Implementação do Analisador Léxico: Linguagem CStr

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

8 de setembro de 2016

1 Implementação e funcionamento do programa

Nesta iteração do projeto foi desenvolvido um analisador léxico para a linguagem CStr, descrita na iteração anterior. Com base nos comentários recebidos pelo corretor, foram feitas algumas alterações no projeto original, que estarão descritas neste documento.

O analisador léxico foi desenvolvido de acordo com o que foi visto nas aulas práticas, utilizando o Flex e expressões regulares.

Os arquivos codigo_correto1.cstr, codigo_correto2.cstr, codigo_incorreto1.cstr e codigo_incorreto2.cstr contém porções de código usados para testar o analisador léxico. Estes arquivos estão mais detalhados em uma sessão posterior deste documento.

No arquivo 090111575.l foi implementado o analisador léxico: O arquivo de entrada (analisado), é lido completamente da esquerda para a direita e de cima pra baixo. Enquanto isso é feito, cada porção de código, os lexemas, é analisada para gerar os tokens de acordo com a gramática proposta para a linguagem.

Para cada token identificado há uma linha na saída que indica a linha e coluna em que aquele token aparece, bem como sua classificação, que pode ser:

- Identificador;
- Número;
- Operador binário;
- Expressão relacional;
- Palavra reservada;
- Delimitador;
- String;
- Fim de instrução;
- Comentário não fechado;

- String não fechada;
- String grade demais;
- Identificador inválido;
- Identificador grade demais;
- Caractere inválido.

Para fazer isso, a cada lexema, busca-se uma expressão regular que combine com as que foram definidas. A ordem de prioridade segue dois critérios, em ordem de prioridade:

- Maior expressão combinada;
- Qual expressão foi definida primeiro.

Além de tokens, também são mostrados na saída quais queres erros identificados.

É importante ressaltar que a identificação de um erro não interrompe a análise léxica. São seis tipos de erros apresentados na saída do programa:

- Comentário multilinha não fechado;
- String não fechada;
- String grande demais;
- Identificador inválido;
- Identificador grande demais;
- Caractere inválido.

A política de tratamento de erro está detalhada em uma sessão posterior deste documento.

A linguagem aceita comentários de uma linha (utilizando a sintaxe "// ...") e multilinhas, também chamado de comentário em bloco (utilizando a sintaxe "/* ... */"). Os comentários não são considerados tokens, portanto são ignorados na saída do programa, porém suas linhas e colunas são contabilizados na contagem.

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].

Algumas alterações foram feitas com base nos comentários feitos pelo corretor da primeira iteração:

Comentário: "- Se Function for a variável inicial de sua gramática, programas escritos nela terão somente uma função?"

Alteração: Na verdade a linguagem havia sido projetada para ter uma função principal, descrita pela variável inicial "Function", e funções relacionadas a strings, descritas pela variável StringFunction.

A variável Function foi alterada para que a linguagem suporte mais de uma função. Como consequência disso a variável StringFunction foi removida por redundância.

Foi cogitado fazer alterações mais profundas para que a linguagem passe a suportar variáveis globais, porém, optei por manter a simplicidade e não permitir variáveis globais.

Essas alterações estão marcadas na gramática em amarelo.

Comentário: "- Assumindo que, em alguns lugares "string" seja uma variável (como em StringConcat, mas não em Type), faltou sua definição."

A regra StringConcat foi modificada para aceitar tanto strings soltas como variáveis do tipo string (a verificação do tipo não cabe ao analisador léxico e sim ao analisador semântico).

Ou seja, sintaticamente, é permitido fazer, respectivamente:

```
string variavel_string_de_exemplo;
variavel_string_de_exemplo = "1 " . "2 ";
variavel_string_de_exemplo2 = variavel_string_de_exemplo . "3.";
variavel_string_de_exemplo3 = variavel_string_de_exemplo . variavel_string_de_exemplo2;
```

Onde o resultado de "variavel_string_de_exemplo" será:

```
1 2 1 2 3.
```

É importante ressaltar que se optou, durante o desenvolvimento do analisador léxico, por não suportar strings, delimitadas por aspas duplas, com mais de uma linha. Se isso acontece, um erro de string não fechada é detectado e avisado. O comprimento máximo escolhido para strings foi de 9999 caracteres.

Essas alterações estão marcadas na gramática em rosa.

Comentário: "- Todas as funções devem ter pelo menos um argumento? (ver definição de FormalArgList e seu uso em Function)"

A modificação foi simples, adicionou-se a operação estrela (*) ao argumento de "Function" Essa alteração está marcada na gramática em azul.

