BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

UFMG-ICEX-DCC

COMPILADORES

2^{o} semestre de 2019

Trabalho Prático

Data de Entrega: 26/11/2019

Valor: 40 pontos

1 Compilador para a linguagem MLM

Considere a Mini-Linguagem M (MLM) definida a seguir.

program	::=	**program** identifier ";" decl_list compound_stmt
decl_list	::=	decl_list " ;" decl
	|	decl
decl	::=	ident_list ":" type
ident_list	::=	ident_list " ," identifier
	|	identifier
type	::=	**integer**
	|	**real**
	|	**boolean**
	|	**char**
compound_stmt	::=	**begin** stmt_list **end**
stmt_list	::=	stmt_list ";" stmt
	|	stmt
stmt	:=	assign_stmt
	|	if_stmt
	|	loop_stmt
	|	read_stmt
	|	write_stmt
	|	compound_stmt
assign_stmt	::=	identifier " :=" expr
if_stmt	::=	**if** cond **then** stmt
	|	**if** cond **then** stmt **else** stmt
cond	::=	expr
loop_stmt	::=	stmt_prefix **do** stmt_list stmt_suffix
stmt_prefix	::=	**while** cond
	|	$\mathcal{E}$

stmt_suffix	::=	**until** cond
	|	**end**
read_stmt	::=	**read** "(" ident_list ")"
write_stmt	::=	**write** "(" expr_list ")"
expr_list	::=	expr
	|	expr_list " ," expr
expr	::=	simple_expr
	|	simple_expr RELOP simple_expr
simple_expr	::=	term
	|	simple_expr ADDOP term
term	:=	factor_a
	|	term MULOP factor_a
fator_a	:=	"−" factor
	|	factor
factor	:=	identifier
	|	constant
	|	" (" expr ")"
	|	NOT factor
constant	::=	integer_constant
	|	real_constant
	|	char_constant
	|	boolean_constant
boolean_constant	:=	**false** | **true**

Considere as seguintes convenções léxicas:

1. Identificadores são definidos pelas seguintes expressões regulares:

letter	::=	A | B | $\cdots$ Z
	|	a | b | $\cdots$ z
digit	::=	**0** | **2** | **3** | **4** |**5** | **6** | **7** | **8** | **9**
identifier	::=	letter (letter | digit)$\star$

   Na implementação pode-se limitar o tamanho do identificador.

2. Constantes são definidas da seguinte forma:

unsigned_integer	::=	digit digit$\star$
sign	::=	+ | − | $\mathcal{E}$
scale_factor	:=	"E" sign unsigned_integer
unsigned_real	::=	unsigned_integer ($\mathcal{E}$ | "." digit$\star$)($\mathcal{E}$ | scale_factor)
integer_constant	::=	unsigned_integer
real_constant	::=	unsigned_real
char_constant	::=	"'" caractereASCII "'"

3. Os operadores de relação (RELOP's) são:

RELOP	::=	= | < | $\leq$ | > | $\geq$ | != |NOT

4. Os operadores de adição (ADDOP's) são:

$$\text{ADDOP} \quad ::= \quad + \mid - \mid \textbf{or}$$

5. Os operadores (MULOP's) são:

$$\text{MULOP} \quad ::= \quad \star \mid / \mid \textbf{div} \mid \textbf{mod} \mid \textbf{and}$$

6. A linguagem MLM é um subconjunto bastante reduzido da linguagem Pascal. A sua semântica supomos ser óbvia, mas os casos de dúvida prevalesce a semântica do Pascal.

7. O comando *loop* é uma combinação dos comandos *while* e *repeat* do Pascal. No caso de se omitir ambas as condições de fim de loop, um *loop infinito ocorre*.

2 Descrição das Fases de Compilação

Nesta seção são descritas cada uma das fases do compilador.

