Universidade Federal de Roraima
Departamento de Ciência da Computação
Introdução à Compiladores
Prof.: Dr. Luciano Ferreira
Aluno: Felipe Derkian de Sousa Freitas

Compiler Expressions

BOA VISTA - RR, 09 DE DEZEMBRO DE 2020

INTRODUÇÃO

O projeto compiler expresssions trata-se do desenvolvimento de um projeto de compilador que faz as seguintes etapas:

- Análise léxica.
- Análise sintática.
- Análise semântica.
- Geração de código intermediário.
- Otimização do código intermediário.
- Geração de código nas instruções da arquitetura de processador escolhido.

O compiler expressions faz apenas o tratamento de expressões matemática contendo as seguintes operações básica:

- Adição.
- Subtração.
- Divisão.
- Multiplicação.

ARQUITETURA DO COMPILER EXPRESSIONS

O compiler expressions foi desenvolvido com a linguagem C++ sem a presença de uma interface gráfica pois foi desenvolvido seguindo o modelo do compilador para linguagem C e C++ sendo respectivamente gcc e g++ onde os mesmos são executados diretamente pelo terminal dando ao usuário maior usabilidade pela simples chamada no terminal.

ARQUIVOS DO COMPILER EXPRESSIONS

O projeto foi dividido em módulos sendo eles:

- **main.cpp:** Inicializa o sistema verificando os parâmetros enviados no argumento da chamada do programa. Então faz as verificações e posteriormente chama as funções de leitura de arquivo presente na scannerFile e depois chama o analisador léxico.
- **scannerFile.h / scannerFile.cpp:** Faz a leitura do arquivo e adiciona todos os caracteres lidos do arquivo em uma fila para ser processada sob demanda posteriormente pelo analisador léxico.
- **lexico.h / lexico.cpp:** Apresentam as funções para realizar a análise léxica seguindo o automato desenhado para o reconhecimento da linguagem.

  O analisador léxico reconhece na gramática regular:

    - identificadores,
    - números,
    - atribuição,
    - final de expressão,
    - operadores matemáticos.

- **sintatico.h / sintatico.cpp:** Apresentam as funções para realizar a análise sintática da linguagem livre de contexto utilizando como base a análise preditiva.
- **semantica.h / semantica.cpp:** Apresentam as funções para realizar a análise semântica onde são verificadas as declarações das variáveis utilizadas nas funções como parâmetro.
- **geracaoCodigo.h / geracaoCodigo.cpp:** Apresentam as funções para realizar a geração de código intermediário, ou seja, ele transforma as expressão em quadruplas e também contém a função de otimização do código intermediário fazendo a substituição dos valores declarados diretamente nos parâmetros das fórmulas.
- **CodigoArquiteturaMIPS.h / codigoArquiteturaMIPS.cpp**: Contém as funções que convertem o código otimizado localizado nas quadruplas em código da arquitetura do processador MIPS.
- **errors.h / errors.cpp:** Arquivo que contém as funções para apresentar ao usuário os erros capturados nas etapas do analisador léxico, sintático e semântico.
- **temporario.h / temporario.cpp:** Contém as funções para geração de variáveis no padrão MIPS dadas sequencialmente sob demanda.

FORMATO DE ARQUIVO ACEITO PELO COMPILADOR

O compiler expressions aceita somente arquivos no formato **.txt** para serem analisados.

ex.: linguagem.txt

COMPILAÇÃO DOS ARQUIVOS FONTES DO COMPILADOR

Para realizar a compilação dos arquivos fontes do compiler expressions é necessário o compilador para linguagem C++ chamada g++ estando instalada corretamente no computador e um terminal para rodar os comandos de compilação e de execução do compilador:

“**g++ main.cpp scannerFile.h scannerFile.cpp errors.h errors.cpp lexico.h lexico.cpp sintatico.h sintatico.cpp semantica.h semantica.cpp geracaoCodigo.h geracaoCodigo.cpp temporario.h temporario.cpp codigoArquiteturaMIPS.h codigoArquiteturaMIPS.cpp -o compiladorExpressions**”

Ao executar o comando na pasta com os arquivos do compilador é necessário rodar o seguinte comando para executar o programa e realizar à verificação das expressões informadas no arquivo .txt:

“./**compiladorExpressions nome_do_arquivo_fonte.txt**”

e o programado executará e compilará para a linguagem mips as expressões informadas para análise.

