Trabalho Prático II

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Compiladores I

João Francisco B. S. Martins, Pedro D. V. Chaves
{joaofbsm, pedrodallav}@dcc.ufmg.br

30 de Outubro de 2017

1 Introdução

Um compilador é composto por duas partes principais: o *front end* e o *back end*. O *front end* analisa o código fonte a fim de construir uma representação interna do programa, chamada de representação intermediária. Para tal ele se utiliza de uma estrutura de dados chamada tabela de símbolos, a qual será passada adiante junto com a representação gerada. Já o *back end* trata da construção do programa objeto a partir da representação intermediária e da tabela de símbolos, realizando otimizações no código, quando possível.

O trabalho em questão tem como objetivo implementar o *front end* de um compilador, cujas fases estão destacadas na Figura 1, para a linguagem **SmallL**. Essa linguagem é descrita pela gramática apresentada na Figura 2. Para codificação dos componentes do compilador se utilizou a linguagem de programação **Java** (*v.8*).

O trabalho foi desenvolvido utilizando a ferramenta de versionamento Git juntamente com a plataforma de desenvolvimento remoto GitHub. O repositório do projeto contém não só o código fonte, mas também os scripts auxiliares desenvolvidos e os arquivos de teste, podendo ser acessado no endereço `https://github.com/joaofbsm/smallL`.

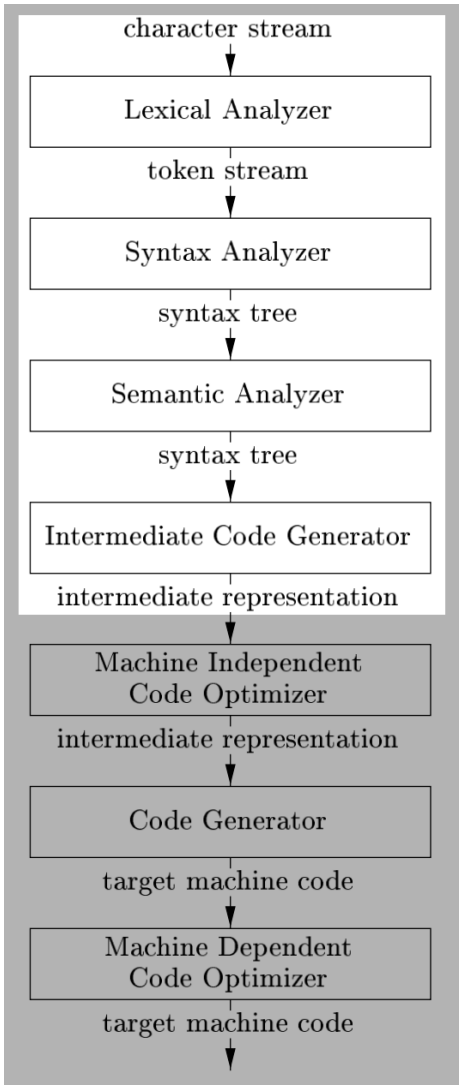

Figura 1: Componentes do *front end* de um compilador (destacados em branco)

.

