Universidade Federal do Amazonas
Instituto de Computação
Compiladores
Prof. Edson Nascimento Silva Júnior

Trabalho 02 – Análise Léxica (flex) e sintática (yacc)

Descrição geral

O objetivo deste trabalho consiste na implementação de um compilador funcional, para a linguagem que chamaremos a partir de agora de first2023.. Esta etapa do trabalho consiste em fazer um analisador léxico, **utilizando a ferramenta de geração de reconhecedores lex (ou flex)**, e um analisador sintático, **utilizando a ferramenta de geração de reconhecedores yacc (ou bizon)**, e completar o preenchimento da tabela de símbolos, guardando o texto e tipo dos lexemas/tokens.

Análise Léxica

A sua análise léxica deve fazer as seguintes tarefas:

A. reconhecer as expressões regulares que descrevem cada tipo de lexema;
B. classificar os lexemas reconhecidos em tokens retornando as constantes definidas no arquivo tokens.h fornecido ou códigos ascii para caracteres simples;
C. incluir os identificadores e os literais (inteiros, reais, caracteres e strings) em uma tabela de símbolos global implementada com estrutura hash;
D. controlar o número de linha do arquivo fonte, e fornecer uma função declarada como int getLineNumber(void) a ser usada nos testes e pela futura análise sintática;
E. ignorar comentários de única linha e múltiplas linhas;
F. informar erro léxico ao encontrar caracteres inválidos na entrada, retornando o token de erro;
G. definir e atualizar uma variável global, e, uma função int isRunning(void), que mantém e retorna valor true (diferente de 0) durante a análise e muda para false (igual a 0) ao encontrar a marca de fim de arquivo;

Alfabeto

O Compilador deve ler um arquivo de entrada que contém símbolos válidos da linguagem. No caso do C--, estes símbolos são:

- Letras: **ab ... zAB ... Z**
- Dígitos: **0123456789**
- Símbolos Especiais: **, ; ( ) = < > + - * / % [ ] “ ‘ _ $ { } ? : ! . etc**
- Separadores**:** espaço, enter, tab

Tokens

Existem tokens que correspondem a caracteres particulares, como vírgula, ponto-e-vírgula, parênteses, para os quais é mais conveniente usar seu próprio código ascii, convertido para inteiro, como valor de retorno que os identifica. Para os tokens compostos, como palavras reservadas e identificadores, cria-se uma constante (#define em C ANSI) com um código maior do que 255 para representá-los.

Os tokens representam algumas categorias diferentes, como palavras reservadas, operadores de mais de um caractere e literais, e as constantes definidas no código do trabalho são precedidas por um prefixo para melhor identificar sua função, separando-as de outras constantes que serão usadas no compilador.

Palavras reservadas

As palavras reservadas da linguagem neste semestre são:

char, int, real, bool, if, then, else, while, input, output, return.

Para cada uma deve ser retornado o token correspondente.

Caracteres especiais

Os caracteres simples especiais empregados pela linguagem são listados abaixo (estão separados apenas por espaços), e devem ser retornados com o próprio código ascii convertido para inteiro.

Você pode fazer isso em uma única regra léxica. São eles:

, ; ( ) [ ] { } = + - * / % < > & | ~

Operadores Compostos

A linguagem possui, além dos operadores representados por alguns dos caracteres acima, operadores compostos, que necessitam mais de um caractere (somente dois) para serem representados no código fonte. São somente quatro operadores relacionais, conforme a tabela:

Representação original	Token retornado
<=	**OPERATOR_LE**
>=	**OPERATOR_GE**
==	**OPERATOR_EQ**
!=	**OPERATOR_DIF**

Identificadores

Os identificadores da linguagem são usados para designar variáveis, vetores e nomes de funções, são formados por uma sequência de um ou mais caracteres alfabéticos minúsculos ou maiúsculos e também os caracteres **'ponto'('.')** e **underline (´_´)**, e não podem conter dígitos em nenhuma posição;