CAMADAS DO COMPILADOR

ESCÂNER

O escâner faz a leitura do arquivo e enfileira para ser processado posteriormente pelo analisador léxico, isso sendo chamado sob demanda para que os caracteres sejam identificados pelo analisador léxico e então sejam categorizados seguindo a gramática regular.

ANALISADOR LÉXICO

O analisador léxico tem a função de pegar os caracteres da fila do escâner e categorizá-los sendo que eles podem ser categorizados como:

- **Numeral inteiro:** A expressão regular equivalente em automato para reconhecer o valor inteiro seria “**(0-9)+**”.
- **Identificador:** A expressão regular equivalente em automato para reconhecer o identificador (variável) seria “**(a-z | A-Z)(a-z | A-Z | 0-9)***” , podendo começar com qualquer letra seguida de letra ou número.
- **Operações matemáticas:** A expressão regular equivalente em automato para reconhecer as operações é “**(+ | - | / | *)**”
- **Sinal de atribuição:** A expressão regular equivalente em automato para reconhecer o sinal de atribuição seria “**:=**”
- **Sinal de fim de expressão:** A expressão regular equivalente em automato para reconhecer o final de expressão matemática seria “**;**”

EXEMPLOS DE EXPRESSÕES VÁLIDAS

Seguem alguns exemplos de expressões válidas no compiler expressions:

- a := b + c;
- c := b * 10 + d;

Onde o analisador léxico reconheceria da seguinte forma:

- **a, b, c, d :** Identificadores.
- **+, * :** Operadores.
- **:= :** Atribuição
- **; :** Final de expressão

EXEMPLOS DE EXPRESSÕES VÁLIDAS

Seguem alguns exemplos de expressões inválidas no compiler expressions:

- a := b + c → Falta o símbolo de final de expressão.
- a : b + c; → Sinal de atribuição inválido.
- a := b c; → Falta o operador aritmético.

LINGUAGEM FORMAL DO ANALISADOR LÉXICO

Como o analisador léxico é um autômato de finito determinístico (AFD) então pode-se representar formalmente por uma quíntupla **(Q,Σ,δ,q0,F)**, onde:

- **Q** é um conjunto finito de estados. { **q0, q1, q2, q3, q4, q5, q6** }
- **Σ** é um conjunto finito de símbolos, chamado de alfabeto do autômato. { **=, : , ; , +, -, *, /, (a – z), (A - Z), (0-9)** }
- **δ** é as funções de transições:
  {
  q0 x ; → q1 ,
  q0 x (0-9) → q2,
  q2 x (0-9) → q2,
  q0 x (a-z | A-Z) → q3,
  q3 x (a-z | A-Z | 0-9) → q3,
  q0 x (+| -| *| /) → q4,
  q0 x : → q5,
  q5 x = → q6,
  q0 x ‘ ‘ → q0
  }
- **q0** é o estado inicial neste caso é **q0.**
- **F** é um conjunto de estados de Q (isto é, F ⊆ Q) chamado de estados de aceitação. { **q1, q2, q3, q4, q6** }

