Implementação do Analisador Sintático: Linguagem CStr

Eduardo Furtado — 09/0111575
Departamento de Ciência da Computação
Universidade de Brasília

6 de outubro de 2016

1 Implementação e funcionamento do programa

Nesta iteração do projeto foi desenvolvido um analisador sintático para a linguagem CStr, descrita na primeira iteração. O analisador sintático utiliza o fluxo de tokens gerado pelo análisador léxico feito na iteração anterior.

Para o desenvolvimento, utilizou-se o Bison. A implementação está contida no arquivo 090111575.y.

A primeira coisa a ser feita nessa etapa foi integrar o Flex e o Bison em meu ambiente de desenvolvimento.

Após conseguir compilar o projeto, passei a fazer com que ambos se comuniquem. Diversas mudanças foram feitas no arquivo .l, com o código do Lexer, como a remoção da função main, que foi passada para o arquivo .y. Várias outras modificações foram feitas com os valores retornados, pois já não era mais objetivo imprimir os tokens.

Uma vez que o fluxo de tokens passou a servir de entrada para o Parser foi a hora de implementar minha gramática no Bison. Este processo foi extremamente demorado, mais de 30 horas de trabalho, pois envolveu fazer diversos *refactorings* até deixar a gramática livre de conflitos de shift/reduce ou reduce/reduce.

O próximo passo foi desenvolver as estruturas e funções auxiliares para a árvore de sintaxe e a tabela de símbolos, para finalmente construir elas com as ações semânticas do Bison.

Ambas a árvore de sintaxe e a tabela de símbolos são impressos na saída padrão quando se executa o código.

Durante todo o desenvolvimento escrevi este relatório e testei o código com diversos arquivos de teste, descritos em sessão posterior.

Para compilar e executar o programa foi disponibilizado um script para ambiente Unix, que pode ser executado como descrito no arquivo leia-me.txt. Este arquivo também contém informações sobre como compilar e executar em ambiente Windows.

2 Especificação da gramática da linguagem

A gramática de CStr é baseada em uma versão reduzida da linguagem C [1], com acréscimo do tipo primitivo string e o operador de concatenação (:), e métodos segundo [2].

Na entrega da versão anterior desse trabalho, o símbolo a ser usado para concatenação era o ponto (.). Este símbolo foi trocado para o dois pontos (:) para facilitar a implementação e remover ambiguidade da gramática.

Uma alteração foi feita com base no comentário feito pelo corretor da segunda iteração:

Comentário: "- Para palavras-chave, sugiro que não se defina um valor RESERVED para todas elas, mas que cada uma possua um valor próprio. Por exemplo, IF para if, ELSE para else, etc. Isso facilita seu uso na hora de fazer a gramática, pois são utilizadas em lugares diferentes."

Alteração: Foi utilizado um valor reservado para cada uma delas, como sugerido. Na verdade, fiz a mudança antes de receber a correção do trabalho anterior, pois quando comecei a fazer o Analisador Sintático ficou claro que deveria ter reservado uma palavra-chave para cada uma delas.

A gramática livre de contexto a seguir descreve a linguagem CStr proposta, com modificações, onde variáveis (símbolos não-terminais) começam com letras maiúsculas, Function é a variável inicial e todos os outros símbolos são terminais. A barra vertical | é usada para indicar definições alternativas para um não-terminal.

Alterações em relação a entrega anterior estão marcadas na gramática em amarelo.

Elementos adicionados estão marcados em azul.

Elementos removidos estão riscados (~~exemplo~~) e marcados em rosa.

Begin	→	Function
Function	→	Type Identifier (FormalArgList) CompoundStmt
		| Function Type Identifier (FormalArgList) CompoundStmt
Identifier	→	char (char | digit | _)*
FormalArgList	→	ε
		| FormalArg
		| FormalArgList , FormalArg
FormalArg	→	Type Identifier
ArgList	→	ε
		| Arg
		| ArgList , Arg
Arg	→	Factor
		| FunctionCall
		| StringConcat
		| ~~Identifier~~
Declaration	→	Type Identifier
		| Type Attribution
		| ~~Type Identifier ;~~
Type	→	int
		| ~~float~~
		| string
Stmt	→	WhileStmt
		| Expr ;

| IfStmt
| CompoundStmt
| Declaration ;
| Declaration ;
| ReturnStmt ;
| ;

WhileStmt → while (Expr) CompoundStmt

IfStmt → if (Expr) CompoundStmt
| if (Expr) CompoundStmt else CompoundStmt

~~ElsePart~~ → ~~else Stmt~~
| ε

CompoundStmt → { StmtList }

StmtList → StmtList Stmt
| ε

ReturnStmt → return
| return Expr

Expr → ~~Identifier = Expr~~
| Rvalue
| Attribution
| FunctionCall
| StringConcat