2.1 Análise Léxica

Esta fase é responsável pelo agrupamento de seqűências de caracteres em *tokens*. Para implementá-la, pode ser utilizada a ferramenta JLex. Essa ferramenta recebe como entrada o arquivo de especificação MLM.lex. Esse arquivo contém a especificação do analisador léxico a ser construído e é organizado em três seções, separadas pela diretiva %%, como mostrado a seguir:

Código do usuário

%%
Diretivas JLex
%%
Expressões regulares

onde:

Código do usuário: este código é copiado no topo do arquivo fonte do analisador léxico a ser gerado, sendo útil, por exemplo, para declaração de pacote ou importação de classes externas.

Diretivas JLex: esta seção engloba diversas diretivas para definição de macros, declaração de estados, além de customizações do analisador.

Expressões regulares: consiste em uma série de regras responsáveis pela quebra da entrada em *tokens*.

Para a geração do analisador léxico, executa-se o comando `java JLex.Main MLM.lex` de dentro do diretório `Parse` e obtêm-se o arquivo `MLM.lex.java`, o qual é renomeado para `Ylex.java` para manter a compatibilidade com o restante do programa.

Para gerar o arquivo Ylex.class o projetista do compilador deve executar o comando `javac Parse/Ylex.java` de dentro do diretório mais externo,

Por exemplo:
`/ mariza/Cursos/CompiladoresI/Geral/Projeto2019-2/Proj2019-2/javac Parse/Ylex.java`

2.2 Análise Sintática

Esta fase é responsável pelo agrupamento dos tokens produzidos pela análise léxica em frases gramaticais. A tarefa de construir tabelas sintáticas LR(1) [1] ou LALR(1) [1] é uma tarefa muito simples de ser automatizada, de modo que ela raramente é implementada de outra forma que não por meio do uso de ferramentas geradoras de analisadores sintáticos. CUP 3.2.2 é uma ferramenta para gerar analisadores sintáticos, similar à difundida Yacc [5]..

CUP recebe como entrada um arquivo contendo a especificação da gramática para a qual é necessária a construção do analisador, MLM.cup, juntamente com rotinas responsáveis pela invocação do analisador léxico. A especificação desse arquivo possui um preâmbulo, seguido pelas regras gramaticais. No preâmbulo são declarados os terminais e não-terminais da gramática, além de ser especificado de que modo o analisador léxico se comunica com o sintático. As regras gramaticais são produções da forma

exp ::= exp ADDOP exp,
onde *exp* é um não-terminal e *ADDOP* é um terminal conforme a definição dada.

Esta fase do compilador produz como saída um conjunto de arquivos contendo o código referente ao analisador sintático, os quais, após compilação e execução, informam se a linguagem de entrada para o compilador está ou não com a sintaxe correta.

Em MLM, para a geração do analisador pode ser utilizada a ferramenta CUP em conjunto com o arquivo MLM.cup, invocando o comando

```
java java_cup.Main -parser Grm -expect 6 < MLM.cup > Grm.out 2> Grm.err
```

de dentro do diretório `Parse` para obter os arquivos `Grm.java` e sym.java. O arquivo `CUP$Grm$actions.class` é gerado na fase se análise semântica.

A diretiva `-expect 6` é usada para ignorar os *warnings* durante a geração do analisador.

Para gerar os arquivos `Grm.class` e `sym.class` o projetista do compilador deve executar os comandos `javac Parse/sym.java` e `javac Parse/Grm.java` de dentro do diretório mais externo.

Por exemplo:
`/ mariza/Cursos/CompiladoresI/Geral/Projeto2019-2/Proj2019-2/javac Parse/Grm.java`

2.3 Análise Semântica e Código Intermediário

Esta fase é responsável pela verificação de erros semânticos no programa fonte, além de realizar a captura de informações de tipo para a etapa subseqűente de geração de código.

A verificação de tipos consiste em verificar se todo termo possui o tipo para ele esperado.

Uma representação intermediária de código é uma forma de se escrever código de baixo nível não atrelado a nenhuma linguagem ou máquina específica, garantindo a portabilidade do compilador.

Para a implementação do código intermediário, deve ser gerada quádruplas.

Para gerar o arquivo`CUP$Grm$actions.class` o projetista do compilador deve executar o comando `javac Parse/Grm.java` de dentro do diretório mais externo, por exemplo:

`/ mariza/Cursos/CompiladoresI/Geral/Projeto2019-2/Proj2019-2/javac Parse/Grm.java`

durante a análise sintática, uma vez que o arquivo `CUP$Grm$actions.class` está relacionado com as rotinas semânticas associadas às produções da gramática.

