Universidade Federal Fluminense
Instituto de Computação
Departamento de Ciência da Computação

Relatório do Trabalho I

Professor: **Luis Antonio Brasil Kowada**

Grupo: **Lucas Silveira Serrano e Matheus Marques Barros**

Niterói - RJ
2024.2

Sumário

1 Introdução

Neste relatório, abordamos o desenvolvimento de um compilador para a linguagem **MiniJava+**. O projeto foi estruturado em etapas progressivas, começando pela implementação do scanner (analisador léxico) e do parser (analisador sintático). Posteriormente, avançamos para a análise semântica e a geração de código MIPS.

A construção do scanner seguiu uma abordagem de análise lexical, responsável por processar o código-fonte, identificar palavras-chave, operadores e delimitadores, e gerar uma tabela de tokens. Já o parser foi desenvolvido com base em uma abordagem top-down, que processa os símbolos da entrada para construir uma árvore sintática a partir da raiz até as folhas. Para garantir eficiência e correção no processo, realizamos ajustes na gramática original, como a eliminação de recursões à esquerda.

A biblioteca `Graphviz` foi integrada ao projeto para estruturar e visualizar a árvore sintática, fornecendo uma representação gráfica que facilita a compreensão das etapas de análise sintática realizadas.

Durante a análise semântica, utilizamos uma tabela de símbolos gerada pelo parser. Essa tabela permitiu verificar a consistência de declarações de variáveis, parâmetros de funções e outras construções, assegurando conformidade com as regras da linguagem `MiniJava+`.

Na etapa de geração de código MIPS, o objetivo foi traduzir as instruções do código-fonte em comandos compatíveis com a arquitetura MIPS. Essa tradução foi baseada no código intermediário que foi gerado, levando em consideração aspectos como alocação de registradores e gerenciamento de memória.

2 Primeira Entrega

Na primeira entrega, foi solicitada a implementação do scanner (analisador léxico) e do parser (analisador sintático).

2.1 Scanner

A primeira etapa do algoritmo consiste em realizar a análise léxica por meio do Scanner. Assim, a partir de um código em **MiniJava+**, são extraídos e categorizados os tokens para a análise sintática que será vista posteriormente.

2.1.1 Abordagem do Scanner

Antes de nos aprofundarmos na implementação do Scanner, é importante contextualizar sua lógica. Basicamente, este programa lê um código em **MiniJava+** como uma variável 'string'. Em seguida, são removidas as linhas de comentários, pois estes não são do interesse de um compilador (não são tokens). O conteúdo 'limpo' do código é examinado uma posição por vez. Quando um token é identificado, é criada uma tupla contendo seu tipo e seu valor. Esta tupla é adicionada à lista que será retornada pelo Scanner. A identificação de comentários e tokens é realizada com o auxílio da biblioteca Regex, a qual será explicada posteriormente.

2.1.2 Entrada e saída

O programa **scanner.py** espera receber como argumento o nome do arquivo contendo o código na linguagem **MiniJava+**. A saída deste programa é uma lista de tuplas contendo o tipo do token e seu valor.

2.1.3 Implementação do Scanner

Visando melhorar tanto a legibilidade quanto a manutenibilidade do nosso algoritmo, organizamos cada programa em classes com funções específicas para cada tarefa. A função de inicialização (**__init__**) relaciona os tipos dos tokens com suas respectivas expressões regulares, permitindo a identificação por meio da biblioteca Regex. A função **get_tokens** recebe o conteúdo do código com os comentários já removidos (retorno da função **remove_comments**) e lista cada token na ordem em que aparece, identificando seu tipo por meio do Regex.

2.1.4 Regex

A biblioteca **re** do Python, também conhecida como Regex (Expressões Regulares), é uma ferramenta poderosa para realizar buscas e manipulações de texto de maneira eficiente. Ela permite a definição de padrões que podem ser usados para encontrar sequências de caracteres em uma string, verificar correspondências e até mesmo substituí-las. A função **re.match** é uma das principais ferramentas utilizadas neste programa para a identificação de tokens. Ela tenta casar um padrão no início de uma string ou em uma parte específica dela. Caso o padrão seja encontrado, **re.match** retorna um objeto de correspondência contendo as informações sobre o trecho da string que corresponde ao padrão. Caso contrário, retorna `None`.