Attribution → Identifier = Expr

Rvalue → Rvalue Compare Addition
| Addition

Compare → ==
| <
| >
| <=
| >=
| ! =

Addition → Addition + Multiplication
| Addition - Multiplication
| Multiplication

Multiplication → Multiplication * Factor
| Multiplication / Factor
| Term

Term → (Term)
| - Term
| + Term
| Factor

Factor → ~~(Expr)~~
| ~~- Factor~~
| ~~+ Factor~~

| Identifier
| number
| text

FunctionCall → Identifier (ArgList)
| Identifier . Identifier (ArgList)

StringConcat → StringConcat : text
| StringConcat : Identifier
| text : text
| Identifier : text
| text : Identifier
| Identifier : Identifier
| ~~string . string~~
| ~~Identifier . string~~

A ordem em que as operações aparecem determina a precedência de cada operador, onde a primeira tem menor procedência e a última tem a maior procedência.

Alguns itens mudaram de nome para aumentar a clareza da gramática. Por exemplo “Mag” agora está devidamente nomeado como “Addition”. Essas mudanças não foram marcadas porque não modificam o que a gramática expressa.

As várias mudanças na gramática foram resultado do processo de eliminação de conflitos de shift/reduce e reduce/reduce, bem como eliminar ambiguidade da gramática.

O problema da ambiguidade das expressões “if-else” foi abordado de acordo com um tutorial [3].

Como discutido em sala de aula, e também segundo as referências [4] e [3], foi utilizado recursão à esquerda, para evitar problemas de falta de memória para a pilha que podem acontecer com o uso de recursão à direita.

3 Política de tratamento de erro adotada

Quando ocorre um erro sintático, é exibido na tela a linha em que o erro ocorreu, qual era o(s) símbolo(s) esperado(s) e qual foi o símbolo encontrado pelo Parser.

Ao encontrar erros sintáticos a execução do Parser continua. Este mecanismo foi feito de acordo com o manual do Bison [5], ou seja, adicionando uma regra de sincronização em lugares estratégicos:

Ao encontrar um erro em “Stmt”, o Parser busca o próximo token “;” ou ”{” e retoma a análise logo após este token. No caso de “CompoundStmt”, busca-se o seguinte token ”}”. Quando se trata de “Function”, a procura é pelo token ”}”.

Dessa maneira garante-se que os erros sintáticos serão exibidos, o que facilita o processo de *debugging*.

Outro ponto a ser explanado é que a Árvore Sintática e a Tabela de Símbolos não são exibidos ao final, quando há erros no código parseado.

O tratamento de erros encontrados pelo Analisador Léxico continua funcionando como na entrega anterior do projeto:

O analisador léxico implementado identifica erros, porém não para com a análise quando isso acontece.

Os erros são mostrados na saída, em meio aos tokens identificados, com um aviso "ERRO!", seguido da posição do erro no código, linha e coluna), e acompanhado da classificação do erro. Com exceção de comentário não fechado e string grande demais, também é mostrado o pedaço de código referente a aquele erro.

A classificação dos erros é a seguinte:

- Comentário multilinha não fechado: blocos de código comentados com o delimitador /* e que não são fechados com o delimitador */;
- String não fechada: strings devem estar entre aspas duplas. Se não for encontrado as aspas duplas que fecham essa string, na mesma linha, um erro é identificado;
- String grande demais: foi estipulado que o tamanho máximo para strings é de 9999 caracteres;
- Identificador inválido: um exemplo de identificador inválido é aquele que começa com um número, por exemplo: "2i";
- Identificador grande demais: foi estipulado que o tamanho máximo para identificadores é de 255 caracteres;
- Caractere inválido: um caractere inexistente na gramática da linguagem CStr. Por exemplo "'", pois a linguagem não suporta o tipo char.

4 Funções alteradas/introduzidas

Para lidar com erros a função yyerror de tratamento de erros do Bison foi modificada para comportar-se como descrito na sessão 3 deste documento.