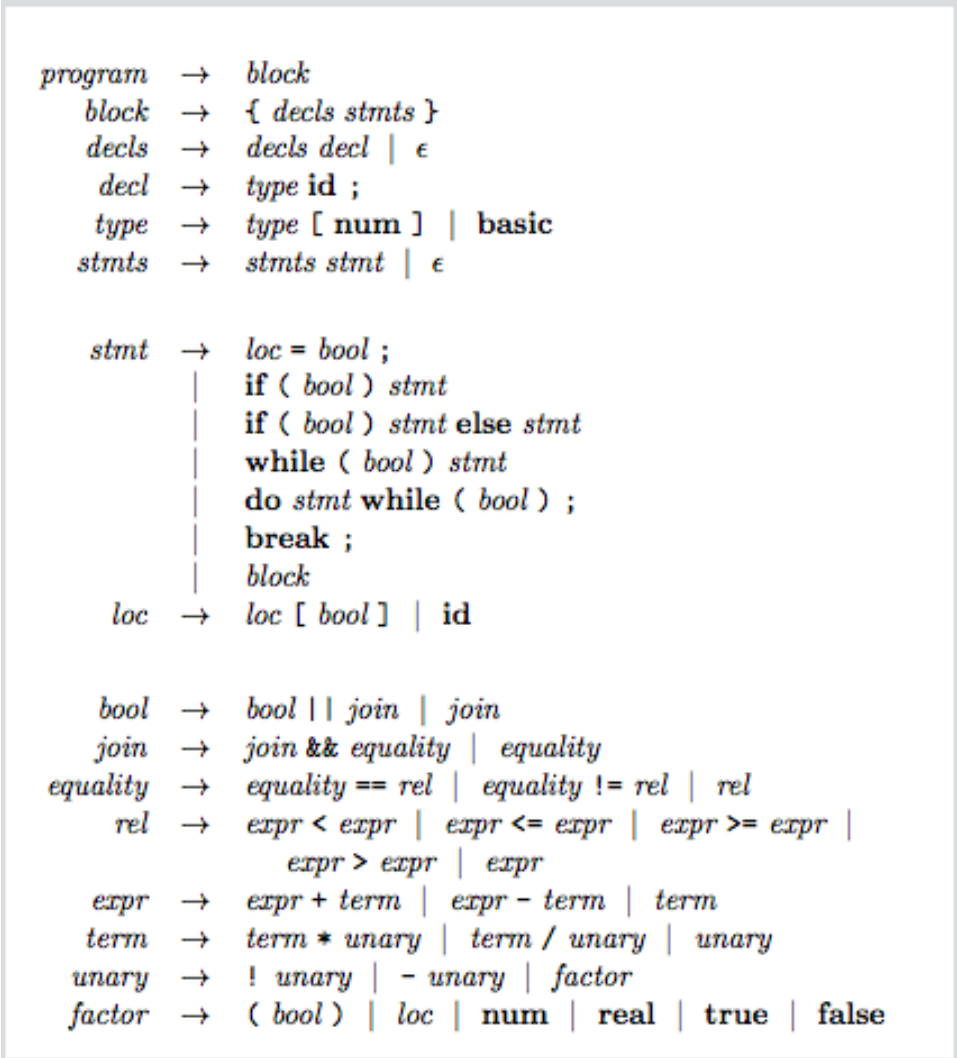

Figura 2: Gramática inicial que descreve a linguagem **SmallL**

.

2 Desenvolvimento

A implementação do *front end* teve como base o código Java disponibilizado no apêndice A do livro-texto [1]. A seguir serão discutidas as estruturas de dados utilizadas em cada uma das partes da implementação do *front end*, bem como as suas relações com o funcionamento do programa.

2.1 Analisador Léxico

A análise léxica do *front end* é implementada através do pacote `lexer`. O funcionamento geral é baseado na identificação de **tokens** a partir da entrada. Tais tokens podem ser **constantes**, **palavras-chave** (reservadas) ou **identificadores**. Uma vez identificados, os tokens são passados para o **parser**, a fim de criar a tabela de símbolos.

- Classe **Tag**: é responsável pela definição de constantes para os tokens.
- Classe **Word**: é responsável por gerenciar lexemas para identificadores, palavras-chave e *tokens* compostos.
- Classe **Real**: é responsável pelos números de ponto flutuante.
- Classe **Num**: é responsável pelos números inteiros.
- Classe **Token**: é responsável pelas decisões de parse (lexemas ou valores), como pode ser visualizado na figura abaixo.
- Classe **Lexer**: implementa função *scan()* para receber os *tokens*.

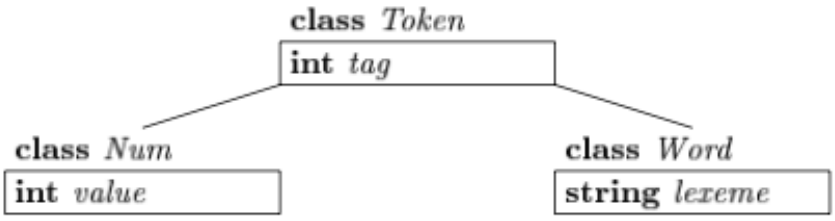

Figura 3: Decisões de parse baseadas na `tag` da classe **Token** (lexemas ou valores)

.

2.2 Tabela de Símbolos

A tabela de símbolos do *front end* é responsável por guardar informações sobre as construções do programa fonte. As entradas da tabela contém informação sobre identificadores (lexema, tipo e posição de armazenamento).

A tabela de símbolos deve ser capaz de guardar múltiplas declarações do mesmo identificador, cada uma em um escopo diferente. A implementação feita utiliza a noção de escopos para garantir essa premissa.