Na função **get_tokens**, o comando **re.match** é utilizado para verificar se um token correspondente pode ser extraído a partir da posição atual da string de entrada.

2.2 Parser

Após a implementação do scanner, foi gerada uma tabela de tokens durante a análise léxica, a qual será consumida pelo parser na análise sintática para a construção de uma árvore sintática.

2.2.1 Implementação do Parser

Para uma melhor organização, assim como no caso do scanner, foi desenvolvida a classe `MiniJavaParser`, responsável por consumir cada token da tabela de tokens e verificar sua correção. Além disso, caso todos os tokens sejam validados corretamente, a classe utiliza a biblioteca `Graphviz` tanto para a estruturação da árvore quanto para a geração de um arquivo `tree.png`, que contém uma representação gráfica da árvore sintática.

Primeiramente, foram implementadas funções na classe para lidar com diferentes aspectos da leitura e manipulação da tabela de tokens, incluindo:

- Retornar o token atual sem consumir.
- Consumir o token atual e avançar para o próximo.
- Verificar se o token atual corresponde à categoria e valor.

- Garantir que o token atual é o esperado; caso contrário, gera um erro.

Para lidar com as regras da `EBNF`, foi criada uma função correspondente para cada símbolo não terminal presente na gramática. Vale ressaltar que a `EBNF` original precisou ser ajustada em algumas regras para eliminar recursões à esquerda. Por exemplo, a regra para expressões lógicas foi modificada da seguinte forma:

$$\text{Regra original: EXP} \rightarrow \text{EXP \&\& REXP} \mid \text{REXP}$$

$$\text{Regra atual: EXP} \rightarrow \text{REXP EXP\_AUX}$$

$$\text{Regra adicional: EXP\_AUX} \rightarrow \text{\&\& REXP EXP\_AUX} \mid \varepsilon$$

2.2.2 Exemplo de saída

Agora, para exemplificar a saída, vamos gerar a árvore sintática correspondente ao seguinte código escrito em MiniJava+:

Código em MiniJava+

```
class Factorial{
    public static void main(String[] a){
        n = 1 + 2 * 3;
    }
}
```