2.4 Compilador Completo

Esta fase é responsável por traduzir o código intermediário para um interpretador, pode ser a Máquina Virtual de Java, de forma a ser possível executar programas em MLM.

3 Metodologia para o Desenvolvimento do seu Compilador MLM

A sua tarefa é construir um **compilador** para a mini-linguagem MLM. O compilador deve gerar código para a linguagem intermediária de quádruplas, cujas instruções disponíveis devem ser definidas conforme a necessidade. Você deve traduzir o resultado do *front-end* do compilador para os *bytecodes* da Máquina Virtual Java, de forma a ser possível executar programas em MLM. O seu compilador para MLM pode ser implementado na linguagem C, C++ ou Java, mas antes de começar a projetá-lo, você deve se familiarizar com a linguagem MLM.

3.1 Interface com o Usuário

A execução do compilador pode ser feita a partir de qualquer plataforma que contenha um interpretador *java* instalado.

Para executar o compilador deve-se digitar a seguinte linha de comando:

`java Main.Main <NOME DO ARQUIVO> [-opções]` onde:

`<NOME DO ARQUIVO>`**:** é o arquivo contendo o programa MLM a ser compilador e as opções podem ser:

listinput: imprime na saída padrão a listagem do arquivo de entrada.
listparse: imprime na saída padrão os estados percorridos pelo analisador sintático.
intermcode: imprime na saída padrão o código intermediário gerado.

Para redirecionar a saída das opções para um arquivo basta digitar a seguinte linha de comando:

`java Main.Main <NOME DO ARQUIVO> [-opções] > <nome-arquivo-saida>`

onde:
`<nome-arquivo-saida>`: é o arquivo contendo a saída das opções requisitadas.

3.2 Ferramentas de Apoio

3.2.1 JLex

A ferramenta JLex é um gerador de analisador léxico para JAVA que recebe como entrada um arquivo com a especificação léxica da linguagem na forma de expressões regulares e gera o código fonte JAVA correspondente ao analisador léxico.

Esta ferramenta encontra-se disponível no endereço [6] em formato de código fonte JAVA e, portanto, deve ser compilada antes de sua execução. Deve-se, ainda, ajustar-se o CLASSPATH para que a mesma possa ser referenciada a partir de qualquer caminho do ambiente em questão. A seguir apresentamos como isto pode

ser feito em ambiente Unix com bash `cshell`. Supondo que os arquivos correspondentes ao JLex estejam no caminho `/java/JLex`, inclua no arquivo `.cshrc` a diretiva:

`setenv CLASSPATH ./:/java` Para executar o JLex entre com a linha de comando:

```
java JLex.Main <NOME DO ARQUIVO>
```

onde `<NOME DO ARQUIVO>` é o arquivo de especificação léxica para a linguagem em questão. Como resultado da execução desta linha de comando, será gerado o arquivo `<NOME DO ARQUIVO>.java` se não houver erros no arquivo de entrada.

3.2.2 CUP

A ferramenta CUP é um gerador de analisador sintático LALR para JAVA. Como resultado, ela gera o código fonte JAVA correspondente ao analisador sintático.

Esta ferramenta pode ser obtida no endereço [7] e deve ser compilada pelo interpretador JAVA antes de ser executada. Tal como na ferramenta JLex, ela deve ser incluída no CLASSPATH para ser referenciada a partir de qualquer caminho do ambiente. Supondo-se que a ferramenta foi instalada no caminho `/java/java_cup`, podemos referenciar a variável CLASSPATH da seguinte forma:

`setenv CLASSPATH ./:/java` [1]

Para executar o CUP entre com a linha de comando:

```
java java_cup.Main < <NOME DO ARQUIVO>
```

onde `<NOME DO ARQUIVO>` é o arquivo contendo a especificação da gramática para a linguagem em questão. Não havendo erros durante a execução do CUP, são gerados dois arquivos fonte em JAVA, `parser.java` e `sym.java`, além do arquivo binário `CUP$Grm$actions.class`.

