Universidade Federal do Ceará
Disciplina de Construção de Compiladores

Relatório de Implementação

Bruno Anderson
Carlos Átila
Leonardo Cardoso
Pedro Crispim

Junho - 2011

Módulos implementados

CAP 2. Análise Léxica	Concluido
CAP 3. Análise Sintática	Concluido
CAP 4. Árvore abstrata sintática	Concluido
CAP 5. Tabela de símbolos, Verificação de tipos	Concluido
CAP 6. Registros de ativação	Concluido
CAP 7. Representação Intermediária	Concluido
CAP 8. Blocos básicos e traços	Concluido
CAP 9. Seleção de instruções	Concluido
CAP 10. Análise de fluxo de dados	Iniciado, não concluido
CAP 11. Alocação de registradores	Iniciado, não concluido

Divisão do trabalho

Bruno Anderson: Cap 2, 3, 4, 5, 6 , 7, relatório

Carlos Átila: Cap 2, 3, 4, 5, 6 , 7, relatório

Leonardo Cardoso: Cap 2, 3, 4, 5, 6 , 7, relatório

Pedro Crispim: Cap 2, 3, 4, 5, 6 , 7, 8, 9, 10, 11 relatório, testes

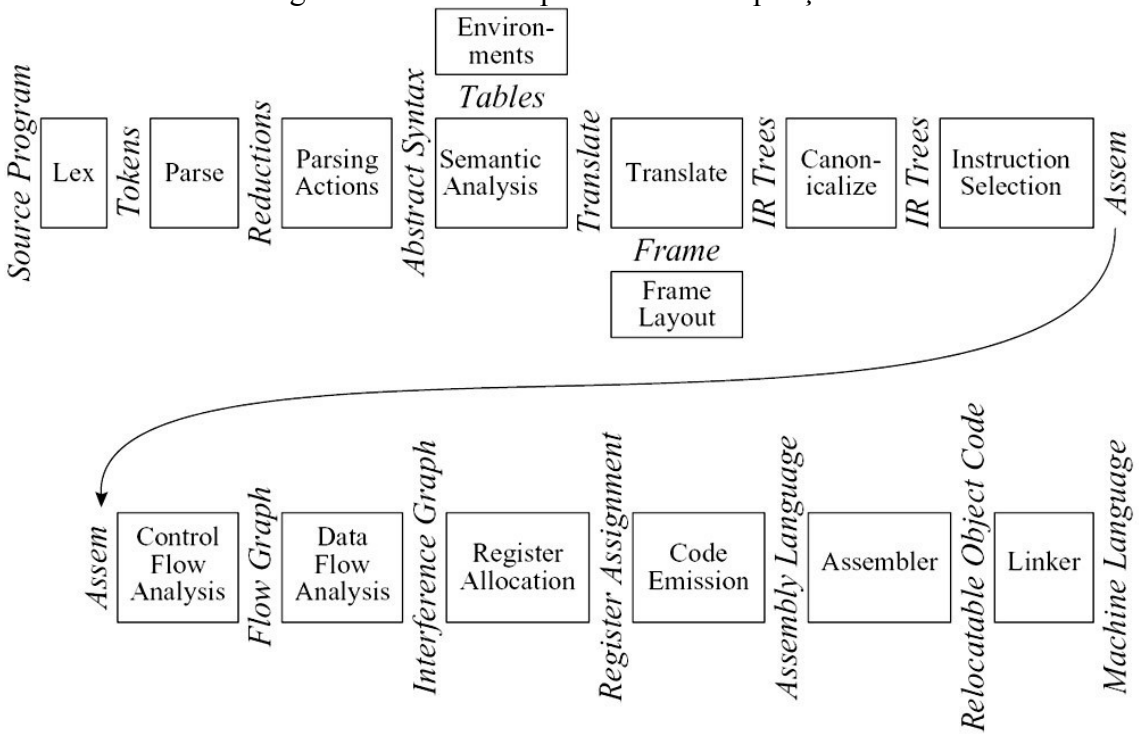
Introdução

O objetivo deste projeto visa a implementação de um compilador em linguagem Java, para ser mais preciso com o uso da linguagem Mini-Java, que é um subconjunto da linguagem Java, apresentada pelo livro Modern Compiler Implementation in Java adotado como livro-base da disciplina.

A linguagem Mini-Java nos permite construir um compilador, conforme sua referência MiniJava?

