Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
DIM0442 - COMPILADORES

Problema 5: Verificação de Tipos e Geração de Código

Aluno:
VALMIRO RIBEIRO DA SILVA

Natal / RN

Introdução

A linguagem de programação LogFácil foi especificada inicialmente com o intuito de ajudar nas disciplinas introdutórias de programação bem como ajudar a introduzir conceitos básicos com relação à lógica desde o primeiro contato dos programadores com uma linguagem de programação. Porém, em virtude de complicações em termos de implementação, alguns recursos da linguagem foram adaptados e/ou removidos.

Esse documento contém:

Com relação à linguagem: as especificações atualizadas da linguagem em termos de seu paradigma de programação, informação sobre o tempo de vinculação dos atributos de variáveis, regras de resolução de escopo e apresentação das estruturas sintáticas acompanhadas de descrições semânticas informais.

Com relação ao compilador: temos uma breve descrição das ferramentas e comandos utilizados para a geração do compilador, principais estruturas de dados utilizadas (incluindo detalhes sobre a tabela de símbolos), os resultados esperados para os programas de teste, grafos de depêndencia do esquema de tradução dirigido pela sintaxe e comandos para a utilização do compilador gerado.

Manual da Linguagem

A LogFácil é uma linguagem de programação imperativa, ou seja, comandos mudam o estado das variáveis. A linguagem possui *tipagem estática*, ou seja, as variáveis são declaradas juntamente com seus respectivos tipos. Por esse motivo, a linguagem vincula as variáveis aos seus respctivos tipos em tempo de compilação

Os tipos de dados presentes na linguagem são: inteiro, ponto flutuante, caractere, booleano,string e vetores (estáticos).

A LogFácil possui expressões para serem avaliadas, como literais, construções, chamadas de função (onde as funções não permitem sobrecarga), expressões aritiméticas e expressões booleanas.

Os comandos serão atribuições, skip, chamada de procedimento(onde os procedimentos também não podem ser sobrecarregados), o comando condicional se-senão, os comandos iterativos enquanto, faca-enquanto. Por questões de facilitar o ensino, foi optou-se por utilizar a semantica de cópia no quesito de atribuição de variáveis.

A linguagem é de *escopo estático*, portanto o corpo de um procedimento/função é executado no ambiente de definição do mesmo. A escolha do escopo estático está ligada ao fato da linguagem possuir tipagem estática, para evitar possíveis erros na hora da compilação.

Abaixo seguem as descrições sintáticas das estruturas acompanhadas das descrições semânticas informais

Declaração de Variáveis e atribuições

A estrutura básica de uma declaração é:
<tipo> <identificador> [= <expressão>]? ;

A estrutura de uma atribuição é:
<identificador> = <expressão>;

exemplo:

```
programa{
      inteiro x;            //declaração de variável
      x = 10;               //atribuição
      inteiro y = 20;       //declaração + atribuição
      escreva(x + y);
}
```

Expressões booleanas

Expressões booleanas sempre retornar um valor booleano (verdadeiro ou falso) e sua estrutura é:

<expressão> <operador booleano> <expressão>

Os operadores booleano presentes na linguagem são a conjunção(&&), disjunção (||), implicação (->), biimplicação(<->), igualdade (==) e negação (!), sendo esse último unário.

Exemplo:

```
programa{
      booleano teste = verdadeiro ou !verdadeiro;
}
```

No exemplo acima, a variável teste recebe o valor ‘verdadeiro’ após ser avaliada

Se - Senão

Comando que avalia uma expressão booleana antes de executar um bloco de comandos.

Sua estrutura é:
se (<expressao>) {<programa>} [senao <programa>]?

Onde <programa> é um bloco de comandos da linguagem.

Exemplo:

```
programa{
      inteiro y;
      leia(y);
      se (y % 2 == 0) entao {
            escreva("y eh multiplo de 2\n"); //parte1
      } senao {
            escreva("y nao eh multiplo de 2\n"); //parte2
      }
}
```

No exemplo acima, se o valor do resto da divisão de y for zero o programa executa a parte1, e caso contrário a parte 2 é executada.