4 Trabalho Prático: Compilador MLM

4.1 TP1: Analisador Léxico

Cada aluno de compiladores deve implementar um analisador léxico para a linguagem MLM. O aluno deve entregar para essa fase do compilador:

- O arquivo fonte Ylex.lex e demais arquivos se existir.
- A documentação.

A documentação deve conter:

[1] o caminho `/java/java_cup` descrito é aquele que contém o arquivo `Main.class`.

- Nome dos componentes do grupo.
- Relato dos erros encontrados na especificação, se houver.
- Questões não solucionadas nessa etapa do trabalho.
- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar o analisador léxico, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Descrição do objetivo de cada classe, método em seu programa. Esta parte deve ser organizada hierarquicamente.

4.2 TP2: Analisador Sintático

Cada aluno de compiladores deve implementar neste segundo trabalho prático um analisador sintático para a linguagem MLM de acordo com a gramática fornecida neste texto. Essa parte do compilador está coberta nas páginas 83-84 do livro de Appel [2]. Pode ser útil ler a Seção 3.4 desse mesmo livro.

Nessa parte do trabalho o aluno deve entregar:

- O arquivo fonte Grm.cup e demais arquivos fontes necessários no seu programa.
- A documentação.

A documentação deve conter:

- Nome dos componentes do grupo.
- Relato dos erros encontrados na especificação, se houver.
- Questões não solucionadas nessa etapa do trabalho.
- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar essa etapa do programa, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Descrição do objetivo de cada classe, método em seu programa.
- Listagem de cada conflito *shift-reduce*, se houver, na gramática com uma explicação de como cada um deles foi resolvido e porque a solução adotada é uma solução correta.

4.3 TP3: Análise Semântica e Geração de Código Intermediário

Cada aluno de compiladores deve fazer neste terceiro trabalho prático a análise semântica e a geração do código intermediário. Essa fase cobre a construção das quádruplas, e a verificação de tipos, geração de código intermediário para a linguagem MLM de acordo com a gramática fornecida nesse texto.

Nessa parte do trabalho o aluno deve entregar:

- Todos arquivos fontes necessários para rodar seu programa.
- A documentação.

A documentação deve conter:

- Nome dos componentes do grupo.
- Relato dos erros encontrados na especificação, se houver.

- Questões não solucionadas nessa etapa do trabalho.
- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar essa etapa do programa, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Descrição do objetivo de cada classe, método em seu programa.
- Uma explicação de como foi tratado os comandos `loop` *while* ou *repeat*.
- Listagem dos fontes: Semantic.java e Env.java.

4.4 TP4: Compilador MLM + Resultados da Execução dos Testes

Cada aluno de compiladores deve fazer neste quarto trabalho prático a tradução das quádruplas para as instruções de um Interpretador, pode ser JVM, e mostrar os resultados dos testes submetidos ao compilador MLM. Outras opções de interpretadores são: TAM e MEPA. Ambos estão disponíveis na página do curso.

Independente da opção do interpretador escolhido, as quádruplas devem ser percorridas a partir da primeira, de modo que para cada uma seja produzida a instrução correspondente a do interpretador escolhido.

Nessa parte do trabalho o aluno deve entregar:

- Todos arquivos fontes necessários para rodar seu programa.
- A documentação.

A documentação deve conter:

- Nome dos componentes do grupo.
- Relato dos erros encontrados na especificação, se houver.
- Questões não solucionadas nessa etapa do trabalho.
- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar essa etapa do programa, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Listagem de todos os fontes.

5 Conclusões

Boa sorte para vocês.

A Máquina Virtual JAVA

A Máquina Virtual JAVA(MVJ) [8] é uma máquina abstrata que pode ser implementada como um interpretador, um compilador ou mesmo em *hardware*, conferindo grande flexibilidade à arquitetura na qual ela será aplicada.

