컴공423
Compiler
Final Report

컴퓨터공학과
9725018
오태호

1998년 6월 17일

1 Symbol Table

symbol table에는 현재 어떤 symbol이 어떠한 형태고 어디에 저장되어 있는지를 관리한다.

1.1 Symbol Table에 저장되는 내용

현재 symbol table에 저장되는 내용은 lexeme, token, offset, global이다. 각각의 기능은 다음과 같다. STEntry.cc 를 보면 symbol table에 어떻게 저장되는지 알 수 있다.

- lexeme
  "result"와 같이 변수명 이나 함수명이 저장된다.
- token
  현재 symbol table에 넣는 symbol이 어떤 형인지 저장한다. INTTYPE, ARRAYTYPE 등이 있다. int result와 같이 선언을 하게 되면 "result"가 INTTYPE이라는 것이 저장된다.
- offset
  현재 이 symbol이 메모리상에 어디에 위치하게 되는지 저장한다. 하지만 이것은 컴파일시에 결정하는 것이 불가능하기 때문에 frame pointer로부터 상대위치를 offset에 저장한다. 나중에 참조할 때는 현재의 frame pointer와 symbol table에 저장되는 offset의 값을 더해서 위치를 알 수 있다.
- global
  현재 이 symbol이 global인지 local인지를 저장한다. global이면 1을 저장하고 local이면 0을 저장한다. 이렇게 global과 local을 나누는 이유는 local과 같은 경우에는 현재의 frame pointer로부터 위치를 구하면 되지만 global과 같은 경우에는 frame pointer로부터 위치를 구하는 것이 불가능하고 memory top이나 memory bottom 으로부터의 위치를 저장해 놔야 되기 때문이다.

1.2 Symbol Table의 관리

symbol table은 SymbolTable.cc 에서 관리한다. lexeme을 Hash.cc를 통해 hash한 다음 저장하게 된다. insert와 lookup method를 제공하며 insert를 할 때는 위의 4개를 지정해서 insert를 하고 lookup을 하게 되면 STEntry의 pointer가 return이 된다.

1.3 Symbol Table에서 Scope의 구현

c-에서 symbol table은 scope 처리를 할 수 있어야 한다. 처음에는 '{'를 만났을 때 scope를 하나 증가시키고 '}'를 만났을 때 scope를 하나 감소시키려고 하였지만 function을 선언할 때 parameter는 '{'의 바로 직전에 나옴에도 불구하고 '{' 다음에 나오는 symbol과 동일한 scope에 속해야 되기 때문에 그렇게 구현하지 못했다. 대신에 function의 선언, if문의 시작, else문의 시작, while문의 시작 부분을 만나면 scope를 하나 증가시키고 function의 선언의 끝, if문의 끝, else문의 끝, while문의 끝을 만나면 scope를 다시 감소시키도록 구현하였다. scope의 구현은 SSymbolTable.cc 에 되어 있다. operator를 overloading을 해서 ++는 scope의 증가 −는 scope의 감소를 하도록 overloading을 했다. SSymbolTable.cc에서 scope를 증가시키면 새로운 symbol table이 하나 생성되고 scope를 감소시키면 현재의 symbol table이 사라진다. symbol table에 값을 넣을 때는 global인가 local인가를 insert해야 되는데 이것은 SSymbolTable.cc에서 자동적으로 해 준다. 현재의 scope가 0이면 global이고 1이상아면 local이기 때문이다. SSymbolTable.cc 에도 SymbolTable.cc와 마찬가지로 insert와 lookup이 있다.

