TRƯỜNG ĐẠI HỌC CÔNG NGHỆ, ĐHQGHN

Phạm Nguyễn Tuấn Hoàng

Đỗ Việt Hưng

BÁO CÁO BÀI TẬP LỚN II

Chương trình dịch

HÀ NỘI - 2023

Mục lục

Chương 1. Giới Thiệu

1.1 Ngôn ngữ VC

VC là một biến thể của C. Nó gồm chủ yếu là một tập con của C cộng với một số tính năng ngôn ngữ từ Java. Ngôn ngữ này được thiết kế để sử dụng trong một môn học biên dịch ở bậc đại học, bao gồm một số kiểu nguyên thủy, mảng một chiều, cấu trúc điều khiển, biểu thức, câu lệnh phức hợp (tức là khối) và hàm, chủ yếu được lấy từ C. Những khía cạnh chính của Java trong ngôn ngữ này là như sau:

1. VC cung cấp một kiểu dữ liệu boolean, boolean, được mượn từ Java. Trong VC, tất cả các biểu thức boolean phải đánh giá thành một giá trị của kiểu boolean, là `true` hoặc `false`. C không có kiểu boolean trong khi C++ chuẩn hỗ trợ kiểu boolean. Trong C và C++, bất kỳ giá trị khác không nào đại diện cho `true` và bất kỳ giá trị bằng không nào đại diện cho `false`.

2. Trong VC, giống như trong Java, các toán hạng của một toán tử được đảm bảo được đánh giá theo một thứ tự đánh giá cụ thể, tức là từ trái sang phải. Trong C và C++, thứ tự đánh giá được để ngỏ. Trong trường hợp của `f() + g()`, ví dụ, VC quy định rằng `f` luôn được đánh giá trước `g`.

Ngoài mảng kiểu C, mọi hàm VC đều là một phương thức Java hợp ngữ pháp. Do đó, chỉ có những tính năng ngôn ngữ độc đáo của C được mô tả chi tiết.

1.2 LL(1) parser

LL(1) parser là một loại top-down parser được sử dụng trong khoa học máy tính để phân tích các ngôn ngữ lập trình. Nó được gọi là LL(1) vì nó sử dụng một ký hiệu nhìn trước để dự đoán quy tắc sản xuất tiếp theo. Thuật toán phân tích cú pháp LL(1) bắt đầu với một ngăn xếp và một luồng đầu vào, và hoạt động bằng cách liên tục so sánh đỉnh của ngăn xếp với ký hiệu tiếp theo trong luồng đầu vào, và áp dụng quy tắc sản xuất thích hợp cho ngăn xếp. Để xác định quy tắc nào được áp dụng, trình phân tích cú pháp sử dụng một bảng phân tích cú pháp được tạo ra từ ngữ pháp của ngôn ngữ đang được phân tích. Bảng phân tích cú pháp LL(1) được xây dựng bằng cách sử dụng các tập FIRST và FOLLOW của các ký hiệu ngữ pháp, và nó cung cấp cho trình phân tích

```
void main() {
    int a = 1;
}
```

Hình 1.1: Hình ảnh minh họa cho mã nguồn đầu vào

cú pháp một cơ chế nhìn trước dự đoán cho phép nó xử lý bất kỳ chuỗi đầu vào có thể trong một cách xác định. Trong khi phân tích cú pháp LL(1) đủ mạnh để xử lý hầu hết các ngôn ngữ lập trình, nó có một số hạn chế và có thể yêu cầu một số sửa đổi cho ngữ pháp để giải quyết bất kỳ xung đột nào có thể xảy ra trong quá trình phân tích cú pháp.

1.3 Định nghĩa bài toán

Trong bài báo cáo này, nhóm sẽ trình bày về quá trình xây dựng LL(1) parser cho ngôn ngữ VC với các chức năng sau:

- Xây dựng AST phục vụ cho việc phân tích cú pháp.
- Báo lỗi với các lỗi cú pháp.

Đầu vào: file text (*.vc) chứa code. Hình 1.1 minh họa cho mã nguồn đầu vào

Đầu ra: file text (*.vcps) minh họa đầu ra dưới dạng cây. Hình 3.2 là đầu ra cho ví dụ trên.