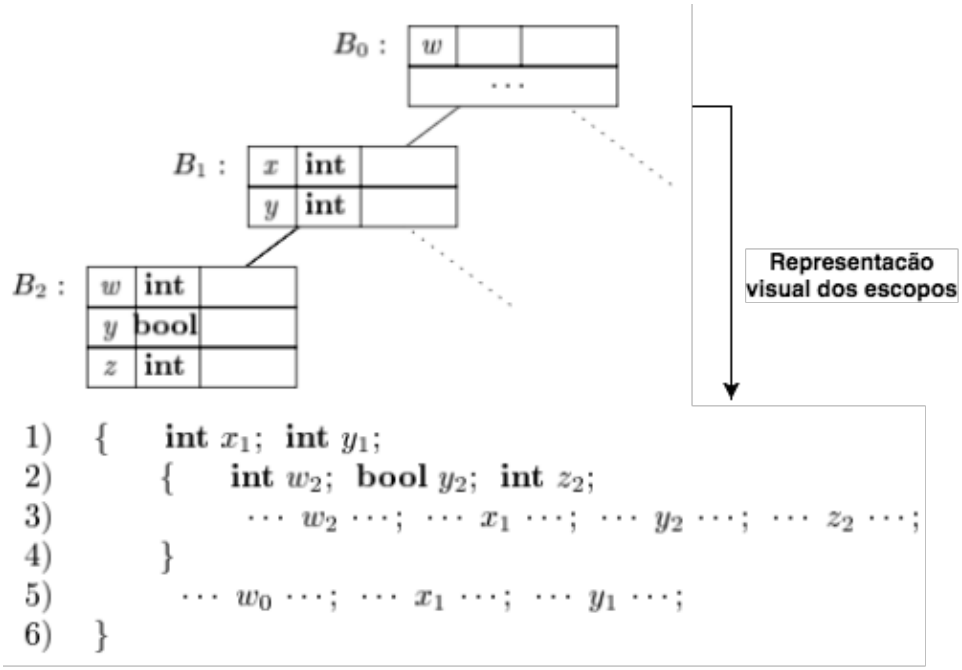

Figura 4: Exemplo de blocos de códigos e seus respectivos escopos na tabela de símbolos

.

A tabela de símbolos é na verdade uma "cadeia"de tabela de símbolos, cada uma representando seu respectivo escopo.

A noção de escopos foi implementada através da classe **Env** presente no pacote `symbols`. A estrutura de dados utilizada para cada tabela de símbolos foi uma tabela **hash**. Quando encadeadas, as tabelas de símbolo formam uma estrutura de árvore. A classe **Env** contém três operações básicas (funções/construtores):

- **construtor Env()**: cria nova tabela de símbolos através de uma *Hash-table*. A criação é baseada no escopo corrente, ou seja, se já foi criado algum escopo anterior, o próximo escopo é criado e 'linkado' com o anterior através da variável `prev`, do tipo `Env`.

- **função put()**: coloca nova entrada na tabela corrente baseada em uma **chave** (entrada da classe **Token**) e um **valor** (entrada da classe **Id**, do pacote `iter`)

- **função get()**: recuperar uma entrada para um identificador, procurando na cadeia de tabelas de símbolos.

A seguir é apresentado o trecho de código referente à classe **Env**:

Algoritmo 1: Classe Env.java

```
public class Env {
    private Hashtable table;
    protected Env prev;

    public Env(Env n) { table = new Hashtable(); prev = n; }

    public void put(Token w, Id i) { table.put(w, i); }

    public Id get(Token w) {
        for( Env e = this; e != null; e = e.prev ) {
            Id found = (Id)(e.table.get(w));
            if( found != null ) return found;
        }
        return null;
    }
}
```

2.3 Analisador Sintático

A implementação do analisador léxico se faz presente no pacote `parser`, num arquivo de mesmo nome. A gramática original da linguagem **SmallL** precisava de adaptações para ser reconhecida por análise descendente(*top-down*). Portanto, os procedimentos implementados na classe **Parser** são baseados na gramática resultante após a remoção da recursão à esquerda na gramática original.

Se utilizando do fluxo de entrada de tokens, o analisador sintático constrói a árvore de sintaxe com o apoio das funções construtoras do pacote `inter` e da tabela de símbolos. Um exemplo ilustrativo de árvore de sintaxe pode ser visto na Figura 5, não necessariamente refletindo a gramática da linguagem tratada neste trabalho.

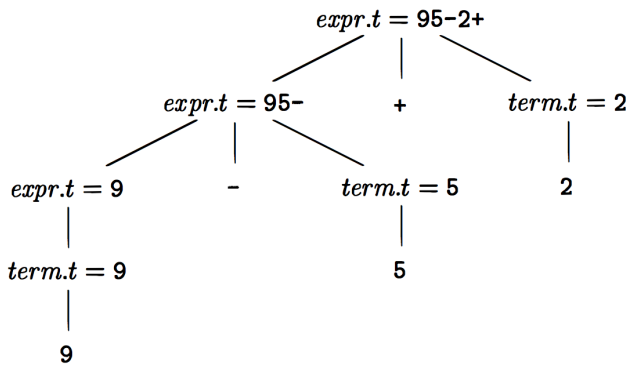

Figura 5: Valores de atributos nos nós de uma árvore de sintaxe.