2 Lexical Analyzer

lexical analyzer는 lex의 gnu버전인 flex를 사용했다. flex에서 사용하는 파일은 c-lex.l이다. 처음에는 yylval에 attribute를 저장하고 token을 return하는 방법을 사용했는데 parser를 구현하는데 어려움이 발견되서 yylval을 사용하지 않고 모두 각각을 따로따로 token을 return하도록 구현하였다. return하는 token은 ELSE, IF, INT, RETURN, VOID, WHILE, SPARENL, SPARENR, MPARENL, MPARENR, BPARENL, BPARENR, ID, NUMBER, LT, LE, GT, GE, EQ, NE, PLUS, MINUS, MULT, DIVI, ASSOP, SEMICLN, COMMA가 있다. lexical analyzer를 만드는데는 그리 여려움이 없었지만 comment처리가 쉽지 않았다. comment를 다음과 같은 expression을 사용해서 찾아냈다.

```
"/*"([^\*\/]|\*[^/]|[^\*]\/)*"*/"
```

위와같이 하면 comment를 처리할 수 있다. /*와 */ 사이에 나오는 모든 것을 없애면 될 것이라고 찾각할 수도 있겠지만 flex는 /*가 나오고 제일 마지막에 나오는 */사이에 있는 모든 것을 없애 주기 때문에 의도하지 않는 방향으로 작동할 수 있기 때문이다.

3 Syntax Analyzer

syntax analyzer는 yacc의 gnu버전인 bison을 사용했다. bison에서 사용하는 파일은 c-yacc.y이다. syntax analyzer에서는 c- 의 문법에 맞는지 검사를 하고 parse를 하게 된다. 그리고 가장 중요한 일은 syntax tree를 만드는 것이다. syntax tree를 만들지 않고 바로 semantic action에 목적코드를 넣도록 구현할 수도 있지만 그렇게 하게 되면 구현이 쉽지 않고 bison의 사용에 익숙하지 않을 경우에는 debugging이 쉽지 않아서 syntax tree를 만들도록 하였다.

4 Syntax Tree

syntax tree는 현재 소스코드의 구조를 분석해서 이를 tree형태로 만든 것이다. syntax tree는 목적코드를 만들기 위해 반드시 필요한 것은 아니지만 syntax tree를 만들게 되면 훨씬 편하게 목적코드를 생성해 낼 수 있다.

4.1 Syntax Tree Node에 저장되는 내용

syntax tree는 c-yacc.y에 만드는 방법이 정의되어 있다. Node.cc에 보면 syntax tree의 node의 형태를 볼 수 있다. syntax tree node에 들어가는 내용은 다음과 같다.

- type
  node가 어느 부분에서 parse되었는지 나타낸다. 예를 들어 fundeclaration → typespecifier ident SPARENL params SPARENR compoundstmt 와같이 되었을 때 type에 6을 저장한다. 이와같이 parse가 될 때 어느 부분이 parse되어서 생겨난 node인지를 type에 저장한다.

- token
  token에는 node에 INT, VOID와 같이 token을 저장할 때 사용한다.

- value
  value는 token이 NUMBER일 때 숫자를 저장할 때 사용한다.

- string
  string에는 token이 ID일 때 현재 lexeme를 자장할 때 사용한다. 이것은 나중에 symbol table에서 lexeme를 이용해 entry를 찾을때 사용하게 된다.

4.2 Syntax Tree의 구조

syntax tree는 child와 sibling을 가질 수 있다. child관계는 포함관계에 있을 때 해당되고 sibling관계는 서로 대등한 관계에 있을 때 해당된다. fundeclaration → typespecifier ident SPARENL params SPARENR compoundstmt 와 같은 경우에는 fundeclaration이 typespecifier, ident, params를 가지고 되고 params는 여러개의 params중 한개의 param만을 가진다. 그리고 param 들은 전부 sibling으로 연결되게 된다.

