. 소스를 모두 묶으면 파서 실행파일을 만들 수 있다.

```
% bison -d cool.y
% flex cool.l
% gcc -o cool_parser lex.yy.c cool.tab.c -ll
```

- **macOS** 환경에서는 기본적으로 내장되어 있어서 별도의 설치가 필요없다. Flex와 Bison를 사용하여 구문분석기를 생성하는 과정은 Linux와 같다.

파서 구현

이번 과제에서 요구하는 파서 구현은 크게 두 단계로 나눌 수 있다. 첫 번째 단계는 구문오류만 검사하는 것이고, 다음 단계는 추상구문트리(Abstract Syntax Tree; AST)를 생성하고 출력하는 것이다.

- **구문오류 검사**: Bison 소스파일인 `cool.y`에 COOL언어 토큰을 명시하고, 문법을 BNF (Backus-Naur Form)으로 정의한다. 올바르게 정의하였다면 COOL 프로그램이 문법적으로 맞는지 틀리는지 검사할 수 있는 파서를 만들 수 있다. 제공된 `Makefile`을 사용하여 실행파일 `cool_parser`를 생성한다. COOL언어로 작성된 `bad.cl` 프로그램은 다수의 문법적 오류가 들어 있다. 생성된 파서가 `bad.cl`의 구문오류를 제대로 검사하는지 실행해본다.

```
% ./cool_parser bad.cl
```

- **추상구문트리 생성**: 추상구문트리를 만들기 위해서는 Bison 소스인 `cool.y` 내에 의미행위를 정의해야 한다. 트리를 만들기 위해서는 노드 생성이 필요한데, 이에 필요한 모든 타입과 함수를 `node.h`와 `node.c`에 정의한다. 파싱 중 구문오류가 발생하면 오류 메시지를 출력하고 구문오류 없이 파싱이 끝나면 추상구문트리를 출력하고 종료한다. 파서가 `good.cl`의 추상구문트리를 제대로 생성하는지 실행해본다.

```
% ./cool_parser good.cl
```

추상구문트리 출력

파싱이 끝나고 오류가 없으면 추상구문트리를 출력한다. 트리의 출력은 루트부터 깊이우선(depth-first) 방식으로 재귀적으로 출력한다. 다만 구조상 출력하지 않아도 알 수 있는 일부 키워드와 구두점은 생략한다. 트리를 구성하는 노드의 유형에 따라 다음과 같은 방식으로 출력한다.

- ‘,’, ‘:’, ‘;’: 쉼표, 콜론, 세미콜론은 출력하지 않는다.
- `TYPE`: 모든 `TYPE`은 문자열 양쪽에 대괄호를 붙여 출력한다.
- `ID`: 모든 `ID`는 문자열 그대로 출력한다.
- *class*: 키워드 `class`와 `inherits`는 출력하지 않는다. 클래스가 끝나면 다음 클래스를 시작하기 전에 줄바꿈한다.
- *feature*: 양쪽에 중괄호를 붙여 출력한다.
- *formal*: 앞에서 언급한 규칙에 따라 `ID[TYPE]` 형식으로 출력한다.

- '<-', '=>': 생략하지 않고 그대로 출력한다.
- dispatch expr: 앞에서 언급한 규칙에 따라 출력하되 '@'과 '.'는 생략하지 않는다.
- if expr1 then expr2 else expr3 fi: `(if expr1 expr2 expr3)` 형식으로 출력한다.
- while expr1 loop expr2 pool: `(while expr1 expr2)` 형식으로 출력한다.
- {expr1; expr2; expr3; ...}: `(expr1 expr2 expr3 ...)` 형식으로 출력한다.
- let ID-list in expr: `(let (ID-list) expr)` 형식으로 출력한다.
- case expr of CASE-list esac: `(case expr (CASE-list))` 형식으로 출력한다.
- op expr: 단항 연산식은 프리오더(preorder) 형식인 `(op expr)`으로 출력한다.
- expr1 op expr2: 이항 연산식도 프리오더 형식인 `(op expr1 expr2)`으로 출력한다.
- string: 큰따옴표가 붙은 문자열로 출력한다.
- integer: 정수값으로 출력한다.
- true, false: 그대로 출력한다.

