Compiladores - Trabalho Prático 2

Vitor Menezello 2013007854

1 Introdução

O objetivo do trabalho é construir um **front-end** para uma linguagem **Small-L**, definida a seguir. O código foi implementado em Java, baseado no código do livro texto.

2 Desenvolvimento

2.1 Linguagem Fonte

A linguagem fonte, descrita por uma gramática que é reconhecida em uma análise descendente:

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{block} \\
\textit{block} &\rightarrow \{\ \textit{decls}\ \textit{stmts}\ \} \\
\textit{decls} &\rightarrow \textit{decls}\ \textit{decl} \mid \epsilon \\
\textit{decl} &\rightarrow \textit{type}\ \textbf{id}\ \textbf{;} \\
\textit{type} &\rightarrow \textit{type}\ [\ \textbf{num}\ ] \mid \textbf{basic} \\
\textit{stmts} &\rightarrow \textit{stmts}\ \textit{stmt} \mid \epsilon \\
\textit{stmt} &\rightarrow \textit{loc} = \textit{bool}\ \textbf{;} \\
&\mid \textbf{if}\ (\ \textit{bool}\ )\ \textit{stmt} \\
&\mid \textbf{if}\ (\ \textit{bool}\ )\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
&\mid \textbf{while}\ (\ \textit{bool}\ )\ \textit{stmt} \\
&\mid \textbf{do}\ \textit{stmt}\ \textbf{while}\ (\ \textit{bool}\ )\ \textbf{;} \\
&\mid \textbf{break}\ \textbf{;} \\
&\mid \textit{block} \\
\textit{loc} &\rightarrow \textit{loc}\ [\ \textit{bool}\ ] \mid \textbf{id} \\
\\
\textit{bool} &\rightarrow \textit{bool} \mid\mid \textit{join} \mid \textit{join} \\
\textit{join} &\rightarrow \textit{join}\ \&\&\ \textit{equality} \mid \textit{equality} \\
\textit{equality} &\rightarrow \textit{equality} == \textit{rel} \mid \textit{equality}\ \textbf{!=}\ \textit{rel} \mid \textit{rel} \\
\textit{rel} &\rightarrow \textit{expr} < \textit{expr} \\
&\mid \textit{expr} <= \textit{expr} \\
&\mid \textit{expr} > \textit{expr} \\
&\mid \textit{expr} >= \textit{expr} \\
&\mid \textit{expr} \\
\textit{expr} &\rightarrow \textit{expr} + \textit{term} \mid \textit{expr} - \textit{term} \mid \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{unary} \mid \textit{expr} / \textit{unary} \mid \textit{unary} \\
\textit{unary} &\rightarrow \textbf{!}\ \textit{unary} \mid - \textit{unary} \mid \textit{factor} \\
\textit{factor} &\rightarrow (\ \textit{bool}\ ) \mid \textit{loc} \mid \textbf{num} \mid \textbf{real} \mid \textbf{true} \mid \textbf{false}
\end{aligned}
$$

2.2 Código

O código implementado pode ser encontrado no repositório do GitHub:
https://github.com/VitorMenezello/front_end_small_L.

O **front-end** construído consiste de um programa principal que recebe como entrada um programa fonte escrito na linguagem definida. Os analisadores léxico e sintático recebem esse código como um fluxo de tokens e o lê caractere a caractere, produzindo uma árvore sintática com nós, implementados como objetos. Mais sobre como os objetos são definidos na próxima seção.

Utilizando nomenclatura e funcionalidades equivalentes para os pacotes, o código criado possui cinco deles, chamados: **Main**, **Lexer**, **Symbols**, **Parser** e **Inter**. O programa principal, que está no pacote **Main**, é executado com um código fonte de argumento de execução, e cria então uma instância do analisador léxico e sintático (**Lexer**) que gera a árvore de sintaxe atravéz do **Parser**.