Árvore Sintática

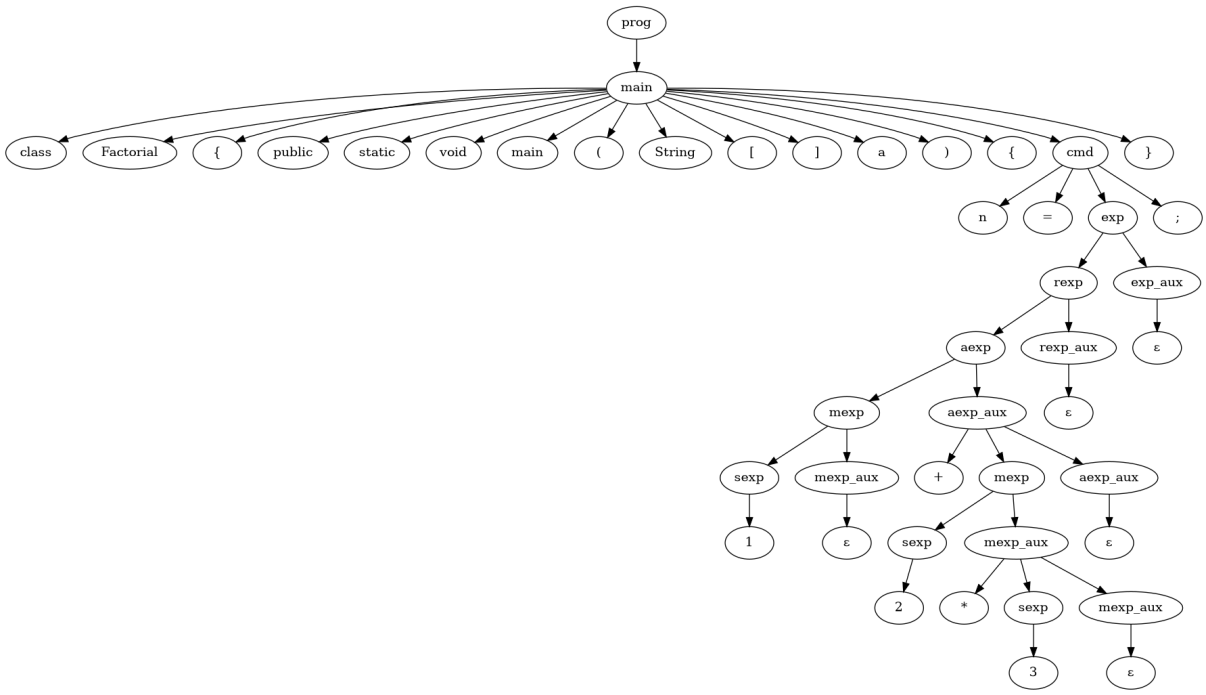

Figura 1: Árvore sintática gerada para o código em MiniJava+.

3 Segunda Entrega

Na segunda entrega do trabalho, foi solicitado que o compilador realizasse tanto a análise semântica quanto a geração de código MIPS. Essas duas etapas estão intimamente relacionadas, pois a análise semântica garante que o código fonte seja semanticamente correto, o que é essencial para a geração de código eficiente e funcional.

3.1 Análise Semântica

A análise semântica é responsável por assegurar que o código segue as regras semânticas da linguagem, indo além da estrutura gramatical verificada pela análise sintática. Nessa etapa, o compilador avalia o **contexto** em que as construções do código estão inseridas. São verificadas, por exemplo, se todas as variáveis foram declaradas antes de serem utilizadas, se chamadas de funções fornecem o número e os tipos corretos de parâmetros e entre outros.

Por exemplo, ao processar uma atribuição como `x = y + 2`, o analisador semântico verifica:

- Se `x` e `y` foram previamente declarados.
- Se os tipos de `x` e de `y + 2` são compatíveis para a operação de atribuição.

De forma semelhante, ao encontrar uma chamada de função como `foo(a, b)`, o compilador verifica:

- Se a função `foo` foi declarada.
- Se o número de argumentos fornecidos está correto.
- Se os tipos de `a` e `b` são compatíveis com os parâmetros esperados por `foo`.

Para realizar essas verificações, o compilador utiliza estruturas como a **tabela de símbolos**, que registra informações sobre variáveis, funções, classes e outros identificadores, incluindo seus tipos, escopos, etc.

Somente após garantir que o código está semanticamente correto, o compilador prossegue para a etapa de geração de código MIPS, onde transforma o código fonte em instruções adequadas para a arquitetura-alvo.

3.2 Tabela de Símbolos

Para realizar a análise semântica no trabalho, foi utilizada a tabela de símbolos, que, como mencionado anteriormente, registra informações sobre identificadores, como classes, variáveis, métodos e parâmetros.

Nossa tabela de símbolos ficou estruturada da seguinte maneira:

Escopo	Símbolo	Tipo	Parâmetros/Info
1	`Fac1`	`class`	
1	`num_aux1`	`int`	
2	`num`	`int`	`param`
2	`ComputeFac`	`method int`	`int`

Tabela 1: Tabela de símbolos gerada pelo compilador.

A tabela de símbolos apresentada fornece informações detalhadas sobre o **escopo** do identificador, seu **nome**, o **tipo** associado e, no caso de parâmetros, se o identificador corresponde a um **parâmetro** de uma função. Para funções, a coluna *Parâmetros/Info* exibe tanto a quantidade quanto os tipos dos parâmetros requisitados.

Por exemplo, considere a função `foo(int a, bool b)`. Na coluna *Parâmetros/Info*, seriam exibidos `int, bool`, indicando que o método `foo` requer dois parâmetros: o primeiro do tipo `int` e o segundo do tipo `bool`.

Essas informações são fundamentais para realizar análises semânticas, como verificar se uma variável foi declarada antes de ser utilizada e validar corretamente os parâmetros fornecidos nas chamadas de funções.