Com esta entrega, foram introduzidas as seguintes funções:

- void printSymbolTable() - imprime a Tabela de Símbolos;
- void addSymbolTableLine(data_for_the_symbol_table*) - Cria uma nova linha na Tabela de Símbolos e aponta o elemento anterior e o que segue;
- void setCurrentScope(const char*) - função que ajuda na construção da tabela de símbolos que mantém registrado qual é o escopo atual;
- NODE* addTreeNode(description_and_value_data_for_the_tree*) - aloca memória para adicionar um novo nó à Árvore Sintática;
- void addNodeToListOfNodeBrothers(NODE*, NODE*) - função que auxiliar da função "addChildToTreeNode";
- void addChildToTreeNode(NODE*, NODE*) - insere nós na Árvore Sintática;

- void setValueInSymbolTableEntry(char *) - inclui na tabela de símbolos o valor de uma variável que foi inicializada na declaração;
- void printSyntaxTree(NODE* root, int identation_counter) - imprime a Árvore Sintática;
- void setLastValueByIdentifierName(char *) - função auxiliar, necessária pela função "setValueInSymbolTableEntry" para incluir na tabela de símbolos o valor de uma variável que foi inicializada na declaração.

As seguintes funções foram introduzidas na etapa anterior e seguem sendo necessárias no Lexer:

parse_multiple_line_comments() trata de ignorar caracteres dentro do bloco comentado e contar as colunas em cada linha e as linhas dentro do comentário. Essa função também trata do caso de erro em que o comentário multilinha não for fechado.

parse_single_line_comments() trata de ignorar caracteres na linha comentada e incrementa o contador de linhas.

parse_string() trata strings, delimitadas por aspas duplas. Identifica erros de strings maiores do que as permitidas pela linguagem (maiores do que 9999 caracteres), ou strings não fechadas. Optou-se por suportar somente strings delimitadas por aspas duplas em uma única linha.

5 Árvore sintática e tabela de símbolos

A criação de um nó na árvore se dá quando uma regra é identificada pelo Analisador Sintático. A informação é passada através da estrutura de dados "description_and_value_data_for_the_tree".

É alocado memória para um nó, em seguida os campos da estrutura são preenchidos com os valores lidos e é feito a conexão com os filhos a serem criados e pai do nó, de acordo com a situação: A quantidade de filhos depende da regra que que resultou em um *match*.

Algumas informações presentes na Tabela de Símbolos também foram guardadas na Árvore Sintática, são elas o nome de identificadores e os valores de variáveis, quando possível. O escopo de variáveis e o tipo de funções de variáveis também estão representados na árvore.

A seguir estão as estruturas de dados relacionadas à Árvore Sintática (o texto não está marcado com acentos porque trata-se de código):

```
// Estrutura usada para adicionar um novo no na arvore
typedef struct description_and_value_data_for_the_tree {
    char description[255];
    char value[255];
} description_and_value_data_for_the_tree;

// Arvore Sintatica:
typedef struct tree_node_struct {
    char description[255];
    char value[255];
    int node_brother_count;
    struct tree_node_struct *father, *node_brothers_list, *child;
} NODE;
```

A Tabela de Símbolos é preenchida com o nome dos identificadores de funções, que sempre tem escopo global, campo de valor nulo, e tipo de retorno int ou string.

Além disso também é preenchida com o nome dos identificadores de variáveis, que sempre estão no escopo de alguma função. O campo de valor nulo, caso seja um argumento de função ou variável declarada e não inicializada, ou, no caso em que se trate de uma variável declarada e inicializada, pode contar o valor do número se for do tipo int ou um texto, se for do tipo string. Os dois tipos possíveis são int e string.

A seguir estão as estruturas de dados relacionadas à Tabela de Símbolos (o texto não está marcado com acentos porque trata-se de código):

```
// Estrutura usada para adicionar novas linhas na Tabela de Simbolos:
typedef struct data_for_the_symbol_table {
    char name[255];
    char type[255];
    char category[255];
} data_for_the_symbol_table;

// Tabela de Simbolos:
typedef struct symbol_table_struct {
    char name[255];
    char value[255];
    // o escopo pode ser global (no caso de funcoes) ou o nome da funcao em que
    // esta (no caso de variaveis)
    char scope[255];
    char type[255];
    char category[255];
    struct symbol_table_struct *previous, *next;
} symbol_table_struct;
```

6 Descrição dos arquivos de teste

Foram produzidos quatro arquivos de testes com código correto:

- codigo_correto1.cstr - contém código que seria escrito em uma linguagem C baseada na gramática reduzida que usei de base para o projeto [1]. Este é o mesmo arquivo entregue na etapa anterior.

- codigo_correto2.cstr - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings! Este é o mesmo arquivo entregue na etapa anterior.

- teste.cstr - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings! Este código possui um código com o máximo de elementos sintáticos possíveis, para contemplar as possibilidades da linguagem.

- teste2.cstr - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings! Este código possui um código simples, que tem como intuito visualizar bem a árvore de sintaxe e a tabela de símbolos.

Também foram entregues os arquivos de testes com problemas léxicos no código, os mesmos entregues na etapa anterior:

O arquivo codigo_incorreto1.cstr - contém código incorreto que seria escrito em uma linguagem C baseada na gramática reduzida que usei de base para o projeto [1].