Enquanto

Executa um mesmo bloco de comandos enquanto uma expressão booleana tiver o valor verdadeiro.

enquanto (<expressao>) { <programa> }

Exemplo:

```
programa{
      inteiro x = 10;
      enquanto (x > 0){
            escreva ("x eh maior que zero.\n);
      }
}
```

Faça-Enquanto

Semelhante ao Enquanto, porém, a expressão booleana só é avaliada após a primeira execução do bloco de comandos.

faca { <programa> } enquanto (<expressao>)

Exemplo:

```
programa{
      inteiro x = 10;
      faca{
            escreva ("teste\n");
            x = x -1;
      } enquanto ( x > 10);
}
```

Funções

Funções são abstrações sobre expressões que retornam um valor.

<tipo> <identificador> (<sequência de parametros>) {<programa>}
Onde dentro de <programa> deve conter a indicação de retorno:

retorna <expressão>; | retorna <identificador>;

Exemplo:

```
inteiro fatorial(inteiro n){
      se ( n <= 1 ) entao {
            retorne 1;
      } senao {
            retorne n * fatorial (n - 1);
      }
}
```

Procedimentos

Procedimentos são abstrações sobre comandos, logo, alteram variáveis.

procedimento <identificador> (<sequência de parametros>) {<programa>}

Exemplo:

```
procedimento dobrar (inteiro b){
      b = b*2;
}

programa{
      inteiro c = 10;
      dobrar(C);
      escreva(c);
}
```

Manual do Compilador

Geração do Compilador

Para geração do compilador, foram utilizadas as ferramentas LEX, para a análise léxica, e YACC, para a analise sintática, onde o código que fica no arquivo YACC é o responsável por fazer a análise semântica.

No arquivo LEX, que tem a extenção .l , foram expecificadas as expressões regulares e foram utilizadas funções auxiliares para lidar com os tokens. No arquivo YACC, que tem a extenção . y, foi especificada a gramática da linguagem, onde a ferramenta gera a partir daí um analisador sintático Bottom-Up.

Os comandos para gerar os arquivos necessários para a compilação e execução dos testes segue em anexo com esssse relatório, no arquivo “Makefile”, onde basta descompactar os arquivos do compilador e executar o comando make no terminal, utilizando o Sistema Operacional Ubuntu.

Estruturas de Dados Utilizadas e Tabela de Símbolos:

Foram criadas estruturas para representar os tipos, as variáveis, funções e procedimentos. Optu-se por criar uma Tabela de Símbolos, onde para cada variável existe um ponteiro, e as mesmas ficam salvas em um vetor.

Abaixo seguem exemplos de como serão implementadas as estruturas:

A estrutura ASTNode representa os nós da árvore sintática abstrata, por exemplo:

```
enum ASTNodeID { SOMA, SUBTRACAO, ENQUANTO, LITERAL, … };
struct ASTNode {
        ASTNodeID id;
        ASTNode** filhos;
        int valor;
};
```

A soma 2 + 3 pode, por exemplo, ser construida da seguinte forma:

```
ASTNode dois;
ASTNode tres;
ASTNode soma;

dois.id = LITERAL;
dois.valor = 2;

tres.id = LITERAL;
```

```
tres.valor = 3;

soma.id = SOMA;
soma.filhos[0] = &dois;
soma.filhos[1] = &tres;
```

Uma função é constituida por seus parâmetros, suas variáveis locais e pelo seu corpo que por questões de praticidade, será um ponteiro para um nó da árvore abstrata. A praticidade está no fato de que o código de geração de código receberá como entrada um valor do tipo ASTNode. Dessa forma, evita-se construção de representações intermediárias.

```
struct Funcao {
	Variavel** parametros;
	Variavel** variaveis;
	ASTNode* corpo;
}

struct Variavel {
	char nome[255];
	Tipo* tipo;
};

enum TipoID { INTEIRO, CARACTERE, BOOLEANO, FLUTUANTE, ... };
struct Tipo {
	TipoID id;
	Tipo** filhos;
};
```

Grafos de Dependência

Abaixo temos alguns exemplos dos grafos de dependencia:

expressao

```
: expressao_atribuicao
;
```

expressao_atribuicao

```
: expressao_biimplica
| expressao_unaria operador_atribuicao expressao_atribuicao
```

;

Conflitos:

O compilador não tem nenhum conflito shift/reduce remanescente.

Exemplos:

Segue em anexo um arquivo chamado “programas_teste.zip”, contendo os programas de teste da linguagem, que possuem a extenção .zuno e os arquivos em C com a saída esperada da geração de código.

Talvez alguma coisa mude com relação à implementação do compilador, fazendo com que esse relatório esteja sujeito a erratas futuramente.