A geração da tabela de símbolos e a análise semântica ocorrem durante a análise sintática, enquanto a árvore sintática é criada (Etapa 1). Caso o compilador percorra e gere a árvore sem encontrar erros, o processo avança para a próxima etapa: a geração do código MIPS. Se houver erros semânticos, eles são registrados e retornados ao fim da execução da análise, permitindo que o desenvolvedor os corrija antes de prosseguir.

Abaixo segue um código em MiniJava+ com erros e a saída gerara pelo compilador:

Código em MiniJava+ com erros semânticos

```
class Factorial{
    public static void main(String[] a){
        System.out.println(new Fac2().ComputeFac(10)); //Fac2 não
            existe
    }
}
class Fac {
    int num_aux1;
    public int ComputeFac(int num){
        int num_aux1;
        if (num < 1)
            num_aux2 = 2 + 8; //Variável num_aux2 não foi
               declarada
        else
            num_aux1 = num * (this.ComputeFac(num-1, 7)); //2 par
               âmetros são passados na função
        return num_aux1;
    }

    public int ComputeFac(int num){ //ComputeFac declarado
       anteriormente
        num_aux1 = num * (this.ComputeFac(true)); //Parâmetro com
            tipo errado
        return num_aux1;
    }
}
```

Saída do Compilador

```
5 erro(s) encontrado(s):
Erro 1: Símbolo 'num_aux2' não encontrado.
Erro 2: Erro na chamada da função 'ComputeFac': esperado 1
   parâmetros, mas 2 foram fornecidos.
Erro 3: Método 'ComputeFac' já declarado no escopo atual.
Erro 4: Erro no parâmetro 1 durante a chamada da função '
   ComputeFac': esperado tipo 'int', mas 'boolean' foi
   fornecido.
Erro 5: Símbolo 'Fac2' não encontrado.
```

Durante o processo de compilação, foram identificados e relatados quatro erros semânticos:

- **Erro 1: Símbolo 'num_aux2' não encontrado.** Esse erro ocorre porque a variável `num_aux2` foi utilizada sem ser declarada previamente no escopo atual. Em linguagens como MiniJava+, é obrigatório declarar todas as variáveis antes de usá-las.

- **Erro 2: Erro na chamada da função 'ComputeFac': esperado 1 parâmetro, mas 2 foram fornecidos.** O compilador identificou uma chamada ao método `ComputeFac` com um número incorreto de argumentos. O método foi declarado para aceitar apenas um parâmetro do tipo especificado, mas a chamada forneceu dois, o que viola a assinatura da função.

- **Erro 3: Método 'ComputeFac' já declarado no escopo atual.** Esse erro indica que o método `ComputeFac` foi declarado mais de uma vez no mesmo escopo. Em linguagens baseadas em classes, métodos devem ter assinaturas únicas dentro de um mesmo escopo para evitar ambiguidade.

- **Erro 4: Erro no parâmetro 1 durante a chamada da função 'ComputeFac': esperado tipo int, mas boolean foi fornecido.** Esse erro ocorre porque o tipo do primeiro parâmetro fornecido na chamada do método `ComputeFac` não corresponde ao tipo esperado declarado na assinatura da função. O compilador espera um argumento do tipo `int`, mas foi recebido um `boolean`, causando uma incompatibilidade de tipo.

- **Erro 5: Símbolo 'Fac2' não encontrado.** O compilador tentou acessar o símbolo `Fac2`, mas não conseguiu localizá-lo em nenhum escopo válido. Isso pode ser resultado de um erro de digitação, uma referência a um símbolo não declarado ou a tentativa de usar um símbolo fora de seu escopo de visibilidade.

Fluxo após os erros: Esses erros impedem que o processo de compilação avance para as etapas subsequentes, como a geração de código MIPS. Após o relatório detalhado dos problemas, o desenvolvedor pode analisar o código-fonte, corrigir os erros identificados e reiniciar o processo de compilação.