Literais

- Literais são formas de descrever constantes no código fonte.
- Literais inteiros são formados por uma sequência de um ou mais dígitos decimais.
- Literais do tipo caractere são representados por um único caractere entre ***aspas simples*** (mais precisamente apóstrofo, ASCII decimal 39), como por exemplo: **'a', 'X', '-'**.
- Literais do tipo de dado real são definidos por dois literais decimais separados pelo caractere **'.'** (sem espaços, como, por exemplo, **"2.5"**).
- Literais do tipo string são quaisquer sequências de caracteres entre aspas duplas, como por exemplo **"meu nome"** ou **"Mensagem!"**, e servem apenas para imprimir mensagens com o comando **"output"**. Strings consecutivas não podem ser consideradas como apenas uma, o que significa que o caractere de aspas duplas não pode fazer parte de uma string. Para incluir os caracteres de aspas duplas e final de linha, devem ser usadas sequências de escape, como **"\""** e **"\n"**.

Comentários

Comentários de uma única linha começam em qualquer ponto com a sequência **"\\"** e terminam na próxima marca de final de linha, representada pelo caractere **'\n'**.

Comentários de múltiplas linhas iniciam pela sequência **"\\\"** e terminam pela sequência **"///"**, sendo que podem conter quaisquer caracteres, que serão todos ignorados, incluindo uma ou mais quebras de linha, as quais, entretanto, devem ser contabilizadas para controle do número de linha.

Caracteres em branco

Os caracteres de espaço, tabulação e nova linha são considerados como **"caracteres em branco"** e serão ignorados pelo analisador léxico da linguagem. Portanto, eles podem ocorrer entre quaisquer outros lexemas, e serão usados apenas para definir a disposição visual no editor e para separar diferentes lexemas entre si.

Análise Sintática

A sua análise sintática deve fazer as seguintes tarefas:

H. O programa principal deve receber um nome de arquivo por parâmetro e chamar a rotina yyparse para reconhecer se o conteúdo do arquivo faz parte da linguagem. Se concluída com sucesso, a análise deve retornar o valor 0 (zero) com exit(0);

I. imprimir uma mensagem de erro sintático para os programas não reconhecidos, informando a linha onde o erro ocorreu, e retornar o valor 3 como código genérico de erro sintático, chamando exit(3);

J. os nodos armazenados na tabela hash devem distinguir entre os tipos de símbolos armazenados, e o nodo deve ser associado ao token retornado através da atribuição para yylval.symbol;

Descrição Geral da Linguagem

Um programa na linguagem first2023 é composto por uma lista de declarações globais, que podem ser de variáveis ou funções, em qualquer ordem, mesmo intercaladas. Cada função é descrita por um cabeçalho seguido de seu corpo, sendo que o corpo da função é um bloco, como definido adiante. Os comandos podem ser de atribuição, controle de fluxo ou os comandos output e return. Um bloco também é considerado sintaticamente como um comando, podendo aparecer no lugar de qualquer comando, e a linguagem também aceita o comando vazio.

Declarações de variáveis globais

Cada variável é declarada pela sequência de seu tipo, nome, o sinal de igual e valor de inicialização, que é obrigatório, e pode ser um literal inteiro ou um literal caractere, para as variáveis char ou int, ou literal real para as variáveis do tipo real. Você deve optar, entretanto, por permitir qualquer tipo de literal para qualquer tipo de variável de forma a simplificar a descrição sintática nesta etapa. Isso significa que inicializar char ou int com literal real ou inicializar variável real com qualquer literal não serão considerados erros sintáticos.

Todas as declarações de variáveis são terminadas por ponto-e-vírgula (‘;’). A linguagem inclui também a declaração de vetores, feita pela definição de seu tamanho inteiro positivo entre colchetes, colocada imediatamente à direita do nome.

No caso dos vetores, a inicialização é opcional, e quando presente, será dada pela sequência de valores literais separados apenas por *espaços*, após os colchetes e antes do terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o tamanho do vetor. Vetores

também podem ser dos tipos char, int e real. Note que essa definição de “separados por espaços” significa que na sintaxe não há nenhum outro token entre eles, a separação pelos espaços já é dada pela análise léxica, e essa separação pode ter sido dada pelos caracteres ‘espaço’, ‘tabulação’, ‘quebra de linha’, ou mesmo pelo final da formação possível de tokens diferentes, como nas situações que irão ocorrer em outras listas da linguagem.