5 Type Check

compile을 정상적으로 수행하기 위해서는 일단 syntax상에 error가 있는지 확인해야 되고 type을 맞게 사용하고 있는지 확인해 봐야 된다. 그리고 code를 generate할 때는 현재 source코드가 에러가 전혀 없다는 가정하에 작업을 할 수 있기 때문에 code를 generate하는데 있어서 훨씬 수월하게 된다. TypeCheck.cc를 보면 type check하는 것을 자세히 알 수 있다. void a와 같은 선언이 불가능해야 하고 array형에 직접 int형을 assign하는 것이 불가능해야 하는등 여러가지를 check해줘야 된다. type check은 syntax를 recursive하게 traverse하면서 검사를 하도록 했다. c-yacc.y 안에 직접 적어 넣을 수도 있겠지만 그렇게 하면 debugging이 힘들어 지고 구조가 복잡해질 우려가 있어서 syntax tree를 traverse하였다.

6 Tiny Machine분석

Tiny Machine(이하 tm)은 580줄정도 되는 아주 간단한 machine이다. tm의 특징을 대충 알아보면 다음과 같다.

- 8개의 register
  0 ~7번 register가 있으며 7번 register는 program counter이다.
- 1024개의 int형 data memory
- 1024 * 4개의 int형 instruction memory
- 7개의 RO(Register Only) instruction
  HALT, IN, OUT, ADD, SUB, MUL, DIV가 있다.
- 10개의 RM(Register Memory) instruction
  LD, ST, LDA, LDC, JLT, JLE, JGT, JGE, JEQ, JNE가 있다.

tm에서 작동하도록 compile을 하려면 tm의 간단함 때문에 많은 어려움이 있다. 어려움을 살펴보면 다음과 같다.

- stack이 없다.
  stack은 function call을 하거나 expression을 계산할 때 반드시 사용해야 된다. 하지만 tm에는 하나의 memory만 존재할 뿐 stack이 존재하지 않는다. 그렇기 때문에 memory 뒤쪽부터 stack같이 사용해야 한다.
- push, pop이 없다.
  stack이 없기때문에 당연히 push와 pop도 구현되어 있지 않다. push와 pop은 적당한 register 하나를 stack pointer로 사용하면 그리 어렵지 않게 구현이 가능하다.
- call, ret가 없다.
  function call을 구현할 때 call을 사용하면 편리하게 할 수 있다. 하지만 tm에는 call과 ret가 없다.

- label을 사용할 수 없다.
어느 특정 address로 jump를 할 때 label을 사용해서 jump하면 간단히 구현할 수 있다. 하지만 tm에서는 그렇게 할 수 없고 반드시 jump할 곳의 address를 사용해야 한다.

7 Code Generate

7.1 tm의 부족함 극복

tm의 부족함을 극복해서 code를 generate하기 위해서는 다음과 같이 하면 된다.

- stack의 구현
6번 register를 stack pointer로 사용한다. stack pointer는 현재 stack의 top을 나타내며 stack의 top은 현재 비어있는 stack의 top을 나타낸다.

- push, pop의 구현
push r은 아래와 같이 구현할 수 있다.

```
ST r, 0, 6
LDA 6, -1, 6
```

pop r은 아래와 같이 구현할 수 있다.

```
LDA 6, 1, 6
LD r, 0, 6
```

- call과 ret의 구현
4번 register를 frame pointer로 사용하고 5번 resigter는 memory 0 address를 가리키는 register로 사용한다. 그리고 아래와 같이 구현하면 된다.
call n의 구현 (n은 function이 구현되어 있는 address를 나타낸다.)

```
ST 4, 0, 6
LDA 4, 0, 6
LDA 6, -1, 6
LDA 3, 3, 7
ST 3, 0, 6
LDA 6, -1, 6
LDA 7, n, 5
```

ret의 구현

```
LDA 6, 0, 4
LD 4, 0, 6
LD 7, -1, 6
```

function의 argument처리는 caller가 argument를 전부 push시켜 주고 callee가 frame pointer를 기준으로 참조하면 되고 return 값은 0번 register에 저장해 주는 방법으로 caller에게 전달해 주면 된다. return될 때 push한 내용을 모두 pop을 해 어서 function call과정을 마무리하면 된다. 그리고 function call을 할 때 frame을 새로 만들고 이전의 frame pointer를 old frame pointer로 해서 새 frame에 저장한다. 이것은 나중에 return해서 이전 frame으로 이동할 때 쓰인다.