AFD DO ANALISADOR LÉXICO

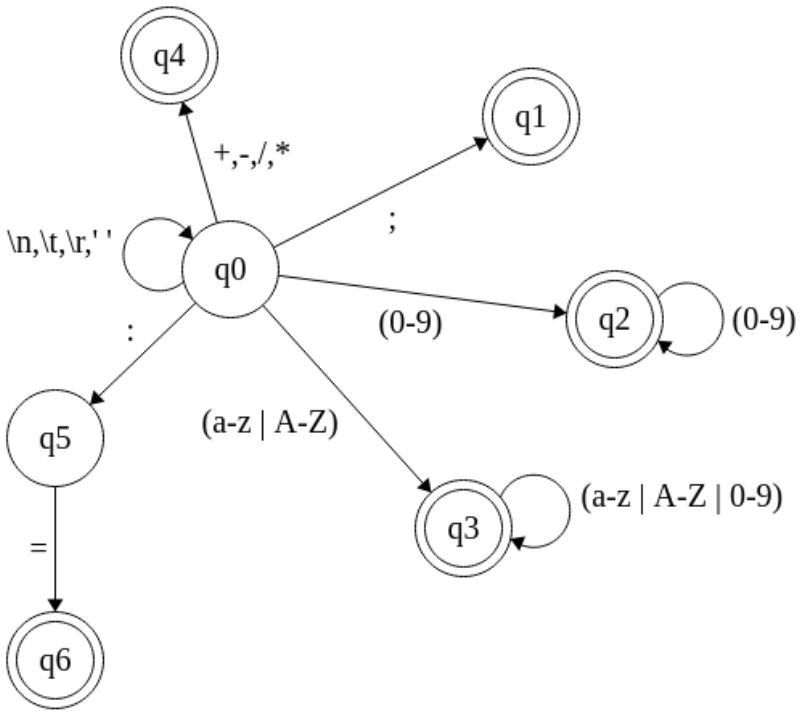

Figura 1: Autômato do analisador léxico da linguagem

O autômato acima mostram os estados do analisador léxico com seus estados terminais em formato de anel que são os estados onde são reconhecidos os lexemas identificados que são chamados de estados finais do autômato finito determinístico.

Caso seja encontrado algum caractere inválido que não pertence a linguagem ou um caractere não esperado na sequencia do automato finito é disparado uma mensagem de erro léxico informando o usuário do ocorrido.

ANALISADOR SINTÁTICO

O analisador sintática precisa ser formado por uma linguagem livre de contexto sem recursão a esquerda para não haver ambiguidade sendo indicada a forma de análise preditiva ou de precedência fraca.

Para esse trabalho optou-se pelo analisador preditivo por conta da facilidade de implementação em comparação com a de precedência fraca para o reconhecimento de expressões aritméticas.

Caso seja encontrado alguma produção que não esteja na sequência correta da regra da gramática livre de contexto estabelecida para o compiler expressions é detectado como um erro sintático por não seguir a regra da gramática e então é disparado uma mensagem de erro sintático informando o usuário do ocorrido.

GRAMÁTICA G LIVRE DE CONTEXTO

A gramática abaixo mostra as produções da linguagem, porém não podem ser utilizadas como analisadores sintáticos preditivo por conta das recursões a esquerda que devem ser eliminadas podendo ter apenas recursões a direita tornando-a uma gramática preditiva do tipo LL(1).

G = (VN, VT, P, S)

VN = E, T, OP, I, FE, T’ → NÃO TERMINAIS

VT = id, numero, +, -, /, *, :=, ; → TERMINAIS

S = E → ESTADO INICIAL

P → PRODUÇÕES

PRODUÇÕES

E → T' I (E OP T | T) FE

T' → id

T → id | numero

OP → + | - | * | /

I → :=

FE → ;

GRAMÁTICA G' LL(1)

Com a gramática sem as recursões a esquerda, temos uma gramática para análise preditiva do tipo LL(1) que não possui ambiguidade por ter somente recursões a direita e compre com o objetivo de ser livre de contexto que reconhecerá a estrutura de formação das expressões informadas.

1) E → T' I T E' FE

2) E' → OP T E'

3) T → id | numero

4) OP → + | - | * | /

5) T' → id

6) I → :=

7) FE → ;

Caso seja encontrado alguma produção que não esteja na sequência correta dos tipos dos tokens de formação das expressões, ou seja as regras gramaticais, é detectado um erro sintático sendo disparado uma mensagem de erro sintático informando o usuário do ocorrido.