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.

Function	→	Function+
		| Type Identifier (FormalArgList*) CompoundStmt
Identifier	→	char (char | digit | _)*
FormalArgList	→	FormalArg
		| FormalArgList , FormalArg
FormalArg	→	Type Identifier
ArgList	→	Arg
		| ArgList , Arg
Arg	→	Identifier
Declaration	→	Type IdentList ;
Type	→	int
		| float
		| string
~~StringFunction~~	→	~~string . Identifier ()~~
		~~| string . Identifier (ArgList)~~
StringConcat	→	string . string
		| Identifier . string
		| Identifier . Identifier
IdentList	→	Identifier , IdentList
		| Identifier
Stmt	→	WhileStmt
		| Expr ;
		| IfStmt
		| CompoundStmt
		| Declaration
		| StringFunction
		| StringConcat
		| ;
WhileStmt	→	while (Expr) Stmt
IfStmt	→	if (Expr) Stmt ElsePart
ElsePart	→	else Stmt
		| ε
CompoundStmt	→	{ StmtList }
StmtList	→	StmtList Stmt
		| ε
Expr	→	Identifier = Expr
		| Rvalue
Rvalue	→	Rvalue Compare Mag
		| Mag
Compare	→	==
		| <
		| >
		| <=
		| >=

	| ! =
Mag →	Mag + Term
	| Mag - Term
	| Term
Term →	Term * Factor
	| Term / Factor
	| Factor
Factor →	(Expr)
	| - Factor
	| + Factor
	| Identifier
	| number

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.

Houve também *feedback* que não estava relacionado com a gramática da linguagem:

Comentário: "- Em seu próximo documento, por favor seja um pouco mais específico quanto a qual algoritmo de String matching será utilizado."

Será implementado o algoritmo de Knuth-Morris-Pratt's (KMP). Ainda não sei mais detalhes sobre a implementação dele, porque isso ainda é um problema a ser solucionado no futuro.

Comentário: "- Em geral, não foi possível compreender a semântica das operações pelas descrições apresentadas."

O objetivo é poder concatenar strings no código da mesma maneira que JavaScript, por exemplo, faz com o uso do operador "+", por exemplo:

```
txt1 = "John";
txt2 = "Doe";
txt3 = txt1 + " " + txt2;
```

O resultado de "txt3" será:

```
John Doe
```

A diferença é que na linguagem CStr isso será feito com o operador ".".

3 Política de tratamento de erro adotada

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.

Cogitou-se implementar algo para corrigir erros, por exemplo, buscar qual é a primeira palavra reservada em um comentário multilinhas que não foi fechado e assumir que o comentário acaba antes dessa palavra reservada. Porém, fazer um tratamento inteligente requer muito esforço, e poderia ser a principal proposta de um compilador, como a do aluno que implementou um compilador para disléxicos.

4 Funções alteradas/introduzidas

Foram introduzidas três novas funções:

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.

O contador de linhas e o contador de colunas são variáveis globais que são manipuladas ao longo do programa para serem mostradas na saída.

Por orientação de um amigo que já cursou a disciplina, foi passado ao Flex as opções bison-bridge, que declara a variável inicial "yyval" para integração com o bison, e noyywrap, que remove a necessidade de implementação do "yywrap".

5 Descrição dos arquivos de teste

Foram produzidos dois 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];
- codigo_correto2.cstr - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings!

Foram produzidos dois arquivos de testes com código incorreto:
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.

6 Dificuldades encontradas

A maior dificuldade durante o desenvolvimento desta fase foi com a curva de aprendizagem do Flex. A documentação foi de grande ajuda [3].

Também foi difícil tomar decisões sobre que tokens deveriam estar contemplados pelas expressões regulares, o que exigiu um profundo entendimento da gramática. O livro adotado pela disciplina ajudou bastante [4], bem como a monitoria da disciplina.

Outra dificuldade foi para melhorar o projeto segundo os comentários que recebi sobre a primeira entrega. Isso foi difícil porque cometi o erro de optar por fazer as modificações requisitadas nos comentários após o "término" do desenvolvimento do analisador léxico. Como não fiz as alterações antes de começar a desenvolver o analisador léxico tive que voltar atrás para fazer modificações em coisas que supostamente estavam terminadas.

Algo mais simples, porém que tomou uma quantidade considerável de tempo, foi projetar arquivos de testes que contemplem cada possibilidade de erro ou token identificado.

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://flex.sourceforge.net/manual/`. [Acessado em 31 de agosto de 2016].

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