2.2.1 Lexer

O pacote **Lexer** é uma extensão do código do livro texto. Na classe **Tag** são definidas constantes para os tokens. Há também a classe **Token** e as classes que a extendem, como **Num**, **Real** e **Word**. A classe principal do pacote, a classe **Lexer**, possui métodos de leitura de caracteres para retornar os objetos (tokens) corretos de acordo com a sua definição.

github.com/VitorMenezello/front_end_small_L/tree/master/src/Lexer.

2.2.2 Parser

Este pacote possui apenas uma classe, chamada **Parser**, que trata da leitura e do parsing dos tokens lidos através de cada derivação da gramática. Na classe são definidos métodos para todos os tipos de expressões e declarações, e o tratamento das exceções geradas caso existam erros de sintáxe.

github.com/VitorMenezello/front_end_small_L/tree/master/src/Parser.

2.2.3 Symbols

O pacote responsável pela tabela de símbolos é o **Symbols**. A classe **Env** possui a tabela hash que mantém todos os Ids encontrados no código durante a leitura. Na classe **Type** definimos os tipos básicos e na classe **Array** são definidos os arranjos.

github.com/VitorMenezello/front_end_small_L/tree/master/src/Symbols.

2.2.4 Inter

O pacote **Inter** contém a hierarquia da classe **Node**, os nós da árvore de sintáxe. Suas duas subclasses são **Expr** para nós de expressão e **Stmt** para nós de comando.

As subclasses de **Expr** são **Id**, **Temp**, **Constant**, **Op** e **Logical**. Essas duas últimas se extendem em inúmeras outras, como as classes de operações de acesso, operações aritméticas e operadores unários, que extendem **Op**, e as classes de comparação e lógica, que extendem **Logical**.

As subclasses de **Stmt** são todos os controladores de fluxo. Classes como **If**, **Else** e **While** são alguns exemplos, além de atribuições de valores para variáveis.

github.com/VitorMenezello/front_end_small_L/tree/master/src/Inter.

3 Testes

O primeiro teste foi realizado com o código a seguir, fornecido na especificação do trabalho:

```
{
        int i; int j; float v; float x; float[100] a;
        while(true) {
                do i = i+1;
                while (a[i] < v);
                do j = j-1;
                while(a[j] > v);
                if(i >= j) break;
                x = a[i]; a[i] = a[j]; a[j] = x;
        }
}
```

Para esse teste, o **front-end** produz o seguinte resultado. Resultado este igual ao esperado, como mostra a especificação.

```
L1:L3:  i = i + 1
L5:     t1 = i * 8
        t2 = a [ t1 ]
        if t2 < v goto L3
L4:     j = j - 1
L7:     t3 = j * 8
        t4 = a [ t3 ]
        if t4 > v goto L4
L6:     iffalse i >= j goto L8
L9:     goto L2
L8:     t5 = i * 8
        x = a [ t5 ]
L10:    t6 = i * 8
        t7 = j * 8
        t8 = a [ t7 ]
        a [ t6 ] = t8
L11:    t9 = j * 8
        a [ t9 ] = x
        goto L1
L2:
```

O segundo teste foi realizado com um código feito pelos alunos, com comandos simples mas escopos um pouco diferentes do que o código fornecido:

```
{
        int a;
        a = 2;
        if (a < 3)
                a = 3;
        while (a < 10) {
                int b;
                b = a;
                do {
                        b = b - 1;
                } while(b > 0);
                a = a + 1;
        }
}
```

E esse teste produz o seguinte resultado:

```
L1:     a = 2
L3:     iffalse a < 3 goto L4
L5:     a = 3
L4:     iffalse a < 10 goto L2
L6:     b = a
L7:     b = b - 1
L9:     if b > 0 goto L7
L8:     a = a + 1
        goto L4
L2:
```

4 Conclusão

A geração de código intermediário é um processo que consiste de várias etapas, na qual todos os analisadores verificam pequenas partes do código até gerar o resultado. O código intermediário gerado está em uma linguagem de mais baixo nível para que a próxima fase da compilação realize suas operações como alocação de memória para as variáveis encontradas e registradas na tabela de símbolos. Mas, de forma geral, o **front-end** gerado realiza as operações requisitadas na especificação do trabalho.