A relação que existe entre a MVJ e a linguagem JAVA se dá via um arquivo denominado *class*. Este arquivo contém as instruções da MVJ, ou *bytecodes*, uma tabela de símbolos, e algumas informações auxiliares. Para fins de segurança, seu formato é bastante rígido e restrições estruturais são impostas a seu código. Apesar disso, qualquer linguagem que possa ser expressa em termos de um arquivo *class* válido pode ser hospedada pela MVJ. Este arquivo possui todas as definições necessárias para a execução de código na máquina virtual, e nele encontramos uma lista ordenada, estruturada, que define variáveis, constantes, objetos com seus atributos e métodos representados na forma de tabelas e seqűência de *opcodes* e operandos.

O ciclo de vida de uma classe é dividido em duas partes: carregamento e execução. O carregamento é composto de três fases:

1. Carregamento: carrega o fluxo binário contido no arquivo *class* para a estrutura interna de dados.
2. Ligamento: [2] esta fase subdivide-se em três partes:
    a. Verificação: assegura que o arquivo obedeça à semântica da linguagem JAVA e não viole a integridade da máquina virtual.
    b. Preparação: aloca memória para as variáveis de classe, inicializando-as com valores `default`.
    c. Resolução: efetua trocas de referência, mudando-as de simbólica para direta. É opcional, já que pode ser executada no momento de referência ao símbolo.
3. Inicialização: liga as variáveis de classe previamente alocadas com os valores determinados pelo programador.

Na execução é feita a interpretação e execução dos *bytecodes*. Para armazenar todos os dados da execução de uma aplicação é definida uma área de execução de dados da seguinte forma: área de métodos, heap, pilhas JAVA, registradores PC, pilhas de métodos nativos.

A área de métodos e o *heap* são compartilhados por todos os *threads*, sendo que na área de métodos são armazenadas as informações sobre as classes do programa e no *heap* cada objeto instanciado.

Cada *thread* possui um conjunto de registradores PC, *stack pointer*, uma pilha, etc. Esta pilha armazena *frames*, os quais são compostos por variáveis locais, parâmetros de chamada, valores de retorno e resultados intermediários.

A MVJ não possui registradores de uso geral, sendo a pilha a responsável por armazenar valores intermediários. Esta técnica foi adotada em virtude da proposta de independência de plataforma, de modo que quanto menos registradores fossem exigidos, maior seria o número de arquiteturas capazes de permitir a execução da máquina virtual.

Para realizar a execução, um conjunto de instruções é tomado como base, sendo que a maioria das instruções envolve operação de pilha, uma vez que a MVJ é baseada em pilha.

Os tipos de dados sobre os quais a MVJ opera são os seguintes:

- Primitive Types
    - Numeric Types
        - Floating-Point Types
            - float
            - double
        - Integral Types
            - byte
            - short
            - int
            - long
            - char
    - returnAddress
- Reference Types
    - reference
        - class types
        - interface types
        - array types

[2] do inglês Linkage

Referências

[1] Aho, A. V.; Sethi, R.; Ullman, J.D., *Compilers Principles, Techniques, and Tools*, Addison Wesley, 1986.

[2] Appel, Andrew W., *Modern Compiler Implementation in JAVA*, Cambridge University Press, 1997.

[3] Entsminger, Gary, *The Way of JAVA*, Prentice Hall, Inc, 1997.

[4] Meyer, Bertrand, *Object-oriented Software Construction*, Prentice Hall, C.A.R. Hoare, 1998.

[5] Johnson, S.C., *Yacc-yet another compiler compiler*, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[6] Berk, Elliot, *JLex: A Lexical Analyser Generator for* $JAVA^{(TM)}$, http://www.cs.princeton.edu/~appel/modern/java/JLex, 1997.

[7] Hudson, Scott, *LALR Parser Generator for* $JAVA^{TM}$, http://www.cs.princeton.edu/~appel/modern/java/CUP, 1998.

[8] http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

[9] Escalda, G. e Bigonha, M., *Um compilador para a linguagem TIGER usando a Linguagem Java como Ferramenta de Desenvolvimento*, Relatório Técnico do Laboratório de Linguagens de Programa'cão, LLP11/99.

[10] Bigonha, Mariza A. S., SIC: Sistema de Implementação de Compiladores, Tese de Mestrado, DCC/UFMG, 1984.