- Na linha 3 há um identificador maior do que o permitido pela linguagem (255 caracteres);
- Na linha 4 há um identificador inválido, que começa com um caractere numérico;
- Na linha 5 há um caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 6 há outro caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 7 há caracteres inválidos, não contemplados pela gramática da linguagem, pois tentou-se utilizar o tipo char, não suportado pela linguagem;
- Na linha 8 há outro caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 9 há outro caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 10 há um comentário multilinha que não foi fechado, bem como um *Easter egg* para o leitor.

O arquivo codigo_incorreto2.cstr - contém código incorreto que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings!

- Na linha 2 há um identificador inválido, que começa com um caractere numérico.;
- Na linha 3 há um caractere inválido, não contemplado pela gramática da linguagem, seguido de um identificador inválido, onde o programador colocou um "f" desnecessário ao final de um número do tipo float;
- Na linha 4 há uma string que não foi fechada, pois o programador confundiu aspas duplas com aspas simples;
- Na linha 4 há uma string maior do que os 9999 caracteres que a linguagem permite.

Do ponto de vista sintático, estes arquivos também contém erros. O arquivo codigo_incorreto1.cstr, contém:

- Linha 1: "syntax error, unexpected Identifier, expecting INT or STRING"
- Linha 3: "syntax error, unexpected =, expecting Identifier"
- Linha 4: "syntax error, unexpected ;, expecting Identifier"
- Linha 5: "syntax error, unexpected Identifier, expecting ;"
- Linha 5: "syntax error, unexpected ;"

- Linha 6: “syntax error, unexpected Identifier, expecting ;”
- Linha 7: “syntax error, unexpected Identifier, expecting ;”
- Linha 8: “syntax error, unexpected :, expecting ;”
- Linha 64: “syntax error, unexpected $end”

Similarment, o arquivo codigo_incorreto2.cstr, contém:

- Linha 2: “syntax error, unexpected ;, expecting Identifier”
- Linha 3: “syntax error, unexpected ., expecting ;”
- Linha 5: “syntax error, unexpected RETURN”

7 Dificuldades encontradas

A primeira dificuldade durante o desenvolvimento desta fase foi com a curva de aprendizagem do Bison. A documentação foi de grande ajuda [5], bem como os tutoriais com exemplos disponibilizados no Moodle e nas referências [4] e [3] e o livro adotado pela disciplina [6].

A maior dificuldade para o desenvolvimento desta etapa foi no cunho da minha vida pessoal, pois não estou mais trabalhando, para poder ter tempo de me dedicar à disciplina. Por sorte não estou cursando outras disciplinas, exceto Trabalho de Graduação 2, que está recebendo muito menos atenção do que merece. De qualquer maneira, trata-se de uma das épocas mais infelizes de minha vida, por conta de diversas oportunidades de trabalho que estão sendo deixadas de lado, e dos desafios crescentes enfrentados na disciplina. Os companheiros da turma, que se encontram em situação bem parecidas, e os amigos pessoais, os quais, atualmente, entro em contato apenas pela internet, são de grande ajuda para não desistir.

Outra dificuldade foi a resolução de conflitos e ambiguidades na gramática, pois foi requerido muito raciocínio e em diversas vezes tomei decisões certas para algo pontual que rompiam algo em outro lugar e que vinham a ser percebidas quando o problema estava propagado a vários pontos, me forçando a recomeçar o processo diversas vezes, por ter escolhido um ramo errado.

Algo mais simples, porém que tomou uma quantidade considerável de tempo, foi levar em consideração a especificação da próxima etapa para tomar minhas decisões durante este desenvolvimento.

Referências

[1] MiniC Grammar. `http://www2.ufersa.edu.br/portal/view/uploads/setores/184/AppendixA.pdf`. [Acessado em 20 de agosto de 2016].

[2] C string lib reference. `http://www.cplusplus.com/reference/cstring/`. [Acessado em 20 de agosto de 2016].

[3] Lexical Analysis With Flex. http://epaperpress.com/lexandyacc/index.html. [Acessado em 5 de outubro de 2016].

[4] Lexical Analysis With Flex. http://aquamentus.com/flex_bison.html. [Acessado em 5 de outubro de 2016].

[5] Lexical Analysis With Flex. https://www.gnu.org/software/bison/manual/bison.html. [Acessado em 5 de outubro de 2016].

[6] A.V. AHO, R. Sethi, and S. Lam. *Compiladores: princípios, técnicas e ferramentas*. LONGMAN DO BRASIL, 2008.