골격 파일

이 프로젝트를 수행하는데 필요한 `Makefile`, `cool.skeleton.l`, `cool.skeleton.y`, `node.h`, `node.c` 파일을 학생에게 제공한다. `cool.skeleton.l`은 Flex로 작성된 `cool.l`의 골격파일로 어휘분석의 규칙을 정의하는 곳이다. `cool.skeleton.y`는 Bison으로 작성된 `cool.y`의 골격파일로 구문분석의 규칙을 정의하는 곳이다. 학생들은 이 파일에 필요한 규칙과 행위를 넣음으로써 구문분석기를 완성한다. `node.h`와 `node.c`는 추상구문트리를 만들 때 필요한 데이터 타입과 함수를 정의하는 곳이다. 이 부분은 각자가 설계한 방식에 따라 필요한 코드를 넣는다. `Makefile`은 모든 소스파일을 컴파일하여 실행파일 `cool_parser`를 생성한다.

검증

구문분석기를 검증하는데 필요한 `bad.cl`, `bad.cl.out`, `good.cl`, `good.cl.out`, `chk_examples` 파일과 다수의 COOL 프로그램이 들어 있는 `examples\` 폴더도 골격파일과 함께 제공한다. 구문분석기의 첫 번째 단계가 완성되면 올바르게 동작하는지 주어진 샘플파일인 `bad.cl`을 실행해본다.

```
% ./cool_parser bad.cl
syntax error in line 9 at "int"
syntax error in line 9 at "string"
...
syntax error in line 40 (unexpected EOF)
14 error(s) found
```

이 샘플파일은 다수의 문법적 오류가 포함되어 있는 COOL 프로그램으로 구문분석기의 버그를 빨리 찾기 위해서 작성되었다. 이 샘플파일의 실행결과와 `bad.cl.out`과 비교해본다. 꼭 일치하지 않아도 되지만 최대한 많은 오류를 찾아낼 수 있도록 한다. 오류의 개수에서 차이가 나는 이유는 Bison의 `error` 프로덕션을 어떻게 활용하느냐에 따라 그 결과가 다르기 때문이다. 만족할 만한 결과가 나올 때까지 다양한 방식으로 시도해본다.

구문분석기의 두 번째 단계인 추상구문트리 생성이 완성되면 제대로 출력이 되는지 주어진 샘플파일인 `good.cl`을 실행해본다.

```
% ./cool_parser good.cl
[A]
[B][C]
[C][A]{x[Int]<-(~ 35)}{y[B]<-(new B)}{z[Str]<-"hello"}
...
[H]{foo()[Int](case (+ a (* b c))(x[Int]=>a y[Int]=>(+ a b) z[Int]=>(+ a (* b c))))}
```

이 샘플파일에는 간단한 클래스부터 점진적으로 복잡한 클래스가 정의되어 있다. 추상구문트리를 단계적으로 복잡하게 만들어서 구문분석기의 버그를 찾을 수 있게 하기 위함이다. 이 샘플파일의 실행결과와 good.cl.out과 비교해본다. 출력은 화이트 스페이스를 제외하고 나머지는 정확하게 일치해야 한다. 만일 일치하지 않으면 추상구문트리가 잘못 되었거나 출력형식을 따르지 않았기 때문이다. 일치하도록 수정한다.

마지막으로 examples\ 폴더에 있는 COOL 프로그램을 실행해본다. 실행결과는 동일한 폴더 안에 있는 COOL파일명.out과 화이트 스페이스를 제외하고 정확하게 일치해야 한다. chk_examples는 셸스크립트로 실행과 검증과정을 자동으로 수행한다.

```
% ./chk_examples
examples/arith.cl --> PASSED
examples/atoi.cl --> PASSED
...
examples/sort_list.cl --> PASSED
```

제출물

컴파일 과정과 실행결과를 보여주는 화면캡쳐, 소스코드, 그리고 자신의 결과를 보여주는데 필요한 부가 설명이나 기타 자료를 스스로 판단하여 제출한다. 모든 자료에는 학번과 이름을 명시하고, 소스코드를 제외한 나머지 제출물은 PDF 형식이어야 한다. 여기에는 다음과 같은 것이 반드시 포함되어야 한다.

- 컴파일(make)과 실행(chk_examples) 과정을 보여주는 화면 캡처
- 실행 결과에 대한 설명, 소감, 문제점
- 소스파일 (cool.l, cool.y, node.h, node.c) 별도 제출
- 샘플 프로그램 실행 결과 (bad.cl.txt, good.cl.txt) 별도 제출

HK