Chương 2. Phương pháp thực hiện

2.1 Chuẩn hóa văn phạm về LL(1)

2.1.1 Loại bỏ các kí tự *, +, ? và |

Đầu tiên, các kí tự *, +, ? trong văn phạm của ngôn ngữ VC cần phải được loại bỏ và thay thế bằng các production đơn giản hơn.

Cách thức khử các kí tự *, +, ? và | trong ngôn ngữ VC:

- Với production có dạng $S \rightarrow x^*$ sẽ được phân tích thành 2 production: $S \rightarrow A$ và $A \rightarrow xA|\epsilon$.
- Với production có dạng $S \rightarrow x?$ sẽ được phân tích thành 2 production: $S \rightarrow A$ và $A \rightarrow x|\epsilon$.
- Với production có dạng $S \rightarrow x^+$ sẽ được phân tích thành $S \rightarrow xA$ và $A \rightarrow xA|\epsilon$.
- Với production có dạng $S \rightarrow A_1|A_2|...|A_n$ sẽ được phân tích thành các production $S \rightarrow A_1$, $S \rightarrow A_2$, ..., $S \rightarrow A_n$.

2.1.2 Xử lý nhập nhằng

Sau khi phân tích ngữ pháp của VC, nhóm nhận thấy ngữ pháp có 2 lỗi. Lỗi đầu tiên là do production if-stmt 2.2. Nhóm đã xử lý bằng cách, trong production `if-stmt` nhóm đã thay thế `stmt` đầu tiên bằng `compound_stmt`. Lỗi thứ hai là do 2 production func-decl và var-decl 2.3. Ở lỗi này nhóm chỉ sử dụng func-decl thay vì sử dụng cả hai.

```
float_literal -> digit* fraction exponent?
               | digit+ .
               | digit+ .? exponent
fraction -> . digit+
```

Hình 2.1: Hình ảnh minh họa về việc sử dụng các kí tự *, +, ? và | trong văn phạm

```
if-stmt -> if "(" expr ")" stmt ( else stmt )?
```

Hình 2.2: Lỗi dangling else gây nhập nhằng

```
program -> ( func-decl | var-decl )*
func-decl -> type identifier para-list compound-
   stmt
var-decl -> type init-declarator-list ";"
```

Hình 2.3: Hai production gây nhập nhằng

2.2 Tìm tập hợp FIRST, FOLLOW

Tập FIRST của một ký hiệu phi kết thúc là tập các ký hiệu kết thúc có thể xuất hiện ở vị trí đầu tiên trong bất kỳ chuỗi nào được suy ra từ ký hiệu phi kết thúc đó.

Quy tắc để tính FIRST(X):

1. Nếu X là một ký tự kết thúc (terminal), thì FIRST(X) là tập hợp X.
2. Nếu X là một ký tự phi kết thúc (non-terminal) và X $\rightarrow \beta_1|\beta_2|...|\beta_n$ thì FIRST(X) là tập hợp FIRST(β_1) $\bigcup$ FIRST(β_2) $\bigcup$ ... $\bigcup$ FIRST(β_n).
3. Nếu X có dạng AB, FIRST(X) bao gồm tất cả các ký tự trong FIRST(A) ngoại trừ ϵ, cộng thêm tất cả các ký tự trong FIRST(B) nếu ϵ nằm trong FIRST(A).

Tập FOLLOW của một ký hiệu phi kết thúc là tập các ký hiệu kết thúc có thể xuất hiện ngay sau ký hiệu phi kết thúc đó trong bất kỳ chuỗi hợp lệ nào của ngữ pháp.

Quy tắc để tính FOLLOW:

1. FOLLOW(X) luôn bao gồm ký tự kết thúc "$".
2. Nếu có một quy tắc $B \rightarrow \alpha A \beta$:
   - FOLLOW(A) bao gồm tất cả các ký tự trong FIRST(β);
   - Nếu ϵ thuộc FIRST(β) thì cộng thêm các kí tự trong FOLLOW(B).

2.3 Tạo Parsing Table

Parsing table có kích thước bằng số lượng các ký tự không kết thúc nhân với số lượng các ký tự kết thúc. Mỗi ô trong bảng chứa một production rules, thể hiện cách di chuyển của parser khi gặp một ký tự non-terminal cụ thể đến một ký tự terminal cụ thể.