Esse ciclo de feedback é essencial no desenvolvimento, pois permite que o compilador funcione como uma ferramenta de apoio ao programador, garantindo que o código produzido seja consistente com as regras da linguagem e esteja livre de falhas semânticas.

3.3 Árvore Sintática Abstrata

Finalizada a **análise semântica**, o próximo passo do compilador seria gerar o código *MIPS*. A forma mais fácil e eficaz de gerar esse código é por meio de um **código intermediário**, o qual pode ser extraído de uma **árvore sintática abstrata** (*AST*). Essa nova estrutura pode ser obtida a partir da árvore criada na análise sintática.

Para obter a nova árvore, devem-se "podar" nós indesejados. Esses são, praticamente, quaisquer nós que representem símbolos não-terminais. Deste modo, a árvore sintática abstrata pode ser organizada de forma que operadores sejam nós intermediários e operandos sejam folhas.

O programa 1 demonstra como calcular o fatorial de um número (10) na linguagem *MiniJava+*. A árvore sintática concreta para este programa possui 236 nós (intermediários e folhas), o que impossibilita exibi-la com clareza neste documento. Por outro lado, a árvore abstrata - vista na imagem 2 - requer somente 26 nós, o que facilita tanto a leitura humana quanto a extração da informação necessária para a construção do código intermediário.

Listing 1: Programa em MiniJava+ para calcular o fatorial de um número.

```
class Factorial {
    public static void main(String[] a) {
        System.out.println(new Fac().ComputeFac(10));
    }
}

class Fac {
    public int ComputeFac(int num) {
        int num_aux;
        if (num < 1) {
            num_aux = 1;
        } else {
            num_aux = num * (this.ComputeFac(num - 1));
        }
        return num_aux;
    }
}
```