.

2.4 Geração de Código Intermediário

As classes indispensáveis para geração do código intermediário no nosso *front end* se encontram no pacote `inter`. Neste pacote implementamos a hierarquia da classe **Node**.

Os dois descendentes diretos de **Node** são:

- **classe Expr**: Responsável por nós de expressões. Alguns de seus métodos, juntamente com suas subclasses, geram códigos de desvio para expressões booleanas, tendo o método **jumping** como essencial nessa tarefa, pois é ele quem vai declarar o desvio propriamente dito. Expressões lógicas e aritméticas são exemplos de construções possíveis com **Expr** e suas subclasses.

- **classe Stmt**: Responsável por nós de comandos. Os comandos tem a ver principalmente com o fluxo de execução do código, como *loops* **while**, comandos de decisão, como **if** e **else**, e interruptores de fluxo do tipo **break**. Apesar disso, a operação de atribuição é uma variação de **Stmt**, tendo sido implementada nas classes **Set** e **SetElem**.

Chamados durante a execução do analisador sintático(**parser**), são esses nós que representarão a árvore sintática, apresentada na subseção anterior, a partir da qual é gerado o código intermediário.

3 Código e Utilização

3.1 Obtendo o código fonte

Por ser muito extenso, preferimos não descrever todo o código do compilador neste documento, disponibilizando-o no repositório mencionado na seção de introdução.

Para obter o código, basta clonar o repositório utilizando o comando:

```
git clone https://github.com/joaofbsm/smallL.git
```

ou baixar o `.zip` disponibilizado ao clicar em **"Clone or download"** e depois em **"Download ZIP"** (na página do repositório).

Caso tenha optado pela segunda opção, basta descompactar e entrar na pasta descompactada.

3.2 Compilando e executando

Para facilitar a utilização do *front end* foram criados dois scripts *bash* que condensam as tarefas de compilar o código e executar testes em apenas duas chamadas na linha de comando.

- `compile.sh`: responsável por compilar as classes Java necessárias para o funcionamento do *front end*.
- `execute.sh`: reponsável por testar todas as entradas de teste disponibilizadas no diretório `tests`.

Para criar um caso de teste, basta adicionar um arquivo `.txt`, contendo o teste desejado, no diretório `tests` presente no diretório raiz do *front end*

Para rodar os scripts, siga os seguintes passos:

```
// mude para o diretorio raiz do front end
cd /caminho_para_diretorio_raiz/smallL
// compila
./compile.sh
// executa testes
./execute.sh
```

A saída dos testes é a padrão, ou seja, será escrita no terminal para cada um dos testes presentes no diretório.

4 Testes

Vários testes foram implementados para que a geração de código de desvio para expressões booleanas, realizada pelo compilador, fosse avaliada. Os testes a seguir mostram a sensitividade do *front end* em relação ao seguintes erros:

1. Erros de sintaxe.
2. Erros de tipo.
3. Erros de declarações inexistentes.
4. Erros de uso inadequado de tipos em condições (if, do, while).

Antes de elucidar a capacidade de retornar erros quando necessário, mostraremos o funcionamento padrão e correto para um programa fonte de acordo com a linguagem **SmallL**:

Algoritmo 2: Teste com entrada sem erros.

```
{
    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;
    }
}
```

A saída é retornada sem erros, gerando as quadruplas identificadas pelo *front end*:

```
Compiling test1.txt
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:
```

Figura 6: Saída para entrada sem erros.

.

Nesta saída é possível identificarmos dois dos principais processos de geração no código intermediário: geração de código para expressões e para comandos. Na linha 1 temos o código gerado para expressão $i = i + 1$, composta por uma expressão aritmética de adição seguida de um comando de atribuição. Na linha 9 temos o código gerado para o comando `if( i >= j ) break`. A geração de código para declarações não é explícita pois elas resultam em entradas na tabela de símbolos para identificadores.

Nas seções a seguir mostraremos os testes para identificações de erros.