Các bước để tạo bảng Parsing Table dựa trên các tập FIRST và FOLLOW như sau:

1. Với mỗi production $A \rightarrow \alpha, \forall t \in FIRST(\alpha)$, ta đặt vào ô có vị trí (A,t) trong parsing table với giá trị là $A \rightarrow \alpha$.
2. Nếu ϵ thuộc $FIRST(\alpha)$ với production $A \rightarrow \alpha, \forall t \in (FIRST(\alpha) - \epsilon) \cup FOLLOW(\alpha)$, ta đặt vào ô có vị trí (A,t) trong parsing table với giá trị là $A \rightarrow \alpha$.

2.4 Tạo Abstract Syntax Tree từ parsing table

Sau khi xây dựng được parsing table, Abstract Syntax Tree(AST) tương ứng với đầu vào sẽ được tạo ra. Cây AST biểu diễn cấu trúc văn phạm của đầu vào và thường được sử dụng để phân tích ngữ nghĩa.

Các bước để tạo ra AST từ parsing table được mô tả như sau:

1. Khởi tạo stack rỗng và đưa kí tự bắt đầu của văn phạm ("$") vào stack.
2. Lấy kí tự ở đỉnh stack rồi tiến hành tạo ra node cho AST, nếu stack rỗng thì chuyển sang bước 5.
3. Đọc kí tự đầu tiên của đầu vào và tra cứu parsing table rồi tìm production phù hợp dựa trên kí tự vừa đọc và kí tự được lấy trong ngăn xếp.
4. Thực hiện production được tìm thấy từ parsing table, rồi tiến hành nạp các kí tự là đầu ra của production này vào ngăn xếp với thứ tự đảo ngược.
    - Nếu kí tự nằm ở đầu ngăn xếp trùng với kí tự được đọc từ đầu vào thì tiến hành tiếp tục bước 2, 3, 4 với kí tự tiếp theo từ đầu vào.
    - Nếu không trùng thì tiếp tục thực hiện bước 2, 3, 4 cho kí tự này.
5. Thuật toán kết thúc và hoàn thành việc xây dựng AST tương ứng với đầu vào.

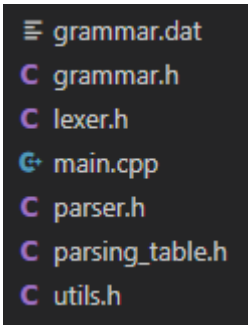
Khi xây dựng AST, mỗi node trong của AST tương ứng với một kí tự phi kết thúc được sinh ra từ 1 production, trong khi đó, lá của AST tương ứng với các kí tự kết thúc từ chuỗi đầu vào.