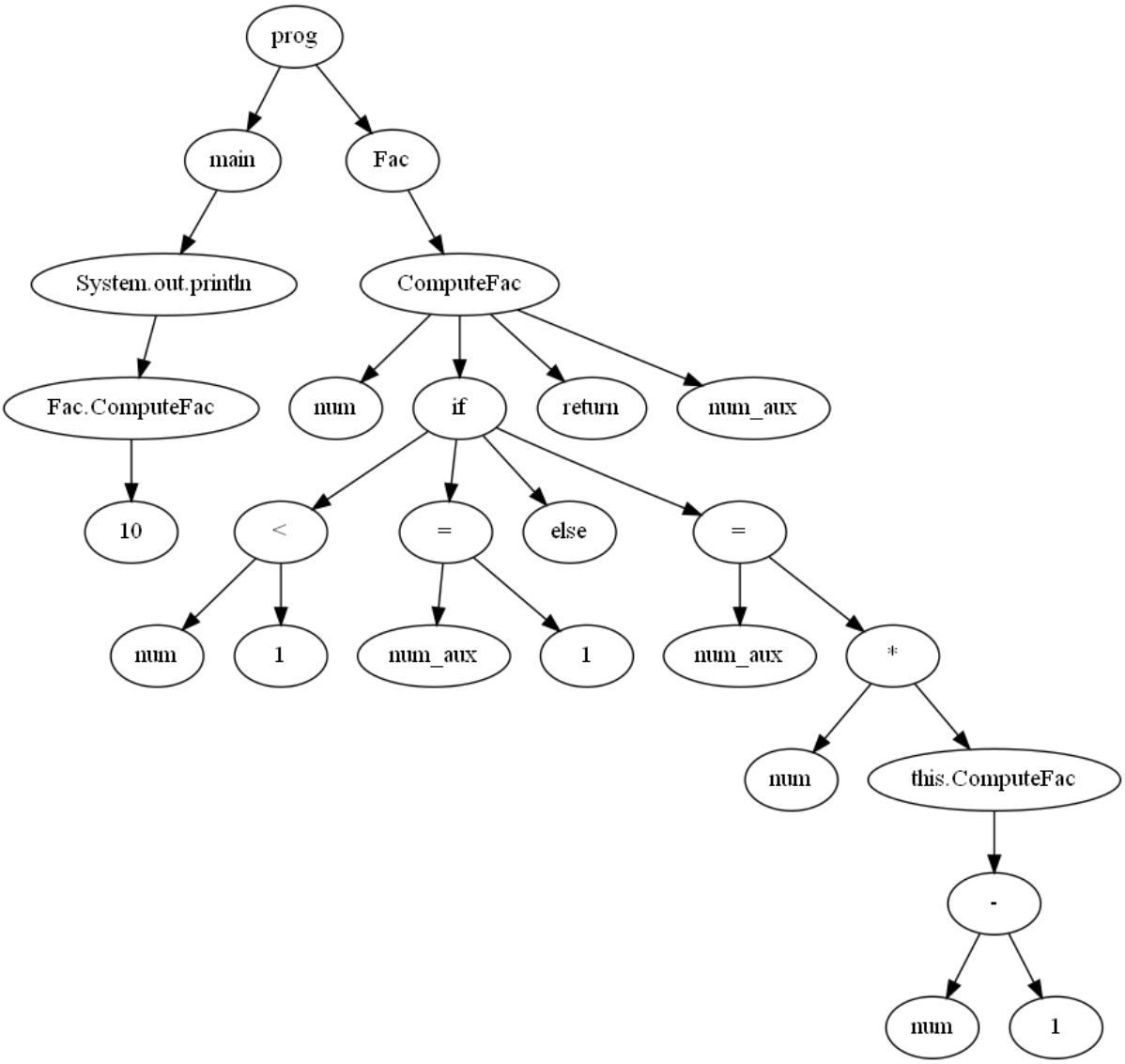

Figura 2: Árvore sintática abstrata gerada para o código em MiniJava+.

3.3.1 Desenvolvimento do Programa - *AST*

A confecção da árvore abstrata foi realizada pelo programa *ast.py*, o qual utilizou três bibliotecas: *re*, *Digraph* e *defaultdict*.

A primeira permitiu criar expressões regulares para localizar os nomes (*labels*) dos nós e suas arestas de acordo com a formatação do código fonte da árvore original. A segunda pertence ao módulo *graphviz* e foi utilizada para montar a árvore conforme os nós eram extraídos. A terceira, que faz parte do módulo *collections*, facilitou a organização da hierarquia existente entre os nós.

De modo geral, a árvore original passou por uma sequência de funções, cada uma lidando com um nó específico (ou conjunto de nós). O retorno de uma função era a árvore passada como entrada mais enxuta. Ao final desse processo, somente os nós de interesse estavam presentes.

3.4 Código Intermediário

O **código intermediário**, comumente representado em forma de *código de três endereços*, é uma representação abstrata usada para a geração de código de máquina (*MIPS*) a partir de uma **árvore sintática abstrata (AST)**. Ele é composto por instruções simples, cada uma manipulando no máximo três operandos.

3.4.1 Desenvolvimento do Programa - Código Intermediário

A tradução da *AST* para o **código intermediário** não foi uma tarefa simples. O principal motivo é pelo formato como a biblioteca *Digraph* salva esse objeto. Diferente de uma estrutura de árvore convencional, não há uma forma trivial de realizar uma busca em largura ou em profundidade. Foi necessário estudar a organização do código fonte da *AST* para poder extrair suas instruções na ordem correta.

Passada esta curva de aprendizagem, foram criadas funções específicas para cada tipo de nó que pudesse ser encontrado, além de funções auxiliares para tratar seus filhos. Nessa parte, a lógica se assemelha à aplicada no programa de confecção da *AST*. O resultado final do programa foi um arquivo de texto contendo uma instrução por linha.

3.5 Otimização

Antes de passar para a etapa de geração do código em linguagem de máquina (MIPS), buscou-se otimizar o **código intermediário** obtido até então.

Isso foi feito por meio do programa *otimizer.py*, o qual realiza quatro tipos de otimização local no código gerado pelo processo anterior.

A primeira é a otimização algébrica, a qual busca reescrever determinadas expressões de forma a torná-las mais simples do ponto de vista computacional. A segunda é referente à atribuição simples, na qual cada registrador aparece somente uma vez do lado esquerdo de uma atribuição. A terceira é a propagação de cópia, ou seja, o valor de um registrador assume seu lugar nas expressões seguintes. Por fim, há o desdobramento de constantes, resolvendo expressões que possuem operandos com valores explícitos (não em registradores).