4.1 Erro de sintaxe

Algoritmo 3: Teste de erro de sintaxe.

```
{
    int a; char b;
    while {
        int c;
    }
}
```

O teste acima apresenta um erro de sintaxe após o comando **while** (não é apresentado sua condição). Tal situação configura um erro, segundo a

gramática da linguagem. Logo, o *front end* retorna erro, como pode ser observado abaixo:

```
Compiling test2_syntax_error.txt
Exception in thread "main" java.lang.Error: near line 3: syntax error
        at code.parser.Parser.error(Parser.java:18)
        at code.parser.Parser.match(Parser.java:22)
        at code.parser.Parser.stmt(Parser.java:81)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.block(Parser.java:33)
        at code.parser.Parser.program(Parser.java:26)
        at code.main.Main.main(Main.java:11)
```

Figura 7: Saída para entrada com erros de sintaxe.

.

4.2 Erro de tipo

Algoritmo 4: Teste de erro de tipo.

```
{
  int a; char b; float c;
  a = 1;
  b = 2;
  if (a == b) {
    c = 0;
  }
}
```

Nesse teste são declaradas duas variáveis de tipos diferentes (int a e char b) e a seguir ambas são utilizadas na comparação do **if**. Tal situação não é válida segundo a linguagem **SmallL**. Portanto, o *front end* retorna o erro abaixo:

```
Compiling test3_type_error.txt
Exception in thread "main" java.lang.Error: near line 5: type error
        at code.inter.Node.error(Node.java:10)
        at code.inter.Logical.<init>(Logical.java:13)
        at code.inter.Rel.<init>(Rel.java:7)
        at code.parser.Parser.equality(Parser.java:141)
        at code.parser.Parser.join(Parser.java:131)
        at code.parser.Parser.bool(Parser.java:123)
        at code.parser.Parser.stmt(Parser.java:72)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.block(Parser.java:33)
        at code.parser.Parser.program(Parser.java:26)
        at code.main.Main.main(Main.java:11)
```

Figura 8: Saída para entrada com erros de tipo.

.

4.3 Erro de declarações inexistentes

Algoritmo 5: Teste de declarações inexistentes.

```
{
  float a;
  a = 4;
  if (b == 2){
    a = 5;
  }
}
```

Nesse teste usa-se uma variável (b) em uma condição do **if** que não foi declarada previamente. Tal situação representa um erro e o *front end* identifica isso de forma correta, como mostra a saída abaixo:

```
Compiling test4_undeclared_id_ref.txt
Exception in thread "main" java.lang.Error: near line 4: b undeclared
        at code.parser.Parser.error(Parser.java:18)
        at code.parser.Parser.factor(Parser.java:210)
        at code.parser.Parser.unary(Parser.java:179)
        at code.parser.Parser.term(Parser.java:165)
        at code.parser.Parser.expr(Parser.java:157)
        at code.parser.Parser.rel(Parser.java:147)
        at code.parser.Parser.equality(Parser.java:139)
        at code.parser.Parser.join(Parser.java:131)
        at code.parser.Parser.bool(Parser.java:123)
        at code.parser.Parser.stmt(Parser.java:72)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.block(Parser.java:33)
        at code.parser.Parser.program(Parser.java:26)
        at code.main.Main.main(Main.java:11)
```

Figura 9: Saída para entrada com erro de declarações inexistentes.

.

4.4 Erro de uso inadequado de tipos em condições

Algoritmo 6: Teste de uso inadequado de tipos em condições.

```
{
    int a;
    int c;
    if (a){
        c = 2;
    }
}
```

Nesse teste é usado uma variável do tipo **int** (a) na condição do **if**, o que não é esperado de acordo com a gramática da linguagem. Logo, o *front end* retorna um erro, como mostra a saída abaixo:

```
Compiling test5_non_boolean_in_cond.txt
Exception in thread "main" java.lang.Error: near line 3: boolean required in if
        at code.inter.Node.error(Node.java:10)
        at code.inter.If.<init>(If.java:10)
        at code.parser.Parser.stmt(Parser.java:74)
        at code.parser.Parser.stmts(Parser.java:61)
        at code.parser.Parser.block(Parser.java:33)
        at code.parser.Parser.program(Parser.java:26)
        at code.main.Main.main(Main.java:11)
```

Figura 10: Saída para entrada com erro de uso inadequado de tipos.

.

5 Conclusão

Com a finalização desse trabalho finalmente temos o *front end* do compilador pronto. Os analisadores funcionaram corretamente, e a geração de código intermediário se deu com sucesso.

Agora só no resta a implementação do *back end* do compilador, para que assim possamos executar de fato os códigos escritos na linguagem **smallL**. Em trabalhos futuros iremos implementar um tradutor do código intermediária aqui gerado para código de máquina, mais especificamente o código da máquina virtual **TAM**.

Referências

[1] A. V. Aho, *Compilers: principles, techniques, and tools*, 2007, vol. 2.