Chương 3. Cấu trúc mã nguồn

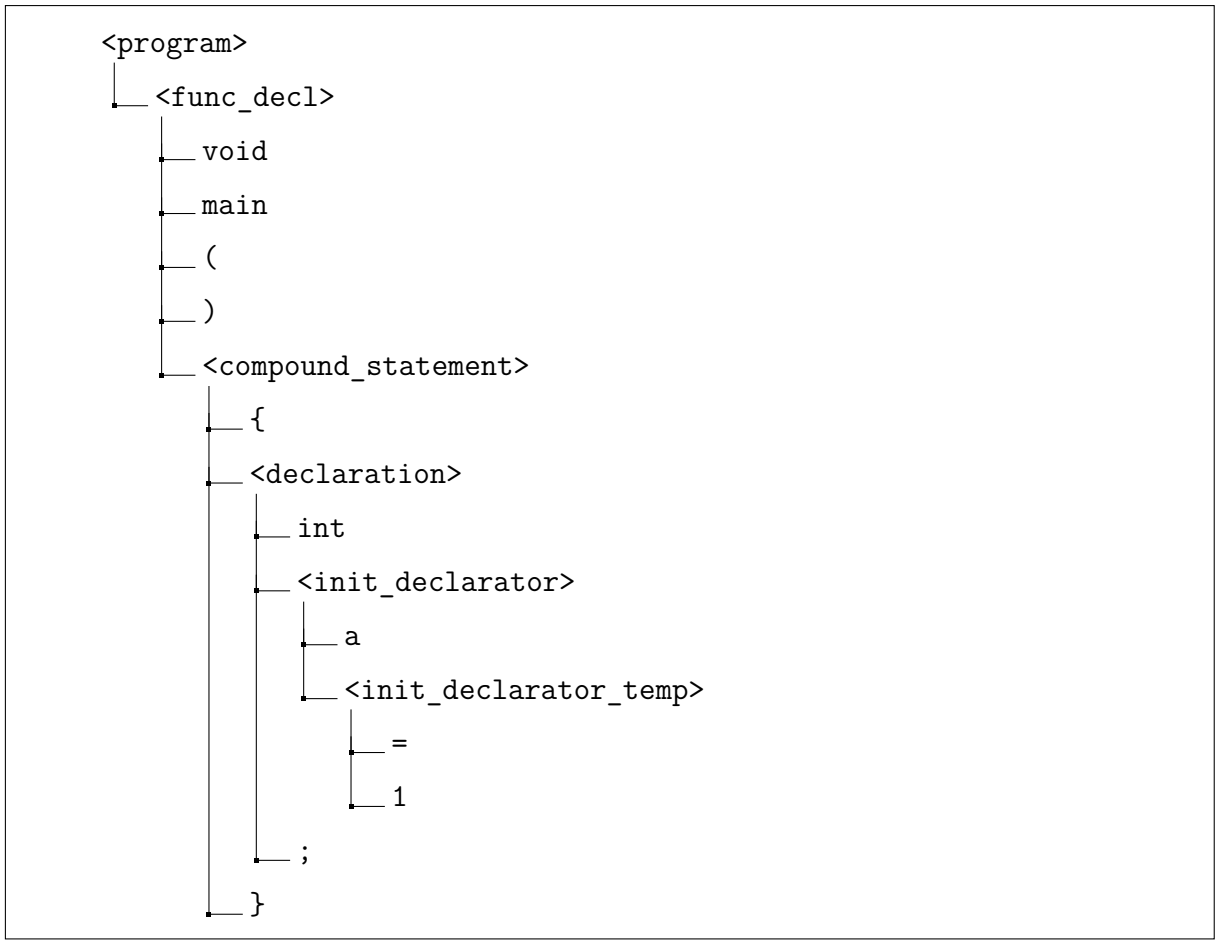
3.1 Cấu trúc chương trình

Chương trình được xây dựng trên ngôn ngữ lập trình C++. Hình 3.1 mô tả cấu trúc của chương trình, bao gồm:

- **grammar.dat**: Tệp bao gồm các ngữ pháp đã được chuẩn hóa về LL(1).
- **grammar.h**: Tệp bao gồm một lớp giúp đọc và kiểm tra tệp grammar.dat và các lớp để định nghĩa production và rule.
- **lexer.h**: Tệp bao gồm một lớp lớp giúp đọc tệp đầu vào và trả lại các token là đầu vào của parser để tiến hành phân tích.
- **main.cpp**: Tệp bắt đầu chương trình bao gồm các bước để xử lý bài toán.
- **parser.h**: Tệp bao gồm một lớp để định nghĩa đối tượng AST và một lớp để tạo và thu gọn đối tượng AST.
- **parsing_table.h**: Tệp bao gồm một lớp để định nghĩa và tạo parsing table.
- **utils.h**: Tệp bao gồm các hàm để tính các tập FIRST và FOLLOW.



Hình 3.1: Các thành phần chính của chương trình



Hình 3.2: Hình ảnh minh họa AST

3.2 Đầu ra

Để minh họa AST một cách dễ hiểu, nhóm minh họa đầu ra của bài toán dưới dạng cây. Đầu ra đã được lược bỏ các node không cần thiết được hiển thị như hình 3.2, bao gồm:

- Loại bỏ các node mà các nút lá chỉ có giá trị EPSILON.
- Loại bỏ các node phi kết thúc mà chỉ có duy nhất một node con.

Chương 4. Cài đặt và thực nghiệm

4.1 Cài đặt và chạy chương trình

- Ngôn ngữ C++ 17 (MSYS2 MinGW64)
- Để build executable file, chạy lệnh:
  `g++ -std=c++17 -g main.cpp -o main.exe`
- chạy file `main.exe` với câu lệnh:
  `main.exe [input_file] [output_file] [grammar_file]`

Lưu ý: Nhập tên file mà không có đuôi (Ví dụ: chỉ nhập `"in"` thay vì `"in.vc"`, tương tự với `"*.vcps"` và `"*.dat"`).