Definição de funções

Cada função é definida por seu cabeçalho seguido de seu corpo. O cabeçalho consiste no tipo do valor de retorno e o nome da função, seguido de uma lista, possivelmente vazia, entre parênteses, de parâmetros de entrada, separados por vírgula, onde cada parâmetro é definido por seu tipo e nome, não podem ser do tipo vetor e não têm inicialização.

O corpo da função é definido como um bloco. As declarações de funções não são terminadas por ‘;’.

Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência de comandos, terminados por ponto-e-vírgula. Um bloco de comandos é considerado como um comando, recursivamente, e pode ser utilizado em qualquer lugar que aceite um comando.

O bloco, entretanto, é uma exceção na lista de comandos (dentro do próprio bloco), por não exigir o ponto-e-vírgula como terminador.

Observe que para o terminador ‘;’ não aparecer após a ocorrência do bloco, ele terá que ser modelado nos comandos em si, e não irá aparecer na definição nem no final do bloco e nem no final dos comandos de controle de fluxo, já que estes devem terminar por outro comando (aninhado), o qual, terá ';' ou não (se for bloco).

Comandos Simples

Os comandos da linguagem podem ser: atribuição, construções de controle de fluxo, output, return, e comando vazio. O comando vazio segue as mesmas regras dos demais, **e deve ser terminado por ‘;’**. Isso significa que aparentemente pode haver um ‘;’ após um bloco, dentro de um bloco, pois isso representa que há um comando vazio após ele. Na atribuição usa-se uma das seguintes formas:

variável = expressão

vetor [expressão] = expressão

Os tipos corretos para o assinalamento e para o índice serão verificados somente na análise semântica.

O comando output é identificado pela palavra reservada output, seguida de uma lista de elementos separados por vírgula, onde cada elemento pode ser um string ou uma expressão. O comando return é identificado pela palavra reservada return seguida de uma expressão que dá o valor de retorno. Os comandos de controle de fluxo são descritos adiante.

Expressões Aritméticas

As expressões aritméticas têm como folhas identificadores, opcionalmente seguidos de expressão inteira entre colchetes, para acesso a posições de vetores, ou podem ser literais numéricos e literais de caracteres. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para a associatividade.

Os operadores válidos são: **+,-,*,/,<,>,<=,>=,==,!=,&,|,~,** listados já no trabalho anterior. Nesta etapa, ainda não haverá verificação ou consistência entre operadores e operandos. A descrição sintática deve aceitar qualquer operador e sub-expressão de um desses tipos como válidos, deixando para a análise semântica verificar a validade dos operandos e operadores.

Outra expressão possível é uma chamada de função, feita pelo seu nome, seguido de lista de argumentos entre parênteses, separados por vírgula, onde cada argumento é uma expressão, como definido aqui, recursivamente.

Outra expressão possível é a ocorrência da palavra reservada input, seguida de um tipo entre parênteses, que significa que o programa pedirá um valor para o stream de entrada padrão do processo;

Comandos de Controle de Fluxo

Para controle de fluxo, a linguagem possui as três construções estruturadas listadas abaixo.

if (expr) comando

if (expr) comando else comando

if (expr) loop comando

Tipos e Valores na Tabela de Símbolos

A tabela de símbolos até aqui poderia representar o tipo do símbolo usando os mesmos **#defines** criados para os tokens (agora gerados pelo yacc). Mas logo será necessário fazer mais distinções, principalmente pelo tipo dos identificadores. Assim, é preferível criar códigos especiais para símbolos, através de definições como:

```
#define SYMBOL_LIT_INTE 1

#define SYMBOL_LIT_CARA 2

...

#define SYMBOL_IDENTIFIER 7
```

Controle e Organização do Seu Código fonte

O arquivo tokens.h deve ser usado na etapa de análise léxica. Neste caso, você deve manter o arquivo tokens.h intacto, e separar a sua função main em um arquivo especial chamado main.c, já que a função main não pode estar contida no código de scanner.l. Você deve usar essa estrutura de organização, manter os nomes tokens.h. e scanner.l

Para a etapa de análise sintática você deve seguir as demais regras especificadas na análise léxica, entretanto. A função main escrita por você agora será usada sem alterações para os testes da etapa de análise sintática e seguintes. Você deve utilizar um Makefile para que seu programa seja completamente apagado com make clean e compilado com o comando make. O formato de entrega será o mesmo do trabalho 1, e todas as regras devem ser observadas.