O trabalho está sendo feito com o JavaCC que é um gerador de analisador sintático que produz código Java. Ele permite que uma determinada linguagem seja definida de maneira simples, por meio de uma notação semelhante à EBNF. Como saída produz o código-fonte de algumas classes Java que implementam os analisadores léxico e sintático para aquela linguagem. Provê também maneiras de incluir, junto à definição da linguagem, código Java para, por exemplo, construir-se a árvore de derivação do programa analisado.

Abaixo um diagrama das fases do processo de compilação:



Iremos detalhar a seguir cada um destes processos:

Análise Léxica

A principal função do Analisador Léxico é de fragmentar o programa fonte em componentes básicos, chamados de átomos ou tokens. Do ponto de vista de implementação do compilador, o analisador léxico atua como uma interface entre o analisador sintático e o texto de entrada, convertendo a seqüência de caracteres que formam o programa fonte em uma seqüência de tokens que o analisador sintático consumirá.

O analisador léxico executa usualmente uma série de funções, que são de grande importância para a correta operação das outras partes do compilador. As funções mais importantes do analisador léxico são:

- Extração e Classificação dos tokens: extração e classificação dos tokens que compõem o texto do programa fonte. Entre a classe de tokens mais encontradas em analisadores léxicos estão: identificadores, palavras reservadas, números inteiros, cadeias de caracteres, caracteres especiais simples e compostos.
- Eliminação de Delimitadores e Comentários: espaços em branco, símbolos separadores e comentários são irrelevantes na geração de código podendo ser eliminados pelo analisador léxico.
- Recuperação de Erros: detecção de cadeias de caracteres que não obedecem a nenhuma lei de formação das classes de tokens que o analisador léxico reconhece.

Resumindo esta análise, o Analisador Léxico recebe um programa fonte como entrada e o analisa formando gerando uma seqüência de símbolos chamados “tokens” que podem ser facilmente manipulados por um parser (leitor de saída). Ele lê caracter por caracter do texto fonte verificando se estes caracteres idos pertencem ao alfabeto da linguagem, desprezando comentários e espaços vazios desnecessários.

Os tokens constituem classes de símbolos tais como palavras reservadas, identificadores, ... e podem ser representados, internamente, através do próprios símbolo ou por um par ordenado, no qual o primeiro elemento indica a classe do símbolo e o segundo um índice (uma entrada) numa tabela de identificadores.

Além da identificação de tokens, o Analisador Léxico, em geral, inicia a construção da Tabela de Símbolos. A saída do Analisador Léxico é uma cadeira de tokens que é passada para a Análise Sintática e no nosso caso, o Analisador Léxico é implementado como uma subrotina que funciona sob o comando do Analisador Sintático.

Nesta etapa, toda nossa programação concentrou-se no arquivo MiniJava?.jj onde foi declarado a Gramática e as produções correspondentes a Gramática. Através do JavaCC foram criados os tokens para uso posterior do Analisador Sintático, vide MiniJavaParserConstants?.java.

Análise Sintática

É a fase da compilação responsável por determinar se uma dada cadeia de entrada pertence ou não à linguagem definida por uma gramática, gramática esta definida no arquivo MiniJava?.jj. A análise sintática, ou parsing, é o núcleo principal do front-end do compilador. O parser é o responsável pelo controle de fluxo do compilador através do código fonte em compilação. É ele quem ativa o Analisador Léxico para que este lhe retorne os tokens, citados acima, e também ativa a Análise Semântica e a geração de código. O resultado de seu processamento produz uma estrutura conhecidamente denominada AST – Abstract Syntax Tree. Esta árvore não precisa necessariamente ser gerada "fisicamente".

Nesta etapa de desenvolvimento, com o uso do JavaCC, temos gerado todos os Scanner e Parser de nosso compilador, tendo sido gerados os arquivos abaixo:

MiniJavaParser?.java
MiniJavaParserConstants?.java
MiniJavaParserTokenManager?.java
ParserException?.java
SimpleCharStream?.java
Token.java
TokenMgrError?.java
MiniJava?.jj

Nós modificamos as derivações da gramática disponibilizada (presente no livro texto da disciplina), para que esta não tivesse ambigüidades e fosse utilizado lookahead 1 no parser preditivo. Essa foi a maior dificuldade que nós encontramos até o momento nesta etapa. A gramática disponibilizada necessitaria de lookahead maior que 1. Os tokens gerados foram baseados nas regras dessa gramática.

Análise Semântica

As Análises Léxica e Sintática não estão preocupadas com o significado ou semântica dos programas que elas processam. O papel do Analisador Semântico é prover métodos pelos quais as estruturas construídas pelo Analisador Sintático possam ser avaliadas ou executadas. As gramáticas livres de contexto não são suficientemente poderosas para descrever uma série de construções das linguagens de programação, como por exemplo regras de escopo, regras de visibilidade e consistência de tipos.