4.2 Kết quả và kiểm thử

Hình 4.1 là đầu ra của hai lỗi đã được đề cập ở phần 2.1.2. Ở phần lỗi của 2 production func-decl và var-decl, nhóm đã thử thay đổi ngữ pháp, nhưng lỗi vẫn xảy ra, nên nhóm đã quyết định chỉ xử dụng func-decl thay vì sử dụng cả hai.

```
parsing table: duplicate entry - prod: <else-stmt> - symbol: else
--------------------------------------------------------------------------------
parsing table: duplicate entry - prod: <program_tail> - symbol: void
parsing table: duplicate entry - prod: <program_tail> - symbol: int
parsing table: duplicate entry - prod: <program_tail> - symbol: float
parsing table: duplicate entry - prod: <program_tail> - symbol: boolean
parsing table: duplicate entry - prod: <program> - symbol: void
parsing table: duplicate entry - prod: <program> - symbol: int
parsing table: duplicate entry - prod: <program> - symbol: float
parsing table: duplicate entry - prod: <program> - symbol: boolean
```

Hình 4.1: Mô tả đầu ra của hai lỗi ở phần 2.1.2. Phần trên là lỗi if-else, phần dưới là lỗi của 2 production func-decl và var-decl.

Test case	Input	Output	Expect
Define variable	int a = 5;	Good	Good
Compare operator	a != 1; a == 1; a >= 1; a <= 1;	Good	Good
Math operator	a + 1; a - 1; a * 1; a / 1;	Good	Good
While	while(a) {};	Good	Good
If	if(a) {};	Good	Good
If without ()	if a {};	Error	Error
If else	if(a) {} else a = 1 ;	Good	Good
For	for(i = 1; i <2; i = i + 1) a = a + 1;	Good	Good
Logic	a = !true && false;	Good	Good
Logic	a = true || !a;	Good	Good
Array	int a[];	Good	Good
Array with length	int a[10];	Good	Good
Wrong Array	int[] a;	Error	Error
Function call	int a = sum(1,2);	Good	Good
Function init	int sum(int a, int b) { return a + b;}	Good	Good
Define argument in function	int square(int a=1) { return a*a;}	Error	Error
Miss ';'	int a = 1	Error	Error

Bảng 4.1: Kết quả kiểm thử của một số ví dụ

Kết luận

Có thể nhận thấy LL(1) parser được nhóm xây dựng đã hoàn thành các mục tiêu đề ra và đáp ứng được yêu cầu của đề tài. Hệ thống đã thành công phân tích ngữ pháp và báo lỗi khi có ngữ pháp nhập nhằng, hệ thống cũng đã phân tích thành công các đoạn mã nguồn viết bằng ngôn ngữ VC và báo lỗi khi gặp lỗi biên dịch và lỗi cú pháp trong mã nguồn. Kết quả trả về là môt AST đại diện cho cú pháp của mã nguồn.

Tuy nhiên, hệ thống LL(1) parser vẫn còn nhiều vấn đề gặp phải, ví dụ như xử lý nhập nhằng, trường hợp if-else là một ví dụ cụ thể cho thấy hạn chế của hệ thống. Hệ thống vẫn có thể được cải thiện trong tương lai bằng cách tối ưu mã nguồn giúp cho thời gian chạy ngắn và sử dụng tài nguyên hiệu quả hơn, làm tiền đề cho việc xây dựng cả hệ thống chương trình dịch cho ngôn ngữ lập trình này

Tổng thể, hệ thống đã đạt được kết quả tốt và có tiềm năng để phát triển và cải thiện trong tương lai. Điều này sẽ giúp cho người dùng trong việc phát triển các ứng dụng viết bằng ngôn ngữ lập trình VC. Các công nghệ và kỹ thuật đã sử dụng trong dự án này cũng có thể được áp dụng để phát triển các ứng dụng khác trong lĩnh vực phân tích mã nguồn.

Tài liệu tham khảo

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.