ANALISADOR SEMÂNTICO

O analisador semântico tem como objetivo, neste trabalho, verificar a declaração das variáveis previamente ao serem utilizadas nas expressões reconhecidas e validadas pelo analisador sintático. Neste caso, para poder esta validado uma expressão matemática que contenham identificadores presentes nas expressões, ele deve estar inicializado com algum valor que será substituído na expressão posteriormente.

Portanto, ele fica responsável em validar a declaração das variáveis utilizadas nas expressão, onde a falta de declaração acarreta em erro de semântica por falta de declaração da mesma.

Caso seja encontrado alguma variável que não esteja declarada então é detectado um erro semântico e então é disparado uma mensagem de erro informando o usuário do ocorrido.

GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

A geração de código tem o objetivo de transformar as expressão em fragmentos de código menores que são armazenados em **quadruplas**, neste trabalho, que são tabela que contém espaço para o **operador, resultado, argumento1 e argumento2**.

Para cada fragmento de código, o código é armazenado de forma que será realizado o processo de otimização desse código gerado, podendo este receber certas otimizações.

CÓDIGO OTIMIZADO

O código intermediário (quadruplas) é enviado para uma função que faz uma otimização do código fonte intermediário, sendo feito da seguinte forma, verifica-se as variáveis inicializadas e então seu valor é substituído em cada local onde a variável foi utilizada nas expressões podendo assim otimizar o tempo de execução do código pelo processador escolhido.

As variáveis inicializadas são verificadas e então é substituída em cada local onde a mesma é usada como parâmetro das funções, tendo assim uma otimização visto que os valores são substituídos pelo nome da variável que o representa.

GERAÇÃO DE CÓDIGO NA ARQUITETURA

A geração de código na arquitetura transforma o código otimizado em código que o processador vai entender. Nesse caso foi escolhido a arquitetura MIPS. Onde essa etapa é realizada varrendo a quadrupla otimizada e para cada linha de instrução é gerado o código equivalente na arquitetura do processador escolhido.

ARQUITETURA MIPS

A arquitetura MIPS é do tipo RISC, ou seja, apresenta instruções simples e reduzidas mas que cabe muito bem para este experimento de compilador.

A arquitetura apresenta a seguinte estrutura para instruções do tipo R para operações aritméticas contendo a seguinte sintaxe para as instruções em MIPS:

- **Adição** → add $1, $2, $3
- **Subtração** → sub $1, $2, $3
- **Multiplicação** → mult $1, $2
- **Divisão** → div $1, $2
- **Adição** Imediata → addi $1, $2, valor
- **Carregamento Imediato** → li $1, valor

GERAÇÃO DE CÓDIGO FINAL

A geração de código acontece analisando a tabela de quadruplas com as instruções otimizadas que é processado e transformado para o código equivalente na arquitetura desejada. Dependendo do tipo de instrução faz-se necessário pequenos ajustes e o código final é gerado na arquitetura desejada. É bom ter atenção visto que as instruções diferem quanto ao seu tipo de construção. Então, devem ser analisadas antes de serem geradas no arquivo final como é o caso de adição e multiplicação que por exemplo tem recebem parâmetros diferentes para realizar o cálculo no processador.

CONCLUSÃO

O projeto do compiler expressions ensinou detalhadamente todos os processos de construção de analisadores léxicos, sintáticos, semânticos na prática e deixando bem claro a diferença de cada um deles e o que cada um responsável por analisar. Depois tem a geração de código que transforma o código em parcelas menores de instruções mais primitivas e que depois é otimizada visando maximizar o desempenho do processador reduzindo desperdícios de ciclos clock e por último é gerado o código fonte para a arquitetura especificada já com o código otimizado independente da arquitetura escolhida e para gerar o código para diferentes processadores somente é necessário mudar a parte da geração de código para cada arquitetura de processadores.