Através da nossa gramática, para cada derivação, nós temos ações semânticas que nos ajudam a construir a Árvore Sintática Abstrata. Ela representa as regras da gramática. A motivação de utilizarmos essa árvore é facilitar o processo de construção da tabela de símbolos e da checagem de tipos, que fazem parte da Análise Semântica.

Ao percorrer a árvore sintática do programa e verificar se os tipos das expressões estão condizentes com as informações armazenadas na tabela de símbolos, realizamos a "checagem de tipos". Visando a modularidade do programa, que é um dos cuidados especiais na implementação do nosso compilador, implementamos tanto a construção da tabela de símbolos quanto a checagem de tipos utilizando o padrão Visitor.

O padrão Visitor foi utilizado por ser um padrão comportamental em que podemos representar que operação vai ser executada sobre elementos de uma estrutura, no nosso caso a árvore. Assim, podemos definir novas operações sem a necessidade de modificar a classe dos elementos em que opera. As operações necessitam tratar cada tipo da árvore de maneira diferente.

Nesta etapa, a principal dificuldade foi encontrar uma estrutura de dados que facilitasse a manipulação da tabela de símbolos.

Tradução para representação intermediária

Após transformar o código fonte em uma árvore sintática abstrata e executar ações semânticas para cada nó, o compilador gera uma nova árvore que representa instruções abstratas simples de código de máquina, essa árvore é construida utilizando a classe IRVisitor, que da mesma forma que os outros visitors implementados, percorre a AST executando ações correspondentes a cada nó.

A IR tree será utilizada posteriormente em conjunto com as classes Frame e Temp para gerar o código alvo para um processador específico, porém nessa fase ainda não estamos preocupados com detalhes de processador, mas sim em um conjunto de instruções primitivas que possam ser compatíveis com qualquer arquitetura. As classes Frame e Temp também abstraem dos detalhes do frame da arquitetura e da quantidade de registradores da arquitetura alvo.

Blocos básicos e traços

O resultado da fase de tradução para a representação intermediária são vários fragmentos de códigos associados a cada frame da pilha. Cada fragmento tem uma árvore que corresponde a uma expressão, porém na linguagem alvo não serão avaliadas expressões, alguns aspectos da linguagem IR não correspondem exatamente a realidade das linguagens de máquina.

Nesta fase, o compilador irá reescrever a árvore para que ela possa ser transformada em uma lista de instruções. A classe Canon utiliza o método linearize para esta tarefa, o método recebe como entrada as árvores associadas a cada fragmento e a retorna no formato apropriado para a tradução em código de máquina. A classe BasicBlocks transforma essa lista de instruções em um conjunto de blocos, onde cada bloco é começado por um label e pode terminar com um salto.

Seleção de instruções

Nesta etapa foram utilizadas as classes do pacote Assem do framework, que correspondem a instruções mais próximas do nível de máquina, em conjunto com a classe Codegen, que é especfíca de uma arquitetura, que é responsável por percorrer a árvore na linguagem IR e reconhecer padrões (ladrilhos) na árvore para selecionar as instruções relativas a sua arquitetura.

Para testar a emissão de código assembly, foi utilizada a arquitetura proposta no livro, a Jouette Assembly, que contem a maioria das instruções utilizadas em processadores habituais como ADD, MOVE, STORE, etc. O algoritmo utilizado para o reconhecimento de padrões na árvore foi o Maximal Munch, implementado na classe Codegen, que busca por uma solução otimal, ou seja, seleciona o maior ladrilho possível a partir do nó atual.

Conclusão

Esse trabalho prático nos ajudou a compreender as tarefas essenciais de um compilador clássico, outras diversas funcionalidades que podem otimizar a compilação do programa e como implementar um compilador mais genérico que se adapte mais facilmente as diversas arquiteturas. Acreditamos que ter implementado este compilador utilizando das abstrações providas pelo framework nos deu uma visão mais ampla de técnicas para projetar um compilador robusto que possa agregar diversas funcionalidades. Apesar das vantagens de se trabalhar em alto nível, sem se preocupar com detalhes da programação do parser por exemplo, tivemos dificuldade de compreender como as partes se interligavam na fase correspondente a implementação do backend, pois em algumas partes o livro peca na clareza, o que dificulta o entendimento do código exibido no livro. Porém o aprendizado adquirido nos módulos implementados nos serão muito úteis como base para futuras implmentações de um compilador.