Como as otimizações são locais, elas estão restritas aos blocos básicos. Por um lado, isto facilita a implementação. Por outro, impede certas possibilidades de melhorias. Se a implementação global fosse desenvolvida, otimizações como "remoção de código morto" seriam mais eficientes. Contudo, até compiladores profissionais podem apresentar dificuldade ao realizar essa prática. Portanto, foi optado por manter somente as mencionadas anteriormente.

O programa 2 foi escrito na linguagem *MiniJava+* para realizar diversas operações aritméticas. Apesar de sua funcionalidade não fazer muito sentido do ponto de vista da programação, ele serve como um bom exemplo didático. O código intermediário 3 é uma versão equivalente a esse programa. O código 4 é o resultado das otimizações locais. Nota-se que houve uma diminuição de 25% das linhas de instruções. Além disso, observa-se que o *if* pôde ser simplificado para um desvio não condicional.

Vale ressaltar que a presença de código morto - atribuições não utilizadas ou instruções não alcançáveis - se dá devido à escolha de não implementar uma otimização global.

Listing 2: Programa em MiniJava+ usado como exemplo didático.

```
class Otimizacao{
    public static void main(String[] a){
        System.out.println(new Exemplo1().Contas(2));
    }
}
class Exemplo1 {
    int num_aux1;
    int num_aux2;
    public int Contas(int num){
        num_aux1 = 5;
        num_aux2 = 3 * num_aux1;
        num_aux1 = num * 2;
        if (num_aux2 < 100) {
            num_aux1 = 22;
            num_aux1 = num_aux1 * 8;
        }
        else {
            num_aux1 = 5;
            num_aux1 = num_aux1 - 1;
        }
        return num_aux2;
    }
}
```

Listing 3: Código intermediário antes das otimizações.

```
main:
param 2
t1 := call Exemplo1.Contas, 1
print t1

Exemplo1.Contas(num):
num_aux1 := 5
t2 := 3 * num_aux1
num_aux2 := t2
t3 := num * 2
num_aux1 := t3
t4 := num_aux2 < 100
ifFalse t4 goto else_1
true_1:
num_aux1 := 22
t5 := num_aux1 * 8
num_aux1 := t5
goto end_else_1
else_1:
num_aux1 := 5
t6 := num_aux1 - 1
num_aux1 := t6
end_else_1:
return num_aux2
```

Listing 4: Código intermediário após as otimizações.

```
main:
param 2
t1 := call Exemplo1.Contas, 1
print t1

Exemplo1.Contas(num):
new_num_aux1 := 5
num_aux2 := 15
t3 := num << 1
num_aux1 := t3
new_num_aux1 := 22
num_aux1 := 176
goto end_else_1
else_1:
new_num_aux1 := 5
num_aux1 := 4
end_else_1:
return num_aux2
```

3.5.1 Desenvolvimento do Programa - Otimização

A implementação do programa *otimizer.py* foi relativamente simples. Uma função foi criada para cada tipo de otimização, e o **código intermediário** foi quebrado em blocos básicos. Deste modo, o bloco passava por uma função de cada vez.

Como a realização de uma otimização pode possibilitar outras, foi projetado um *loop* que garantia a execução sequencial das funções três vezes seguidas. Esse número provou-se suficiente para otimizar ao máximo os códigos usados como teste.

3.6 Código MIPS

A partir da versão otimizada do código, sua tradução para a linguagem de máquina foi realizada sem dificuldades. Foi necessário percorrer uma linha de cada vez, identificando qual era a ação executada por ela (operação, atribuição, desvio condicional, etc). Assim, era chamada uma função específica para realizar a tradução.

A única biblioteca utilizada para o programa *mips.py* foi a *re*, a qual esteve presente em quase todas as etapas deste trabalho. Seu uso, desta vez, foi simples: extrair quais eram os parâmetros de um método. Isso permitiu que o programa identificasse se uma variável estava armazenada na pilha do *frame pointer* (*$fp*).