Equipes

O trabalho pode ser feito em grupos com no máximo 2 alunos.

O que deve ser entregue:

Além da entrega do código fonte, na plataforma do colabweb, o aluno deverá produzir:

1. **Manual do usuário** (uma página) Num arquivo chamado **mu.txt** ou **mu.doc**, contendo uma explicação de como se utilizar o analisador (explicar o formato da entrada e da saída do programa).
2. **um vídeo (ou mais)** explicando como as funções foram pensadas. As principais funções devem também ser explicadas. No vídeo, há a necessidade de apresentar o código fonte, a compilação e a execução do programa. Não há necessidade do aluno aparecer no vídeo.

Os vídeos devem ser postados em uma plataforma de vídeo (youtube, por exemplo), de modo que o professor possa acessar, e fazer parte da avaliação do trabalho. Para gravar pode usar serviços gratuitos e online. Por exemplo:

https://online-screen-recorder.com/pt

https://www.veed.io/pt-BR

https://streamyard.com/

ou se quiser, há aplicativos que podem ser baixados:

https://www.movavi.com/pt/learning-portal/gravadores-de-video.html

Atenção: Os vídeos não precisam ter alta produção. Para subir os vídeos para o Youtube, há uma necessidade de ajuste no seu perfil do youtube, com pelo menos 24h de antecedência. Portanto, sugiro que esse ajuste seja feito brevemente.

A nota será composta assim

- De 0 a 0,5 pontos pelo estilo de programação (nomes bem definidos, lugares de declarações, comentários).
- De 0 a 0,5 ponto pela compilação.
- De 0 a 0,5 ponto pelo formato de apresentação dos resultados
- De 0 a 7 pontos pela solução apresentada.
- De 0 a 1,5 pontos pelas informações dos vídeos.

Comentários Gerais

1. Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.
2. Clareza, indentação e comentários no programa também vão valer pontos.
3. Trabalhos copiados serão penalizados conforme anunciado

Verifique regularmente os documentos e mensagens da disciplina para informar-se de alguma eventual atualização que se faça necessária ou dicas sobre estratégias que o ajudem a resolver problemas particulares. Em caso de dúvida, consulte o professor.

Gramática

Declarações

1. programa → lista-decl lista-com
2. lista-decl → lista-decl decl | decl
3. decl → decl-var | decl-func
4. decl-var → espec-tipo var ;
5. espec-tipo → INT | VOID | FLOAT
6. decl-func → espec-tipo ID (params) com-comp
7. params → lista-param | void | ε
8. lista-param → lista-param , param | param
9. param → espec-tipo var
10. decl-locais → decl-locais decl-var | ε

Comandos

11. lista-com → comando lista-com | ε
12. comando → com-expr | com-atrib | com-comp | com-selecao | com-repeticao | com-retorno
13. com-expr → exp ; | ;
14. com-atrib → var = exp ;
15. com-comp → { decl-locais lista-com }
16. com-selecao → IF (exp) comando | IF (exp) com-comp ELSE comando
17. com-repeticao → WHILE (exp) comando ;
18. com-retorno → RETURN ; | RETURN exp ;

Expressões

19. exp → exp-soma op-relac exp-soma | exp-soma
20. op-relac → <= | < | > | >= | == | !=
21. exp-soma → exp-soma op-soma exp-mult | exp-mult
22. op-soma → + | -
23. exp-mult → exp-mult op-mult exp-simples | exp-simples
24. op-mult → * | / | %
25. exp-simples → (exp) | var | cham-func | literais

26. literais → NUM | NUM.NUM

27. cham-func → ID (args)

28. var → ID | ID [NUM]

29. args → lista-arg | ε

30. lista-arg → lista-arg , exp | exp