- label
  tm머신의 특징을 보면 처음에 address가 나오고 뒤에 instruction이 나오는 구조로 되어 있는데 address의 입력 순서는 아무렇게나 해도 상관이 없다. 예를 들어 1, 2, 5, 4, 3과 같은 순서의 address로 입력을 하면 1, 2, 3, 4, 5의 순서로 tm이 인식해서 받아들인다. label을 사용해 jump를 해야 되는 경우는 두가지가 있는데 첫번째는 if문이나 while문을 사용할 때 이고 두번째는 function을 call할 때 이다. 첫번째의 문제는 if나 while을 시작할 때 jump를 하는 부분을 넣을 address를 비워 두고 뒤쪽을 전부 구현한 다음 jump를 해야되는 곳이 나오면 앞에 구현하지 못한 부분을 찾아서 jump하도록 만들어 주면 된다. 그리고 function을 call할 때 jump는 symbol table을 참조해서 절대 address를 찾아서 해 주면 된다.

7.2 Expression Code Generate

+, -, *, /를 비롯해서 많은 연산을 c-에서 제공한다. 이런 것을 연산하기 위해서는 stack을 사용해야 한다. syntax tree를 traverse하면서 expression을 만나면 먼저 왼쪽에 있는 expression을 recursive하게 traverse한다. 그러면 계산의 결과값이 0번 register에 저장된다. 0번에 저장되게 하기 위해서는 token이 NUM이나 ID일때 value값을 0번 register에 저장하고 function을 call할 때 return값을 0번 register에 저장하도록 구현하면 된다. 이렇게 해서 왼쪽에 있는 expression의 계산이 끝나 0번 register에 저장되면 이 값을 stack에 push한다. 그리고 위와 같은 과정으로 오른쪽에 있는 expression에 반복 수행한다. 이 결과값은 register 0번에 저장된다. 그런 다음 아까 push한 것을 1번 register에 pop을 한다. 이 두개를 연산자에 맞도록 계산을 해서 다시 0번 register에 넣어주면 모든 expression이 수행되게 된다.

7.3 Array 처리

Array는 int array만 제공한다. array는 function call을 할 때 call by reference 로 이루어 져야 한다. 그렇기 때문에 array를 하나 만들게 되면 memory에 data와 address가 잡히게 된다. 그리고 symbol table에는 address가 저장되어 있는 위치를 저장한다. 나중에 array를 access하게되서 symbol table을 참조해 해당 주소를 찾아 보면 그곳에는 data가 있지 않고 address가 있게 된다. symbol table을 보면 해당 symbol이 int형인지 array형인지 알 수 있기 때문에 memory에 있는 것이 data인지 address인지 알 수가 있어서 array의 경우 실제 data가 있는 장소를 access할 수 있다. 또한 function call이 일어날때도 address 값을 parameter로 넘겨 주면 call by reference가 자연스럽게 구현이 된다.

8 결론 및 느낌점

이번 프로젝트를 통해 컴파일러를 손수 만들어 봄으로써 컴파일러의 구조를 상당히 파악할 수 있었다. 처음에 tm의 간결함때문에 구현이 쉽지 않을 것으로 예상되었지만 하나하나 문제를 해결해나가다 보니 의외로 모두 잘 해결되었다. 특히 push, pop, call, ret는 tm에서 사용할 수 있도록 그 과정을 정리해서 다른 학생들에게 알려 줘서 다른 학생들이 나만큼 헤매지 않아서 좋았다. lex와 yacc를 사용해서 컴파일러를 만들어 보았는데 lex와 yacc가 없다면 컴파일러를 만드는데 상당한 시간과 노력이 필요할 것 같았다.